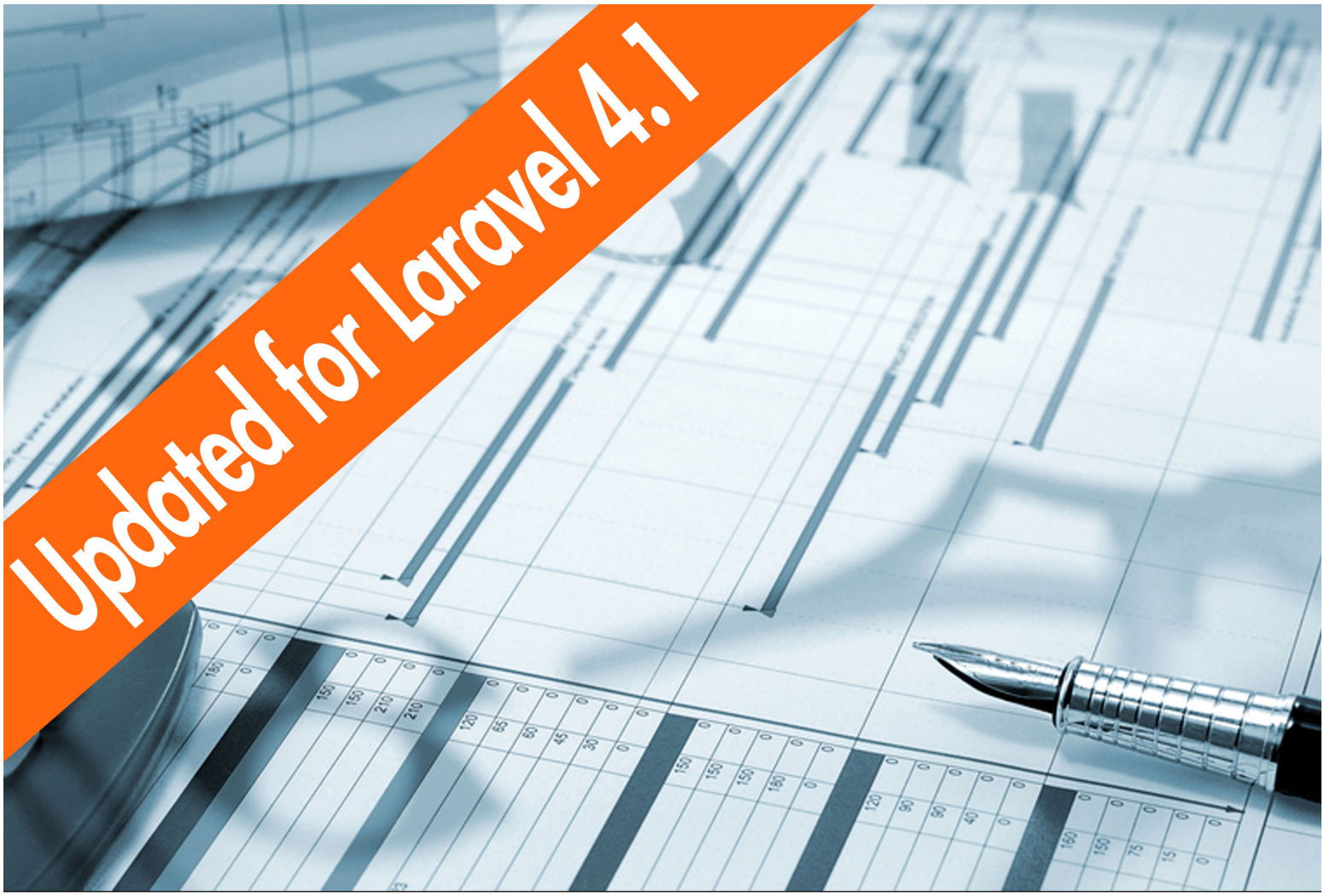


Getting Stuff Done *with Laravel 4*

A journey through application design and development using PHP's hottest new framework

by Chuck Heintzelman

Getting Stuff Done with Laravel 4

A journey through application design and development using PHP's hottest new framework

Chuck Heintzelman

This book is for sale at http://leanpub.com/gettingstuffdonelaravel

This version was published on 2014-02-02

Tweet This Book!

Please help Chuck Heintzelman by spreading the word about this book on Twitter!

The suggested hashtag for this book is #LaravelGSD.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#LaravelGSD

Contents

Thank You

I want to sincerely thank you for purchasing this book. I hope you find it engaging, entertaining, and most of all, useful.

Other Places to Learn Laravel

- The website[1]. This is always my first stop to look something up. Check out the forums there. It's chock-full of information.
- NetTuts[2]. There's some nice Laravel tutorials on the site.
- Laravel Testing Decoded[3] by Jeffery Way. This book is an awesome resource on how to test your Laravel code.
- Code Bright[4] by Dayle Rees. This book is both fun and informative.
- From Apprentice to Artisan[5] by Taylor Otwell. By the creator of Laravel … need I say more.
- Implementing Laravel[6] by Chris Fidao. This book focuses on implementing projects with Laravel. It covers structure and and common patterns. A great book.
- Laravel 4 Cookbook[7] by Christopher Pitt. This book contains various projects built in Laravel 4.
- Laravel in Action[8] by Maks Surguy. This book is now available from Manning Publications's Early Access program.

---

[1] http://laravel.com
[2] http://net.tutsplus.com/
[3] http://leanpub.com/laravel-testing-decoded
[4] http://leanpub.com/codebright
[5] https://leanpub.com/laravel
[6] https://leanpub.com/implementinglaravel
[7] https://leanpub.com/laravel4cookbook
[8] http://www.manning.com/surguy/

Revision History

Current version: 1.2

Version	Date	Notes
1.2	02-Feb-2014	Typos and updates for Laravel 4.1
1.1	28-Nov-2013	Typos and general cleanup.
1.0	2-Nov-2013	Fixed many typos and released on Leanpub.
0.9	27-Oct-2013	Finished and released on Leanpub.
0.8	20-Oct-2013	Added 4 chapters, released on Leanpub.
0.7	13-Oct-2013	Added 3 chapters, two appendices. Released on Leanpub.
0.6	6-Oct-2013	Added 8 chapters, finishing Part 3. Released on Leanpub.
0.5	29-Sep-2013	Added 7 chapters to Part 3 and released on Leanpub.
0.4	22-Sep-2013	Added 7 chapters to Part 3 and released on Leanpub.
0.3	15-Sep-2013	Cleanup draft through Part 2. Decided to release first version on Leanpub.
0.2	8-Sep-2013	Finish 1st draft of Part 2
0.1	31-Aug-2013	Finish 1st draft of Welcome and Part 1
0.0	3-Aug-2013	Start writing first draft

A special thank you

Here's a list of non-anonymous people who have helped me by finding typos and other issues in these pages.

- Peter Steenbergen
- Jeremy Vaught
- George Gombay
- Mike Bullock
- Kristian Edlund

Thank you very much!

Cover Image

[9] http://www.dreamstime.com/kemaltaner_info
[10] http://www.dreamstime.com/

Help Wanted

I know there are some errors within these pages. Unfortunately, I don't know where they are. With modern self-publishing the time from creation to production can be a matter of minutes.

As such, there's no middle-man. I write something, check for errors, hit a few keys and WAM! you've got the latest version.

But I'm not perfect. *(Even if I do try to convince my wife that I am. Shhh. Don't tell her.)*

Please help me make this book better. If you find misspellings, confusing words, whatever, please contact me at chuckh@gmail.com[11] and let me know. I appreciate the help and will give you credit on the revisions page. *(If you mention page numbers, please also provide your version of the book from the Revision History page.)*

Here's a list of things I could use help with:

- Words: Misspelled words, missing words, and incorrectly used words.
- Instructions for systems other than Ubuntu.
- Incorrect facts. (Sometimes I just make 'em up as I go along.)
- Grammar … not really. I'd hafta rewrite the whole book to make it grammatically correct. Know what I'm sayin?

Translations

After the English (or should I say, the American) version of this book has settled down and the kinks have been worked out, I will be translating it into other languages. It makes me sad that many of the idioms will have to be killed–oh, well.

If you're interested in translating this work into a different language, please shoot me an email.

[11] mailto:chuckh@gmail.com

Source Code on GitHub

The source code for this book is available on GitHub in my getting-stuff-done-laravel-code[12] repository.

Since each chapter of this book that has code that builds on a previous chapter, I set up a branch for each chapter.

For example, the first chapter with code is *Chapter 19 - Installing Laravel*. It's available in the **chapter19** branch.

It's pretty easy to pull down from github just the chapter you want. Let's say you wanted chapter32. In linux you'd do something like:

```
$ git clone -b chapter32 \
> https://github.com/ChuckHeintzelman/getting-stuff-done-laravel-code.git
Cloning into 'getting-stuff-done-laravel-code'...
remote: Counting objects: 635, done.
remote: Compressing objects: 100% (273/273), done.
remote: Total 635 (delta 306), reused 622 (delta 295)
Receiving objects: 100% (635/635), 100.08 KiB, done.
Resolving deltas: 100% (306/306), done.
```

[12] https://github.com/ChuckHeintzelman/getting-stuff-done-laravel-code

Welcome

Welcome to *Getting Stuff Done with Laravel*. The **Welcome** part of the book explains what you'll get out of the book.

Here's how things are broadly organized.

Welcome
: The first part of the book explains what you'll get out of the book.

Part 1 - Design Philosophies and Principles
: This part talks about general principles of design we'll follow in creating the application.

Part 2 - Designing the Application
: This is the part where we design the actual application.

Part 3 - The Console Application
: Next, we make the application usable from the console.

Part 4 - The Web Application
: Now we'll take our creation and put a web skin on it. **Mwaa**, **haa**, **ha**.

Appendices
: Supplemental information. Like how to install Composer.

Chapter 1 - This book's purpose

This book will take you on a journey through Laravel. Hopefully, you'll go places you've never been and see you things you've never seen. It's a travelogue of sorts. We'll have a definite destination (the application we're creating) and I'll point out some amazing sights along the way. When you reach the end, drop me a note at chuckh@gmail.com[13]. I'm very interested in what you thought of the journey.

This book is meant to be experienced. To be used. Please follow along and build the application chapter by chapter. Each chapter leads to the next. The sections within a chapter flow forward. Each part of the book builds on the previous.

You could think of the sections in each chapter as cities. Then the chapters themselves are countries, and the book's parts are continents and … *Okay, enough with the labored traveling analogy.*

The focus throughout the book is the step-by-step creation of an application using Laravel 4.

This is not a typical technical manual

I've attempted to mimic the actual process of design and development as closely as possible. This means there are false starts, design changes, and refactoring along the way.

You've been warned <grin>.

What's not in this book

- Every aspect of Laravel. This is not a reference book on the entire framework.
- Caching, Events, or Logging. These are important topics, but the application we're creating doesn't require them.
- Queues, Authentication, Cookies, or Sessions. Again, important stuff, but we don't need it.
- Database. Yeah, it almost pains me to admit this. One of the greatest aspects of Laravel is it's *Fluent Query Builder* and *Eloquent ORM*. I mean, what great names. Names that the implementation fully lives up to. Sadly, I don't touch on this because … you guessed it … the application we're creating doesn't need it.

[13] mailto:chuckh@gmail.com

What's in this book

Mostly, me blabbing away about why I'm doing what I'm doing in creating the application. You may agree with me some of the time. You may argue with me some of the time. Sometimes you may think I'm a complete idiot. Hopefully, at times you'll think *"Oh yeah. Good one."* But in the end, you're getting the nuts-and-bolts of creating a real system that you can use.

Chapter 2 - Who are you?

Most books start with information about the author but the more important question really is "who are you?"

I'm making the following assumptions:

- You know more about computers than most people.
- You are a programmer.
- You know how to program in PHP. Maybe a little. Maybe a lot.
- You've heard of Laravel[14]. *(This isn't a deal-breaker because I'm going to tell you about it.)*
- You love programming or want to get back the passion that first drove you to make computers do your bidding.
- Your name is not Taylor Otwell because if it is, then I'm not worthy.

I'll do my best to make the material approachable to beginners, yet interesting and in-depth enough for intermediate programmers to find the material useful.

Bottom line

You want to learn more about Laravel.

[14] http://laravel.com

Chapter 3 - Who am I?

This is the typical rah-rah, ain't I great chapter. It's not going to teach you a single thing about Laravel. The smartest move you could make right now is to skip to the next chapter.

Hello. My name is Chuck Heintzelman and I write computer programs.

(That felt like I was in front a support group. I hope nobody said "Hi Chuck.")

Seriously. I've written programs since that day in 9th grade when I stayed home "sick" from school with a borrowed *BASIC Language Reference* manual and wrote out on paper a game that was like Asteroids[15] except instead of asteroids flying at you it was other ships firing long white blocks of death at you.

After long hours of debugging and waiting for the TRS-80 to load/save my program to it's "mass storage" (a cassette tape), the game finally worked. This was 33 years ago. Back in the day of computer dinosaurs, large ferocious beasts filling climate-controlled rooms. No, I've never *actually* used punched cards, but have seen them in use.

Since then I've written programs in Fortran, COBOL (yeah, I know), Assembly Language, Basic, C, C++, C#, Java, Pascal, Perl, Javascript, and PHP. I've tinkered with many, many other languages, but have not written programs that people actually used.

I've created systems for Fortune 500 companies, as well as small Mom-and-Pop stores. Everything from mail order systems running in Xenix to web applications running in PHP. I've started several companies before the days of the Internet *(not before the **real** beginning of the Internet, just before the excitement starting in the mid-90s)*, and a few dot coms since then. And through it all I've did what I loved to do–write computer programs.

Whew! Okay, enough about how great I am.

Here's my point

Throughout my career I've never felt the need to create a book about programming until now. The sole reason I'm writing this is because of Laravel.

[15] http://en.wikipedia.org/wiki/Asteroids_(video_game)

Chapter 4 - What is Laravel?

Raise your hand if this sounds familiar

> *You've been tasked with adding a feature to your company's existing system. Unfortunately, the system was written in PHP 4 and whoever the original programmer was, you suspect they watched a few too many "Wordpress Gone Wild" videos.*
>
> *You've inherited this codebase with *gasp* no classes, a glut of global variables, and a structure not unlike a 50,000 piece jigsaw puzzle.*
>
> *You curse your job, the short sightedness of the management-team, and whatever possessed you to want to make money by programming in the first place.*
>
> *After all, programming should be fun. Right?*

We've all been there.

Enter Laravel.

(Cue the sound of kettle drums: duh-duh duh-duh duh-da-duh)

Laravel is a framework for PHP which makes programming fun again.

Come on man ... it's just a framework

Laravel is not a new language. It's only a framework. If you cut through all the hyperbole and look at its essence, Laravel is simply a PHP Framework.

Although, I do agree with the motto from the Laravel web site:

The PHP Framework for Web Artisans.

Ruby On Rails is *just a framework.* Yet look at the fandom behind it.

Laravel's not going to magically fix your PHP spaghetti code, but it provides you with a new, fast and elegant way to get stuff done. *(Note, the concept of Getting Stuff Done is a reoccurring theme in this book.)*

In short, Laravel provides you with an architecture that makes PHP programming a joy. You'll be able to refactor existing code in a way that is expressive, stylish, and will be easy to maintain and expand in the future.

Laravel is not a panacea. If your existing codebase sucks, it's going to be painful to get from **where it is now** to **where it should be**. That's the nature of our industry.

But, if you want to move to a framework that allows simple expressivity *(is that even a word?)* then Laravel is the answer.

Chapter 5 - How to justify Laravel

Here's the problem (or a problem) ...

> *You must work under the constraints your company places on you. Namely, that you must support the existing software and develop new code that plays nice with your existing systems. There's a mix of .NET, some Java, but most of the existing code is PHP.*

You've recently discovered Laravel and like it and want to use it for new development.

How can you justify switching to Laravel?

Let's put on our detective hat for a minute.

Hmmm. The detectives I know (from TV of course) seem to follow the money when looking for suspects and motives. So let's follow the money ...

Customers provide money to businesses in exchange for goods and services. The better the product, and the more customers really want the product, the more money they fork over to the business.

Managers want the business to thrive. They want as many customers as possible to give them as much money as possible as frequently as possible.

Think about it from management's perspective ...

- I want my customers to be happy.
- I want new customers.
- Customer happiness equates to expectations being met.
- I want my programmers to be able to deliver the requirements on time.
- I want the programming team to be agile. *(Whatever that means ... see the box below.)*
- I want to facilitate customer's requests in a timely manner.
- I want great developers delivering great products

What does Agile even mean?

You ever say or write a word so often that it loses all meaning? Almost like the Smurfs ... everything is smurfing, smurfable, smurferific. Agile seems to be one of those words. It's way

> past the buzzword stage. Everything is Agile this, Agile that. Are people talking about the iterative software process or something else? Something magical? I really don't know.

If the above list is the management perspective, then Laravel is easily justified:

- Customers are happy when their needs are addressed and met.
- Customers are even happier when their expectations are exceeded.
- Laravel provides a framework that …
  - Makes it easy to extend functionality.
  - Follows best-practices in design.
  - Allows multiple programmers to collaborate efficiently.
  - Makes programmers happy. *(Remember managers: a happy programmer is a productive programmer.)*
  - Let's stuff get done faster.
  - Encourages unit testing, considering testing a core component of every application.

Laravel provides managers the ability for their programmers to get more done, quicker, and eliminates many of the obstacles inherent in web application development. I'll expand on this later.

Pretty easy to justify, ain't it?

Chapter 6 - Why programmers like Laravel

Let's cut to the chase … why would you, as a programmer, want to use Laravel as a framework?

Let me talk a bit about Framework Envy.

> *(Here I picture talking to a therapist. Him nodding sagely, taking a drag on his pipe and saying, "Talk about zee framework envy.")*

I'd been given projects written in PHP. These were bloated, PHP 4 projects written by a developer whose only concept of "**class**" was that it is something at school to be skipped. And I'd look across the street at the Ruby developers and silently wish for some natural disaster–earthquake, tornado, even lightning–to level their building.

Does this make me a bad person?

This was at a time when Ruby was all shiny and new. What made Ruby cool wasn't the language itself (although there are very nice aspects to the language). No, what made Ruby cool was Ruby on Rails.

All the developers were flocking to Ruby on Rails.

Why were they flocking to it?

Because it promised a way of development that was fun. And by fun, I mean powerful, expressive, and quick to implement. I credit RoR on creating an atmosphere making programming a delight again. The coding joy instilled by RoR is the exact same feeling as that initial impetus that made us all want to be programmers.

How sad was it that we were mired in the PHP world? Where any Tom, Dick or Henrietta was a "PHP Programmer" because they could hack a Wordpress install.

(See the next chapter about Wordpress - The Good, The Bad, The Ugly)

But, no, we were stuck with the requirements that our projects be in PHP. We couldn't be a cool kid like all those Ruby developers. They were cutting edge. They were the ones pushing the boundaries, making a name for themselves.

Along comes Laravel. It takes the best of Ruby on Rails and brings it to the PHP world. Suddenly, a PHP developer is dealing with routes to controllers instead of individual scripts. Concepts like DRY (Don't Repeat Yourself) now have more meaning. Suddenly, we have a "blade" template engine incorporating the essence of PHP in a way Smarty Templates only dreamed of. We have, quite literally, the potential of PHP Nirvana.

Does it sound like I think Laravel's awesome? I hope so.

Chapter 7 - Wordpress: The Good, The Bad, The Ugly

Wordpress revolutionized blogging. It brought blogging to the masses. Sure, there are other platforms like blogger and livejournal, but what Wordpress did was put out in the public domain a large, popular system written PHP.

With the advent of Wordpress, anybody could hack the PHP scripts to make the blogging platform do what they wanted it do it.

> "With great power comes great responsibility." – Uncle Ben (from Spiderman)

Unfortunately, the power Wordpress availed was not met with great responsibility. Scripts were hacked with no thought toward overall design or usability. To make matters worse, Wordpress started in the days of PHP 4, when the language didn't allow the constructs that allowed true programmers to create maintainable systems.

Wordpress was the best thing that happened to PHP, but it also was the worst thing that happened to the language.

It's a case of too much success in the hands of too few artisans.

This attached a stigma to PHP.

Softwarati[16]
: Self absorbed programming intellectuals who comment on languages.

For your consideration … a commonly heard quote by the Softwarati:

> "Oh. PHP's a ghetto language. Ugly, hardly maintainable, but it works … most of the time"

Thank goodness Laravel came along to kick those Softwarati in their upturned noses.

[16] Yes, this is a word I totally made up.

Chapter 8 - Conventions Used in this Book

There are several conventions used through this book.

Code is indented 2 spaces

Usually, I indent code 4 spaces but since this book is available in a variety of eBook formats some of the smaller screens have horizontal space at a premium.

```
1  for ($i = 0; $i < 10; $i++)
2  {
3    echo "I can count to ", $i, "\n";
4  }
```

This is a tip

It is used to highlight a particularly useful piece of information.

This is a warning

It is used to warn you about something to be careful of.

This is an information block

Used to reiterate an important piece of information

This is something to do

When there's code, or other actions you should take, it's always proceeded by this symbol.

Trailing ?> is used when opening tag is used.

When coding, I always drop the trailing ?> in a file. But the editor I'm writing this book in makes everything look wonky when I do it. So, within this book, if I open a PHP block with the PHP tag, I always close it in the code too. For example:

```
<?php
class SomethingOrOther {
    private $dummy;
}
?>
```

PHP Opening and Closing Tags

In the code examples sometimes the opening PHP tag (`<?php`) is used when it's not needed (such as when showing a portion of a file.) Sometimes the closing PHP tag ('?>') is used when it's not needed.

```
<?php
  function somethingOrOther()
  {
    $this->callSetup();
  }
?>
```

With real PHP Code I *always* omit the closing tag at the end of the file. I'll leave it to you to determine whether or not the tags are needed. Be aware the opening and closing tags in the code examples should not be taken verbatim.

What OS am I using?

I'm writing this manual, the code, etc., using Linux Mint 16[17] which based on Debian and Ubuntu. It's basically the same as Ubuntu 13.10[18].

[17] http://www.linuxmint.com/
[18] http://www.ubuntu.com/

Part 1 - Design Philosophies and Principles

There's not much code in this part of the book. Sorry, it's all about the design at this point. Here I'll discuss general design principles used in building the application.

You may be thinking, "just take me to the code." For the most part, I concur. It's often quickest and easiest just to jump in the code and learn by doing. If you understand the concepts: SOLID Object Design, Interface as Contract, Dependency Injection, Decoupling, and Inversion of Control, then skip to Part 2 to begin designing the application.

Chapter 9 - Yee Haw! Cowboy Coding is Great.

In This Chapter

In this chapter I discuss Cowboy Programming (or Cowboy Coding) and how it's often used in a derogatory manner ... but it shouldn't be.

Here's my usual development workflow for creating new systems:

1. **What's it gonna do?** I try to figure out what the new system is going to do. This may be as simple as a single sentence. Or it may involve wire framing[19] a portion of user interface. More often than not, it's simply something in my head, not written down, and I want to jump in and code the dang thang.
2. **Start coding** I start a new project and begin coding. Here I just run straight at what I'm trying to accomplish.
3. **Regroup** Eventually I realize step 2 didn't work as well as I thought it would. Now, a bit of "refactoring" is in order. But ... since this is a new project I don't refactor, I recode. In other words, I set aside the existing code and start fresh, only using the existing code as a rough blueprint, and pulling in whole chunks (classes, functions, files) as I need.
4. **Get bored** When I'm 90% finished with the project, I get bored to tears with the whole mess.

What is Refactoring?

Refactoring is a process in which the internal structure of the code is changed without changing external results. Often it's a process of removing duplication, separating complex modules into simpler patterns, or renaming files, classes, or methods for better consistency.

This four-step system works quite well for me. (Except for the last step, where I just *"grunt it out."*) What's really going on is step #2 is that I'm designing the system. I'm designing it by coding it. So step #3 is the "real" coding phase.

[19] http://en.wikipedia.org/wiki/Website_wireframe

When coding using this method, experimentation is quick-and-easy, because you're not bogged down with a restrictive workflow.

This method of creating projects is often called "Cowboy Programming" or "Cowboy Coding" and it's looked down upon by … guess who? Management. It's not a process that management can easily control so they don't like it.

Is there anything wrong with this method? Absolutely not! Hotshot programmers can get more done in less time by Cowboy Programming, and that's what matters in the end, isn't it?

Don't get my wrong. Agile, RAD, Extreme Programming, even TDD (especially TDD) are all great methodologies. And I'm game for any of them, but only if I can keep as much autonomy over the process as possible.

Does Cowboy coding work with teams? Absolutely, as long as everyone stays in their own corral. (Ugh, I know. But couldn't resist the metaphor.)

I've been doing this for enough years that the patterns and structures I use are subconscious. I use what I need in the white-hot fire of coding, automatically following techniques that have worked countless times in the past.

Ah, so Cowboy Coding is good?

Probably not.

(Now you're thinking "Chuck. What the …? You just extolled the virtues of Cowboy Coding. Dude, you're sending mixed messages.")

Sorry about that. But the problem with Cowboy Coding isn't so much in what *I* call Cowboy Coding, as it is in not having an exact definition. Lack of a standard definition leads to every negative comment about programming to be lumped under the label "Cowboy Coding."

"He won't comment his code. He's a gosh-durn Cowboy Coder."

"She thinks her code is perfect and blames lack of decent equipment for failures. She's one of those dang Cowgirl Programmers."

"She's a total prima donna. Won't test her code cuz she said it never has a bug. Get a rope."

"He can't work with others. Thinks he's the lone ranger, that one does."

"Why the heck did he go off again and spend hours recoding an unimportant system that wasn't broke in the first place. I hate these Cowboys!"

You get the picture. I think Cowboy Coding is the best way to code, but I also believe:

- You must work well with others.
- You must not have ego about your code.
- You must be able to get things done.

- You must “own” your problems and fix them.
- You must realize your way isn’t always the best way.
- You must know sometimes *good enough* is better than *perfect*.

Chapter 10 - Every Programmer Is Different

In This Chapter

This chapter discusses how every programmer is different and what works best for me may be totally different than what works best for you.

Why am I devoting an entire chapter to one simple statement?

Because, I'm sick of everyone believing there's **one true way**.

How you develop code and how I develop code may be diametrically opposed. Maybe you can't write a line of code without having every step fully designed. And I want to write code and call it design. Hah. It doesn't matter, as long as we both are being effective.

Some programmers are better at behind the scenes, doing library work.

Others really enjoy creating beautiful user interfaces.

Point is, we're all different and have different strengths and weaknesses. And what works for you may not work for me and visa-versa. If you take ten developers, line them up, march them off a cliff *(no wait, that's something else...)* I mean line them up and have them perform the same programming task, every one will accomplish it differently. *(I'm not talking about something simple, like echoing "Hello World!". I'm talking about a problem that takes 30-60 minutes to solve.)*

Use What Works For You

If any technique, or philosophy, or piece of advice within this book resonates with you ... then great. Use it. If something doesn't fit your tastes, just move on and ignore it.

Everything is on the table. Isn't that great? That's one of the coolest things about writing code: there's a thousand ways to skin a cat and you can learn new ways to do it all the time. You never stop learning.

A Quick Litmus Test

The joy of programming is in the journey, but the effectiveness of the coding is the destination.

So let's have some quick points that we can all agree on to evaluate the destination. Three quick questions (although I do cheat a little with #3 and make it a long question)

1. Does the program work as expected?
2. Was the coding speed sufficient?
3. Is the code both maintainable and extendable (by someone other than the original developer?)

If we achieve all three, regardless of the journey taken, then success is assured.

Chapter 11 - Decoupling is Good

Coupling measures how things are tied together in a computer program. It is the dependency one part of the program has on another part.

There are different forms: Global Coupling, Subclass Coupling, Data Coupling, … and if I thought about it I could come up with a few more or even make some up.

Coupling is measured along a continuum. When one part of a software system is dependent on another part, it is considered tightly coupled. The other extreme is to have parts loosely coupled. Of course, there are other terms that mean exactly the same thing. All these terms are used interchangeably:

- Loose Coupling = Low Coupling = Weak Coupling
- Tight Coupling = High Coupling = Strong Coupling

Decoupling is the process of loosening the coupling between two parts.

As your software system grows larger the amount of coupling indicates how tough the system will become to maintain, debug, and extend.

Why?

Because the more parts depend on each other, the more a developer must be aware of the dependencies. Code changes, be it bug fixes or adding features, necessitates time spent in different areas ensuring functionality is not broken. Tightly coupled systems that grow in size, scope, and complexity can quickly become impossible to enhance.

I've seen this countless times over the years. Applications work well initially. But features are added and the systems eventually growing too difficult to maintain. Hell, I've designed systems which become more and more difficult to maintain over time.

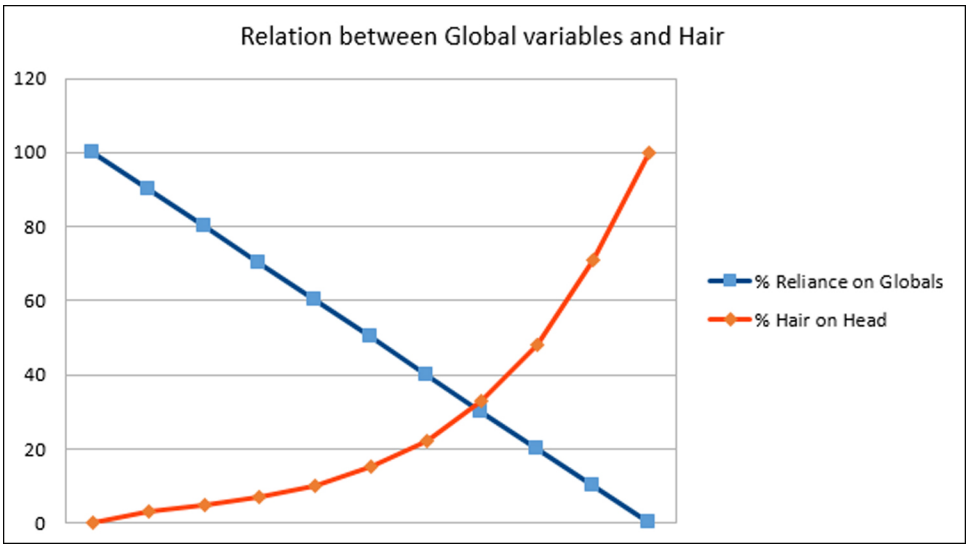
It's All About Hair Loss

Let's take PHP Global variables …

If a program relies heavily on global variables, then you could say it has tight global coupling. Every time you change a global variable then you have to take into consideration the effects to other areas of the program (be it objects, functions, methods, …) that use that particular global variable.

In a very short period of time, you will be bald from the constant hair pulling. (Yes, even you women out there.) It's a dangerous road.

Here I've created a simple chart to illustrate how the reliance on global variables affects the amount of hair a programmer has:



Scientific Analysis

As you can see, on the leftmost edge of the chart, when a programmer relies on global variables 100% of the time, they have no hair on their head.

Makes you think, doesn't it?

Every PHP programmer has heard "Global variables are bad." But isn't it cooler to say "Global Coupling is bad." It means the same thing but sounds vaguely dirty.

A Simple Decoupling Example

So how does coupling work with objects?

Let's take an typical error handling class.

```
1  class ErrorHandler {
2    public function notify($errorMessage)
3    {
4      $fp = fopen('/error.log');
5      fputs($fp, "$errorMessage\n");
6      fclose($fp);
7    }
8  }
```

Pretty straightforward. What's the problem?

The issue here is that the `ErrorHandler` class is tied to the notification of errors, namely the file system writing to an `/error.log` file.

What happens if the location of the log file changes? Or if we want to send emails instead of logging errors? To achieve this the `ErrorHandler` class must be modified.

A better solution is to abstract the actual notification of errors and pull the functionality out of the `ErrorHandler` class.

To do this we used an interface.

```
1  interface ErrorNotifierInterface {
2    public funtion sendError($errorMessage);
3  }
4
5  class ErrorHandler {
6    protected $notifier;
7
8    public function __construct(ErrorNotifierInterface $notifier)
9    {
10     $this->notifier = $notifier;
11   }
12
13   public function notify($errorMessage)
14   {
15     $this->notifier->sendError($errorMessage);
16   }
17 }
```

Now we've decoupled error notifications from the error handler. When the `ErrorHandler` object is created you specify the notifier to use. *(This process is known as **Dependency Injection** and will be discussed later.)*

We could implement an `ErrorNotifierInterface` that does exactly the same as our initial `ErrorHandler` and write to `/error.log`.

```
class StupidErrorLogImplementation implements ErrorNotifierInterface {

  public function sendError($errorMessage)
  {
    $fp = fopen('/error.log');
    fputs($fp, "$errorMessage\n");
    fclose($fp);
  }
}
```

Advantages of this approach should be self-evident.

1. The `ErrorHandler` class can be tested apart from everything else.
2. To modify how error notifications are handled, there's no need to modify the `ErrorHandler` class, simply create a new class implementing the `ErrorNotificationInterface` or update the existing implementation.

(Of course, if this was a real-life example it'd be better to set up a chain of notifiers, but the concept is the same–notification decoupled from the handler.)

This example follows SOLID principles (which I'll explain in far too much detail later in this book.)

Chapter 12 - Don’t Be a WET Programmer

There is a deceptively simple principle in software development called **DRY**, or **Don’t Repeat Yourself**.

You can apply this principle to everything from database design to documentation.

I say this principle is “deceptively simple” because the principle itself is a commandment on what to do. It’s easy to think “Oh, yeah. DRY makes it quicker to get stuff done … and there’s less code.”

Well … often programming DRY does speed up development, but sometimes it doesn’t. But development speed is not the most important benefit of DRY.

Correctly applied, DRY means there is one place in the software that is the authority for any given, uh, **thing**. *(Wow, that’s unclear. Isn’t it?)* Let’s bust apart **thing**. It is: a piece of knowledge, a business rule, a step, a function, an action.

Doesn’t really matter what we’re talking about with DRY, the biggest benefit is that there’s one place that you go to in the code to change that **thing**.

Principles aren’t laws

Remember all design principles are just that … principles. There’s no Code Gestapo that’s going to take a ball peen hammer to your kneecaps if you don’t follow a principle. The master programmer ignores principles all the time. As long as you’re aware of what principle you’re breaking and why … then great.

Chapter 13 - Dependency Injection

Dependency Injection is setting up a structure that allows the decision of what classes an object uses to happen outside the object itself.

Why? It all comes back to decoupling. Consider the following method in a class.

```
class AutomaticStockTrader {
  public function doTrades($symbol)
  {
    $analyzer = new StockTradeAnalyzer();
    $numShares = $analyzer->countSellsNeeded($symbol);
    if ($numShares > 0)
    {
      $broker = new StockBroker();
      $broker->executeSell($symbol, $numShares);
    }
  }
}
```

Here we have a nifty little class to sell some stock automatically. The `doTrades()` method is dependent on the `StockTradeAnalyzer` and `StockBroker` classes. Dependency is okay, in fact you really can't get away from it. The important thing is where the decisions on dependencies are made.

That's important, let me repeat. Dependencies cannot be avoided so ...

> The important thing is where dependency decisions are made.

In the case of objects, the question of "where" often becomes "who." *(Because I like to think of objects as people. They're my friends.)*

In the above example, the `doTrades()` method itself makes the decision on who it's dependent on.

Watch for the new keyword

Using **new** within methods is often a big, red flag there's some tight coupling going on.

What we want to do is move the decision out of the `doTrades()` method. We'll do this in 3 steps.

Get it done quicker

When actually coding, the following three steps are combined into one step, allowing you to get it done quicker. I'm breaking the steps apart for illustration.

Step 1 - Move the dependency decision to the class level

The first step is to let the class make the decision, not the method.

```
1  class AutomaticStockTrader {
2    protected $analyzer;
3    protected $broker;
4    public function __construct()
5    {
6      $this->analyzer = new StockTradeAnalyzer();
7      $this->broker = new StockBroker();
8    }
9    public function doTrades($symbol)
10   {
11     $numShares = $this->analyzer->countSellsNeeded($symbol);
12     if ($numShares > 0)
13     {
14       $this->broker->executeSell($symbol, $numShares);
15     }
16   }
17 }
```

Pretty straightforward. No new classes. Just a few extra lines of code. But now all the dependencies are in one place: the constructor.

Step 2 - Use manual injection

Now that our class has all the decisions about which dependencies to use in the constructor, let's move the decision outside the class by using dependency injection.

```php
interface StockTradeAnalyzerInterface {
    public function countSellsNeeded($symbol);
}
interface StockBrokerInterface {
    public function executeSell($symbol, $numShares);
}
class AutomaticStockTrader {
  protected $analyzer;
  protected $broker;
  public function __construct(StockTradeAnalyzerInterface $analyzer,
    StockBrokerInterface $broker)
  {
    $this->analyzer = $analyzer;
    $this->broker = $broker;
  }
  public function doTrades($symbol)
  {
    $numShares = $this->analyzer->countSellsNeeded($symbol);
    if ($numShares > 0)
    {
      $this->broker->executeSell($symbol, $numShares);
    }
  }
}
```

Another small step. Here the decision must be made by whoever creates the `AutomaticStockTrader` instance. The dependencies are *injected* into the class at construction time.

Notice I also created interfaces here instead of the classes named `StockTradeAnalyzer` and `StockBroker`? This is because we really don't care what objects are injected, as long as they implement these methods we want (`countSellsNeeded()` and `executeSell()`).

Unit Testing and Dependency Injection

When your classes use Dependency Injection, unit testing is a breeze. Just mock any dependencies, inject them in your tests, and BAM! Bob's your uncle.

Step 3 - Use automatic injection

Laravel provides an often overlooked, but important feature: interface binding. With interface binding you specify the default concrete classes to use with interfaces. Interface binding let's you achieve automatic dependency injection.

Very cool.

```
App::bind('StockTradeAnalyzerInterface', function()
{
  return new StockTradeAnalyzer();
});
App::bind('StockBrokerInterface', 'StockBroker');
```

(The first binding above uses a Closure. It could just as easily specified the class name like the second binding does. Just two ways to achieve the same thing.)

This is IoC Binding

The above code example introduces the concept of binding. It uses Laravel's IoC container which will be discussed in the next chapter.

Now the decision on dependencies can still occur when you construct a new `AutomaticStockTrader` instance. (Such as when unit testing.) Or, you can decide to use whatever the defaults are.

To use bound interfaces, a different method of construction should be used.

```
// Instead of
$trader = new AutomaticStockTrader();

// Use
$trader = App::make('AutomaticStockTrader');
```

Laravel is smart enough to implement any automatically injected dependencies when you use `App::make()`.

Don't forget setters

The above illustrated *Constructor Dependency Injection*. You should be aware of *Setter Dependency Injection*. For example we could add a `setAnalyzer()` method to the `AutomaticStockTrader` class.

```
class AutomaticStockTrader {
  public function setAnalyzer(StockTradeAnalyzerInterface $analyzer)
  {
    $this->analyzer = $analyzer;
  }
}
```

Chapter 14 - Inversion of Control

Inversion of Control is one of those software engineering concepts that carries a lot of baggage. What does it mean? What does it do? Why are we even talking about it? Will the Red Sox win the series?

Well …

The problem is that *Inversion of Control* is a general concept, and usually we talk about it in specific contexts.

A General Example

Let's think of it this way. The way we're taught a program works is linear. Consider a "Hello World" application.

1. The user executes the hello world program.
2. The operating system loads the program, any associate libraries, and passes "control" to the program.
3. The program says "System, I am the master. You will do my bidding. I command you to display "Hello World."
4. The system thinks "Uh, okay. Wow. The ego of some programs. Whatever." and displays "Hello World."
5. The program, satisfied with a job well done, dies.
6. The operating system regains control, shaking its head at the hubris of some programs.

If we were to flip things around, then a different scenario emerges.

1. The user executes the hello world program.
2. The operating system loads the program and any associated libraries.
3. The operating system asks the program if it has any messages.
4. The program pauses for a bit, then responds in a rush "Uh sorry sir. I was … I was thinking. Uh, yeah, can you please display 'Hello World'."
5. The system doesn't respond. It received the message but it's got more important things to worry about.
6. The program kills itself. Nobody mourns its passing.
7. The system finds itself with an idle few milliseconds, says "What was I doing? Oh yeah." and displays "Hello World."

This silly example illustrates inverting the control between and operating system and program.

It's really the same within a program, where methods and classes give up control where possible.

In the context of software development, it's all about decoupling. Yes, **decoupling again**.

Dependency Injection Again

In the previous chapter about Dependency Injection, we had a class named `AutomaticStockTrader` and within the class, a method named `doTrades()`. First the `doTrades()` gave up control of it's dependencies to the class. Next we had the class give up control of it's dependencies to whoever constructed the class.

Dependency Injection is an implementation of Inversion of Control–as are Factory Patterns.

But, with Laravel, most often Inversion of Control is talking about …

The IoC Container

Laravel implements an Inversion of Control container. In fact, the entire framework is built on this container.

The IoC Container is Laravel's skeleton

When you strip everything else away from Laravel: the elegant ORM, the cool routing, the style, the class, the beauty, you are left with the bare bones of the framework. These bones are the IoC Container.

Laravel's IoC container is a global object registry. But it's more than that. It's a registry of **how to create objects**. Think about that for a second. What power?

And the purpose behind all of this is (yet again I will flog this almost dead horse) to allow decoupling.

The Problem With Dependency Injection

Dependency Injection, or more specifically, Constructor and Setter Dependency Injection does have a problem. When the number of dependencies a class relies on is more than 2 or 3, then things get hairy. The programmer must constantly look up what to inject. Can you remember the order to pass six different arguments to the constructor every time you create an object? I know I can't. Luckily, using IoC Binding takes care of this need.

IoC Binding

As discussed in the previous chapter, IoC Binding let's you bind interfaces to the actual implementation. In other words you can specify how an interface will, by default, resolve to a class.

```
App::bind('SomeInterface', function()
{
  return new ClassForSomeInterface();
});
```

App is Laravel's Application class which is really a facade of the IoC container. Interesting, huh? *(If you don't fully understand what that sentence means, don't worry. You will before finishing this book.)*

Here's a question ... since you're setting up *defaults* for your interfaces using IoC Binding, then why pass arguments to the constructor?

The answer ... you don't need to.

Let's revisit that class from last chapter. Here's what we had:

```
interface StockTradeAnalyzerInterface {
    public function countSellsNeeded($symbol);
}
interface StockBrokerInterface {
    public function executeSell($symbol, $numShares);
}
class AutomaticStockTrader {
  protected $analyzer;
  protected $broker;
  public function __construct(StockTradeAnalyzerInterface $analyzer,
    StockBrokerInterface $broker)
  {
    $this->analyzer = $analyzer;
    $this->broker = $broker;
  }
  public function doTrades($symbol)
  {
    $numShares = $this->analyzer->countSellsNeeded($symbol);
    if ($numShares > 0)
    {
      $this->broker->executeSell($symbol, $numShares);
    }
  }
}
App::bind('StockTradeAnalyzerInterface', function()
{
  return new StockTradeAnalyzer();
});
App::bind('StockBrokerInterface', 'StockBroker');
```

We could construct an `AutomaticStockTrader` without any constructor arguments and still have the constructor inject dependencies. What if we rewrote the class and used the IoC container to inject dependencies?

```
class AutomaticStockTrader {
  protected $analyzer;
  protected $broker;
  public function __construct()
  {
    $this->analyzer = App::make('StockTradeAnalyzerInterface');
    $this->broker = App::make('StockBrokerInterface');
  }
  public function doTrades($symbol)
  {
    $numShares = $this->analyzer->countSellsNeeded($symbol);
    if ($numShares > 0)
    {
      $this->broker->executeSell($symbol, $numShares);
    }
  }
}
```

Perfect.

You may be thinking *"Which way's better?"*

How I Decide

If the class only has 2 or 3 dependencies needing injecting, I'll usually use constructor arguments. This makes testing slightly easier. Any more than 2 or 3 dependencies and I have the constructor inject the dependencies using the `App::make()` method. This makes it easier on my brain.

Chapter 15 - Interface As Contract

I've discussed using interfaces already in this book. There is a nifty way to think about interfaces: *the interface is a contract.*

I must admit, the first time I heard (or read) Taylor Otwell mention **Interface as Contract** I wasn't very interested. I thought he was talking about **Design by Contract**, which is a formal process I didn't want to get bogged down in.

How wrong I was.

He was literally talking about using interfaces as contracts. (Heh, pretty much exactly what he said, I was just too dumb to realize.) In other words, as a PHP programmer you make use of interfaces. And you program to the interfaces. The interfaces become a sort of a mini-contract as to what the actual class is going to do.

How simple. I used to use interfaces quite a bit in my coding, now I use them all the time. Maybe almost as much as a .NET programmer.

Interfaces hide code

Surprisingly, not every PHP programmer has jumped on the interface bandwagon. I think it's because of wrong thinking about what an interface is used for.

Obviously I'm not in your head, and I don't know what you're thinking, but I can describe what was rattling around in my brain when I met Mr. PHP Interface.

ME: "Oh, Hello Mr. Interface. Looking for a job I see?"

Interface: Nods enthusiastically.

ME: "Well, I do like how clean you look."

Interface: Blushes.

ME: "But you know, I use lots of abstract classes. I think I got the job handled."

Interface: Trudges off, dejectedly.

My problem was I didn't understand the greatest benefit of interfaces: HIDING CODE.

See, an interface allows you to design a class without implementing it and it hides the actual implementation of the class from you. An interface contains no actual code. It is a contract for what the code does. Simple.

```php
interface OrderProcessorInterface {
    public function checkOrderStatus(Order $order);
    public function fulfillOrder(Order $order);
    public function voidOrder(Order $order);
}
```

So, in this example I could write code based on the `OrderProcessorInterface` without ever even looking at the actual implementation. And, if I'm part of a team on a project, I may never view the implementation.

Chapter 16 - SOLID Object Design

There is a set of principles in object-oriented design called **SOLID**. It's an acronym standing for:

- **[S]**ingle Responsibility
- **[O]**pen/Closed Principle
- **[L]**iskov Substitution Principle
- **[I]**nterface Segregation Principle
- **[D]**ependency Inversion Principle

Together, they represent a set of best practices which, when followed, makes it more likely software you develop will be easier to maintain and extend over time.

This chapter explains each principle in greater detail.

Do me a SOLID?

If a programmer comes up to you and says, "Can you do me a SOLID?" Make sure you understand what you're being asked.

Single Responsibility Principle

The first letter, 'S' in the SOLID acronym, stands for Single Responsibility. The principle states:

> *"Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility."*

Another way to think about this is that a class should have one, and only one, reason to change.

Let's say you have a class that compiles and emails the user a marketing report. What happens if the delivery method changes? What if the user needs to get the report via text? Or on the web? What if the report format changes? Or the report's data source changes? There are many potential reasons for this class to change.

State the class Purpose without AND

A simple way to implement the Single Responsibility Principle is to be able to define what the class does without using the word AND

Let's illustrate this with some code ...

```php
// Class Purpose: Compile AND email a user a marketing report.
class MarketingReport {
  public function execute($reportName, $userEmail)
  {
    $report = $this->compileReport($reportName);
    $this->emailUser($report, $userEmail);
  }
  private function compileReport($reportName) { ... }
  private function emailUser($report, $email) { ... }
}
```

In this case we'd break the class into two classes and push the decision as to what report and who/how to send it up higher in the food chain.

```php
// Class Purpose: Compiles a marketing report.
class MarketingReporter {
  public function compileReport($reportName) { ... }
}

interface ReportNotifierInterface {
  public function sendReport($content, $destination);
}

// Class Purpose: Emails a report to a user
class ReportEmailer implements ReportNotifierInterface {
  public function sendReport($content, $email) { ... }
}
```

Notice how I snuck that interface in there? Aye, I be a slippery bugger with them thar interfaces.

More Robust Classes

By following the Single Responsibility Principle you'll end up with robust classes. Since you're focusing on a single concern, there's less likelihood of any code changes within the class breaking functionality outside the class.

Open/Closed Principle

The second letter in the SOLID acronym stands for the Open/Closed principle. This principle states:

> *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."*

In other words, once you've coded a class you should never have to change that code except to fix bugs. If additional functionality is needed then the class can be extended.

This principle forces you to think "how could this be changed?" Experience is the best teacher here. You learn to design software setting up patterns that you've commonly used in earlier situations.

Don't Be Too Strict

I've found strict adherence to this principle can result in software that is over-engineered. This occurs when programmers try to think of every potential way a class could be used. A little bit of this thinking is a good thing, but too much can result in more complex classes or groups of classes that have no true need of being so complex.

Sorry if this offends die-hard fans of this principle, but hey, I just calls 'em like I sees 'em. It's a principle, not a law.

Nevertheless, let's work out a common example. Let's say you're working on an account biller, specifically the portion that processes refunds.

```
1  class AccountRefundProcessor {
2    protected $repository;
3
4    public function __construct(AccountRepositoryInterface $repo)
5    {
6      $this->repository = $repo;
7    }
8    public function process()
9    {
10     foreach ($this->repository->getAllAccounts() as $account)
11     {
12       if ($account->isRefundDue())
13       {
14         $this->processSingleRefund($account);
15       }
16     }
17   }
18 }
```

Okay, let's look at the above code. We're using Dependency Injection thus the storage of accounts is separated off into its own repository class. Nice.

Unfortunately, you arrive at work one day and your boss is upset. Apparently, management has decided that accounts due large refunds should have a manual review. Uggh.

Okay, what's wrong with the code above? You're going to have to modify it, so it's not closed to modifications. You could extend it with a subclass, but then you'll have to duplicate most of the `process()` code.

So you set out to refactor the refund processor.

```
interface AccountRefundValidatorInterface {
  public function isValid(Account $account);
}
class AccountRefundDueValidator implements AccountRefundValidatorInterface {
  public function isValid(Account $account)
  {
    return ($account->balance > 0) ? true : false;
  }
}
class AccountRefundReviewedValidator implements
  AccountRefundValidatorInterface {
  public function isValid(Account $account)
  {
    if ($account->balance > 1000)
    {
      return $account->hasBeenReviewed;
    }
    return true;
  }
}
class AccountRefundProcessor {
  protected $repository;
  protected $validators;

  public function __construct(AccountRepositoryInterface $repo,
    array $validators)
  {
    $this->repository = $repo;
    $this->validators = $validators;
  }
  public function process()
  {
    foreach ($this->repository->getAllAccounts() as $account)
    {
      $refundIsValid = true;
```

```
36        foreach ($this->validators as $validator)
37        {
38          $refundIsValid = ($refundIsValid and $validator->isValid($account));
39        }
40        if ($refundIsValid)
41        {
42          $this->processSingleRefund($account);
43        }
44      }
45    }
46  }
```

Now `AccountRefundProcess` takes an array of validators during construction. The next time business rules change you can whip out a new validator, add it to the class construction, and you're golden.

Liskov Substitution Principle

The "L" in SOLID stands for Liskov substitution principle. This principle states:

> *"In a computer program if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be substituted for objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)."*

Huh? I'd hazard a guess that this part of the SOLID design causes more confusion than just about any other.

It's really all about **Substitutability**, but if we used the 'S' from Substitutability our acronym becomes SOSID instead of SOLID. Can you imagine the conversations?

Programmer 1: "We should follow the S.O.S.I.D. principles when designing classes."

Programmer 2: "Sausage?"

Programmer 1: "No, SOSID."

Programmer 2: "Saucy?"

Programmer 1: Calls up Michael Feathers on the phone ... "Hey, we need a better acronym."

Simply stated, the Liskov Substitution principle means that your program should be able to use sub-types of classes wherever they use a class.

In other words if you have a Rectangle class, and your program uses a Rectangle class and you have a Square class derived from the Rectangle. Then the program should be able to use the Square class anywhere it uses a Rectangle.

Still confused? I know I was confused initially because I wanted to say "Duh? This is pretty obvious", yet I feared there was something I didn't understand. I mean, why have this as a major tenet of SOLID design if it's so obvious?

Turns out there are some subtle aspects to this principle stating preconditions of subtypes should not be strengthened and postconditions should not be weakened. Or is it the other way around? And you can't have subtypes throw exceptions that aren't the same (or derived from) exceptions the supertype throws.

Wow! What else? There are other details about Liskov's principle I won't get into here and I fear I've really spent too much time on it.

Liskov's principle doesn't matter.

What?

Yeah! I said it. (Technically the principle does matter, so I'm lying to you, but I'll try to justify my lie.)

In PHP, if you follow three guidelines then you're covered 99% of the time with Liskov.

1 - Use Interfaces

Use interfaces. Use them everywhere that makes sense. There's not much overhead in adding interfaces and the result is that you have a "pure" level of abstraction that almost guarantees Liskov Substitution is followed.

2 - Keep implementation details out of interfaces

When you're using interfaces, don't have the interface expose any implementation details. Why would you have a `UserAccountInterface` provide any details on storage space left?

3 - Watch your type-checking code.

When you write code that operates differently on certain types, you may be breaking Liskov's Substitution Principle (and probably several other SOLID principles).

Consider the following code:

```php
class PluginManager {
  protected $plugins; // array of plugins

  public function add(PluginInterface $plugin)
  {
    if ($plugin instanceof SessionHandler)
    {
      // Only add if user logged on.
      if ( ! Auth::check())
      {
        return;
      }
    }
    $this->plugins[] = $plugin;
  }
}
```

What's incorrect about that piece of code? It breaks Liskov because the task performed is different depending on the object type. This is a common enough snippet of code and in a small application, in the spirit of getting stuff done, I'd be perfectly happy to let it slide.

But ... why should an `add()` method get picky about what's added. If you think about it, `add()`'s being a bit of a control freak. It needs to take a deep breath and give up control. *(Hmmm, Inversion of Control)*

Interface Segregation Principle

The 'I' in SOLID stands for the Interface Segregation Principle. Interface Segregation states:

> *"No client should be forced to depend on methods it does not use."*

In plain English this means don't have bloated interfaces. If you follow the "Single Responsibility Principle" with your interfaces, then this usually isn't an issue.

The app we're designing here will be a small application and this principle probably won't apply too much. Interface Segregation becomes more important the larger your application becomes.

Often this principle is violated with drivers. Let's say you have a cache driver. At it's simplest a cache is really nothing more than temporary storage for a data value. Thus a `save()` or `put()` method would be needed and a corresponding `load()` or `get()` method. But often what happens is the designer of the cache wants it to have extra bells and whistles and will design an interface like:

```
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
  public function getLastAccess($key);
  public function getNumHits($key);
  public function callBobForSomeCash();
}
```

Let's pretend we implement the cache and realize that the `getLastAccess()` is impossible to implement because your storage doesn't allow it. Likewise, `getNumHits()` is problematic. And we have no clue how to `callBobForSomeCash()`. *Who is this Bob guy? And does he give cash to anyone that calls?* When implementing the interface we decide to just throw exceptions.

```
class StupidCache implements CacheInterface {
  public function put($key, $value, $expires) { ... }
  public function get($key) { ... }
  public function clear($key) { ... }
  public function clearAll() { ... }
  public function getLastAccess($key)
  {
    throw new BadMethodCallException('not implemented');
  }
  public function getNumHits($key)
  {
    throw new BadMethodCallException('not implemented');
  }
  public function callBobForSomeCache()
  {
    throw new BadMethodCallException('not implemented');
  }
}
```

Ugh. Ugly right?

That's the crux of the Interface Segregation Principle. Instead, you should create smaller interfaces such as:

```php
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
}
interface CacheTrackableInterface {
  public function getLastAccess($key);
  public function getNumHits($key);
}
interface CacheFromBobInterface {
  public function callBobForSomeCash();
}
```

Make sense?

Dependency Inversion Principle

The final 'D' in SOLID stands for Dependency Inversion Principle. This states two things:

> *"A. High-level modules should not depend on low-level modules. Both should depend on abstractions."*
>
> *"B. Abstractions should not depend upon details. Details should depend upon abstractions."*

Great so what does this mean?

High-level
: High-level code is usually more complex and relies on the functioning of low-level code.

Low-level
: Low-level code performs basic, focused operations such as accessing the file system or managing a database.

There's a spectrum between low-level and high-level. For example, I consider session management low-level code. Yet session management relies on other low-level code for session storage.

It's useful to think of high-level and low-level in relation to each other. Session management is definitely lower than application specific functionality such as logging a user in, but session management is high-level to the database access layer.

I've heard people say ... "Oh, Dependency Inversion is when you implement Inversion of Control." or "No. Dependency Injection is what Dependency Inversion is." Both answers are partially correct, because both answers can implement Dependency Inversion.

The Dependency Inversion Principle could be better stated as a technique of decoupling class dependencies. By decoupling, both the high-level logic and low-level objects don't rely directly on each other, instead they rely on abstractions.

In the PHP world ... Dependency Inversion is best achieved through what? You guessed it: interfaces.

Isn't it interesting how all these design principles work together? And how often these principles bring into play that most unused of PHP constructs: the interface?

The Simple Rule of Dependency Inversion

Make objects your high-level code uses **always** be through interfaces. And have your low-level code implement those interfaces, and you've nailed this principle.

An Example

User authentication is a good example. At the highest level is the code that authenticates a user.

```
1  <?php
2  interface UserInterface {
3    public function getPassword();
4  }
5
6  interface UserAuthRepositoryInterface {
7    /**
8     * Return UserInterface from the $username
9     */
10   public function fetchByUsername($username);
11 }
12
13 class UserAuth {
14   protected $repository;
15
16   /**
17    * Inject repository dependency
18    */
19   public function __construct(UserAuthRepositoryInterface $repo)
```

```
20    {
21      $this->repository = $repo;
22    }
23
24    /**
25     * Return true if the $username and $password are valid
26     */
27    public function isValid($username, $password)
28    {
29      $user = $this->repository->fetchByUsername($username);
30      if ($user and $user->getPassword() == $password)
31      {
32        return true;
33      }
34      return false;
35    }
36  }
37  ?>
```

So we have the high-level class `UserAuth`, relying on the abstractions of `UserAuthRepositoryInterface` and `UserInterface`. Now, implementing those two abstractions are almost trivial.

```
1   <?php
2   class User extends Eloquent implements UserInterface {
3     public function getPassword()
4     {
5       return $this->password;
6     }
7   }
8   class EloquentUserRepository implements UserAuthRepositoryInterface {
9     public function fetchByUsername($username)
10    {
11      return User::where('username', '=', $username)->first();
12    }
13  }
14  ?>
```

Easy, peasey, lemon-squeezy.

Now, to use the `UserAuth` we would either construct it with the `EloquentUserRepository` or bind the interface to the `EloquentUserRepository` to automatically inject it.

Do not confuse the authentication examples used in this chapter with the Laravel `Auth` facade. These are just examples to illustrate the principle. Laravel's implementation, though similar to these, is far better.

Chapter 17 - A Dirty, Little Design Secret

> *You probably want to skip this chapter. It doesn't add anything to this book. I'll probably be in a curmudgeony mood soon and delete this chapter.*

Okay, I'm going to go on one final tangent before we get to designing the application … a dirty little secret about design.

(You think this is really the last tangent I'll go on? Oy, you should be so lucky!)

Software design does not occur like you learn in school.

Since I've been doing this over 3 decades, I learned the old-school method:

1. Design the output
2. Design the input
3. Design the database
4. Design the functionality

There's all these new-fangled software development methodologies. Sometimes it seems there's too much focus on paperwork. If you've ever heard someone say "OMG, Gary's flipping out because your FSD didn't account for a whole section of my PRD" then you know what I mean. (Translation of the proceeding: you're in trouble because the program specs you created missed part of the requirements doc.)

Why did we have to get so OCD about software development? I mean, you're the programmer, just make it work. All that other stuff, fuggetaboutit.

Even the way I learned, the simple 4 steps above doesn't work because of one simple reason: IT'S ALL INTERALATED. YOU MUST DO IT ALL AT ONCE.

Design is messy.

You work on step 1, then step 2, then back to step 1, then skip to step 3, then step 1 again, then step 4. Yeesh.

It's easier to teach a linear process: first this, then that. In reality, you do it all simultaneously for the quickest result.

That's the point: *Getting Stuff Done Quickly*.

That's what my step #2 in the chapter on **Cowboy Coding** is all about, doing it all simultaneously.

Programming like a Novelist

A novelist combines art and craft to create a work of fiction. There's definite craft past the word choices and grammar. Issues like pacing, characterization, cliff hangers, hooking the reader, to just name a few. Yet, novelists use this craft to create art.

There are two extremes in novel writing methodologies: those that outline everything before writing a word and those that sit down and just make it up as they go along. I liken software development methodologies to outlining first–but unlike most novelists, this outlining is often done in teams. And we know how productive meetings often are.

At first glance the novelist that just wings it seems irresponsible. I mean they're just making stuff up. But what's really going on is that novelist has the craft of writing so ingrained, it's down there at the subconscious level. They are free to focus on the story they're writing and all the craft they need pours out of the fingertips as they type.

I enjoy programming like a novelist: being a programmer who just sits down and is able to focus on creating great software and all the programming craft just seeps into the project from the subconscious when needed.

Sorry if this sounds a bit "woo-woo," but I truly believe to the extent we as programmers are able to do this, we start combining the best of being a craftsman with being an artist.

And that's why Cowboy Coding is the best way to program. Laravel provides a solid framework which frees PHP programmers from mundane craft issues, allowing them to focus on creating great software.

Part 2 - Designing the Application

Finally, after all those long and boring discussions in the last part of the book, we'll get to designing the actual application.

Yes, there's gonna be some actual code to write in this part of the book. But first …

Chapter 18 - What application will we create?

In this Chapter

In this chapter we'll finally get around to deciding what application we're going to create. Still no actual coding, but it's coming real soon now.

What application are we going to create in this book? I want something cool. Something people can use. Something … well, let's make a list:

- The application must be usable and helpful.
- The application should illustrate creating command line utilities in Laravel.
- The application should also have a web component.
- The application should be easily customized by any programmer in order to make it more useful for him or her.
- The application should be simple to create.
- The application must show some cool Laravel tricks.
- Best practices of design should be followed creating the application.

Da, da-duh. And we'll be creating a "TODO List".

Okay, I can hear a collective mental groan (odd because the groan is in the future from the point in the time-stream that I'm writing this. That's the magic of writing.).

Just hold on a minute. Let me explain why this will be a great little application.

First let's store our todo lists (yeah, plural, more than one list) in simple text files. This way we can edit our lists using text editors if that's what we want to do. Similar to todo.txt[20] by Gina Trapani.

Let's have a series of commands we can use from the console window that makes it easy to add, list, mark things complete, etc. Again, I'm thinking of something similar to todo.txt.

Let's add a simple web interface that we can use on our own machine that lets us add, list, mark things complete, etc., from a browser window.

Are you still groaning, thinking this is going to be lame? I hope you've moved from believing this is going to be a stupid waste of time to maybe–just maybe–having a bit of skepticism, but being cautiously hopeful.

[20] https://github.com/ginatrapani/todo.txt-cli

Naming the application

I named this book and the application pretty much the same thing. This allows the book to have multiple meanings. It's not only what I hope you get out of the book, but is also the name of the application.

Slick, huh?

Here's how the name came about ...

At first I was thinking of calling this book *Getting Things Done with Laravel* but felt that was too close to the David Allen Getting Things Done[21]. Although, I do borrow heavily from his system.

Next I thought of using *Todo with Laravel* but that just seemed really weak.

I briefly entertained the idea of *Git-R-Done with Laravel* but didn't want Larry the Cable Guy[22] to sue me.

Getting Shit Done with Laravel sounded good to me, but just a bit too crass. So I softened the title to *Getting Stuff Done with Laravel.*

That's the name of this application **Getting Stuff Done** or GSD for short.

What is GTD

Getting Things Done by David Allen is both a method and workflow for managing all the "stuff" we have to accomplish. I recommend it highly. This book isn't going to create a GTD system, but it could be used for that.

Most of GTD is really workflow. Clear you inbox, decide what to do with each item once while you're "Processing". If it's something you can do in under 2 minutes, then just do it. If it's something actionable then is it part of a project? Is it something you want to do later? All these decisions help you determine what list the item belongs to.

So, besides the workflow, GTD is really nothing other than a way to manage lists of stuff to do. You have your "Waiting For" list, your list of projects, a list for each project, a bucket for all your next actions, and so forth.

> *"It's full of stars!" – David Bowman, 2001: A Space Odyssey"*

Or in our case: *it's full of lists.* An application that helps manage a bunch of todo lists.

This will be fun to create.

[21] http://en.wikipedia.org/wiki/Getting_Things_Done
[22] http://en.wikipedia.org/wiki/Larry_the_Cable_Guy

Chapter 19 - Installing Laravel

In this Chapter

In this chapter we'll create the project and establish its structure.

Since we're going to design some classes and interfaces, we want to have a place to put source code files. So let's set up a new Laravel project.

It's a bit of a misnomer to say you're *"Installing Laravel"* because that's not precisely what's going on. You're actually creating a project structure and downloading Laravel as one of the dependencies.

But if you tell me you're "Installing Laravel" I know what you mean.

You need to have PHP 5.4 or greater, Composer[23] installed, and PHPUnit[24] installed.

Try checking this from your terminal window[25].

```
$ php --version
PHP 5.4.6-1ubuntu1.4 (cli) (built: Sep  4 2013 19:36:09)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.4.0, Copyright (c) 1998-2012 Zend Technologies
    with Xdebug v2.2.1, Copyright (c) 2002-2012, by Derick Rethans
$ composer --version
Composer version 815f7687c5d58af2b31df680d2a715f7eb8dbf62
$ phpunit --version
PHPUnit 3.7.26 by Sebastian Bergmann.
```

As long as you have PHP 5.3 or above, most everything in this book should work. But since I'm using PHP 5.4, I know it works with this version. I've successfully used Laravel 4 with version PHP 5.3.3, but I wasn't using the `Hash` functions (they require PHP 5.3.7 or above).

If Composer isn't installed on your machine, head down to Appendix I for the simple instructions to get it installed. Likewise, refer to Appendix II if you need instructions for installing PHPUnit.

If you're using phpunit and it's version 3.6 then you may get errors when mocking objects. Upgrade to the latest, following the instructions in the appendix.

[23] http://getcomposer.org/

[24] https://github.com/sebastianbergmann/phpunit/

[25] Whenever I mention *terminal window* in this book I'm referring to Gnome Terminal in Linux. You would use the equivalent in your Operating System.

Creating the project

Unlike most of the other frameworks, Laravel uses composer which means the framework is installed within your project's `vendor` directory. This allows different projects to run with different versions of Laravel.

