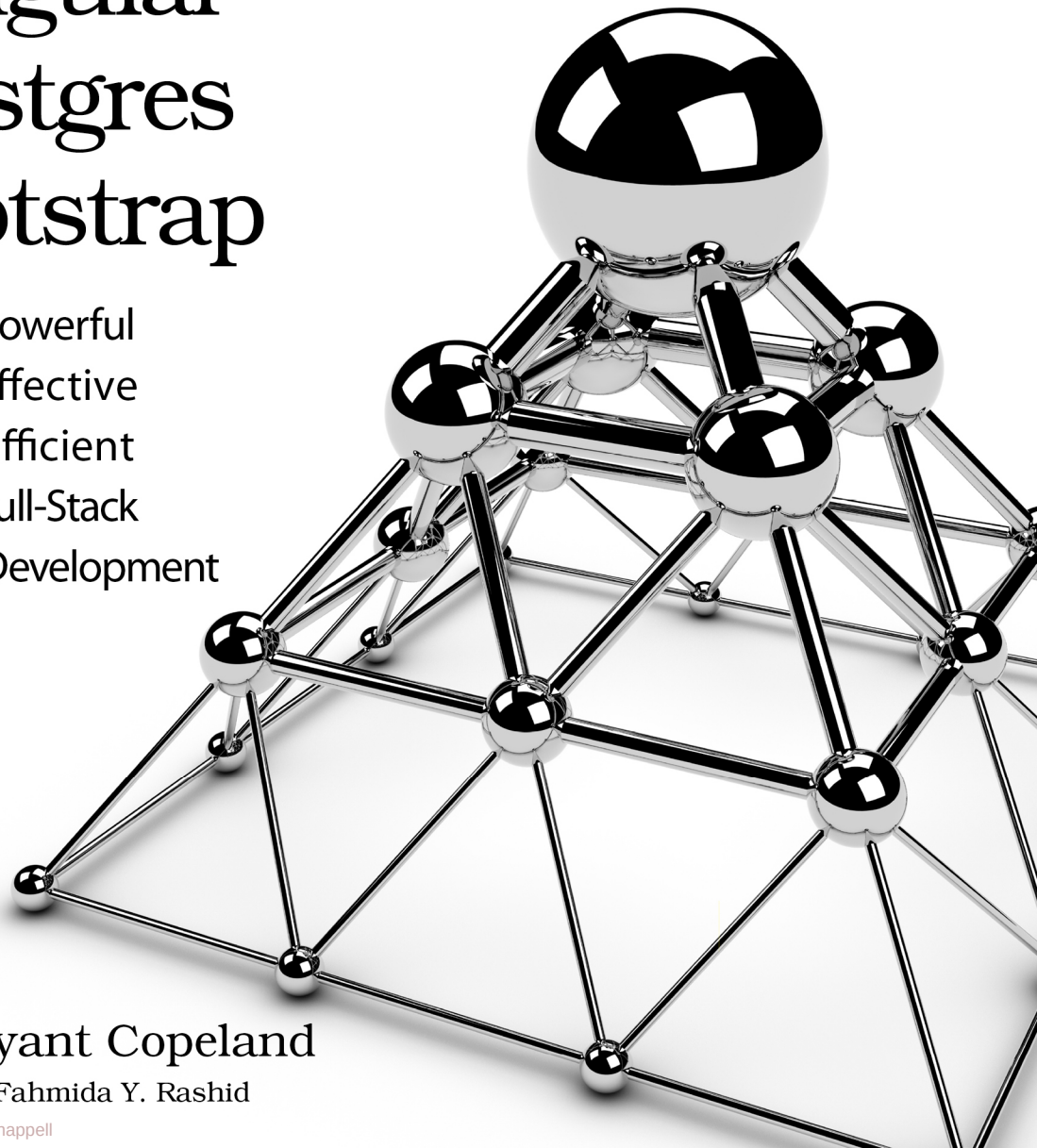
The Pragmatic Programmers

Rails Angular Postgres Bootstrap

Powerful
Effective
Efficient
Full-Stack
Web Development



David Bryant Copeland
edited by Fahmida Y. Rashid

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/dcbang/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Andy & Dave

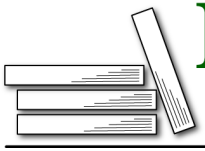
Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development

David Bryant Copeland

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-126-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: B2.0—September 9, 2015

Contents

Change History

The book you're reading is in beta. This means we update it frequently. This page lists the major changes that have been made at each beta release of the book, with the most recent changes first.

Beta 2.0—September 9, 2015

- The missing chapters have been written. We fixed some typos. Putting in technical review comments. Errata has been addressed.