To create a laravel project, issue the command below. *(Note the `gsd` portion of the command is the destination folder ... where our project will live.)*

```
$ composer create-project laravel/laravel --prefer-dist gsd
```

After a few minutes, all kinds of messages will scroll by:

```
Installing laravel/laravel (v4.0.6)
  - Installing laravel/laravel (v4.0.6)
    Downloading: 100%

Created project in gsd
Loading composer repositories with package information
Installing dependencies (including require-dev)
  - Installing psr/log (1.0.0)
    Downloading: 100%

  [snip many, many lines]

Writing lock file
Generating autoload files
Generating optimized class loader
Application key [xW1cd5f4im7GiWN9LFAbir3GUnr51DeE] set successfully.
```

Of course, the specifics of what shows will be different for you.

(Your setup may not show the Application key assignment at the end. It's not required for this application, but if you'd like to set it simply issue a `php artisan key:generate` command.)

The project hierarchy

Now, that you've successfully created a new project, you have a series of directories. Here's a list and what each is used for.

```
gsd : project directory
|- app : application directory
|---- commands : console commands
|---- config : configuration files
|---- controllers : where controllers go
|---- database : database stuff
|------- migrations : where migrations are stored
|------- seeds : where seeding logging is stored
|---- lang : language translations
|---- models : where models are stored
|---- start : application startup files
|---- storage : directories for disk storage
|------- cache : where cached data is stored
|------- logs : where logs are stored
|------- meta : meta information storage
|------- session : session storage area
|------- views : storage of assembled blade templates
|---- tests : unit tests
|---- views : blade templates
|- bootstrap : framework bootstrap files
|- public : document root for web applications
|- vendor : composer installed dependencies
```

We're going to change this structure a bit and use PSR-0[26] structuring to organize our project.

Delete unneeded directories

Because we're changing the structure to PSR-0, we can get rid of the directories that usually hold certain "types" of files. And since this application isn't going to use a database, the folder for migrations and seeds can be wiped out also.

```
~$ cd gsd/app
~/gsd/app$ rm -rf commands
~/gsd/app$ rm -rf controllers
~/gsd/app$ rm -rf database
~/gsd/app$ rm -rf models
```

All gone.

[26] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md

Create the Source directories

We'll put all of our source code in a new structure that will follow the namespacing of each class.

```
~$ cd gsd/app
~/gsd/app $ mkdir src
~/gsd/app $ mkdir src/GSD
~/gsd/app $ mkdir src/GSD/Commands
~/gsd/app $ mkdir src/GSD/Controllers
~/gsd/app $ mkdir src/GSD/Entities
~/gsd/app $ mkdir src/GSD/Repositories
~/gsd/app $ mkdir src/GSD/Providers
```

These are just the standard directories I set up. If I don't need them then they'll be deleted later. Here's a breakdown of what they're used for.

Commands
: Holds console commands.

Controllers
: Holds web controllers.

Entities
: Similar to the old `models` directory, this is the place to stash entities.

Repositories
: Here's where our data storage code will be stashed.

Providers
: If this application ends up using any Service Providers, then I'll stash them here.

Update Composer

Now we'll edit Composer's configuration file to tell it about our new structure and then update our installation. So first, edit `composer.json` to match what's below.

```
{
  "name": "gsd",
  "description": "Getting Stuff Done with Laravel.",
  "require": {
    "laravel/framework": "4.0.*"  (this may be 4.1 too, don't change it)
  },
  "autoload": {
    "psr-0": {
      "GSD": "app/src"
    },
    "classmap": [
      "app/tests/TestCase.php"
    ]
  },

  ** rest of file unchanged **
```

Next, you'll want to update everything. You probably could get by just doing a `composer dump-autoload` but I like doing a `composer update` instead.

```
~$ cd gsd
~/gsd$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)

  [you may have multiple package names listed]

Writing lock file
Generating autoload files
Generating optimized class loader
```

Test Your Installation

Although there's no functionality yet, you can run `artisan` and `phpunit` to make sure all the pieces are in place.

```
~$ cd gsd
~/gsd$ php artisan --version
Laravel Framework version 4.0.6
~/gsd$ phpunit
PHPUnit 3.7.26 by Sebastian Bergmann.

Configuration read from /home/chuck/gsd/phpunit.xml

.

Time: 0 seconds, Memory: 8.25Mb

OK (1 test, 1 assertion)
```

We're done setting things up.

Chapter 20 - Designing a Todo List

In this Chapter

In this chapter we'll start planning out what a todo list actually looks like.

The only thing I know about this application at this point is that it's going to manage a list of todo lists. And that I'll store the individual lists in files on my hard disk to allow direct editing if I want.

Configuration Data

We need to know where the folder containing todo lists will be. This is a piece of configuration data to add in Laravel's configuration.

Add the following values to the top of `app/config/app.php`.

```
return array(

  'gsd' => array(
    'folder' => '/home/chuck/Documents/gsd/', // use value appropriate for you
    'extension' => '.txt',
  ),
  // rest of file the same
```

Now Laraval's `Config` facade could be used to access these configuration values.

```
$folder = Config::get('app.gsd.folder');
$extension = Config::get('app.gsd.extension');
```

About Facades

A common misconception about Laravel 4 is there are too many static classes. Nothing can be further from the truth. Laravel 4 makes extensive use of static facades.

```php
// It looks like a static class
$x = Config::get('some.config.value');

// But the above equates to
$x = Facade::$app['config']->get('some.config.value');
```

The staticness is really just syntactical sugar. Doesn't the simple `Config::get` look prettier than the alternative?

Laravel's Configuration and Environments

You may have noticed there's a number of files in the `app/config` directory. There's even some subdirectories. These files are where you store the configuration of your application.

The file itself is the configuration category. So if you're configuring the **session**, you'd find the settings in `app/config/session.php`.

The subdirectories allow configuration settings to be overridden based on the environment you're working in. For example, in most of my Laravel projects, I have the following in my **`bootstrap/start.php`** file.

```php
$env = $app->detectEnvironment(array(

    'chuck-dev' => array('babu'),

));
```

This is because the hostname of my development machine is `babu` and I have it set up to know *chuck-dev* is my environment. Now, any time I want a configuration value within my application, Laravel will first load all the settings in `app/config/filename.php` and merge in any settings if there's a `app/config/chuck-dev/filename.php`. *(Note `filename` is whatever the top level configuration value is.)*

Let's say I create two files `app/config/stupid.php` and `app/config/chuck-dev/stupid.php` with the following:

```
<?php
// app/config/stupid.php
return array(
  'test1' => false,
  'test2' => array(
    'el1' => 'one',
    'el2' => 'two',
  ),
);
?>

<?php
// app/config/chuck-dev/stupid.php
return array(
  'test1' => true,
  'test2' => array(
    'el1' => 1,
    'el3' => 3.0,
  ),
);
?>
```

Now, I could write the following code:

```
// Somewhere in your code
var_dump(Config::get('stupid.test1'));
var_dump(Config::get('stupid.test2'));
```

And expect the following output:

```
bool(true)
array(3) {
  'el1' =>
  int(1)
  'el2' =>
  string(3) "two"
  'el3' =>
  double(3)
}
```

The first line shows `true`, indicating the value from `app/config/chuck-dev/stupid.php` was picked up. The rest of the lines show data as expected when the values are merged in.

Pretty slick, huh?

What's our todo list look like

Okay, back from the Configuration tangent. My next thought, in designing this application, is what will an actual todo list look like? I want to be able to edit them by hand if needed.

I'm thinking of a simple list that looks like this.

```
# Things To Do

* Figure out what this app looks like.
* Plan out the "models"
* Change the oil in my truck
* Finish this manual
* Learn to fly without mechanical aid
```

Okay, that's a bit simplistic. I'm using Markdown[27] format here. I like Markdown because it easily can be edited, converted to html, displayed, etc.

But basically, I can see a list has two pieces of information: List Info and the Tasks.

Initial List and Task objects

In my head I've created a couple classes to store information about the lists. *(The land inside my head is a magical place where all code is perfect. And bugs ... nope. There's this little guy in there, chomping on a cigar who will tell you "We ain't got no stinkin bugs.")*

No need to create these classes, I'm just explaining my thought process.

```
class TodoList {
  protected $file;        // full path to the file
  protected $isLoaded;    // has file been loaded into this object
  protected $title;       // title of list
  protected $tasks;       // array of TodoTask objects
}

class TodoTask {
  protected $isComplete;
  protected $description;
}
```

[27] http://daringfireball.net/projects/markdown/

Note that so far I am limiting my thinking to just the data associated with the todo items and not the actual actions to be performed. There are immediate actions that jump to mind, like adding new tasks, marking tasks complete, and so forth but I wanted to step back and examine these two classes, think about them, before continuing.

Are these really the best representation of the data? Sure, there's things we know we'll probably want to add like due dates, maybe even priorities or contexts, but at this initial level how is the design looking.

Meh. Yeah, there's a whole lot of boring there. Several problems jump out at me.

1. The `TodoList` is concerned with the implementation of the storage, namely the `$file` and the `$isLoaded` values. Better to abstract those.
2. What if I want to track additional information on a list in the future? Like a subtitle, or last modified date? Maybe I should track attributes, of which '$title' is one?
3. I have commented `$tasks` as an array of TodoTask objects, but shouldn't that be a collection of interfaces?

Now, what if we review these initial classes with SOLID Object Design in mind??

- `TodoList` breaks the Single Responsibility Principle because we'd have to change the class if the storage changes or if we track additional attributes.
- `TodoList` breaks the Open/Closed Principle because we'd have to modify the class itself if we want anything other than `$title`.
- Liskov is likely ignored too (hard to tell only looking at data), but the implication from comments is that we're tracking `TodoTask` objects. How will our yet-to-be-defined methods handle replacing `TodoTask` with a subclass?
- The Interface Segregation Principle? What the ...? There aren't any interfaces.
- Dependency Inversion? Again it's tough to tell without actual code, but since there's no interfaces, we're probably going to not do well here either.

That's okay. No code has actually been written yet. It'll be easy to fix these issues in the first version.

TodoRepositoryInterface

Let's create an interface we'll use when thinking of todo list storage. We already have the perfect place to put this interface, in the 'Repositories' directory.

```php
<?php namespace GSD\Repositories;

// File: app/src/GSD/Repositories/TodoRepositoryInterface.php

use GSD\Entities\TodoListInterface;

interface TodoRepositoryInterface {

  /**
   * Load a TodoList from its id
   * @param string $id ID of the list
   * @return TodoListInterface The list
   * @throws InvalidArgumentException If $id not found
   */
  public function load($id);

  /**
   * Save a TodoList
   * @param string $id ID of the list
   * @param TodoListInterface $list The TODO List
   */
  public function save($id, TodoListInterface $list);
}
?>
```

Yea! The first file of our project. Finally, after what 60 pages or so? (Or 1000 pages if you're viewing this on an iPhone.) Still, it's only a single interface, 23 lines of code and only two methods which are only commented and not actually implemented. Heh, that's okay. We'll start creating things faster.

Chapter 21 - Thinking about the Tasks

In This Chapter

In this chapter we'll start thinking of the tasks we want to track and build the interfaces needed to track them. A discussion of Laravel Facades is also included.

TodoTaskInterface

My initial thinking about the `TodoTask` at the beginning of the last chapter was very basic.

```
1  class TodoTask {
2    protected $isComplete;
3    protected $description;
4  }
```

What do we want to track with the tasks?

- **due date** - Yes, I like this one.
- **priority** - Many people like assigning a priority to their tasks. I'm not going to do that here (because I don't roll that way), but feel free to add it in your version.
- **context** - Here's another one people sometimes use. What's the context in which you'll perform the task. Things like *@home, @work,* or *@calls*. I don't use contexts personally. I'm skipping contexts in this book.
- **next action** - Is the task something which can be worked on next? I'm adding this in my version because I like seeing a list of the next things I can immediately take action on.

With the above in mind, we can define the initial interface for a Todo Task. Create the following file.

```php
<?php namespace GSD\Entities;

// File: app/src/GSD/Entities/TodoTaskInterface.php

interface TodoTaskInterface {

  /**
   * Has the task been completed?
   * @return boolean
   */
  public function isComplete();

  /**
   * What's the description of the task
   * @return string
   */
  public function description();

  /**
   * When is the task due?
   * @return mixed Either null if no due date set, or a Carbon object.
   */
  public function dateDue();

  /**
   * When was the task completed?
   * @return mixed Either null if not complete, or a Carbon object
   */
  public function dateCompleted();

  /**
   * Is the task a Next Action?
   * @return boolean
   */
  public function isNextAction();
}
?>
```

Notice how the structure changed from the initial thinking of a `TodoTask` class to the `TodoTaskInterface`? The first one was simply data, now there's only methods. And I've only defined the *getters* at this point. That's okay. It's enough to start with.

Also, I got to thinking as I was writing the interface that it would be nice to know when a task was complete, so I threw in the `dateCompleted()` method. And, speaking of dates, I decided while writing the interface that dates would always be returned as a `null` or as a Carbon[28] date. `Carbon` is a dependency which Laravel automatically pulls in and it provides some nice extensions to the standard PHP DateTime class.

Do your own thing

As you follow along, please don't copy code exactly as I have created it. Instead, customize the todo task to what works for you. I mentioned some examples earlier (like **context** and **priority**).

Make this a project you want to use.

How about a ToDo facade

It occurs to me as I'm writing this that it may be fun to work toward a ToDo facade. Then I could do things like:

```
1  $items = Todo::list('waiting')->getAll();
2  Todo::list('next-actions')->add(Todo::item('something new'));
```

Something to think about.

TaskCollectionInterface

Now that we have a `TodoTask` defined (or the interface, at least), let's define an interface to manage a collection of tasks. This `TaskCollectionInterface` is not the `TodoList`, but the tasks within the `TodoList`.

I could use a built-in PHP type, maybe `ArrayObject` for this, but I really don't know how this is going to be used. I expect that the list of tasks will be sorted in some sort of standard format, and that I'll want to be able to add to the collection, remove tasks from the collection, and maybe iterate over the collection.

For now, Let's just create an interface with what we know is needed. (It won't surprise me if this changes as things become better defined.)

[28] https://github.com/briannesbitt/Carbon

```php
<?php namespace GSD\Entities;

// File: app/src/GSD/Entities/TaskCollectionInterface.php

interface TaskCollectionInterface {

  /**
   * Add a new task to the collection
   * @param TodoTaskInterface $task
   */
  public function add(TodoTaskInterface $task);

  /**
   * Return task based on index
   * @param integer $index 0 is first item in collection
   * @return TodoTaskInterface The Todo Task
   * @throws OutOfBoundsException If $index outside range
   */
  public function get($index);

  /**
   * Return array containing all tasks
   * @return array Array of TodoTaskInterface objects
   */
  public function getAll();

  /**
   * Remove the specified task
   * @param integer $index The task to remove
   * @throws OutOfBoundsException If $index outside range
   */
  public function remove($index);
}
?>
```

Pretty straightforward. This will probably expand with another method or two. Right now I'm thinking we may need to have some sort of `find()` method, but I'll wait until I need it to add it to the definition.

Notice the comments?

These interfaces contain more documentation than code. If I'm working in a team, then this is pretty much how things are documented. If I'm working by myself I still tend to document the non-obvious things like exceptions and return values.

The Wonderful World of Facades

I've mentioned facades a couple of times. Laravel makes heavy use of this pattern.

So what? What's so great about facades?

I'm so glad you asked. Facades allow you to write expressive code such as:

```
$user = User::find(1);
$isDebug = Config::get('app.debug');
$data = Cache::get($key);
```

So what? It looks like static classes.

You're right. They do look like static classes, but they're not. See, the facade pattern allows your code to be decoupled from the actual implementation. This is crazy cool, so let me repeat it:

Code using facades is decoupled from the implementation.

Now if that doesn't blow your mind, I don't know what will.

So what? Why would I want to do that?

Hmmm. I can see you're a skeptic here. Or maybe you just don't like the cut of my jib. I'll give you a few reasons.

Facade Reason #1 - Prettiness

Let's face it. The default way of creating objects has a lot of housework involved. You must create an object, using the **new** keyword. Then you might have to do some special setup. Finally, you have a variable containing the object and you call the methods you want to call.

All that initial setup happens behind the scenes.

When you make the call:

```
$isDebug = Config::get('app.debug');
```

You're really only concerned about getting that debug flag. But what happens behind the scenes is Laravel looks at your call and thinks: *Have I created a config object before? Nope. Then let me create one. Now I'll read the app.php file and merge in settings from the environment overrides. Finally, I'll return that value.*

In my book, `Config::get()` is just plain more elegant.

Facade Reason #2 - Testing

Testing is the most obvious reason to use facades. And Laravel's `Facade` class provides nice integration with Mockery[29] for object mocking. Consider the following:

For example:

```
public function addTwoNumbers($a, $b)
{
  return Adder::addem($a, $b);
}
function addTwoNumbers($test)
{
  Adder::shouldReceive('addem')->once()->with(1, 2)->andReturn(3);
  $result = addTwoNumbers(1, 2);
  $test->assertEquals($result, 3);
}
```

Actually, that example is a bit contrived. I need to update it. But the `shouldReceive()` method thinks *"Okay, they want to test, so I'll return a mock object instead of the standard `Adder` object."*

Isn't that cool? (At this point I'm realizing I'm using the word cool way too much. But, dang it. It is cool.)

Facade Reason #3 - API Versions

Here's a common use case for facades. Let's say your application uses the DOOGLE API V3.1. Unfortunately V3.1 is going to be retired soon. You're working on V3.2 to replace it.

Your application code using this API might be something like:

```
$user = DoogleAPI::getUser($key);
if ( ! $user)
{
  DoogleAPI::createUser($key, $name, ...);
  $user = DoogleAPI::getUser($key);
}
```

So somewhere in your code (likely in the Doogle Service Provider) there's a snippet of code that binds the facade `DoogleAPI` to `DoogleApiVer31`.

[29] https://github.com/padraic/mockery

```
1   public function register()
2   {
3     $this->app['doogle-api'] = $this->app->share(function()
4     {
5       return new DoogleApiVer31;
6     });
7   }
```

All you need to do is change the `DoogleApiV31` to your new version and hopefully no more code changes.

> Obviously, there are more than three reasons to use facades. Those are just the first three that came to mind.

Facades are powerful

Since earlier in this chapter I had the idea to implement a `Todo` facade, I wanted to give you some background on why facades make sense.

Chapter 22 - Creating the TaskListInterface

In This Chapter

In this chapter we'll code the TodoListInterface and recap the source code created so far.

TaskListInterface

We're creating another interface. Crazy huh?

Initially, I thought I'd just have a `TodoList` class, but I already defined the `TodoRepositoryInterface` to expect a `TodoListInterface`. I could go back to the `TodoRepositoryInterface` and change it to expect a `TodoList`, but let's think things through:

- A repository needs to be able to return a todo list (with the `load()` method).
- A todo list needs to save itself, or tell the repository to `save()`.

And in both cases, I want to deal with abstractions and use interfaces. This best follows the SOLID design principles.

So let's get this interface defined. Create the `app/src/GSD/Entities/TaskListInterface.php` as follows

```
1  <?php namespace GSD\Entities;
2
3  // File: app/src/GSD/Entities/TaskListInterface.php
4
5  interface TaskListInterface {
6
7    /**
8     * Loads the task list by id
9     */
10   public function load($id);
```

```
11
12      /**
13       * Save the task list
14       */
15      public function save();
16
17      /**
18       * Return the id
19       */
20      public function id();
21
22      /**
23       * Is the list dirty?
24       */
25      public function isDirty();
26
27      /**
28       * Return a list attribute
29       */
30      public function get($name);
31
32      /**
33       * Set a list attribute
34       */
35      public function set($name, $value);
36
37      /**
38       * Return the title (alias for get('title'))
39       */
40      public function title();
41
42      /**
43       * Add a new task to the collection
44       * @param TodoTaskInterface $task
45       */
46      public function addTask(TodoTaskInterface $task);
47
48      /**
49       * Return a task
50       */
51      public function getTask($index);
52
```

```
53    /**
54     * Return all tasks as an array
55     */
56    public function allTasks();
57
58    /**
59     * Remove the specified task
60     */
61    public function removeTask($index);
62  }
63  ?>
```

I threw in a `isDirty()` method, thinking that it might make sense to know if a Todo List is dirty and needs to be saved. Dunno if that'll stick around.

Criminy! This seems to be getting a bit long for an interface. Hmmm. It looks like most of these methods are actually going to be wrappers. What does it look like if I group methods by category?

Wrappers on the Repository
: the `load()` and `save()` methods could be implemented as skinny wrappers on top the repository object.

Wrappers on the Task Collection
: the `getTask()`, `allTasks()`, and `removeTask()` methods could be implemented as wrappers on the task collection object.

Methods dealing with list properties
: the rest of the methods would deal with properties of the list itself.

If I was paranoid I might be worried about the *Interface Segregation Principle*. We might be on the edge with the principle and it probably wouldn't hurt to break the interface apart into smaller interfaces. But you know what? I'm not going to do it.

This is one of those thousand small level design decisions that are made when coding. Another programmer could very well break things apart differently. I'm not too worried about the `Interface Segregation Principle` in this context.

Remember there's no one right way.

A Mini-Recap

So far we've created four interfaces.

1. **`TodoRepositoryInterface`** - This interface defines our storage.

2. **`TaskCollectionInterface`** - This interface defines a collection of tasks.
3. **`TaskListInterface`** - This interface defines our Todo List
4. **`TodoTaskInterface`** - This interface defines a single task.

Maybe I'm being a bit inconsistent on naming and the `TaskCollectionInterface` should be renamed to `TodoTaskCollectionInterface` and the `TaskListInterface` should be renamed to `TodoTaskListInterface`, but I'll let it slide (for now).

So far I think we're building a simple and solid foundation. Soon we'll get to actually using these interfaces, but first I want to think more about those text files that are going to hold our todo lists.

Chapter 23 - Format of the text file

In This Chapter

In this chapter we'll decide what the text files holding individual todo lists look like, how they're formatted, and any rules associated with the format.

File formatting rules

We want to be able to edit lists with text editors. So the file itself must be in text format.

- The first line of the file contains the title (prepended with a hash mark #).
- After the title all the tasks are listed, each task takes a single line.
- Tasks are grouped with next actions first, followed by normal tasks, followed by completed tasks.
- Blank lines are skipped when the file is read.
- Blank lines are inserted before each group of tasks when the file is written.

Example list

```
# List Title

* First Next action
* Second Next action

- First item on the list :due:2013-09-04
Next item on the list
- Last item on the list

x 2013-09-03 Finish chapter two.
x 2013-09-02 Finish chapter one
```

Rules about the tasks in a list

- Next actions begin with an `*[space]`
- Completed tasks begin with `x[space]date[space]`
- Normal list tasks begin with `-[space]` but if a line is missing that format, and doesn't match the other two formats, it is considered a normal task. *(The "Next item on the list" in the above example.)*
- Tasks are stored alphabetically within the task type (next action, normal), but completed items are stored sorted by completion date (descending).
- if a word in the task begins with `:due:` it represents the due date.

What else? The only other thing I can think of is that we'll be using the name of the file (the base filename) as the list id.

Add your own rules

Maybe you want contexts to appear within the task as `@home` or `@work`, or maybe you are using priorities and not next actions and have the top priority appear with a `1[space]` before it, second priority a `2[space]`. However you're customizing the list, make notes on the formatting.

How individual lists are sorted

Since we'll be dealing with multiple todo lists, I want to define how they're sorted when I list all my todo lists. For example I'd like to issue a command similar to:

```
$ todo lists
Your todo lists
===============
Inbox - 0 items
Actions - 4 items
Waiting - 1 item
Someday - 0 items
Calendar - 12 items

Build Spaceship - 3 items
Defeat Death - 1 item
```

So what do I have here? I've got the normal "buckets" of things at the top, and I've got them organized how I like, in the order I like. Then all of my project lists follow these standard buckets in alphabetical order.

Let's modify the configuration and add the standard buckets, knowing that in the future when I get to the command to list the lists, I'll make use of it. Modify the top of `app/config/app.php`

```
1  return array(
2
3    'gsd' => array(
4      'folder' => '/home/chuck/Documents/gsd/',
5      'extension' => '.txt',
6      'listOrder' => array(
7        'inbox', 'actions', 'waiting', 'someday', 'calendar',
8      ),
9      'defaultList' => 'actions',
10   ),
11   // rest of file the same
```

Now we have configuration data that not only lists the order of my lists, but defines the default list. I figured the default list will very likely come into play later.

List Purpose or Subtitle

It occurs to me that I may forget what a particular list is used for. So, I'm adding a rule ... *(The second line of my list's text file, if it begins and ends with parentheses, then it's a one line description of the list)*

(It's tempting for me to go back to the top of this chapter and simply revise my existing set of rules and example list to include subtitle. But I'm trying to illustrate how designing a system is an organic, non-linear process.)

Now have a list that looks something like:

```
# Actions
(These are misc actions, I can work on)

* First Next action
* Second Next action :due:2013-09-04

- Something else to do

x 2013-09-02 Finish chapter two
x 2013-09-01 Finish chapter one
```

Snazzy! I'm not exactly following the Markdown format now, but that was only a starting point anyway.

Chapter 24 - Application Functions

In This Chapter

In this chapter we'll list the functions we want our application to perform. And we'll map out what using a `Todo` facade could look like.

Time to figure out what this application is actually going to do. By the very nature of what a todo list is *(namely, a list of items, uh, to do)*, we've been able to do the planning so far, but now let's get to the nitty-gritty.

Laundry list of functions

Functions dealing with Lists

Create New List
: We want to be able to easily create new todo lists for new projects.

List Lists
: We want to list all our todo lists.

Edit List
: We want to be able to edit the attributes of a list (such as title, subtitle)

Delete List
: We want to delete lists, maybe archive them?

Functions dealing with Tasks

Add Task
: We want to be able to add tasks to a specific (or default) list

List Tasks
: We want to see all the tasks on a list. (Or even across all lists). And we'll need the ability to filter tasks to show only Next Actions, or completed.

Edit Task
: We want to be able to edit details about the task. This includes marking a task complete, toggling the "Next Action" state.

Remove Task
: We want to be able to delete a task from a list.

Move Task
: We want to be able to move tasks from one list to another

Search for Task
: We want to be able to search through the lists for a particular task.

Your Turn

How have you customized the list so far? Are there additional things you'd like to accomplish? Additional functionality? Maybe you want to show all tasks with a certain context? Or maybe move completed tasks to a monthly archive. Whatever it is, now's the time to think through what functions you want to add.

Using a Todo Facade

In an earlier chapter I mentioned it would be cool to create a Todo facade. I still think that's a pretty nifty idea.

How would this facade work? What would it look like if we already had it developed and wanted to use it?

I'll write some PHP code to perform the functions outlined above using a Todo facade. Obviously, this code won't work yet. It's only to illustrate how the Todo facade should operate.

```
1  // Create a new list
2  Todo::makeList('project-a', 'New List Title');
3
4  // Get list of all the lists
5  $lists = Todo::allLists();
6
7  // Edit list
8  Todo::get('project-a')
9      ->set('title', 'Tasks for Project A')
10     ->save();
11
12 // Delete list (I'll archive it instead)
13 Todo::get('project-a')
14     ->archive();
15
```

```
16 // Add task
17 Todo::get('project-a');
18     ->addTask('Another thing to do')
19     ->save();
20
21 // List tasks
22 $tasks = Todo::get('project-a')->allTasks();
23
24 // Edit task
25 Todo::get('project-a')
26     ->setTask(1, 'New task description')
27     ->save();
28
29 // Remove task
30 Todo::get('project-a')
31     ->removeTask(1)
32     ->save();
33
34 // Move task
35 $list1 = Todo::get('project-a');
36 $list2 = Todo::get('project-b');
37 $list2->addTask($list1->getTask(1))
38       ->save();
39 $list1->removeTask(1)
40       ->save();
```

Now wouldn't that be nifty if we could write code as stylish as that? We will.

One thing I'm noticing is the above code uses a lot of method chaining.

Method Chaining

Method chaining is a technique where methods of an object return the object when complete. This allows code to be structured like:

```
1 $obj->method1('arg')->method2('arg')->method3();
```

It's an important part of elegant coding.

I'll want to update the documentation of the interfaces to reflect the method chaining. I could do it now, but I think I'll wait until we revisit the individual methods.

Chapter 25 - Facade Planning

In This Chapter

In this chapter I'll discuss what's involved in setting up a Facade in Laravel. We'll create a few classes and do a couple unit tests.

Facade Components

There are three components to creating a facade in Laravel.

1. The Facade class shell
2. The Facade implementation
3. The setup to tie things together

Let's break these down.

The Todo Facade Class Shell

Laravel provides a base class for facades, `Illuminate\Support\Facades\Facade`. You can find this in the `vendor/laravel/framework/src/Illuminate/Support/Facades` directory, in the file `Facade.php`.

Browse the Laravel Code

Since composer installs laravel in our `vendor` directory, it's easy to browse `vendor/laravel/framework/src/` and view any of the classes. Do this! It's an excellent way to learn.

I call this a *shell* class because it is super-simple, you need only implement one method.

Let's say we're going to call our Facade Class Shell `GSD\Providers\TodoFacade.php`. *(Using the location `GSD/Providers` is a completely arbitrary decision. I know I'm going to stick some code to tie things together in this folder, so I just decided to keep everything in that folder. If this application has a lot of facades, I'd probably break things apart differently and stick this class in a `GSD/Facades` folder.)*

Since the facade class shell is so small, let's implement it now. Create the `app/src/GSD/Providers/TodoFacade.php` file as outlined below.

```
1  <?php namespace GSD\Providers;
2
3  use Illuminate\Support\Facades\Facade;
4
5  class TodoFacade extends Facade {
6
7    protected static function getFacadeAccessor()
8    {
9      return 'todo';
10   }
11 }
12 ?>
```

Yep, that's the full implementation of the class. See why I call it a shell class? The method `getFacadeAccessor()` returns the name of the implementation class. Well, that's not exactly true– it's not returning a class name, it's returning the name of the binding in the **IoC Container**. This way the parent `Facade` class knows how to make an instance of the implementation class.

Don't worry if this doesn't make perfect sense yet. It will by the time you reach the end of this chapter.

This class is so simple, I'm not doing unit testing, but from this point forward we'll start adding unit tests where it makes sense.

The Todo Facade Implementation

The next thing we need to implement is the class that actually does the work. Remember, the facade will automatically load the class bound to `todo`. This class needs to have a `makeList()` method that gets called when we type `Todo::makeList()`.

Create the following file in `app/src/GSD/Providers/TodoManager.php`. Right now I'm going to create a single, stupid method which we can use in a test when we tie things together to make sure they're tied together correctly.

```
<?php namespace GSD\Providers;

class TodoManager {

  /**
   * A stupid method for testing
   */
  public function imATeapot()
  {
    return "I'm a teapot.";
  }
}
?>
```

Testing the Todo Facade Implementation

Let's set up a simple unit test for the `TodoManager`. You should already have phpunit installed from the beginning of Chapter 19. Let's make sure it works.

```
~$ cd gsd
~/gsd$ phpunit
PHPUnit 3.7.26 by Sebastian Bergmann.

Configuration read from /home/chuck/gsd/phpunit.xml

.

Time: 0 seconds, Memory: 7.50Mb

OK (1 test, 1 assertion)
```

Laravel loves your code. It really does. That's why it wants it to work. To help you it installs `phpunit.xml` automatically, all configured and ready for you to use.

Let's create a unit test for the Todo Facade. Create the directory `app/tests/GSD/Providers` and within the directory create the file `TodoManagerTest.php` with the content below.

```
<?php
// File: app/tests/GSD/Providers/TodoManagerTest.php

use GSD\Providers\TodoManager;

class TodoManagerTest extends TestCase {

  public function testImATeapot()
  {
    $obj = new TodoManager;
    $this->assertEquals($obj->imATeapot(), "I'm a teapot.");
  }
}
?>
```

What we're doing here is creating a new `TodoManager` object and making sure it thinks it's a teapot.

Run PHP Unit again to make sure you have two tests and two assertions.

```
~$ cd gsd
~/gsd$ phpunit
PHPUnit 3.7.26 by Sebastian Bergmann.

Configuration read from /home/chuck/gsd/phpunit.xml

.

Time: 0 seconds, Memory: 8.00Mb

OK (2 test, 2 assertions)
```

I mostly use the `--tap` option of phpunit so I can see what's happening better.

```
~$ cd gsd
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoManagerTest::testImATeapot
1..2
```

Some people prefer the `--testdox` option of phpunit.

```
~$ cd gsd
~/gsd$ phpunit --testdox
PHPUnit 3.7.26 by Sebastian Bergmann.

Configuration read from /home/chuck/gsd/ch25/phpunit.xml

Example
 [x] Basic example

TodoManager
 [x] Im a teapot
```

We're ready to move on to the third step implementing a our facade.

Tying things together

The final step on using facades is to tie everything together. This can be as simple as binding the `TodoManager` to the **IoC** container. But Laravel Best Practices involves three distinct steps:

1. Creating a Service Provider to perform any bindings.
2. Configuring the application to use the Server Provider.
3. Aliasing the Facade

Step 1 - Creating a Service Provider. Create the `TodoServiceProvider.php` file in the `app/src/GSD/Providers` directory with the following content.

```php
<?php namespace GSD\Providers;

// File: app/src/GSD/Providers/TodoServiceProvider.php

use Illuminate\Support\ServiceProvider;

class TodoServiceProvider extends ServiceProvider {

  /**
   * Register the service provider
   */
  public function register()
  {
    $this->app['todo'] = $this->app->share(function()
    {
      return new TodoManager;
    });
  }
}
?>
```

What we're doing here is creating a service provider that will bind a `TodoManager` to the **IoC** container with a key named `todo`.

To Defer or Not To Defer. That is the question.

Whether 'tis nobler in mind to suffer, uh. Okay. When the question arises whether to defer service provider loading or not, ask yourself if the services provided are needed on every (or most) requests. If so, don't defer them. That's what we did above. If you'd like to defer loading the services to only when needed, then set the `$defer` flag and implement the `provides()` method of your provider.

Careful of register()

(Wow. Two tips in a row ... have we reached some sort of tipping point? Groan. Hey. I can groan at my own puns. I do it all the time.) Never call services within the `register()` which are provided by another service provider. If this is needed, do it in a `boot()` method of the provider instead.

Now we have a simple little `TodoServiceProvider`, let's make the app use it.

Step 2 - Configure the app to use the `TodoServiceProvider`. At the end of `providers[]` array in `app/config/app.php`, add the following line.

```
'providers' => array(

  // end of array
  'GSD\Providers\TodoServiceProvider',
),
```

Easy as pie, the next step is just as easy.

Step 3 - Alias the facade. In the same file (`app/config/app.php`) add the following line to the bottom of the `aliases[]` array.

```
'aliases' => array(

  // end of array
  'Todo' => 'GSD\Providers\TodoFacade',
),
```

Then run `composer dump-auto`

```
$ composer dump-auto
```

At this point, theoretically, we can use `Todo::imATeapot()` in our code. So, why not test it?

Testing the Todo Facade

Normally, I like to structure unit tests using same directory hierarchy as the source code. This makes it easy to know where your tests are. Plus you can do some other tricks like automatic testing when your source code changes. *(See Jeffery Way's Laravel Testing Decoded[30] for details.)*

I suppose we could go in an create test files for the `TodoServiceProvider`, but I'm lazy and am just going to add another test to our existing `TodoManagerTest.php`. Really, all I'm worried about is that things are set up and tied together correctly for using the facade.

Add the following method to `tests/GSD/Providers/TodoManagerTest.php`

[30] http://leanpub.com/laravel-testing-decoded

```
  public function testFacade()
  {
    $this->assertEquals(Todo::imATeapot(), "I'm a teapot.");
  }
```

And now, let's test it (fingers crossed)

```
~/gsd $ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoManagerTest::testImATeapot
ok 3 - TodoManagerTest::testFacade
1..3
```

WOO HOO! Works first time. *(Well, first time after I fixed a typo you'll never know about.)*

Chapter 26 - Midstream Refactoring

In This Chapter

In this chapter we're going to change some class and filenames for consistency.

Remember a few chapters back when I noted some of my naming was inconsistent? In places I used `TaskListInterface` and then I used `TodoTaskInterface`. Why not `TodoTaskListInterface`?

Well, I started thinking, *"Why even have the word 'todo' in class and interface names? I mean, this whole stinkin application is a todo manager."*

How about the source organization? The `Entities` directory is fine, I've got interfaces for the collection, the list, and the task. That seems to make sense. The only other directory in which there's source code is the `Providers` directory. It contains the provider, the todo facade, and the todo manager. I'm not going to change anything there yet, but part of me wants to move the facade and manager out of there. If you want to, then have at it.

Now's a good point to clean this up because we don't have lots of code yet.

It's Messy

If I was simply writing this application, I'd just go back, quickly refactor, change names, etc., and be done. But part of my goal with this book is to illustrate how messy the process of design and development can be. It's much more work for me to explain these things, and why to do them, than it is to go back and simply fix them.

But I'm keeping the process real. :)

In the foggy woods

This is how coding often is for me. I'll start a project and it's like I'm in my car at night, traveling through the woods. It's foggy outside. I only see the woods in front of me illuminated by my headlights. There are a few crystal clear sections of the woods lit by giant spotlights shining down from the sky, but for the most part I'm traveling into the unknown, trying to make my way to one of those brightly lit places.

Luckily, my car is equipped with magic headlights and once I light up part of the woods, it stays lit. Very often I make my way to a clear place and discover there was a shortcut I missed, or a different route that was paved when I just stumbled along a gravel road. So I'll back up and speed through the new and better way. In other words, *refactoring*.

(I'm not saying this is how it is for everyone, remember "Every Programmer is Different". This is how it is for me.)

TaskListInterface

The first file I'm picking is the `TaskListInterface`. I'm starting here because it has all those methods defined and that was kind of bugging me.

Here's a list of the changes I made:

- Renamed file to `ListInterface.php` cause that's cleaner ... it's a todo list, and we already know everything is todo related in this project.
- Renamed the interface to `ListInterface` too.
- Renamed `addTask()` method to `taskAdd()`
- Renamed `getTask()` method to just plain `task()`, allowing me to add the `taskGet()` for returning a tasks attribute at the list level.
- Renamed `allTasks()` method to `tasks()`
- Renamed `removeTask()` method to `taskRemove()`
- I reorganized the methods, organizing them by category

The reason for those task renames is because now it's very apparent that those methods are doing something with tasks, not the list itself. I want to keep the method name reminding me of this fact.

I also threw in `taskCount()`, `taskGet()`, `taskSet()` and `isArchived()` and fleshed out the documentation.

The new version is below.

```
1  <?php namespace GSD\Entities;
2
3  interface ListInterface {
4
5    // List attributes ------------------------
6
7    /**
8     * Return the id
9     */
10   public function id();
```

```
11
12      /**
13       * Is the list archived?
14       * @return bool
15       */
16      public function isArchived();
17
18      /**
19       * Is the list dirty?
20       * @return bool
21       */
22      public function isDirty();
23
24      /**
25       * Return a list attribute
26       * @param string $name id|isArchived|isDirty|title
27       * @return mixed
28       * @throws InvalidArgumentException If $name is invalid
29       */
30      public function get($name);
31
32      /**
33       * Set a list attribute
34       * @param string $name id|isArchived|isDirty|title
35       * @param mixed $value Attribute value
36       * @return ListInterface for method chaining
37       * @throws InvalidArgumentException If $name is invalid
38       */
39      public function set($name, $value);
40
41      /**
42       * Return the title (alias for get('title'))
43       */
44      public function title();
45
46      // List operations -------------------------
47
48      /**
49       * Archive the list. This makes the list only available from the archive.
50       * @return ListInterface For method chaining
51       */
52      public function archive();
```

```
53
54      /**
55       * Loads the task list by id
56       * @param string $id The id (name) of the list
57       * @return ListInterface for method chaining
58       * @throws InvalidArgumentException If $id not found
59       */
60      public function load($id);
61
62      /**
63       * Save the task list
64       * @return ListInterface for method chaining
65       */
66      public function save();
67
68      // Task operations -------------------------
69
70      /**
71       * Add a new task to the collection
72       * @param string|TaskInterface $task Either a TaskInterface or a string we
73       *                                   can construct one from.
74       * @return ListInterface for method chaining
75       */
76      public function taskAdd($task);
77
78      /**
79       * Return number of tasks
80       * @return integer
81       */
82      public function taskCount();
83
84      /**
85       * Return a task
86       * @param integer $index Task index #
87       * @return TaskInterface
88       * @throws OutOfBoundsException If $index outside range
89       */
90      public function task($index);
91
92      /**
93       * Return a task attribute
94       * @param integer $index Task index #
```

```
95      * @param string $name Attribute name
96      * @return mixed
97      * @throws OutOfBoundsException If $index outside range
98      */
99     public function taskGet($index, $name);
100
101    /**
102     * Return all tasks as an array.
103     * @return array All the TaskInterface objects
104     */
105    public function tasks();
106
107    /**
108     * Set a task attribute
109     * @param integer $index Task index #
110     * @param string $name Attribute name
111     * @param mixed $value Attribute value
112     * @return ListInterface for method chaining
113     * @throws OutOfBoundsException If $index outside range
114     */
115    public function taskSet($index, $name, $value);
116
117
118    /**
119     * Remove the specified task
120     * @throws OutOfBoundsException If $index outside range
121     */
122    public function taskRemove($index);
123 }
124 ?>
```

That was a pretty good chunk of restructuring, but I'm happy with it.

You may have noticed I changed the argument to the `taskAdd()` method. This is because I started thinking about how we're going to add tasks to the system. I like the idea of adding a task from an object already built, the `TaskInterface`, but I'll probably also want to add a task just from the string. *(How to do that is a question I'm putting off as long as I can.)*

With PHP we can have the argument work both ways.

Types that Quack

We've had type hinting in PHP since version 5. This means the classname (or interface) can be specified in the arguments to functions/methods and PHP will puke if the wrong type variables is passed. Before this, all parameters were *Duck Type*. (If it looks like a duck and quacks like a it's a duck.)

Generally, it's good to strongly type methods but occasionally, as in the `taskAdd()` method, Duck Typing is a good thing.

TaskCollectionInterface

This interface isn't too bad. Just rename the `TodoTaskInterface` argument in the `add()` method with the new `TaskInterface`. Also, change any documentation to match.

I'm not going to bother presenting this class to you.

TodoTaskInterface

- Rename the file to `TaskInterface.php`
- Rename the interface from `TodoTaskInterface` to `TaskInterface`

Currently there's five *getter* methods. (In my version anyway, you may have different ones if you decided to track contexts, or priorities, or pygmies, or whatever)

Getter methods:

- `isComplete()` - is the task finished?
- `description()` - description of task
- `dateDue()` - due date (null or Carbon)
- `dateCompleted()` - date completed (null or Carbon)
- `isNextAction()` - is this a next action.

Since I'm in a refactoring mood, I'm adding four setter methods.

- `setIsComplete($complete)` - set from boolean
- `setDescription($description)` - set from string
- `setDateDue($date)` - set to null, or from string or Carbon
- `setIsNextAction($nextAction)` - set from boolean

Notice I skipped `setDateCompleted()`. That should automatically be set if `setIsCompleted()` is called.

And finally I'm adding a generic `get()` and `set()` method.

- `set($name, $value)` - set a property
- `get($name)` - get a property

Oh man, am I rambling on this one? Here's some stylistic notes I have about this class:

- The decision to use explicit getters/settings (ie. dueDate() or setDescription()) is because this is an interface and I want the "data" to be somewhat visible (if that makes sense).
- I also want explicit getters/settings because there may be different actions occurring when values are set (such as the `isComplete()` method). If I used a generic `set()` method with a big switch statement implementing these attribute specific actions, it's harder to see. This becomes important when maintaining the code.
- I added the `set()`/`get()` methods because that will play nicely with the next interface to be refactored.

Here's the final version (to this point) of my refactored `TaskInterface`

```
1  <?php namespace GSD\Entities;
2
3  interface TaskInterface {
4
5    /**
6     * Has the task been completed?
7     * @return boolean
8     */
9    public function isComplete();
10
11   /**
12    * What's the description of the task
13    * @return string
14    */
15   public function description();
16
17   /**
18    * When is the task due?
19    * @return mixed Either null if no due date set, or a Carbon object.
20    */
21   public function dateDue();
```

```
22
23      /**
24       * When was the task completed?
25       * @return mixed Either null if not complete, or a Carbon object
26       */
27      public function dateCompleted();
28
29      /**
30       * Is the task a Next Action?
31       * @return boolean
32       */
33      public function isNextAction();
34
35      /**
36       * Set whether task is complete. Automatically updates dateCompleted.
37       * @param bool $complete
38       */
39      public function setIsComplete($complete);
40
41      /**
42       * Set task description
43       * @param string $description
44       */
45      public function setDescription($description);
46
47      /**
48       * Set date due
49       * @param null|string|Carbon $date null to clear, otherwise stores Carbon
50       *        date internally.
51       */
52      public function setDateDue($date);
53
54      /**
55       * Set whether task is a next action
56       * @param bool $nextAction
57       */
58      public function setIsNextAction($nextAction);
59
60      /**
61       * Set a property. (Ends up calling specific setter)
62       * @param string $name isComplete|description|dateDue|isNextAction
63       * @param mixed $value The value to set
```

```
64      * @throws InvalidArgumentException If $name is invalid
65      */
66     public function set($name, $value);
67 
68     /**
69      * Get a property.
70      * @param string $name isComplete|description|dateDue|isNextAction|dateCompleted
71      * @return mixed
72      * @throws InvalidArgumentException If $name is invalid
73      */
74     public function get($name);
75 }
76 ?>
```

Finishing Up and Testing

The other source files (those three in the `Providers` folder) shouldn't be affected.

It's a good idea to run unit tests when you do something like this. There's no unit test we currently have that should be affected, but still ... I like to do it because it gives me a warm and fuzzy feeling.

```
1 ~/gsd $ phpunit --tap
2 TAP version 13
3 ok 1 - ExampleTest::testBasicExample
4 ok 2 - TodoManagerTest::testImATeapot
5 ok 3 - TodoManagerTest::testFacade
6 1..3
```

Heh, heh. Looks good. Feeling warm and fuzzy?

Chapter 27 - Starting the TodoManager class

In This Chapter

In this chapter we'll start coding `TodoManager` and testing it along the way.

Looking back on Chapter 24, where I wrote the example Todo facade code, I realize there's only three methods to the facade.

1. `makeList()` which creates a new list.
2. `get()` which returns a `ListInterface`.
3. `allLists()` which returns an array of all the lists.

So let's code these.

TodoManager::makeList()

The goal is to be able to write the following code:

```
1  Todo::makeList('project-a', 'New List Title');
```

Since we know that `Todo` is aliased to `TodoFacade` and the `TodoFacade` will create an instance of `TodoManager`, we know we want `makeList()` to be a method of `TodoManager`.

Edit the `TodoManager.php` file in `app/src/GSD/Providers` so it looks like the code below.

```php
<?php namespace GSD\Providers;

use App;

class TodoManager {

  /**
   * A stupid method for testing
   */
  public function imATeapot()
  {
    return "I'm a teapot.";
  }

  /**
   * Create a new TODO List
   * @param string $id The basename of the list
   * @param string $title The title of the list
   * @return ListInterface The newly created list
   * @throws InvalidArgumentException If the list already exists
   */
  public function makeList($id, $title)
  {
    $repository = App::make('TodoRepositoryInterface');
    if ($repository->exists($id))
    {
      throw new \InvalidArgumentException("A list with id=$id already exists");
    }
    $list = App::make('ListInterface');
    $list->set('id', $id)
         ->set('title', $title)
         ->save();
    return $list;
  }
}
?>
```

Pretty straightforward. Let's think about it.

It doesn't make sense to create *new* 'archived' lists since before a list becomes archived it must first exist as an active list. So I won't worry about the whole archive thang. And I decided most of these methods should just throw exceptions if there are problems. That makes using the methods easier. (I don't have to check the return of every method called. Instead, I can just assume methods return

correctly and catch the exceptions as, uh, exceptions.)

I'm using `App::make()` to create new instances of both the repository and an actual list. Should I move either of these to the constructor? Naw, there's only three methods planned. Maybe I'll refactor this later. I don't know. I actually haven't bound these interfaces to a concrete class yet. That's okay, because I plan on testing it with mock objects.

Hmm, I used an `exists()` method of the `TodoRepositoryInterface` which wasn't defined, so let's fix that.

Update the `TodoRepositoryInterface.php` file to match what's below. (Dang, I didn't rename this during the refactoring last chapter. Oh well. I think I actually like it with the Todo. You may think I'm contradicting myself. Pbbth! I like to call it being *complex*.)

```
1  <?php namespace GSD\Repositories;
2
3  // File: app/src/GSD/Repositories/TodoRepositoryInterface.php
4
5  use GSD\Entities\ListInterface;
6
7  interface TodoRepositoryInterface {
8
9    /**
10    * Does the todo list exist?
11    * @param  string $id ID of the list
12    * @return boolean
13    */
14   public function exists($id);
15
16   /**
17    * Load a TodoList from it's id
18    * @param string $id ID of the list
19    * @return ListInterface The list
20    * @throws InvalidArgumentException If $id not found
21    */
22   public function load($id);
23
24   /**
25    * Save a TodoList
26    * @param string $id ID of the list
27    * @param ListInterface $list The TODO List
28    */
29   public function save($id, ListInterface $list);
```

```
}
?>
```

(Oops, I found a few straggling TodoListInterface in the above that I didn't catch during the refactoring and fixed them.)

Now, let's test this thing.

Installing Mockery

To test the `TodoManager::makeList()` method, we'll want to create mock objects for the `TodoRepositoryInterfa` and `ListInterface` that `makeList()` creates. Keep in mind these are still interfaces and we have not yet developed the concrete classes which implement the interface.

That's okay. We can use the excellent Mockery[31] package to create objects for the interface and fake up the methods we want to hit.

First step to install Mockery is to edit the `composer.json` to require the dependency. Edit the file as specified below.

```
{
  "name": "gsd",
  "description": "Getting Stuff Done with Laravel.",
  "require": {
    "laravel/framework": "4.0.*"
  },
  "require-dev": {
    "mockery/mockery": "dev-master"
  },
  // everything else is the same
```

Next run `composer update` to install the dependency.

[31] https://github.com/padraic/mockery

```
~$ cd gsd
~/gsd$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing mockery/mockery (dev-master b6fe71c)
    Loading from cache

Generating autoload files
Generating optimized class loader
```

That's it, package installed. Ain't composer great?

Testing TodoManager::makelist()

Now, we'll update the `TodoManagerTest.php` to test that `makeList()` throws an exception as expected when the list id exists.

Add the following method to `tests/GSD/Providers/TodoManagerTest.php` file, making it a new method within the `TodoManagerTest` class.

```
  /**
   * @expectedException InvalidArgumentException
   */
  public function testMakeListThrowsExceptionWhenExists()
  {
    // Mock the repository
    App::bind('TodoRepositoryInterface', function()
    {
      $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
      $mock->shouldReceive('exists')
           ->once()
           ->andReturn(true);
      return $mock;
    });

    // Should throw an error
    Todo::makeList('abc', 'test abc');
  }
```

What we're doing here is first using the *docblock* to say this test should throw an error. Then, within the test we'll bind the `TodoRepositoryInterface` to return a mock object. The object is expected to receive a call to `exists()` and when it does, return `true`.

Run phpunit to test it

```
~$ cd gsd
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoManagerTest::testImATeapot
ok 3 - TodoManagerTest::testFacade
ok 4 - TodoManagerTest::testMakeListThrowsExceptionWhenExists
1..4
```

Perfect, now let's test what should happen it the list doesn't exist. This time we'll need to create another object, the `ListInterface` and mock up some expected calls.

Add the following method to `tests/GSD/Providers/TodoManagerTest.php` file, making it a new method within the `TodoManagerTest` class.

```
  public function testMakeList()
  {
    // Mock the repository
    App::bind('TodoRepositoryInterface', function()
    {
      $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
      $mock->shouldReceive('exists')
           ->once()
           ->andReturn(false);
      return $mock;
    });

    // Mock the list object
    App::bind('ListInterface', function()
    {
      $mock = Mockery::mock('GSD\Entities\ListInterface');
      $mock->shouldReceive('set')->twice()->andReturn($mock, $mock);
```

```
18        $mock->shouldReceive('save')->once()->andReturn($mock);
19        return $mock;
20      });
21
22      $list = Todo::makeList('abc', 'test abc');
23      $this->assertInstanceOf('GSD\Entities\ListInterface', $list);
24    }
```

Run phpunit to test it

```
~$ cd gsd
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoManagerTest::testImATeapot
ok 3 - TodoManagerTest::testFacade
ok 4 - TodoManagerTest::testMakeListThrowsExceptionWhenExists
ok 5 - TodoManagerTest::testMakeList
1..5
```

Yes! *(You can't see this, but right now I'm pumping my fist in the air.)* Our `TodoManager::makeList()` method has been tested.

Chapter 28 - Finishing the TodoManager class

In This Chapter

In this chapter we'll finish coding and testing `TodoManager`.

Creating TodoManager::allLists()

Back when we planned out the facade, we wanted to be able to do something like:

```
1 // Get list of all the lists
2 $lists = Todo::allLists();
```

That wasn't very well defined though. Should we get an array of `ListInterface` objects or just an array of list ids? My vote is to just return the list of ids (which, remember, is the basename of the todo list file).

Add the following method to the file `app/GSD/Providers/TodoManager.php`, in the `TodoManager` class.

```
1   /**
2    * Return a list of all lists
3    * @param boolean $archived Return archived lists?
4    * @return array of list ids
5    */
6   public function allLists($archived = false)
7   {
8     $repository = App::make('TodoRepositoryInterface');
9     return $repository->getAll($archived);
10  }
```

Pretty simple. We're just wrapping the repository code. You'll notice that I added an optional `$archived` flag. What if we want to get a list of all the lists that are archived?

You probably figured out the `getAll()` method doesn't yet exist in the `TodoRepositoryInterface`. Let's fix that and at the same time add `$archive` flags to the methods that need them.

Update your `TodoRepositoryInterface.php` to match what's below.

```
1  <?php namespace GSD\Repositories;
2
3  use GSD\Entities\ListInterface;
4
5  interface TodoRepositoryInterface {
6
7    /**
8     * Does the todo list exist?
9     * @param  string $id ID of the list
10    * @param  boolean $archived Check for archived lists only?
11    * @return boolean
12    */
13   public function exists($id, $archived = false);
14
15   /**
16    * Return the ids of all the lists
17    * @param  boolean $archived Return archived ids or unarchived?
18    * @return array of list ids
19    */
20   public function getAll($archived = false);
21
22   /**
23    * Load a TodoList from it's id
24    * @param string $id ID of the list
25    * @param boolean $archived Load an archived list?
26    * @return ListInterface The TODO list
27    * @throws InvalidArgumentException If $id not found
28    */
29   public function load($id, $archived = false);
30
31   /**
32    * Save a TodoList
33    * @param string $id ID of the list
```

```
34      * @param ListInterface $list The TODO List
35      * @param boolean $archived Save an archived list?
36      */
37     public function save($id, ListInterface $list, $archived = false);
38 }
39 ?>
```

Nothing drastic changed, but we did add the `$archived` parameter to every method. The only place so far we're using the `TodoRepositoryInterface` is the `TodoManager::makeList()` method, but since we're providing a default parameter of `false`, there's no changes needed there.

So let's figure out how to test this thing.

Testing TodoManager::allLists()

Add the method below to your `tests/GSD/Providers/TodoManagerTest.php`.

```
1   public function testAllListsReturnsArray()
2   {
3     // Mock the repository
4     App::bind('TodoRepositoryInterface', function()
5     {
6       $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
7       $mock->shouldReceive('getAll')
8            ->once()
9            ->andReturn(array());
10      return $mock;
11    });
12
13    $result = Todo::allLists();
14    $this->assertInternalType('array', $result);
15
16  }
```

Run phpunit to test it

```
~$ cd gsd
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoManagerTest::testImATeapot
ok 3 - TodoManagerTest::testFacade
ok 4 - TodoManagerTest::testMakeListThrowsExceptionWhenExists
ok 5 - TodoManagerTest::testMakeList
ok 6 - TodoManagerTest::testAllListsReturnsArray
1..6
```

Seems almost silly to test this, doesn't it. I can't tell you how many times I've had super simple unit tests like this fail later in a project because I decided to change the internal logic of a method. As soon as you do that, it forces it to rethink how you want to test the method. That said, I probably wouldn't normally test this particular method since it just wraps the repository object. I want this chapter to really hammer home testing, though, so I'm going a bit overboard with my testing.

To be totally consistent, let's add another test for archived lists. This is exactly the same except we're calling `Todo::allLists()` with a `true` argument.

Add the method below to your `tests/GSD/Providers/TodoManagerTest.php`.

```
  public function testAllArchivedListsReturnsArray()
  {
    // Mock the repository
    App::bind('TodoRepositoryInterface', function()
    {
      $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
      $mock->shouldReceive('getAll')
           ->once()
           ->andReturn(array());
      return $mock;
    });

    $result = Todo::allLists(true);
    $this->assertInternalType('array', $result);

  }
```

Run phpunit to test it

```
~$ cd gsd
~/gsd$ phpunit --testdox
PHPUnit 3.7.26 by Sebastian Bergmann.

Configuration read from /home/chuck/gsd/ch28/phpunit.xml

Example
 [x] Basic example

TodoManager
 [x] Im a teapot
 [x] Facade
 [x] Make list throws exception when exists
 [x] Make list
 [x] All lists returns array
 [x] All archived lists returns array
```

Heh, I ran phpunit differently ... just to mix things up.

Creating TodoManager::get()

Only one more function to add to `TodoManager`.

First add the method to `app/src/GSD/Providers/TodoManager.php`

```
  /**
   * Get the list specified
   * @param string $id The list id
   * @param boolean $archived Return archived lists?
   * @return ListInterface
   * @throws RuntimeException If list is not found.
   */
  public function get($id, $archived = false)
  {
    $repository = App::make('TodoRepositoryInterface');
    if ( ! $repository->exists($id, $archived))
    {
        throw new \RuntimeException("List id=$id not found");
    }
```

```
15      return $repository->load($id, $archived);
16    }
```

Next add a couple test methods. One to test for the exception, the other to test without an exception. Remember this is in the file `TodoManagerTest.php` in the `app/tests/GSD/Providers` directory.

```
1     /**
2      * @expectedException RuntimeException
3      */
4     public function testGetListThrowsExceptionWhenMissing()
5     {
6       // Mock the repository
7       App::bind('TodoRepositoryInterface', function()
8       {
9         $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
10        $mock->shouldReceive('exists')
11             ->once()
12             ->andReturn(false);
13        return $mock;
14      });
15
16      // Should throw an error
17      $list = Todo::get('abc');
18    }
19
20    public function testGetListReturnsCorrectType()
21    {
22      // Mock the repository
23      App::bind('TodoRepositoryInterface', function()
24      {
25        $list = Mockery::mock('GSD\Entities\ListInterface');
26        $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
27        $mock->shouldReceive('exists')->once()->andReturn(true);
28        $mock->shouldReceive('load')->once()->andReturn($list);
29        return $mock;
30      });
31
32      $list = Todo::get('abc');
33      $this->assertInstanceOf('GSD\Entities\ListInterface', $list);
34    }
```

Run phpunit to test it

```
~$ cd gsd
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoManagerTest::testImATeapot
ok 3 - TodoManagerTest::testFacade
ok 4 - TodoManagerTest::testMakeListThrowsExceptionWhenExists
ok 5 - TodoManagerTest::testMakeList
ok 6 - TodoManagerTest::testAllListsReturnsArray
ok 7 - TodoManagerTest::testAllArchivedListsReturnsArray
ok 8 - TodoManagerTest::testGetListThrowsExceptionWhenMissing
ok 9 - TodoManagerTest::testGetListReturnsCorrectType
1..9
```

All done with the `TodoManager`. And we've created unit tests to make sure it keeps working. Now we can go on our merry way and forget the details, confident that it'll all work as expected.

Chapter 29 - Implementing ListInterface

In This Chapter

In this chapter we'll start implementing the `ListInterface`

We still haven't finished the `Todo` facade we laid out back in Chapter 24. We've wrapped up the `TodoManager` class, and now, since most of the `TodoManager` methods return `ListInterface` I figured it'd be a good time to start creating a concrete class behind it.

Creating a TodoList shell

First of all, what are we going to call the concrete class that will implement the `ListInterface`. I think a class called `List` is just too vague. For a few moments I entertained calling it `TheList`, but that makes it sound like super-important (as apposed to `AList` which doesn't sound important at all.)

Finally, I'm settling on calling it `TodoList`. Yeah, I know. I stripped out the word `Todo` from a bunch of things a few chapters back. Oh well.

Let's create `TodoList.php` in the `app/src/GSD/Entities` directory with the following content.

```
1  <?php namespace GSD\Entities;
2
3  // File: app/src/GSD/Entities/TodoList.php
4
5  class TodoList implements ListInterface {
6
7      // List attributes -------------------------
8
9      // List operations -------------------------
10
11     // Task operations -------------------------
```

```
12
13      // Not yet implemented ---------------------
14
15      public function id()
16      {
17          throw new \Exception('not implemented');
18      }
19
20      public function isArchived()
21      {
22          throw new \Exception('not implemented');
23      }
24
25      public function isDirty()
26      {
27          throw new \Exception('not implemented');
28      }
29
30      public function get($name)
31      {
32          throw new \Exception('not implemented');
33      }
34
35      public function set($name, $value)
36      {
37          throw new \Exception('not implemented');
38      }
39
40      public function title()
41      {
42          throw new \Exception('not implemented');
43      }
44
45      public function archive()
46      {
47          throw new \Exception('not implemented');
48      }
49
50      public function load($id)
51      {
52          throw new \Exception('not implemented');
53      }
```

```
54
55      public function save()
56      {
57          throw new \Exception('not implemented');
58      }
59
60      public function taskAdd($task)
61      {
62          throw new \Exception('not implemented');
63      }
64
65      public function taskCount()
66      {
67          throw new \Exception('not implemented');
68      }
69
70      public function task($index)
71      {
72          throw new \Exception('not implemented');
73      }
74
75      public function taskGet($index, $name)
76      {
77          throw new \Exception('not implemented');
78      }
79
80      public function tasks()
81      {
82          throw new \Exception('not implemented');
83      }
84
85      public function taskSet($index, $name, $value)
86      {
87          throw new \Exception('not implemented');
88      }
89
90      public function taskRemove($index)
91      {
92          throw new \Exception('not implemented');
93      }
94  }
95  ?>
```

Yeah, that class is a whole lot of ugly. I took every method from the `ListInterface` and implemented it with a "not implemented" exception. I kept the comments on the method categories (List attributes, List operations, etc.) at the top with the plan to move each method up to it's correct category when implemented.

Binding the ListInterface

Now that we have a concrete class that implements the `ListInterface`, let's bind the interface to the class. The logical place to do this is in our service provider.

Update `TodoServiceProvider.php` in `app/src/GSD/Providers` to match the following code.

```
1  <?php namespace GSD\Providers;
2
3  // File: app/src/GSD/Providers/TodoServiceProvider.php
4
5  use Illuminate\Support\ServiceProvider;
6
7  class TodoServiceProvider extends ServiceProvider {
8
9    /**
10    * Register the service provider
11    */
12   public function register()
13   {
14     $this->app['todo'] = $this->app->share(function()
15     {
16       return new TodoManager;
17     });
18
19     $this->app->bind('GSD\Entities\ListInterface', 'GSD\Entities\TodoList');
20   }
21 }
22 ?>
```

You may be asking yourself, what's the difference between `app->share()` and `app->bind()` in the above code? The difference is that `app->share()` implements the binding as a singleton. Man, Laravel thinks of everything, doesn't it?

To give us that warm and fuzzy feeling, let's write a unit test to make sure the `ListInterface` is automagically bound to the `TodoList`.

Create the `app/tests/GSD/Entities` directory and create the file `TodoListTest.php` in that directory with the following code.

```
<?php
// File: app/tests/GSD/Entities/TodoListTest.php

class TodoListTest extends TestCase {

  public function testBoundToInterface()
  {
    $obj = App::make('GSD\Entities\ListInterface');
    $this->assertInstanceOf('GSD\Entities\TodoList', $obj);
  }
}
?>
```

The test creates a new `ListInterface` object and then makes sure that object is an instance of the `TodoList`.

Think it will work? Really? Have I steered you wrong yet? I mean besides all that getting rid of **Todo** in class names, then deciding to use **Todo** again.

Let's find out.

```
~$ cd gsd
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoListTest::testBoundToInterface
ok 3 - TodoManagerTest::testImATeapot
ok 4 - TodoManagerTest::testFacade
ok 5 - TodoManagerTest::testMakeListThrowsExceptionWhenExists
ok 6 - TodoManagerTest::testMakeList
ok 7 - TodoManagerTest::testAllListsReturnsArray
ok 8 - TodoManagerTest::testAllArchivedListsReturnsArray
ok 9 - TodoManagerTest::testGetListThrowsExceptionWhenMissing
ok 10 - TodoManagerTest::testGetListReturnsCorrectType
1..10
```

Hah! Test #2 shows it working.

The TodoList::__construct()

`TodoList` is going to need three other things to operate:

1. An object implementing `TodoRepositoryInterface`
2. An object implementing `TaskCollectionInterface`
3. An array to store the list's attributes.

Let's create a constructor to set that up.

Make the top of `TodoList.php` look like the following

```
<?php namespace GSD\Entities;

// File: app/src/GSD/Entities/TodoList.php

use GSD\Repositories\TodoRepositoryInterface;

class TodoList implements ListInterface {

  protected $repository;
  protected $tasks;
  protected $attributes;

  /**
   * Inject the dependencies during construction
   * @param TodoRepositoryInterface $repo The repository
   * @param TaskCollectionInterface $collection The task collection
   */
  public function __construct(TodoRepositoryInterface $repo,
    TaskCollectionInterface $collection)
  {
    $this->repository = $repo;
    $this->tasks = $collection;
    $this->attributes = array();
  }
```

```
25
26    // rest of the file as is
27  ?>
```

Nice, we're injecting dependencies in the constructor. Now, what do you think would happen if you ran phpunit?

Go ahead and do it. I'll wait.

(Theme of Jeopardy plays ...)

You got errors. Heh. Know why?

Depending on how you ran phpunit, you might be able to figure it out. What's going on is Laravel realizes the `GSD\Entities\ListInterface` interface is now bound to `GSD\Entities\TodoList`, so when the `testBoundToInterface()` test runs it tries to create an instance of the `TodoList`.

Earlier, this worked, but now we added a couple interfaces to the `TodoList::__construct()`. Laravel tries. It really does. It tries to create default parameters for the `__construct()` method, but dang. You cannot instantiate an interface.

Thus, the error.

Easy fix. Edit the `TodoListTest.php` file and add the following method at the top.

```
1   public function setup()
2   {
3     parent::setup();
4     App::bind('GSD\Repositories\TodoRepositoryInterface', function()
5     {
6       return Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
7     });
8     App::bind('GSD\Entities\TaskCollectionInterface', function()
9     {
10      return Mockery::mock('GSD\Entities\TaskCollectionInterface');
11    });
12  }
```

Now, before the `testBoundToInterface()` method fires, this `setup()` method will bind mock objects to the interfaces.