Beta 1.0—July 23, 2015

- Initial beta release

CHAPTER 1

Introduction

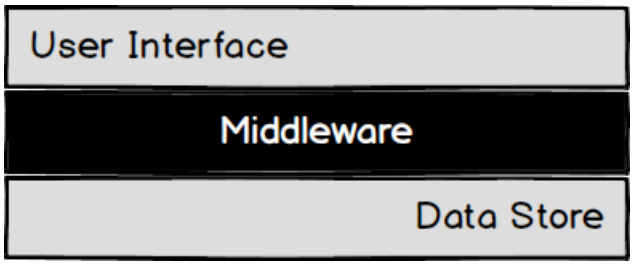
Think about what part of an application you are most comfortable working with. If you are a Rails developer, there's a good chance you prefer the back-end, the Ruby code that powers the business logic of your application. What if you felt equally comfortable working with the database, such as tweaking queries and using advanced features of your database system? What if you were *also* comfortable working with the JavaScript and CSS necessary to make dynamic, usable, attractive user interfaces?

If you had that level of comfort at every level of the application stack, you would possess great power as a developer to quickly produce high-quality software. Your ability to solve problems would not be restricted by the tools available via a single framework, nor would you be at the mercy of hard-to-find specialists to help you with what are, in reality, simple engineering tasks.

As a Rails developer, you are encouraged by the framework not to peer too closely into the database. Rails steers you away from JavaScript frameworks in favor of its *sprinkling* approach, where content is all rendered server-side. This book is going to open your eyes to all the things you can accomplish with your database, and set you on a path that includes JavaScript frameworks. With Rails acting as the foundation of what you do, you're going to learn how to embrace all other parts of the application stack.

The Application Stack

Many web applications—especially those built with Ruby on Rails—use a layered architecture that is often referred to as a *stack*, since most diagrams (like the following one) depict the layers as stacked blocks.



Rails represents the middle of the stack, and is called *middleware*. This is where the core logic of your application lives. The bottom of the stack—the data store—is where the valuable data saved and manipulated by your application lives. This is often a Relational Database Management System or RDBMS. The top of the stack is the user interface. In a web application, this is HTML, CSS, and JavaScript served to a browser.

Each part of the stack plays a crucial role in making software valuable. The data store is the canonical location of any organization's most important asset—its data. Even if your organization lost all of its source code, as long as it retained its data, it could survive. Losing all of the data, however, would be catastrophic.

The top of the stack is also important, as it's the way the users view and enter data. To the users, the user interface *is* the database. The difference between a great user interface and a poor one can be the difference between happy users and irritated users, accurate data and unreliable data, a successful product and a dismal failure.

What's left is the part of the stack where most developers feel most comfortable: the middleware. Poorly-constructed middleware is hard to change, meaning the cost of change is high, and thus the ability of the organization to respond to changes is more difficult.

Each part of the stack plays an important role in making a piece of software successful. As a Rails developer, you have amassed many techniques for making the middleware as high quality as you can. Rails (and Ruby) makes it easy to write clean, maintainable code.

Digging deeper into the other two parts of stack will have a great benefit for you as a developer. You'll have more tools in your toolbox, making you more effective. You'll also have a much easier time working with specialists, when you *do* have access to them, since you'll have a good grasp of both the database and the front-end. That's what you'll learn in this book. When you're done, you'll have a holistic view of application development, and you'll have a new and powerful set of tools to augment your knowledge of Rails. With

this holistic view, you can build seemingly complex features easily, sometimes even trivially.

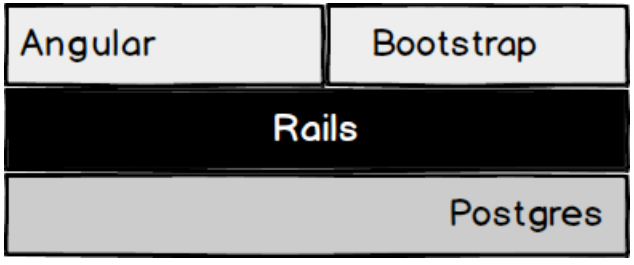
We'll learn *PostgreSQL*, *AngularJS* and *Bootstrap*, but you can apply many of the lessons here to other data stores, JavaScript libraries, and CSS frameworks. Outside of seeing just how powerful these specific tools can be, you're going to be emboldened to think about writing software beyond what is provided by Rails.