If you run phpunit again, it all will be well with the world again.

Interface Binding is Powerful

'nuff said.

Implementing TodoList::save()

Looking back over the example facade usage at the end of Chapter 24 the `save()` method is used in most of the examples. So let's get that one out of the way.

> ## Every Program is Different
>
> I went on a bit in an earlier section how every programmer is different, but it's also worth noting every program is different. I'm not talking about what the program does, because duh, but how it is developed. For instance I had no clue how much the facade planning of Chapter 24 would drive the development of this app. On another application, it could be the User Interface that drives the development. It's always different.

Update `TodoList.php` in the `app/src/Entities` directory to match what's below.

```
1  <?php namespace GSD\Entities;
2
3  // File: app/src/GSD/Entities/TodoList.php
4
5  use GSD\Repositories\TodoRepositoryInterface;
6
7  class TodoList implements ListInterface {
8
9    protected $repository;
10   protected $tasks;
11   protected $attributes;
12   protected $isDirty;
13
14   /**
15    * Inject the dependencies during construction
16    * @param TodoRepositoryInterface $repo The repository
17    * @param TaskCollectionInterface $collection The task collection
18    */
19   public function __construct(TodoRepositoryInterface $repo,
20     TaskCollectionInterface $collection)
21   {
22     $this->repository = $repo;
```

```
23      $this->tasks = $collection;
24      $this->attributes = array();
25      $this->isDirty = false;
26    }
27
28    // List attributes -------------------------
29
30    // List operations -------------------------
31
32    /**
33     * Save the list
34     * @return $this For method chaining
35     * @throws RuntimeException If cannot save.
36     */
37    public function save()
38    {
39      if ($this->isDirty)
40      {
41        $archived = !empty($this->attributes['archived']);
42        if ( ! array_key_exists('id', $this->attributes))
43        {
44          throw new \RuntimeException("Cannot save if id not set");
45        }
46        $id = $this->attributes['id'];
47        if ( ! $this->repository->save($id, $this, $archived))
48        {
49          throw new \RuntimeException("Repository could not save");
50        }
51        $this->isDirty = false;
52      }
53      return $this;
54    }
55
56    // Rest of the file is the same (except delete the other save() method)
57  ?>
```

To implement the save, I deleted the `save()` method further down in the code and put it under the "List operations" category, but then I realized I only wanted to save the list if it's dirty and I neglected to have an `$isDirty` property. So I added the property and set it to `false` in the constructor.

Now, how can we test this method?

We can't.

Why?

Because of that `isDirty` method. The class has no way to actually set the `isDirty` method to `true`. If we were to write a test method at this point, it wouldn't do a lot of good.

The Simple Way to Unit Test

When writing unit tests, you should create a separate test for each possible execution path in the code. If this seems too complicated, then perhaps the method you're testing needs to be refactored.

The `isDirty` flag will be set true any time an attribute or task within the list changes. Since it seems easier right now to set a list attribute, let's implement the `set()` method.

Implementing TodoList::set() and TodoList::get()

Since we're going to implement `TodoList::set()`, let's do `TodoList::get()` at the same time. This allows us to write unit tests, testing `set()` with `get()`.

Update `TodoList.php` so the top portion of the file matches the code below. Also you'll need to delete the `set()` and `get()` method elsewhere in the file.

```
1  <?php namespace GSD\Entities;
2
3  // File: app/src/GSD/Entities/TodoList.php
4
5  use GSD\Repositories\TodoRepositoryInterface;
6
7  class TodoList implements ListInterface {
8
9    protected static $validAttribs = array('id', 'archived', 'subtitle', 'title');
10
11   protected $repository;
12   protected $tasks;
13   protected $attributes;
14   protected $isDirty;
15
16   /**
17    * Inject the dependencies during construction
18    * @param TodoRepositoryInterface $repo The repository
19    * @param TaskCollectionInterface $collection The task collection
```

```
20     */
21    public function __construct(TodoRepositoryInterface $repo,
22      TaskCollectionInterface $collection)
23    {
24      $this->repository = $repo;
25      $this->tasks = $collection;
26      $this->attributes = array();
27      $this->isDirty = false;
28    }
29
30    // List attributes -------------------------
31
32    /**
33     * Return a list attribute
34     * @param string $name id|archived|subtitle|title
35     * @return mixed
36     * @throws InvalidArgumentException If $name is invalid
37     */
38    public function get($name)
39    {
40      if ( ! in_array($name, static::$validAttribs))
41      {
42        throw new \InvalidArgumentException("Invalid attribute named $name");
43      }
44      if (array_key_exists($name, $this->attributes))
45      {
46        return $this->attributes[$name];
47      }
48      return null;
49    }
50
51    /**
52     * Set a list attribute
53     * @param string $name id|archived|subtitle|title
54     * @param mixed $value Attribute value
55     * @return $this for method chaining
56     * @throws InvalidArgumentException If $name is invalid
57     */
58    public function set($name, $value)
59    {
60      if ( ! in_array($name, static::$validAttribs))
61      {
```

```
62        throw new \InvalidArgumentException("Invalid attribute named $name");
63      }
64      if ($name == 'archived') $value = !! $value;
65      $this->attributes[$name] = $value;
66      $this->isDirty = true;
67      return $this;
68    }
69
70    // Rest of file is the same (except deleting get() and set())
71  ?>
```

Comments on the above code:

- I added `$validAttribs` as a static property to have one place to define the valid list attributes (DRY principle).
- I changed the comments for both `get()` and `set()` slightly from what was defined in `ListInterface`. (I'll leave it to you to update `ListInterface` to match these comments.)
- The only specific attribute I'm doing anything special with is the `archived` flag. When setting it I want to make sure it's a truthy value.
- You may have noticed I removed `isDirty` from the list of attributes in the comments. Yes, it is an attribute, but only when the list is loaded. The text file on the hard disk has no concept of dirtiness. It seems cleaner this way.

Testing TodoList::set() and TodoList::get()

Testing these functions is a snap. Add the following methods to the class in `app/tests/GSD/Entities/TodoListTest.php`

```
1   <?php
2
3     /**
4      * @expectedException InvalidArgumentException
5      */
6     public function testGetInvalidNameThrowsException()
7     {
8       $list = App::make('GSD\Entities\TodoList');
9       $list->get('bogus');
10    }
11
```

```
12    /**
13     * @expectedException InvalidArgumentException
14     */
15    public function testSetInvalidNameThrowsException()
16    {
17      $list = App::make('GSD\Entities\TodoList');
18      $list->set('bogus', true);
19    }
20
21    public function testGetSetWorks()
22    {
23      $list = App::make('GSD\Entities\TodoList');
24      $result = $list->set('id', 'abc');
25      $this->assertSame($list, $result);
26      $result = $list->get('id');
27      $this->assertEquals($result, 'abc');
28    }
29  ?>
```

And run your tests

```
1  ~$ cd gsd
2  ~/gsd$ phpunit --tap
3  TAP version 13
4  ok 1 - ExampleTest::testBasicExample
5  ok 2 - TodoListTest::testBoundToInterface
6  ok 3 - TodoListTest::testGetInvalidNameThrowsException
7  ok 4 - TodoListTest::testSetInvalidNameThrowsException
8  ok 5 - TodoListTest::testGetSetWorks
9  ok 6 - TodoManagerTest::testImATeapot
10 ok 7 - TodoManagerTest::testFacade
11 ok 8 - TodoManagerTest::testMakeListThrowsExceptionWhenExists
12 ok 9 - TodoManagerTest::testMakeList
13 ok 10 - TodoManagerTest::testAllListsReturnsArray
14 ok 11 - TodoManagerTest::testAllArchivedListsReturnsArray
15 ok 12 - TodoManagerTest::testGetListThrowsExceptionWhenMissing
16 ok 13 - TodoManagerTest::testGetListReturnsCorrectType
17 1..13
```

Almost done with this chapter. Just one last thing. We need to go back to that untested `save()` method and do some testing.

Testing TodoList::save()

Now that `set()` is done. We can set a value, forcing the `isDirty` flag on, and test all the paths through the `save()` method.

Add the following tests to your `TodoListTest.php` file.

```
<?php
  public function testSaveNotDirtyDoesNothing()
  {
    $list = App::make('GSD\Entities\TodoList');
    //$list->set('id', '123');
    $result = $list->save();
  }

  /**
   * @expectedException RuntimeException
   * @expectedExceptionMessage Cannot save if id not set
   */
  public function testSaveNoIdThrowsException()
  {
    $list = App::make('GSD\Entities\TodoList');
    $list->set('title', 'My Title');
    $list->save();
  }

  /**
   * @expectedException RuntimeException
   * @expectedExceptionMessage Repository could not save
   */
  public function testSaveThrowsExceptionIfRepoFails()
  {
    App::bind('GSD\Repositories\TodoRepositoryInterface', function()
    {
      $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
      $mock->shouldReceive('save')->once()->andReturn(false);
      return $mock;
    });
    $list = App::make('GSD\Entities\TodoList');
    $list->set('id', 'listname');
```

```
34      $list->save();
35    }
36
37    public function testSaveWorksAsExpected()
38    {
39      App::bind('GSD\Repositories\TodoRepositoryInterface', function()
40      {
41        $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
42        $mock->shouldReceive('save')->once()->andReturn(true);
43        return $mock;
44      });
45      $list = App::make('GSD\Entities\TodoList');
46      $list->set('id', 'listname');
47      $result = $list->save();
48      $this->assertSame($list, $result);
49    }
50  ?>
```

Let me explain each test ...

testSaveNotDirtyDoesNothing()

This test method is just checking that nothing really happens if the list is not dirty. Since no changes to the list occur after construction, then it shouldn't be dirty? How do you know it's working? That's why I have that one line:

```
1     //$list->set('id', '123');
```

If you remove the comment, then it sets the `id` attribute, the side effect of which is flipping on `isDirty`. If you run the test with this line active, there'll be an error because our mock repository (created in `setup()`, remember) has no `save()` method.

testSaveNoIdThrowsException()

In this test I call `set()` on the list, but do not set the `id` attribute. This triggers the exception within `TodoList::save()` which makes sure `id` is always set. Since `TodoList::save()` can return multiple exceptions, the docblock above the test method also contains the expected exception method.

testSaveThrowsExceptionIfRepoFails()

Here I bind a new implementation of the mock object for the repository. This mock object expects that `save()` will be called and returns `false`, triggering the exception within `TodoList::save()`.

Keep in mind that the `setup()` method already bound a mock object to `TodoRepositoryInterface`, but the mock object there didn't do anything. We're replacing it with a new binding.

testSaveWorksAsExpected()

This method is almost identical to the last one, but the mock repository will return `true` when `save()` is called. Therefore no exception. All should be working smoothly.

Let's test it

```
~$ cd gsd
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TodoListTest::testBoundToInterface
ok 3 - TodoListTest::testGetInvalidNameThrowsException
ok 4 - TodoListTest::testSetInvalidNameThrowsException
ok 5 - TodoListTest::testGetSetWorks
ok 6 - TodoListTest::testSaveNotDirtyDoesNothing
ok 7 - TodoListTest::testSaveNoIdThrowsException
ok 8 - TodoListTest::testSaveThrowsExceptionIfRepoFails
ok 9 - TodoListTest::testSaveWorksAsExpected
ok 10 - TodoManagerTest::testImATeapot
ok 11 - TodoManagerTest::testFacade
ok 12 - TodoManagerTest::testMakeListThrowsExceptionWhenExists
ok 13 - TodoManagerTest::testMakeList
ok 14 - TodoManagerTest::testAllListsReturnsArray
ok 15 - TodoManagerTest::testAllArchivedListsReturnsArray
ok 16 - TodoManagerTest::testGetListThrowsExceptionWhenMissing
ok 17 - TodoManagerTest::testGetListReturnsCorrectType
1..17
```

Boo Ya! Who's your daddy?

Whew! This chapter was a bit long. I need to take a break and go eat a sandwich.

Chapter 30 - Finishing the TodoList class

In This Chapter

In this chapter we'll finish implementing the `TodoList` class.

What's left to do? The `TodoList` class still has a long list of unimplemented methods. For a moment, I couldn't quite decide how to proceed. Should I continue following the facade mock-ups from back in Chapter 24 or go through the list of unimplemented methods in `TodoList`?

Really, either approach would work at this point, but I'd kind of like to wrap up the `TodoList` class and focus on other missing classes. So let's finish it up.

Finishing the "List Attribute" methods

It looks like the unimplemented "List Attribute" methods will be easy to wrap up.

Add the following four methods to the "List Attribute" section of your `TodoList` class. Be sure and remove the existing, unimplemented versions.

```
1  <?php
2    /**
3     * Return the list's id (base filename)
4     */
5    public function id()
6    {
7      return $this->get('id');
8    }
9
10   /**
11    * Is the list archived?
12    * @return bool
13    */
14   public function isArchived()
```

```
15    {
16      return !! $this->get('archived');
17    }
18
19    /**
20     * Is the list dirty?
21     * @return bool
22     */
23    public function isDirty()
24    {
25      return $this->isDirty;
26    }
27
28    /**
29     * Return the title (alias for get('title'))
30     */
31    public function title()
32    {
33      return $this->get('title');
34    }
35  ?>
```

You know what? I'm not even going to bother unit testing those methods. I mean they were all one-liners.

Removing TodoList::load()

When thinking about implementing the `TodoList::load()` I realize that this functionality is already provided by the `TodoRepositoryInterface`. The `TodoManager::get()` method calls the repository version. Do I need to duplicate the load functionality here?

Of course not.

Remove the definition of the `load()` method from both the `ListInterface` and the `TodoList` class.

Implementing TodoList::archive()

Alrighty then. Let's implement the archive(). How should an archive work? I'm thinking if the list is not already archived, then archiving it will save it in an `archived` directory (overwriting it if already there), then delete the original list.

Add the following method to your `TodoList` in the "List Operations" section

```
<?php
  /**
   * Archive the list. This makes the list only available from the archive.
   * @return ListInterface For method chaining
   * @throws RuntimeException If cannot save.
   */
  public function archive()
  {
    // If already archived, then return this
    if ($this->isArchived())
    {
      return $this;
    }

    if ( ! array_key_exists('id', $this->attributes))
    {
      throw new \RuntimeException("Cannot archive if id not set");
    }
    $id = $this->attributes['id'];

    // Delete existing, unarchived list if it exists
    if ($this->repository->exists($id, false) and
      ! $this->repository->delete($id, false))
    {
      throw new \RuntimeException("Repository failed deleting unarchived list");
    }

    // Set archived and save
    $this->set('archived', true);
    return $this->save();
  }
?>
```

The comments should make the above logic easy to follow. The only point I wanted to make is that, at the bottom of the method, I chose to call the `set()` method to set the archived flag instead of doing it manually. This will force the `isDirty` flag on, thus making sure `save()` will save.

We'll test this in a moment, but the `TodoRepositoryInterface` doesn't yet have a `delete()` method, so let's add that first.

Add the following code to your `TodoRepositoryInterface`

```
  /**
   * Delete the todo list
   * @param string $id ID of the list
   * @return boolean True if successful
   */
  public function delete($id, $archived = false);
```

Okay, let's do the unit tests.

Add the methods below to the `TodoListTest.php` file.

```
<?php
  public function testArchiveWhenAlreadyArchivedDoesNothing()
  {
    $list = App::make('GSD\Entities\TodoList');
    $list->set('archived', true);
    $list->archive();
  }

  /**
   * @expectedException RuntimeException
   * @expectedExceptionMessage Cannot archive if id not set
   */
  public function testArchiveWithNoIdThrowsException()
  {
    $list = App::make('GSD\Entities\TodoList');
    $list->archive();
  }

  /**
   * @expectedException RuntimeException
   * @expectedExceptionMessage Repository failed deleting unarchived list
   */
  public function testArchiveWhenRepoFailsOnDelete()
  {
    App::bind('GSD\Repositories\TodoRepositoryInterface', function()
```

```
26      {
27        $mock = Mockery::mock('GSD\Repositories\TodoRepositoryInterface');
28        $mock->shouldReceive('exists')->once()->andReturn(true);
29        $mock->shouldReceive('delete')->once()->andReturn(false);
30        return $mock;
31      });
32      $list = App::make('GSD\Entities\TodoList');
33      $list->set('id', 'actions');
34      $list->archive();
35    }
36  ?>
```

Those three tests will follow three of the four paths through our `TodoList::archive()` method. I decided not to test the final path, the one that hits the last two lines of the `archive()` method because we've already tested both the `set()` and `save()` methods.

I'll leave it to you run phpunit to make sure the code is working. (It works for me at this point, so if you have any issues check for typos.)

Learning from Unit Tests

A great way to figure out how code operates that you didn't write is to examine the unit tests. This will give you insight into exactly how the initial developer believed his or her code was working.

All that's left with the `TodoList` class are the unimplemented "Task Operation" methods. I'm pretty sure they're going to be wrappers on the `TaskInterface` or the `TaskCollectionInterface`, let's code them and see.

Implementing TodoList::taskAdd()

Let's see ... The `TaskCollectionInterface` has a `add()` method we can wrap.

Update the `TodoList.php`, implementing the `taskAdd()` as follows.

```php
/**
 * Add a new task to the collection
 * @param string|TaskInterface $task Either a TaskInterface or a string
                                    we can construct one from.
 * @return $this for method chaining
 */
public function taskAdd($task)
{
  $this->tasks->add($task);
  $this->isDirty = true;
  return $this;
}
```

That's simple enough I'm not going to bother with a unit test.

Did you notice the `TaskCollectionInterface::add()` takes a different argument than this method? Good eyes. Let's fix it.

Update the `add()` method in `TaskCollectionInterface.php` to match the code below.

```php
/**
 * Add a new task to the collection
 * @param string|TaskInterface $task Either a TaskInterface or a string
                                    we can construct one from.
 */
public function add($task);
```

TodoList::taskCount(), TodoList::task(), and TodoList::taskGet()

I'm kind of getting bored with this chapter and just want it to end. So let's tackle three implementations at once.

Update the three methods below in your `TodoList.php`

```php
<?php
  /**
   * Return number of tasks
   * @return integer
   */
  public function taskCount()
  {
    return count($this->tasks->getAll());
  }

  /**
   * Return a task
   * @param integer $index Task index #
   * @return TaskInterface
   * @throws OutOfBoundsException If $index outside range
   */
  public function task($index)
  {
    return $this->tasks->get($index);
  }

  /**
   * Return a task attribute
   * @param integer $index Task index #
   * @param string $name Attribute name
   * @return mixed
   * @throws OutOfBoundsException If $index outside range
   * @throws InvalidArgumentException If $name is invalid
   */
  public function taskGet($index, $name)
  {
    $task = $this->tasks->get($index);
    return $task->get($name);
  }
?>
```

Those were simple. That's what happens when you design things organically, from the ground up and use interfaces. You get to the point of implementation and very often it's a breeze.

I'm almost tempted to do some unit testing, but since they're wrapping other methods I figure when I get to the underlying methods I can test them. *(Plus, I'm still bored with this chapter?)*

I did notice, when double-checking the methods I'm calling above that the `taskGet()` has the potential of throwing two different types of exceptions. I updated the docblock here and in

`ListInterface` to reflect this.

The final three TodoList::tasks(), TodoList::taskSet(), and TodoList::taskRemove()

Update the final three methods below in your `TodoList.php` file.

```
<?php
  /**
   * Return all tasks as an array.
   * @return array All the TaskInterface objects
   */
  public function tasks()
  {
    return $this->tasks->getAll();
  }

  /**
   * Set a task attribute
   * @param integer $index Task index #
   * @param string $name Attribute name
   * @param mixed $value Attribute value
   * @return $this for method chaining
   * @throws OutOfBoundsException If $index outside range
   * @throws InvalidArgumentException If $name is invalid
   */
  public function taskSet($index, $name, $value)
  {
    $task = $this->tasks->get($index);
    $task->set($name, $value);
    return $this;
  }

  /**
   * Remove the specified task
   * @return $this for method chaining
   * @throws OutOfBoundsException If $index outside range
   */
```

```
32    public function taskRemove($index)
33    {
34      $this->tasks->remove($index);
35      return $this;
36    }
37  ?>
```

Easy. And yeah, still not adding the unit tests. Although I encourage you to add them.

Like the `taskGet()` method earlier, `taskSet()` was missing that second exception type in the docblock. Also, The `@return` was missing from the `taskRemove()` method. I added both of these both in the `TodoList` class and the `ListInterface`.

That's it. Really, there's only three more interfaces that need concrete classes behind them: the `TaskCollectionInterface`, the `TaskInterface`, and the `TodoRepositoryInterface`. Which should I do next?

I'll use the time tested method programmers have used since caveman days to decide things: *Eenie, meenie, minie, moe. Catch a tiger by the toe. If it hollers ...*

Chapter 31 - The TaskCollection and Task classes

In This Chapter

In this chapter we'll completely code both the `TaskCollection` class and the `Task` class and hope they work.

I'm going to try something different with this chapter. I'm going to completely implement two of the remaining interfaces. Yep, just coding the implementation blindly, without testing, just to get it done. Then I'll go back and write tests for the implementations.

The TaskCollection class

Add the file `TaskCollection.php` to your `app/src/GSD/Entities` directory with the following content.

```
1  <?php namespace GSD\Entities;
2
3  // File: app/GSD/Entities/TaskCollection.php
4
5  class TaskCollection implements TaskCollectionInterface {
6
7    protected $tasks;   // Array of TaskInterfaces
8
9    /**
10    * Constructor
11    */
12   public function __construct()
13   {
14     $this->tasks = array();
15   }
16
17   /**
```

```
18     * Add a new task to the collection
19     * @param string|TaskInterface $task Either a TaskInterface or a string
20     *                                   we can construct one from.
21     * @throws InvalidArgumentException If $task not string or TaskInterface
22     */
23    public function add($task)
24    {
25      if ( ! ($task instanceof TaskInterface))
26      {
27        if ( ! is_string($task))
28        {
29          throw new \InvalidArgumentException(
30            '$task must be string or TaskInterface');
31        }
32        $newTask = App::make('GSD/Entities/TaskInterface');
33        if ( ! $newTask->setFromString($task))
34        {
35          throw new \InvalidArgumentException('Cannot parse task string');
36        }
37        $task = $newTask;
38      }
39      $this->tasks[] = $task;
40      $this->sortTasks();
41    }
42
43    /**
44     * Return task based on index
45     * @param integer $index 0 is first item in collection
46     * @return TaskInterface The Todo Task
47     * @throws OutOfBoundsException If $index outside range
48     */
49    public function get($index)
50    {
51      if ($index < 0 || $index >= count($this->tasks))
52      {
53        throw new \OutOfBoundsException('$index is outside range');
54      }
55      return $this->tasks[$index];
56    }
57
58    /**
59     * Return array containing all tasks
```

```
60      * @return array
61      */
62     public function getAll()
63     {
64       return $this->tasks;
65     }
66
67     /**
68      * Remove the specified task
69      * @param integer $index The task to remove
70      * @throws OutOfBoundsException If $index outside range
71      */
72     public function remove($index)
73     {
74       if ($index < 0 || $index >= count($this->tasks))
75       {
76         throw new \OutOfBoundsException('$index is outside range');
77       }
78       unset($this->tasks[$index]);
79       $this->sortTasks();
80     }
81
82     /**
83      * Sort the tasks where:
84      *  1) Next actions are alphabetically first
85      *  2) Normal tasks are alphabetically next
86      *  3) Completed tasks are sorted by date completed, descending
87      */
88     protected function sortTasks()
89     {
90       $next = array();
91       $normal = array();
92       $completed = array();
93       foreach ($this->tasks as $task)
94       {
95         if ($task->isComplete())
96         {
97           $completed[] = $task;
98         }
99         elseif ($task->isNextAction())
100        {
101          $next[] = $task;
```

```
102         }
103         else
104         {
105           $normal[] = $task;
106         }
107       }
108       usort($next, 'TaskCollection::cmpDescription');
109       usort($normal, 'TaskCollection::cmpDescription');
110       usort($completed, 'TaskCollection::cmpCompleted');
111       $this->tasks = array_merge($next, $normal, $completed);
112     }
113
114     /**
115      * Compare two tasks by description
116      */
117     public static function cmpDescription($a, $b)
118     {
119       return strnatcmp($a->description(), $b->description());
120     }
121
122     /**
123      * Compare two tasks by completion date
124      */
125     public static function cmpCompleted($a, $b)
126     {
127       $stamp1 = $a->dateCompleted()->timestamp;
128       $stamp2 = $b->dateCompleted()->timestamp;
129       if ($stamp1 == $stamp2)
130       {
131         return strnatcmp($a->description(), $b->description());
132       }
133       return $stamp1 - $stamp2;
134     }
135 }
136 ?>
```

Everything within the `TaskCollection` should be straightforward and easy to follow. I decided while coding that I'd keep the list sorted. So when the `add()` method or `remove()` method are called, it hits an internal `sort()` method to do it.

Also, I needed a `setFromString()` method on the `TaskInterface` to set all the task's information from a string. Let's go add that to the interface. And while we're at it, let's add a method to return the task as a string. I'll use the magic method `__toString()` to convert to a string because, well,

anything with the word **magic** in it has got to be good.

(Um, as I'm checking for typos I realize the movie "Magic Mike" has the word magic in it. I didn't see the movie, but I cannot imagine I'd like it. So, I'll reverse my earlier statement ... not everything with the word "magic" in it is necessarily good.)

Add the method below to the `TaskInterface`.

```
<?php
  /**
   * Set all the tasks attributes from a string.
   * @param string $info The task info
   * @return bool True on success, false otherwise
   */
  public function setFromString($info);

  /**
   * Return the task as a string
   */
  public function __toString();
?>
```

Will it work? I really hope so. Let's keep going though.

The Task class

I'm doing the same thing and completely coding the `Task` class. It's almost like I'm attempting to fail in public here because code is very seldom perfect the first time around. *(My code, at least.)*

Add the file `Task.php` to your `app/src/GSD/Entities` directory with the following content.

```php
<?php namespace GSD\Entities;

// File: app/src/GSD/Entities/Task.php

use Carbon\Carbon;

class Task implements TaskInterface {

  protected $complete;        // Is the task complete?
  protected $description;     // Task description
  protected $due;             // null or Carbon
  protected $whenCompleted;   // null or Carbon
  protected $nextAction;      // Is this a next action?

  /**
   * Constructor
   */
  public function __construct()
  {
    $this->clear();
  }

  /**
   * Clear all task attributes
   */
  protected function clear()
  {
    $this->complete = false;
    $this->description = '';
    $this->due = null;
    $this->whenCompleted = null;
    $this->nextAction = false;
  }

  /**
   * Has the task been completed?
   * @return boolean
   */
  public function isComplete()
  {
    return $this->complete;
  }
```

```
43
44    /**
45     * What's the description of the task
46     * @return string
47     */
48    public function description()
49    {
50      return $this->description;
51    }
52
53    /**
54     * When is the task due?
55     * @return mixed Either null if no due date set, or a Carbon object.
56     */
57    public function dateDue()
58    {
59      return $this->due;
60    }
61
62    /**
63     * When was the task completed?
64     * @return mixed Either null if not complete, or a Carbon object
65     */
66    public function dateCompleted()
67    {
68      return $this->whenCompleted;
69    }
70
71    /**
72     * Is the task a Next Action?
73     * @return boolean
74     */
75    public function isNextAction()
76    {
77      return $this->nextAction;
78    }
79
80    /**
81     * Set whether task is complete. Automatically updates dateCompleted.
82     * @param bool $complete
83     */
84    public function setIsComplete($complete)
```

```
85    {
86      $this->complete = !! $complete;
87      if ($this->complete)
88      {
89        $this->whenCompleted = new Carbon;
90      }
91      else
92      {
93        $this->whenCompleted = null;
94      }
95    }
96
97    /**
98     * Set task description
99     * @param string $description
100    */
101   public function setDescription($description)
102   {
103     $this->description = $description;
104   }
105
106   /**
107    * Set date due
108    * @param null|string|Carbon $date null to clear, otherwise stores Carbon
109    *        date internally.
110    * @throws InvalidArgumentException If $date is not null or Carbon
111    */
112   public function setDateDue($date)
113   {
114     if ( ! is_null($date) and ! ($date instanceof Carbon))
115     {
116       throw new \InvalidArgumentException('$date is not null or Carbon');
117     }
118     $this->due = $date;
119   }
120
121   /**
122    * Set whether task is a next action
123    * @param bool $nextAction
124    */
125   public function setIsNextAction($nextAction)
126   {
```

```
127       $this->nextAction = !! $nextAction;
128     }
129
130     /**
131      * Set a property. (Ends up calling specific setter)
132      * @param string $name isComplete|description|dateDue|isNextAction
133      * @param mixed $value The value to set
134      * @throws InvalidArgumentException If $name is invalid
135      */
136     public function set($name, $value)
137     {
138       switch ($name)
139       {
140         case 'isComplete':
141           $this->setIsComplete($value);
142           break;
143         case 'description':
144           $this->setDescription($value);
145           break;
146         case 'dateDue':
147           $this->setDateDue($value);
148           break;
149         case 'isNextAction':
150           $this->setIsNextAction($value);
151           break;
152         default:
153           throw new \InvalidArgumentException("Invalid attribute $name");
154       }
155     }
156
157     /**
158      * Get a property.
159      * @param string $name isComplete|description|dateDue|isNextAction|dateCompleted
160      * @return mixed
161      * @throws InvalidArgumentException If $name is invalid
162      */
163     public function get($name)
164     {
165       switch ($name)
166       {
167         case 'isComplete':
168           return $this->isComplete();
```

```
169         case 'description':
170           return $this->description();
171         case 'dateDue':
172           return $this->dateDue();
173         case 'isNextAction':
174           return $this->isNextAction();
175         case 'dateCompleted':
176           return $this->dateCompleted();
177         default:
178           throw new \InvalidArgumentException("Invalid attribute $name");
179       }
180     }
181
182     /**
183      * Set all the tasks attributes from a string.
184      * @param string $info The task info
185      * @return bool True on success, false otherwise
186      */
187     public function setFromString($info)
188     {
189       $this->clear();
190
191       // Remove dup spaces and split into words
192       $info = preg_replace('/\s\s+/', ' ', $info);
193       $words = explode(' ', trim($info));
194       if (count($words) == 0)
195       {
196         return false;
197       }
198
199       // Completed item
200       if ($words[0] == 'x')
201       {
202         $this->complete = true;
203         array_shift($words);
204         try
205         {
206           $this->whenCompleted = new Carbon(array_shift($words));
207         }
208         catch (\Exception $e)
209         {
210           return false;
```

```
211         }
212       }
213
214       // Next action
215       else if ($words[0] == '*')
216       {
217         $this->nextAction = true;
218         array_shift($words);
219       }
220
221       // Normal item
222       else if ($words[0] == '-')
223       {
224         array_shift($words);
225       }
226
227       // Look for a due date
228       for ($i = 0; $i < count($words); $i++)
229       {
230         if (substr($words[$i], 0, 5) == ':due:')
231         {
232           $this->due = new Carbon(substr($words[$i], 5));
233           unset($words[$i]);
234           break;
235         }
236       }
237
238       $this->description = join(' ', $words);
239       return true;
240     }
241
242     /**
243      * Return the task as a string
244      */
245     public function __toString()
246     {
247       $build = array();
248       if ($this->complete)
249       {
250         $build[] = 'x';
251         $build[] = $this->whenCompleted->format('Y-m-d');
252       }
```

```
253       elseif ($this->nextAction)
254       {
255         $build[] = '*';
256       }
257       else
258       {
259         $build[] = '-';
260       }
261       $build[] = $this->description;
262       if ($this->due)
263       {
264         $build[] = ':due:' . $this->due->format('Y-m-d');
265       }
266       return join(' ', $build);
267     }
268 }
269 ?>
```

Whew! That seems like a lot of code, but really it was easy to write. This is because most of the kinks were worked out already using the `TaskInterface` and `TaskCollectionInterface`. So this chapter really was just following a map to write code.

Binding the Interfaces

Since we've implemented the concrete classes behind `TaskCollectionInterface` and `TaskInterface`, let's bind the interfaces to them.

Where do you think this should happen? You are correct In the service provider.

Update the `TodoServiceProvider.php` file in your `app/src/GSD/Providers` directory to match the code below.

```php
<?php namespace GSD\Providers;

// File: app/src/GSD/Providers/TodoServiceProvider.php

use Illuminate\Support\ServiceProvider;

class TodoServiceProvider extends ServiceProvider {

  /**
   * Register the service provider
   */
  public function register()
  {
    $this->app['todo'] = $this->app->share(function()
    {
      return new TodoManager;
    });

    $this->app->bind('GSD\Entities\ListInterface', 'GSD\Entities\TodoList');
    $this->app->bind('GSD\Entities\TaskInterface', 'GSD\Entities\Task');
    $this->app->bind('GSD\Entities\TaskCollectionInterface',
      'GSD\Entities\TaskCollection');
  }
}
?>
```

Now any time we do an `App::make('TaskInterface')` Laravel will automatically return an instance `Task` class.

I am a wee bit nervous with all of this blind coding. I can almost guarantee you there's going to be something wrong. But who knows? Sometimes people get a hole-in-one and how impossible is that?

Let's see what happens with testing in the next chapter.

Chapter 32 - Testing the TaskCollection and Task classes

In This Chapter

In this chapter we'll test those two classes we created in the last chapter.

Should Getters and Setters be Unit Tested

Sure, if you want to. Me? I often don't unit test any getter or setter unless there's some type of side effect going on. Like setting the `completed` flag on a task with the side effect of the `dateCompleted` also being set.

But again, every programmer's different. Some people like the feeling of completeness of having each and every method testing. I like that feeling too, but the feeling I like even more is the one you have when it's all done. And, honestly, sometimes in the spirit of *Getting Stuff Done* I'll forgo many tests that I plan on coming back to. Yet, somehow, I never seem to have the time to come back and do it.

Testing the Task class

Add the `TaskTest.php` file to your `app/tests/GSD/Entities` directory with the contents below.

```
1  <?php
2  // File: app/tests/GSD/Entities/TaskTest.php
3
4  class TaskTest extends TestCase {
5
6    protected function newTask()
7    {
8      return App::make('GSD\Entities\TaskInterface');
9    }
10
11   public function testGetters()
```

```
12    {
13      $task = $this->newTask();
14
15      // Use specific getters
16      $this->assertFalse($task->isComplete());
17      $this->assertEquals('', $task->description());
18      $this->assertNull($task->dateDue());
19      $this->assertNull($task->dateCompleted());
20      $this->assertFalse($task->isNextAction());
21
22      // Use generic getter
23      $this->assertFalse($task->get('isComplete'));
24      $this->assertEquals('', $task->get('description'));
25      $this->assertNull($task->get('dateDue'));
26      $this->assertNull($task->get('dateCompleted'));
27      $this->assertFalse($task->get('isNextAction'));
28    }
29
30    public function testSettingCompleteUpdatesWhenComplete()
31    {
32      $task = $this->newTask();
33
34      $task->setIsComplete(true);
35      $this->assertInstanceOf('Carbon\Carbon', $task->dateCompleted());
36      $this->assertEquals(date('Y-m-d'), $task->dateCompleted()->format('Y-m-d'));
37    }
38
39    /**
40     * @expectedException InvalidArgumentException
41     */
42    public function testSetDueThrowsException()
43    {
44      $task = $this->newTask();
45      $task->setDateDue(123);
46    }
47
48    public function testOtherSetters()
49    {
50      $task = $this->newTask();
51
52      $test1 = 'Test description';
53      $test2 = 'Another test';
```

```
54      $task->setDescription($test1);
55      $this->assertEquals($test1, $task->description());
56      $task->set('description', $test2);
57      $this->assertEquals($test2, $task->description());
58
59      $test1 = new Carbon\Carbon('1/1/2013');
60      $task->setDateDue($test1);
61      $this->assertEquals($test1, $task->dateDue());
62      $task->set('dateDue', null);
63      $this->assertNull($task->dateDue());
64
65      $task->setIsNextAction(true);
66      $this->assertTrue($task->isNextAction());
67      $task->set('isNextAction', false);
68      $this->assertFalse($task->isNextAction());
69    }
70
71    /**
72     * @expectedException InvalidArgumentException
73     */
74    public function testGetWithBadNameThrowsError()
75    {
76      $task = $this->newTask();
77      $task->get('something');
78    }
79
80    /**
81     * @expectedException InvalidArgumentException
82     */
83    public function testSetWithBadNameThrowsError()
84    {
85      $task = $this->newTask();
86      $task->set('something', 'bla');
87    }
88
89    /**
90     * @dataProvider stringTests
91     */
92    public function testStringVariations($string, $valid, $stringSame)
93    {
94      $task = $this->newTask();
95
```

```
96      $result = $task->setFromString($string);
97      if ($valid)
98      {
99        $this->assertTrue($result);
100       if ($stringSame)
101       {
102         $this->assertEquals($string, (string)$task);
103       }
104       else
105       {
106         $this->assertNotEquals($string, (string)$task);
107       }
108     }
109     else
110     {
111       $this->assertFalse($result);
112     }
113   }
114   public function stringTests()
115   {
116     return array(
117       array('', false, false),
118       array('* Simple next action', true, true),
119       array('* Next with due date :due:2013-09-14', true, true),
120       array('- task with  an extra space', true, false),
121       array('x bad', false, false),
122       array('- Due date :due:2013-09-14 in middle', true, false),
123       array('x 2013-08-03 Start Laravel Book: Getting Stuff Done', true, true),
124     );
125   }
126 }
127 ?>
```

I'm not going to attempt to pull the wool over your eyes and tell you I coded this entire test and it worked first time. I didn't and it didn't. I wrote one test, ran phpunit, then moved to the next test case. During this process I had to fix a few typos in `TaskTest` (I always type `assertEqual()` instead of `assertEquals()` ... don't ask me why).

And during this process something bad happened ...

I ran into a bug in the existing `Task` code.

Amazingly enough there was not a single typo. I consider that a huge win. (Although I did triple check my code in the last chapter before moving on.)

We'll fix the bug, that keeps the tests from operating is in the next section of this chapter.

Another note. Notice how I'm creating a new instance of class using the interface name in `newTask()`? That tells us the interface binding is working.

Fixing the mistake in the Task class

In the `setFromString()` method I tried checking if the string was empty after I split it into words with the logic:

```
1   // Remove dup spaces and split into words
2   $info = preg_replace('/\s\s+/', ' ', $info);
3   $words = explode(' ', trim($info));
4   if (count($words) == 0)
5   {
6     return false;
7   }
```

The problem, of course, is that if my string is empty, `$words` will be an array, 1 big, with an empty string.

Update the `Task` class, replacing the `setFromString()` method with the following code

```
1   /**
2    * Set all the tasks attributes from a string.
3    * @param string $info The task info
4    * @return bool True on success, false otherwise
5    */
6   public function setFromString($info)
7   {
8     $this->clear();
9
10    // Remove dup spaces and split into words
11    $info = preg_replace('/\s\s+/', ' ', $info);
12    $words = explode(' ', trim($info));
13    if (count($words) == 1 && $words[0] == '')
14    {
15      return false;
16    }
17
18    // the rest of the method is unchanged
```

That will make all the `TaskTest` tests pass.

Do it. Run phpunit. Everything should be working before moving on.

Fixing Timezone in the Configuration

Another thing I discovered in creating the `TaskTest` class above is that I neglected to set the timezone of my application.

Edit your `app/config/app.php` file and change the timezone as follows.

```
1   // change this
2   'timezone' => 'UTC',
3
4   // to this (actually, use your own timezone)
5   'timezone' => 'America/Los_Angeles',
```

Testing the TaskCollection class

Before presenting the unit tests to you, let's go fix a few of the errors. *(I don't know what to say. I thought my code was perfect. Turns out I was wrong.)* These errors keep the next set of unit tests from working. So gotta fix 'em now.

Or ...

If you want, create the unit tests yourself and trace down the following 3 errors to fix yourself.

1st problem

In the `TaskCollection::sortTasks()` method, I had the following 3 lines:

```
1   usort($next, 'TaskCollection::cmpDescription');
2   usort($normal, 'TaskCollection::cmpDescription');
3   usort($completed, 'TaskCollection::cmpCompleted');
```

Well, those 2nd arguments aren't valid callbacks.

Change those lines to:

```
    usort($next, 'static::cmpDescription');
    usort($normal, 'static::cmpDescription');
    usort($completed, 'static::cmpCompleted');
```

2nd problem

In the `TaskCollection::add()` method, I use the `App` facade without namespacing it correctly.

Fix it by adding a slash before the facade name.

```
    // Change this
    $newTask = App::make('GSD/Entities/TaskInterface');

    // to this
    $newTask = \App::make('GSD/Entities/TaskInterface');
```

3rd problem

In the same file, on the same exact line, I have the slashes wrong.

Fix it by changing the slashes to backslashes.

```
    // Change this
    $newTask = \App::make('GSD/Entities/TaskInterface');

    // to this
    $newTask = \App::make('GSD\Entities\TaskInterface');
```

That's all the problems. Not perfect, but still not too awful.

Create the `TaskCollectionTest.php` file in your `app/tests/GSD/Entities` directory with the following content.

```php
<?php
// File: app/tests/GSD/Entities/TaskCollectionTest.php

class TaskCollectionTest extends TestCase {

  protected function newCollection()
  {
    return App::make('GSD\Entities\TaskCollectionInterface');
  }

  public function testAddFromClassWorks()
  {
    $tasks = $this->newCollection();
    $task = App::make('GSD\Entities\Task');
    $task->setDescription('a simple test');
    $tasks->add($task);
  }

  /**
   * @expectedException InvalidArgumentException
   * @expectedExceptionMessage $task must be string or TaskInterface
   */
  public function testAddWithInvalidTypeThrowsError()
  {
    $tasks = $this->newCollection();
    $tasks->add(3.0);
  }

  /**
   * @expectedException InvalidArgumentException
   * @expectedExceptionMessage Cannot parse task string
   */
  public function testAddWithEmptyStringThrowsError()
  {
    $tasks = $this->newCollection();
    $tasks->add('');
  }

  public function testAddWithValidString()
  {
    $tasks = $this->newCollection();
    $description = 'Something todo';
```

```
43      $tasks->add($description);
44      $this->assertEquals($description, $tasks->get(0)->description());
45    }
46
47    /**
48     * @expectedException OutOfBoundsException
49     */
50    public function testGetWhenEmptyThrowsException()
51    {
52      $tasks = $this->newCollection();
53      $tasks->get(0);
54    }
55
56    public function testGetAll()
57    {
58      $tasks = $this->newCollection();
59      $result = $tasks->getAll();
60      $this->assertSame(array(), $result);
61      $tasks->add("Item 1");
62      $tasks->add("Item 2");
63      $result = $tasks->getAll();
64      $this->assertInternalType('array', $result);
65      $this->assertCount(2, $result);
66    }
67
68    /**
69     * @expectedException OutOfBoundsException
70     */
71    public function testRemoveThrowsException()
72    {
73      $tasks = $this->newCollection();
74      $tasks->add('item 1');
75      $tasks->remove(1);
76    }
77
78    public function testAddSortRemove()
79    {
80      $tasks = $this->newCollection();
81      $tasks->add('Zebra painting');
82      $tasks->add('Alligator wrestling');
83      $tasks->add('Monkey business');
84      $this->assertEquals('Alligator wrestling', $tasks->get(0)->description());
```

```
85      $tasks->remove(0);
86      $tasks->remove(1);
87      $result = $tasks->getAll();
88      $this->assertInternalType('array', $result);
89      $this->assertCount(1, $result);
90      $this->assertEquals('Monkey business', $result[0]->description());
91    }
92  }
93  ?>
```

Now if you run phpunit, it should generate that odor that we all love–the sweet smell of success!

Run php unit

```
1  ~$ cd gsd
2  ~/gsd$ phpunit --testdox
3  PHPUnit 3.7.26 by Sebastian Bergmann.
4
5  Configuration read from /home/chuck/gsd/phpunit.xml
6
7  Example
8   [x] Basic example
9
10 TaskCollection
11  [x] Add from class works
12  [x] Add with invalid type throws error
13  [x] Add with empty string throws error
14  [x] Add with valid string
15  [x] Get when empty throws exception
16  [x] Get all
17  [x] Remove throws exception
18  [x] Add sort remove
19
20 Task
21  [x] Getters
22  [x] Setting complete updates when complete
23  [x] Set due throws exception
24  [x] Other setters
25  [x] Get with bad name throws error
26  [x] Set with bad name throws error
```

```
 [x] String variations

TodoList
 [x] Bound to interface
 [x] Get invalid name throws exception
 [x] Set invalid name throws exception
 [x] Get set works
 [x] Save not dirty does nothing
 [x] Save no id throws exception
 [x] Save throws exception if repo fails
 [x] Save works as expected
 [x] Archive when already archived does nothing
 [x] Archive with no id throws exception
 [x] Archive when repo fails on delete

TodoManager
 [x] Im a teapot
 [x] Facade
 [x] Make list throws exception when exists
 [x] Make list
 [x] All lists returns array
 [x] All archived lists returns array
 [x] Get list throws exception when missing
 [x] Get list returns correct type
```

All good. We're up to 41 tests now with 70 assertions.

Chapter 33 - Implementing the TodoRepository

In This Chapter

In this chapter we'll code the `TodoRepository`. This is the last concrete class needed to support all of our interfaces.

A dab of Refactoring

As I started thinking about implementing the `TodoRepository` I realized that I don't like the `save()` method. It takes an `$id` and `$archived` parameter, both of which the list being saved can provide.

So let's refactor those arguments away.

Change the `save()` method in `TodoRepositoryInterface` to what's below.

```
1   /**
2    * Save a TodoList
3    * @param ListInterface $list The TODO List
4    * @return boolean True if successful
5    */
6   public function save(ListInterface $list);
```

Change the `save()` method in the `TodoList` class to what's below.

```
  /**
   * Save the list
   * @return $this For method chaining
   * @throws RuntimeException If cannot save.
   */
  public function save()
  {
    if ($this->isDirty)
    {
      if ( ! array_key_exists('id', $this->attributes))
      {
        throw new \RuntimeException("Cannot save if id not set");
      }
      if ( ! $this->repository->save($this))
      {
        throw new \RuntimeException("Repository could not save");
      }
      $this->isDirty = false;
    }
    return $this;
  }
```

That kills a couple arguments and a couple lines of code. Excellent. Any time you're able to remove code without loss of functionality is good. If you run phpunit, tests should be passing.

TodoRepository

Okay, I'm going to implement the repository using the PHP file system methods.

Create the `TodoRepository.php` file in the `app/src/Repositories` directory with the following content.

```php
<?php namespace GSD\Repositories;

// File: app/src/GSD/Repositories/TodoRepository.php

use Config;
use GSD\Entities\ListInterface;

class TodoRepository implements TodoRepositoryInterface {

  protected $path;
  protected $extension;

  /**
   * Constructor. We'll throw exceptions if the paths don't exist
   */
  public function __construct()
  {
    $this->path = str_finish(Config::get('app.gsd.folder'), '/');
    if ( ! is_dir($this->path))
    {
      throw new \RuntimeException("Directory doesn't exist: $this->path");
    }
    if ( ! is_dir($this->path.'archived'))
    {
      throw new \RuntimeException("Directory doesn't exist: $this->path".
        'archived');
    }
    $this->extension = Config::get('app.gsd.extension');
    if ( ! starts_with($this->extension, '.'))
    {
      $this->extension = '.' . $this->extension;
    }
  }

  /**
   * Delete the todo list
   * @param string $id ID of the list
   * @return boolean True if successful
   */
  public function delete($id, $archived = false)
  {
    $file = $this->fullpath($id, $archived);
```

```
43      if (file_exists($file))
44      {
45        return unlink($file);
46      }
47      return false;
48    }
49
50    /**
51     * Does the todo list exist?
52     * @param  string $id ID of the list
53     * @param  boolean $archived Check for archived lists only?
54     * @return boolean
55     */
56    public function exists($id, $archived = false)
57    {
58      $file = $this->fullpath($id, $archived);
59      return file_exists($file);
60    }
61
62    /**
63     * Return the ids of all the lists
64     * @param  boolean $archived Return archived ids or unarchived?
65     * @return array of list ids
66     */
67    public function getAll($archived = false)
68    {
69      $match = $this->path;
70      if ($archived)
71      {
72        $match .= 'archived/';
73      }
74      $match .= '*' . $this->extension;
75      $files = glob($match);
76      $ids = array();
77      foreach ($files as $file)
78      {
79        $ids[] = basename($file, $this->extension);
80      }
81      return $ids;
82    }
83
84    /**
```

```
85      * Load a TodoList from it's id
86      * @param string $id ID of the list
87      * @param boolean $archived Load an archived list?
88      * @return ListInterface The list
89      * @throws InvalidArgumentException If $id not found
90      */
91     public function load($id, $archived = false)
92     {
93       if ( ! $this->exists($id, $archived))
94       {
95         throw new \InvalidArgumentException(
96           "List with id=$id, archived=$archived not found");
97       }
98       $lines = file($this->fullpath($id, $archived));
99
100      // Pull title
101      $title = array_shift($lines);
102      $title = trim(substr($title, 1));
103
104      // Pull subtitle
105      if (count($lines) && $lines[0][0] == '(')
106      {
107        $subtitle = trim(array_shift($lines));
108        $subtitle = ltrim($subtitle, '(');
109        $subtitle = rtrim($subtitle, ')');
110      }
111
112      // Setup the list
113      $list = \App::make('GSD\Entities\ListInterface');
114      $list->set('id', $id);
115      $list->set('title', $title);
116      if ( ! empty($subtitle))
117      {
118        $list->set('subtitle', $subtitle);
119      }
120      $list->set('archived', $archived);
121
122      // And add the tasks
123      foreach ($lines as $line)
124      {
125        $line = trim($line);
126        if ($line)
```

```
127         {
128           $list->taskAdd($line);
129         }
130       }
131
132       return $list;
133     }
134
135     /**
136      * Save a TodoList
137      * @param ListInterface $list The TODO List
138      * @return boolean True if successful
139      */
140     public function save(ListInterface $list)
141     {
142       $id = $list->get('id');
143       $archived = $list->get('archived');
144       $build = array();
145       $build[] = '# ' . $list->get('title');
146       $subtitle = $list->get('subtitle');
147       if ($subtitle)
148       {
149        $build[] = "($subtitle)";
150       }
151       $lastType = '';
152       $tasks = $list->tasks();
153       foreach ($tasks as $task)
154       {
155         $task = (string)$task;
156         $type = $task[0];
157         if ($type != $lastType)
158         {
159           $build[] = '';  // Blank line between types of tasks
160           $lastType = $type;
161         }
162         $build[] = $task;
163       }
164       $content = join("\n", $build);
165       $filename = $this->fullpath($id, $archived);
166       $result = file_put_contents($filename, $content);
167
168       return $result !== false;
```

```
169     }
170
171     /**
172      * Return the path to the list file
173      */
174     protected function fullpath($id, $archived)
175     {
176       $path = $this->path;
177       if ($archived)
178       {
179         $path .= 'archived/';
180       }
181       $path .= $id . $this->extension;
182       return $path;
183     }
184 }
185 ?>
```

I didn't write this blindly without testing. We found out what happens when I do that :)

Now let's bind the interface.

Edit your `TodoServiceProvider` and within the `register()` method add another binding as specified below.

```
1     $this->app->bind('GSD\Repositories\TodoRepositoryInterface',
2       'GSD\Repositories\TodoRepository');
```

Now let's start testing this puppy.

Creating Test data

The first step is to create some test data. The idea here is that we want to be able to control our testing environment. So we'll create a directory with some test lists. This way we can use the same todo lists with every test. We'll have a *known state* of data.

Create an `app/config/testing/app.php` file with the following content.

```
<?php
// testing config
return array(
    'gsd' => array(
        'folder' => app_path().'/tests/GSD/Repositories/data',
    ),
);
?>
```

Now, for testing, our folder points to something safe.

Create the directory `app/tests/GSD/Repositories/data/archived` and all needed subdirectories.

Now we'll create a couple test files.

Create the file `test1.test` in the newly created `app/tests/GSD/Repositories/data` directory with the following content.

```
# Test List 1

- Something simple

x 2013-09-13 Something I finished
```

Create the file `test2.test` in the same directory with the following content.

```
# Test List 2
(With a subtitle, yea!)

* What I want to do next

- something i'll do
- something that's due soon :due:2013-09-15
```

Alrighty, we have some lists to test with. Let's get testing.

Testing the Repository

I threw together this test pretty quickly. The important thing to note within the test is that it clears the `*.txt` files out of the `data` and `data/archived` directories. Then it copies our two text files over to be usable. This way every test starts with the same set of data.

Create the `TodoRepositoryTest.php` in the `tests/GSD/Repositories` directory. Make it match the following:

```
1  <?php
2  // File: app/tests/GSD/Repositories/TodoRepositoryTest.php
3
4  class TodoRepositoryTest extends TestCase {
5
6    protected $repo;
7
8    // Deletes any existing lists and copies the blank ones over
9    public function setup()
10   {
11     $path = __DIR__.'/data/';
12     $ext = Config::get('app.gsd.extension');
13     $files = array_merge(
14       glob($path.'*'.$ext),
15       glob($path.'archived/*'.$ext)
16     );
17     foreach ($files as $file)
18     {
19       unlink($file);
20     }
21     copy($path.'test1.test', $path.'test1'.$ext);
22     copy($path.'test2.test', $path.'test2'.$ext);
23
24     $this->repo = App::make('GSD\Repositories\TodoRepositoryInterface');
25   }
26
27   public function testDelete()
28   {
29     $result = $this->repo->delete('test1');
30     $this->assertTrue($result);
31   }
32
```

```
33    public function testDeleteArchived()
34    {
35      // Save test1 as archived
36      $list = $this->repo->load('test1');
37      $list->set('archived', true);
38      $list->set('id', 'test-archived');
39      $list->save();
40
41      // Delete it
42      $result = $this->repo->delete('test-archived', true);
43      $this->assertTrue($result);
44    }
45
46    public function testDeleteMissingReturnsFalse()
47    {
48      $result = $this->repo->delete('bogus');
49      $this->assertFalse($result);
50    }
51
52    public function testExists()
53    {
54      $this->assertTrue($this->repo->exists('test2'));
55      $this->assertFalse($this->repo->exists('bogus'));
56    }
57
58    public function testExistsArchived()
59    {
60      // Save test1 as archived
61      $list = $this->repo->load('test1');
62      $list->set('archived', true);
63      $list->set('id', 'test-archived');
64      $list->save();
65
66      $this->assertTrue($this->repo->exists('test-archived', true));
67      $this->assertFalse($this->repo->exists('something-else', true));
68    }
69
70    public function testGetAll()
71    {
72      $result = $this->repo->getAll();
73      $this->assertCount(2, $result);
74      $result = $this->repo->getAll(true);
```

```
75      $this->assertSame(array(), $result);
76    }
77
78    /**
79     * @expectedException InvalidArgumentException
80     */
81    public function testLoadMissingThrowsException()
82    {
83      $this->repo->load('bogus');
84    }
85
86    public function testLoadedComponents()
87    {
88      $list = $this->repo->load('test1');
89      $this->assertEquals('Test List 1', $list->get('title'));
90      $this->assertNull($list->get('subtitle'));
91      $this->assertEquals('Something simple', $list->taskGet(0, 'description'));
92      $this->assertEquals(
93          '2013-09-13',
94          $list->taskGet(1, 'dateCompleted')->format('Y-m-d')
95      );
96    }
97
98    public function testSaves()
99    {
100     $list = $this->repo->load('test1');
101     $list->set('id', 'test-one');
102     $list->save();
103     $file1 = $this->loadAndTrim(__DIR__.'/data/test1.txt');
104     $file2 = $this->loadAndTrim(__DIR__.'/data/test-one.txt');
105     $this->assertEquals($file1, $file2);
106
107     $list = $this->repo->load('test2');
108     $list->set('archived', true);
109     $list->save();
110     $file1 = $this->loadAndTrim(__DIR__.'/data/test2.txt');
111     $file2 = $this->loadAndTrim(__DIR__.'/data/archived/test2.txt');
112     $this->assertEquals($file1, $file2);
113   }
114
115   protected function loadAndTrim($filename)
116   {
```

```
117     $content = file_get_contents($filename);
118     $content = str_replace("\r\n", "\n", $content);
119     return trim($content);
120   }
121 }
122 ?>
```

Whew! Only one thing left to do.

Run phpunit to test things

```
1  ~$ cd gsd
2  ~/gsd$ phpunit
3  PHPUnit 3.7.26 by Sebastian Bergmann.
4
5  Configuration read from /home/chuck/gsd/phpunit.xml
6
7  ..................................................
8
9  Time: 233 ms, Memory: 22.75Mb
10
11 OK (50 tests, 86 assertions)
```

All good.

That's the end of Part 2

We have all the support work designed and completed. I kind of feel like I'm all dressed up and have no place to go. That's okay. Because the next part of this manual we're going places ... Part 3 we'll use all this work and actually be able to create some, real live lists.

Part 3 - The Console Application

In this section of the book, we're going to take the application and make it work from a console window. You'll be able to list the things you need to do, add tasks, edit tasks, and mark things completed right from your console window.

Chapter 34 - Artisan Tightening

In This Chapter

In this chapter we'll discuss the `artisan` utility and tighten it up a bit by removing commands our application won't need.

Laravel provides a command line interface named **artisan**. I just love the name. It's like Laravel wants to remind you how great it thinks you are.

Artisan
: A worker in a skilled trade, esp. one that involves making things by hand.

Yeah … exactly.

Artisan in 30 Seconds

The cool thing about artisan is that it can be extended. You can tap into it's structure and create your own console utilities. Use it for reoccurring tasks, cron jobs, database imports/exports, batch jobs, or anything else you can think of.

In your project's root (`gsd` in this book's examples) you can bring up the artisan commands either by typing `php artisan` or `./artisan`.

```
~$ cd gsd
~/gsd$ ./artisan
Laravel Framework version 4.0.7

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v|vv|vvv Increase the verbosity of messages: 1 for
    normal output, 2 for more verbose output and 3 for debug
  --version        -V Display this application version.
```

```
  --ansi              Force ANSI output.
  --no-ansi           Disable ANSI output.
  --no-interaction -n Do not ask any interactive question.
  --env               The environment the command should run under.

Available commands:
  changes            Display the framework change list
  clear-compiled     Remove the compiled class file
  down               Put the application into maintenance mode
  dump-autoload      Regenerate framework autoload files
  help               Displays help for a command
  list               Lists commands
  migrate            Run the database migrations
  optimize           Optimize the framework for better performance
  routes             List all registered routes
  serve              Serve the application on the PHP development server
  tinker             Interact with your application
  up                 Bring the application out of maintenance mode
  workbench          Create a new package workbench
asset
  asset:publish      Publish a package's assets to the public directory
auth
  auth:reminders     Create a migration for the password reminders table
cache
  cache:clear        Flush the application cache
command
  command:make       Create a new Artisan command
config
  config:publish     Publish a package's configuration to the application
controller
  controller:make    Create a new resourceful controller
db
  db:seed            Seed the database with records
key
  key:generate       Set the application key
migrate
  migrate:install    Create the migration repository
  migrate:make       Create a new migration file
  migrate:refresh    Reset and re-run all migrations
  migrate:reset      Rollback all database migrations
  migrate:rollback   Rollback the last database migration
queue
```

```
  queue:listen       Listen to a given queue
  queue:subscribe    Subscribe a URL to an Iron.io push queue
  queue:work         Process the next job on a queue
session
  session:table      Create a migration for the session database table
```

Whew! That's a lot of stuff you can do right out of the box.

Use an Alias

On Linux, I have the command `alias art='php artisan'` in my `.bashrc`. This allows me to simply type `art command` instead of the longer path. Also, by using this shortcut, I'm reminded that I'm creating art. (Believe me, sometimes I need to keep reminding myself.) In Windows you can set up a batch file to do the same thing.

Where are these commands?

The commands in the above list are the ones that ship standard with Laravel. They're provided by service providers. Often the question is:

Where do *I* put commands?

Out of the box Laravel provides an `app/commands` directory, giving you a logical place to dump all your commands. We changed that structure back in Chapter 19 and instead created `app/src/GSD/Commands`. That's where we'll be placing all the console commands created in this book.

Another common question is:

Where do *I* register my commands?

Registering is the process of telling Laravel it can use the commands you've created. You can register them in service providers, or you may register them in a *special location* Laravel provides to register your commands (in `app/start/artisan.php`). We'll use this second method a bit later in this part of the book.

But first you might be wondering.

How do I get rid of the commands I don't want.

Easy, let's kill all the artisan commands that won't be needed by our *Getting Stuff Done* application.

Removing default commands

Most of the artisan commands that ship with Laravel are registered by service providers. Since we're not using many aspects of the framework in this book, let's remove them.

When creating Laravel applications I try to make it a practice to remove every unused service provider and alias before the application goes to production.

Removing session commands

Since we're not using sessions in this application, let's remove the `artisan session:table` command.

In your `app/config/app.php`, *comment out* the Session Command Service provider by putting a double-slash (//) before the line. This is in the `providers[]` array.

```
// find the line that says
'Illuminate\Session\CommandsServiceProvider',

// Change it to:
//'Illuminate\Session\CommandsServiceProvider',
```

When you finish issue the `artisan` command again (or `art` if you've aliased it) to make sure the **session** section in the command list is gone.

Removing migrate and queue commands

Let's get rid of all the migrate and queue commands in the same way.

> ### Using Queues
>
> Although we're not using queues in this app, queues are one of the most powerful features of Laravel. Learn them. Use them. You'll be glad you did.

Since we're not using queues or migrations, let's comment out those service providers.

Update your `app/config/app.php` file and find the two lines specified below, comment them out. These two lines are in the `providers[]` array, but probably not adjacent to each other

```
1   // find these lines
2   'Illuminate\Database\MigrationServiceProvider',
3   'Illuminate\Queue\QueueServiceProvider',
4
5   // put comments before them
6   //'Illuminate\Database\MigrationServiceProvider',
7   //'Illuminate\Queue\QueueServiceProvider',
```

Run `artisan` when done to check progress.

Remove key:generate, other database and package commands

Let's see what other commands won't we be using?

We don't need `key:generate` because we already have a key in our app (and we won't be using anything that even needs this key).

We don't need the `db:seed` or `auth:reminders` commands, because we're not using any database.

And, we're not using packages, so let's kick `asset:publish` and `config:publish` to the curb. What else? I can't imagine this application needs a cache, so we'll get rid of the `cache:clear` command too.

Update your `app/config/app.php` again.

```
1   // find these lines
2   'Illuminate\Foundation\Providers\KeyGeneratorServiceProvider',
3   'Illuminate\Database\SeedServiceProvider',
4   'Illuminate\Auth\Reminders\ReminderServiceProvider',
5   'Illuminate\Foundation\Providers\PublisherServiceProvider',
6   'Illuminate\Cache\CacheServiceProvider',
7
8   // Comment them out. I don't really need to provide an example
```

Now, if you do an artisan command, the list is far smaller:

```
~$ cd gsd
~/gsd$ ./artisan
Laravel Framework version 4.0.7

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v|vv|vvv Increase the verbosity of messages: 1 for
  normal output, 2 for more verbose output and 3 for debug
  --version        -V Display this application version.
  --ansi              Force ANSI output.
  --no-ansi           Disable ANSI output.
  --no-interaction -n Do not ask any interactive question.
  --env               The environment the command should run under.

Available commands:
  changes           Display the framework change list
  clear-compiled    Remove the compiled class file
  down              Put the application into maintenance mode
  dump-autoload     Regenerate framework autoload files
  help              Displays help for a command
  list              Lists commands
  optimize          Optimize the framework for better performance
  routes            List all registered routes
  serve             Serve the application on the PHP development server
  tinker            Interact with your application
  up                Bring the application out of maintenance mode
  workbench         Create a new package workbench
command
  command:make      Create a new Artisan command
controller
  controller:make   Create a new resourceful controller
```

Removing the rest

That's much better, but you know want? I think we can get it even smaller.

You can always add them back

Feel free to remove whatever service providers you don't think you'll need. Worst case, if you find you need them, just add them back in. That's why I like to only comment the service provider out and not actually delete them.

Since this is a personal project, not public on the web. I really don't think we need a "maintenance mode". That will get rid of the `down` and `up` commands. I won't be using the `serve` command here (keep this if you're going to use it though). The `tinker` command is beyond the scope of this book. As is the `workbench` command.

I'm keeping the `changes` command, because I like it even though it doesn't help the *Getting Stuff Done* application we're building. I'm planning on using the `command:make` so we'll keep that one too. And I'm not sure about the `controller:make`, so I'll leave it for now.

Remove the rest

```
1   // Here's the remaining lines to comment out
2   'Illuminate\Foundation\Providers\MaintenanceServiceProvider',
3   'Illuminate\Foundation\Providers\ServerServiceProvider',
4   'Illuminate\Foundation\Providers\TinkerServiceProvider',
5   'Illuminate\Workbench\WorkbenchServiceProvider'
```

And now our artisan command output almost fits on one screen.

```
1  ~$ cd gsd
2  ~/gsd$ ./artisan
3  Laravel Framework version 4.0.7
4
5  Usage:
6    [options] command [arguments]
7
8  Options:
9    --help           -h Display this help message.
10   --quiet          -q Do not output any message.
11   --verbose        -v|vv|vvv Increase the verbosity of messages: 1 for
12   normal output, 2 for more verbose output and 3 for debug
13   --version        -V Display this application version.
14   --ansi              Force ANSI output.
15   --no-ansi           Disable ANSI output.
16   --no-interaction -n Do not ask any interactive question.
```

```
  --env              The environment the command should run under.

Available commands:
  changes           Display the framework change list
  clear-compiled    Remove the compiled class file
  dump-autoload     Regenerate framework autoload files
  help              Displays help for a command
  list              Lists commands
  optimize          Optimize the framework for better performance
  routes            List all registered routes
command
  command:make      Create a new Artisan command
controller
  controller:make   Create a new resourceful controller
```

Tight.

Chapter 35 - Planning Our Commands

In This Chapter

In this chapter we plan out, generally, the new commands `artisan` should supply to let us *Get Stuff Done*.

Planning on Planning ... Let's Get Meta

> "No Battle Plan Survives Contact With the Enemy"
>
> *–Helmuth von Moltke, German military strategist*

Good ol' Helmuth. That's true but it's not an excuse to not plan *(even if the plan is only in your head)* because ...

> "In preparing for battle I have always found that plans are useless, but planning is indispensable."
>
> *–Dwight D. Eisenhower, 34th President of the United States*

My take is that when reality hits your plan, the plan falls apart and you have to adapt and change the plan.

Hmm. That's been pretty much the whole book to this point.