PostgreSQL, Angular, and Bootstrap: The Missing Parts of Our Stack

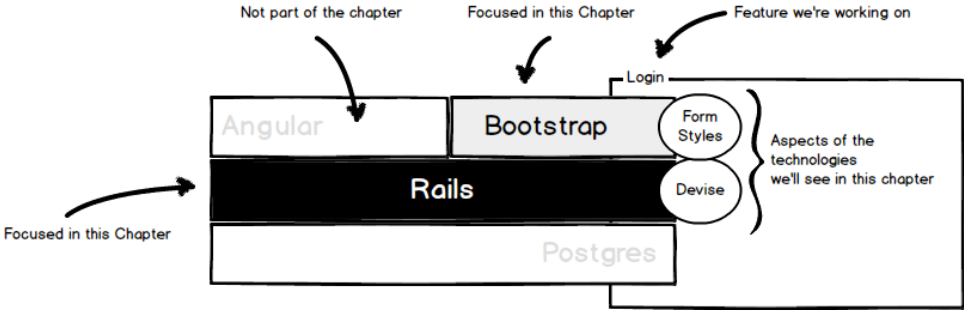
If all you've done with your database is create tables, insert data, and query it, you're going to be excited when you see what else you can do. Similarly, if all you've done with your web views is sprinkle some jQuery calls to your server-rendered HTML, you'll be amazed at what you can do with very little code when you have a full-fledged JavaScript framework. Lastly, if you've been hand-rolling your own CSS, a framework like Bootstrap will make your life so much simpler, and your views look and feel so much better.

In this book, we're going to focus on PostgreSQL as our data store—the bottom of the stack—and AngularJS with Bootstrap as our front-end—the top of the stack. Each of these technologies are widely used and very powerful. You're likely to encounter them in the real world, and they both underscore the sorts of features you can use to deliver great software outside of what you get with Rails.

With these chosen technologies, our application stack looks like so:



In each chapter, we'll highlight the parts of the stack we'll be focusing on, and calling out the different aspects of these technologies we'll be learning. Not every chapter will focus on all parts of the stack, so at the start of each chapter, you'll see a roadmap like this of what we'll be learning.



Let's get a taste of what each has to offer, starting with PostgreSQL.

PostgreSQL

PostgreSQL (or simply *Postgres*) is an open-source SQL database released in 1997. It supports many advanced features not found in other popular open-source databases such as MySQL[1] or commercial databases such as Microsoft SQL Server[2]. Here are some of the features we'll learn about (including how to use them with Rails):

Check Constraints You can create highly complex constraints on your table columns beyond what you get with `not null`. For example, you can require that a user's email address be on a certain domain (which we'll see in Chapter 3, *Secure the Login Database with Postgres Constraints*, on page 41), that the state in a U.S. address be written exactly as two upper-case characters, or even that the state in the address must already be on a list of allowed state codes.

While you can do this with Rails, doing it in the database layer means that no bug in your code, no existing script, no developer at a console, and no future program can put bad data into your database. This sort of data integrity just isn't possible with Rails alone.

Advanced Indexing In many database systems, you can only index the values in the columns of the database. In Postgres, you can index the *transformed* values. For example, you can index the lower-cased version of someone's name, so that a case-insensitive search is just as fast as an exact-match search. We'll see this in Chapter 4, *Use Fast Queries with Advanced Postgres Indexes*, on page 49.

Materialized Views A database view is a logical table based on a `SELECT` statement. In Postgres a *materialized view* is a view whose contents are

1. https://www.mysql.com/
2. http://www.microsoft.com/en-us/server-cloud/products/sql-server/

stored in backing table—accessing a materialized view won't run the query again like it would in a normal view. We'll use one in Chapter 10, *Cache Complex Queries Using Materialized Views*, on page 173.

Advanced Data Types Postgres has support for enumerated types, arrays, and dictionaries (called `HSTORE`s). In most database systems, you have to use separate tables to model these data structures.

Free-form JSON…that's indexed Postgres supports a `JSON` data type, allowing you to store arbitrary data in a column. This means you can use Postgres as a document data store, or for storing data that doesn't conform to a strong schema (something you'd otherwise have to use a different type of database for). And, by using the `JSONB` data type, the JSON fields can be indexed, just like a structured table's fields.

Although you can serialize hashes to JSON in Rails using the `TEXT` data type, you can't query it, and you certainly can't index it. `JSONB` fields can interoperate with many systems other than Rails, and provide great performance.

AngularJS

AngularJS[3] (or just *Angular*) is a JavaScript Model-View-Controller (MVC) framework created and maintained by Google (Angular bills itself as a Model-View-Whatever framework, but for this book, the *Whatever* will be a controller). Angular treats your view not as a static bit of HTML, but as a full-blown application. By adopting the mindset that your front-end is a dynamic, connected interface, and not a set of static pages, you open up many new possibilities for your users.

Angular provides powerful tools for organizing your code and lets you structure your markup to create intention-revealing, testable, managable front-end code. And it doesn't matter how small or large the task. As your UI gets more complex, Angular will scale much better than something more basic like jQuery.