So let's plan to plan what the *gsd* commands will look like, but also plan on not using our plan.

I'm good with that.

The List of commands in our application

I'm cheating a bit here by going back to Chapter 24 (Where we mapped out the original Todo facade). I'll just take the comments. They were:

- Create a new list
- Get list of all the lists

- Edit list
- Delete list (I'll archive it instead)
- Add task
- List tasks
- Edit task
- Remove task
- Move task

This will be the basis of our application commands, let's go through them one-by-one.

Create a new list

We need a way from the console to create a new list. Remember a list is a filename and can have a title and even a subtitle associated with it.

I'm thinking:

```
1 $ php artisan gsd:create +name --title="My title" --subtitle="My subtitle"
```

Seems pretty verbose doesn't it? I'm not really liking that aspect, but Laravel provides the ability to have short options. I could have it down to.

```
1 $ art gsd:create +name -t "My title"
```

Of course, what if I accidentally create a list and named it badly. If I have my console window open, I don't want to have to navigate to my gsd directory (`home/chuck/Documents/gsd` in my case). Let's compliment the `gsd:create` command with an uncreate.

```
1 $ art gsd:uncreate +name
```

The reason I put a plus (+) in front of the name is I was thinking that it would be easy to pull out a listname (aka project name, heh) from a string that way. Just like, if you were using contexts in your lists you might be watching for the *at symbol* (@) for @home, @work, etc.

List All Lists

I want to be able to list all the lists. Remember there's going to be multiple todo lists here and it'd be useful to see what they all are.

```
$ art gsd:listall --tasks=all|done|next|normal --archived
```

The `listall` command with no arguments would just list all the todo lists, but only the ones not archived. If we add the `--archived` flag then it would list just the archived lists.

What if I want to list all the tasks across the lists? That's why I added the `tasks` option. The default would be `all`, to list all the types of tasks. But I could list all the completed tasks (`done`), or all the next actions (`next`) or even just the normal tasks (`normal`).

Don't Forget Your Customizations

As you're going through these commands, if you've added features to your version of `Getting Stuff Done`, be sure to update the commands. Maybe you added contexts and want to add a `--context=XXX` option.

After thinking about the `gsd:listall` command, there's one thing I really don't like. That's how long it is. I want to type as few keys as possible. The options can easily be shorted to `-t` for `--tasks` and `-a` for archived, but the command itself ... I want to have a shorter version.

I'll just plan out short aliases and worry about how to implement them later. Here's what I'm thinking with the `gsd:listall`

```
# Full version
$ art gsd:listall --tasks=all|done|next|normal --archived

# Short version
$ art gsd:lsa -t type -a

# Short version just to list all the next actions
$ art gsd:lsna
```

Yeah. I'm liking that better. I can type a short command and get a list of my lists or even everything I can work on next.

Edit List

Editing a list should be straight forward. There's only two "list" attributes we really want to change: title and subtitle.

Already Refactoring the Design ...

Hmmm. Now I'm second guessing what I did earlier. I mean creating a list (and if needed uncreating) is not something that happens very frequently. Maybe I should have both processes prompt for list id, title, subtitle.

(Smoke is coming out of my ears now as I try to imagine which way I like better. Uh. How about both.)

Okay, here's my new rules:

- `gsd:create` will prompt for arguments if none provided.
- `gsd:uncreate` will show a list of lists having no tasks, allowing the user to select which one to *uncreate*.

I like that. Now back to the regularly scheduled planning.

List editing can occur the same way as the create command. If no arguments are passed, then it will prompt for each.

```
# Normal version
$ art gsd:editlist +name --title="title" --subtitle="subtitle"

# Shorter version
$ art gsd:editlist +name -t "title" -s "subtitle"

# Version that prompts user
$ art gsd:editlist
```

I'm not worried about shortening the actual command `gsd:editlist` on this one because I'm not going to be using it as often as `gsd:lsna`, but if you want to have a short version ... please do.

Archive list

Since deciding earlier the `gsd:uncreate` command should show a list that the user can select from, I'm starting to dig that idea for these one-off type tasks. I mean a `gsd:create` won't happen too often. A `gsd:uncreate` will happen almost never. A `gsd:archive` will happen more frequently than the `gsd:uncreate`, but a `gsd:unarchive` won't happen very often.

I want this designed so the things I don't do very often don't have to be remembered. The great thing about hooking into Laravel's `artisan` command is if you type in a command that doesn't have enough arguments it'll tell you what the arguments are.

I can picture myself, merrily programming away, when a new project hits my desk and deciding to create gsd project file. I know the command is `gsd:create`. (Or maybe I don't. Maybe I'm a senile programmer now. So I type `art` to get a list of commands and then see the `gsd:create` command.) Regardless of my senility I'd probably just type `gsd:create` with no arguments to see what happens.

Whew. That's a lot of talking just to get to:

```
# Archive a list, always prompts
$ art gsd:archive

# Unarchive a list, always prompts
$ art gsd:unarchive
```

I don't mind going through a couple, three prompts on these seldom performed commands.

Rename List

This wasn't in my initial List of commands, but before moving onto the task specific commands I tried thinking if there'd be any other list oriented commands. I can imagine a scenario where I want to rename a list.

```
# Rename a list, always prompts
$ art gsd:rename
```

Add a task

Adding a task should be the most often performed command. Following closely by marking a task complete *(which, of course, wasn't in my initial List of commands.)* So I want this to be eeeeaaasssy to use.

```
# Add a task
$ art gsd:add [+name] "something to do" --next

# Short version
$ art gsd:a [+name] "something to do" -n

# Super short version
$ gsd a "something to do"
```

The first version is self-explanatory. The list or project is specified with `+name`. If not specified then the default list will be used. The string specifying the task is passed. This string could have an embedded `:due:date` tag in it. If the `--next` option is used, it specifies the task can be flagged as a next action.

The short version should be obvious too. So what's up with the *Super short version*?

Well, I got to thinking. I'll wrap the artisan command with a shell script. (You Windows users will have to use batch files.). The reasons I'm going to do this are:

- As much as I like typing `art`, it eliminating it does eliminate a wee bit of typing.
- I can put the `gsd` script in my path and access it from whatever directory I'm in.
- I'm not sure how tough it will be to make the `artisan` command accept aliases, such as `gsd:a` for `gsd:add`. The Symfony Command class (which Laravel's command is built on) takes aliases, but I'm not sure how to do it in Laravel.

I could figure the last point, but with the other reasons to use a shell script, I know I'm going to doing it. I'll just handle any aliasing in the shell script.

Makes sense, doesn't it?

I'm tempted to allow multiple tasks to be added at once. Maybe a `gsd a` with no arguments? I'll keep thinking about that. For now, I'll leave the add command as it is.

Another thing I'm tempted to do is have a prompted version of Add a task. Hmmm. Who knows? By the time this part of the book is finished I may decide that would be a nice option.

Marking a task complete

Before I forget about marking a task complete, let me plan it out.

```
1 $ gsd do [+name] n
```

Where *n* is the task number. Where does that come from? The next command.

Listing Tasks

When I list the tasks I want to have a pretty little listing that shows something like:

```
+---+------+-------------------------------------+--------------+
| # | Next | Description                         | Extra        |
+---+------+-------------------------------------+--------------+
| 1 | YES  | Finish Chapter 36                   | Due Sep-20   |
| 2 | YES  | Balance Checkbook                   |              |
| 3 |      | Start project for world domination  |              |
| 4 |      | Read Dr. Evil's Memoirs             |              |
|   | done | Finish Chapter 35                   | Done 9/19/13 |
+---+------+-------------------------------------+--------------+
```

This would be a different listing that the one possibly created by `gsd lsa`, because if we're listing tasks with the listall command, then the name of the list would be displayed. Also, showing a `#` doesn't make sense if the list is archived because we don't want to edit tasks on archived lists.

This version is to list tasks in a single todo list.

Here's the command to do it:

```
# Long version
$ gsd list [+name] --next --nodone

# Short version
$ gsd ls [+name] -n -nd
```

If the `--next` option is used, then it would only list next actions. If the `--nodone` option is used, then it wouldn't list the completed items.

Notice the first column? That's the *n* that is used to reference a particular list item.

Typical Workflow

I'm imagining the typical work flow using GSD would be:

- Add items as they come up
- Maybe move them to other lists
- Use the `gsd ls` command to list tasks.
- Use the `gsd do #` to mark them done.

Edit Task

Here's a command I'd like to work two ways: totally from command arguments or prompted.

```
# Long version
$ gsd edit [+name] 1 "the description" --next[=off]

# Short version
$ gsd ed [+name] 1 "the description" -n[=off]

# Version that prompts
$ gsd ed [+name]
```

Again, any due dates would be embedded into the description.

With the prompting, it occurs to me that maybe I want the user to be able to select from a list of projects when the name's not specified. Yet, I also want the ability to just default to the default list when not specified.

Maybe another configuration option?

Edit the top of your `app/config/app.php` to match the following code. (Keep your folder relevant to your installation, though.)

```
<?php
return array(
  'gsd' => array(
    'folder' => '/home/chuck/Documents/gsd/',
    'extension' => '.txt',
    'listOrder' => array(
      'inbox', 'actions', 'waiting', 'someday', 'calendar',
    ),
    'defaultList' => 'actions',     // default list when not specified
    'noListPrompt' => true,         // true=prompt for list, false=use default
  ),
  // rest of file is same
)?>
```

Remove task

The command to remove a task is easy. We'll have it prompt if the list id is not specified.

```
# Long version
$ gsd:remove [+name] 1 --force

# Short version
$ gsd rm [+name] 1 -f

# Version that prompts
$ gsd rm
```

I added the `--force` option to remove because be default the remove command ought to ask something like *"This will permanently delete: The task description. Are you sure (yes/no)?"* If `--force` is specified, this verification step will be skipped.

How about a –quiet option

Often console utilities will have a `--quiet` option that means do not output anything. With Laravel's artisan command this is built in. As long as we output text in the correct manner, our commands will automatically supress output when needed.

Move Tasks

Our final command will be one to move a task from one list to another.

```
# Long version
$ gsd move [+name] 1 +dest

# Short version
$ gsd mv [+name] 1 +dest

# Prompted version
$ gsd mv
```

Easy. No need to come up with a long explanation for this.

Final List of all Commands

Here's the final list (to this point) of all the console commands

```
# Create list
$ gsd create +name --title="My title" --subtitle="My subtitle"
$ gsd create +name -t "My title"
$ gsd create

# Uncreate list
$ gsd uncreate

# List all lists (possibly tasks too)
$ gsd listall --tasks=all|done|next|normal --archived
$ gsd lsa -t type -a
$ gsd lsna

# Edit list
$ gsd editlist +name --title="title" --subtitle="subtitle"
$ gsd editlist +name -t "title" -s "subtitle"
$ gsd editlist

# Archive/Unarchive/Rename a list
$ gsd archive
$ gsd unarchive
$ gsd rename

# Adding tasks
$ gsd add [+name] "something to do" --next
$ gsd a [+name] "something to do" -n

# Making a task complete
$ gsd do [+name] n

# Listing tasks
$ gsd list [+name] --next --nodone
$ gsd ls [+name] -n -nd

# Edit task
$ gsd edit [+name] 1 "the description" --next[=off]
$ gsd ed [+name] 1 "the description" -n[=off]
$ gsd ed [+name]

# Remove task
$ gsd:remove [+name] 1 --force
$ gsd rm [+name] 1 -f
```

```
$ gsd rm

# Move task
$ gsd move [+name] 1 +dest
$ gsd mv [+name] 1 +dest
$ gsd mv
```

Have I forgot anything? Probably. The only way to find out is to keep moving.

Chapter 36 - Pseudo-coding

In This Chapter

In this chapter we'll write some quick psuedo-code for the commands we defined in the last chapter

Normally, I don't do much pseudo-coding. I tend to lean heavily toward the *jumping in with both feet, coding, and refactoring as I go* methodology. But, because one of my beliefs is that **Every Programmer is Different**, I figured it would be helpful to present a few different techniques. Thus, this chapter is all about Pseudo-Coding.

Pseudo-code
: a notation resembling a simplified programming languaged, used in program design.

There's no **gold standard** in pseudo-code. I generally try to keep my pseudo-code kind of a "comment" on what's we're doing. Then it becomes a very simple process to turn each line of pseudo-code into a comment proceding the few lines of actual code that implement it.

Pseduo-Coding Tip

Try to keep your pseduo-code describing what you're doing, not how you're doing it. In other words, focus on the *problem domain*, not the *implementation domain*.

I'm not going to comment much on the pseudo-code below, only where it seems important that my meaning is clear.

Create List Pseudo-code

```
Get options and arguments
If no options or arguments
   Prompt user for new list-id
   Validate list-id
   Prompt user for list-title
   Prompt user for list-subtitle
Else
   Validate list-id
EndIf
Create new list
Set list-title if needed
Set list-subtitle if need
Save list
```

Uncreate List Pseudo-code

```
Prompt user for list-id
Validate list-id has no tasks
Delete list
```

List all Lists Pseudo-code

```
Get/validate options
Get all list-ids (archived or non-archived)
Sort list-ids
Loop through list-ids
   Load current list-id
   If tasks option
      Output tasks desired
   Else
      Ouput list-id, list-title, and task counts
   EndIf
EndLoop
```

Edit List Pseudo-code

```
Get options and arguments
Prompt for list-id if not specified
Load list
If no arguments
   Prompt user for list-title
   Prompt user for list-subtitle
EndIf
Set list-title if needed
Set list-subtitle if need
Save list
```

Archive List Pseudo-code

```
Prompt user for list-id
If archived version exists
   Warn will overwrite
EndIf
Ask if they're sure
Load existing list
Save list as archive
Delete existing list
```

Unarchive List Pseudo-code

```
Prompt user for archived list-id
If unarchived version exists
   Warn will overwrite
EndIf
Ask if they're sure
Load existing list
Save list as unarchive
Delete existing archived list
```

Rename List Pseudo-code

```
Prompt user for archived or unarchived
Prompt user for appropriate list-id
Prompt user for new list-id
Load existing list
Save as new list-id
Delete existing list
```

Add Task Pseudo-code

```
Get options and arguments
Validate arguments
Prompt for list-id if needed
Load list
Create New task description
Set next action if needed
Add task to list
Save list
```

Do Task Pseudo-code

```
Get arguments
Prompt for list-id if needed
Load list
Prompt for task # if needed
Set task # completed
Save list
```

NOTE: I decided the `gsd do` command should be able to be called without any id which would then prompt for the task #.

Listing Tasks Pseudo-code

```
Get arguments and options
Prompt for list-id if needed
Load list
Loop through tasks
   If task not filtered by option
      show task
   EndIf
EndLoop
```

Edit Task Pseudo-code

```
Get arguments and options
Prompt for list-id if needed
Load list
If no task # specified
   Prompt for task #
   Prompt for new description
   Prompt for Is Next Action
End
Save Task Description
Save NextAction if needed
Save list
```

Remove Task Pseudo-code

```
Get arguments and options
Prompt for list-id if needed
Load list
Prompt for task # if needed
Show warning if not forced
Delete task from list
Save list
```

Move Task Pseudo-code

```
Get arguments and options
Prompt for list-id if needed
Load source list
Prompt for task # if needed
Prompt for dest list-id if needed
Ask if they're sure if not forced
Load task # from source list
Save task in dest list
Save dest list
Delete task # from source list
Save source list
```

NOTE: I decided the `gsd mv` command should also have a `--force` option. This way there's that final "Are you sure?" question if `--force` is ommitted.

Final Thoughts on Pseudo-coding

I know developers that love to pseudo-code. I do it on occassion, but like I said at the beginning of this chapter, it's not my preferred method.

One of the nice benefits of pseudo-coding is that you can see common functionality earlier in the coding process. This allows you to keep your code DRY without excessive refactoring.

And that's the subject of the next chapter ...

Chapter 37 - Using Helper Functions

In This Chapter

In this chapter we'll implement some of the common functions identified during the psuedo-coding of the last chapter. We'll create helper functions to do this.

The Most Frequent Functions

I scanned through the pseudo-code from last chapter, trying to determine which functions are called most frequently. Ignoring the functionality that will easily be implemented by existing `Todo` methods, there're three that jump out at me:

1. Prompt for list-id. This is the most used method. Suprisingly, it's used more than the next function.
2. Get options and arguments. Since artisan handles this quite, uh, handily, I'll ignore it.
3. Prompt for task #. This function is used frequently enough to warrant not repeating ourself.

Since I'm ignoring #2, that means there's two functions to think about.

Creating Helper Functions

So where should these functions go? I can see several possibilities:

1. Expand the `Todo` facade with additional methods
2. Create a `BaseCommand` class which implements these methods. Then our Command classes would extend this class.
3. Stick them in a new class, as static methods.
4. Use helper functions.

Honestly, if I was doing this project just for myself, I'd choose #1 because facades are just so gosh-durn awesome. Without Laravel, I'd choose #2 or maybe #3. But this chapter is supposed to be about #4.

I'd really like to explain how to implement helper functions and thought a few helper functions would shake loose at this point. Yes, I could implement a `prompt_user_for_list()` and `prompt_user_for_task()` helper function but that doesn't seem perfect. (I'm really hung up on implementing thse as part of the `Todo` facade. A `Todo::getListFromUser()` and `Todo::getTaskFromUser()` method seems more elegant to me.

Eureka!

(What happened late last night, before I wrote "Eureka!", is that I went to bed thinking I'd scratch this chapter, end it with the explanation that sometimes I go down one direction, realize I was wrong and back up to start over in a different direction. I planned on either doing that or implement those two functions as helper functions and be done with it. But when I awoke, I realized something obvious.)

> Both these functions have a common task: presenting a list of choices to the user and having the user pick one.

The choices are "which list" or "which task".

Nice, now I can continue on without looking like an idiot. Although, I did tell you what happened so you can draw your own conclusion on my idiotricity.

helper function
: a helper function in Laravel is a support function always available to other functions or methods.

There are three simple steps to implement helpers in your Laravel app.

Step 1 - Create the file `app/src/GSD/helpers.php` with the following content.

```
1  <?php
2
3  function gsd_helper()
4  {
5      return "booyah";
6  }
7  ?>
```

This is just a stupid function we can call to make sure our helpers are loaded.

Step 2 - Update `composer.json`, so it matches what's below.

```
{
  "name": "gsd",
  "description": "Getting Stuff Done with Laravel.",
  "require": {
    "laravel/framework": "4.0.*",
    "mockery/mockery": "dev-master"
  },
  "autoload": {
    "psr-0": {
      "GSD": "app/src"
    },
    "classmap": [
      "app/tests/TestCase.php"
    ],
    "files": [
      "app/src/GSD/helpers.php"
    ]
  },
  // rest of file is the same
```

The `files` section of `autoload` tells composer which files to always load.

Step 3 - Regenerate the autoload files.

This is the `composer dump-autoload` command.

```
~$ cd gsd
~/gsd$ composer dump-autoload
Generating autoload files
```

That's it. All done. Now any function we put in the `app/src/GSD/helpers.php` file will always be loaded and available.

Don't believe me? Hmmm. Well, let's unit test that bad boy then.

Forgetting to do the whole `composer dump-autoload` thing has bitten me in the posterior a few times. I cannot tell you the number of times I can't figure out why something's not working, only to finally realize I neglected to do this step. This is why way back in Chapter 19 I removed all those `app/commands`, `app/controllers`, ... directories and opted to use the PSR-0[32] standard instead. PSR-0 does not require a classmap to find your classes.

[32] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md

Unit Testing Our Helper Functions

Ah. Now maybe it makes sense why I put such a ludicrous function named `gsd_helper()` in the `helpers.php`. I wanted to be able to quickly set up a unit test to prove the helper functions are indeed loaded.

Create the file `HelpersTest.php` in your `app/tests/GSD` directory with the following content.

```
<?php
class HelpersTest extends TestCase {
  public function testGsdHelper()
  {
    $this->assertEquals('booyah', gsd_helper());
  }
}
?>
```

Now, run phpunit to make sure it works

```
~$ cd gsd
~/gsd$ phpunit --tap --filter HelpersTest
TAP version 13
ok 1 - HelpersTest::testGsdHelper
1..1
```

Our test suite is getting a little long, so I filtered it to only run the tests in the `HelpersTest` class. Notice that it worked. Now we can get around to creating our functions and testing them.

Creating pick_from_list()

The two facade methods we'll eventually implement will both will present a list of options to the user and ask for them to pick one. Let's create a basic function that will let the user pick from a list of choices.

Edit `helpers.php` in `app/src/GSD`, delete the existing `gsd_helper()` function and add the `between()` and `pick_from_list()` functions below.

```php
<?php
use Illuminate\Console\Command;

/**
 * Return TRUE if a value is between two other values
 */
function between($value, $min, $max, $inclusive = true)
{
    if ($inclusive)
    {
        return ($value >= $min and $value <= $max);
    }
    return ($value > $min and $value < $max);
}

/**
 * Present a list of choices to user, return choice
 * @param Command $command The command requesting input
 * @param array $choices List of choices
 * @param int $default Default choice (1-array size)
 * @param string $abort String to tag on end for aborting selection
 * @throws InvalidArgumentException If argument is invalid
 */
function pick_from_list(Command $command, $title, array $choices,
  $default = 0, $abort = null)
{
  if ($abort)
  {
    $choices[] = $abort;
  }
  $numChoices = count($choices);
  if ( ! $numChoices)
  {
    throw new \InvalidArgumentException("Must have at least one choice");
  }
  if ($default > $numChoices || $default < 0)
  {
    throw new \InvalidArgumentException("Invalid value, default=$default");
  }
  $question = "Please select 1 to $numChoices";
  if ($default > 0)
  {
```

```
43      $question .= " (default is $default)";
44    }
45    elseif ($default < 0)
46    {
47      $question .= " (enter to abort)";
48    }
49    $question .= '?';
50    while(1)
51    {
52      $command->info($title);
53      for ($i = 0; $i < $numChoices; $i++)
54      {
55        $command->line(($i + 1).". ".$choices[$i]);
56      }
57      $answer = $command->ask($question);
58      if ($answer == '')
59      {
60        $answer = $default;
61      }
62      if (between($answer, 1, $numChoices))
63      {
64        if ($abort and $answer == $numChoices)
65        {
66          $answer = -1;
67        }
68        return (int)$answer;
69      }
70    }
71  }
72  ?>
```

I use the `between()` function all the time. The `pick_from_list()` function is a simple menu picker. Nothing too fancy. Very retro.

The only thing mildly interesting is the usage of the class `Illuminate\Console\Command`. This is the base class of the commands we'll be creating. The class provides methods for getting input from the user and presenting information to them. The `$command->info()` method displays the line in green. The `$command->line()` method outputs the text in normal black and white.

Testing pick_from_list()

Update `HelpersTest.php`, remove the existing test and make the file look like below.

```php
<?php

class HelpersTest extends TestCase {

  /**
   * @expectedException InvalidArgumentException
   */
  public function testPickFromListEmptyArrayThrowsError()
  {
    $command = Mockery::mock('Illuminate\Console\Command');
    pick_from_list($command, 'title', array());
  }

  /**
   * @expectedException InvalidArgumentException
   */
  public function testPickFromListBadDefaultThrowsError()
  {
    $command = Mockery::mock('Illuminate\Console\Command');
    pick_from_list($command, 'title', array('option 1'), -1);
  }

  /**
   * @expectedException InvalidArgumentException
   */
  public function testPickFromListBadDefaultThrowsError2()
  {
    $command = Mockery::mock('Illuminate\Console\Command');
    pick_from_list($command, 'title', array('option 1'), 2);
  }

  public function testPickFromListWorksExample1()
  {
    $command = Mockery::mock('Illuminate\Console\Command');
    $command->shouldReceive('info')->once();
    $command->shouldReceive('line')->times(2);
    $command->shouldReceive('ask')->once()->andReturn(1);
    $choice = pick_from_list($command, 'title', array('option'));
    $this->assertEquals(1, $choice);
  }

  public function testPickFromListWorksExample2()
```

```
43    {
44      $command = Mockery::mock('Illuminate\Console\Command');
45      $command->shouldReceive('info')->once();
46      $command->shouldReceive('line')->times(3);
47      $command->shouldReceive('ask')->once()->andReturn(2);
48      $choice = pick_from_list($command, 'title', array('option 1', 'opt2'));
49      $this->assertEquals(2, $choice);
50    }
51
52    // First time through loop user selects bad choice, causing a second loop
53    public function testPickFromListWorksExample3()
54    {
55      $command = Mockery::mock('Illuminate\Console\Command');
56      $command->shouldReceive('info')->times(2);
57      $command->shouldReceive('line')->times(4);
58      $command->shouldReceive('ask')->times(2)->andReturn('x', 1);
59      $choice = pick_from_list($command, 'title', array('option'));
60      $this->assertEquals(1, $choice);
61    }
62
63    public function testPickFromListWorksWithDefault()
64    {
65      $command = Mockery::mock('Illuminate\Console\Command');
66      $command->shouldReceive('info')->once();
67      $command->shouldReceive('line')->times(3);
68      $command->shouldReceive('ask')->once()->andReturn('');
69      $choice = pick_from_list($command, 'title', array('option 1', 'opt2'), 2);
70      $this->assertEquals(2, $choice);
71    }
72
73    public function testPickFromListWorksWithAbort()
74    {
75      $command = Mockery::mock('Illuminate\Console\Command');
76      $command->shouldReceive('info')->once();
77      $command->shouldReceive('line')->times(3);
78      $command->shouldReceive('ask')->once()->andReturn(2);
79      $choice = pick_from_list($command, 'title', array('option'), 0, "Abort");
80      $this->assertEquals(-1, $choice);
81    }
82
83  }
84  ?>
```

I didn't put a unit test for `between()` because, well, it's only three lines of code and I know it works.

Run phpunit to test this.

```
~$ cd gsd
~/gsd$ phpunit --tap --filter HelpersTest
TAP version 13
ok 1 - HelpersTest::testPickFromListEmptyArrayThrowsError
ok 2 - HelpersTest::testPickFromListBadDefaultThrowsError
ok 3 - HelpersTest::testPickFromListBadDefaultThrowsError2
ok 4 - HelpersTest::testPickFromListWorksExample1
ok 5 - HelpersTest::testPickFromListWorksExample2
ok 6 - HelpersTest::testPickFromListWorksExample3
ok 7 - HelpersTest::testPickFromListWorksWithDefault
ok 8 - HelpersTest::testPickFromListWorksWithAbort
1..8
```

Nice. We're getting close to actually implementing our first console command.

Chapter 38 - The ListAllCommand

In This Chapter

In this chapter we'll implement part of the `gsd:listall` command.

Hold onto your hats folks, this chapter is gonna be fun!

The Plan

Back in Chapter 35, we planned the `gsd:listall` command. Our variations were:

```
$ gsd listall --tasks=all|done|next|normal --archived
$ gsd lsa -t type -a
$ gsd lsn
```

I'm not going to worry about the `--tasks` option yet. Let's just get the command done so we can list out any lists we have.

The pseudo-code we came up with in Chapter 36 was:

```
Get/validate options
Get all list-ids (archived or non-archived)
Sort list-ids
Loop through list-ids
   Load current list-id
   If tasks option
      Output tasks desired
   Else
      Ouput list-id, list-title, and task counts
   EndIf
EndLoop
```

Creating the ListAllCommand

I know I went on earlier about how much Laravel wants you to succeed, about how Laravel likes you. Creating commands is another area where Laravel looks out for you and tries to make your life easier. One simple `artisan` command will create a new console command for you.

Create the `ListAllCommand` with the line below.

```
~/gsd$ art command:make ListAllCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

This creates a new file in `app/src/GSD/Commands` named `ListAllCommand.php`. It's the skeleton for the command we're building.

Edit the newly created `ListAllCommand.php` to make it match what's below.

```
<?php namespace GSD\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;

class ListAllCommand extends Command {

  /**
   * The console command name.
   *
   * @var string
   */
  protected $name = 'gsd:listall';

  /**
   * The console command description.
   *
   * @var string
   */
```

```
21    protected $description = 'Lists all todo lists (and possibly tasks).';
22
23    /**
24     * Create a new command instance.
25     *
26     * @return void
27     */
28    public function __construct()
29    {
30      parent::__construct();
31    }
32
33    /**
34     * Execute the console command.
35     *
36     * @return void
37     */
38    public function fire()
39    {
40      $this->line('Stick em up, pardner!');
41    }
42
43    /**
44     * Get the console command arguments.
45     *
46     * @return array
47     */
48    protected function getArguments()
49    {
50      return array();
51    }
52
53    /**
54     * Get the console command options.
55     *
56     * @return array
57     */
58    protected function getOptions()
59    {
60      return array(
61        array('archived', 'a', InputOption::VALUE_NONE,
62          'use archived lists?'),
```

```
63      );
64    }
65  }
66  ?>
```

You should have only had to change a few lines from the shell that Laravel created for you.

Just a few notes on this:

- The `$name` and `$description` are self explanatory.
- The `fire()` method is what gets executed when you issue the command from the console. Here we're using the `line()` method to output a line to the console.
- The only option we're worried about right now is the `--archived` option. The second array element is the short version (`-a`). For the third element we used the `InputOption::VALUE_NONE` constant to indicate this option has no additional expected value. (Options like `--option=bla` expect values).

Issue the artisan command to see your list of commands

```
1  ~/gsd$ art
```

Do you see the new `gsd:listall` command? Hah! Gotcha. You won't see it yet because artisan doesn't know about it ... yet.

Telling Artisan About the ListAllCommand

Edit the `artisan.php` file in your `app/start` directory and add the following line.

```
1  Artisan::add(new GSD\Commands\ListAllCommand);
```

Now if you issue the artisan command again it'll show something like:

```
~/gsd$ art
Laravel Framework version 4.0.7

[I snipped a few lines]

Available commands:
  changes           Display the framework change list
  clear-compiled    Remove the compiled class file
  dump-autoload     Regenerate framework autoload files
  help              Displays help for a command
  list              Lists commands
  optimize          Optimize the framework for better performance
  routes            List all registered routes
command
  command:make      Create a new Artisan command
controller
  controller:make   Create a new resourceful controller
gsd
  gsd:listall       Lists all todo lists (and possibly tasks).
```

Right there on the bottom. Awesome. Try running it.

```
~/gsd$ art gsd:listall
Stick em up, pardner!
```

He he he, ha, ho, heh, heh. Sometimes all this power makes me giddy.

Fleshing out the fire() method a bit

Update the `ListAllCommand.php` file and make the `fire()` method look like what's below.

```
  public function fire()
  {
    $archived = $this->option('archived');
    $title = 'Listing all ';
    if ($archived) $title .= 'archived ';
    $title .= 'lists';
    $this->info($title);

    $lists = \Todo::allLists($archived);
    print_r($lists);
  }
```

Here we're setting the `$archived` variable to true or false, depending on whether the `--archived` option is used. Next we're outputting a title that will say either "Listing all lists" or "Listing all archived lists". Using the command's `info()` method will make this appear a nice pretty green.

Next, we're fetching all the lists using the `Todo` facade. *(Note the backslash, that's needed because we're namespaced. You could also put a `use Todo;` at the top of the file and get by without the backslash.)*

Finally, we dump the list out using `print_r()`. I figured there was enough to discuss with those few lines of code.

Let's give it a try. Run the command.

```
~/gsd$ art gsd:listall
Listing all lists

  [ReflectionException]
  Class TodoRepositoryInterface does not exist

gsd:listall [-a|--archived]
```

What the ...? Crap. There's a bug that made it through the last part of the book. Uggh. I'm not feeling so giddy now. Sheesh. You start to feel pretty good, like you're invincible and WHAM! Laravel brings you back down to earth. Not that I blame Laravel. I know I was getting a big head there for a moment and Laravel only wants what's best for me and thought it was time for a reality check.

The bug is in the `TodoManager.php` code. I forgot some namespacing.

Edit the `TodoManager.php` file as follows

1. Search for all instances of `App::make('TodoRepositoryInterface')`
2. Replace that with `App::make('GSD\Repositories\TodoRepositoryInterface')`

There should be three replacements.

In the same file, make the following replacement.

1. Search for `App::make('ListInterface')`
2. Replace it with `App::make('GSD\Entities\ListInterface')`

There should only be one replacement

To be safe, let's run phpunit.

```
~/gsd$ phpunit --tap
TAP version 13
ok 1 - ExampleTest::testBasicExample
ok 2 - TaskCollectionTest::testAddFromClassWorks
[snipped about 40 lines]
ok 43 - TodoManagerTest::testFacade
not ok 44 - Failure: TodoManagerTest::testMakeListThrowsExceptionWhenExists
  ---
  message: 'Failed asserting that exception of type "ReflectionException"
  matches expected exception "InvalidArgumentException". Message was: "Class
  ListInterface does not exist".'
  severity: fail
  ...
ok 45 - TodoManagerTest::testMakeList
[snipped the rest]
```

Dang this still isn't working. What's going on? At this point I'm going to need to dig into the code and do some serious debugging. Why don't you go out. Grab a bite to eat. Maybe catch a movie. When you get back I'll have it figured out.

...

Back so soon? Well, I found a couple problems.

The first is more of the namespacing issue.

Make the following changes to `TodoManagerTest.php`

1. Search for all instances of `App::bind('TodoRepositoryInterface`
2. Replace with `App::bind('GSD\Repositories\TodoRepositoryInterface`

There should be 6 replacements.

In the same file `TodoManagerTest.php` make the following changes.

1. Search for all instances of `App::bind('ListInterface'`
2. Replace with `App::bind('GSD\Entities\ListInterface'`

Only one replacement this time.

Finally, in the same file `TodoManagerTest.php`, add the following method at the top of the class

```
1   public function tearDown()
2   {
3     $this->refreshApplication();
4   }
```

This last change hacks in a fix that's a bit more insidious than forgetting the namespacing. The problem is that some of the unit tests `bind()` interfaces to mock objects. Well, what happens is that from test to test, the **IoC** container is not rebuilt, the result is when we create new instances of bound names, we get the mock objects. So using the `refreshApplication()` method will rebuild the **IoC** container (setting $app back to the defaults).

I say this is a hack, because I only fixed it this one place. Ideally, the application's state should be the same after the test as it was before.Everywhere a test does a `bind()` it should undo what it did at the end. Even though the unit tests pass now, let's fix it completely through the unit tests.

To make sure the **IoC** container is refreshed any time we do a `bind()` add the same `tearDown()` method to the top of the `TodoList` class in `TodoListTest.php`

This is still a bit *hacky*, because we're not complementing each and every `bind()` with a corresponding reset. But it's good enough for now.

```
  public function tearDown()
  {
    $this->refreshApplication();
  }
```

Sorry about that side track. I'm really not feeling the least bit giddy now. Thanks Laravel.

Let's try the command again

```
~/gsd$ art gsd:listall
Listing all lists


  [RuntimeException]
  Directory doesn't exist: /home/chuck/Documents/gsd/

gsd:listall [-a|--archived]
```

Okay, that error makes sense because I don't have my folder for lists set up. That's fine. Let's use the data in the directory we set up for unit tests.

Try the command with `--env=testing`

```
~/gsd$ art gsd:listall --env=testing
Listing all lists
Array
(
    [0] => test-one
    [1] => test1
    [2] => test2
)
~/gsd$ art gsd:listall --env=testing --archived
Listing all archived lists
Array
(
    [0] => test2
)
```

Looks like it's working. That `--env` command is automatically built into every artisan command is nice.

Let's make the output better than `print_r()`.

Using Symfony's Table Helper

Since the Laravel `Illuminate\Console\Command` class is built from the `Symfony\Component\Console\Command` class, we can tap into the power of Symfony. One of the features Symfony provides is command helpers. Let's use the table helper to output what we want.

Update your `fire()` method of `ListAllCommand.php` to match what's below. Also, add the `sortListIds()` stub method.

```
<?php
  public function fire()
  {
    $archived = $this->option('archived');
    $title = 'Listing all ';
    if ($archived) $title .= 'archived ';
    $title .= 'lists';
    $this->info($title);

    $lists = \Todo::allLists($archived);
    $lists = $this->sortListIds($lists);

    $headers = array('list', 'next', 'todos', 'completed');
    $rows = array();
    foreach ($lists as $listId)
    {
      $list = \Todo::get($listId, $archived);
      $rows[] = array(
        $listId,
        $list->taskCount('next'),
        $list->taskCount('todo'),
        $list->taskCount('done'),
      );
    }

    // Output a pretty table
    $table = $this->getHelperSet()->get('table');
```

```
28      $table
29        ->setHeaders($headers)
30        ->setRows($rows)
31        ->render($this->getOutput());
32
33    }
34
35    /**
36     * Sort the list ids
37     */
38    protected function sortListIds(array $listIds)
39    {
40      return $listIds;
41    }
42  ?>
```

We're loading each list, tallying up the data, stashing the data in a `$rows[]` array. Then where getting Symfony's table helper, telling it the headers and rows and to render the table.

How does it look?

Test the `gsd:listall` command from the console.

```
1  ~/gsd$ art gsd:listall --env=testing
2  Listing all lists
3  +----------+------+-------+-----------+
4  | list     | next | todos | completed |
5  +----------+------+-------+-----------+
6  | test-one | 2    | 2     | 2         |
7  | test1    | 2    | 2     | 2         |
8  | test2    | 3    | 3     | 3         |
9  +----------+------+-------+-----------+
10 ~/gsd$ art gsd:listall --env=testing --archived
11 Listing all archived lists
12 +-------+------+-------+-----------+
13 | list  | next | todos | completed |
14 +-------+------+-------+-----------+
15 | test2 | 3    | 3     | 3         |
16 +-------+------+-------+-----------+
```

I'm still using the testing data which is left over from the last time we perform unit tests. The output looks great. There's only two problems:

1. All the numbers for next, todos, and completed are the same. That's because the `taskCount()` method always returns the total number of tasks in the list.
2. The lists aren't sorted. Yeah, we haven't implemented the `sortListIds()` method yet.

Let's fix those issues and wrap up this chapter.

Refactoring taskCount()

The goal here is to add an option argument to this method that will count certain types of tasks. We want to do this in such a way that any existing code won't break, so we'll have the default be **all**.

Update the `ListInterface::taskCount()` method's definition.

```
1   /**
2    * Return number of tasks
3    * @param string $type Either 'all', 'done', 'todo', or 'next'
4    * @return integer
5    */
6   public function taskCount($type = 'all');
```

And update the implementation in `TodoList.php`.

```
1   <?php
2     /**
3      * Return number of tasks
4      * @param string $type Either 'all', 'done', 'todo', or 'next'
5      * @return integer
6      */
7     public function taskCount($type = 'all')
8     {
9       $count = 0;
10      foreach ($this->tasks->getAll() as $task)
11      {
12        switch($type)
13        {
14          case 'done':
```

```
15            if ($task->isComplete())
16            {
17              $count++;
18            }
19            break;
20          case 'todo':
21            if ( ! $task->isComplete() and ! $task->isNextAction())
22            {
23              $count++;
24            }
25            break;
26          case 'next':
27            if ( ! $task->isComplete() and $task->isNextAction())
28            {
29              $count++;
30            }
31            break;
32          default:
33            $count++;
34        }
35      }
36      return $count;
37    }
38  ?>
```

Okay, maybe not the greatest feat of software engineering, but hey, it's getting the job done.

Sorting the List ids

Remember way back in Chapter 23 when we added the `gsd.listOrder` array to the config? Let's use that now to sort the list.

Basically, if any of the list ids match an entry in the array, it will be sorted in the order of the array, anything else will come afterward alphabetically.

Update the `sortListIds()` method in `ListAllCommand.php` to match below.

```php
<?php
  /**
   * Sort the list ids
   */
  protected function sortListIds(array $listIds)
  {
    // Pull the names
    $special = array();
    foreach (\Config::get('app.gsd.listOrder') as $name)
    {
      $special[$name] = false;
    }

    // Peel off the specials
    $tosort = array();
    foreach ($listIds as $listId)
    {
      if (array_key_exists($listId, $special))
      {
        $special[$listId] = true;
      }
      else
      {
        $tosort[] = $listId;
      }
    }

    // Put the specials first then sort the remaining and add them in
    $return = array();
    foreach ($special as $listId => $flag)
    {
      if ($flag)
      {
        $return[] = $listId;
      }
    }
    natcasesort($tosort);
    return array_merge($return, $tosort);
  }
?>
```

I'm not going to bore you with an explanation because I told you before the code what my intention was. Hopefully, it'll work.

I'll leave it to you to create a unit test it if you want. *(Hint: to unit test this method you'll probably want to make it `public` instead of `protected`.)*

Run the `gsd:listall` command again to see if it looks like it's working.

```
~/gsd$ art gsd:listall --env=testing
Listing all lists
+----------+------+-------+-----------+
| list     | next | todos | completed |
+----------+------+-------+-----------+
| test-one | 0    | 1     | 1         |
| test1    | 0    | 1     | 1         |
| test2    | 1    | 2     | 0         |
+----------+------+-------+-----------+
~/gsd$ art gsd:listall --env=testing --archived
Listing all archived lists
+-------+------+-------+-----------+
| list  | next | todos | completed |
+-------+------+-------+-----------+
| test2 | 1    | 2     | 0         |
+-------+------+-------+-----------+
```

Seems okay to me.

> Sorry I got sidetracked on finding a bug in this chapter. That wasn't the intention at the chapter's beginning.
>
> Sometimes programming is all about the bug hunt.

Chapter 39 - The CreateCommand

In This Chapter

In this chapter we'll implement the `gsd:create` command.

I'm not going to declare this chapter is going to be fun. I did that last chapter and what happened?–I uncovered some ugly bugs.

I will say this chapter *should* be fun. But at the moment I'm a bit gun-shy about being overly exuberant.

The Plan

Back in Chapter 35, we planned the `gsd:create` command. Our variations were:

```
$ gsd create +name --title="My title" --subtitle="My subtitle"
$ gsd create +name -t "My title" -s "My subtitle"
$ gsd create
```

The last option will prompt for name, title and subtitle.

The pseudo-code we came up with in Chapter 36 was:

```
Get options and arguments
If no options or arguments
   Prompt user for new list-id
   Validate list-id
   Prompt user for list-title
   Prompt user for list-subtitle
Else
   Validate list-id
EndIf
Create new list
Set list-title if needed
Set list-subtitle if need
Save list
```

Creating the CreateCommand

Let's create the command shell with artisan

```
~/gsd$ art command:make CreateCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Edit the newly created `CreateCommand.php` in your `app/src/GSD/Commands` directory and make it match what's below.

```
<?php namespace GSD\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class CreateCommand extends CommandBase {

  /**
   * The console command name.
   *
   * @var string
   */
  protected $name = 'gsd:create';

  /**
   * The console command description.
   *
   * @var string
   */
  protected $description = 'Create new list.';

  /**
   * Create a new command instance.
   *
```

```
27      * @return void
28      */
29     public function __construct()
30     {
31       parent::__construct();
32     }
33
34     /**
35      * Execute the console command.
36      *
37      * @return void
38      */
39     public function fire()
40     {
41       //
42     }
43
44     /**
45      * Get the console command arguments.
46      *
47      * @return array
48      */
49     protected function getArguments()
50     {
51       return array(
52         array('+name', InputArgument::OPTIONAL, 'List name to create'),
53       );
54     }
55
56     /**
57      * Get the console command options.
58      *
59      * @return array
60      */
61     protected function getOptions()
62     {
63       return array(
64         array('title', 't', InputOption::VALUE_REQUIRED,
65           'Title of list.', null),
66         array('subtitle', 's', InputOption::VALUE_REQUIRED,
67           'Subtitle of list.', null),
68       );
```

```
69    }
70  }
71  ?>
```

Did you catch that `CommandBase` that we're extending this class from? I want to be able to add routines to a common parent, so we're setting the structure up right off to allow that.

I know, I talked about implementing things in the `Todo` facade. But I realized this is a perfectly fine way to *Get Stuff Done* and it's a method I haven't used in this book.

Also, I added the `use Todo` at the top. It's not being used, but I know I'm going to use our cool Todo facade in this implementation.

Create `CommandBase.php` in your `app/src/GSD/Commands` directory with the following content.

```
1  <?php namespace GSD\Commands;
2
3  use Illuminate\Console\Command;
4
5  class CommandBase extends Command {
6
7  }
8  ?>
```

Nothing in it yet. But we know it's there.

Finally, link the command in by editing `app/start/artisan.php` to match what's below.

```
1  <?php
2
3  Artisan::add(new GSD\Commands\ListAllCommand);
4  Artisan::add(new GSD\Commands\CreateCommand);
5
6  ?>
```

Here we removed the comments, left the `ListAllCommand`, and added the `CreateCommand`.

Now let's check that the command's available.

```
~/gsd$ art
Laravel Framework version 4.0.7

[snipping everything until the gsd commands]

gsd
  gsd:create        Create new list.
  gsd:listall       Lists all todo lists (and possibly tasks).
```

Great! The skeleton's in place, let's implement it.

Please Note

From this point forward I'm going to be working with live todo lists. So set up whatever folder that is specified in your `app/config/app.php`. (Don't forget th archived subdirectory.)

In my case, I'm doing the following.

```
~/gsd$ mkdir /home/chuck/Documents/gsd
~/gsd$ mkdir /home/chuck/Documents/gsd/archived
```

(Change the path above to be relevant for your installation.)

Adding the all_null() Helper

As I started coding the implementation, I realized there would be a handy function that could check multiple arguments and return true if they're all null. So let's implement the helper and write a unit test for it.

Add the following function to the top of your `helpers.php`. Remember, this file is in `src/GSD` and is always loaded.

```php
/**
 * Return TRUE if every arg is null.
 * @usage all_null(arg1, arg2, ...)
 * @return bool
 */
function all_null()
{
  foreach (func_get_args() as $arg)
  {
    if ( ! is_null($arg))
    {
      return false;
    }
  }
  return true;
}
```

Add the following method to the end of the class in `HelpersTest.php`

```php
  public function testAllNull()
  {
    $this->assertTrue(all_null());
    $this->assertTrue(all_null(null));
    $this->assertTrue(all_null(null, null, null, null));
    $this->assertFalse(all_null(0));
    $this->assertFalse(all_null(null, null, '', null));
    $this->assertFalse(all_null(null, null, null, 33));
  }
```

Do the unit tests

```
~/gsd: phpunit --tap --filter HelpersTest
TAP version 13
ok 1 - HelpersTest::testPickFromListEmptyArrayThrowsError
ok 2 - HelpersTest::testPickFromListBadDefaultThrowsError
ok 3 - HelpersTest::testPickFromListBadDefaultThrowsError2
ok 4 - HelpersTest::testPickFromListWorksExample1
ok 5 - HelpersTest::testPickFromListWorksExample2
ok 6 - HelpersTest::testPickFromListWorksExample3
ok 7 - HelpersTest::testPickFromListWorksWithDefault
ok 8 - HelpersTest::testPickFromListWorksWithAbort
ok 9 - HelpersTest::testAllNull
1..9
```

Okay, it works. If you do `phpunit` without any options you'll see we're up to 100 assertions. Yeah.

Expanding CommandBase

Now, let's flesh out CommandBase

Update `CommandBase.php` to match what's below.

```php
<?php namespace GSD\Commands;

use App;
use Illuminate\Console\Command;

class CommandBase extends Command {

  protected $repository;

  /**
   * Constructor
   */
  public function __construct()
  {
    parent::__construct();
    $this->repository = App::make('GSD\Repositories\TodoRepositoryInterface');
  }

```

```
19    /**
20     * Prompt the user for a list id
21     * @param bool $existing Prompt for existing list or new list?
22     * @param bool $allowCancel Allow user to cancel
23     * @param bool $archived Use archived list?
24     * @return mixed string list id or null if user cancels
25     */
26    public function askForListId($existing = true, $allowCancel = true,
27      $archived = false)
28    {
29      if ($existing)
30      {
31        throw new \Exception('existing not done');
32      }
33
34      $prompt = 'Enter name of new list';
35      if ($allowCancel) $prompt .= ' (enter to cancel)';
36      $prompt .= '?';
37      while(true)
38      {
39        if ( ! ($result = $this->ask($prompt)))
40        {
41          if ($allowCancel)
42          {
43            return null;
44          }
45          $this->outputErrorBox('You must enter something');
46        }
47        else if ($this->repository->exists($result, $archived))
48        {
49          $this->outputErrorBox("You already have a list named '$result'");
50        }
51        else
52        {
53          return $result;
54        }
55      }
56    }
57
58    /**
59     * Output an error box
60     * @param string $message The message
```

```
61      */
62    protected function outputErrorBox($message)
63    {
64      $formatter = $this->getHelperSet()->get('formatter');
65      $block = $formatter->formatBlock($message, 'error', true);
66      $this->line('');
67      $this->line($block);
68      $this->line('');
69    }
70  }
71  ?>
```

Okay, let's go over this real quick.

Constructor - we inject the `$repository` into the class using whatever the binding is for the interface. This is done because the repository will be useful to classes built upon `CommandBase`.

askForListId() - this method can be used when we want to prompt the user for a list id. If `$existing` is set, then we'll prompt for an existing list (but this isn't yet implemented). If the `$existing` flag is false, then we'll get a list id that doesn't yet exist.

outputErrorBox() - this is used by `askForListId()` for outputting a big fat red box to make the error very apparent to the user. I'm hooking into Symfony's Format Helper to build the box.

Implementing fire()

Update `CreateCommand.php` and replace the `fire()` method with what's below.

```
1   <?php
2     /**
3      * Execute the console command.
4      *
5      * @return void
6      */
7     public function fire()
8     {
9       // Get options and arguments
10      $name = $this->argument('+name');
11      $title = $this->option('title');
12      $subtitle = $this->option('subtitle');
```

```
13
14      // Prompt for everything
15      if (all_null($name, $title, $subtitle))
16      {
17        if ( ! ($name = $this->askForListId(false, true)))
18        {
19          $this->outputErrorBox('*aborted*');
20          exit;
21        }
22        $title = $this->ask("Enter list title (enter to skip)?");
23        $subtitle = $this->ask("Enter list subtitle (enter to skip)?");
24      }
25
26      // Validate arguments
27      else if (is_null($name))
28      {
29        throw new \InvalidArgumentException(
30          'Must specify +name if title or subtitle used');
31      }
32      else if ($name[0] != '+')
33      {
34        throw new \InvalidArgumentException(
35          'The list name must begin with a plus (+)');
36      }
37      else
38      {
39        $name = substr($name, 1);
40        if ($this->repository->exists($name))
41        {
42          throw new \InvalidArgumentException(
43            "The list '$name' already exists");
44        }
45      }
46
47      // Create the list, defaulting title if needed
48      $title = ($title) ? : ucfirst($name);
49      $list = Todo::makeList($name, $title);
50
51      // Set the subtitle if needed
52      if ($subtitle)
53      {
54        $list->set('subtitle', $subtitle)->save();
```

```
55      }
56
57      $this->info("List '$name' successfully created");
58    }
59  ?>
```

Yeah! I'm not going to explain the implementation because it's well documented, it follows our pseudo-code fairly close, and there's no magic there.

What about unit tests?

You could set up unit tests for this, but I'm not going to. Many of the underlying classes have been unit tested. And at this level, we've moved up the testing spectrum from Unit Tests to Integration Tests. Unfortunately, Integration Testing is outside the scope of this book.

Chapter 40 - The UncreateCommand

In This Chapter

In this chapter we'll implement the `gsd:uncreate` command.

I've been looking forward to this chapter since Chapter 37. Not because uncreating a list is going to be such a fantastic bit of coding. No, I've been looking forward to it because finally we'll get to use that `pick_from_list()` helper function we created.

The Plan

Back in Chapter 35, we planned the `gsd:uncreate` command. It was simple.

```
$ gsd uncreate
```

Uncreate would be used so seldom that I decided to always let the user pick from a list to uncreate.

The pseudo-code we came up with in Chapter 36 was:

```
Prompt user for list-id
Validate list-id has no tasks
Delete list
```

Should be a snap to implement.

Creating the UncreateCommand

Let's create the command shell with artisan

```
~/gsd$ art command:make UncreateCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Edit the newly created `UncreateCommand.php` in your `app/src/GSD/Commands` directory and make it match what's below.

```php
<?php namespace GSD\Commands;

use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class UncreateCommand extends CommandBase {

  /**
   * The console command name.
   *
   * @var string
   */
  protected $name = 'gsd:uncreate';

  /**
   * The console command description.
   *
   * @var string
   */
  protected $description = 'Destroy an empty list.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Prompt user for list-id
    if ( ! ($name = $this->askForListId(true, true)))
    {
      $this->outputErrorBox('*aborted*');
```

```
34        exit;
35      }
36
37      // Validate list has no tasks
38      $list = Todo::get($name);
39      if ($list->taskCount() > 0)
40      {
41        throw new \UnexpectedValueException(
42          'Cannot uncreate a list with tasks');
43      }
44
45      // Delete list
46      if ( ! $this->repository->delete($name))
47      {
48        throw new \RuntimeException("Repository couldn't delete list '$name'");
49      }
50      $this->info("The list '$name' is now in the big bitbucket in the sky");
51    }
52
53    /**
54     * Get the console command arguments.
55     *
56     * @return array
57     */
58    protected function getArguments()
59    {
60      return array();
61    }
62
63    /**
64     * Get the console command options.
65     *
66     * @return array
67     */
68    protected function getOptions()
69    {
70      return array();
71    }
72  }
73  ?>
```

Pretty simple implementation. We're extending `CommandBase`. There's no options or arguments to this command. And check out the comments in the `fire()` method. Hey, they're pretty much the

pseudo-code we started with.

Also, I deleted the constructor because it wasn't doing anything.

In fact. Go into `CreateCommand.php` and `ListAllCommand.php` and delete the constructors. They aren't needed.

I used to work with this guy ... he was in sales and didn't understand much about software. Yet, somehow he sold it. Often we'd be talking about the work that needed to be done, integrating one part of the system with another, and he'd say "Just bolt 'em together." APIs? He didn't care about that stuff, "Just bolt 'em together." I'd roll my eyes thinking this guy has no clue.

Anyway, artistan needs to know about our new `gsd:uncreate` command so let's "Just bolt 'em together."

Edit `app/start/artisan.php` to match what's below.

```
<?php

Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\UncreateCommand);

?>
```

Now let's check that the command's available.

```
~/gsd$ art
Laravel Framework version 4.0.7

[snipping everything until the gsd commands]

gsd
  gsd:create        Create new list.
  gsd:listall       Lists all todo lists (and possibly tasks).
  gsd:uncreate      Destroy an empty list.
```

Of course the commands there. You knew it would be, didn't you?

Getting Stack Traces on Your Command

Try to run our new command.

```
~/gsd$ art gsd:uncreate

  [Exception]
  existing not done

gsd:uncreate
```

Of course. The `CommandBase::askForListId()` was not finished. But what if we couldn't remember that? Easy ... use the `-v` option with the artisan command.

Try to run the command with `-v`

```
~/gsd$ art gsd:uncreate -v

  [Exception]
  existing not done

Exception trace:
 () at .../app/src/GSD/Commands/CommandBase.php:30
 GSD\Commands\CommandBase->askForListId() at
  .../app/src/GSD/Commands/UncreateCommand.php:31
 GSD\Commands\UncreateCommand->fire() at
  .../framework/src/Illuminate/Console/Command.php:108
 Illuminate\Console\Command->execute() at
  .../Symfony/Component/Console/Command/Command.php:244
 Symfony\Component\Console\Command\Command->run() at
  .../framework/src/Illuminate/Console/Command.php:96
 Illuminate\Console\Command->run() at
  .../Symfony/Component/Console/Application.php:897
 Symfony\Component\Console\Application->doRunCommand() at
  .../Symfony/Component/Console/Application.php:191
 Symfony\Component\Console\Application->doRun() at
```

```
21    .../console/Symfony/Component/Console/Application.php:121
22   Symfony\Component\Console\Application->run() at
23     /home/chuck/gsd/artisan:59
24
25  gsd:uncreate
```

Nice. Now I know to look at line 30 of `CommandBase.php` to figure out why the exception was thrown.

(I cleaned up the stack trace above, shortening paths, adding the ... because I wanted the text to fit better on smaller screens.)

Implementing askForListId() for existing lists

Add `use Todo;` in the top area of `CommandBase.php`, then update the `fire()` method to match what's below.

```
1   <?php
2     /**
3      * Prompt the user for a list id
4      * @param bool $existing Prompt for existing list or new list?
5      * @param bool $allowCancel Allow user to cancel
6      * @return mixed string list id or null if user cancels
7      */
8     public function askForListId($existing = true, $allowCancel = true,
9       $archived = false)
10    {
11      if ($existing)
12      {
13        $title = 'Choose which list to destroy:';
14        $abort = 'cancel - do not destroy a list';
15        $choices = Todo::allLists();
16        if (count($choices) == 0)
17        {
18          throw new \RuntimeException('No lists to choose from');
19        }
20        $result = pick_from_list($this, $title, $choices, 0, $abort);
21        if ($result == -1)
22        {
23          return null;
24        }
```

```
25          return $choices[$result-1];
26      }
27      // rest of file unchanged
28  ?>
```

Now what happens when you run the `gsd:uncreate` command?

```
1  ~/gsd$ art gsd:uncreate
2  Choose which list to destroy:
3  1. test-1
4  2. test-10
5  3. test-2
6  4. waiting
7  5. cancel - do not destroy a list
8  Please select 1 to 5?
```

(Obvoiusly, your choices will appear different.)

It actually works! Nice. Play around with it and create lists, then uncreate them. Test out some of the possibilities.

Everything's working great for me ... but, there's a couple things I want to clean up.

A Little Cleanup

First, a housekeeping task. When I was implementing the code above in `askForListId()` I thought `pick_from_list()` would return `null` if the abort choice was selected. Nope. It returns `-1`. That's fine, but it's not documented in the docblock. Let's fix that.

Also, a few cosmetic changes. I want a blank line after the menu's title and a blank line before the prompt. And instead of the prompt 'Please select 1 to X?', I want it to say 'Please enter a number between 1 and x:'. Finally, if the wrong value is entered I want a message saying so.

Edit `helpers.php` and update the `pick_from_list()` function to match what's below.

```php
<?php
/**
 * Present a list of choices to user, return choice
 * @param Command $command The command requesting input
 * @param array $choices List of choices
 * @param int $default Default choice (1-array size), -1 to abort
 * @param string $abort String to tag on end for aborting selection
 * @return int -1 if abort selected, otherwise one greater than $choice index
 *             (in other words, choosing $choice[0] returns 1)
 * @throws InvalidArgumentException If argument is invalid
 */
function pick_from_list(Command $command, $title, array $choices,
  $default = 0, $abort = null)
{
  if ($abort)
  {
    $choices[] = $abort;
  }
  $numChoices = count($choices);
  if ( ! $numChoices)
  {
    throw new \InvalidArgumentException("Must have at least one choice");
  }
  if ( $default == -1 && empty($abort))
  {
    throw new \InvalidArgumentException(
      'Cannot use default=-1 without $abort option');
  }
  if ( ! between($default, -1, $numChoices))
  {
    throw new \InvalidArgumentException("Invalid value, default=$default");
  }
  $question = "Please enter a number between 1-$numChoices";
  if ($default > 0)
  {
    $question .= " (default is $default)";
  }
  elseif ($default < 0)
  {
    $question .= " (enter to abort)";
    $default = $numChoices;
  }
```

```
43    $question .= ':';
44    while(1)
45    {
46      $command->line('');
47      $command->info($title);
48      $command->line('');
49      for ($i = 0; $i < $numChoices; $i++)
50      {
51        $command->line(($i + 1).". ".$choices[$i]);
52      }
53      $command->line('');
54      $answer = $command->ask($question);
55      if ($answer == '')
56      {
57        $answer = $default;
58      }
59      if (between($answer, 1, $numChoices))
60      {
61        if ($abort and $answer == $numChoices)
62        {
63          $answer = -1;
64        }
65        return (int)$answer;
66      }
67
68      // Output wrong choice
69      $command->line('');
70      $formatter = $command->getHelperSet()->get('formatter');
71      $block = $formatter->formatBlock('Invalid entry!', 'error', true);
72      $command->line($block);
73    }
74  }
75  ?>
```

I also tweaked the code a bit from the previous version.

Fixing the unit tests

Wouldn't you know it. I broke the unit tests. Let's fix that real quick before wrapping up this chapter.

The issue is `HelpersTest::testPickFromListWorksExample3`. That's the test where the first time we mock a wrong selection. With the new code above, now it's wanting to call `$command->getHelperSet()` which doesn't exist on our mock object.

Fine. I don't usually like to do unit tests and create mock objects which return mock objects which return other values. But it's easy to do with `Mockery`.

Edit `HelpersTest.php` and update `testPickFromListWorksExample3` to match the code below.

```
<?php
  // First time through loop user selects bad choice, causing a second loop
  public function testPickFromListWorksExample3()
  {
    $formatter = Mockery::mock('stdClass');
    $formatter->shouldReceive('formatBlock')->once()->andReturn('');
    $helperset = Mockery::mock('stdClass');
    $helperset->shouldReceive('get')->once()->andReturn($formatter);
    $command = Mockery::mock('Illuminate\Console\Command');
    $command->shouldReceive('info')->times(2);
    $command->shouldReceive('line')->times(4);
    $command->shouldReceive('getHelperSet')->once()->andReturn($helperset);
    $command->shouldReceive('ask')->times(2)->andReturn('x', 1);

    $choice = pick_from_list($command, 'title', array('option'));
    $this->assertEquals(1, $choice);
  }
?>
```

Now if you run `phpunit`, everything should pass with flying colors.

Chapter 41 - The EditListCommand

In This Chapter

In this chapter we'll implement the `gsd:editlist` command.

The Plan

The 'gsd:editlist' command, as planned back in Chapter 35, looked like the following:

```
$ gsd editlist +name --title="title" --subtitle="subtitle"
$ gsd editlist +name -t "title" -s "subtitle"
$ gsd editlist
```

The last usage would prompt for everything. In fact, if the user omits the name then we have a programming decision to make ... do we use the default list defined in the configuration or prompt the user for the list?

Remember we already have the configuration option `noListPrompt` to tell us what to do.

Hmmm. It provides more flexibility to throw in an additional option. Maybe `listname=prompt|config`. That way the `noListPrompt` config setting can be overridden. This allows the default behavior to be specified by our configuration, yet the user has the power to override the default behavior.

I like having the power.

In Chapter 36 we determine the pseudo-code would be something like:

```
Get options and arguments
Prompt for list-id if not specified
Load list
If no arguments
   Prompt user for list-title
   Prompt user for list-subtitle
EndIf
Set list-title if needed
Set list-subtitle if need
Save list
```

The logic must change slightly to handle the different methods of determining the list name. In fact, this functionality will be used by other console commands we'll be creating. Let's create a method in `CommandBase` that's smart enough to pick the correct list name when omitted, either from the config default or user selected.

Updating CommandBase

Since where updating `CommandBase`, let's add a couple other useful bits to the class.

Edit your `CommandBase.php` file, replace the contents with what's below.

```
1  <?php namespace GSD\Commands;
2
3  use App;
4  use Config;
5  use Illuminate\Console\Command;
6  use Symfony\Component\Console\Input\InputOption;
7  use Symfony\Component\Console\Input\InputArgument;
8  use Todo;
9
10 class CommandBase extends Command {
11
12   protected $repository;
13   protected $nameArgumentDescription = 'List name.';
14
15   /**
16    * Constructor
17    */
18   // NO CHANGES
19
20   /**
21    * Prompt the user for a list id
22    * @param bool $existing Prompt for existing list or new list?
23    * @param bool $allowCancel Allow user to cancel
24    * @return mixed string list id or null if user cancels
25    */
26   // NO CHANGES
27
28   /**
```

```
29      * Output an error box
30      * @param string $message The message
31      */
32     // NO CHANGES
33
34     /**
35      * The console command arguments. Derived classes could replace this
36      * method entirely, or merge its own arguments with these.
37      *
38      * @return array
39      */
40     protected function getArguments()
41     {
42       return array(
43         array('+name', InputArgument::OPTIONAL,
44           $this->nameArgumentDescription),
45       );
46     }
47
48     /**
49      * The console command options. Derived classes could replace this
50      * method entirely, or merge its own options with these
51      *
52      * @return array
53      */
54     protected function getOptions()
55     {
56       return array(
57         array('listname', 'l', InputOption::VALUE_REQUIRED,
58           "Source of list name, 'prompt' or 'default'"),
59       );
60     }
61
62
63     /**
64      * Get the list id (of existing lists).
65      *
66      * This can happen in a variety of ways. If specified as an argument, then
67      * it's returned (without the + of course). Otherwise, look to see if the
68      * `--listname` argument is used and determine the list accordingly.
69      * Finally, we fallback to the method specified by Config's
70      * 'app.gsd.noListPrompt' setting
```

```
71      *
72      * @return $string Existing list id (or null if user aborts)
73      * @throws InvalidArgumentException If something's not right
74      */
75     protected function getListId()
76     {
77       $archived = $this->input->hasOption('archived') and
78                   $this->option('archived');
79       $name = $this->argument('+name');
80       $listnameOption = $this->option('listname');
81       if ($name)
82       {
83         $name = substr($name, 1);
84         if ( ! is_null($listnameOption))
85         {
86           throw new \InvalidArgumentException(
87             'Cannot specify +name and --listname together');
88         }
89       }
90       else
91       {
92         if (is_null($listnameOption))
93         {
94           $listnameOption = Config::get('app.gsd.noListPrompt')
95             ? 'prompt' : 'config';
96         }
97         if ($listnameOption == 'prompt')
98         {
99           $name = $this->askForListId(true, true, $archived);
100          if (is_null($name))
101          {
102            return null;
103          }
104        }
105        else
106        {
107          $name = Config::get('app.gsd.defaultList');
108        }
109      }
110
111      // Throw error if list doesn't exist
112      if ( ! $this->repository->exists($name, $archived))
```

```
113       {
114         $archived = ($archived) ? '(archived) ' : '';
115         throw new \InvalidArgumentException(
116           "List $archived'$name' not found");
117       }
118       return $name;
119     }
120   }
121   ?>
```

Here's a rundown of the changes made to `CommandBase`:

1. There's a couple more **use** statements.
2. A new `$nameArgumentDescription` property was added. It's used in the `getArguments()` method.
3. There's no changes in the existing three methods of `CommandBase`.
4. The `getArguments()` method is implemented here. As the comments state, subclasses can override or use. Here we use that new property, `$nameArgumentDescription`.
5. Same deal with `getOptions()`, derived classes can use it or ignore it.
6. The new `getListId()`, I'll explain this in more detail in a few minutes.

Let's create the EditListCommand to use this new functionality.

Creating the EditListCommand

Create the command shell with artisan

```
1  ~/gsd$ art command:make EditListCommand --path=app/src/GSD/Commands \
2  > --namespace="GSD\Commands"
3  Command created successfully.
```

Edit the newly created `EditListCommand.php` in your `app/src/GSD/Commands` directory and make it match what's below.

```php
<?php namespace GSD\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Todo;

class EditListCommand extends CommandBase {

  protected $name = 'gsd:editlist';
  protected $description = "Edit a list's title or subtitle.";
  protected $nameArgumentDescription = 'List name to edit.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId();
    var_dump($name);
  }

  /**
   * Get the console command options.
   */
  protected function getOptions()
  {
    return array_merge(parent::getOptions(), array(
      array('title', 't', InputOption::VALUE_REQUIRED,
        'Title of list.', null),
      array('subtitle', 's', InputOption::VALUE_REQUIRED,
        'Subtitle of list.', null),
    ));
  }
}
?>
```

Right now I'm just dumping out the name, so I can test a few choices. Sort of a pretest.

(I'm right on the verge of creating a unit test for this, but since I have a feeling this chapter's going to run long, I decided no unit tests. That may come as a shock to some testing purists out there. Please, if you feel the need to do some unit testing, then have at it.)

Telling Artisan About EditListCommand

Edit `start/artisan.php` so artisan knows about our new command

```
<?php

Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\EditListCommand);
Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\UncreateCommand);
?>
```

Notice I decided to keep the commands alphabetically.

Pretesting EditListCommand

Here's what I tested.

```
# Testing a non-existent list name
~/gsd$ art gsd:editlist +baddie
Result: error message as expected

# Testing an existing list
~/gsd$ art gsd:editlist +waiting
string(7) "waiting"

# Tests with no list specified config noListPrompt=true
~/gsd$ art gsd:editlist
Result: yes, I get the prompt

~/gsd$ art gsd:editlist --listname=default
string(7) "actions"

~/gsd$ art gsd:editlist --listname=prompt
Result: yes, I get the prompt

# Tests with no list specified config noListPrompt=false
```

```
~/gsd$ art gsd:editlist
string(7) "actions"

~/gsd$ art gsd:editlist --listname=default
string(7) "actions"

~/gsd$ art gsd:editlist --listname=prompt
Result: yes, I get the prompt
```

Okay. Good enough, let's finish this command. *(For those of you keeping track, I changed the `config/app.php` back to `'noListPrompt' = true`.)*

Finishing EditListCommand::fire()

One line of code in the existing `fire()` implementation and we've handled getting the list name in a myriad of ways. Nice.

Let's finish the `gsd:editlist` command by updating `EditListCommand.php` and replacing the current `fire()` method with what's below.

```php
<?php
  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId();
    if (is_null($name))
    {
      $this->outputErrorBox("EditList aborted");
      return;
    }
    $list = Todo::get($name);

    $title = $this->option('title');
    $subtitle = $this->option('subtitle');

    if (all_null($title, $subtitle))
```

```
21      {
22        $this->info(sprintf("Editing '%s'", $name));
23        $this->line('');
24        $title = $this->ask("Enter list title (enter to skip)?");
25        $subtitle = $this->ask("Enter list subtitle (enter to skip)?");
26        $this->line('');
27        if (all_null($title, $subtitle))
28        {
29          $this->comment('Nothing changed. List not udpated.');
30          return;
31        }
32      }
33
34      if ($title)
35      {
36        $list->set('title', $title);
37      }
38      if ($subtitle)
39      {
40        $list->set('subtitle', $subtitle);
41      }
42      $list->save();
43      $this->info(sprintf("List '%s' updated", $name));
44    }
45  ?>
```

I don't really need to explain this. The logic almost matches the original pseudo-code. It's just a bit cleaner with that call to `$this->getListId()`

Is it list-id or name?

I've been using these terms interchangeably. That's not likely to change soon. In my mind, when I'm thinking of it from the database or repository perspective, it's an id. But when I think of it from the context of *list*, it's the name of the list, the base filename of the list.

What can I say? Programming is a messy sport. You try to achieve that pristine ideal, but it often falls short. It's all about keeping it as clean as you can. Which is a perfect segue into the next chapter.

Chapter 42 - Refactoring Files and Config

In This Chapter

In this chapter we'll do some more refactoring.

Yes, more refactoring in this chapter. I'm just trying to keep things in as good as shape as I can. You think this book has lots of code refactoring? It's nothing like how I refactor in real life. It's just part of the process. Write some code, refactor it, write more, refactor. You should see me ... on the other hand, maybe you shouldn't. Often I'm sporting three days worth beard stubble, a grungy tee-shirt, and hair like Yahoo Serious.

Refactoring the Config

So far, our configuration has been stored in the `app/config/app.php` file, in the `gsd` section. I'm starting to feel a bit like an unwanted house guest in that file, so I'm going to create an `app/config/todo.php` file that stores all those settings.

Why didn't you set it up like that at first?

I thought about it, but then thought "Nah. There'll only be one or two config options."

Cut the whole `gsd` section from your `app/config/app.php` and paste it into a new file `app/config/todo.php`. Then clean up `todo.php` to match what's below.

```
1  <?php
2  // Config for our awesome getting stuff done application
3  return array(
4    'folder' => '/home/chuck/Documents/gsd/',
5    'extension' => '.txt',
6    'listOrder' => array(
7      'inbox', 'actions', 'waiting', 'someday', 'calendar',
8    ),
9    'defaultList' => 'actions',     // default list when not specified
10   'noListPrompt' => true,         // true=prompt for list, false=use default
11 );
12 ?>
```

Remember your `folder` should be specific to your installation.

Now rename the `app/config/testing/app.php` file to `app/config/testing/todo.php` and clean it up, too. It should match what's below.

```
<?php
// testing config
return array(
    'folder' => app_path().'/tests/GSD/Repositories/data',
);
?>
```

Now, change everywhere in the system that has `Config::get('app.gsd.` to `Config::get('todo.`

Here's a list of files and lines to change:

File: **`src/GSD/Repositories/TodoRepository.php`**
$this->path = str_finish(Config::get('app.gsd.folder'), '/');

File: **`src/GSD/Repositories/TodoRepository.php`**
$this->extension = Config::get('app.gsd.extension');

File: **`src/GSD/Commands/ListAllCommand.php`**
foreach (Config::get('app.gsd.listOrder') as $name)

File: **`src/GSD/Commands/CommandBase.php`**
$listnameOption = Config::get('app.gsd.noListPrompt') ...

File: **`src/GSD/Commands/CommandBase.php`**
$name = Config::get('app.gsd.defaultList');

File: **`tests/GSD/Repositories/TodoRepositoryTest.php`**
$ext = Config::get('app.gsd.extension');

That's it! It's gratifying to discover while doing this that all five of our configuration options are actually used.

It's a good idea to run `phpunit` after doing something like this.

```
~/gsd$ phpunit
PHPUnit 3.7.27 by Sebastian Bergmann.

Configuration read from /home/chuck/gsd/phpunit.xml

.....................................................

Time: 196 ms, Memory: 24.25Mb

OK (59 tests, 100 assertions)
```

All good.

Refactoring to use Laravel's File class

A reader pointed out ... or, more accurately, a reader asked "Why don't you use Laravel's Filesystem methods?"

I didn't think of it at the time. I still think of `file_exists()` instead of `File::exists()`. But this is supposed to be a book on Laravel, right? So let's use Laravel's class.

Easy. All file access methods should be located in the `TodoRepository` implementation.

Here's a list of what we're changing:

- all references to `is_dir()` to `File::isDirectory()`
- all references to `file_exists()` to `File::exists()`
- `TodoRepository::delete()` made simpler by using `File::delete()`
- changing call `glob()` to `File::glob()`
- changing call to `file()` to `explode("\n", File::get())`
- changing call to `file_put_contents()` to `File::put()`

After making those changes and running the unit tests, a weird bug popped up. The file `TodoRepository.php` contained the line below which started puking.

```
if (count($lines) && $lines[0][0] == '(')
```

Hmmm. Well, I changed it as below and it passes unit tests:

```
if (count($lines) && starts_with($lines[0], '('))
```

Strange how that line wasn't failing before.

You're updated `TodoRepository.php` should match what's below.

```php
<?php namespace GSD\Repositories;

// File: app/src/GSD/Repositories/TodoRespository.php

use Config;
use File;
use GSD\Entities\ListInterface;

class TodoRepository implements TodoRepositoryInterface {

  protected $path;
  protected $extension;

  /**
   * Constructor. We'll throw exceptions if the paths don't exist
   */
  public function __construct()
  {
    $this->path = str_finish(Config::get('todo.folder'), '/');
    if ( ! File::isDirectory($this->path))
    {
      throw new \RuntimeException("Directory doesn't exist: $this->path");
    }
    if ( ! File::isDirectory($this->path.'archived'))
    {
      throw new \RuntimeException("Directory doesn't exist: $this->path".
        'archived');
    }
    $this->extension = Config::get('todo.extension');
    if ( ! starts_with($this->extension, '.'))
    {
      $this->extension = '.' . $this->extension;
    }
  }

  /**
   * Delete the todo list
   * @param string $id ID of the list
   * @return boolean True if successful
   */
  public function delete($id, $archived = false)
  {
```

```
43      return File::delete($this->fullpath($id, $archived));
44    }
45
46    /**
47     * Does the todo list exist?
48     * @param  string $id ID of the list
49     * @param  boolean $archived Check for archived lists only?
50     * @return boolean
51     */
52    public function exists($id, $archived = false)
53    {
54      $file = $this->fullpath($id, $archived);
55      return File::exists($file);
56    }
57
58    /**
59     * Return the ids of all the lists
60     * @param  boolean $archived Return archived ids or unarchived?
61     * @return array of list ids
62     */
63    public function getAll($archived = false)
64    {
65      $match = $this->path;
66      if ($archived)
67      {
68        $match .= 'archived/';
69      }
70      $match .= '*' . $this->extension;
71      $files = File::glob($match);
72      $ids = array();
73      foreach ($files as $file)
74      {
75        $ids[] = basename($file, $this->extension);
76      }
77      return $ids;
78    }
79
80    /**
81     * Load a TodoList from it's id
82     * @param string $id ID of the list
83     * @param boolean $archived Load an archived list?
84     * @return ListInterface The list
```

```
85      * @throws InvalidArgumentException If $id not found
86      */
87     public function load($id, $archived = false)
88     {
89       if ( ! $this->exists($id, $archived))
90       {
91         throw new \InvalidArgumentException(
92           "List with id=$id, archived=$archived not found");
93       }
94       $lines = explode("\n", File::get($this->fullpath($id, $archived)));
95
96       // Pull title
97       $title = array_shift($lines);
98       $title = trim(substr($title, 1));
99
100      // Pull subtitle
101      if (count($lines) && starts_with($lines[0], '('))
102      {
103        $subtitle = trim(array_shift($lines));
104        $subtitle = ltrim($subtitle, '(');
105        $subtitle = rtrim($subtitle, ')');
106      }
107
108      // Setup the list
109      $list = \App::make('GSD\Entities\ListInterface');
110      $list->set('id', $id);
111      $list->set('title', $title);
112      if ( ! empty($subtitle))
113      {
114        $list->set('subtitle', $subtitle);
115      }
116      $list->set('archived', $archived);
117
118      // And add the tasks
119      foreach ($lines as $line)
120      {
121        $line = trim($line);
122        if ($line)
123        {
124          $list->taskAdd($line);
125        }
126      }
```

```
127
128       return $list;
129     }
130
131     /**
132      * Save a TodoList
133      * @param ListInterface $list The TODO List
134      * @return boolean True if successful
135      */
136     public function save(ListInterface $list)
137     {
138       $id = $list->get('id');
139       $archived = $list->get('archived');
140       $build = array();
141       $build[] = '# ' . $list->get('title');
142       $subtitle = $list->get('subtitle');
143       if ($subtitle)
144       {
145         $build[] = "($subtitle)";
146       }
147       $lastType = 'z';
148       $tasks = $list->tasks();
149       foreach ($tasks as $task)
150       {
151         $task = (string)$task;
152         $type = $task[0];
153         if ($type != $lastType)
154         {
155           $build[] = '';      // Blank line between types of tasks
156           $lastType = $type;
157         }
158         $build[] = $task;
159       }
160       $content = join("\n", $build);
161       $filename = $this->fullpath($id, $archived);
162       $result = File::put($filename, $content);
163
164       return $result !== false;
165     }
166
167     /**
168      * Return the path to the list file
```

```
169     */
170   protected function fullpath($id, $archived)
171   {
172     $path = $this->path;
173     if ($archived)
174     {
175       $path .= 'archived/';
176     }
177     $path .= $id . $this->extension;
178     return $path;
179   }
180 }
181 ?>
```

We're all done refactoring. (For now :)

Let's get back to the regularly scheduled coding.

Chapter 43 - The AddTaskCommand

In This Chapter

In this chapter we'll create the add task command

If I follow the list of commands from the end of Chapter 35, next up would be the Archive List, Unarchive List, and Rename List commands. But I can't wait to actually add task to our lists. So let's work on that first.

The Plan

In Chapter 35 we planned on having the console command work like what's below for adding tasks.

```
$ gsd add [+name] "something to do" --next
$ gsd a [+name] "something to do" -n
```

And the pseudo-code from the following chapter was:

```
Get options and arguments
Validate arguments
Prompt for list-id if needed
Load list
Create New task description
Set next action if needed
Add task to list
Save list
```

We'll mostly follow the pseudo-code, but definitely use the `CommandBase::getListId()` method to fetch the list's name. Should be simple to implement.

Creating the AddTaskCommand

The first step, as always when creating new commands, is to use `artisan` to create the skeleton.

```
~/gsd$ art command:make AddTaskCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Edit the newly created `AddTaskCommand.php` in to match what's below.

```
<?php namespace GSD\Commands;

use App;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class AddTaskCommand extends CommandBase {

  protected $name = 'gsd:addtask';
  protected $description = 'Add a new task to a list.';
  protected $nameArgumentDescription = 'List name to add the task to.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $this->info('do it');
  }

  /**
   * Get the console command arguments.
   */
  protected function getArguments()
  {
    return array_merge(array(
      array('task', InputArgument::REQUIRED,
        "The task's description."),
    ), parent::getArguments());
```

```
34    }
35
36    /**
37     * Get the console command options.
38     */
39    protected function getOptions()
40    {
41      return array_merge(parent::getOptions(), array(
42        array('action', 'a', InputOption::VALUE_NONE,
43          'Make task a Next Action.'),
44      ));
45    }
46  }
47  ?>
```

The `$name` and `$description` properties are standard. The `$nameArgumentDescription` is set so if you do a `art help gsd:addtask` then `+name` option will output something more meaningful that the description that `CommandBase` provides.

Let's ask artisan for help on this command.

First we need to add the needed line to `start/artisan.php`.

```
1  <?php
2
3  Artisan::add(new GSD\Commands\AddTaskCommand);
4  Artisan::add(new GSD\Commands\CreateCommand);
5  Artisan::add(new GSD\Commands\EditListCommand);
6  Artisan::add(new GSD\Commands\ListAllCommand);
7  Artisan::add(new GSD\Commands\UncreateCommand);
8  ?>
```

(I'm still keeping the commands alphabetical. That's just how I roll.)

Now get help on the command we just added

```
~/gsd$ art help gsd:addtask
Usage:
 gsd:addtask [-l|--listname="..."] [-a|--action] task [+name]

Arguments:
 task                  The task's description.
 +name                 List name to add the task to.

Options:
 --listname (-l)       Source of list name, 'prompt' or 'config'
 --action (-a)         Make task a Next Action.
 --help (-h)           Display this help message.
 --quiet (-q)          Do not output any message.
 --verbose (-v|vv|vvv) Increase the verbosity of messages: 1 for normal
  output, 2 for more verbose output and 3 for debug
 --version (-V)        Display this application version.
 --ansi                Force ANSI output.
 --no-ansi             Disable ANSI output.
 --no-interaction (-n) Do not ask any interactive question.
 --env                 The environment the command should run under.
```

Pretty slick, huh? You'll notice I had to change the order of the arguments from our original plan so that `task` comes before `+name`. This is because required arguments must always come before optional arguments and `+name` is optional.

Let's make the rule, for consistency, that the `+name` will always come last.

Also, I changed the `--next` option to `--action`. This is because a short option can only be a single character and `-n` was already taken so I decided the short version should be `-a` for *next **A**ction*. Then it made more sense for the long version of the option to be `--action`.

Adding the code to the fire() method

Edit `AddTaskCommand.php` and update the `fire()` method to match what's below.

```
<?php
  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId();
    if (is_null($name))
    {
      $this->outputErrorBox("AddTask aborted");
      return;
    }
    $list = Todo::get($name);

    $task = App::make('GSD\Entities\TaskInterface');
    if ( ! $task->setFromString($this->argument('task')))
    {
      throw new \InvalidArgumentException('Cannot parse task string');
    }
    $type = 'Todo';
    if ($this->option('action'))
    {
      $task->setIsNextAction(true);
      $type = 'Next Action';
    }
    $list->taskAdd($task);
    $list->save();
    $this->info("$type successfully added to $name");
  }
?>
```

We followed the pseudo-code fairly close. I didn't comment the code because what's happening is self-explanatory.

Should the `task` argument be an option instead?

Hmmm.

Or maybe the `task` should be optional and we could prompt for it if missing?

Hmmm, again.

I don't think I'll change it at this point. It'd be easy to change in the future should we want to.

No unit tests, but I'm going to go through a quick, manual test of the command.

Manually testing

Most of the underlying components have been unit tested. This includes the following:

- The `Todo` facade's `get()` method.
- Creating a task from the interface.
- Calling `setFromString()` on a `Task` object.
- Calling `setIsNextAction()` on a `Task` object.
- Calling `taskAdd()` and `save()` on a `TodoList` object.

Likewise, we've already manually tested the `getListId()` method of `CommandBase`. As I see it there's really only a couple things to test:

- Adding a task to a list.
- Adding a task and a next action to a list.

I have a list called `actions`, so I tested this with the following. (Be sure to change your listname and path to your list as needed).

```
~\gsd$ art gsd:addtask "Test add task" +actions
Todo successfully added to actions
~\gsd$ art gsd:addtask "Test add default" -l config
Todo successfully added to actions
~\gsd$ art gsd:addtask "Test add next action" +actions -a
Next Action successfully added to actions
~\gsd$ cat /home/chuck/Documents/gsd/actions.txt
# Next Actions
(Things to do next)

* Test add next action

- Test add default
- Test add task
```

Looks good to me. Now lets go mark something complete.

Chapter 44 - The DoTaskCommand

In This Chapter

In this chapter we'll create the complete task command

Now that we can add tasks to our todo lists, let's add the ability to mark things complete.

The Plan

Our plan from Chapter 35 was pretty basic.

```
$ gsd do [+name] n
```

But there is a problem with this plan. Last chapter we discovered that optional arguments must come after required arguments. So we made the *executive decision* to always have the `+name` argument last.

Also, I want to have the `n`, er, `task-number` argument also optional. We have that nifty `pick_from_list()` helper function we can use to preset a list to the user to choose from.

The new version should look like this.

```
$ gsd do [task-number] [+name]
```

The pseudo-code for marking tasks complete was:

```
Get arguments
Prompt for list-id if needed
Load list
Prompt for task # if needed
Set task # completed
Save list
```

Interesting, the pseudo-code already had the "prompt for task # if needed" step. I had forgot all about that. What I really need is some application that I could use to keep track of lists of things I want to remember.

Creating the DoTaskCommand

The first step is to use `artisan` to create the skeleton.

```
~/gsd$ art command:make DoTaskCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Edit the newly created `DoTaskCommand.php` to match the following.

```
<?php namespace GSD\Commands;

use Todo;

class DoTaskCommand extends CommandBase {

  protected $name = 'gsd:do';
  protected $description = 'Mark a task as complete.';
  protected $nameArgumentDescription = 'List name with completed task.';
  protected $taskNoDescription = 'Task # to mark complete.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId();
    if (is_null($name))
    {
      $this->outputErrorBox("DoTask aborted");
      return;
    }
    $list = Todo::get($name);

```

```
27      $taskNo = $this->getTaskNo($list, true, true, false);
28      if (is_null($taskNo))
29      {
30        $this->outputErrorBox("DoTask aborted");
31        return;
32      }
33
34      $description = $list->taskGet($taskNo, 'description');
35      $list->taskSet($taskNo, 'isComplete', true)
36           ->save();
37      $this->info("Task '$description' marked complete.");
38    }
39  }
40  ?>
```

This was quick and fun to code. Will it work? I hope so and expect it to, but I'm not able to tell you absolutely 100% that the code is flawless ... yet.

Let me explain things, using the line numbers as reference (Dang. I probably should have done this all along.)

Line 3
: No real need for the other use statements.

Lines 5 - 9
: Almost the same as the last command we developed. We're building on `CommandBase`, setting up the command and description and what we're going to call the `+name` option.

Line 10
: Here I got to thinking that `CommandBase` should be smart enough to check if this property is present. Then it could automatically set up the `task-number` argument.

Lines 19 - 25
: This is just like the last command. We figure the list name and abort if the user was prompted and chose to abort.

Lines 27 - 32
: This is similar to getting the `+name` argument, but we'll be getting the `task-number`. This method doesn't exist yet, but the arguments will be `$list`, `$showNext`, `$showNormal`, and `$showComplete`.

Lines 34 - 37
: Here we mark the task complete, save the list, and output the message.

Next, update `start/artisan.php` so artisan knows about the command.

```
<?php

Artisan::add(new GSD\Commands\AddTaskCommand);
Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\DoTaskCommand);
Artisan::add(new GSD\Commands\EditListCommand);
Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\UncreateCommand);

?>
```

You could run artisan and see from the list of commands, that `gsd:do` is there, but we need to add the code to `CommandBase` before it will work.

Updating CommandBase

Make the top of `CommandBase.php` match the snippet below.

```
<?php
// above this point is the same
class CommandBase extends Command {

  protected $repository;
  protected $nameArgumentDescription = 'List name.';
  protected $taskNoDescription = null;

  // below this point is the same
?>
```

This adds the property `$taskNoDescription`, which we'll use next.

Replace the `getArguments()` method in `CommandBase.php` with the following.

```
<?php
  /**
   * The console command arguments. Derived classes could replace this
   * method entirely, or merge its own arguments with them
   *
   * @return array of argument definitions
   */
  protected function getArguments()
  {
    $args = array();
    if ( ! is_null($this->taskNoDescription))
    {
      $args[] = array(
        'task-number',
        InputArgument::OPTIONAL,
        $this->taskNoDescription
      );
    }
    $args[] = array(
      '+name',
      InputArgument::OPTIONAL,
      $this->nameArgumentDescription
    );
    return $args;
  }
?>
```

All we're doing here is checking if the `$taskNoDescription` is set and, if it is, then adding the `task-number` argument *before* the `+name` argument.

Finally, add the `getTaskNo()` method to the `CommandBase` class.

```php
<?php
  /**
   * Get the task # of a list, either from the argument or prompt the user.
   * Keep in mind the # present to the user always begins with 1, but the
   * number we return is always one less (starting with 0)
   *
   * @param ListInterface $list The Todo List
   * @param bool $showNext Show next actions in prompt list
   * @param bool $showNormal Show normal tasks in prompt list
   * @param bool $showComplete Show completed tasks in prompt list
   * @return mixed NULL if user aborts, otherwise integer of task number
   */
  protected function getTaskNo(\GSD\Entities\ListInterface $list,
    $showNext, $showNormal, $showComplete)
  {
    // Return the # if provided on command line
    $taskNo = $this->argument('task-number');
    if ( ! is_null($taskNo))
    {
      return (int)$taskNo - 1;
    }

    // Build list of tasks
    $tasks = array();
    foreach ($list->tasks() as $task)
    {
      if ($task->isComplete())
      {
        if ($showComplete)
          $tasks[] = (string)$task;
      }
      elseif ($task->isNextAction())
      {
        if ($showNext)
          $tasks[] = (string)$task;
      }
      elseif ($showNormal)
      {
        $tasks[] = (string)$task;
      }
    }

```

```
43      // Let user pick from list, return result
44      $result = pick_from_list($this, $this->taskNoDescription,
45        $tasks, 0, "cancel, do not perform action");
46      if ($result == -1)
47      {
48        return null;
49      }
50      return $result - 1;
51    }
52  ?>
```

The code seems pretty simple. It even has comments. I don't think I need to explain it.

Testing DoTaskCommand

Let's test it. Again, much of the underlying support classes and methods have been unit tested. So I'm just going to manually test things.

```
~/gsd$ art gsd:do 33 +actions

  [OutOfBoundsException]
  $index is outside range
```

Good this is what we wanted.

```
~/gsd$ art gsd:do -l config

Task # to mark complete.

1. * Test add next action
2. - Test add default
3. - Test add task
4. cancel, do not perform action

Please enter a number between 1-4:4

  DoTask aborted
```

Again, what was expected, let's mark #2 as complete.

```
~/gsd$ art gsd:do -l config

Task # to mark complete.

1. * Test add next action
2. - Test add default
3. - Test add task
4. cancel, do not perform action

Please enter a number between 1-4:2
Task 'Test add default' marked complete.
```

Yes! Man are we good or what? Let's check the file, see how it looks.

```
~/gsd$ cat /home/chuck/Documents/gsd/actions.txt
# Next Actions
(Things to do next)

* Test add next action

x 2013-09-28 Test add default

- Test add task
```

What the heck? It marked the task complete, but it's not at the end of the file. Invariably, I start to think things are all good and a bug jumps out of nowhere.

Killing the Bug

Upon examining how the list is sorted, I realize sorting occurs within the `TaskCollection` whenever items are added to or removed from the list. We could fix this a number of ways:

1. Force the list to be sorted before saving.
2. Change our `DoTaskCommand` to remove the task when marking it complete, then add it back.
3. Have the `TaskCollection` always return a sorted list.
4. Have the tasks automatically sort the list whenever they change.

Hmmm. So which one is best? The answer is #3.

The first choice requires the `TodoRepository` to have knowledge of the `TaskCollection`. Likewise, #2 requires us to have knowledge in the `DoTaskCommand`. #4 is very problematic because there's no way for a task to know what collection it belongs to.

Don't spread the knowledge

Unlike real life, where it's good to share knowledge and learn and help others to learn, when programming a good goal is to keep things as stupid as possible. It's the KISS principle[a], but more. Yes, it's about keeping things decoupled.

[a] http://en.wikipedia.org/wiki/KISS_principle

Because any solution other than #3 requires our classes to be smarter than they need to be, let's just have the `TaskCollection` take care of this.

Edit `TaskCollection` and change the `get()` method and the `getAll()` method as below.

```php
<?php
  /**
   * Return task based on index
   * @param integer $index 0 is first item in collection
   * @return TaskInterface The Todo Task
   * @throws OutOfBoundsException If $index outside range
   */
  public function get($index)
  {
    $this->sortTasks();
    if ($index < 0 || $index >= count($this->tasks))
    {
      throw new \OutOfBoundsException('$index is outside range');
    }
    return $this->tasks[$index];
  }

  /**
   * Return array containing all tasks
   * @return array
   */
  public function getAll()
  {
    $this->sortTasks();
    return $this->tasks;
```

```
26     }
27 ?>
```

This is adding that `$this->sortTasks()` at the beginning of each method, thus making sure the list is always sorted.

I'll leave it to you to test and make sure it works. I did and it does.

Chapter 45 - The ListTasksCommand

In This Chapter

In this chapter we'll create the list tasks command

Finally, after pages and pages of me blabbing away we'll start being able to see what we need to get done.

Remember this list we mocked up about 10 chapters ago?

```
+---+------+-------------------------------------+--------------+
| # | Next | Description                         | Extra        |
+---+------+-------------------------------------+--------------+
| 1 | YES  | Finish Chapter 36                   | Due Sep-20   |
| 2 | YES  | Balance Checkbook                   |              |
| 3 |      | Start project for world domination  |              |
| 4 |      | Read Dr. Evil's Memoirs             |              |
|   | done | Finish Chapter 35                   | Done 9/19/13 |
+---+------+-------------------------------------+--------------+
```

Now that's a nice looking list. Very pretty for a console application. I really can't wait to start #3 on the list. But I may have to do #4 first. We'll see.

The Plan

Back in Chapter 35, we came up with the following:

```
$ gsd list [+name] --next --nodone
$ gsd ls [$name] -n -nd
```

But to keep consistent with other commands we've created, I'm changing the `--next` command to `--action`. Also, I'm not liking the `--nodone` option. I don't know, it's just … yegh. I read it as 'nod one'. So let's implement the command as below.

```
$ gsd list [+name] --action --skip-done
$ gsd ls [$name] -a -x
```

The peudo-code we had was:

```
Get arguments and options
Prompt for list-id if needed
Load list
Loop through tasks
  If task not filtered by option
    show task
  EndIf
EndLoop
```

Pretty simple.

Creating the ListTasksCommand

Create the command skeleton.

```
~/gsd$ art command:make ListTasksCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Update `ListTasksCommand.php` to match what's below.

```php
<?php namespace GSD\Commands;

use Symfony\Component\Console\Input\InputOption;
use Todo;

class ListTasksCommand extends CommandBase {

  protected $name = 'gsd:list';
  protected $description = 'List tasks.';
  protected $nameArgumentDescription = 'List name to display tasks.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId();
    if (is_null($name))
    {
      $this->outputErrorBox("ListTasks aborted");
       return;
    }
    $list = Todo::get($name);

    $nextOnly = $this->option('action');
    $skipDone = $this->option('skip-done');
    if ($nextOnly and $skipDone)
    {
      $this->outputErrorBox(
        "Options --action and --skip-done can't be used together."
      );
      return;
    }

    // Gather rows to display
    $rows = array();
    $rowNo = 1;
    foreach ($list->tasks() as $task)
    {
      if ($task->isComplete())
```

```
43        {
44          if ($skipDone or $nextOnly) continue;
45          $rows[] = array(
46            '',
47            'done',
48            $task->description(),
49            'Done '.$task->dateCompleted()->format('n/j/y'),
50          );
51        }
52        elseif ($task->isNextAction() or ! $nextOnly)
53        {
54          $next = ($task->isNextAction()) ? 'YES' : '';
55          $due = ($task->dateDue()) ?
56            'Due '.$task->dateDue()->format('M-j') : '';
57          $rows[] = array(
58            $rowNo++,
59            $next,
60            $task->description(),
61            $due,
62          );
63        }
64      }
65
66      // Output a pretty table
67      $title = ($nextOnly) ? "Next Actions" :
68        (($skipDone) ? "Active Tasks" : "All Tasks");
69      $this->info("$title in list '+$name'\n");
70      if (count($rows) == 0)
71      {
72        $this->error("Nothing found");
73        return;
74      }
75      $table = $this->getHelperSet()->get('table');
76      $table
77        ->setHeaders(array('#', 'Next', 'Description', 'Extra'))
78        ->setRows($rows)
79        ->render($this->getOutput());
80    }
81
82    /**
83     * Get the console command options.
84     *
```

```
85     * @return array
86     */
87    protected function getOptions()
88    {
89      return array_merge(array(
90        array('action', 'a', InputOption::VALUE_NONE,
91          'Show only next actions.', null),
92        array('skip-done', 'x', InputOption::VALUE_NONE,
93          'Skip completed actions.', null),
94      ), parent::getOptions());
95    }
96  }
97  ?>
```

What the code does

Lines 1 - 25
All pretty standard stuff that doesn't need explanation.

Lines 27 - 35
Here we pull the options and if both `--action` and `--skip-done` are present, an error is displayed. It doesn't really make sense to allow both of those.

Lines 37 - 64
This loop is building an array of arrays. The data that will be displayed in our table.

Lines 66 - 80
We output the title and the pretty table

Lines 87 - 95
We return the options this command takes.

Edit `start/artisan.php` and tell artisan about the new command.

```
<?php

Artisan::add(new GSD\Commands\AddTaskCommand);
Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\DoTaskCommand);
Artisan::add(new GSD\Commands\EditListCommand);
Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\ListTasksCommand);
Artisan::add(new GSD\Commands\UncreateCommand);

?>
```

That's it. Run it. It works.

Testing the ListTasksCommand

Here's some examples of me just listing tasks. Your output may look different.

```
~/gsd$ art gsd:list +actions
All Tasks in list '+actions'

+---+------+----------------------+--------------+
| # | Next | Description          | Extra        |
+---+------+----------------------+--------------+
| 1 | YES  | Test add next action |              |
| 2 |      | Another test         |              |
| 3 |      | Something            | Due Oct-1    |
| 4 |      | Test add task        |              |
|   | done | Test add default     | Done 9/28/13 |
+---+------+----------------------+--------------+
~/gsd$ art gsd:list +actions --action
Next Actions in list '+actions'

+---+------+----------------------+-------+
| # | Next | Description          | Extra |
+---+------+----------------------+-------+
| 1 | YES  | Test add next action |       |
+---+------+----------------------+-------+
~/gsd$ art gsd:list +actions -x
Active Tasks in list '+actions'

```

```
+---+------+----------------------+-----------+
| # | Next | Description          | Extra     |
+---+------+----------------------+-----------+
| 1 | YES  | Test add next action |           |
| 2 |      | Another test         |           |
| 3 |      | Something            | Due Oct-1 |
| 4 |      | Test add task        |           |
+---+------+----------------------+-----------+
```

Hey! This thing is starting to become usable.

Chapter 46 - Eating Our Own Dog Food

In This Chapter

In this chapter we'll start using our application to track what to do.

What is Eating Your Own Dog Food

Eating one's own dogfood basically means using the product being developed. It's also called *dogfooding*. Here's a few examples.

In the early 1990s, Microsoft forced the Windows NT teams to use the OS they were developing. Even though it was a painful, crash-prone time eventually they worked out the kinks (well, most of them).

Apple supposedly got rid of all typewriters back in 1980, forcing all in-house typing to occur on their own products.

I most often use it in systems I build by consuming the system's own API. In other words if a function is exposed to a user (via web interface or console) and the functionality is also exposed via API, I build the web interface or console app to use the API.

So let's use the *Getting Stuff Done* application, as it is now, to start tracking what needs to happen within the *Getting Stuff Done* application.

Setting up the gsd todo list

First create a new list named **gsd**

```
~/gsd$ art gsd:create +gsd -t "Getting Stuff Done" \
> -s "Things to do in the gsd app"
List 'gsd' successfully create
```

Now add the following tasks to it.

```
~/gsd$ art gsd:addtask "List tasks in ListAllCommand" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create ArchiveListCommand" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create UnarchiveListCommand" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create RenameListCommand" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create EditTaskCommand" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create RemoveTaskCommand" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create MoveTaskCommand" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create shell gsd script" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create web application wireframe" +gsd
```

What's our todo list look like now?

```
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'

+---+------+----------------------------------+-------+
| # | Next | Description                      | Extra |
+---+------+----------------------------------+-------+
| 1 |      | Create ArchiveListCommand        |       |
| 2 |      | Create EditTaskCommand           |       |
| 3 |      | Create MoveTaskCommand           |       |
| 4 |      | Create RemoveTaskCommand         |       |
| 5 |      | Create RenameListCommand         |       |
| 6 |      | Create UnarchiveListCommand      |       |
| 7 |      | Create shell gsd script          |       |
| 8 |      | Create web application wireframe |       |
| 9 |      | List tasks in ListAllCommand     |       |
+---+------+----------------------------------+-------+
```

Looking good. I want to work on the Archive/Unarchive next and there's something that's bugging me about the output. It's that blank line after it displays the title **All Tasks in list '+gsd'**. Let's remove that blank line so it matches out the `art gsd:listall` command looks.

Hmmm. Dang it. We can't edit a task yet. Well, that will be the next chapter. After I can edit the list then I'll move to the Archive/Unarchive.

For now, lets just add the blank line task. And see our list of tasks.

```
~/gsd$ art gsd:addtask "Remove blank line after gsd:list title" -a +gsd
Next Action successfully added to gsd.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'

+----+------+----------------------------------------+-------+
| #  | Next | Description                            | Extra |
+----+------+----------------------------------------+-------+
| 1  | YES  | Remove blank line after gsd:list title |       |
| 2  |      | Create ArchiveListCommand              |       |
| 3  |      | Create EditTaskCommand                 |       |
| 4  |      | Create MoveTaskCommand                 |       |
| 5  |      | Create RemoveTaskCommand               |       |
| 6  |      | Create RenameListCommand               |       |
| 7  |      | Create UnarchiveListCommand            |       |
| 8  |      | Create shell gsd script                |       |
| 9  |      | Create web application wireframe       |       |
| 10 |      | List tasks in ListAllCommand           |       |
+----+------+----------------------------------------+-------+
```

Cool. We're now dogfooding.

Chapter 47 - The EditTaskCommand

In This Chapter

In this chapter we'll created the EditTaskCommand

The Plan

The original thoughts on the console command were.

```
$ gsd edit [+name] 1 "the description" --next[=off]
$ gsd ed [+name] 1 "the description" -n[=off]
$ gsd ed [+name]
```

But, to be consistent I want to make it this:

```
$ gsd edit [task-number] [+name] --desc="the description" --action[=off]
$ gsd ed
```

I'm changing the description to be an option instead of an argument. The list name will be prompted if not provided, likewise the task-number will be prompted. And, if nothing's provided, then both the description and action will be prompted.

The pseudo-code was.

```
Get arguments and options
Prompt for list-id if needed
Load list
If no task # specified
   Prompt for task #
   Prompt for new description
   Prompt for Is Next Action
End
Save Task Description
Save NextAction if needed
Save list
```

Adding a str2bool() helper

Since the `--action` option will take a on, off, yes, no, true, false, type value. Let's start by adding a helper that makes this easy.

Edit `helpers.php` and add the following function.

```
<?php
/**
 * Returns TRUE if the string is a true value
 */
function str2bool($value)
{
  return filter_var($value, FILTER_VALIDATE_BOOLEAN);
}
?>
```

This function uses the PHP built in boolean validator.

Edit `HelpersTest.php` and add the following test method.

```
<?php
  public function testStr2Bool()
  {
    $this->assertTrue(str2bool('yes'));
    $this->assertTrue(str2bool('True'));
    $this->assertTrue(str2bool('1'));
    $this->assertTrue(str2bool('oN'));
    $this->assertFalse(str2bool('x'));
    $this->assertFalse(str2bool(''));
    $this->assertFalse(str2bool('0'));
    $this->assertFalse(str2bool('no'));
    $this->assertFalse(str2bool('false'));
    $this->assertFalse(str2bool('off'));
  }
?>
```

Run phpunit to test

```
~/gsd$ phpunit
PHPUnit 3.7.27 by Sebastian Bergmann.

Configuration read from /home/chuck/gsd/phpunit.xml

.....................................................

Time: 208 ms, Memory: 25.25Mb

OK (60 tests, 110 assertions)
```

Okay, all good.

Creating the EditTaskCommand

First, create the command skeleton.

```
~/gsd$ art command:make EditTaskCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Update `EditTaskCommand.php` to match what's below.

```php
<?php namespace GSD\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class EditTaskCommand extends CommandBase {

  protected $name = 'gsd:edit';
  protected $description = 'Edit a task.';
  protected $nameArgumentDescription = 'List name with task to edit.';
  protected $taskNoDescription = 'Task # to edit.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Should we prompt for everything?
    $promptAll = all_null(
      $this->argument('+name'),
      $this->argument('task-number'),
      $this->option('descript'),
      $this->option('action')
    );

    // Get list
    $name = $this->getListId();
    if (is_null($name))
    {
      $this->outputErrorBox("EditTask aborted");
      return;
    }
    $list = Todo::get($name);

    // Get task-number
    $taskNo = $this->getTaskNo($list, true, true, false);
    if (is_null($taskNo))
    {
```

```
43        $this->outputErrorBox("EditTask aborted");
44        return;
45      }
46
47      $currDescript = $list->taskGet($taskNo, 'description');
48      $currAction = $list->taskGet($taskNo, 'isNextAction');
49
50      // Prompt for description and next action
51      if ($promptAll)
52      {
53        $currActionState = ($currAction) ? 'is' : 'is not';
54        $this->line("Current description: $currDescript");
55        $descript = $this->ask("New description (enter to skip)?");
56        $this->line("Task $currActionState currently a Next Action.");
57        $next = $this->ask("Is Next Action (enter skip, yes or no)?");
58      }
59
60      // Pull description and next action from command
61      else
62      {
63        $descript = $this->option('descript');
64        $next = $this->option('action');
65      }
66      $action = is_null($next) ? null : str2bool($next);
67
68      if ((is_null($descript) || $descript == $currDescript) &&
69          (is_null($action)   || $action == $currAction))
70      {
71        $this->outputErrorBox("Nothing changed");
72        return;
73      }
74
75      // Make changes and save the list
76      $task = $list->task($taskNo);
77      if ( ! is_null($action))
78      {
79        $task->setIsNextAction($action);
80      }
81      if ( ! is_null($descript))
82      {
83        $task->setDescription($descript);
84      }
```

```
85      $list->save(true);
86
87      $this->info("Task in $name updated to: ".(string)$task);
88    }
89
90    /**
91     * Get the console command options.
92     *
93     * @return array
94     */
95    protected function getOptions()
96    {
97      return array_merge(array(
98        array('descript', 'd', InputOption::VALUE_REQUIRED,
99          'New description for task.'),
100       array('action', 'a', InputOption::VALUE_REQUIRED,
101         'Is task a next action (yes|no).'),
102     ), parent::getOptions());
103   }
104 }
105 ?>
```

Lines 1 - 20
All standard stuff we've been doing

Lines 22 - 28
We want to know if no options or arguments were passed. This will trigger prompting for the description and next action.

Lines 30 - 45
Again, stuff we've did before in other commands.

Lines 47 - 48
We're stashing the task's current description and next action status.

Lines 50 - 65
Here we're either prompting for the description and next action, or pulling the values from any options passed.

Line 66
$action will either be null, true, or false

Lines 68 - 73
If nothing was changed, then output a big red box saying so

Lines 75 - 84

Here we pull the task object and update it as needed. Since PHP uses references for objects, the task still is in the list's collection. We do this because referencing it by a number is likely to change if we change the description or the next action status.

Line 85

Since we changed the task outside the list, the list doesn't know about the change and didn't mark itself dirty. Thus, we'll refactor the save() method to take an optional $force flag to force saving even if not dirty.

Lines 87 - 105

The rest of the file is pretty standard stuff

Make artisan aware of `EditTaskCommand` by making your `start/artisan.php` match what's below

```
1  <?php
2  Artisan::add(new GSD\Commands\AddTaskCommand);
3  Artisan::add(new GSD\Commands\CreateCommand);
4  Artisan::add(new GSD\Commands\DoTaskCommand);
5  Artisan::add(new GSD\Commands\EditListCommand);
6  Artisan::add(new GSD\Commands\EditTaskCommand);
7  Artisan::add(new GSD\Commands\ListAllCommand);
8  Artisan::add(new GSD\Commands\ListTasksCommand);
9  Artisan::add(new GSD\Commands\UncreateCommand);
10 ?>
```

Refactoring TodoList save()

Before actually testing this, we need to go and refactor that save() method to allow a new, optional option. Easy and quick to do.

Edit `ListInterface.php` and change the `save()` definition like below.

```
<?php
  /**
   * Save the task list
   * @param bool $force Force saving, even if list isn't dirty
   * @return ListInterface for method chaining
   */
  public function save($force = false);
?>
```

Edit `TodoList.php` and change the implementation of `save()` as specified below.

```
<?php
  /**
   * Save the list
   * @param bool $force Force saving, even if list isn't dirty
   * @return $this For method chaining
   * @throws RuntimeException If cannot save.
   */
  public function save($force = false)
  {
    if ($this->isDirty || $force)

    // rest of file is the same
?>
```

Now, wasn't that a painless refactoring? I could have done this when the `TodoList` class was created, back in Chapter 29. I wasn't aware that I'd need to force a list to save.

In some ways I don't like doing this. It forces us to know the implementation details of the `TodoList` and we must use this knowledge to code the implementation how it was coded.

So what's the alternative? To keep this detail totally within the `TodoList` itself. One way to do this would be to keep the starting state of all tasks (the position, the description, etc.) and whenever the `TodoList` wants to save itself it could check all the starting states with the existing states of the classes. That's a perfectly acceptable alternative. But I'm not going back to change the code now.

Testing EditTask

To test ... just play around with the edit task. Try going through all the possible permutations of usage.

It seems to work fine for me (notice the wishy washy way I said this ... *"seems to work fine"* that's standard programmer speak. It's used because although I'm confident it is working, I'm not 100%, bet-my-life-on-it positive. If you find a bug, let me know.)

Dogfooding

Let's mark the EditTaskCommand as done. And now we can move the ArchiveListCommand and UnarchiveListCommand as next actions.

Below is how I did it. Note if your list is different the numbers may be different.

```
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'

+----+------+---------------------------------------+-------+
| #  | Next | Description                           | Extra |
+----+------+---------------------------------------+-------+
| 1  | YES  | Remove blank line after gsd:list title |       |
| 2  |      | Create ArchiveListCommand              |       |
| 3  |      | Create EditTaskCommand                 |       |
| 4  |      | Create MoveTaskCommand                 |       |
| 5  |      | Create RemoveTaskCommand               |       |
| 6  |      | Create RenameListCommand               |       |
| 7  |      | Create UnarchiveListCommand            |       |
| 8  |      | Create shell gsd script                |       |
| 9  |      | Create web application wireframe       |       |
| 10 |      | List tasks in ListAllCommand           |       |
+----+------+---------------------------------------+-------+
~/gsd$ art gsd:edit 2 +gsd --action=yes
Task in gsd updated to: * Create ArchiveListCommand
~/gsd$ art gsd:edit 7 +gsd --action=yes
Task in gsd updated to: * Create UnarchiveListCommand
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'

+----+------+---------------------------------------+-------+
| #  | Next | Description                           | Extra |
+----+------+---------------------------------------+-------+
| 1  | YES  | Create ArchiveListCommand              |       |
```

```
| 2  | YES  | Create UnarchiveListCommand           |       |
| 3  | YES  | Remove blank line after gsd:list title |       |
| 4  |      | Create EditTaskCommand                |       |
| 5  |      | Create MoveTaskCommand                |       |
| 6  |      | Create RemoveTaskCommand              |       |
| 7  |      | Create RenameListCommand              |       |
| 8  |      | Create shell gsd script               |       |
| 9  |      | Create web application wireframe      |       |
| 10 |      | List tasks in ListAllCommand          |       |
+----+------+---------------------------------------+-------+
~/gsd$ art gsd:do 4 +gsd
Task 'Create EditTaskCommand' marked complete.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'

+---+------+---------------------------------------+--------------+
| # | Next | Description                           | Extra        |
+---+------+---------------------------------------+--------------+
| 1 | YES  | Create ArchiveListCommand             |              |
| 2 | YES  | Create UnarchiveListCommand           |              |
| 3 | YES  | Remove blank line after gsd:list title |              |
| 4 |      | Create MoveTaskCommand                |              |
| 5 |      | Create RemoveTaskCommand              |              |
| 6 |      | Create RenameListCommand              |              |
| 7 |      | Create shell gsd script               |              |
| 8 |      | Create web application wireframe      |              |
| 9 |      | List tasks in ListAllCommand          |              |
|   | done | Create EditTaskCommand                | Done 9/29/13 |
+---+------+---------------------------------------+--------------+
```

And I'm reminded to remove that blank line after gsd:list title by doing this. I had totally forgot about that, but since it's on the list now I know it'll get done.

Chapter 48 - ArchiveListCommand and UnarchiveListCommand

In This Chapter

In this chapter we'll created the ArchiveListCommand and its complement, the UnarchiveListCommand.

The Plan

The original thoughts on the console commands were.

```
$ gsd archive
$ gsd unarchive
```

Pretty simple. The goal of archiving lists is to move it into the `archived` directory. Remember, archived lists cannot be edited–no new tasks, no task editing, and no list editing. Let's say you have a project you want to put on hold to maybe, someday get back to. Archive it away. You can always *unarchive* it when the project becomes active again.

The Pseudo-code for archiving a list was:

```
Prompt user for list-id
If archived version exists
   Warn will overwrite
EndIf
Ask if they're sure
Load existing list
Save list as archive
Delete existing list
```

As I'm writing this I had the thought *"Chuck, should there be a –force option"*. I answered myself *"Nope."* The reason I don't have a –force option when archiving a list is because archiving isn't something set up to be automated. If later I change my mind, then I'd need to add the `+name` argument for `--force` to make any sense.

Hmmm. I'm rereading that paragraph and wondering how clear my meaning is. The primary purpose of a `--force` option is to eliminate prompting because it's a command you want to call automatically, like in a shell script or batch file. If I provided a `--force` option without the `+name` argument, you'd still be prompted for the list's name.

Anyway ... not gonna do a force option right now.

The Pseudo-code for unarchiving a list was:

```
Prompt user for archived list-id
If unarchived version exists
   Warn will overwrite
EndIf
Ask if they're sure
Load existing list
Save list as unarchive
Delete existing archived list
```

The unarchive logic is pretty much the same logic as archiving logic. The difference is the list's location. You see why we're doing these two commands together?

Creating the Commands

First, create the command skeleton for each command.

```
~/gsd$ art command:make ArchiveListCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
~/gsd$ art command:make UnarchiveListCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Then add the commands to `start/artisan.php`

```
<?php

Artisan::add(new GSD\Commands\AddTaskCommand);
Artisan::add(new GSD\Commands\ArchiveListCommand);
Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\DoTaskCommand);
Artisan::add(new GSD\Commands\EditListCommand);
Artisan::add(new GSD\Commands\EditTaskCommand);
Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\ListTasksCommand);
Artisan::add(new GSD\Commands\UnarchiveListCommand);
Artisan::add(new GSD\Commands\UncreateCommand);

?>
```

Updating ArchiveListCommand

Edit the newly created `ArchiveListCommand.php` file to match the following.

```
<?php namespace GSD\Commands;

use Config;
use Todo;

class ArchiveListCommand extends CommandBase {

  protected $name = 'gsd:archive';
  protected $description = 'Archive a todo list.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Get list name to archive
    $name = $this->askForListId();
```

```
20      if (is_null($name))
21      {
22        $this->outputErrorBox('*Archive aborted*');
23        return;
24      }
25      if (Config::get('todo.defaultList') == $name)
26      {
27        $this->outputErrorBox('Cannot archive default list');
28        return;
29      }
30
31      // Warn if list exists
32      if ($this->repository->exists($name, true))
33      {
34        $msg = "WARNING!\n\n"
35             . "  An archived version of the list '$name' exists.\n"
36             . "  This action will destroy the old archived version.";
37        $this->outputErrorBox($msg);
38      }
39      $result = $this->ask(
40        "Are you sure you want to archive '$name' (yes/no)?");
41      if ( ! str2bool($result))
42      {
43        $this->outputErrorBox('*Archive aborted*');
44        return;
45      }
46
47      // Archive the list
48      $list = Todo::get($name);
49      $list->archive();
50      $this->info("List '$name' has been archived");
51    }
52
53    /**
54     * No arguments.
55     */
56    protected function getArguments()
57    {
58      return array();
59    }
60
61    /**
```

```
62      * No options.
63      */
64     protected function getOptions()
65     {
66       return array();
67     }
68 }
69 ?>
```

Lines 1 - 9
Standard setup stuff

Lines 18 - 29
Here we use the `CommandBase` method `askForListId()` to get the name of the list. This will prompt the user. If they choose cancel we catch it. Also, if they try to archive the default list, we don't let them.

Lines 31 - 45
If a list of the same name already exists, then we output a big, red warning box. Regardless, we ask if they're sure. If they don't type 'yes' (or some string equivalent) then we abort.

Lines 47 - 50
Since the `ListInterface` has an `archive()` method, we use that to archive the list instead of following our pseudo-code.

Lines 56 to the end
The rest of the file returns empty arrays for arguments and options. We're not taking any arguments or options for this utility.

Give it a try

Create a list, archive it. Create a list of the same name and archive it. It all seems to work great ...

... but ...

There's a stinking bug in CommandBase. Uggh.

Fixing the CommandBase bug

The problem is with the `askForListId()`. If the method prompts for a list, it uses the prompt: *"Choose which list to destroy:"* and the cancel option says *"cancel - do not destroy a list"*.

We don't want that. So let's fix it in such a way that existing code won't break.

Update `CommandBase.php` and make the top of the file match what's below:

```
1  <?php namespace GSD\Commands;
2
3  use App;
4  use Config;
5  use Illuminate\Console\Command;
6  use Symfony\Component\Console\Input\InputOption;
7  use Symfony\Component\Console\Input\InputArgument;
8  use Todo;
9
10 class CommandBase extends Command {
11
12   protected $repository;
13   protected $nameArgumentDescription = 'List name.';
14   protected $taskNoDescription = null;
15   protected $askForListAction = 'destroy';
16
17   /**
18    * Constructor
19    */
20   public function __construct()
21   {
22     parent::__construct();
23     $this->repository = App::make('GSD\Repositories\TodoRepositoryInterface');
24   }
25
26   /**
27    * Prompt the user for a list id
28    * @param bool $existing Prompt for existing list or new list?
29    * @param bool $allowCancel Allow user to cancel
30    * @return mixed string list id or null if user cancels
31    */
32   public function askForListId($existing = true, $allowCancel = true,
33     $archived = false)
34   {
35     if ($existing)
36     {
37       $title = "Choose which list to $this->askForListAction:";
38       $abort = "cancel - do not $this->askForListAction a list";
39       $choices = Todo::allLists($archived);
40 ?>
```

Line 15

We're adding a new property ... the *action* when we ask the user for a list. The default is

'destroy' so the `askForListId()` method will present the same prompt as before.

Lines 37 and 38
Here we're using that new property.

Line 39
Another bug I noticed. We haven't ran into this yet. But when we're getting the list of all lists, what happens if we want to pick a list that's been archived? It wouldn't have worked. Now we pass the `$archived` value as I'm sure I meant to do initially.

Now edit `ArchiveListCommand.php` and add the following property to the top of the class.

```
1   protected $askForListAction = 'archive';
```

If you test the `art gsd:archive` command, it will prompt you correctly.

Is this the best bug fix?

Nope. The better solution would have been to provide an additional argument to the `askForListId()` method. The problem with the solution we implemented is that it's not immediately clear when we call the method what all the options (aka arguments) to the method are.

But it's an example of real-life code. I'm tempted to go back and do it right, but for now it's good enough and it works.

Updating UnarchiveListCommand

Unarchived will be slightly different than our archive implementation because the `ListInterface` does not have an `unarchive()` method to complement the `archive()` method. We could implement this within `ListInterface`, but I'll just add the logic to the `UnarchiveListCommand`, following the pseudo-code.

Update the `UnarchiveListCommand.php` to match what's below.

```php
<?php namespace GSD\Commands;

use App;
use Todo;

class UnarchiveListCommand extends CommandBase {

  protected $name = 'gsd:unarchive';
  protected $description = 'Unarchive a todo list.';
  protected $askForListAction = 'unarchive';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Prompt user for list name
    $name = $this->askForListId(true, true, true);
    if (is_null($name))
    {
      $this->outputErrorBox('*Unarchive aborted*');
      return;
    }

    // Warn if unarchived version exists
    if ($this->repository->exists($name, false))
    {
      $msg = "WARNING!\n\n"
           . "  An active version of the list '$name' exists.\n"
           . "  This action will destroy the active version,\n"
           . "  replacing it with the archived version.";
      $this->outputErrorBox($msg);
    }

    // Ask if user is sure?
    $result = $this->ask(
      "Are you sure you want to unarchive '$name' (yes/no)?");
    if ( ! str2bool($result))
    {
      $this->outputErrorBox('*Unarchive aborted*');
```

```
43        return;
44      }
45
46      // Load existing list and save as unarchived
47      $list = Todo::get($name, true);
48      $list->set('archived', false);
49      $list->save();
50
51      // Delete existing archived list
52      if ( ! $this->repository->delete($name, true))
53      {
54        $this->outputErrorBox('ERROR deleting archived version.');
55        return;
56      }
57      $this->info("List '$name' has been unarchived");
58    }
59
60    /**
61     * No arguments.
62     */
63    protected function getArguments()
64    {
65      return array();
66    }
67
68    /**
69     * No options.
70     */
71    protected function getOptions()
72    {
73      return array();
74    }
75  }
76  ?>
```

Lines 1 - 44

These lines are almost identical to those of the ArchiveListCommand. We don't need to use `Config`, but we do need `App`. And the words and methods that deal with *archive* are reversed.

Lines 46 - 49

Here we follow the pseudo-code, loading the archived list, changing the archive flag and saving it.

Lines 51 - 56
: We use the repository to do the deleting.

Lines 59 to end
: The rest of the file returns empty arrays for arguments and options. There aren't any arguments or options for this utility.

Give it a shot, unarchive and archive work now.

Dogfooding

Let's mark the tasks complete which we finished and decide what our next actions could be.

Below is how I did it. Your list numbers may be different if you've made any modifications to the `gsd` todo list.

```
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'

+---+------+-----------------------------------------+--------------+
| # | Next | Description                             | Extra        |
+---+------+-----------------------------------------+--------------+
| 1 | YES  | Create ArchiveListCommand               |              |
| 2 | YES  | Create UnarchiveListCommand             |              |
| 3 | YES  | Remove blank line after gsd:list title  |              |
| 4 |      | Create MoveTaskCommand                  |              |
| 5 |      | Create RemoveTaskCommand                |              |
| 6 |      | Create RenameListCommand                |              |
| 7 |      | Create shell gsd script                 |              |
| 8 |      | Create web application wireframe        |              |
| 9 |      | List tasks in ListAllCommand            |              |
|   | done | Create EditTaskCommand                  | Done 9/29/13 |
+---+------+-----------------------------------------+--------------+
~/gsd$ art gsd:do 2 +gsd
Task 'Create UnarchiveListCommand' marked complete.
~/gsd$ art gsd:do 1 +gsd
Task 'Create ArchiveListCommand' marked complete.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'

```

```
+---+------+---------------------------------------+--------------+
| # | Next | Description                           | Extra        |
+---+------+---------------------------------------+--------------+
| 1 | YES  | Remove blank line after gsd:list title |              |
| 2 |      | Create MoveTaskCommand                |              |
| 3 |      | Create RemoveTaskCommand              |              |
| 4 |      | Create RenameListCommand              |              |
| 5 |      | Create shell gsd script               |              |
| 6 |      | Create web application wireframe      |              |
| 7 |      | List tasks in ListAllCommand          |              |
|   | done | Create EditTaskCommand                | Done 9/29/13 |
|   | done | Create ArchiveListCommand             | Done 10/4/13 |
|   | done | Create UnarchiveListCommand           | Done 10/4/13 |
+---+------+---------------------------------------+--------------+
~/gsd$ art gsd:edit 2 +gsd -a yes
Task in gsd updated to: * Create MoveTaskCommand
~/gsd$ art gsd:edit 3 +gsd --action=yes
Task in gsd updated to: * Create RemoveTaskCommand
~/gsd$ art gsd:edit 4 +gsd --action=yes
Task in gsd updated to: * Create RenameListCommand
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'

+---+------+---------------------------------------+-------+
| # | Next | Description                           | Extra |
+---+------+---------------------------------------+-------+
| 1 | YES  | Create MoveTaskCommand                |       |
| 2 | YES  | Create RemoveTaskCommand              |       |
| 3 | YES  | Create RenameListCommand              |       |
| 4 | YES  | Remove blank line after gsd:list title |       |
| 5 |      | Create shell gsd script               |       |
| 6 |      | Create web application wireframe      |       |
| 7 |      | List tasks in ListAllCommand          |       |
+---+------+---------------------------------------+-------+
~/gsd$ art gsd:addtask "Create Appendix for Apache install" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create Appendix for nginx install" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Create Chapter on setting up webserver and hostname" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'
```

```
+----+------+-------------------------------------------------------+-------+
| #  | Next | Description                                           | Extra |
+----+------+-------------------------------------------------------+-------+
| 1  | YES  | Create MoveTaskCommand                                |       |
| 2  | YES  | Create RemoveTaskCommand                              |       |
| 3  | YES  | Create RenameListCommand                              |       |
| 4  | YES  | Remove blank line after gsd:list title                |       |
| 5  |      | Create Appendix for Apache install                    |       |
| 6  |      | Create Appendix for nginx install                     |       |
| 7  |      | Create Chapter on setting up webserver and hostname   |       |
| 8  |      | Create shell gsd script                               |       |
| 9  |      | Create web application wireframe                      |       |
| 10 |      | List tasks in ListAllCommand                          |       |
+----+------+-------------------------------------------------------+-------+
```

Line 1
: I listed out the current tasks to see the numbers

Lines 18 and 20
: I marked the two tasks we finished as complete. Notice I did them in reverse order. Since I know how the list is sorted, I felt safe doing it this way.

Line 22
: Then I listed the tasks to see those marked complete.

Line 39, 41, and 43
: Here I marked the three tasks as Next Actions that I know we're going to do next. Maybe not the very next chapter, but soon.

Line 45
: Again I listed the tasks, the next actions are all nicely on the top.

Line 59, 61, and 63
: Here I added three tasks I know will need to be done, not next, but early into the Part 4 of this book.

Line 66
: And one final listing of the tasks

Ain't dogfooding fun? Still plenty to do, but the console application is getting closed to being finished.

Chapter 49 - The RenameListCommand

In This Chapter

In this chapter we'll implement the RenameListCommand and, take care of the blank line after gsd:list title

Blank line after gsd:list title

The goal is to remove that single carriage return in the `ListTasksCommand` that puts a blank line after the title so the output is consistent with other commands. Only, I really hate doing such a little dinky change. I takes all of 30 seconds to do, but there was something I was thinking about back when the `ListTasksCommand` was first built: formatting of dates.

`ListTasksCommand` outputs dates in two places: the due date of a task and the completed date. Let's make the format of these dates configuration options.

Update `config/todo.php` to match what's below.

```
1  <?php
2  // Config for our awesome getting stuff done application
3  return array(
4    'folder' => '/home/chuck/Documents/gsd/',
5    'extension' => '.txt',
6    'listOrder' => array(
7      'inbox', 'actions', 'waiting', 'someday', 'calendar',
8    ),
9    'defaultList' => 'actions',      // default list when not specified
10   'noListPrompt' => true,          // true=prompt for list, false=use default
11   'dateCompleteFormat' => 'n/j/y', // date format for completed tasks
12   'dateDueFormat' => 'M-j',        // date format for due tasks
13 );
14 ?>
```

If you want to change these two formats for your locale, have at it. I only use the Month and Day for due tasks because I don't usually schedule things out more than a month or two in advance. *(Heck, look at this book. I only plan things out a chapter or two at a time.)*

Update `ListTasksCommand.php` to match what's below.

```
<?php namespace GSD\Commands;

use Config;
use Symfony\Component\Console\Input\InputOption;
use Todo;

class ListTasksCommand extends CommandBase {

  protected $name = 'gsd:list';
  protected $description = 'List tasks.';
  protected $nameArgumentDescription = 'List name to display tasks.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId();
    if (is_null($name))
    {
      $this->outputErrorBox("ListTasks aborted");
      return;
    }
    $list = Todo::get($name);

    $nextOnly = $this->option('action');
    $skipDone = $this->option('skip-done');
    if ($nextOnly and $skipDone)
    {
      $this->outputErrorBox(
        "Options --action and --skip-done can't be used together."
      );
```

```
35        return;
36      }
37
38      // Gather rows to display
39      $completeFmt = Config::get('todo.dateCompleteFormat');
40      $dueFmt = Config::get('todo.dateDueFormat');
41      $rows = array();
42      $rowNo = 1;
43      foreach ($list->tasks() as $task)
44      {
45        if ($task->isComplete())
46        {
47          if ($skipDone or $nextOnly) continue;
48          $rows[] = array(
49            '',
50            'done',
51            $task->description(),
52            'Done '.$task->dateCompleted()->format($completeFmt),
53          );
54        }
55        elseif ($task->isNextAction() or ! $nextOnly)
56        {
57          $next = ($task->isNextAction()) ? 'YES' : '';
58          $due = ($task->dateDue()) ?
59            'Due '.$task->dateDue()->format($dueFmt) : '';
60          $rows[] = array(
61            $rowNo++,
62            $next,
63            $task->description(),
64            $due,
65          );
66        }
67      }
68
69      // Output a pretty table
70      $title = ($nextOnly) ? "Next Actions" :
71        (($skipDone) ? "Active Tasks" : "All Tasks");
72      $this->info("$title in list '+$name'");
73      if (count($rows) == 0)
74      {
75        $this->error("Nothing found");
76        return;
```

```
77      }
78      $table = $this->getHelperSet()->get('table');
79      $table
80        ->setHeaders(array('#', 'Next', 'Description', 'Extra'))
81        ->setRows($rows)
82        ->render($this->getOutput());
83    }
84
85    /**
86     * Get the console command options.
87     *
88     * @return array
89     */
90    protected function getOptions()
91    {
92      return array_merge(array(
93        array('action', 'a', InputOption::VALUE_NONE,
94          'Show only next actions.', null),
95        array('skip-done', 'x', InputOption::VALUE_NONE,
96          'Skip completed actions.', null),
97      ), parent::getOptions());
98    }
99  }
100 ?>
```

The newline was removed from the string on line #72. On lines #39 and #40 we pull the date formats and use them later in the loop.

Small changes, but we did something significant enough to warrant changing the source code ... the addition of the date formats. Yes, removing the line feed was the only requirement (from our list of todo tasks), but doesn't it give you a good feeling to go beyond and make things better?

The Plan for RenameListCommand

The original thought on the console command was simple.

```
1  $ gsd rename
```

And the pseudo-code shows that we planned on prompting for everything.

```
Prompt user for archived or unarchived
Prompt user for appropriate list-id
Prompt user for new list-id
Load existing list
Save as new list-id
Delete existing list
```

I'm going to skip the first prompt in the pseudo-code. The only way for a user to rename an archived list will be to use a command line option `--archived`.

Creating the RenameListCommand

First, create the command skeleton.

```
~/gsd$ art command:make RenameListCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Update `RenameListCommand.php` to match what's below.

```
<?php namespace GSD\Commands;

use Config;
use Symfony\Component\Console\Input\InputOption;
use Todo;

class RenameListCommand extends CommandBase {

  protected $name = 'gsd:rename';
  protected $description = 'Rename a list.';
  protected $askForListAction = 'rename';

  /**
   * Execute the console command.
   *
```

```
16      * @return void
17      */
18     public function fire()
19     {
20       // Get archived flag and list to rename
21       $archived = $this->option('archived');
22       $name = $this->askForListId(true, true, $archived);
23       if (is_null($name))
24       {
25         $this->outputErrorBox('*aborted*');
26         return;
27       }
28       if ( ! $archived && Config::get('todo.defaultList') == $name)
29       {
30         $this->outputErrorBox('Cannot rename default list');
31         return;
32       }
33
34       // Prompt for new list name
35       $newName = $this->askForListId(false, true, $archived);
36       if (is_null($name))
37       {
38         $this->outputErrorBox('*aborted*');
39         return;
40       }
41
42       // Load existing list, save with new name
43       $list = Todo::get($name, $archived);
44       $newList = clone $list;
45       $newList->set('id', $newName);
46       $newList->save();
47
48       // Delete existing list and we're done
49       $list->delete();
50       $listType = ($archived) ? 'Archived list' : 'List';
51       $this->info($listType . " '$name' renamed to '$newName'");
52     }
53
54     /**
55      * No arguments.
56      */
57     protected function getArguments()
```

```
58    {
59      return array();
60    }
61
62    /**
63     * Just the --archived option
64     */
65    protected function getOptions()
66    {
67      return array(
68        array('archived', 'a', InputOption::VALUE_NONE,
69          'Use archived lists?', null),
70      );
71    }
72  }
73  ?>
```

Lines 1 - 11
All standard stuff.

Lines 20 - 32
Let the user select which list to rename, abort if they cancel, and if we're not renaming an archived list then make sure it's not the default list.

Lines 34 - 40
Ask for the new name and abort if the user changes their mind.

Lines 42 - 46
Pull the list object, clone it because we'll want to use it shortly without modification, change the list's name and save it.

Lines 48 - 51
Delete the original list and output a success message. Note that the `ListInterface` doesn't have a `delete()` method ... yet.

Lines 57 - the end
No arguments and only the one `--archived` option.

Edit `start/artisan.php` to make artisan aware of the new command

```
<?php
Artisan::add(new GSD\Commands\AddTaskCommand);
Artisan::add(new GSD\Commands\ArchiveListCommand);
Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\DoTaskCommand);
Artisan::add(new GSD\Commands\EditListCommand);
Artisan::add(new GSD\Commands\EditTaskCommand);
Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\ListTasksCommand);
Artisan::add(new GSD\Commands\RenameListCommand);
Artisan::add(new GSD\Commands\UnarchiveListCommand);
Artisan::add(new GSD\Commands\UncreateCommand);
?>
```

We could attempt running this new command now, but first let's implement the `ListInterface::delete()` method.

Implementing ListInterface::delete()

I probably should have added this during the last chapter when I made use of the repository within the `UnarchiveListCommand`, but at the time I quickly scanned ahead to see what was left to do and didn't think we'd need to delete a list again.

I was wrong.

But again, one of the nice things about programming is when you make a mistake, you can quietly fix it so nobody's the wiser.

Modify the `ListInterface.php` file and add the following within the **List operations** section of the class

```
<?php
  /**
   * Delete the task list
   * @return boolean TRUE on success
   */
  public function delete();
?>
```

Modify the `TodoList.php` file and add the following method within the **List operations** section of the class

```
<?php
  /**
   * Delete the task list
   * @return boolean TRUE on success
   */
  public function delete()
  {
    return $this->repository->delete($this->id(), $this->isArchived());
  }
?>
```

Wow. That was amazingly trivial to implement.

Give it a shot. Rename some lists. Everything seems to be working great.

Dogfooding

Let's mark the tasks completed that we've finished. And, of course, I thought of a few more things to do.

Below is how I did it. Your list numbers may be different if you've made any modifications to the list.

```
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'
+----+------+-------------------------------------------------------+-------+
| #  | Next | Description                                           | Extra |
+----+------+-------------------------------------------------------+-------+
| 1  | YES  | Create MoveTaskCommand                                |       |
| 2  | YES  | Create RemoveTaskCommand                              |       |
| 3  | YES  | Create RenameListCommand                              |       |
| 4  | YES  | Remove blank line after gsd:list title                |       |
| 5  |      | Create Appendix for Apache install                    |       |
| 6  |      | Create Appendix for nginx install                     |       |
| 7  |      | Create Chapter on setting up webserver and hostname   |       |
| 8  |      | Create shell gsd script                               |       |
| 9  |      | Create web application wireframe                      |       |
| 10 |      | List tasks in ListAllCommand                          |       |
+----+------+-------------------------------------------------------+-------+
~/gsd$ art gsd:do 4 +gsd
Task 'Remove blank line after gsd:list title' marked complete.
```

```
~/gsd$ art gsd:do 3 +gsd
Task 'Create RenameListCommand' marked complete.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'
# Too wide to display so I'm editing a task
~/gsd$ art gsd:edit 5 +gsd \
>-d "Chapter on setting up webserver"
Task in gsd updated to: - Chapter on setting up webserver
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------------+--------------+
| # | Next | Description                           | Extra        |
+---+------+---------------------------------------+--------------+
| 1 | YES  | Create MoveTaskCommand                |              |
| 2 | YES  | Create RemoveTaskCommand              |              |
| 3 |      | Chapter on setting up webserver       |              |
| 4 |      | Create Appendix for Apache install    |              |
| 5 |      | Create Appendix for nginx install     |              |
| 6 |      | Create shell gsd script               |              |
| 7 |      | Create web application wireframe      |              |
| 8 |      | List tasks in ListAllCommand          |              |
|   | done | Create EditTaskCommand                | Done 9/29/13 |
|   | done | Create ArchiveListCommand             | Done 10/4/13 |
|   | done | Create UnarchiveListCommand           | Done 10/4/13 |
|   | done | Create RenameListCommand              | Done 10/5/13 |
|   | done | Remove blank line after gsd:list title | Done 10/5/13 |
+---+------+---------------------------------------+--------------+
~/gsd$ art gsd:addtask "Add CommandBase::abort()" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Check gsd help consistency" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Add $prompt to askForListId()" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:addtask "Use ListInterface::delete()" +gsd
Todo successfully added to gsd
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'
+----+------+-----------------------------------+-------+
| #  | Next | Description                       | Extra |
+----+------+-----------------------------------+-------+
| 1  | YES  | Create MoveTaskCommand            |       |
| 2  | YES  | Create RemoveTaskCommand          |       |
```

```
| 3  |      | Add CommandBase::abort()           |       |
| 4  |      | Add to askForListId()              |       |
| 5  |      | Chapter on setting up webserver    |       |
| 6  |      | Check gsd help consistency         |       |
| 7  |      | Create Appendix for Apache install |       |
| 8  |      | Create Appendix for nginx install  |       |
| 9  |      | Create shell gsd script            |       |
| 10 |      | Create web application wireframe   |       |
| 11 |      | List tasks in ListAllCommand       |       |
| 12 |      | Use ListInterface::delete()        |       |
+----+------+------------------------------------+-------+
~/gsd$ art gsd:edit 3 +gsd -a "yes"
Task in gsd updated to: * Add CommandBase::abort()
~/gsd$ art gsd:edit 4 +gsd -a "yes"
Task in gsd updated to: * Add to askForListId()
~/gsd$ art gsd:edit 6 +gsd -a "yes"
Task in gsd updated to: * Check gsd help consistency
~/gsd$ art gsd:edit 11 +gsd -a "yes"
Task in gsd updated to: * List tasks in ListAllCommand
~/gsd$ art gsd:edit 12 +gsd -a "yes"
Task in gsd updated to: * Use ListInterface::delete()
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'
+----+------+------------------------------------+-------+
| #  | Next | Description                        | Extra |
+----+------+------------------------------------+-------+
| 1  | YES  | Add CommandBase::abort()           |       |
| 2  | YES  | Add to askForListId()              |       |
| 3  | YES  | Check gsd help consistency         |       |
| 4  | YES  | Create MoveTaskCommand             |       |
| 5  | YES  | Create RemoveTaskCommand           |       |
| 6  | YES  | List tasks in ListAllCommand       |       |
| 7  | YES  | Use ListInterface::delete()        |       |
| 8  |      | Chapter on setting up webserver    |       |
| 9  |      | Create Appendix for Apache install |       |
| 10 |      | Create Appendix for nginx install  |       |
| 11 |      | Create shell gsd script            |       |
| 12 |      | Create web application wireframe   |       |
+----+------+------------------------------------+-------+
```

Explanation of above

Lines 1 - 20
I listed our tasks and marked the two things on the list that we completed as done.

Line 21
I listed the tasks again, this time with the completed items showing and the list was too wide to fit in this book. Not a problem in the console window, and usually if this happens I just edit away some extra spaces. (Of course, if you're looking at this on an iPhone there's still lots of wrapping going on.)

Lines 24 - 25
Instead I shorted the "Create Chapter on setting up webserver and hostname" task description to be shorter.

Lines 27 - 45
Now the list of everything, todo and done, in this project fits nicely.

Lines 46 - 53
Added several more things to do. Looks like some refactoring is coming up.

Lines 54 - 71
I relisted the tasks not complete. Dang, I should have added those last ones as Next Actions

Lines 72 - 81
No matter. I edited each of the tasks that should be finished soon as Next Actions.

Lines 82 - 99
And the final list of things to do.

I'm Getting Stuff Done, knocking things off the list, yet somehow the todo list is still growing.

Chapter 50 - Refactoring again

In This Chapter

In this chapter we'll knock all the refactoring items off our *gsd* todo list.

Will this be the last time we refactor in this book? I can't say for sure, but if I was a betting man I'd put money on there being more refactoring in the future. It's part of the process–well, my process anyway.

I'm going to not show all the source code modified until after all the modifications have occurred. Instead, on each task I'll note what was changed where and why.

Adding CommandBase::abort()

There seems to be an awful lot of instances within the commands where the `outputErrorBox()` method is called with some type of error message and the program execution ends. Since this happens so frequently, it makes sense to implement an `abort()` method in `CommandBase`.

But, will adding an `abort()` method break something? `CommandBase` is a subclass of `Illuminate\Console\Commar` which is a subclass of something else. You can dig through Laravel's source, then Symfony's source code, tracing your way up the class hierarchy to see if there's any `abort()` method implemented, but there's an easier way.

Just call `abort()` and see what happens.

Modify `AddTaskCommand.php` and place the following code at the top of the `fire()` method.

```
1   $this->abort();
2   dd('abort exists');
```

If we get an exception, **Method Not Found** or something similar, then we're good. If "abort exists" outputs, then not good.

Now try running the add task command

```
~/gsd$ art gsd:addtask x
PHP Fatal error:  Call to undefined method
  GSD\Commands\AddTaskCommand::abort() in
  /home/chuck/gsd/app/src/GSD/Commands/AddTaskCommand.php on line 22
  [snip]
```

Cool. The method `abort()` is safe to add. **Remove those two lines from AddTaskCommand.php**.

Add the new method to `CommandBase.php`

```
  /**
   * Output an error message and die
   * @param string $message Optional message to output
   */
  protected function abort($message = '*aborted*')
  {
    $this->outputErrorBox($message);
    exit;
  }
```

I've went through every command and every place that followed the pattern of outputting the error box and exiting, I've replaced with a call to `abort()`. If any exceptions are thrown within the `fire()` method, then those have been replaced with a call to `abort()`, also.

Here's a list of the files and number of changes.

- `AddTaskCommand.php` - 2 changes
- `ArchiveListCommand.php` - 3 changes
- `CreateCommand.php` - 4 changes
- `DoTaskCommand.php` - 2 changes
- `EditListCommand.php` - 1 change
- `EditTaskCommand.php` - 3 changes
- `ListTasksCommand.php` - 3 changes *(Also changed a call to `error()`)*
- `RenameListCommand.php` - 3 changes
- `UnarchiveListCommand.php` - 3 changes
- `UncreateCommand.php` - 3 changes

Dang. I should have counted lines of code before that and after. Oh well, that probably saves about 30 lines of code. And it makes things more consistent.

Add to askForListId()

Instead of using the property `$askForListAction` I'm going to add a fourth option to `askForListId()`. You knew I was going to change that, didn't you?

Careful of too many options

The `askForListId()` is getting dangerously close to too many options. This can be a sign that the method's getting too complicated and needs to be refactored.

I'm not changing the property `$taskNoDescription` because its existence tells the `CommandBase`'s `getArguments()` method to provide the optional task-number argument. This property kills two birds with one stone. Perhaps, ideally, the `getTaskNo()` should take a title and the property should only indicate if the argument's used. But I'm not changing that at the moment.

Here's a list of what changed in each file.

CommandBase.php
: Remove the `$askForListAction` property. Add `$selectTitle` argument to `askForListId()` and use it. Add a `$selectTitle` argument to `getListId()` and use it when calling `askForListId()`.

AddTaskCommand.php
: Provide a title on the `getListId()` call.

ArchiveListCommand.php
: Remove the `$askForListAction` property. Add title to `askForListId()` call.

DoTaskCommand.php
: Add title on the `getListId()` call.

EditListCommand.php
: Add title on the `getListId()` call.

EditTaskCommand.php
: Add title on the `getListId()` call.

ListTasksCommand.php
: Add title on the `getListId()` call.

RenameListCommand.php
: Remove `$askForListActon` property. Add title to `askForListId()` call.

UnarchiveListCommand.php
Remove `$askForListActon` property. Add title to `askForListId()` call.

UncreateListCommand.php
Add title to `askForListId()` call.

Check gsd help consistency

I want to do two things here.

1. Make sure the help on all the commands is consistent. This means capitalization and punctuation usage.
2. Make sure every method has an appropriate docblock with all arguments (unless the function doesn't really need them.)

This second item is what I call a **code trace**.

> **Code Trace**
>
> The term `code trace` can mean several things. For me, it is examining every line of code with a specific intention in mind. If the intent is to follow the logic, then the code is examined and followed as it would execute. My intent is to trace the docblocks. So that's all I'm focusing on.

Here's a list of the files and what changed in each.

TodoRepositoryInterface.php
Added to docblocks for delete().

TodoRepository.php
Added to docblocks for delete() and fullpath().

UncreateCommand.php
Removed docblock for common properties.

ListAllCommand.php
Derived from CommandBase instead of Command. Removed docblock for common properties. Capitalized help for –archived.

CreateCommand.php
Added $nameArgumentDescription property. Remove docblocks for common properties. Remove getArguments() method to use CommandBase's default.

CommandBase.php

Added to docblock for askForListId() and getListId(). Added period to `--listname` option help, updated `getTaskNo()` to replace ending period with a colon when list is selected.

Whew! Lots of little, bitty things.

Use ListInterface::delete()

I've changed my mind on this one. There's two places we could use a `ListInterface::delete()`: the UncreateCommand and UnarchiveListCommand. But after doing the code trace, it doesn't really seem that important to keep the command code from using the repository. I mean, it's a property set up by `CommandBase` for gosh sakes. I'm not sure what was going through my head with wanting to implement this one.

Unfortunately, we can't delete tasks yet. Which means the *gsd* todo list will still keep this task. *(Next chapter we'll delete this task.)*

The Changed Files

The next umpteen pages are source code dumps of each file changed in this chapter.

AddTaskCommand.php

```
1  <?php namespace GSD\Commands;
2
3  use App;
4  use Illuminate\Console\Command;
5  use Symfony\Component\Console\Input\InputOption;
6  use Symfony\Component\Console\Input\InputArgument;
7  use Todo;
8
9  class AddTaskCommand extends CommandBase {
10
11   protected $name = 'gsd:addtask';
12   protected $description = 'Add a new task to a list.';
13   protected $nameArgumentDescription = 'List name to add the task to.';
14
15   /**
16    * Execute the console command.
17    *
18    * @return void
```

```
19      */
20     public function fire()
21     {
22       $name = $this->getListId('Select list to add the task to:');
23       if (is_null($name))
24       {
25         $this->abort();
26       }
27       $list = Todo::get($name);
28
29       $task = App::make('GSD\Entities\TaskInterface');
30       if ( ! $task->setFromString($this->argument('task')))
31       {
32         $this->abort('Cannot parse task string');
33       }
34       $type = 'Todo';
35       if ($this->option('action'))
36       {
37         $task->setIsNextAction(true);
38         $type = 'Next Action';
39       }
40       $list->taskAdd($task);
41       $list->save();
42       $this->info("$type successfully added to $name");
43     }
44
45     /**
46      * Get the console command arguments.
47      */
48     protected function getArguments()
49     {
50       return array_merge(array(
51         array('task', InputArgument::REQUIRED,
52           "The task's description."),
53       ), parent::getArguments());
54     }
55
56     /**
57      * Get the console command options.
58      */
59     protected function getOptions()
60     {
```

```
61      return array_merge(parent::getOptions(), array(
62        array('action', 'a', InputOption::VALUE_NONE,
63          'Make task a Next Action.'),
64      ));
65    }
66  }
67  ?>
```

ArchiveListCommand.php

```
1   <?php namespace GSD\Commands;
2
3   use Config;
4   use Todo;
5
6   class ArchiveListCommand extends CommandBase {
7
8     protected $name = 'gsd:archive';
9     protected $description = 'Archive a todo list.';
10
11    /**
12     * Execute the console command.
13     *
14     * @return void
15     */
16    public function fire()
17    {
18      // Get list name to archive
19      $selectTitle = 'Select list to archive:';
20      $name = $this->askForListId(true, true, false, $selectTitle);
21      if (is_null($name))
22      {
23        $this->abort();
24      }
25      if (Config::get('todo.defaultList') == $name)
26      {
27        $this->abort('Cannot archive default list');
28      }
29
30      // Warn if list exists
31      if ($this->repository->exists($name, true))
32      {
```

```
33        $msg = "WARNING!\n\n"
34           . "  An archived version of the list '$name' exists.\n"
35           . "  This action will destroy the old archived version.";
36        $this->outputErrorBox($msg);
37      }
38      $result = $this->ask(
39        "Are you sure you want to archive '$name' (yes/no)?");
40      if ( ! str2bool($result))
41      {
42        $this->abort();
43      }
44
45      // Archive the list
46      $list = Todo::get($name);
47      $list->archive();
48      $this->info("List '$name' has been archived");
49    }
50
51    /**
52     * No arguments.
53     */
54    protected function getArguments()
55    {
56      return array();
57    }
58
59    /**
60     * No options.
61     */
62    protected function getOptions()
63    {
64      return array();
65    }
66  }
67  ?>
```

CommandBase.php

```php
<?php namespace GSD\Commands;

use App;
use Config;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class CommandBase extends Command {

  protected $repository;
  protected $nameArgumentDescription = 'List name.';
  protected $taskNoDescription = null;

  /**
   * Constructor
   */
  public function __construct()
  {
    parent::__construct();
    $this->repository = App::make('GSD\Repositories\TodoRepositoryInterface');
  }

  /**
   * Prompt the user for a list id
   * @param bool $existing Prompt for existing list or new list?
   * @param bool $allowCancel Allow user to cancel
   * @param bool $archived Prompt for archived list?
   * @param string $selectTitle Title to use if list selection occurs.
   * @return mixed string list id or null if user cancels
   */
  public function askForListId($existing = true, $allowCancel = true,
    $archived = false, $selectTitle = 'Select a list:')
  {
    if ($existing)
    {
      $abort = "Cancel";
      $choices = Todo::allLists($archived);
      if (count($choices) == 0)
      {
        throw new \RuntimeException('No lists to choose from');
```

```
43        }
44        $result = pick_from_list($this, $selectTitle, $choices, 0, $abort);
45        if ($result == -1)
46        {
47          return null;
48        }
49        return $choices[$result-1];
50      }
51
52      $prompt = 'Enter name of new';
53      if ($archived) $prompt .= ' archived';
54      $prompt .= ' list';
55      if ($allowCancel) $prompt .= ' (enter to cancel)';
56      $prompt .= '?';
57      while(true)
58      {
59        if ( ! ($result = $this->ask($prompt)))
60        {
61          if ($allowCancel)
62          {
63            return null;
64          }
65          $this->outputErrorBox('You must enter something');
66        }
67        else if ($this->repository->exists($result, $archived))
68        {
69          $this->outputErrorBox("You already have a list named '$result'");
70        }
71        else
72        {
73          return $result;
74        }
75      }
76    }
77
78    /**
79     * Output an error box
80     * @param string $message The message
81     */
82    protected function outputErrorBox($message)
83    {
84      $formatter = $this->getHelperSet()->get('formatter');
```

```
85      $block = $formatter->formatBlock($message, 'error', true);
86      $this->line('');
87      $this->line($block);
88      $this->line('');
89    }
90
91    /**
92     * The console command arguments. Derived classes could replace this
93     * method entirely, or merge its own arguments with them
94     *
95     * @return array of argument definitions
96     */
97    protected function getArguments()
98    {
99      $args = array();
100     if ( ! is_null($this->taskNoDescription))
101     {
102       $args[] = array(
103         'task-number',
104         InputArgument::OPTIONAL,
105         $this->taskNoDescription
106       );
107     }
108     $args[] = array(
109       '+name',
110       InputArgument::OPTIONAL,
111       $this->nameArgumentDescription
112     );
113     return $args;
114   }
115
116   /**
117    * The console command options. Derived classes could replace this
118    * method entirely, or merge their own options with these.
119    *
120    * @return array
121    */
122   protected function getOptions()
123   {
124     return array(
125       array('listname', 'l', InputOption::VALUE_REQUIRED,
126         "Source of list name, 'prompt' or 'default'."),
```

```
127       );
128     }
129
130     /**
131      * Get the list id (of existing lists).
132      *
133      * This can happen in a variety of ways. If specified as an argument, then
134      * it's returned (without the + of course). Otherwise, look to see if the
135      * `--listname` argument is used and determine the list accordingly.
136      * Finally, we fallback to the method specified by Config's
137      * 'app.gsd.noListPrompt' setting
138      *
139      * @param string $selectTitle Title to use if list selection occurs
140      * @return $string Existing list id (or null if user aborts)
141      * @throws InvalidArgumentException If something's not right
142      */
143     protected function getListId($selectTitle = 'Select a list:')
144     {
145       $archived = $this->input->hasOption('archived') and
146                   $this->option('archived');
147       $name = $this->argument('+name');
148       $listnameOption = $this->option('listname');
149       if ($name)
150       {
151         $name = substr($name, 1);
152         if ( ! is_null($listnameOption))
153         {
154           throw new \InvalidArgumentException(
155             'Cannot specify $name and --listname together');
156         }
157       }
158       else
159       {
160         if (is_null($listnameOption))
161         {
162           $listnameOption = Config::get('todo.noListPrompt') ?
163             'prompt' : 'config';
164         }
165         if ($listnameOption == 'prompt')
166         {
167           $name = $this->askForListId(true, true, $archived, $selectTitle);
168           if (is_null($name))
```

```
169         {
170           return null;
171         }
172       }
173       else
174       {
175         $name = Config::get('todo.defaultList');
176       }
177     }
178
179     // Throw error if list doesn't exist
180     if ( ! $this->repository->exists($name, $archived))
181     {
182       $archived = ($archived) ? '(archived) ' : '';
183       throw new \InvalidArgumentException(
184         "List $archived'$name' not found");
185     }
186     return $name;
187   }
188
189
190   /**
191    * Get the task # of a list, either from the argument or prompt the user.
192    * Keep in mind the # present to the user always begins with 1, but the
193    * number we return is always one less (starting with 0)
194    *
195    * @param ListInterface $list The Todo List
196    * @param bool $showNext Show next actions in prompt list
197    * @param bool $showNormal Show normal tasks in prompt list
198    * @param bool $showComplete Show completed tasks in prompt list
199    * @return mixed NULL if user aborts, otherwise integer of task number
200    */
201   protected function getTaskNo(\GSD\Entities\ListInterface $list, $showNext,
202     $showNormal, $showComplete)
203   {
204     // Return the # if provided on command line
205     $taskNo = $this->argument('task-number');
206     if ( ! is_null($taskNo))
207     {
208       return (int)$taskNo - 1;
209     }
210
```

```
211       // Build list of tasks
212       $tasks = array();
213       foreach ($list->tasks() as $task)
214       {
215         if ($task->isComplete())
216         {
217           if ($showComplete)
218             $tasks[] = (string)$task;
219         }
220         elseif ($task->isNextAction())
221         {
222           if ($showNext)
223             $tasks[] = (string)$task;
224         }
225         elseif ($showNormal)
226         {
227           $tasks[] = (string)$task;
228         }
229       }
230
231       // Let user pick from list, return result
232       $selectTitle = rtrim($this->taskNoDescription, '.') . ':';
233       $result = pick_from_list($this, $selectTitle, $tasks, 0, "Cancel");
234       if ($result == -1)
235       {
236         return null;
237       }
238       return $result - 1;
239     }
240
241     /**
242      * Output an error message and die
243      * @param string $message Optional message to output
244      */
245     protected function abort($message = '*aborted*')
246     {
247       $this->outputErrorBox($message);
248       exit;
249     }
250   }
251   ?>
```

CreateCommand.php

```
<?php namespace GSD\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class CreateCommand extends CommandBase {

  protected $name = 'gsd:create';
  protected $description = 'Create new list.';
  protected $nameArgumentDescription = 'List name to create.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Get options and arguments
    $name = $this->argument('+name');
    $title = $this->option('title');
    $subtitle = $this->option('subtitle');

    // Prompt for everything
    if (all_null($name, $title, $subtitle))
    {
      if ( ! ($name = $this->askForListId(false, true)))
      {
        $this->abort();
      }
      $title = $this->ask("Enter list title (enter to skip)?");
      $subtitle = $this->ask("Enter list subtitle (enter to skip)?");
    }

    // Validate arguments
    else if (is_null($name))
    {
      $this->abort('Must specify +name if title or subtitle used');
```

```
41      }
42      else if ($name[0] != '+')
43      {
44        $this->abort('The list name must begin with a plus (+)');
45      }
46      else
47      {
48        $name = substr($name, 1);
49        if ($this->repository->exists($name))
50        {
51          $this->abort("The list '$name' already exists");
52        }
53      }
54
55      // Create the list, defaulting title if needed
56      $title = ($title) ? : ucfirst($name);
57      $list = Todo::makeList($name, $title);
58
59      // Set the subtitle if needed
60      if ($subtitle)
61      {
62        $list->set('subtitle', $subtitle)->save();
63      }
64
65      $this->info("List '$name' successfully created");
66    }
67
68    /**
69     * Get the console command options.
70     *
71     * @return array
72     */
73    protected function getOptions()
74    {
75      return array(
76        array('title', 't', InputOption::VALUE_REQUIRED,
77          'Title of list.', null),
78        array('subtitle', 's', InputOption::VALUE_REQUIRED,
79          'Subtitle of list.', null),
80      );
81    }
82  }
```

```
?>
```

DoTaskCommand.php

```
<?php namespace GSD\Commands;

use Todo;

class DoTaskCommand extends CommandBase {

  protected $name = 'gsd:do';
  protected $description = 'Mark a task as complete.';
  protected $nameArgumentDescription = 'List name with completed task.';
  protected $taskNoDescription = 'Task # to mark complete.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId('Select list with task to mark complete:');
    if (is_null($name))
    {
      $this->abort();
    }
    $list = Todo::get($name);

    $taskNo = $this->getTaskNo($list, true, true, false);
    if (is_null($taskNo))
    {
      $this->abort();
    }

    $description = $list->taskGet($taskNo, 'description');
    $list->taskSet($taskNo, 'isComplete', true)
         ->save();
    $this->info("Task '$description' marked complete.");
  }
}
?>
```

EditListCommand.php

```
<?php namespace GSD\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Todo;

class EditListCommand extends CommandBase {

  protected $name = 'gsd:editlist';
  protected $description = "Edit a list's title or subtitle.";
  protected $nameArgumentDescription = "List name to edit.";

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId('Select list to edit:');
    if (is_null($name))
    {
      $this->abort();
    }
    $list = Todo::get($name);

    $title = $this->option('title');
    $subtitle = $this->option('subtitle');

    if (all_null($title, $subtitle))
    {
      $this->info(sprintf("Editing '%s'", $name));
      $this->line('');
      $title = $this->ask("Enter list title (enter to skip)?");
      $subtitle = $this->ask("Enter list subtitle (enter to skip)?");
      $this->line('');
      if (all_null($title, $subtitle))
      {
        $this->comment('Nothing changed. List not udpated.');
        return;
```

```
41        }
42      }
43
44      if ($title)
45      {
46        $list->set('title', $title);
47      }
48      if ($subtitle)
49      {
50        $list->set('subtitle', $subtitle);
51      }
52      $list->save();
53      $this->info(sprintf("List '%s' updated", $name));
54    }
55
56    /**
57     * Get the console command options.
58     */
59    protected function getOptions()
60    {
61      return array_merge(parent::getOptions(), array(
62        array('title', 't', InputOption::VALUE_REQUIRED,
63          'Title of list.', null),
64        array('subtitle', 's', InputOption::VALUE_REQUIRED,
65          'Subtitle of list.', null),
66      ));
67    }
68
69  }
70  ?>
```

EditTaskCommand.php

```php
<?php namespace GSD\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class EditTaskCommand extends CommandBase {

  protected $name = 'gsd:edit';
  protected $description = 'Edit a task.';
  protected $nameArgumentDescription = 'List name with task to edit.';
  protected $taskNoDescription = 'Task # to edit.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Should we prompt for everything?
    $promptAll = all_null(
      $this->argument('+name'),
      $this->argument('task-number'),
      $this->option('descript'),
      $this->option('action')
    );

    // Get list
    $name = $this->getListId('Select list with task to edit:');
    if (is_null($name))
    {
      $this->abort();
    }
    $list = Todo::get($name);

    // Get task-number
    $taskNo = $this->getTaskNo($list, true, true, false);
    if (is_null($taskNo))
    {
      $this->abort();
```

```
43      }
44
45      $currDescript = $list->taskGet($taskNo, 'description');
46      $currAction = $list->taskGet($taskNo, 'isNextAction');
47
48      // Prompt for description and next action
49      if ($promptAll)
50      {
51        $currActionState = ($currAction) ? 'is' : 'is not';
52        $this->line("Current description: $currDescript");
53        $descript = $this->ask("New description (enter to skip)?");
54        $this->line("Task $currActionState currently a Next Aciton.");
55        $next = $this->ask("Is Next Action (enter skip, yes or no)?");
56      }
57
58      // Pull description and next action from command
59      else
60      {
61        $descript = $this->option('descript');
62        $next = $this->option('action');
63      }
64      $action = is_null($next) ? null : str2bool($next);
65
66      if ((is_null($descript) || $descript == $currDescript) &&
67          (is_null($action)   || $action == $currAction))
68      {
69        $this->abort("Nothing changed");
70      }
71
72      // Make changes and save the list
73      $task = $list->task($taskNo);
74      if ( ! is_null($action))
75      {
76        $task->setIsNextAction($action);
77      }
78      if ( ! is_null($descript))
79      {
80        $task->setDescription($descript);
81      }
82      $list->save(true);
83
84      $this->info("Task in $name updated to: ".(string)$task);
```

```
85    }
86
87    /**
88     * Get the console command options.
89     */
90    protected function getOptions()
91    {
92      return array_merge(array(
93        array('descript', 'd', InputOption::VALUE_REQUIRED,
94          'New description for task.'),
95        array('action', 'a', InputOption::VALUE_REQUIRED,
96          'Is task a next action (yes|no).'),
97      ), parent::getOptions());
98    }
99  }
100 ?>
```

ListAllCommand.php

```
1   <?php namespace GSD\Commands;
2
3   use Symfony\Component\Console\Input\InputOption;
4   use Symfony\Component\Console\Input\InputArgument;
5
6   class ListAllCommand extends CommandBase {
7
8     protected $name = 'gsd:listall';
9     protected $description = 'Lists all todo lists (and possibly tasks).';
10
11    /**
12     * Execute the console command.
13     *
14     * @return void
15     */
16    public function fire()
17    {
18      $archived = $this->option('archived');
19      $title = 'Listing all ';
20      if ($archived) $title .= 'archived ';
21      $title .= 'lists';
22      $this->info($title);
23
```

```
24      $lists = \Todo::allLists($archived);
25      $lists = $this->sortListIds($lists);
26
27      $headers = array('list', 'next', 'todos', 'completed');
28      $rows = array();
29      foreach ($lists as $listId)
30      {
31        $list = \Todo::get($listId, $archived);
32        $rows[] = array(
33          $listId,
34          $list->taskCount('next'),
35          $list->taskCount('todo'),
36          $list->taskCount('done'),
37        );
38      }
39
40      // Output a pretty table
41      $table = $this->getHelperSet()->get('table');
42      $table
43        ->setHeaders($headers)
44        ->setRows($rows)
45        ->render($this->getOutput());
46    }
47
48    /**
49     * Sort the list ids
50     */
51    protected function sortListIds(array $listIds)
52    {
53      // Pull the names
54      $special = array();
55      foreach (\Config::get('todo.listOrder') as $name)
56      {
57        $special[$name] = false;
58      }
59
60      // Peel off the specials
61      $tosort = array();
62      foreach ($listIds as $listId)
63      {
64        if (array_key_exists($listId, $special))
65        {
```

```
66            $special[$listId] = true;
67          }
68          else
69          {
70            $tosort[] = $listId;
71          }
72        }
73
74        // Put the specials first then sort the remaining and add them in
75        $return = array();
76        foreach ($special as $listId => $flag)
77        {
78          if ($flag)
79          {
80            $return[] = $listId;
81          }
82        }
83        natcasesort($tosort);
84        return array_merge($return, $tosort);
85      }
86
87      /**
88       * No arguments.
89       */
90      protected function getArguments()
91      {
92        return array();
93      }
94
95      /**
96       * Just the `--archived` option
97       */
98      protected function getOptions()
99      {
100       return array(
101         array('archived', 'a', InputOption::VALUE_NONE,
102           'Use archived lists?'),
103       );
104     }
105   }
106   ?>
```

ListTasksCommand.php

```
<?php namespace GSD\Commands;

use Config;
use Symfony\Component\Console\Input\InputOption;
use Todo;

class ListTasksCommand extends CommandBase {

  protected $name = 'gsd:list';
  protected $description = 'List tasks.';
  protected $nameArgumentDescription = 'List name to display tasks.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $name = $this->getListId('Select list to show tasks:');
    if (is_null($name))
    {
      $this->abort();
    }
    $list = Todo::get($name);

    $nextOnly = $this->option('action');
    $skipDone = $this->option('skip-done');
    if ($nextOnly and $skipDone)
    {
      $this->abort(
        "Options --action and --skip-done can't be used together."
      );
    }

    // Gather rows to display
    $completeFmt = Config::get('todo.dateCompleteFormat');
    $dueFmt = Config::get('todo.dateDueFormat');
    $rows = array();
    $rowNo = 1;
```

```
41      foreach ($list->tasks() as $task)
42      {
43        if ($task->isComplete())
44        {
45          if ($skipDone or $nextOnly) continue;
46          $rows[] = array(
47            '',
48            'done',
49            $task->description(),
50            'Done '.$task->dateCompleted()->format($completeFmt),
51          );
52        }
53        elseif ($task->isNextAction() or ! $nextOnly)
54        {
55          $next = ($task->isNextAction()) ? 'YES' : '';
56          $due = ($task->dateDue()) ?
57              'Due '.$task->dateDue()->format($dueFmt) : '';
58          $rows[] = array(
59            $rowNo++,
60            $next,
61            $task->description(),
62            $due,
63          );
64        }
65      }
66
67      // Output a pretty table
68      $title = ($nextOnly) ? "Next Actions" :
69          (($skipDone) ? "Active Tasks" : "All Tasks");
70      $this->info("$title in list '+$name'");
71      if (count($rows) == 0)
72      {
73        $this->abort("No tasks in list");
74      }
75      $table = $this->getHelperSet()->get('table');
76      $table
77        ->setHeaders(array('#', 'Next', 'Description', 'Extra'))
78        ->setRows($rows)
79        ->render($this->getOutput());
80    }
81
82    /**
```

```
83     * Get the console command options.
84     */
85    protected function getOptions()
86    {
87      return array_merge(array(
88        array('action', 'a', InputOption::VALUE_NONE,
89          'Show only next actions.', null),
90        array('skip-done', 'x', InputOption::VALUE_NONE,
91          'Skip completed actions.', null),
92      ), parent::getOptions());
93    }
94  }
95  ?>
```

RenameListCommand.php

```
1   <?php namespace GSD\Commands;
2
3   use Config;
4   use Symfony\Component\Console\Input\InputOption;
5   use Todo;
6
7   class RenameListCommand extends CommandBase {
8
9     protected $name = 'gsd:rename';
10    protected $description = 'Rename a list.';
11
12    /**
13     * Execute the console command.
14     *
15     * @return void
16     */
17    public function fire()
18    {
19      // Get archived flag and list to rename
20      $archived = $this->option('archived');
21      $selectTitle = 'Select list to rename:';
22      $name = $this->askForListId(true, true, $archived, $selectTitle);
23      if (is_null($name))
24      {
25        $this->abort();
26      }
```

```
27      if ( ! $archived && Config::get('todo.defaultList') == $name)
28      {
29        $this->abort('Cannot rename default list');
30      }
31
32      // Prompt for new list
33      $newName = $this->askForListId(false, true, $archived);
34      if (is_null($name))
35      {
36        $this->abort();
37      }
38
39      // Load existing list, save with new name
40      $list = Todo::get($name, $archived);
41      $newList = clone $list;
42      $newList->set('id', $newName);
43      $newList->save();
44
45      // Delete existing list and we're done
46      $list->delete();
47      $listType = ($archived) ? 'Archived list' : 'List';
48      $this->info($listType . " '$name' renamed to '$newName'");
49    }
50
51    /**
52     * No arguments.
53     */
54    protected function getArguments()
55    {
56      return array();
57    }
58
59    /**
60     * Just the --archived option
61     */
62    protected function getOptions()
63    {
64      return array(
65        array('archived', 'a', InputOption::VALUE_NONE,
66          'Use archived lists?', null),
67      );
68    }
```

```
69 }
70 ?>
```

TodoRepository.php and TodoRepositoryInterface.php

This chapter has a boatload of source code. Since only the docblocks changed in these two files, I'm going to skip displaying the source here.

UnarchiveListCommand.php

```
1  <?php namespace GSD\Commands;
2
3  use App;
4  use Todo;
5
6  class UnarchiveListCommand extends CommandBase {
7
8    protected $name = 'gsd:unarchive';
9    protected $description = 'Unarchive a todo list.';
10
11   /**
12    * Execute the console command.
13    *
14    * @return void
15    */
16   public function fire()
17   {
18     // Prompt user for list name
19     $selectTitle = 'Select list to unarchive:';
20     $name = $this->askForListId(true, true, true, $selectTitle);
21     if (is_null($name))
22     {
23       $this->abort();
24     }
25
26     // Warn if unarchived version exists
27     if ($this->repository->exists($name, false))
28     {
29       $msg = "WARNING!\n\n"
30            . "  An active version of the list '$name' exists.\n"
31            . "  This action will destroy the active version,\n"
32            . "  replacing it with the archived version.";
```

```
33        $this->outputErrorBox($msg);
34      }
35
36      // Ask if user is sure?
37      $result = $this->ask(
38        "Are you sure you want to unarchive '$name' (yes/no)?");
39      if ( ! str2bool($result))
40      {
41        $this->abort();
42      }
43
44      // Load existing list and save as unarchived
45      $list = Todo::get($name, true);
46      $list->set('archived', false);
47      $list->save();
48
49      // Delete existing archived list
50      if ( ! $this->repository->delete($name, true))
51      {
52        $this->abort('ERROR deleting archived version.');
53      }
54      $this->info("List '$name' has been unarchived");
55    }
56
57    /**
58     * No arguments.
59     */
60    protected function getArguments()
61    {
62      return array();
63    }
64
65    /**
66     * No options.
67     */
68    protected function getOptions()
69    {
70      return array();
71    }
72  }
73  ?>
```

UncreateCommand.php

```
<?php namespace GSD\Commands;

use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;
use Todo;

class UncreateCommand extends CommandBase {

  protected $name = 'gsd:uncreate';
  protected $description = 'Destroy an empty list.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Prompt user for list-id
    $selectTitle = 'Select list to uncreate:';
    $name = $this->askForListId(true, true, false, $selectTitle);
    if (is_null($name))
    {
      $this->abort();
    }

    // Validate list has no tasks
    $list = Todo::get($name);
    if ($list->taskCount() > 0)
    {
      $this->abort('Cannot uncreate a list with tasks');
    }

    // Delete list
    if ( ! $this->repository->delete($name))
    {
      $this->abort("Repository couldn't delete list '$name'");
    }
    $this->info("The list '$name' is now in the big bitbucket in the sky");
  }
```

```
41
42    /**
43     * No arguments.
44     */
45    protected function getArguments()
46    {
47      return array();
48    }
49
50    /**
51     * No options.
52     */
53    protected function getOptions()
54    {
55      return array();
56    }
57  }
58  ?>
```

Man. My head hurts after all the code.

Dogfooding

Let's mark the tasks completed that we've finished.

```
1  ~/gsd$ art gsd:list +gsd -x
2  Active Tasks in list '+gsd'
3  +----+------+-------------------------------------+-------+
4  | #  | Next | Description                         | Extra |
5  +----+------+-------------------------------------+-------+
6  | 1  | YES  | Add CommandBase::abort()            |       |
7  | 2  | YES  | Add to askForListId()               |       |
8  | 3  | YES  | Check gsd help consistency          |       |
9  | 4  | YES  | Create MoveTaskCommand              |       |
10 | 5  | YES  | Create RemoveTaskCommand            |       |
11 | 6  | YES  | List tasks in ListAllCommand        |       |
12 | 7  | YES  | Use ListInterface::delete()         |       |
13 | 8  |      | Chapter on setting up webserver     |       |
14 | 9  |      | Create Appendix for Apache install  |       |
15 | 10 |      | Create Appendix for nginx install   |       |
16 | 11 |      | Create shell gsd script             |       |
```

```
| 12 |      | Create web application wireframe   |       |
+----+------+------------------------------------+-------+
~/gsd$ art gsd:do 3 +gsd
Task 'Check gsd help consistency' marked complete.
~/gsd$ art gsd:do 2 +gsd
Task 'Add to askForListId()' marked complete.
~/gsd$ art gsd:do 1 +gsd
Task 'Add CommandBase::abort()' marked complete.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'
+---+------+----------------------------------------+--------------+
| # | Next | Description                            | Extra        |
+---+------+----------------------------------------+--------------+
| 1 | YES  | Create MoveTaskCommand                 |              |
| 2 | YES  | Create RemoveTaskCommand               |              |
| 3 | YES  | List tasks in ListAllCommand           |              |
| 4 | YES  | Use ListInterface::delete()            |              |
| 5 |      | Chapter on setting up webserver        |              |
| 6 |      | Create Appendix for Apache install     |              |
| 7 |      | Create Appendix for nginx install      |              |
| 8 |      | Create shell gsd script                |              |
| 9 |      | Create web application wireframe       |              |
|   | done | Create EditTaskCommand                 | Done 9/29/13 |
|   | done | Create ArchiveListCommand              | Done 10/4/13 |
|   | done | Create UnarchiveListCommand            | Done 10/4/13 |
|   | done | Add CommandBase::abort()               | Done 10/5/13 |
|   | done | Add to askForListId()                  | Done 10/5/13 |
|   | done | Check gsd help consistency             | Done 10/5/13 |
|   | done | Create RenameListCommand               | Done 10/5/13 |
|   | done | Remove blank line after gsd:list title | Done 10/5/13 |
+---+------+----------------------------------------+--------------+
```

To bad we can't delete task #4. Let's hurry and implement the RemoveTask command in the next chapter before I change my mind back and decide to implement #4.

Chapter 51 - The RemoveTaskCommand

In This Chapter

In this chapter we'll implement the remove task command.

After all that source code in the last chapter, I want to keep this chapter short and sweet.

The Plan

The original thoughts on the console command were.

```
$ gsd:remove [+name] 1 --force
$ gsd rm [+name] 1 -f
$ gsd rm
```

Since we now have the rule that the list name comes last, then we're modifying this somewhat, but the basic idea's the same.

How about the pseudo-code?

```
Get arguments and options
Prompt for list-id if needed
Load list
Prompt for task # if needed
Show warning if not forced
Delete task from list
Save list
```

Pretty good. We should match that closely. The *"Show warning if not forced"* ought to be followed with a *"Ask user if they're sure"* but besides that the logic will match.

Creating the RemoveTaskCommand

As always, use artisan to create the skeleton.

```
~/gsd$ art command:make RemoveTaskCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Update the newly created `RemoveTaskCommand.php` to match what's below.

```php
<?php namespace GSD\Commands;

use Symfony\Component\Console\Input\InputOption;
use Todo;

class RemoveTaskCommand extends CommandBase {

  protected $name = 'gsd:remove';
  protected $description = 'Remove a task from a list.';
  protected $taskNoDescription = 'Task # to remove.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Load list, prompting if needed
    $name = $this->getListId('Select list with task to remove:');
    if (is_null($name))
    {
      $this->abort();
    }
    $list = Todo::get($name);

    // Prompt for task # if needed
    $taskNo = $this->getTaskNo($list, true, true, false);
    if (is_null($taskNo))
    {
      $this->abort();
    }

```

```
34      // Show warning, get prompt if needed
35      $description = $list->taskGet($taskNo, 'description');
36      if ( ! $this->option('force'))
37      {
38        $this->outputErrorBox(
39          "WARNING! This will remove the task '$description'."
40        );
41        $result = $this->ask("Are you sure (yes/no)?");
42        if ( ! str2bool($result))
43        {
44          $this->abort();
45        }
46      }
47
48      // Delete task from list and save
49      $list->taskRemove($taskNo)
50           ->save();
51      $this->info("Task '$description' removed from '+$name'");
52    }
53
54    /**
55     * Return the options.
56     */
57    protected function getOptions()
58    {
59      return array_merge(parent::getOptions(), array(
60        array('force', 'f', InputOption::VALUE_NONE,
61          'Force the removal, no prompting.'),
62      ));
63    }
64  }
65  ?>
```

No big explanations needed here. The logic in the `fire()` method follows the pseudo-code almost exactly.

Make artisan aware of the new command, editing `start/artisan.php`.

```
<?php
Artisan::add(new GSD\Commands\AddTaskCommand);
Artisan::add(new GSD\Commands\ArchiveListCommand);
Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\DoTaskCommand);
Artisan::add(new GSD\Commands\EditListCommand);
Artisan::add(new GSD\Commands\EditTaskCommand);
Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\ListTasksCommand);
Artisan::add(new GSD\Commands\RemoveTaskCommand);
Artisan::add(new GSD\Commands\RenameListCommand);
Artisan::add(new GSD\Commands\UnarchiveListCommand);
Artisan::add(new GSD\Commands\UncreateCommand);
?>
```

All done coding for this chapter.

Dogfooding

Now we can mark the the *Create RemoveTaskCommand* complete and remove the *Use ListInterface::delete()* task from our gsd todo list.

Wait. Maybe we should use `ListInterface::delete()` in the commands that could use it. I mean, it'd be simple to do, right?

Hah! Scared you didn't I? Nope, let's remove that task from the todo list and hopefully that will remove the temptation to implement it.

Below is how I did it. Your list numbers may be different if you've made any modifications to the *gsd* todo list.

```
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'
+---+------+-----------------------------------+-------+
| # | Next | Description                       | Extra |
+---+------+-----------------------------------+-------+
| 1 | YES  | Create MoveTaskCommand            |       |
| 2 | YES  | Create RemoveTaskCommand          |       |
| 3 | YES  | List tasks in ListAllCommand      |       |
| 4 | YES  | Use ListInterface::delete()       |       |
| 5 |      | Chapter on setting up webserver   |       |
```

```
| 6 |      | Create Appendix for Apache install |       |
| 7 |      | Create Appendix for nginx install  |       |
| 8 |      | Create shell gsd script            |       |
| 9 |      | Create web application wireframe   |       |
+---+------+------------------------------------+-------+
~/gsd$ art gsd:remove 4 +gsd


  WARNING! This will remove the task 'Use ListInterface::delete()'.


Are you sure (yes/no)?yes
Task 'Use ListInterface::delete()' removed from '+gsd'
~/gsd$ art gsd:do 2 +gsd
Task 'Create RemoveTaskCommand' marked complete.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'
+---+------+----------------------------------------+--------------+
| # | Next | Description                            | Extra        |
+---+------+----------------------------------------+--------------+
| 1 | YES  | Create MoveTaskCommand                 |              |
| 2 | YES  | List tasks in ListAllCommand           |              |
| 3 |      | Chapter on setting up webserver        |              |
| 4 |      | Create Appendix for Apache install     |              |
| 5 |      | Create Appendix for nginx install      |              |
| 6 |      | Create shell gsd script                |              |
| 7 |      | Create web application wireframe       |              |
|   | done | Create EditTaskCommand                 | Done 9/29/13 |
|   | done | Create ArchiveListCommand              | Done 10/4/13 |
|   | done | Create UnarchiveListCommand            | Done 10/4/13 |
|   | done | Add CommandBase::abort()               | Done 10/5/13 |
|   | done | Add to askForListId()                  | Done 10/5/13 |
|   | done | Check gsd help consistency             | Done 10/5/13 |
|   | done | Create RemoveTaskCommand               | Done 10/5/13 |
|   | done | Create RenameListCommand               | Done 10/5/13 |
|   | done | Remove blank line after gsd:list title | Done 10/5/13 |
+---+------+----------------------------------------+--------------+
```

No need to explain this line-by-line. You should be getting pretty familiar with these commands by now.

Chapter 52 - The MoveTaskCommand

In This Chapter

In this chapter we'll implement moving tasks between lists

The Plan

The original plan for the command was:

```
$ gsd move [+name] 1 +dest
$ gsd mv [+name] 1 +dest
$ gsd mv
```

Of course, now the listname must come last and the task numbers comes before it. Also, I think the destination should be implemented as an option instead of an argument. And, we'll add the force option This makes the plan:

```
$ gsd move [task-number] [+name] --dest=destname --force
$ gsd move
```

The second option would prompt for everything.

The psuedo-code was:

```
Get arguments and options
Prompt for list-id if needed
Load source list
Prompt for task # if needed
Prompt for dest list-id if needed
Ask if they're sure if not forced
Load task # from source list
Save task in dest list
Save dest list
Delete task # from source list
Save source list
```

The implementation should be very close to this.

Creating the MoveTaskCommand

Create the skeleton

```
~/gsd$ art command:make MoveTaskCommand --path=app/src/GSD/Commands \
> --namespace="GSD\Commands"
Command created successfully.
```

Update the newly created `MoveTaskCommand.php` to match what's below.

```php
<?php namespace GSD\Commands;

use Symfony\Component\Console\Input\InputOption;
use Todo;

class MoveTaskCommand extends CommandBase {

  protected $name = 'gsd:move';
  protected $description = 'Move a task between lists.';
  protected $nameArgumentDescription = 'Source list name.';
  protected $taskNoDescription = 'Task # to move from source list.';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    // Get source list
    $sourceName = $this->getListId('Select list with task to move:');
    if (is_null($sourceName))
    {
      $this->abort();
    }
    $sourceList = Todo::get($sourceName);
```

```
27
28      // Get task to move
29      $taskNo = $this->getTaskNo($sourceList, true, true, false);
30      if (is_null($taskNo))
31      {
32        $this->abort();
33      }
34
35      // Get dest list
36      $destName = $this->option('dest');
37      if (is_null($destName))
38      {
39        $destName = $this->askForListId(
40          true,
41          true,
42          false,
43          'Select destination list:'
44        );
45        if (is_null($destName))
46        {
47          $this->abort();
48        }
49      }
50      if ($destName == $sourceName)
51      {
52        $this->abort('Source and destination cannot be the same');
53      }
54      $destList = Todo::get($destName);
55
56      // Verify
57      $task = $sourceList->task($taskNo);
58      $description = $task->description();
59      $fromTo = sprintf("from '+%s' to '+%s'", $sourceName, $destName);
60      if ( ! $this->option('force'))
61      {
62        $this->outputErrorBox("Moving '$description' $fromTo");
63        $result = $this->ask("Are you sure (yes/no)?");
64        if ( ! str2bool($result))
65        {
66          $this->abort();
67        }
68      }
```

```
69
70      // Remove from source, add to dest, save both
71      $sourceList->taskRemove($taskNo);
72      $destList->taskAdd($task)
73               ->save();
74      $sourceList->save();
75      $this->info("'$description' moved $fromTo");
76    }
77
78    /**
79     * Get the console command options.
80     *
81     * @return array
82     */
83    protected function getOptions()
84    {
85      return array_merge(array(
86        array('dest', 'd', InputOption::VALUE_REQUIRED,
87          'Destination list name.', null),
88        array('force', 'f', InputOption::VALUE_NONE,
89          'Force the move, no prompting.'),
90      ), parent::getOptions());
91    }
92  }
93  ?>
```

Although the `fire()` method is slightly longer than others we've implemented, the logic is very straightforward. No need to explain each step.

And add it to `start/artisan.php`

```
<?php
Artisan::add(new GSD\Commands\AddTaskCommand);
Artisan::add(new GSD\Commands\ArchiveListCommand);
Artisan::add(new GSD\Commands\CreateCommand);
Artisan::add(new GSD\Commands\DoTaskCommand);
Artisan::add(new GSD\Commands\EditListCommand);
Artisan::add(new GSD\Commands\EditTaskCommand);
Artisan::add(new GSD\Commands\ListAllCommand);
Artisan::add(new GSD\Commands\ListTasksCommand);
Artisan::add(new GSD\Commands\MoveTaskCommand);
Artisan::add(new GSD\Commands\RemoveTaskCommand);
Artisan::add(new GSD\Commands\RenameListCommand);
Artisan::add(new GSD\Commands\UnarchiveListCommand);
Artisan::add(new GSD\Commands\UncreateCommand);
?>
```

It should work. Practice moving tasks between lists. Everything is working great! *(Knock on wood).*

Knocking on wood

Although this phrase has different origins in different cultures, with programming I believe the phrase came about from the days when wooden gears were used in computers. These gears were prone to termites. *Knocking on the wooden gears* helped shake loose any bugs in the gears. Thus a programmer's project was more likely to run bug free.

Dogfooding

Let's mark the task complete that we've finished.

Below is how I did it. Yes, your list may look slightly different if it has been modified at all.

```
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'
+---+------+-----------------------------------+-------+
| # | Next | Description                       | Extra |
+---+------+-----------------------------------+-------+
| 1 | YES  | Create MoveTaskCommand            |       |
| 2 | YES  | List tasks in ListAllCommand      |       |
| 3 |      | Chapter on setting up webserver   |       |
| 4 |      | Create Appendix for Apache install |       |
| 5 |      | Create Appendix for nginx install |       |
| 6 |      | Create shell gsd script           |       |
| 7 |      | Create web application wireframe  |       |
+---+------+-----------------------------------+-------+
~/gsd$ art gsd:do 1 +gsd
Task 'Create MoveTaskCommand' marked complete.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------------+--------------+
| # | Next | Description                           | Extra        |
+---+------+---------------------------------------+--------------+
| 1 | YES  | List tasks in ListAllCommand          |              |
| 2 |      | Chapter on setting up webserver       |              |
| 3 |      | Create Appendix for Apache install    |              |
| 4 |      | Create Appendix for nginx install     |              |
| 5 |      | Create shell gsd script               |              |
| 6 |      | Create web application wireframe      |              |
|   | done | Create EditTaskCommand                | Done 9/29/13 |
|   | done | Create ArchiveListCommand             | Done 10/4/13 |
|   | done | Create UnarchiveListCommand           | Done 10/4/13 |
|   | done | Add CommandBase::abort()              | Done 10/5/13 |
|   | done | Add to askForListId()                 | Done 10/5/13 |
|   | done | Check gsd help consistency            | Done 10/5/13 |
|   | done | Create MoveTaskCommand                | Done 10/5/13 |
|   | done | Create RemoveTaskCommand              | Done 10/5/13 |
|   | done | Create RenameListCommand              | Done 10/5/13 |
|   | done | Remove blank line after gsd:list title | Done 10/5/13 |
+---+------+---------------------------------------+--------------+
~/gsd$ art gsd:edit 5 +gsd -a on
Task in gsd updated to: * Create shell gsd script
~/gsd$ art gsd:list +gsd -a
Next Actions in list '+gsd'
+---+------+----------------------------+-------+
```

```
| # | Next | Description                  | Extra |
+---+------+------------------------------+-------+
| 1 | YES  | Create shell gsd script      |       |
| 2 | YES  | List tasks in ListAllCommand |       |
+---+------+------------------------------+-------+
```

Just a couple more chapters left in this part of the book.

Chapter 53 - Listing Tasks Across Lists

In This Chapter

In this chapter we'll expand the ListAllCommand to optionally list tasks.

The Plan

Back in Chapter 38 we created the `gsd:listall` command without implementing the option to show tasks. Remember the original command plan from Chapter 35? It was:

```
$ gsd listall --tasks=all|done|next|normal --archived
$ gsd lsa -t type -a
$ gsd lsna
```

The last one is supposed to be a short cut to list all next actions which we'll implement when creating the shell gsd script so we won't worry about that one yet. The second to last one is a shortcut version with the actual command aliased to something shorter so let's not worry about that version yet either.

So, basically, we just need to modify the existing command to watch for a new `--tasks` option and, if used, output the tasks too.

Update ListAllCommand

Modify `ListAllCommand.php` to match what's below

```php
<?php namespace GSD\Commands;

use Config;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;

class ListAllCommand extends CommandBase {

  protected $name = 'gsd:listall';
  protected $description = 'Lists all todo lists (and possibly tasks).';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    $archived = $this->option('archived');
    $tasks = $this->option('tasks');
    if ( ! is_null($tasks))
    {
      $validTasks = array('all', 'next', 'normal', 'done');
      if ( ! in_array($tasks, $validTasks))
      {
        $msg = sprintf(
          "Invalid --tasks=%s. Must be one of '%s'.",
          $tasks,
          join("', '", $validTasks)
        );
        $this->abort($msg);
      }
      if ($tasks == 'next') $tasks = 'next action';
      $completeFmt = Config::get('todo.dateCompleteFormat');
      $dueFmt = Config::get('todo.dateDueFormat');
    }

    // Get lists
    $lists = \Todo::allLists($archived);
    $lists = $this->sortListIds($lists);

    // Ouput title
```

```
43      $listType = ($archived) ? 'archived lists' : 'lists';
44      $listWhat = is_null($tasks) ? 'all' : "$tasks tasks in all";
45      $this->info("Listing $listWhat $listType");
46
47      // Different headers based on tasks usage
48      if (is_null($tasks))
49      {
50        $headers = array('List', 'Next', 'Normal', 'Completed');
51      }
52      else
53      {
54        $headers = array('List', 'Next', 'Description', 'Extra');
55      }
56      $rows = array();
57      foreach ($lists as $listId)
58      {
59        $list = \Todo::get($listId, $archived);
60
61        // We're just outputing the lists
62        if (is_null($tasks))
63        {
64          $rows[] = array(
65            $listId,
66            $list->taskCount('next'),
67            $list->taskCount('todo'),
68            $list->taskCount('done'),
69            );
70          }
71
72          else
73          {
74            // Loop through tasks to figure which to output
75            foreach ($list->tasks() as $task)
76            {
77              if ($task->isComplete())
78              {
79                if ($tasks == 'done' || $tasks == 'all')
80                {
81                  $done = $task->dateCompleted()
82                               ->format($completeFmt);
83                  $rows[] = array(
84                    $listId,
```

```
                    '',
                    $task->description(),
                    "Done $done"
                  );
                }
              }

              // Other, unfinished tasks
              else
              {
                $next = ($task->isNextAction()) ? 'YES' : '';
                $due = ($task->dateDue()) ?
                  'Due '.$task->dateDue()->format($dueFmt) : '';
                if (($tasks == 'all') or
                    ($tasks == 'next action' && $next == 'YES') or
                    ($tasks == 'normal' && $next == ''))
                {
                  $rows[] = array(
                    $listId,
                    $next,
                    $task->description(),
                    $due
                  );
                }
              }
            }
          }

          // Output a pretty table
          $table = $this->getHelperSet()->get('table');
          $table
            ->setHeaders($headers)
            ->setRows($rows)
            ->render($this->getOutput());
        }

        /**
         * Sort the list ids
         */
        protected function sortListIds(array $listIds)
        {
```

```
127         // Pull the names
128         $special = array();
129         foreach (\Config::get('todo.listOrder') as $name)
130         {
131           $special[$name] = false;
132         }
133 
134         // Peel off the specials
135         $tosort = array();
136         foreach ($listIds as $listId)
137         {
138           if (array_key_exists($listId, $special))
139           {
140             $special[$listId] = true;
141           }
142           else
143           {
144             $tosort[] = $listId;
145           }
146         }
147 
148         // Put the specials first then sort the remaining and add them in
149         $return = array();
150         foreach ($special as $listId => $flag)
151         {
152           if ($flag)
153           {
154             $return[] = $listId;
155           }
156         }
157         natcasesort($tosort);
158         return array_merge($return, $tosort);
159       }
160 
161       /**
162        * No arguments.
163        */
164       protected function getArguments()
165       {
166         return array();
167       }
168 
```

```
169     /**
170      * Just the `--archived` option
171      */
172     protected function getOptions()
173     {
174       return array(
175         array('archived', 'a', InputOption::VALUE_NONE,
176           'Use archived lists?'),
177         array('tasks', 't', InputOption::VALUE_REQUIRED,
178           'Output (all|next|normal|done) tasks?'),
179       );
180     }
181 }
182 ?>
```

Below I'm only discussing the items in the above code that have changed.

Lines 20 - 36
: We pull in the new –tasks option, validate it and since it's used we also load the date formatting from the configuration.

Lines 42 - 45
: The output of the title will also be dependent on the –tasks option.

Lines 47 - 55
: The rows output will be different depending on the –tasks option, so the headers will be different too.

Lines 61 - 70
: This is the same as the previous version when outputting just the lists, but now it's wrapped in an if block.

Lines 75 - 110
: The tasks are looped through. We only add to the $rows[] array if the task matches what's specified in the –tasks option.

Lines 177 - 178
: The new –tasks option is also returned.

Give it a shot. Try running `art gsd:listall` in a variety of ways. Now you can list all the next actions across all projects.

```
~/gsd$ art gsd:listall -t next
Listing next action tasks in all lists
+---------+------+------------------------------+-------+
| List    | Next | Description                  | Extra |
+---------+------+------------------------------+-------+
| actions | YES  | Test2 add next action        |       |
| gsd     | YES  | Create shell gsd script      |       |
| gsd     | YES  | List tasks in ListAllCommand |       |
+---------+------+------------------------------+-------+
```

Dogfooding

Let's mark the task completed that we've finished.

```
~/gsd$ art gsd:list +gsd -x
Active Tasks in list '+gsd'
+---+------+-----------------------------------+-------+
| # | Next | Description                       | Extra |
+---+------+-----------------------------------+-------+
| 1 | YES  | Create shell gsd script           |       |
| 2 | YES  | List tasks in ListAllCommand      |       |
| 3 |      | Chapter on setting up webserver   |       |
| 4 |      | Create Appendix for Apache install |       |
| 5 |      | Create Appendix for nginx install  |       |
| 6 |      | Create web application wireframe  |       |
+---+------+-----------------------------------+-------+
~/gsd$ art gsd:do 2 +gsd
Task 'List tasks in ListAllCommand' marked complete.
~/gsd$ art gsd:list +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------------+--------------+
| # | Next | Description                           | Extra        |
+---+------+---------------------------------------+--------------+
| 1 | YES  | Create shell gsd script               |              |
| 2 |      | Chapter on setting up webserver       |              |
| 3 |      | Create Appendix for Apache install    |              |
| 4 |      | Create Appendix for nginx install     |              |
| 5 |      | Create web application wireframe      |              |
|   | done | Create EditTaskCommand                | Done 9/29/13 |
|   | done | Create ArchiveListCommand             | Done 10/4/13 |
|   | done | Create UnarchiveListCommand           | Done 10/4/13 |
```

```
|   | done | Add CommandBase::abort()                | Done 10/5/13 |
|   | done | Add to askForListId()                   | Done 10/5/13 |
|   | done | Check gsd help consistency              | Done 10/5/13 |
|   | done | Create MoveTaskCommand                  | Done 10/5/13 |
|   | done | Create RemoveTaskCommand                | Done 10/5/13 |
|   | done | Create RenameListCommand                | Done 10/5/13 |
|   | done | Remove blank line after gsd:list title | Done 10/5/13 |
|   | done | List tasks in ListAllCommand            | Done 10/6/13 |
+---+------+-----------------------------------------+--------------+
```

Only one thing left in this part of the book. But really, our console commands are finished at this point. We just want to make them easier to run.

Chapter 54 - Command Aliases and the gsd shell script

In This Chapter

In this chapter we'll add in command aliases and build a shell script to make using the console command even easier.

Command Aliases

Our commands are built upon Laravel's `Command` class. Actually, it's `Illuminate\Console\Command`. There's no direct support for command aliasing in that command, but if we dig deeper we discover Laravel's `Command` class is built on Symfony's Console Command class.

Guess what? Symfony provides command aliasing.

Here's the docblock from the Symfony's `setAliases()` method.

```
<?php
  /**
   * Sets the aliases for the command.
   *
   * @param array $aliases An array of aliases for the command
   *
   * @return Command The current instance
   *
   * @api
   */
  public function setAliases($aliases)
?>
```

This sounds like what we need, let's test it.

Add a constructor the the `ListTasksCommand.php` file as follows.

```php
<?php
  /**
   * Constructor
   */
  public function __construct()
  {
    parent::__construct();
    $this->setAliases(array('ls'));
  }
?>
```

Now check to see what artisan's list of commands looks like.

```
~/gsd$ art
Laravel Framework version 4.0.7

Usage:
  [options] command [arguments]

Options:
  --help           -h Display this help message.
  --quiet          -q Do not output any message.
  --verbose        -v|vv|vvv Increase the verbosity of messages: 1 for
    normal output, 2 for more verbose output and 3 for debug
  --version        -V Display this application version.
  --ansi              Force ANSI output.
  --no-ansi           Disable ANSI output.
  --no-interaction -n Do not ask any interactive question.
  --env               The environment the command should run under.

Available commands:
  changes           Display the framework change list
  clear-compiled    Remove the compiled class file
  dump-autoload     Regenerate framework autoload files
  env               Display the current framework environment
  help              Displays help for a command
  list              Lists commands
  ls                List tasks.
  optimize          Optimize the framework for better performance
  routes            List all registered routes
```

```
command
  command:make      Create a new Artisan command
controller
  controller:make   Create a new resourceful controller
gsd
  gsd:addtask       Add a new task to a list.
  gsd:archive       Archive a todo list.
  gsd:create        Create new list.
  gsd:do            Mark a task as complete.
  gsd:edit          Edit a task.
  gsd:editlist      Edit a list's title or subtitle.
  gsd:list          List tasks.
  gsd:listall       Lists all todo lists (and possibly tasks).
  gsd:move          Move a task between lists.
  gsd:remove        Remove a task from a list.
  gsd:rename        Rename a list.
  gsd:unarchive     Unarchive a todo list.
  gsd:uncreate      Destroy an empty list.
```

I didn't see it at first, but it's up there under *Available commands:*. Nice, I guess we need to alias it as `gsd:ls` instead of just plain `ls`.

Play around with it. It seems to work slick. Now we know how to alias our commands. Excellent!

Remove the constructor added to `ListTasksCommand.php` file.

Planning the aliases and macros

I went through all the console commands, and the list of commands we originally came up with in Chapter 35 and generated the following tables. The rows of the *Command Aliases* table show commands we'll' use the `setAliases()` method to accomplish. The *Macro* table shows commands we want to translate into other commands and options.

Command Aliases

Alias	Command
gsd:lsa	gsd:listall
gsd:add	gsd:addtask
gsd:a	gsd:addtask
gsd:ls	gsd:list
gsd:ed	gsd:edit
gsd:rm	gsd:remove
gsd:mv	gsd:move

Command Macros

Macro	Expands To
gsd:lsna	gsd:listall –tasks=next

Even though there's only one macro at this point, I want to remain aware that new macros could be added in the future.

Implementing the aliases

One way to do this would be to edit each of the source files for the commands that take an alias and add the constructor, having it set the alias as we did in the test earlier. But I'm lazy. How could I implement the aliases with the least amount of work?

Heh, easy. I'll put them in our configuration and let `CommandBase` deal with them. After all, every command uses `CommandBase` as their parent class.

Update your `config/todo.php` adding the section below `// Command aliases`.

```
<?php
// Config for our awesome getting stuff done application
return array(
    'folder' => '/home/chuck/Documents/gsd/',
    'extension' => '.txt',
    'listOrder' => array(
        'inbox', 'actions', 'waiting', 'someday', 'calendar',
    ),
    'defaultList' => 'actions',         // default list when not specified
    'noListPrompt' => true,             // true=prompt for list, false=use default
    'dateCompleteFormat' => 'n/j/y',    // date format for completed tasks
    'dateDueFormat' => 'M-j',           // date format for due tasks

    // Command aliases
    'aliases' => array(
        'gsd:addtask' => array('gsd:a','gsd:add'),
        'gsd:edit'    => array('gsd:ed'),
        'gsd:list'    => array('gsd:ls'),
        'gsd:listall' => array('gsd:lsa'),
        'gsd:move'    => array('gsd:mv'),
        'gsd:remove'  => array('gsd:rm'),
    ),
);
?>
```

Edit `CommandBase.php`'s constructor to match what's below.

```
<?php
  /**
   * Constructor
   */
  public function __construct()
  {
    parent::__construct();
    $this->repository = App::make('GSD\Repositories\TodoRepositoryInterface');
    $aliases = Config::get('todo.aliases');
    if (array_key_exists($this->name, $aliases))
    {
      $this->setAliases($aliases[$this->name]);
    }
```

```
14    }
15  ?>
```

Simple, beautiful, just a tiny bit code to implement all those aliases.

Try it out.

Gah. It didn't work as I thought it would. The very first command I tried created a big red error.

```
1  ~/gsd$ art gsd:a
2
3
4    [InvalidArgumentException]
5    Command "gsd:a" is ambiguous (gsd:addtask, gsd:archive).
```

Just when I thought everything was great, ran into something else. Sheesh.

Okay, after thinking about it for a bit, testing a few things, I came up with the following compromises:

1. Skip the `gsd:a` and change the `gsd:addtask` to be `gsd:add` instead.
2. Don't worry about the `gsd:ed` alias, `gsd:edit` is short enough.
3. Change the `gsd:lsa` alias to `gsd:la`.

Edit the top of `AddTaskCommand.php` and change the `$name` property to what's below.

```
1      protected $name = 'gsd:add';
```

Change the `\\ Command aliases` section of `config/todo.php` to match what's below.

```
  // Command aliases
  'aliases' => array(
    'gsd:list'    => array('gsd:ls'),
    'gsd:listall' => array('gsd:la'),
    'gsd:move'    => array('gsd:mv'),
    'gsd:remove'  => array('gsd:rm'),
  ),
```

Okay, now everything works.

The Bash Script

Now, finally, I want to put a little bash script somewhere in my path that will shorten what I need to type even more.

Create a file named `gsd` and put it in your path. I'm placing my copy in `home/chuck/bin`, but your setup may differ.

NOTE This is not for Windows users. Sorry, but you could create a batch file to do the same.

```
#!/bin/bash
#
# Simple bash script to call artisan gsd:COMMAND
#
ARTISAN=/home/chuck/gsd/artisan
MACROS="lsna"
COMMANDS="add archive create do edit editlist la list listall ls move \
mv remove rename rm unarchive uncreate"

# Process Macros
for cmd in $MACROS
do
  if [ "$cmd" == "$1" ]; then
    # Only the one macro
    $ARTISAN gsd:listall --tasks=next
    exit
  fi
done

# Process Commands
```

```
21 for cmd in $COMMANDS
22 do
23   if [ "$cmd" == "$1" ]; then
24     $ARTISAN "gsd:$@"
25     exit
26   fi
27 done
28
29 # Output usage
30 echo "Usage: gsd command [options]"
31 echo
32 echo "Where command is"
33 echo "   add        Add a new task to a list."
34 echo "   archive    Archive a todo list."
35 echo "   create     Create new todo list."
36 echo "   do         Mark a task as complete."
37 echo "   edit       Edit a task."
38 echo "   editlist   Edit a list's title or subtitle."
39 echo "   list       List tasks."
40 echo "   listall|la List all todo lists (and possibly tasks)."
41 echo "   ls         List tasks."
42 echo "   lsna       List Next Actions in all todo lists."
43 echo "   move|mv    Move a task between todo lists."
44 echo "   remove|rm  Remove a task from a todo list."
45 echo "   rename     Rename a todo list."
46 echo "   unarchive  Unarchive a todo list."
47 echo "   uncreate   Destroy an empty list."
48 echo
49 echo "Use gsd command -h for help on a command";
```

Line 5

Make sure to use the path to where *you* have been building this project.

You may also have to make this file executable.

```
1 ~$ cd bin
2 ~/bin$ chmod +x gsd
```

Now you can use this short script to call the console commands. Best of all, you can call this script no matter which directory you are in.

Play around with it see how it works.

Here's me doing just that.

```
~$ gsd
Usage: gsd command [options]

Where command is
   add        Add a new task to a list.
   archive    Archive a todo list.
   create     Create new todo list.
   do         Mark a task as complete.
   edit       Edit a task.
   editlist   Edit a list's title or subtitle.
   list       List tasks.
   listall|la List all todo lists (and possibly tasks).
   ls         List tasks.
   lsna       List Next Actions in all todo lists.
   move|mv    Move a task between todo lists.
   remove|rm  Remove a task from a todo list.
   rename     Rename a todo list.
   unarchive  Unarchive a todo list.
   uncreate   Destroy an empty list.

Use gsd command -h for help on a command
~$ gsd lsna
Listing next action tasks in all lists
+---------+------+-------------------------+-------+
| List    | Next | Description             | Extra |
+---------+------+-------------------------+-------+
| actions | YES  | Test2 add next action   |       |
| gsd     | YES  | Create shell gsd script |       |
+---------+------+-------------------------+-------+
~$ gsd do -h
Usage:
 gsd:do [-l|--listname="..."] [task-number] [+name]

Arguments:
 task-number           Task # to mark complete.
 +name                 List name with completed task.

Options:
 --listname (-l)       Source of list name, 'prompt' or 'default'.
```

```
 --help (-h)           Display this help message.
 --quiet (-q)          Do not output any message.
 --verbose (-v|vv|vvv) Increase the verbosity of messages: 1 for normal
  output, 2 for more verbose output and 3 for debug
 --version (-V)        Display this application version.
 --ansi                Force ANSI output.
 --no-ansi             Disable ANSI output.
 --no-interaction (-n) Do not ask any interactive question.
 --env                 The environment the command should run under.
```

Dogfooding

From now on I'll just use our super, duper, nifty, little `gsd` script to manage my lists. (Well, that is, until the web application is done.)

Let's mark this last task complete.

```
~$ gsd ls +gsd -x
Active Tasks in list '+gsd'
+---+------+-----------------------------------+-------+
| # | Next | Description                       | Extra |
+---+------+-----------------------------------+-------+
| 1 | YES  | Create shell gsd script           |       |
| 2 |      | Chapter on setting up webserver   |       |
| 3 |      | Create Appendix for Apache install |       |
| 4 |      | Create Appendix for nginx install  |       |
| 5 |      | Create web application wireframe   |       |
+---+------+-----------------------------------+-------+
~$ gsd do 1 +gsd
Task 'Create shell gsd script' marked complete.
~$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------------+--------------+
| # | Next | Description                           | Extra        |
+---+------+---------------------------------------+--------------+
| 1 |      | Chapter on setting up webserver       |              |
| 2 |      | Create Appendix for Apache install    |              |
| 3 |      | Create Appendix for nginx install     |              |
| 4 |      | Create web application wireframe      |              |
|   | done | Create EditTaskCommand                | Done 9/29/13 |
|   | done | Create ArchiveListCommand             | Done 10/4/13 |
|   | done | Create UnarchiveListCommand           | Done 10/4/13 |
```

```
|   | done | Add CommandBase::abort()               | Done 10/5/13 |
|   | done | Add to askForListId()                  | Done 10/5/13 |
|   | done | Check gsd help consistency             | Done 10/5/13 |
|   | done | Create MoveTaskCommand                 | Done 10/5/13 |
|   | done | Create RemoveTaskCommand               | Done 10/5/13 |
|   | done | Create RenameListCommand               | Done 10/5/13 |
|   | done | Remove blank line after gsd:list title | Done 10/5/13 |
|   | done | Create shell gsd script                | Done 10/6/13 |
|   | done | List tasks in ListAllCommand           | Done 10/6/13 |
+---+------+----------------------------------------+--------------+
```

Yea! We're finished with the console application.

Chapter 55 - What's Next with the Console Application

The console application is complete as far as this book is concerned, but that doesn't mean it's complete for you.

How does the console application work for you? In what ways could it be improved to fit within your own workflow?

Here's a list of ideas on ways to improve the app. This list is by no means exhaustive.

- Track contexts like @home, @work, @backpacking-across-europe.
- Add a monthly archive to automatically move completed tasks elsewhere.
- Track priority tasks.
- Add activity logs, tracking whenever changes are made to the list.
- Expand due dates to include time.
- Add time tracking, gsd:startwork/stopwork on projects.
- Add backup/restore feature.
- Add additional task states, in-progress, waiting.
- Add subtasks
- Create a DbRepository to store tasks
- Add people tracking on tasks, maybe %chuck, %bob, %sally.
- Add encryption/decryption to todo lists.

See. All kinds of things you could do. I came up with a dozen features in just a couple minutes.

My point in all this is I really hope you make this project something you can use. And if you expand the feature in some interesting way, please shoot me an email. I'd love to hear from you.

Part 4 - The Web Application

In this last part of the book, we'll make the todo application work from the web. Or, more specifically, one that works from your browser and is hosted by your own machine. Sure, you could put this app on the web. But do you want to share your todo lists with everyone in the world?

Chapter 56 - Setting up the Web Server

In This Chapter

In this chapter we'll set up your web server to be able to serve pages for the Getting Stuff Done web application.

There are many ways you can set up a server on your machine to host web pages. I'll be using apache through this book, but wanted to present a few different alternatives.

Web server permissions

Okay, I'm going to advocate something that is generally frowned upon by the community. Since you're running a web server on your own machine, why not just let the web server act like you? By this I mean use your username and group for writing files?

As long as your firewall doesn't allow access to your machine on port 80, there's no issue.

I would never suggest this on a production machine. The `www-data` user (or whatever apache's using), should be managed separately from any user accounts on the machine.

So, in the two web server set ups below I suggest using your username and group. In my case it's `chuck` and `chuck`.

Using Apache

Apache is the number one web server on the net.

Follow the instructions in Appendix III to set up a host named '**gsd.localhost.com**', pointing it to your public folder of the laravel project you're creating in this book. (`/home/chuck/gsd/public` is what I'm using.) Be sure to set your permissions to use your username/group.

Using Nginx

Nginx is a very popular, high-performance web server. I actually like Nginx more than Apapche, but use Apache mostly so I can keep my work and home configurations as similar as possible.

Follow the instructions in Appendix IV to set up a host named '**gsd.localhost.com**', pointing it to your public folder, and using your username/group.

Using PHP's built in server

Another option is to use PHP's built in web server. I don't like doing this because it's not an optimized server and you must issue the command every time you want to start it (yes, I know you could set it up to automatically, but it's just not as clean in my opinion.).

The advantage? It's super easy to use. So, in a pinch, this is always a quick alternative.

First we have to re-enable the service provider that we commented out back in Chapter 34 when we locked down artisan. Edit `app/config/app.php` as specified below.

```
1   // Find the line below
2
3   //'Illuminate\Foundation\Providers\ServerServiceProvider',
4
5   // And remove the comment
6
7   'Illuminate\Foundation\Providers\ServerServiceProvider',
```

To start the server, just issue the `art serve` command.

```
1 ~/gsd$ art serve
2 Laravel development server started on http://localhost:8000
```

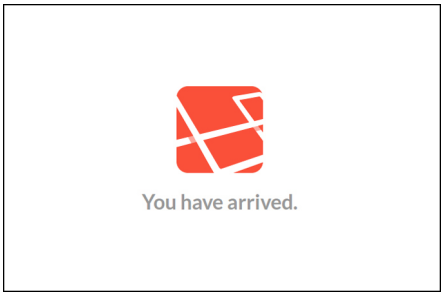
Notice two things:

1. The server is on port 8000
2. You have to leave the command running until you are done with it.

Exit the server with `Ctrl+C`.

You have arrived

Regardless of the web server method you use, bring up the web page `http://gsd.localhost.com` in your browser (or `http://localhost:8000` if you're using PHP's built in server). You should see the Laravel Logo in a page similar to the one below:



Your Browser's Screen

Chapter 57 - Planning the Web Application

In This Chapter

In this chapter we'll plan out what to do with the web application.

Initial Ideas

Here are, in no particular order, my initial thoughts for the web application.

A Single Page Application
: This is a pretty simple application, so let's just have everything on one web page. All the Todo Lists displayed on the left edge of the window, and all the tasks displayed in a larger pane to the right. Any input should happen as modal dialog boxes to the user.

Bootstrap
: I'll use Bootstrap[33] for the front end framework. I'm familiar with it and Bootstrap provides an easy-to-use grid for laying out the application. It also has methods for modal boxes built-in.

AJAX
: Since there's only a single page that all the actions will occur from, we'll need to make some AJAX calls to the load the lists and perform the actions.

Single User
: Since this list is for personal use, let's keep it single user. In other words, no worries about lists changing elsewhere while they're displayed within the browser.

Sounds pretty straight forward.

Planning the AJAX calls

Let's create a list of API endpoints for the web application to use.

[33] http://getbootstrap.com/

Method	Path	Description
POST	/lists	Create a new list
DELETE	/lists/{name}	Uncreate a list
GET	/lists	Returns a list of lists
GET	/lists?archive=1	Returns the archived lists
GET	/lists/{name}	Return the list
GET	/lists/{name}?archived=1	Return the archived list
PUT	/lists/{name}	Update the list named {name}
POST	/lists/{name}/archive	Archive the list
POST	/lists/{name}/unarchive	Unarchive the list
POST	/lists/{source}/rename/{dest}	Rename {source} to {dest}

What about adding tasks, marking tasks complete, etc? Well, in thinking about this single page application I figured it would be quicker to have a *"list"* be both the the list information (title, name, subtitle, archive status) and all the tasks associated with the list. Any time an action causes a call to the server, then the web page will redraw the data. Note, it's not going to redraw all the HTML, but simply refresh the data.

If we add a task, it updates the data internally, makes a call to 'PUT /lists/{name}', displays any success or failure message, and redraws the tasks.

This is not how a large, multi-user task list would be designed. In that case I'd definitely break things down to the task level and minimize the payload between client and server. A true multi-user task list, where multiple people could be editing the same list at the same time would be better served using web sockets.

Are we missing anything in the above table? Let's see:

- Adding tasks would add to current list, update the list on the server, and refresh.
- Marking a task complete would update task in current list, update the list on the server, and refresh.
- Editing task (the task description or next action) would follow the same pattern of updating the current list, updating the list on the server, and refreshing.
- Deleting a task, same pattern.
- Moving a task to a different list. Uggh. We could do an add task on the new list, then a delete task on the current list.

This will work.

Designing the Javascript Objects

When we're dealing with a list of lists, let's use the following structure:

```
var lists = [
  {
    name: "name",
    title: "Title",
    subtitle: null,
    numNextActions: 1,
    numNormal: 2,
    numCompleted: 5
  }, {
    ...
  }
];
```

And when we're dealing with lists, we'll use the following structure:

```
var list = {
  name: "name",
  title: "Title",
  subtitle: "Subtitle",
  tasks: [
    {
      isNext: true,
      isCompleted: false,
      descript: "Task description",
      dateDue: null, // Or javascript timestamp
      dateCompleted: null, // or javascript timestamp
    }, {
      ...
    }
  ]
};
```

And, to be consistent with our REST return values, let's always have the entity as part of a return object, but if there's an error, then the object has an error property.

```
var successReturn = {
  lists: [
    ...
  ]
};
var successReturn2 = {
  list: {
    ...
  }
};
var errorReturn = {
  error: "Error message"
};
```

Usually, I do all this planning in my head. My goal here was to get a general *"feel"* of how the web page is going to operate and for the underlying data structures. But I wanted to step back for a moment and analyze how closely I followed that old school steps I learned years ago.

- **Design the Output** - we did that in a very rough way at the top of this chapter in the *A Single Page Application* topic.
- **Design the Input** - that's going to be user input, the functions they can perform. This seems pretty straight forward. Yes, there are a couple of questions, like *"should the user be able to drag and drop tasks"*, but I'm not worried about the input.
- **Design the Database** - well, the database has already been defined the the repository, but really, to the web app, the database is going to be the REST API and the structures defined to hold the data.
- **Design the Functionality** - the functionality is pretty well defined. We'll be implementing everything the Console Application could do.

I would never normally think *"How close am I following a design methodology?"* The method, my method anyway, is to just do the next logical step. When there are multiple next steps then pick one, carry it forward, then back up, pick up another step and follow it forward. Rinse and repeat until the application is done.

Dogfooding

Since I skipped dogfooding in the last chapter, let's see what's currently in the list.

```
~/gsd$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------------+--------------+
| # | Next | Description                           | Extra        |
+---+------+---------------------------------------+--------------+
| 1 |      | Chapter on setting up webserver       |              |
| 2 |      | Create Appendix for Apache install    |              |
| 3 |      | Create Appendix for nginx install     |              |
| 4 |      | Create web application wireframe      |              |
|   | done | Create EditTaskCommand                | Done 9/29/13 |
|   | done | Create ArchiveListCommand             | Done 10/4/13 |
|   | done | Create UnarchiveListCommand           | Done 10/4/13 |
|   | done | Add CommandBase::abort()              | Done 10/5/13 |
|   | done | Add to askForListId()                 | Done 10/5/13 |
|   | done | Check gsd help consistency            | Done 10/5/13 |
|   | done | Create MoveTaskCommand                | Done 10/5/13 |
|   | done | Create RemoveTaskCommand              | Done 10/5/13 |
|   | done | Create RenameListCommand              | Done 10/5/13 |
|   | done | Remove blank line after gsd:list title | Done 10/5/13 |
|   | done | Create shell gsd script               | Done 10/6/13 |
|   | done | List tasks in ListAllCommand          | Done 10/6/13 |
+---+------+---------------------------------------+--------------+
```

Nice, Items 1-3 are done. Let's mark those done and add a few more tasks.

```
~/gsd$ gsd do 3 +gsd
Task 'Create Appendix for nginx install' marked complete.
~/gsd$ gsd do 2 +gsd
Task 'Create Appendix for Apache install' marked complete.
~/gsd$ gsd do 1 +gsd
Task 'Chapter on setting up webserver' marked complete.
~/gsd$ gsd add "Setup Routes" +gsd -a
Next Action successfully added to gsd
~/gsd$ gsd add "Implement GET /lists" +gsd -a
Next Action successfully added to gsd
~/gsd$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------------+---------------+
| # | Next | Description                           | Extra         |
+---+------+---------------------------------------+---------------+
| 1 | YES  | Implement GET /lists                  |               |
| 2 | YES  | Setup Routes                          |               |
```

```
| 3 |      | Create web application wireframe      |               |
|   | done | Create EditTaskCommand                | Done 9/29/13  |
|   | done | Create ArchiveListCommand             | Done 10/4/13  |
|   | done | Create UnarchiveListCommand           | Done 10/4/13  |
|   | done | Add CommandBase::abort()              | Done 10/5/13  |
|   | done | Add to askForListId()                 | Done 10/5/13  |
|   | done | Check gsd help consistency            | Done 10/5/13  |
|   | done | Create MoveTaskCommand                | Done 10/5/13  |
|   | done | Create RemoveTaskCommand              | Done 10/5/13  |
|   | done | Create RenameListCommand              | Done 10/5/13  |
|   | done | Remove blank line after gsd:list title | Done 10/5/13  |
|   | done | Create shell gsd script               | Done 10/6/13  |
|   | done | List tasks in ListAllCommand          | Done 10/6/13  |
|   | done | Chapter on setting up webserver       | Done 10/12/13 |
|   | done | Create Appendix for Apache install    | Done 10/12/13 |
|   | done | Create Appendix for nginx install     | Done 10/12/13 |
+---+------+---------------------------------------+---------------+
```

Everything above should be self explanatory.

Chapter 58 - Mocking Up the Web Page

In This Chapter

In this chapter we'll mock up the web page.

Setting Up Bootstrap

The first thing we need to do is download bootstrap and set it up in our project so we can use it. In a way, we're bootstrapping bootstrap. *(Ugh, yeah that was a groaner.)*

Downloading and unzipping

The instructions below are for Ubuntu/Mint Linux to download and unzip bootstrap into a temporary directory in our gsd project.

```
# Download it
~/gsd$ wget https://github.com/twbs/bootstrap/archive/v3.0.0.zip

... many lines scroll by

2013-10-13 09:22:38 (1.19 MB/s) - `v3.0.0.zip' saved [1523090/1523090]

# Unzip it
~/gsd$ unzip v3.0.0.zip

... many lines scroll by

 inflating: bootstrap-3.0.0/less/variables.less
 inflating: bootstrap-3.0.0/less/wells.less
 inflating: bootstrap-3.0.0/package.json

# Delete the zip file
~/gsd$ rm v3.0.0.zip
```

Moving what we need into our project

Follow the instructions to move what we need into our public directory.

```
# First the bootstrap core files
~/gsd$ mv bootstrap-3.0.0/dist/* public/

# Then move jquery
~/gsd$ mv bootstrap-3.0.0/assets/js/jquery.js public/js/

# Then wipe out the rest of bootstrap
~/gsd$ rm -r bootstrap-3.0.0
```

Build a basic template

Now, we'll edit the existing "hello" page that Laravel provides out of the box, and turn it into a basic bootstrap template.

First rename the view, magically turning into a Blade Template

```
~/gsd$ mv app/views/hello.php app/views/hello.blade.php
```

Heh. That was pretty easy. Blade templates are just PHP files, but the filenames end with `.blade.php` instead of `.php`.

> Blade is the templating engine provided by Laravel. It is simple, yet powerful. And best of all, it has a clean syntax that lets you embed functionality in HTML files without all those ugly starting and ending `php` tags.

Edit the `app/views/hello.blade.php` file to match what's below.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Getting Stuff Done with Laravel</title>
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      {{ HTML::style('/css/bootstrap.min.css') }}
7    </head>
8    <body>
9      <h1>Hello, world!</h1>
10
11     <!-- Javascript at the end of the page -->
12     {{ HTML::script('/js/jquery.js') }}
13     {{ HTML::script('/js/bootstrap.min.js') }}
14   </body>
15 </html>
```

Line 1
: This is HTML 5

Line 5
: Bootstrap is built for mobile first. This line specifies the width and how the zoom factor of the screen.

Line 6
: Here we use a bit of Blade syntax (the opening and closing {{ }}). This will output the PHP value between the curly braces. What we're outputting is a Laravel HTML helper (it's actually a facade) that will output the HTML for the style sheet we're specifying.

Lines 12 and 13
: Here we're using a different helper to output the HTML to load the two javascript files we need.

That's it! Bring up `http://gsd.localhost.com` in your browser or if you're using the PHP built-in server, localhost on the appropriate port. *(You know what? I'm not going to mention the alternative methods again. If you're using an alternative server or domain name, you know what to do.)*

Expand template to our mockup

Edit the `app/views/hello.blade.php` file to match what's below.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Getting Stuff Done with Laravel</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    {{ HTML::style('/css/bootstrap.min.css') }}
    {{ HTML::style('/css/bootstrap-theme.min.css') }}
    <style type="text/css">
      body {
        margin-top: 70px;
      }
    </style>
  </head>
  <body>

    {{-- Top Navbar --}}
    <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
        <div class="navbar-header">
          <a class="navbar-brand" href="#">Getting Stuff Done</a>
        </div>
        <ul class="nav navbar-nav">
          <li class="active">
            <a href="#">Actions list</a>
          </li>
        </ul>
        <form class="navbar-form navbar-right">
          <button type="submit" class="btn btn-success">
            <span class="glyphicon glyphicon-plus-sign"></span>
            Add Task
          </button>
        </form>
      </div>
    </div>

    <div class="container">
      <div class="row">
        <div class="col-md-3">

          {{-- Active Lists --}}
          <div class="panel panel-info">
```

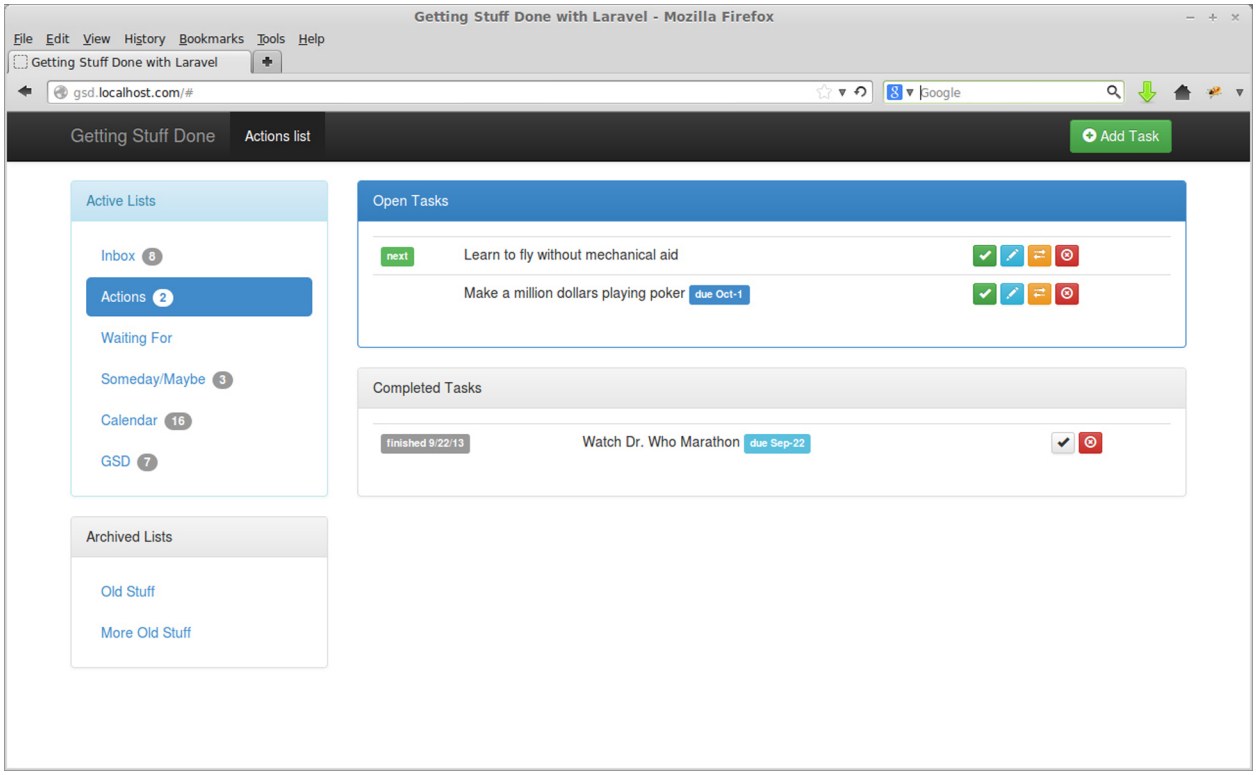
```
43              <div class="panel-heading">Active Lists</div>
44              <div class="panel-body">
45                <ul class="nav nav-pills nav-stacked">
46                  <li><a href="#">Inbox <span class="badge">8</span></a></li>
47                  <li class="active">
48                    <a href="#">Actions <span class="badge">2</span></a>
49                  </li>
50                  <li><a href="#">Waiting For</a></li>
51                  <li>
52                    <a href="#">Someday/Maybe <span class="badge">3</span></a>
53                  </li>
54                  <li>
55                    <a href="#">Calendar <span class="badge">16</span></a>
56                  </li>
57                  <li><a href="#">GSD <span class="badge">7</span></a></li>
58                </ul>
59              </div>
60            </div>
61
62            {{-- Archived Lists --}}
63            <div class="panel panel-default">
64              <div class="panel-heading">Archived Lists</div>
65              <div class="panel-body">
66                <ul class="nav nav-stacked">
67                  <li><a href="#">Old Stuff</a></li>
68                  <li><a href="#">More Old Stuff</a></li>
69                </ul>
70              </div>
71            </div>
72
73          </div>
74          <div class="col-md-9">
75
76            {{-- Open Tasks --}}
77            <div class="panel panel-primary">
78              <div class="panel-heading">Open Tasks</div>
79              <div class="panel-body">
80                <table class="table table-hover">
81                  <tbody>
82                    <tr>
83                      <td><span class="label label-success">next</span></td>
84                      <td>Learn to fly without mechanical aid</td>
```

```
85                      <td>
86                        <a href="#" class="btn btn-success btn-xs"
87                          title="Mark complete">
88                          <span class="glyphicon glyphicon-ok"></span>
89                        </a>
90                        <a href="#" class="btn btn-info btn-xs"
91                          title="Edit task">
92                          <span class="glyphicon glyphicon-pencil"></span>
93                        </a>
94                        <a href="#" class="btn btn-warning btn-xs"
95                          title="Move task">
96                          <span class="glyphicon glyphicon-transfer"></span>
97                        </a>
98                        <a href="#" class="btn btn-danger btn-xs"
99                          title="Delete task">
100                         <span class="glyphicon glyphicon-remove-circle"></span>
101                       </a>
102                     </td>
103                   </tr>
104                   <tr>
105                     <td></td>
106                     <td>
107                       Make a million dollars playing poker
108                       <span class="label label-primary">due Oct-1</span>
109                     </td>
110                     <td>
111                       <a href="#" class="btn btn-success btn-xs"
112                         title="Mark complete">
113                         <span class="glyphicon glyphicon-ok"></span>
114                       </a>
115                       <a href="#" class="btn btn-info btn-xs"
116                         title="Edit task">
117                         <span class="glyphicon glyphicon-pencil"></span>
118                       </a>
119                       <a href="#" class="btn btn-warning btn-xs"
120                         title="Move task">
121                         <span class="glyphicon glyphicon-transfer"></span>
122                       </a>
123                       <a href="#" class="btn btn-danger btn-xs"
124                         title="Delete task">
125                         <span class="glyphicon glyphicon-remove-circle"></span>
126                       </a>
```

```
                    </td>
                  </tr>
                </tbody>
              </table>
            </div>
          </div>

          {{-- Completed Tasks --}}
          <div class="panel panel-default">
            <div class="panel-heading">Completed Tasks</div>
            <div class="panel-body">
              <table class="table table-hover">
                <tbody>
                  <tr>
                    <td>
                      <span class="label label-default">finished 9/22/13</span>
                    </td>
                    <td>
                      Watch Dr. Who Marathon
                      <span class="label label-info">due Sep-22</span>
                    </td>
                    <td>
                      <a href="#" class="btn btn-default btn-xs"
                        title="Mark not completed">
                        <span class="glyphicon glyphicon-ok"></span>
                      </a>
                      <a href="#" class="btn btn-danger btn-xs"
                        title="Delete task">
                        <span class="glyphicon glyphicon-remove-circle"></span>
                      </a>
                    </td>
                  </tr>
                </tbody>
              </table>
            </div>
          </div>

        </div>
      </div>


    </div>
```

```
169
170     <!-- Javascript at the end of the page -->
171     {{ HTML::script('/js/jquery.js') }}
172     {{ HTML::script('/js/bootstrap.min.js') }}
173   </body>
174 </html>
```

I'm not going to explain every line of HTML. This is just a mock up. Save it and bring up your web browser and you should see a page similar to the one below.



Our Mock Up

Now, that's a pretty todo list ain't it?

Dogfooding

Time to wrap this chapter and update our todo list.

```
~/gsd$ gsd ls +gsd -x
Active Tasks in list '+gsd'
+---+------+----------------------------------+-------+
| # | Next | Description                      | Extra |
+---+------+----------------------------------+-------+
| 1 | YES  | Implement GET /lists             |       |
| 2 | YES  | Setup Routes                     |       |
| 3 |      | Create web application wireframe |       |
+---+------+----------------------------------+-------+
~/gsd$ gsd do 3 +gsd
Task 'Create web application wireframe' marked complete.
~/gsd$ gsd add "Add error/success box" +gsd -a
Next Action successfully added to gsd
~/gsd$ gsd add "Fill Active Lists using REST" +gsd -a
Next Action successfully added to gsd
~/gsd$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+----------------------------------------+---------------+
| # | Next | Description                            | Extra         |
+---+------+----------------------------------------+---------------+
| 1 | YES  | Add error/success box                  |               |
| 2 | YES  | Fill Active Lists using REST           |               |
| 3 | YES  | Implement GET /lists                   |               |
| 4 | YES  | Setup Routes                           |               |
|   | done | Create EditTaskCommand                 | Done 9/29/13  |
|   | done | Create ArchiveListCommand              | Done 10/4/13  |
|   | done | Create UnarchiveListCommand            | Done 10/4/13  |
|   | done | Add CommandBase::abort()               | Done 10/5/13  |
|   | done | Add to askForListId()                  | Done 10/5/13  |
|   | done | Check gsd help consistency             | Done 10/5/13  |
|   | done | Create MoveTaskCommand                 | Done 10/5/13  |
|   | done | Create RemoveTaskCommand               | Done 10/5/13  |
|   | done | Create RenameListCommand               | Done 10/5/13  |
|   | done | Remove blank line after gsd:list title | Done 10/5/13  |
|   | done | Create shell gsd script                | Done 10/6/13  |
|   | done | List tasks in ListAllCommand           | Done 10/6/13  |
|   | done | Chapter on setting up webserver        | Done 10/12/13 |
|   | done | Create Appendix for Apache install     | Done 10/12/13 |
|   | done | Create Appendix for nginx install      | Done 10/12/13 |
|   | done | Create web application wireframe       | Done 10/13/13 |
+---+------+----------------------------------------+---------------+
```

The *wireframe* in our todo list was actually this *mockup*. Earlier, I thought we'd do a wireframe for it first, but a mockup was quicker. Besides that I added a couple other tasks that came to mind.

Chapter 59 - Adding Feedback to the User

In This Chapter

In this chapter we'll start working on the live web page, using the mocked up web page as a model, and added notification boxes to display messages to the user.

Structuring the Views

There's only one view in our project, `hello.blade.php`, and since this is going to be a single page web application we could get by with a single view, but I thought it'd be informative to organize the views as if there would be multiple web pages in the application. The means using layouts, including sub-views, using *view namespacing*, etc. All standard Laravel stuff.

The Plan of Attack

The plan is to build up the live web page, using our mockup as the the model to pull from. We'll develop the page iteratively, each iteration bringing us closer to the goal.

Edit the `routes.php` file to contain what's below.

```
1  <?php
2
3  Route::get('/', function()
4  {
5      return View::make('live');
6  });
7  Route::get('mock', function()
8  {
9      return View::make('mockup');
10 });
11 ?>
```

This will give us two web pages, our mockup, and the live one we're building.

Rename the existing `views/hello.blade.php` to `views/mockup.blade.php`

```
~/gsd$ cd app/views
~/gsd/app/views$ mv hello.blade.php mockup.blade.php
```

Bring up `http://gsd.localhost.com/mock` in your browser. You should see the mockup page.

If you don't see the mockup page we built in the last chapter, make sure the **mod-rewrite** is enabled in your apache configuration.

> Since this is more of a process book than a reference book, I thought I'd share how I started working on this chapter. I haven't touched this book or project for a week. Saturday morning I said to myself, "Chuck, it's time to write another half-dozen chapters." So I looked at the todo list with that trusty little `gsd ls +gsd` utility, saw several next actions, and picked one. For me, todo lists are all-important, especially when dealing with multiple projects.

Finish cleaning up the directories and starting our live view.

```
~/gsd$ cd app/views
~/gsd/app/views$ rm -rf emails
~/gsd/app/views$ mkdir layouts
~/gsd/app/views$ mkdir partials
~/gsd/app/views$ echo "LIVE" > live.blade.php
```

Line 2
: We're not sending emails with this application, so kill the default email templates Laravel provides.

Line 3 and 4
: We're structuring the views to allow for a `layouts` namespace and a `partials` namespace.

Line 5

And just a simple live view with the word "LIVE" in it.

Views don't really have *namespacing* in the traditional sense. But we can access them using periods to separate folders. This allows us to do things like `View::make('users.addform')` or `@include ("partials.sidebar")`.

Bring up `http://gsd.localhost.com` in your browser. You should see a page with the word "LIVE" on it.

Building the Skeleton

Now let's build up a very basic live web page.

Edit `views/live/blade.php` to match what's below.

```
1  @extends("layouts.default")
2
3  @section("content")
4
5    Heeere's Johnnny!
6
7  @stop
```

Very simple at this point. The first line says we're going to extend another blade template. Which we'll create shortly. Then it starts a section, naming the section **content**. The section will contain the *Heeere's Johnnny!* line. Basically, everything from the start of the section until the `@stop` will be stored in an internal variable called **content**.

It's important to note that nothing is output to the browser here. This is because nothing is outside a `@section` block other than the blade command `@extends`.

Now create `views/layouts/default.blade.php` with the content below.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>
      @section('title')
        Getting Stuff Done
      @show
    </title>
    @section('styles')
      {{ HTML::style('/css/bootstrap.min.css') }}
      {{ HTML::style('/css/bootstrap-theme.min.css') }}
      {{ HTML::style('/css/gsd.css') }}
    @show
    @section('scripts')
      {{ HTML::script('/js/jquery.js') }}
      {{ HTML::script('/js/bootstrap.min.js') }}
      {{ HTML::script('/js/gsd.js') }}
    @show
  </head>
  <body>
    @yield('content')
  </body>
</html>
```

This is the beginning of our default layout. I took the `<head>` section from our mockup and added the scripts from the bottom of the page. This way all the global scripts and css are right there at the top Don't worry about the `@section` and `@show` sections yet. I'll explain why they're there in a bit. The `@yield` statement will simply output any section named `content`. *(Which we set up in `live.blade.php` as "Heeere's Johnnny!")*

Here we're actually outputting the web page. The sections end with `@show` so instead of just storing the section block internally, it outputs the section to the browser.

Reload your browser page `http://gsd.localhost.com`. You should see a page that says "Heeere's Johnnny!"

Adding gsd style and javascript

I added two additional files in the last step, a `gsd.css` and `gsd.js`. Let's create them.

Create `public/css/gsd.css` with the following content.

```
/* Getting Stuff Done specific styling */
body {
  margin-top: 70px;
}
```

Here we just copied the body margin styling from the mockup. Not much to warrant an entire file, but at least now there's a place to stick any additional styles we want.

Create `public/js/gsd.js` with the following content.

```
/* Getting Stuff Done Javascript */

/**
 * Fire after page loads
 */
$(function()
{
  gsd.initialize();
});

/**
 * gsd object
 */
var gsd = (function()
{
  // Private vars and functions -----------------------------------------

  return {

    // Public vars ------------------------------------------------

    // Public functions -------------------------------------------

    /**
     * Initialization
     */
```

```
27      initialize: function()
28      {
29        console.log("I'm initialized");
30      }
31    };
32  })();
```

Lines 6 - 9
This is jquery notation to call a function after the page loads. In our case, we'll call the `initialize()` method of the `gsd` object.

Line 14
Here we immediately execute a function and store the result in the `gsd` variable.

Line 16
If the `gsd` global variable needs any private vars or functions, then we'll stick them here.

Line 18
The immediate function we're executing and assigning to `gsd` will return an object, which may have variables (properties) and functions (methods).

Line 20
If the `gsd` object needs any public properties, we'll stick them here.

Line 27 - 30
We're saying the `gsd` object has a public method named `initialize()` which will output something to the javascript console. *(Well, technically we're returning an object with an `initialize` property which happens to be a function.)*

Load (or reload) `http://gsd.localhost.com` and see what happens.

The only visible change is the "Heeere's Johnnny!" line moved down on the screen (that's the gsd.css working). If you have a console window on your browser then you'll see an "I'm initialized" message in it.

Now we've got a nice structure to build on.

Adding a message box

So now let's add a little message box to our template.

Edit `views/layouts.default.php` to add the line before the `@yield` statement below

```
<body>
  @include('partials.notifications')
  @yield('content')
</body>
```

This will include the contents of the `notifications.blade.php` file, found in `views/partials`. Of course, this file doesn't exist yet.

So create it

```
<div class="alert alert-success">
  <button type="button" class="close" data-dismiss="alert">&times;</button>
  <div class="glyphicon glyphicon-ok"></div>
  Here's some success message.
</div>
```

Reload `http://gsd.localhost.com` and you should have a success message.

There's not much on the page, but that's okay. We're going to make it so we can display success or failure messages, and have them appear right before the content. (Which will be where the *Open Tasks* box appears on our mocked up page.).

Making message box a function

Edit `notifications.blade.php`, updating it to what's below.

```
1  <div id="message-area"></div>
2
3  <div id="success-message" style="display:none">
4    <div class="alert alert-success" id="success-message-id">
5      <button type="button" class="close" data-dismiss="alert">&times;</button>
6      <div class="glyphicon glyphicon-ok"></div>
7      success-message-text
8    </div>
9  </div>
10
11 <div>
12   <button class="btn btn-primary btn-app"
13     onclick="gsd.successMessage('Test message from button click')">
14   Test Message
15   </button>
16 </div>
```

Line 1

Here's the area we're going to display any alert type messages

Lines 3 to 9

Here we wrap the alert message we had earlier with a hidden div with the id "success-message". We add an id to the alert itself. And have the message text something easily searched for.

Lines 11 to 16

Here's a button to test it. The button calls the not-yet-created `gsd.successMessage()` function.

Let's create the javascript function.

Edit `gsd.js`, updating it to what's below.

```
/* Getting Stuff Done Javascript */

/**
 * Fire after page loads
 */
$(function()
{
    gsd.initialize();
});

/**
 * gsd object
 */
var gsd = (function()
{
    // Private vars and functions ----------------------------------------

    var alertTimer = null;

    /**
     * Display a message or error box
     * @param string msgType Either "success" or "error"
     * @param string message The message
     */
    function commonBox(msgType, message)
    {
        clearTimeout(alertTimer);

        $("#message-area").html(
            $("#" + msgType + "-message")
                .html()
                .replace(msgType + '-message-text', message)
                .replace(msgType + '-message-id', 'alert-id')
        );
        alertTimer = setTimeout(function()
        {
            $("#alert-id").alert("close");
            alertTimer = null;
        }, 8000);
    }


```

```
43      return {
44
45          // Public vars ------------------------------------------------
46
47          // Public functions -------------------------------------------
48
49          /**
50           * Initialization
51           */
52          initialize: function()
53          {
54              console.log("I'm initialized");
55          },
56
57          /**
58           * Display a success message box
59           * @param string message The message to display
60           */
61          successMessage: function(message)
62          {
63              commonBox("success", message);
64          }
65      };
66  })();
```

I'll explain the new stuff ....

Line 18
: Here's private variable to track the alert timer. We want any messages to only display for eight seconds, so we'll set a timer on it to close the box. This keeps track of the timer.

Lines 20 - 26
: Since a success alert and error alert will be almost identical. Here I coded the commonality between the two. The type of message is passed as the first argument.

Line 27
: If there's a timer still going

Lines 29 - 34
: What we're doing here is loading the HTML from the hidden #success-message box, replacing a couple strings, then inserting this HTML into the message display area (#message-area).

Lines 35 - 39
: Then we set an 8 second timer to automatically close the box.

Line 55
Don't forget that trailing comma. We're returning an object and the functions are simply properties of that object.

Lines 61 - 64
Implementing the `successMessage()` method is super easy, just call the private `commonBox()` method.

Give it a shot. Reload `http://gsd.localhost.com` and click the button to test things. You should get a "Test message from button click" alert that lingers around for 8 seconds before disappearing. And you can hit the close "X" in the box and it works.

Implementing the Error Message function

Update `views/partials/notifications.blade.php` to match what's below.

```
1  <div id="message-area"></div>
2
3  <div id="success-message" style="display:none">
4    <div class="alert alert-success" id="success-message-id">
5      <button type="button" class="close" data-dismiss="alert">&times;</button>
6      <div class="glyphicon glyphicon-ok"></div>
7      success-message-text
8    </div>
9  </div>
10
11 <div id="error-message" style="display:none">
12   <div class="alert alert-danger" id="error-message-id">
13     <button type="button" class="close" data-dismiss="alert">&times;</button>
14     <div class="glyphicon glyphicon-remove"></div>
15     error-message-text
16   </div>
17 </div>
```

Just added the #error-message hidden div with the message just like the success, but formatted for an error.

Update the tail end of your `gsd.js` to match what's below.

```
      // above here is exactly the same
    },

    /**
     * Display an error message box
     * @param string message The message to display
     */
    errorMessage: function(message)
    {
      commonBox("error", message);
    }
  };
})();
```

Just a few lines of javascript to implement the `errorMessage()` method.

Looking good. Of course, I stripped out the test button. It works. You can add it back yourself to test the error functionality if you'd like.

Dogfooding

I manually edited the gsd.txt file and removed everything completed earlier than Part 4 of this book.

Then, here's what I did.

```
$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+----------------------------------+---------------+
| # | Next | Description                      | Extra         |
+---+------+----------------------------------+---------------+
| 1 | YES  | Add error/success box            |               |
| 2 | YES  | Fill Active Lists using REST     |               |
| 3 | YES  | Implement GET /lists             |               |
| 4 | YES  | Setup Routes                     |               |
|   | done | Create web application wireframe | Done 10/13/13 |
+---+------+----------------------------------+---------------+
$ gsd do 1 +gsd
Task 'Add error/success box' marked complete.
$ gsd add "Finish Top NavBar" +gsd -a
Next Action successfully added to gsd
$ gsd ls +gsd
All Tasks in list '+gsd'
```

```
+---+------+----------------------------------+---------------+
| # | Next | Description                      | Extra         |
+---+------+----------------------------------+---------------+
| 1 | YES  | Fill Active Lists using REST     |               |
| 2 | YES  | Finish Top NavBar                |               |
| 3 | YES  | Implement GET /lists             |               |
| 4 | YES  | Setup Routes                     |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box            | Done 10/19/13 |
+---+------+----------------------------------+---------------+
```

Marked something off the list. Added something to the list. It's progress.

Chapter 60 - Setting up the AJAX routes

In This Chapter

In this chapter we'll set up the routing to match the AJAX calls defined in chapter 57.

Using a Resource Controller

Laravel provides a neat trick it let's you set up **Resource Controllers**. These controllers take the pain out of setting up RESTful controllers around resources.

The best way to learn about this is just to do it.

Look at your current routes, using artisan

```
~/gsd$ art route
+--------+-----------+------+---------+----------------+---------------+
| Domain | URI       | Name | Action  | Before Filters | After Filters |
+--------+-----------+------+---------+----------------+---------------+
|        | GET /     |      | Closure |                |               |
|        | GET /mock |      | Closure |                |               |
+--------+-----------+------+---------+----------------+---------------+
```

Pretty much what's expected, right? The main page and the mockup page we set up in `app/routes.php`

Add the following line to `app/routes.php`

```
Route::resource('lists', 'ListController');
```

Now look at the routes.

```
~/gsd$ art route
+-------------------------+--------------+-----------------------+
| URI                     | Name         | Action                |
+-------------------------+--------------+-----------------------+
| GET /                   |              | Closure               |
| GET /mock               |              | Closure               |
| GET /lists              | lists.index  | ListController@index  |
| GET /lists/create       | lists.create | ListController@create |
| POST /lists             | lists.store  | ListController@store  |
| GET /lists/{lists}      | lists.show   | ListController@show   |
| GET /lists/{lists}/edit | lists.edit   | ListController@edit   |
| PUT /lists/{lists}      | lists.update | ListController@update |
| PATCH /lists/{lists}    |              | ListController@update |
| DELETE /lists/{lists}   | lists.destroy | ListController@destroy |
+-------------------------+--------------+-----------------------+
```

(I edited the output above, removing several columns to only show what's relevant.)

Pretty cool, huh? That one `Route::resource()` command set up all kinds of routes for us. It also named the routes. Nice.

It did add a couple extra routes which we're not implementing. So let's get rid of those.

Update the line just added to `app/routes.php` to match what's below.

```
Route::resource('lists', 'ListController', array(
    'except' => array('create', 'edit')));
```

Now look at the routes again.

```
~/gsd$ art route
+------------------------+--------------+------------------------+
| URI                    | Name         | Action                 |
+------------------------+--------------+------------------------+
| GET /                  |              | Closure                |
| GET /mock              |              | Closure                |
| GET /lists             | lists.index  | ListController@index   |
| POST /lists            | lists.store  | ListController@store   |
| GET /lists/{lists}     | lists.show   | ListController@show    |
| PUT /lists/{lists}     | lists.update | ListController@update  |
| PATCH /lists/{lists}   |              | ListController@update  |
| DELETE /lists/{lists}  | lists.destroy | ListController@destroy |
+------------------------+--------------+------------------------+
```

That's over half our RESTful routes set up with one `Route` command. Ain't Laravel grand?

Don't worry about the PUT and PATCH duplication. This provides two different HTTP verbes (PUT and PATCH) that are used to do the same thing. We'll use the PUT verb in our application.

Finish the routes

There's only three routes left to add. The ones to archive, unarchive, and rename the list.

Update your `app/routes.php` file to match what's below.

```
<?php

Route::get('/', function()
{
    return View::make('live');
});
Route::get('mock', function()
{
    return View::make('mockup');
});

Route::resource('lists', 'GSD\Controllers\ListController', array(
```

```
13      'except' => array('create', 'edit')));
14  Route::post('lists/{lists}/archive', array(
15      'as'   => 'lists.archive',
16      'uses' => 'GSD\Controllers\ListController@archive',
17  ));
18  Route::post('lists/{lists}/unarchive', array(
19      'as'   => 'lists.unarchive',
20      'uses' => 'GSD\Controllers\ListController@unarchive',
21  ));
22  Route::post('lists/{source}/rename/{dest}', array(
23      'as'   => 'lists.rename',
24      'uses' => 'GSD\Controllers\ListController@rename',
25  ));
26
27  ?>
```

In addition to adding the three routes (lists.archive, lists.unarchive, and lists.rename) I also expanded the controller name to be fully namespaced.

Creating the Controller

Okay, let's create the controller to handle the routes we just added.

Guess what the quickest way to do it is? To use Laravel to help us, of course.

Issue the artisan command below.

```
1  ~/gsd$ art controller:make ListController --except=create,edit \
2  > --path=app/src/GSD/Controllers
3  Controller created successfully!
```

That one command creates a nice skeleton for us. Check out the file created in your `app/src/GSD/Controllers` directory.

Finishing the ListController skeleton

At the time of this writing, I'm using Laravel 4.0 and the namespace option doesn't exist on the artisan **controller:make** command. So we need to do some extra editing.

Below is the edited version of `ListController`. Update your copy to match it.

```
<?php namespace GSD\Controllers;

use Response;
use Todo;

class ListController extends \Controller {

    /**
     * Returns a list of lists
     */
    public function index()
    {
        return Response::json(array('error' => 'index not done'));
    }

    /**
     * Create a new list
     */
    public function store()
    {
        return Response::json(array('error' => 'store not done'));
    }

    /**
     * Return the list
     *
     * @param  string $id The list name
     */
    public function show($id)
    {
        return Response::json(array('error' => 'show not done'));
    }

    /**
     * Update the specified list.
     *
     * @param  string $id The list name
     */
    public function update($id)
```

```
40      {
41          return Response::json(array('error' => 'update not done'));
42      }
43
44      /**
45       * Uncreate the specified list
46       *
47       * @param  string $id The list name
48       */
49      public function destroy($id)
50      {
51          return Response::json(array('error' => 'destroy not done'));
52      }
53
54      /**
55       * Archive the specified list
56       *
57       * @param  string $id The list name
58       */
59      public function archive($id)
60      {
61          return Response::json(array('error' => 'archive not done'));
62      }
63
64      /**
65       * Unarchive the specified list
66       *
67       * @param  string $id The list name
68       */
69      public function unarchive($id)
70      {
71          return Response::json(array('error' => 'unarchive not done'));
72      }
73
74      /**
75       * Rename $source list to $dest
76       *
77       * @param  string $source The source list name
78       * @param  string $dest   The destination list name
79       */
80      public function rename($source, $dest)
81      {
```

```
82          return Response::json(array('error' => 'rename not done'));
83      }
84  }
85  ?>
```

I'm not going to go over every line because this is just a skeleton. Each method returns a json response with an error message.

Testing a ListController method

To make sure our method and routes are all tied together correctly, pull up `http://gsd.localhost.com/lists` in your browser.

This should execute the `ListController::index()` method and you should see the following line in your browser.

```
1  {"error":"index not done"}
```

Dogfooding

```
1  ~/gsd$ gsd ls +gsd -x
2  Active Tasks in list '+gsd'
3  +---+------+------------------------------+-------+
4  | # | Next | Description                  | Extra |
5  +---+------+------------------------------+-------+
6  | 1 | YES  | Fill Active Lists using REST |       |
7  | 2 | YES  | Finish Top NavBar            |       |
8  | 3 | YES  | Implement GET /lists         |       |
9  | 4 | YES  | Setup Routes                 |       |
10 +---+------+------------------------------+-------+
11 ~/gsd$ gsd do 4 +gsd
12 Task 'Setup Routes' marked complete.
13 ~/gsd$ gsd ls +gsd
14 All Tasks in list '+gsd'
15 +---+------+---------------------------------+---------------+
16 | # | Next | Description                     | Extra         |
17 +---+------+---------------------------------+---------------+
18 | 1 | YES  | Fill Active Lists using REST    |               |
19 | 2 | YES  | Finish Top NavBar               |               |
```

```
| 3 | YES  | Implement GET /lists             |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box            | Done 10/19/13 |
|   | done | Setup Routes                     | Done 10/20/13 |
+---+------+----------------------------------+---------------+
```

I just marked as finished the ‘Setup Routes’ item. Good enough. This chapter is wrapped.

Chapter 61 - Adding the Top Nav Bar

In This Chapter

In this chapter we'll start setting up the navigation bar on the top of the screen.

Creating the partial

First let's pull the HTML code from our mockup and add it to the layout. To keep things nice and organized, we'll just stick it in another partial.

Edit `default.blade.php` in `app/views/layouts` as specified below.

```
1    <!-- every above here is the same -->
2    <body>
3      @include('partials.topnavbar')
4      @include('partials.notifications')
5      @yield('content')
6    </body>
7  </html>
```

The line we added will include a new partial, named 'topnavbar', right after the body is opened.

Create `topnavbar.blade.php` in the `app/views/partials` directory with the content below.

```
<div class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <a class="navbar-brand" href="#">Getting Stuff Done</a>
    </div>
    <ul class="nav navbar-nav">
      <li class="active">
        <a href="#">Actions list</a>
      </li>
    </ul>
    <form class="navbar-form navbar-right">
      <button type="submit" class="btn btn-success">
        <span class="glyphicon glyphicon-plus-sign"></span>
        Add Task
      </button>
    </form>
  </div>
</div>
```

Reload `http://gsd.localhost.com` and you should see the navbar.

Loading the default list

When the web page first loads it should display the default list. So let's put in the hooks to make that happen.

Edit `app/routes.php` and change the route for the live page to match what's below.

```
<?php

Route::get('/', function()
{
  return View::make('live')
    ->withDefaultList(Config::get('todo.defaultList'));
});
?>
```

Here we're still returning the view named `live`, but now we're also assigning a variable to it containing the default list. The name of this variable will be `$default_list`.

Laravel provides multiple ways to set view variables.

We could have used the two alternatives below and achieved the same thing.

```
// Variables as an array to make()
return View::make('live', array(
  'default_list' => Config::get('tododefaultList'));

// Using a generic with()
return View::make('live')
  ->with('default_list', Config::get('todo.defaultList'));
```

For a single variable assignment, I like the `withVariableName()` technique because it is concise and expressive.

Edit `default.base.php` in `app/views/partials`. We're only updating the **scripts** section.

```
@section('scripts')
  {{ HTML::script('/js/jquery.js') }}
  {{ HTML::script('/js/bootstrap.min.js') }}
  {{ HTML::script('/js/gsd.js') }}
  <script type="text/javascript">
    gsd.defaultList = "{{ $default_list }}";
  </script>
@show
```

Easy. Here it's just a matter of assigning the value of `$default_list` to a property of the `gsd` javascript object.

Notice the opening double braces ({{) and closing double braces (}})? This tells Laravel's Blade compiler to replace everything between the braces with the value of the variable. Using triple braces will also escape the value for HTML. (Like using htmlentities() on the value.)

Now, let's edit the `gsd.js` file to show the property.

```
// Everything above here's the same

// Public vars ------------------------------------------------

defaultList: null,

// Public functions -------------------------------------------

/**
 * Initialization
 */
initialize: function()
{
  this.successMessage("defaultList = " + this.defaultList);
},

// Everything below here's the same
```

We're adding the `defaultList` as a public property. This isn't absolutely needed, but it is a good practice. Then, in the `initialize()` function, we're outputting a message showing the value.

Reload `http://gsd.localhost.com` in your browser. You should see a green alert telling you the assignment worked.

Structuring the Nav Bar

When examining the mocked up web page I realize that a few things are missing.

- There's no way to archive or unarchive a list
- There's no way to rename a list
- There's no way to create new lists

No problem. The name of the list (where it currently says "Actions list" in the navbar) can become a pulldown menu to add those functions. So we'll do that.

Edit `partials/topnavbar.blade.php` to match what's below.

```
<div class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <a class="navbar-brand" href="/">Getting Stuff Done</a>
    </div>
    <ul class="nav navbar-nav">
      <li class="active">
        <a href="/" data-toggle="dropdown" data-target="#">
          <span id="list-name">Actions</span> list <span class="caret"></span>
        </a>
        <ul class="dropdown-menu">
          <li><a href="/" id="menu-archive">
            <span id="menu-archive-text">Archive</span> list</a>
          </li>
          <li><a href="/" id="menu-rename">Rename list</a></li>
          <li class="divider"></li>
          <li><a href="/" id="menu-create">Create new list</a></li>
        </ul>
      </li>
    </ul>
    <form class="navbar-form navbar-right">
      <button type="submit" class="btn btn-success" id="button-add">
        <span class="glyphicon glyphicon-plus-sign"></span>
        Add Task
      </button>
    </form>
  </div>
</div>
```

Here we added the dropdown to handle the missing actions and added an `id` attribute on several items.

Making our first AJAX call

Now let's set up the javascript `gsd.initialize()` method to load the default list.

Update `gsd.js` and change `initialize()` to match what's below.

```
/**
 * Initialization
 */
initialize: function()
{
  this.loadList(this.defaultList, false);
},
```

Pretty simple. The initialize function is going to call a `loadList()` function to load the default list. The `false` tells the `loadList()` method that the list is not archived.

Now we need to create that `loadList()` method.

Update `gsd.js` and add the new function below.

```
errorMessage: function(message)
{
  commonBox("error", message);
},

/**
 * Load a list via AJAX and display it
 * @param string  name    The list name
 * @param boolean archive Is it an archived list?
 */
loadList: function(name, archived)
{
  var url = "/lists/" + name;
  if (archived) url += "?archived=1";
  $.ajax({
    url: url,
    error: function(hdr, status, error)
    {
      gsd.errorMessage("loadList " + status + " - " + error);
    },
    success: function(data)
    {
      if (data && data.error)
      {
        gsd.errorMessage("loadList error: " + data.error);
        return;
```

```
27        }
28        gsd.successMessage("Cool");
29        console.log(data.list);
30      }
31    });
32  }
```

Line 4
Don't forget to add that trailing comma to the end of `errorMessage()`.

Line 13 - 14
Build the url to call. Add a query argument to specify whether the list is archived or not.

Line 15
We're using jQuery's `ajax()` method to make the call. The arguments we're passing are the url and two functions. One to handle HTTP errors and the other to handle a successful call. Note that this happens asynchronously. Meaning execution in our code continues past the `ajax()` call and the `error` or `success` method is not called until a response is received from our server.

Lines 17 - 20
The reason we're providing an error callback is in case something goes horribly wrong calling our method. What if the URL had a typo? This method will catch those type errors.

Lines 21 - 30
If the AJAX call is successful, we still check if we had an error object and if so, then display the error. Otherwise, just present a green alert saying "Cool".

Give it a shot. Reload `http://gsd.localhost.com`. You should be receiving an error saying "loadList error: show not done".

Doing the server side of the REST

Now we'll edit our controller to load the list successfully.

First make sure the top of `ListController.php` matches what's below.

```
<?php namespace GSD\Controllers;

use GSD\Entities\ListInterface;
use Input;
use Response;
use Todo;

?>
```

A couple other `use` statements were added.

Next update the `show()` method in `ListController` to match what's below.

```
<?php
/**
 * Return the list
 *
 * @param  string $id The list name
 */
public function show($id)
{
  $archived = !! Input::get('archived');
  try
  {
    $list = Todo::get($id, $archived);
    $result = $this->toAssoc($list);
  }
  catch (\RuntimeException $e)
  {
    $result = array(
      'error' => $e->getMessage()
    );
  }
  return Response::json($result);
}
?>
```

This should be pretty self-explanatory. Just a couple notes.

- The list is loaded in a try/catch block in case it doesn't exist. If there's an error then we return an error object with the error's description.

- Once the list is loaded, we pass it to `toAssoc()` to convert to an associative array.

The `toAssoc()` method doesn't exist yet, so let's add it.

Add the new `toAssoc()` method to the bottom of the `ListController` class.

```
<?php
/**
 * Convert a TodoList to an associative array
 * @param ListInterface $list The List
 * @return array The associative array
 */
protected function toAssoc(ListInterface $list)
{
  $return = array(
    'list' => array(
      'name' => $list->get('id'),
      'title' => $list->get('title'),
      'subtitle' => $list->get('subtitle'),
      'archived' => $list->get('archived'),
      'tasks' => array(),
    ),
  );
  foreach ($list->tasks() as $task)
  {
    $array = array(
      'isNext' => $task->isNextAction(),
      'isCompleted' => $task->isComplete(),
      'descript' => $task->description(),
      'dateDue' => $task->dateDue(),
      'dateCompleted' => $task->dateCompleted(),
      );
    if ($array['dateDue'])
    {
      $array['dateDue'] = $array['dateDue']->timestamp * 1000;
    }
    if ($array['dateCompleted'])
    {
      $array['dateCompleted'] = $array['dateCompleted']->timestamp * 1000;
    }
```

```
35      $return['list']['tasks'][] = $array;
36    }
37    return $return;
38  }
39  ?>
```

No real magic here either. Just creating the structure the javascript on the web page is expecting. If any dates exist, then they're converted to Javascript timestamps. These are the same as PHP timestamps, but can track microseconds, so we multiple the PHP timestamp by 1000 to make the value something Javascript can use.

Reload `http://gsd.localhost.com` and you should now receive a green alert that says "Cool".

Dogfooding

After reviewing our +gsd todo list, I've realized there's nothing we can mark off the list. Bummer. I really like knocking things off that list. Oh well. Next chapter we'll mark something off.

Chapter 62 - Finishing the Top Nav Bar

In This Chapter

In this chapter we'll finish the functionality of the top nav bar.

Assigning javascript functions to the navbar

First thing we're doing is going through all the possible actions that can occur on the top nav bar and assigning functions to them.

Add the following javascript functions to `gsd.js`. Put them right after the `commonBox()` function. This way they're private to the object.

```
1  /**
2   * Handle click on the top nav menu archive/unarchive option
3   */
4  function menuArchiveClick()
5  {
6    gsd.errorMessage("gsd.menuArchiveClick() not done");
7    return false;
8  }
9
10 /**
11  * Handle click on the top nav menu rename option
12  */
13 function menuRenameClick()
14 {
15   gsd.errorMessage("gsd.menuRenameClick() not done");
16   return false;
17 }
18
19 /**
20  * Handle click on the top nav menu create option
21  */
```

```
22 function menuCreateClick()
23 {
24   gsd.errorMessage("gsd.menuCreateClick() not done");
25   return false;
26 }
27
28 /**
29  * Handle click on the add task button
30  */
31 function buttonAddClick()
32 {
33   gsd.errorMessage("gsd.buttonAddClick() not done");
34   return false;
35 }
```

These funtions will be the handlers to the various actions that can happen in the top navbar.

Next, modify the `initialize()` method in `gsd.js` to match what's below.

```
1  /**
2   * Initialization
3   */
4  initialize: function()
5  {
6    // Assign various handlers
7    $("#menu-archive").click(menuArchiveClick);
8    $("#menu-rename").click(menuRenameClick);
9    $("#menu-create").click(menuCreateClick);
10   $("#button-add").click(buttonAddClick);
11
12   // Load the default list
13   this.loadList(this.defaultList, false);
14 },
```

We're using jQuery to assign the click handlers to every action the top navbar can perform.

Reload `http://gsd.localhost.com` and test the various actions. They should all show error messages that the handler's not done yet.

Loading the result into the navbar

Edit `gsd.js` and add the following line after the `var alertTimer = null;` line.

```
var currentList = null;
```

This is where we're going to store the list data loaded by the AJAX call.

Edit `gsd.js` and add the following function after the `buttonAddClick()` function

```
/**
 * Update the navbar for the current list
 */
function updateNavBar()
{
  $("#list-name").html("+" + currentList.name);
  $("#menu-archive-text").html(
    currentList.archived ? "Unarchive" : "Archive"
  );
  $("#button-add").prop("disabled", currentList.archived);
}
```

This is a private function of the `gsd` object which will be used to update the Nav bar from `currentList`. It's doing three things:

1. Updating the list name for the pulldown menu, placing a plus (+) in front of it.
2. Updating the first pulldown menu item to say either "Archive list" or "Unarchive list", depending on whether the current list is archived or not.
3. Disabling the "Add Task" button if the list is archived. We can't add tasks to archived lists.

Update the `loadList()` method in `gsd.js` to match what's below.

```
/**
 * Load a list via AJAX and display it
 * @param string  name    The list name
 * @param boolean archive Is it an archived list?
 */
loadList: function(name, archived)
{
  var url = "/lists/" + name;
  if (archived) url += "?archived=1";
  $.ajax({
    url: url,
    error: function(hdr, status, error)
    {
      gsd.errorMessage("loadList " + status + ' - ' + error);
    },
    success: function(data)
    {
      if (data && data.error)
      {
        gsd.errorMessage("loadList error: " + data.error);
        return;
      }
      currentList = data.list;
      updateNavBar();
    }
  });
}
```

The only change here is the last two lines of the `success` callback. We assign the data returned by the function to the private `currentList` variable and then call the private `updateNavBar()` function.

Give it a go. Reload `http://gsd.localhost.com`

I want to wait to implement these four click handlers. It makes more sense to see more of the list on the screen so when we implement the handlers we can visually see what happened.

Dogfooding

```
$ gsd ls +gsd -x
Active Tasks in list '+gsd'
+---+------+------------------------------+-------+
| # | Next | Description                  | Extra |
+---+------+------------------------------+-------+
| 1 | YES  | Fill Active Lists using REST |       |
| 2 | YES  | Finish Top NavBar            |       |
| 3 | YES  | Implement GET /lists         |       |
+---+------+------------------------------+-------+
$ gsd do 2 +gsd
Task 'Finish Top NavBar' marked complete.
$ gsd add "Finish gsd.menuArchiveClick()" +gsd
Todo successfully added to gsd
$ gsd add "Finish gsd.menuRenameClick()" +gsd
Todo successfully added to gsd
$ gsd add "Finish gsd.menuCreateClick()" +gsd
Todo successfully added to gsd
$ gsd add "Finish gsd.buttonAddClick()" +gsd
Todo successfully added to gsd
$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+----------------------------------+---------------+
| # | Next | Description                      | Extra         |
+---+------+----------------------------------+---------------+
| 1 | YES  | Fill Active Lists using REST     |               |
| 2 | YES  | Implement GET /lists             |               |
| 3 |      | Finish gsd.buttonAddClick()      |               |
| 4 |      | Finish gsd.menuArchiveClick()    |               |
| 5 |      | Finish gsd.menuCreateClick()     |               |
| 6 |      | Finish gsd.menuRenameClick()     |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box            | Done 10/19/13 |
|   | done | Finish Top NavBar                | Done 10/20/13 |
|   | done | Setup Routes                     | Done 10/20/13 |
+---+------+----------------------------------+---------------+
```

One task was marked off the list, but four were added. Seems like we're moving backwards. That's okay. I didn't want to forget to do all those handlers. They'll be quick and easy once we get to them.

Chapter 63 - The Side Navigation

In This Chapter

In this chapter we'll build the side navigation for our web application.

Updating the layout

First let's update the layout to have the final structure.

Edit `default.blade.php` in `views/layouts`. The section within the **<body>** tag should match what's below

```
1  <body>
2    @include('partials.topnavbar')
3    @include('partials.notifications')
4
5    <div class="container">
6      <div class="row">
7        <div class="col-md-3">
8
9          @include('partials.sidebar')
10
11       </div>
12       <div class="col-md-9">
13         @yield('content')
14       </div>
15     </div>
16   </div>
17 </body>
```

Line 5 and 6

Here we add a couple divs to start a container, and a row within the container.

Lines 7 - 11
: This adds a three-wide column to the row and pulls in the yet-to-be-created sidebar into that column.

Lines 12 - 14
: Next we add a nine-wide column and have the content from the `live.blade.php` template show in it.

Creating the sidebar

The sidebar will contain two lists: Active lists and Archived lists.

Create a new file `sidebar.blade.php` in the `views/partials` directory. Make the contents match what's below.

```
1  {{-- Active Lists --}}
2  <div class="panel panel-info">
3    <div class="panel-heading">Active Lists</div>
4    <div class="panel-body">
5      <ul class="nav nav-pills nav-stacked" id="active-lists">
6        <li><a href="#">Not Done</a></li>
7      </ul>
8    </div>
9  </div>
10
11 {{-- Archived Lists --}}
12 <div class="panel panel-default">
13   <div class="panel-heading">Archived Lists</div>
14   <div class="panel-body">
15     <ul class="nav nav-pills nav-stacked" id="archived-lists">
16       <li><a href="#">Not Done</a></li>
17     </ul>
18   </div>
19 </div>
```

Pretty straight-forward html.

Finishing the AJAX call.

Next let's finish out the AJAX call that returns the list of lists.

Edit `ListController.php` and update the `index()` method to match the following code.

```
<?php
/**
 * Returns a list of lists
 */
public function index()
{
  $archived = !! Input::get('archived');
  $lists = Todo::allLists($archived);

  $return = array(
    'lists' => array(),
  );
  foreach ($lists as $listId)
  {
    $list = Todo::get($listId, $archived);
    $return['lists'][] = array(
      'name'           => $listId,
      'title'          => $list->title(),
      'subtitle'       => $list->get('subtitle'),
      'isArchived'     => $list->isArchived(),
      'numNextActions' => $list->taskCount('next'),
      'numNormal'      => $list->taskCount('todo'),
      'numCompleted'   => $list->taskCount('done'),
    );
  }
  return Response::json($return);
}
?>
```

Simple, huh? We use the Todo facade to retrieve the lists, then build a return object containing additional details for each list.

Bring up `http://gsd.localhost.com/lists` in your browser and you should see the JSON return.

If you're using FireFox, I suggest installing the JSONovich plugin. It makes JSON render in the browser in an easy-to-view way.

Updating the Javascript

Now that we have a place to display the lists (the sidebar) and an AJAX method to fetch the lists, let's tie everything together with some javascript.

Edit the top of `gsd.js` to match what's below.

```
1  [snip]
2  // Private vars and functions ----------------------------------------
3
4  var alertTimer = null;
5  var currentList = null;
6  var activeLists = null;
7  var archivedLists = null;
8
9  [snip]
```

Lines 6 and 7 were added to provide a place to store the lists.

Add the two functions below to `gsd.js`. Put these functions after the `updateNavBar()` function.

```
1  /**
2   * Show one of the list of lists on the sidebar
3   */
4  function showSidebarList(archived)
5  {
6    var list = archived ? archivedLists : activeLists;
7    var ul = archived ? $("#archived-lists") : $("#active-lists");
8    var build = [];
9
10   // No items in list of lists?
11   if (list.length == 0)
12   {
13     ul.html('<li>No archived lists</li>');
14     return;
15   }
16
```

```
17      // Loop through each item, building html for the LI
18      for (var i = 0; i < list.length; i++)
19      {
20        var html = '<li';
21        var l = list[i];
22        var numTasks = l.numNextActions + l.numNormal;
23        if (archived == currentList.archived && l.name == currentList.name)
24           html += ' class="active"';
25        html += '><a href="javascript:gsd.loadList(\'' + l.name + '\',';
26        html += archived + ')">';
27        html += l.title;
28        if ( ! archived && numTasks > 0)
29        {
30          html += ' <span class="badge">' + numTasks + '</span>';
31        }
32        html += '</a></li>';
33        build.push(html);
34      }
35      ul.html(build.join("\n"));
36  }
37
38  /**
39   * Load the list of lists
40   * @param bool archived Load the archived lists?
41   */
42  function loadLists(archived)
43  {
44    var url = "/lists";
45    if (archived) url += "?archived=1";
46    $.ajax({
47      url: url,
48      error: function(hdr, status, error)
49      {
50        gsd.errorMessage("loadLists " + status + ' - ' + error);
51      },
52      success: function(data)
53      {
54        if (data && data.error)
55        {
56          gsd.errorMessage("loadList error: " + data.error);
57          return;
58        }
```

```
59        if (archived)
60        {
61          archivedLists = data.lists;
62        }
63        else
64        {
65          activeLists = data.lists;
66        }
67        showSidebarList(archived);
68      }
69    });
70  }
```

showSidebarList()

This function builds the HTML for the sidebar navigation and stuffs it into either Active Lists or Archived Lists

loadLists()

This function makes the Ajax callback, assigns the data to the object's private data storage (`archivedLists` or `activeLists`) and calls the `showSidebarList()` function.

Finally, update the `loadList` function in `gsd.js` to match what's below.

```
1   loadList: function(name, archived)
2   {
3     var url = "/lists/" + name;
4     if (archived) url += "?archived=1";
5     $.ajax({
6       url: url,
7       error: function(hdr, status, error)
8       {
9         gsd.errorMessage("loadList " + status + ' - ' + error);
10      },
11      success: function(data)
12      {
13        if (data && data.error)
14        {
15          gsd.errorMessage("loadList error: " + data.error);
16          return;
```

```
17        }
18        currentList = data.list;
19        updateNavBar();
20
21        // Reload the lists
22        loadLists(false);
23        loadLists(true);
24      }
25    });
26  }
```

Lines 21 - 23

These are the only new lines. What happens is after a list is loaded successfully, the data's stored in `currentList`, the navbar is updated, and finally these two functions are called to reload the active and archived list.

Reload `http://gsd.localhost.com` in your browser and test it out. You can now switch lists. No tasks are yet displayed. We'll do that in the next chapter.

Dogfooding

```
1  $ gsd ls +gsd -x
2  Active Tasks in list '+gsd'
3  +---+------+-------------------------------+-------+
4  | # | Next | Description                   | Extra |
5  +---+------+-------------------------------+-------+
6  | 1 | YES  | Fill Active Lists using REST  |       |
7  | 2 | YES  | Implement GET /lists          |       |
8  | 3 |      | Finish gsd.buttonAddClick()   |       |
9  | 4 |      | Finish gsd.menuArchiveClick() |       |
10 | 5 |      | Finish gsd.menuCreateClick()  |       |
11 | 6 |      | Finish gsd.menuRenameClick()  |       |
12 +---+------+-------------------------------+-------+
13 $ gsd do 2 +gsd
14 Task 'Implement GET /lists' marked complete.
15 $ gsd do 1 +gsd
16 Task 'Fill Active Lists using REST' marked complete.
17 $ gsd add "Do Open Tasks" +gsd -a
18 Next Action successfully added to gsd
19 $ gsd add "Do Completed Tasks" +gsd -a
20 Next Action successfully added to gsd
```

```
$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+----------------------------------+---------------+
| # | Next | Description                      | Extra         |
+---+------+----------------------------------+---------------+
| 1 | YES  | Do Completed Tasks               |               |
| 2 | YES  | Do Open Tasks                    |               |
| 3 |      | Finish gsd.buttonAddClick()      |               |
| 4 |      | Finish gsd.menuArchiveClick()    |               |
| 5 |      | Finish gsd.menuCreateClick()     |               |
| 6 |      | Finish gsd.menuRenameClick()     |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box            | Done 10/19/13 |
|   | done | Finish Top NavBar                | Done 10/20/13 |
|   | done | Setup Routes                     | Done 10/20/13 |
|   | done | Fill Active Lists using REST     | Done 10/26/13 |
|   | done | Implement GET /lists             | Done 10/26/13 |
+---+------+----------------------------------+---------------+
```

Nice. Our web page is really taking shape.

Chapter 64 - The Tasks

In This Chapter

In this chapter we'll build the Open Tasks and Completed Tasks section of the web page.

We'll do this in a few iterations.

Iteration #1 - Basic structure

Edit `live.blade.php` so that it matches what's below.

```
@extends("layouts.default")

@section("content")

  {{-- Open Tasks --}}
  <div class="panel panel-primary">
    <div class="panel-heading">Open Tasks</div>
    <div class="panel-body">
      <table class="table table-hover">
        <tbody id="open-tasks">
          <tr><td colspan="3">todo</td></tr>
        </tbody>
      </table>
    </div>
  </div>

  {{-- Completed Tasks --}}
  <div class="panel panel-default">
    <div class="panel-heading">Completed Tasks</div>
    <div class="panel-body">
      <table class="table table-hover">
        <tbody id="completed-tasks">
```

```
            <tr><td colspan="3">done</td></tr>
          </tbody>
        </table>
      </div>
    </div>

@stop
```

Pretty simple. The code displaying tasks was taken from the mock up and placed here. The individual tasks were stripped out, but the structure is identical. The two task lists are identified by the `open-tasks` id and the `completed-tasks` id. We'll stuff the rows of the table into these two locations using javascript.

You can refresh `http://gsd.localhost.com` and even though it's not complete, the screen looks pretty.

Edit `gsd.js` and add the three new functions below after the `loadLists()` function.

```
/**
 * Build table row html for complete task
 * @param object task  Task object
 * @param int    index Index of task within currentList.tasks
 * @return string HTML for a table row representing the task
 */
function buildCompletedTask(task, index)
{
  return '<tr><td colspan="3">completed...</td></tr>';
}

/**
 * Build table row html for open task
 * @param object task  Task object
 * @param int    index Index of task within currentList.tasks
 * @return string HTML for a table row representing the task
 */
function buildOpenTask(task, index)
{
  return '<tr><td colspan="3">open...</td></tr>';
}

/**
```

```
24   * Show the Open Tasks and Completed Tasks
25   */
26  function showTasks()
27  {
28    var open = [];
29    var completed = [];
30    for (var i = 0; i < currentList.tasks.length; i++)
31    {
32      var task = currentList.tasks[i];
33      if (task.isCompleted)
34      {
35        completed.push(buildCompletedTask(task, i));
36      }
37      else
38      {
39        open.push(buildOpenTask(task, i));
40      }
41    }
42    if (open.length === 0)
43      open.push('<tr><td colspan="3">No open tasks</td></tr>');
44    if (completed.length === 0)
45      completed.push('<tr><td colspan="3">No completed tasks</td></tr>');
46    $("#open-tasks").html(open.join("\n"));
47    $("#completed-tasks").html(completed.join("\n"));
48  }
```

buildCompletedTask
: This function will return HTML code to represent a single task in the Completed Tasks box. Right now it's just a stub.

buildOpenTask
: Returns HTML code for a task in the Open Tasks box. Just a stub for now.

showTasks
: This function loops through all the tasks in the current list, formats the task depending on the completion status, and stashes the result. At the end it updates the appropriate tables with the HTML.

Now edit the `loadList` method in `gsd.js` to match what's below.

```
  loadList: function(name, archived)
  {
    var url = "/lists/" + name;
    if (archived) url += "?archived=1";
    $.ajax({
      url: url,
      error: function(hdr, status, error)
      {
        gsd.errorMessage("loadList " + status + ' - ' + error);
      },
      success: function(data)
      {
        if (data && data.error)
        {
          gsd.errorMessage("loadList error: " + data.error);
          return;
        }
        currentList = data.list;
        updateNavBar();
        showTasks();

        // Reload the lists
        loadLists(false);
        loadLists(true);
      }
  });
```

The only change is the addition of the `showTasks()` function call. Save everything and reload `http://gsd.localhost.com`. You should be able to switch lists and see where the tasks will display. Even though the detail isn't displaying, you can see how it works.

Iteration #2 - Showing Open Tasks

Now, let's have the open tasks display like they did in the mockup.

Edit the `buildOpenTask()` method in `gsd.js` to match what's below.

```
/**
 * Build table row html for open task
 * @param object task  Task object
 * @param int    index Index of task within currentList.tasks
 * @return string HTML for a table row representing the task
 */
function buildOpenTask(task, index)
{
  var html = [];
  html.push('<tr>');
  html.push('<td>');
  if (task.isNext)
      html.push('<span class="label label-success">next</span>');
  html.push('</td>');
  html.push('<td>');
  html.push($('<div/>').text(task.descript).html());
  if (task.dateDue)
  {
    var d = new Date(task.dateDue);
    html.push(' <span class="label label-primary">');
    html.push('due ' + d.toDateString());
    html.push('</span>');
  }
  html.push('</td>');
  html.push('<td>');
  if ( ! currentList.archived)
  {
    html.push('<a href="javascript:void(0)" onclick="gsd.doDone(' + index);
    html.push(')" class="btn btn-success btn-xs" title="Mark complete">');
    html.push('<span class="glyphicon glyphicon-ok"></span></a>');
    html.push(' <a href="javascript:void(0)" onclick="gsd.doEdit(' + index);
    html.push(')" class="btn btn-info btn-xs" title="Edit task">');
    html.push('<span class="glyphicon glyphicon-pencil"></span></a>');
    html.push(' <a href="javascript:void(0)" onclick="gsd.doMove(' + index);
    html.push(')" class="btn btn-warning btn-xs" title="Move task">');
    html.push('<span class="glyphicon glyphicon-transfer"></span></a>');
    html.push(' <a href="javascript:void(0)" onclick="gsd.doDelete(' + index);
    html.push(')" class="btn btn-danger btn-xs" title="Delete task">');
    html.push('<span class="glyphicon glyphicon-remove-circle"></span></a>');
  }

  html.push('</td>');
```

```
43    html.push('</tr>');
44
45    return html.join('');
46  }
```

This is just a bunch of simple javascript grunting out the work. There are more elegent ways to do this, but since this isn't a book on cool javascript tools, I stuck to the basics.

One trick that is interesting is on line 16. Here we're using jQuery's text() method to assign an unnamed div a value, then we're taking the html and using that to add to our `html` array. This will escape any html in the task description.

Edit the bottom of `gsd.js`, adding the 4 methods after the `loadList` method.

```
1     }, // need the trailing comma at end of loadList now
2
3     /**
4      * Mark a task as completed
5      */
6     doDone: function(index)
7     {
8         var task = currentList.tasks[index].descript;
9         gsd.errorMessage("gsd.doDone() not done " + task);
10    },
11
12    /**
13     * Edit a task
14     */
15    doEdit: function(index)
16    {
17        gsd.errorMessage("gsd.doEdit() not done");
18    },
19
20    /**
21     * Move a task
22     */
23    doMove: function(index)
24    {
25        gsd.errorMessage("gsd.doMove() not done");
26    },
```

```
27
28      /**
29       * Delete a task
30       */
31      doDelete: function(index)
32      {
33          gsd.errorMessage("gsd.doDelete() not done");
34      }
35
36    };
37  })();
```

These are shell methods that each button clicks to. I made `doDone` display the task description just to check it works correctly.

Save your files, reload `http://gsd.localhost.com` and play around with it. All the buttons should function correctly. (Meaning, they display an appropriate error notice.)

Iteration #3 - Showing completed tasks.

Update the `buildCompletedTask()` function in `gsd.js` to match below.

```
1   /**
2    * Build table row html for complete task
3    * @param object task  Task object
4    * @param int    index Index of task within currentList.tasks
5    * @return string HTML for a table row representing the task
6    */
7   function buildCompletedTask(task, index)
8   {
9     var html = [];
10    html.push('<tr>');
11    html.push('<td><span class="label label-default">finished ');
12    var d = new Date(task.dateCompleted);
13    html.push(d.toDateString() + '</span></td><td>');
14    html.push($('<div/>').text(task.descript).html());
15    if (task.dateDue)
16    {
17      d = new Date(task.dateDue);
```

```
18      html.push(' <span class="label label-info">');
19      html.push('due ' + d.toDateString());
20      html.push('</span>');
21    }
22    html.push('</td><td>');
23
24    if ( ! currentList.archived)
25    {
26      html.push('<a href="javascript:void(0)" onclick="gsd.doDone(' + index);
27      html.push(')" class="btn btn-default btn-xs" title="Mark not complete">');
28      html.push('<span class="glyphicon glyphicon-ok"></span></a>');
29      html.push(' <a href="javascript:void(0)" onclick="gsd.doDelete(' + index);
30      html.push(')" class="btn btn-danger btn-xs" title="Delete task">');
31      html.push('<span class="glyphicon glyphicon-remove-circle"></span></a>');
32    }
33
34    html.push('</td>');
35    html.push('</tr>');
36
37    return html.join('');
38  }
```

The code is very similar to `buildOpenTask()`. The one note I have is that we're calling `gsd.doDone()` to mark a task not complete here. This means the `doDone()` function will need to toggle between complete and non-complete.

That's it for this chapter. Let's eat some dog food.

Dogfooding

```
1  $ gsd ls +gsd -x
2  Active Tasks in list '+gsd'
3  +---+------+------------------------------+-------+
4  | # | Next | Description                  | Extra |
5  +---+------+------------------------------+-------+
6  | 1 | YES  | Do Completed Tasks           |       |
7  | 2 | YES  | Do Open Tasks                |       |
8  | 3 |      | Finish gsd.buttonAddClick()  |       |
9  | 4 |      | Finish gsd.menuArchiveClick() |       |
10 | 5 |      | Finish gsd.menuCreateClick()  |       |
11 | 6 |      | Finish gsd.menuRenameClick()  |       |
12 +---+------+------------------------------+-------+
```

```
$ gsd do 2 +gsd
Task 'Do Open Tasks' marked complete.
$ gsd do 1 +gsd
Task 'Do Completed Tasks' marked complete.
$ gsd add "Finish gsd.doDone()" +gsd
Todo successfully added to gsd
$ gsd add "Finish gsd.doEdit()" +gsd
Todo successfully added to gsd
$ gsd add "Finish gsd.doMove()" +gsd
Todo successfully added to gsd
$ gsd add "Finish gsd.doDelete()" +gsd
Todo successfully added to gsd
$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+----------------------------------+---------------+
| # | Next | Description                      | Extra         |
+---+------+----------------------------------+---------------+
| 1 |      | Finish gsd.buttonAddClick()      |               |
| 2 |      | Finish gsd.doDelete()            |               |
| 3 |      | Finish gsd.doDone()              |               |
| 4 |      | Finish gsd.doEdit()              |               |
| 5 |      | Finish gsd.doMove()              |               |
| 6 |      | Finish gsd.menuArchiveClick()    |               |
| 7 |      | Finish gsd.menuCreateClick()     |               |
| 8 |      | Finish gsd.menuRenameClick()     |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box            | Done 10/19/13 |
|   | done | Finish Top NavBar                | Done 10/20/13 |
|   | done | Setup Routes                     | Done 10/20/13 |
|   | done | Do Completed Tasks               | Done 10/26/13 |
|   | done | Do Open Tasks                    | Done 10/26/13 |
|   | done | Fill Active Lists using REST     | Done 10/26/13 |
|   | done | Implement GET /lists             | Done 10/26/13 |
+---+------+----------------------------------+---------------+
```

Marked two tasks finished and added 4 more to do.

Chapter 65 - Deleting a Task

In This Chapter

In this chapter we'll add the functionality to delete a task from a list in the web page application.

Deleting a task is trivial. We'll simply delete it from the loaded list (`currentList`). Then we'll save the list using the `PUT /lists/{name}` AJAX call. Creating that AJAX call will be the bulk of this chapter.

Refactoring TaskInterface

Let's do a quick refactor of the `TaskInterface::setIsComplete()` method. We're going to need to set the date completed, which has automatically been set in the past.

Edit `TaskInterface.php` and change the function definition as below.

```
1  <?php
2  /**
3   * Set whether task is complete.
4   * @param bool  $complete
5   * @param mixed $when If null then uses current date/time, otherwise
6   *                    a Carbon object or date/time string
7   */
8  public function setIsComplete($complete, $when = null);
9  ?>
```

Edit `Task.php` and update the method to implement this change.

```
<?php
/**
 * Set whether task is complete.
 * @param bool  $complete
 * @param mixed $when If non-null then uses current date/time, otherwise
 *                    a Carbon object or date/time string
 */
public function setIsComplete($complete, $when = null)
{
  $this->complete = !! $complete;
  if ($this->complete)
  {
    if ($when == null)
    {
      $when = new Carbon;
    }
    else if (is_string($when))
    {
      $when = new Carbon($when);
    }
    $this->whenCompleted = $when;
  }
  else
  {
    $this->whenCompleted = null;
  }
}
?>
```

Updating the Controller

Edit the top of `ListController.php`, make sure it matches what's below.

```
<?php namespace GSD\Controllers;

use App;
use Carbon\Carbon;
use GSD\Entities\ListInterface;
use Input;
use Response;
use Todo;

?>
```

Here we just added a couple `use` statements.

Update the `update()` method of `ListController.php` to match below.

```
<?php
  /**
   * Update the specified list.
   *
   * @param  string $id The list name
   */
  public function update($id)
  {
    if ($id != Input::get('name'))
    {
      return Response::json(array('error' => 'List id/name mismatch'));
    }

    // Build new list with values
    $list = App::make('GSD\Entities\ListInterface');
    $list->set('id', $id);
    $list->set('title', Input::get('title'));
    $list->set('subtitle', Input::get('subtitle'));
    $list->set('archived', str2bool(Input::get('archived')));

    // Add tasks to list from values passed
    $tasks = Input::get('tasks');
    if ( ! is_array($tasks)) $tasks = array();
    foreach ($tasks as $task)
    {
```

```
26        $newTask = App::make('GSD\Entities\TaskInterface');
27        $descript = $task['descript'];
28        if ($task['dateDue'])
29        {
30          $d = Carbon::createFromTimestamp($task['dateDue'] / 1000);
31          $descript .= ' :due:' . $d->format('Y-m-d');
32        }
33        $newTask->setDescription($descript);
34        if (str2bool($task['isCompleted']))
35        {
36          $newTask->setIsComplete(
37            true,
38            Carbon::createFromTimestamp($task['dateCompleted'] / 1000)
39          );
40        }
41        if (str2bool($task['isNext']))
42        {
43          $newTask->setIsNextAction(true);
44        }
45        $list->taskAdd($newTask);
46      }
47
48      // Save and return success
49      $list->save();
50      return Response::json(array('success' => true));
51    }
52  ?>
```

Lines 9 - 12

A quick check to make sure the id passed in the URL is the same as the name passed in as post data.

Lines 14 - 19

We make a new list and assign it values from the data passed to `update()`. Notice how we're calling `setIsComplete()`? That's why we needed to refactor that method.

Lines 21 - 42

Next we loop through the tasks, building new ones and adding them to the list we're building.

Lines 44 - 46

Finally we save the list, overwriting what was there before and return a non-error response.

Update the `doDelete()` javascript method.

Edit `gsd.js` and update the `doDelete()` method to match what's below.

```
/**
 * Delete a task
 */
doDelete: function(index)
{
  if ( ! confirm("This will permanently destroy the task. Are you sure?"))
  {
    return;
  }

  // Remove the item from currentList
  currentList.tasks.splice(index, 1);

  // And save the list
  saveCurrentList("Task successfully removed.", "doDelete");
}
```

Simple javascript code. There's no saveCurrentList() method yet, so let's create it.

Edit `gsd.js` and add the function below after the `showTasks()` function.

```
/**
 * Save the current list, then reload everything
 * @param string success_msg The message to show on success
 * @param string from        Name of method we're called from, for errors
 */
function saveCurrentList(success_msg, from)
{
  $.ajax({
    url: "/lists/" + currentList.name,
    method: "PUT",
    data: currentList,
```

```
12      error: function(hdr, status, error)
13      {
14        gsd.errorMessage("saveCurrentList " + status + ' - ' +
15          error + ", from " + from);
16      },
17      success: function(data)
18      {
19        if (data && data.error)
20        {
21          gsd.errorMessage("saveCurrentList error: " +
22            data.error + ", from " + from);
23          return;
24        }
25
26        gsd.loadList(currentList.name, currentList.archived);
27        gsd.successMessage(success_msg);
28      }
29    });
30  }
```

Again, this isn't a javascript book, so I'm not going into detail. We're just making a PUT request to the AJAX update method, when it finishes successfully, then we're calling `gsd.loadList()` to reload the data on the page.

Give it a shot. Reload `http://gsd.localhost.com/` in your browser. Delete a few items. You may have to use the command line utility created in the last section to add lists, tasks, etc., in order to see the delete method working.

Toggling the Completed Flag

This chapter didn't end up being as long as I thought it would, so let's implement the `doDone()` method, too.

Update the `doDone()` method in `gsd.js` to what's below.

```
/**
 * Toggle task completion
 */
doDone: function(index)
{
  // Toggle completion status
  if (currentList.tasks[index].isCompleted)
  {
    currentList.tasks[index].isCompleted = false;
  }
  else
  {
    var d = new Date();
    currentList.tasks[index].isCompleted = true;
    currentList.tasks[index].dateCompleted = d.valueOf();
  }
  saveCurrentList("Task completion updated.", "doDone");
},
```

That was easy.

Dogfooding

```
$ gsd ls +gsd -x
Active Tasks in list '+gsd'
+---+------+-------------------------------+-------+
| # | Next | Description                   | Extra |
+---+------+-------------------------------+-------+
| 1 |      | Finish gsd.buttonAddClick()   |       |
| 2 |      | Finish gsd.doDelete()         |       |
| 3 |      | Finish gsd.doDone()           |       |
| 4 |      | Finish gsd.doEdit()           |       |
| 5 |      | Finish gsd.doMove()           |       |
| 6 |      | Finish gsd.menuArchiveClick() |       |
| 7 |      | Finish gsd.menuCreateClick()  |       |
| 8 |      | Finish gsd.menuRenameClick()  |       |
+---+------+-------------------------------+-------+
$ gsd do 3 +gsd
Task 'Finish gsd.doDone()' marked complete.
$ gsd do 2 +gsd
Task 'Finish gsd.doDelete()' marked complete.
$ gsd ls +gsd
```

```
All Tasks in list '+gsd'
+---+------+----------------------------------+---------------+
| # | Next | Description                      | Extra         |
+---+------+----------------------------------+---------------+
| 1 |      | Finish gsd.buttonAddClick()      |               |
| 2 |      | Finish gsd.doEdit()              |               |
| 3 |      | Finish gsd.doMove()              |               |
| 4 |      | Finish gsd.menuArchiveClick()    |               |
| 5 |      | Finish gsd.menuCreateClick()     |               |
| 6 |      | Finish gsd.menuRenameClick()     |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box            | Done 10/19/13 |
|   | done | Finish Top NavBar                | Done 10/20/13 |
|   | done | Setup Routes                     | Done 10/20/13 |
|   | done | Do Completed Tasks               | Done 10/26/13 |
|   | done | Do Open Tasks                    | Done 10/26/13 |
|   | done | Fill Active Lists using REST     | Done 10/26/13 |
|   | done | Finish gsd.doDelete()            | Done 10/26/13 |
|   | done | Finish gsd.doDone()              | Done 10/26/13 |
|   | done | Implement GET /lists             | Done 10/26/13 |
+---+------+----------------------------------+---------------+
```

Nice, we knocked a couple things off the list and didn't add anything new.

Chapter 66 - Adding and Editing Tasks

In This Chapter

In this chapter we'll add the functionality to create new tasks and edit existing tasks.

The Modal Task Form

The first thing we need is a modal form we can use to add new tasks, or edit existing tasks.

Edit `views/layouts/default.blade.php` and change the bottom of the file as specified below.

```
1     @include('partials.taskmodal')
2   </body>
3 </html>
```

We'll build the modal form in its own view. So line #1 here is a new line telling Blade to include that partial we'll create.

Create `taskmodal.blade.php` in `views/partials` with the following content.

```
1  <div class="modal fade" id="taskbox">
2    <div class="modal-dialog">
3      <div class="modal-content">
4        <div class="modal-header">
5          <button type="button" class="close" data-dismiss="modal">
6            &times;
7          </button>
8          <h4 class="modal-title" id="taskbox-title">title</h4>
9        </div>
10       <div class="modal-body">
```

```
11          <form class="form-horizontal">
12            <input type="hidden" id="task-index" value="-1">
13            <div class="form-group">
14              <div class="col-lg-offset-2 col-lg-10">
15                <div class="checkbox">
16                  <label>
17                    <input type="checkbox" id="task-next"> Next Action
18                  </label>
19                </div>
20              </div>
21            </div>
22            <div class="form-group">
23              <label class="col-lg-2 control-label" for="task-descript">
24                Description
25              </label>
26              <div class="col-lg-10">
27                <input type="text" class="form-control" id="task-descript">
28              </div>
29            </div>
30            <div class="form-group">
31              <label class="col-lg-2 control-label" for="task-due">
32                Due
33              </label>
34              <div class="col-lg-10">
35                <input type="text" class="form-control" id="task-due">
36              </div>
37            </div>
38          </form>
39        </div>
40        <div class="modal-footer">
41          <button type="button" class="btn btn-default" data-dismiss="modal">
42            Close
43          </button>
44          <button type="button" class="btn btn-primary"
45            onclick="gsd.taskboxSave()">Save</button>
46        </div>
47      </div>
48    </div>
49  </div>
```

This is straight, bootstrap flavored HTML. Just a few notes:

- The `taskbox` id will let us access this modal.

- The hidden `task-index` attribute will specify the task's index. For Add Task this will always be -1.
- The `taskbox-title` id will let us change the title depending if this is an Add Task or Edit task function.
- The data elements have their own id for accessing via javascript.
- The **Save** button calls `gsd.taskboxSave()`, which doesn't exist yet.

The Javascript

Add the following `taskboxShow()` function to `gsd.js`. Put it after the `menuCreateClick()` function. (This way, the function appears before any functions that call it. Maybe this isn't required, but this is a habit I started in the early days of Javascript.)

```
1  /**
2   * Display the task modal box
3   * @param string  title Title of the modal box
4   * @param integer index Task index, -1 for new task
5   */
6  function taskboxShow(title, index)
7  {
8    var task = (index == -1) ? {} : currentList.tasks[index];
9    $("#task-index").val(index);
10   $("#taskbox-title").text(title);
11   $("#task-next").prop("checked", (task.isNext === true));
12   $("#task-descript").val(task.descript);
13   if (task.dateDue)
14   {
15     var d = new Date(task.dateDue);
16     $("#task-due").val(d.toDateString());
17   }
18   else
19   {
20     $("#task-due").val("");
21   }
22   $("#taskbox")
23     .modal("show")
24     .on("shown.bs.modal", function()
25     {
26       $("#task-descript").focus().select();
```

```
27      });
28  }
```

Line 8

We assign the `task` variable either an empty object (if we're adding a task) or the task from the current list.

Lines 9 - 21

Assigning various values within the modal task form.

Lines 22 - 27

Here we use the bootstrap function to show the modal box. Then, when the modal is displayed we set the input focus to the description field.

Modify the `buttonAddClick()` function in `gsd.js`

```
1  /**
2   * Handle click on the add task button
3   */
4  function buttonAddClick()
5  {
6    taskboxShow("Add New Task", -1);
7    return false;
8  }
```

Then modify the `doEdit` method in `gsd.js`.

```
1  /**
2   * Edit a task
3   */
4  doEdit: function(index)
5  {
6    taskboxShow("Edit Task", index);
7  },
```

And, to be complete, add a new public method to `gsd.js`, the `taskboxSave()` method.

```
  },  // remember the trailing comma of the last function

  /**
   * Handle adding new tasks or updating existin tasks
   */
  taskboxSave: function()
  {
    $("#taskbox").modal("hide");
    gsd.errorMessage("gsd.taskboxSave not done");
  }
```

If you save everything and reload your browser you can play around with it a bit. Everything should work except when you hit save. Let's fix that.

Finishing taskboxSave

Edit `gsd.js` and update the `taskboxSave()` function to the code below.

```
/**
 * Handle adding new tasks or updating existing tasks
 */
taskboxSave: function()
{
  var index = parseInt($("#task-index").val());
  var dueDate = $("#task-due").val();
  var task = {
    isNext: $("#task-next").prop("checked"),
    isCompleted: false,
    dateCompleted: null,
    descript: $("#task-descript").val()
  };
  if (dueDate === "")
  {
    dueDate = null;
  }
  else
  {
    try  {
```

```
21        dueDate = Date.parse(dueDate);
22      } catch (err)  {
23        dueDate = null;
24      }
25      if (isNaN(dueDate))
26        dueDate = null;
27    }
28    task.dateDue = dueDate;
29    if (index < 0)
30    {
31      currentList.tasks.push(task);
32    }
33    else
34    {
35      currentList.tasks[index] = task;
36    }
37
38    $("#taskbox").modal("hide");
39    saveCurrentList("Task successfully saved.", "taskboxSave");
40  }
```

Lines 5 - 13
Read the values in, build a task object.

Lines 14 - 28
Handle the due date, parsing it, watching for errors, and finally assigning it to the task object.

Line 29 - 36
Either append the task to the current list or replace the existing task in the current list.

Line 38 and 39
Hide the modal and save the current list.

As always, play with it. See how it's working.

Dogfooding

```
$ gsd ls +gsd -x
Active Tasks in list '+gsd'
+---+------+------------------------------+-------+
| # | Next | Description                  | Extra |
+---+------+------------------------------+-------+
| 1 |      | Finish gsd.buttonAddClick()  |       |
| 2 |      | Finish gsd.doEdit()          |       |
| 3 |      | Finish gsd.doMove()          |       |
| 4 |      | Finish gsd.menuArchiveClick() |       |
| 5 |      | Finish gsd.menuCreateClick()  |       |
| 6 |      | Finish gsd.menuRenameClick()  |       |
+---+------+------------------------------+-------+
$ gsd do 2 +gsd
Task 'Finish gsd.doEdit()' marked complete.
$ gsd do 1 +gsd
Task 'Finish gsd.buttonAddClick()' marked complete.
$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------+---------------+
| # | Next | Description                     | Extra         |
+---+------+---------------------------------+---------------+
| 1 |      | Finish gsd.doMove()             |               |
| 2 |      | Finish gsd.menuArchiveClick()   |               |
| 3 |      | Finish gsd.menuCreateClick()    |               |
| 4 |      | Finish gsd.menuRenameClick()    |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box           | Done 10/19/13 |
|   | done | Finish Top NavBar               | Done 10/20/13 |
|   | done | Setup Routes                    | Done 10/20/13 |
|   | done | Do Completed Tasks              | Done 10/26/13 |
|   | done | Do Open Tasks                   | Done 10/26/13 |
|   | done | Fill Active Lists using REST    | Done 10/26/13 |
|   | done | Finish gsd.buttonAddClick()     | Done 10/26/13 |
|   | done | Finish gsd.doDelete()           | Done 10/26/13 |
|   | done | Finish gsd.doDone()             | Done 10/26/13 |
|   | done | Finish gsd.doEdit()             | Done 10/26/13 |
|   | done | Implement GET /lists            | Done 10/26/13 |
+---+------+---------------------------------+---------------+
```

Only 4 things left to do. And here's a little teaser. When we get to the `gsd.doMove()` function, there's going to be a surprise.

Chapter 67 - Archiving and Unarchiving Lists

In This Chapter

In this chapter we'll add the ability to archive and unarchive lists in the web application.

Implementing the AJAX archive method

Edit `ListController.php`, have the top of the file match what's below.

```
1  <?php namespace GSD\Controllers;
2
3  use App;
4  use Carbon\Carbon;
5  use Config;
6  use GSD\Entities\ListInterface;
7  use GSD\Repositories\TodoRepositoryInterface;
8  use Input;
9  use Response;
10 use Todo;
11
12 class ListController extends \Controller {
13
14   protected $repository;
15
16   /**
17    * Constructor
18    */
19   public function __construct(TodoRepositoryInterface $repository)
20   {
21     $this->repository = $repository;
22   }
23 ?>
```

We changed two things here:

1. Additional `use` statements.
2. Added the `__construct()` to inject our repository.

Edit the `archive()` method of `ListController.php` to match the following.

```
<?php
/**
 * Archive the specified list
 *
 * @param  string $id The list name
 */
public function archive($id)
{
  try
  {
    // Throws error if list doesn't exist
    $list = Todo::get($id);

    // Can't archive default list
    if ($id == Config::get('todo.defaultList'))
    {
      throw new \RuntimeException("Cannot archive default list");
    }

    // Throw error if archived list exists
    if ($this->repository->exists($id, true))
    {
      throw new \RuntimeException(
        "Archive list '$id' exists. Try renaming first."
      );
    }
  }
  catch (\Exception $e)
  {
    return Response::json(array('error' => $e->getMessage()));
  }

```

```
33    $list->archive();
34
35    return Response::json(array('success' => true));
36  }
37  ?>
```

The code above should be easy to follow.

We're running the code in the try/catch block and throwing exceptions if something isn't correct.

Calling the AJAX archive() method

Edit the `menuArchiveClick()` method in `gsd.js` to match the following.

```
1   /**
2    * Handle click on the top nav menu archive/unarchive option
3    */
4   function menuArchiveClick()
5   {
6     var url = "/lists/" + currentList.name;
7     if (currentList.archived)
8     {
9       gsd.errorMessage('not implemented');
10      return false;
11    }
12
13    // The archive version
14    url += '/archive';
15    $.ajax({
16      url: url,
17      method: "POST",
18      error: function(hdr, status, error)
19      {
20        gsd.errorMessage("menuArchiveClick " + status + ' - ' + error);
21      },
22      success: function(data)
23      {
24        if (data && data.error)
25        {
```

```
        gsd.errorMessage(data.error);
        return;
      }
      gsd.loadList(currentList.name, true);
      gsd.successMessage("List successfully archived.");
    }
  });
  return false;
}
```

That's it. The Archive List method is implemented!

Implementing the AJAX unarchive method

Edit the `unarchive()` method of `ListController.php` to match the code below.

```
<?php
  /**
   * Unarchive the specified list
   *
   * @param  string $id The list name
   */
  public function unarchive($id)
  {
    try
    {
      // Throws error if list doesn't exist
      $list = Todo::get($id, true);

      // Throw error if active list exists
      if ($this->repository->exists($id, false))
      {
        throw new \RuntimeException(
          "Active list '$id' exists. Try renaming first."
        );
      }

      // Save as unarchived
```

```
23        $list->set('archived', false);
24        $list->save();
25
26        // Delete existing archived list
27        if ( ! $this->repository->delete($id, true))
28        {
29          throw new \RuntimeException(
30            'ERROR deleting archived version.'
31          );
32        }
33      }
34      catch (\Exception $e)
35      {
36        return Response::json(array('error' => $e->getMessage()));
37      }
38
39      return Response::json(array('success' => true));
40    }
41  ?>
```

This code is very similar to the `archive()` method we just updated, but the list is manually deleted using the repository.

Calling the AJAX unarchive() method

Edit the `menuArchiveClick()` method in `gsd.js`. The top of the method should match what's below.

```
1  /**
2   * Handle click on the top nav menu archive/unarchive option
3   */
4  function menuArchiveClick()
5  {
6    var url = "/lists/" + currentList.name;
7    if (currentList.archived)
8    {
9      url += '/unarchive';
10     $.ajax({
11       url: url,
```

```
12        method: "POST",
13        error: function(hdr, status, error)
14        {
15          gsd.errorMessage("menuArchiveClick " + status + ' - ' + error);
16        },
17        success: function(data)
18        {
19          if (data && data.error)
20          {
21            gsd.errorMessage(data.error);
22            return;
23          }
24          gsd.loadList(currentList.name, false);
25          gsd.successMessage("List successfully unarchived.");
26        }
27      });
28
29      return false;
30    }
31
32    // Rest of method is the same
```

This, too, is very similar to how the AJAX archive call is implemented. No need to repeat the explanation.

Time to Refactor?

Actually, I would normally refactor the javascript at this point. There's a bit of duplication with how the AJAX calls are made. It'd be nice to *Keep it DRY*. But since there's only a couple chapters left, I'll leave it to you to refactor.

Try archiving and unarchiving lists. Works good for me.

Dogfooding

```
$ gsd ls +gsd -x
Active Tasks in list '+gsd'
+---+------+------------------------------+-------+
| # | Next | Description                  | Extra |
+---+------+------------------------------+-------+
| 1 |      | Finish gsd.doMove()          |       |
| 2 |      | Finish gsd.menuArchiveClick() |       |
| 3 |      | Finish gsd.menuCreateClick()  |       |
| 4 |      | Finish gsd.menuRenameClick()  |       |
+---+------+------------------------------+-------+
$ gsd do 2 +gsd
Task 'Finish gsd.menuArchiveClick()' marked complete.
$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------+---------------+
| # | Next | Description                     | Extra         |
+---+------+---------------------------------+---------------+
| 1 |      | Finish gsd.doMove()             |               |
| 2 |      | Finish gsd.menuCreateClick()    |               |
| 3 |      | Finish gsd.menuRenameClick()    |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box           | Done 10/19/13 |
|   | done | Finish Top NavBar               | Done 10/20/13 |
|   | done | Setup Routes                    | Done 10/20/13 |
|   | done | Do Completed Tasks              | Done 10/26/13 |
|   | done | Do Open Tasks                   | Done 10/26/13 |
|   | done | Fill Active Lists using REST    | Done 10/26/13 |
|   | done | Finish gsd.buttonAddClick()     | Done 10/26/13 |
|   | done | Finish gsd.doDelete()           | Done 10/26/13 |
|   | done | Finish gsd.doDone()             | Done 10/26/13 |
|   | done | Finish gsd.doEdit()             | Done 10/26/13 |
|   | done | Finish gsd.menuArchiveClick()   | Done 10/26/13 |
|   | done | Implement GET /lists            | Done 10/26/13 |
+---+------+---------------------------------+---------------+
```

Only three things left to do.

Chapter 68 - Creating and Renaming Lists

In This Chapter

In this chapter we'll add the ability to create new lists and rename existing lists.

Adding List Modal

Just like we did with the Task Editing, let's create a new modal box for Creating new lists.

Edit `default.blade.php` in `app/views/layouts`. Below is just a snippet of the bottom of the file.

```
    </div>

    @include('partials.taskmodal')
    @include('partials.listmodal')
  </body>
</html>
```

We added the `@include` line right before the closing body tag.

Create `listmodal.blade.php` in the `app/views/partials` directory.

```
<div class="modal fade" id="listbox">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal">
          &times;
        </button>
        <h4 class="modal-title" id="listbox-title">title</h4>
      </div>
      <div class="modal-body">
        <form class="form-horizontal">
          <div class="form-group">
            <label class="col-lg-3 control-label" for="list-id">
              List Name
            </label>
            <div class="col-lg-9">
              <input type="text" class="form-control" id="list-id">
            </div>
          </div>
          <div class="form-group">
            <label class="col-lg-3 control-label" for="list-title">
              List Title
            </label>
            <div class="col-lg-9">
              <input type="text" class="form-control" id="list-title">
            </div>
          </div>
          <div class="form-group">
            <label class="col-lg-3 control-label" for="list-subtitle">
              List Subtitle
            </label>
            <div class="col-lg-9">
              <input type="text" class="form-control" id="list-subtitle">
            </div>
          </div>
        </form>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-default" data-dismiss="modal">
          Close
        </button>
        <button type="button" class="btn btn-primary"
```

```
          onclick="gsd.listboxSave()">Save</button>
      </div>
    </div>
  </div>
</div>
```

The HTML above is similar to what we did with `taskmodal.blade.php`.

Adding Create List Javascript

Now we'll create the javascript to use the modal we just created.

Edit `gsd.js` and update the `menuCreateClick()` function to match what's below.

```
/**
 * Handle click on the top nav menu create option
 */
function menuCreateClick()
{
  $(".dropdown-menu").dropdown("toggle");
  $("#listbox-title").html("Create New List");
  $("#list-id").val("");
  $("#list-title").val("");
  $("#list-subtitle").val("");
  $("#listbox")
    .modal("show")
    .on("shown.bs.modal", function()
    {
      $("#list-id").focus().select();
    });
  return false;
}
```

The code clears the dropdown menu, sets up the listbox modal, and shows it.

Edit `gsd.js` and create a new `listboxSave()` method after `taskboxSave()`.

```
  }, // as always, the trailing comma of the previous function

  /**
   * Handle creating new list
   */
  listboxSave: function()
  {
    var data = {
      name: $("#list-id").val(),
      title: $("#list-title").val(),
      subtitle: $("#list-subtitle").val()
    };

    $.ajax({
      url: "/lists",
      method: "POST",
      data: data,
      error: function(hdr, status, error)
      {
        gsd.errorMessage("listboxSave " + status + ' - ' + error);
        $("#listbox").modal("hide");
      },
      success: function(data)
      {
        $("#listbox").modal("hide");
        if (data && data.error)
        {
          gsd.errorMessage(data.error);
          return;
        }
        gsd.loadList(data.name, false);
        gsd.successMessage("List successfully created.");
      }
    });
  }
```

This is almost identical to the `taskboxSave()` method. *(I had to slap myself in the face to keep from refactoring.)*

Implenting AJAX store call

The last thing we need to make creating a new list work is the `store()` method on the server side fully implemented.

Update `store()` in `ListController.php` to match what's below.

```
<?php
  /**
   * Create a new list
   */
  public function store()
  {
    try
    {
      $name = strtolower(Input::get("name"));
      $title = Input::get("title");
      if (!$title) $title = ucfirst($name);
      $subtitle = Input::get("subtitle");

      if (empty($name))
        throw new \RuntimeException("List Name $name is required");
      if ($this->repository->exists($name, false))
        throw new \RuntimeException("List '$name' already exists");

      $list = Todo::makeList($name, $title);
      if ($subtitle)
      {
        $list->set('subtitle', $subtitle)->save();
      }

      $result = array(
        'success' => true,
        'name' => $name,
      );
    }
    catch (\Exception $e)
    {
      $result = array(
        'error' => $e->getMessage()
```

```
34        );
35      }
36
37      return Response::json($result);
38    }
39  ?>
```

All done! Reload `http://gsd.localhost.com` in your browser and give it a shot. Try creating a list without a name. Try creating one with a name that already exists. Everything should work great.

You may notice I've started going light on the line-by-line explanation of what the code does. That's because we've been over most everything at least a couple times at this point. I'll continue pointing out new things, though.

Implementing Rename Javascript

Now let's add the javascript to implement the Rename List function.

Edit the `menuRenameClick()` function in `gsd.js` to match the code below.

```
1   /**
2    * Handle click on the top nav menu rename option
3    */
4   function menuRenameClick()
5   {
6     $(".dropdown-menu").dropdown("toggle");
7     var dest = prompt("New name for list '" + currentList.name + "'?");
8     if (!dest)
9     {
10      gsd.errorMessage("Rename canceled");
11      return false;
12    }
13    var url = '/lists/' + currentList.name + '/rename/' + dest;
14    if (currentList.archived) url += '?archived=1';
15    $.ajax({
16      url: url,
17      method: "POST",
18      error: function(hdr, status, error)
```

```
    {
      gsd.errorMessage("menuRenameClick " + status + ' - ' + error);
    },
    success: function(data)
    {
      if (data && data.error)
      {
        gsd.errorMessage(data.error);
        return;
      }
      gsd.loadList(dest, currentList.archived);
      gsd.successMessage("Rename successful.");
    }
  });

  return false;
}
```

Simple. We use the javascript `prompt()` function to ask the user for the new list name. Then we make a POST call to the AJAX method to perform the rename.

Implementing AJAX rename call

Edit the `rename()` method in `ListController.php` to match below.

```
<?php
  /**
   * Rename $source list to $dest
   *
   * @param  string $source The source list name
   * @param  string $dest   The destination list name
   */
  public function rename($source, $dest)
  {
    $archived = !! Input::get('archived');
    $source = trim($source);
    $dest = trim($dest);
    try
```

```
14      {
15        if (empty($source))
16          throw new \RuntimeException("Source list name is required");
17        if (empty($dest))
18          throw new \RuntimeException("Destination list name required");
19        if ($source == Config::get('todo.defaultList') && ! $archived)
20          throw new \RuntimeException("Cannot rename default list");
21        if ($this->repository->exists($dest, $archived))
22          throw new \RuntimeException("Destination list exists");
23
24        // Load existing list, save with new name, then delete old one
25        $list = Todo::get($source, $archived);
26        $newList = clone $list;
27        $newList->set('id', $dest);
28        $newList->save();
29        $list->delete();
30
31        $return = array('success' => true);
32      }
33      catch (\Exception $e)
34      {
35        $return = array('error' => $e->getMessage());
36      }
37
38      return Response::json($return);
39    }
40  ?>
```

This code was easy to implement. After a few checks in the `try/catch` block, the code to do the rename is almost identical to the code in `RenameListCommand.php`

Test it out. See how it works.

Dogfooding

```
$ gsd ls +gsd -x
Active Tasks in list '+gsd'
+---+------+-----------------------------+-------+
| # | Next | Description                 | Extra |
+---+------+-----------------------------+-------+
| 1 |      | Finish gsd.doMove()         |       |
| 2 |      | Finish gsd.menuCreateClick() |       |
| 3 |      | Finish gsd.menuRenameClick() |       |
+---+------+-----------------------------+-------+
$ gsd do 3 +gsd
Task 'Finish gsd.menuRenameClick()' marked complete.
$ gsd do 2 +gsd
Task 'Finish gsd.menuCreateClick()' marked complete.
$ gsd ls +gsd
All Tasks in list '+gsd'
+---+------+---------------------------------+---------------+
| # | Next | Description                     | Extra         |
+---+------+---------------------------------+---------------+
| 1 |      | Finish gsd.doMove()             |               |
|   | done | Create web application wireframe | Done 10/13/13 |
|   | done | Add error/success box           | Done 10/19/13 |
|   | done | Finish Top NavBar               | Done 10/20/13 |
|   | done | Setup Routes                    | Done 10/20/13 |
|   | done | Do Completed Tasks              | Done 10/26/13 |
|   | done | Do Open Tasks                   | Done 10/26/13 |
|   | done | Fill Active Lists using REST    | Done 10/26/13 |
|   | done | Finish gsd.buttonAddClick()     | Done 10/26/13 |
|   | done | Finish gsd.doDelete()           | Done 10/26/13 |
|   | done | Finish gsd.doDone()             | Done 10/26/13 |
|   | done | Finish gsd.doEdit()             | Done 10/26/13 |
|   | done | Finish gsd.menuArchiveClick()   | Done 10/26/13 |
|   | done | Implement GET /lists            | Done 10/26/13 |
|   | done | Finish gsd.menuCreateClick()    | Done 10/27/13 |
|   | done | Finish gsd.menuRenameClick()    | Done 10/27/13 |
+---+------+---------------------------------+---------------+
```

One item left to do. Yeah! That works out because there's one chapter left in the book.

Chapter 69 - Move and Beyond

In This Chapter

In this chapter we'll discuss the move task command and other places you can take the web application.

The Move Task Command

The last item on our todo list (gsd ls +gsd) is to finish the `gsd.doMove()` function. Here's the surprise:

I'm leaving it for you to finish

The logic is straightforward. Here's one approach to implementing it.

Routes

- Set up a new route for the move command in routes.php.
- Suggest route: `lists/{source}/{name}/move/{dest}`
- Set up a skeleton destination method for move in ListController.php

Javascript - gsd.js

- Prompt the user for the destination list
- Make the AJAX call to `lists/{source}/{name}/move/{dest}`
- Upon success load the destination list.

PHP - ListController.php

- Use a try/catch block like we've been doing with the other methods
- Follow a similar logic to `MoveTaskCommand.php`

See. It's really pretty simple. You could just use the javascript `prompt()` method to get the destination list. Or you could get ambitious. Present the user with a popup menu for the destination. Your call.

Where to go next

Even though this book is done, there's no reason to quit development of your personal version of the web application.

Try to think how it could better work for you.

Here's a list of suggestions. You could think of dozens of other ways to improve it.

- Add some basic authentication so nobody but you can see the web page.
- Implement a "delete list" function. (Yes, the destroy() method in ListController was never used.)
- Add contexts like @home or @work to your tasks.
- Refactor javascript to remove duplication in AJAX calls.
- Refactor AJAX methods to return `list` instead of `success` to eliminate a followup call to load the list.
- Implement backups through the web interface. Maybe a "utility" menu option?
- Track priorities on tasks.
- Use better date formatting.
- Add activity logs to track changes to all your todo lists.
- Add a date picker for the date due field.
- Expand due dates to include time.
- Add time tracking. Maybe a whole new section of your list has start/stop tracking times.
- Add sub tasks.
- Add additional task states. Instead of just complete or not complete, maybe an "in-progress" or a "deploying".
- Add people tracking on tasks. %Chuck, %Bob, or %Sally.
- Use a Database instead of text files to store your lists.

This list could go on and on.

I really hope you make this application something you can use. And if you expand the application in some interesting way, please shoot me an email. I'd love to hear from you.

A Final Thank You

I sincerely hope you enjoyed this journey. We spent a lot of time on design and philosophies and almost as much time building the console application. The web application went pretty quickly, but I think it was because of the time spent in the other areas.

This book covered a lot of information about Laravel, but like I said way back at the beginning, it barely scratches the surface of what you can do with this framework. There's so much to learn … so much more to do.

I hesitate to compile an exhaustive list of resources for further Laravel learning because they'll be obsolete soon after this book is in print. I mentioned a few at the beginning of this book, but new ones pop up all the time. Your best bet is to check out laravel.com[34]. In addition to the documentation there is a link to the forums at the top of the page. There's a whole forum devoted to Laravel Resources.

Again, thank you.

Keep Coding!

— Chuck Heintzelman

October 27, 2013

[34] http://laravel.com

Appendices

Or is it *Appendixes*? Or maybe like the plural form of octopus, the word should be *Appendi*? I can never remember.

Appendix I - Composer

Composer[35] is *the* dependency manager for PHP. A dependency manager is different that a package manager. Composer uses packages, yes, but the magic of composer is that it lets you define what packages your project depends on. Then Composer downloads those packages, stuffs them in your project's `vendor` directory, and you're golden.

PHP has needed something like Composer for quite a while. Yeah, there's PEAR[36] and there's lots of great packages in PEAR. But PEAR installs packages globally on your machine.

Composer keeps packages bundled with your project. You can have different versions of the same package within different projects on the same machine!

Very nice.

I hesitate to mention this because I don't want to put down PEAR, because for years that's all us PHP programmers had. And it did a great job. The only problem is … well … sometimes some of the PEAR code seemed a little trashy.

(Wow! Now I really feel like a programming snob. I still like you PEAR, I do. But some of your kids, well, I don't like dealing with them.)

Installing Composer on Unix

It takes two lines of code:

```
$ curl -sS https://getcomposer.org/installer | php
$ mv composer.phar /usr/local/bin/composer
```

If the second line fails, put a `sudo` before the `mv` command.

You can test if it's installed cool by checking the version.

```
$ composer --version
Composer version 815f7687c5d58af2b31df680d2a715f7eb8dbf62
```

[35] http://getcomposer.org

[36] http://pear.php.net/

Appendix II - PHP Unit

PHP Unit[37] is the standard in PHP Unit testing.

I could go on about the benefits of unit testing, why you want to do it, bla bla bla, but I won't. Let's just get it installed.

Installing Composer on Unix

It takes three lines of code:

```
$ wget https://phar.phpunit.de/phpunit.phar
$ chmod +x phpunit.phar
$ mv phpunit.phar /usr/local/bin/phpunit
```

If the last line fails, put a `sudo` before the `mv` command.

You can test if it's installed cool by checking the version.

```
$ phpunit --version
PHPUnit 3.7.26 by Sebastian Bergmann.
```

Wasn't that easy. Hardly seems worth taking a whole appendix up to do it.

[37] http://phpunit.de/manual/current/en/index.html

Appendix III - Apache Setup

Apache is ubiquitous these days. Not only on the web, but in the various operating systems people use for their desktop (or laptop). Unfortunately, there's so many versions, different installation locations, and different ways Apache can be set up, that there's no absolute *do this and it will work* rule for configuration.

That said, I'll present how my machine is configured and provide notes for common alternatives but in the end always consult the Apache Documentation[38] as the authoritative source.

Installing Apache

If apache needs to be installed on your machine, I suggest using whatever package manager comes with your operating system.

With Ubuntu and Linux Mint, this is `apt-get`.

```
$ sudo apt-get install apache2 libapache2-mod-php5
[sudo] password for chuck:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  apache2-mpm-prefork apache2-utils apache2.2-bin apache2.2-common libapr1
  libaprutil1 libaprutil1-dbd-sqlite3 libaprutil1-ldap
Suggested packages:
  apache2-doc apache2-suexec apache2-suexec-custom php-pear
The following NEW packages will be installed:
  apache2 apache2-mpm-prefork apache2-utils apache2.2-bin apache2.2-common
  libapache2-mod-php5 libapr1 libaprutil1 libaprutil1-dbd-sqlite3
  libaprutil1-ldap
0 upgraded, 10 newly installed, 0 to remove and 2 not upgraded.
Need to get 6,379 kB of archives.
After this operation, 20.0 MB of additional disk space will be used.
Do you want to continue [Y/n]?

... lots of stuff scrolls by ...
```

[38] http://httpd.apache.org/docs/

```
21
22 Processing triggers for libc-bin ...
23 ldconfig deferred processing now taking place
24 $
```

To test this is working, point your browser to http://localhost. You should get a **It works!** page or something similar.

For Windows

For windows I set up WampServer[39]. This gives you Apache, PHP, and MySQL. The nice thing about WampServer is that it let's you run multiple versions so you could run PHP 5.4 and PHP 5.3 side-by-side.

Fixing Permissions

To change the User and Group apache uses, edit `/etc/apache2/envvars` and change it to what you need.

```
1 # Using VIM
2 $ sudo vim /etc/apache2/envvars
3
4 # Using Nano
5 $ sudo nano /etc/apache2/envvars
6
7 # Using Sublime Text 2
8 $ sudo subl /etc/apache2/envvars
```

Edit the `APACHE_RUN_USER` and `APACHE_RUN_GROUP` settings to what you need, I'm using *chuck* below.

```
1 export APACHE_RUN_USER=chuck
2 export APACHE_RUN_GROUP=chuck
```

Using Named Virtual Hosts

By default Ubuntu/Mint now installs apache with `NameVirtualHost` enabled. You shouldn't have to do anything. This is configured in /etc/apache2/ports.conf

[39] http://www.wampserver.com/en/

```
NameVirtualHost *:80
List 80
```

These two lines tell apache to listen on port 80 (the standard web port) and allow virtual hosts to be set up based on the hostname.

You may need to search through all your apache configuration files (all those files in the apache folder ending with .conf) and make sure those two lines exist.

Adding an entry to /etc/hosts

If you don't have a domain name registered, you can "fake" one by editing your hosts file.

You'll need to edit this file as `root` using an editor.

```
# Using VIM
$ sudo vim /etc/hosts

# Using Nano
$ sudo nano /etc/hosts

# Using Sublime Text 2
$ sudo subl /etc/hosts
```

Add the entry. In this case we'll say 'myhost.localhost.com'. I personally use the 'name.localhost.com', but have seen others use 'name.dev', or some other variation.

```
# somewhere in the file
myhost.localhost.com   127.0.0.1
```

Setup up a VirtualHost on UbuntuMint

You'll need to create this file as `root` using an editor.

```
# Using VIM
$ sudo vim /etc/apache2/sites-available/filename

# Using Nano
$ sudo nano /etc/apache2/sites-available/filename

# Using Sublime Text 2
$ sudo subl /etc/apache2/sites-available/filename
```

Edit the filename (for example, I use `gsd` for the project in this book) to match what's below. Change the paths and hostname as appropriate.

```
<VirtualHost *:80>
  DocumentRoot "/laravel/project/path/public"
  ServerName myhost.localhost.com
  <Directory "/laravel/project/path/public">
    Options Indexes FollowSymLinks MultiViews
    AllowOverride all
  </Directory>
</VirtualHost>
```

Create a symbolic link to this configuration, enable mod-rewrite and restart apache

```
$ cd /etc/apache2/sites-enabled
$ sudo ln -s /etc/apache2/sites-available/filename
$ sudo a2enmod rewrite
$ sudo service apache restart
```

If there are no errors then your virtual host should be set up cool. You can test it by pointing your browser to it.

Appendix IV - Nginx Setup

Nginx (pronounced engine-x) is rapidly becoming the favorite web server for techies everywhere. It's fast, easy to configure, and doesn't use as much memory as Apache.

I actually prefer Nginx to Apache, but tend to use Apache since that's how the production servers at my work are configured.

Installing Nginx

Use whatever package manager your operating system provides. With Ubuntu/Mint Linux this is `apt-get`

```
$ sudo apt-get install nginx php5-fpm
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  nginx-common nginx-full
Suggested packages:
  php-pear
The following NEW packages will be installed:
  nginx nginx-common nginx-full php5-fpm
0 upgraded, 4 newly installed, 0 to remove and 2 not upgraded.
Need to get 3,068 kB of archives.
After this operation, 9,626 kB of additional disk space will be used.
Do you want to continue [Y/n]?

... lots of stuff scrolls by ...

Processing triggers for ureadahead ...
Setting up nginx-full (1.2.1-2.2ubuntu0.1) ...
Setting up nginx (1.2.1-2.2ubuntu0.1) ...
$ sudo service nginx start
```

To test this is working, point your browser to http://localhost. You should get a **Welcome to nginx!** page or something similar.

Fixing Permissions

To change the User and Group nginx uses, you need to change the values in the fast cgi process nginx uses. Edit `/etc/php5/fpm/pool.d/www.conf` and change it to what you need.

```
# Using VIM
$ sudo vim /etc/php5/fpm/pool.d/www.conf

# Using Nano
$ sudo nano /etc/php5/fpm/pool.d/www.conf

# Using Sublime Text 2
$ sudo subl /etc/php5/fpm/pool.d/www.conf
```

Edit the "`user`" and "`group`" settings to what you need, I'm using *chuck* below.

```
user = chuck
group = chuck
```

To have changes take effect you must restart the php-fpm process.

```
$ sudo service php5-fpm restart
 * Restarting PHP5 FastCGI Process Manager php5-fpm
```

Adding an entry to /etc/hosts

If you don't have a domain name registered, you can "fake" one by editing your hosts file.

You'll need to edit this file as `root` using an editor.

```
# Using VIM
$ sudo vim /etc/hosts

# Using Nano
$ sudo nano /etc/hosts

# Using Sublime Text 2
$ sudo subl /etc/hosts
```

Add the entry. In this case we'll say 'myhost.localhost.com'. I personally use the pattern 'name.localhost.com', but have seen other use 'name.dev', or some other variation.

```
# somewhere in the file
myhost.localhost.com   127.0.0.1
```

Setup up a VirtualHost on UbuntuMint

You'll need to create this file as `root` using an editor.

```
# Using VIM
$ sudo vim /etc/nginx/sites-available/filename

# Using Nano
$ sudo nano /etc/nginx/sites-available/filename

# Using Sublime Text 2
$ sudo subl /etc/nginx/sites-available/filename
```

Edit the filename (for example, I use `gsd` for the project in this book) to match what's below. Change the paths and hostname as appropriate.

```
server {
    listen      80;
    server_name myhost.localhost.com;
    root        /laravel/project/path/public;

    location / {
        index     index.php;
        try_files $uri $uri/ /index.php?q=$uri&$args;
    }

    error_page 404 /index.php;

    location ~ \.php$ {
        include       fastcgi_params;
        fastcgi_index index.php;
        fastcgi_pass  unix:/var/run/php5-fpm.sock;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_param PATH_INFO $fastcgi_path_info;
        fastcgi_param PATH_TRANSLATED $document_root$fastcgi_path_info;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

Create a symbolic link to this configuration and restart nginx

```
$ cd /etc/nginx/sites-enabled
$ sudo ln -s /etc/nginx/sites-available/filename
$ sudo service nginx restart
Restarting nginx: nginx.
```

If there are no errors then your virtual host should be set up cool. You can test it by pointing your browser to it.