As an example, consider showing and hiding a section of the DOM using jQuery. You might do something like this:

jquery_example.html

```
<section>
  <p>You currently owe: $123.45</p>
  <button class="reveal-button">Show Details</button>
```

3. https://angularjs.org

```
  <ul style="display: none" class="details">
    <li>Base fee: $120.00</li>
    <li>Taxes: $3.45</li>
  </ul>
</section>
<script>
  $(".reveal-button").click(function($event) {
    $(".details").toggle();
  });
</script>
```

It's not much code, but if you've ever done anything moderately complex, your markup and JavaScript becomes a soup of magic strings, classes starting with `js-` and oddball `data-` elements.

An Angular version of this might look like this:

angular_example.html

```
<section ng-app="account" ng-model="showDetails" ng-init="showDetails = false">
  <p>You currently owe: $123.45</p>
  <button ng-click="showDetails = !showDetails">Show/Hide Details</button>
  <ul ng-if="showDetails">
    <li>Base fee: $120.00</li>
    <li>Taxes: $3.45</li>
  </ul>
</section>
<script>
  var app = angular.module("account",[]);
</script>
```

Here, the view isn't just a description of static content, but a clear indication of how it should behave. Intent is obvious—you can see how this works without knowing the underlying implementation—and there's a lot less code. This is what a higher-level of abstraction like Angular gives you that would otherwise be a mess with jQuery or just plain JavaScript.

Unlike Postgres—where there are very few comparable open-source alternatives that match its features and power—there are many JavaScript frameworks comparable to Angular. Many of them are quite capable of handling the features we'll cover in this book. We're using Angular for a few reasons. First, it's quite popular, which means there are far more resources online for learning it, including deep dives beyond what we'll get to here. Secondly, it allows you to compose your front-end similarly to how you compose your back-end in Rails, but is flexible enough to allow you to deviate later if you need to.

If you've never done much with JavaScript on the front-end, or if you are just used to jQuery, you'll be pleasantly surprised at what Angular gives you:

Clean Separation of Code and Views Angular models your front-end as an application with its own routes, controllers, and views. This makes organizing your JavaScript easy, and tames a lot of complexity.

Unit Testing from the Start Testing JavaScript—especially when it uses jQuery—has always been a challenge. Angular was designed from the start to make unit testing your JavaScript simple and convienient.

Clean, declarative views Angular views are just HTML. Angular adds special attributes called *directives* that allow you to cleanly connect your data and functions to the markup. You won't have inline code or scripts, and there's a clear separation between view and code.

Huge ecosystem Because of its popularity, there is a large ecosystem of components and modules. Many common problems have a solution in Angular's ecosystem.

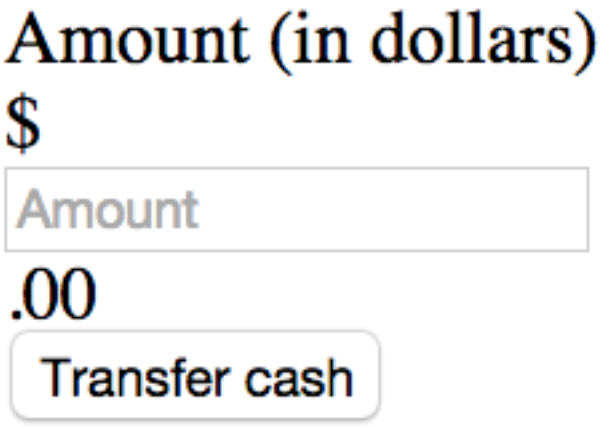
It's actually hard to fully appreciate the power of a JavaScript framework like Angular without using it, but we'll get there. We'll turn a run-of-the-mill search feature into a dynamic, asynchronous live search, with very little code.

Bootstrap

Bootstrap[4] is a *CSS framework* created by Twitter for use in their internal applications. A CSS framework is a set of CSS classes you apply to markup to get a particular look and feel. Bootstrap also includes *components*, which are classes that, when used on particular HTML elements in particular ways, produce a distinct visual artifact, like a form, panel, or an alert message.

The advantage of a CSS framework like Bootstrap is that you can create full-featured user interfaces without writing any CSS. Why be stuck with an ugly and hard-to-use form like this?

4. http://getbootstrap.com



By just adding a few classes to some elements, you can have something polished and professional like this instead:

In Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17, we'll do this to the login and registration forms provided by the Devise gem. We'll have a great-looking user sign-up and sign-in experience, without writing any CSS.

Bootstrap includes a lot of CSS for a lot of different occasions.

Typography Just including Bootstrap in your application, and using semantic HTML will result in pleasing content with good general typography.

Grid Bootstrap's grid makes it easy to layout complex, multi-column components. It can't be overstated how important and powerful this is.

Form Styles Styling good-looking forms can be difficult, but Bootstrap provides many CSS classes that make it easy. Bootstrap-styled forms have great spacing and visual appeal, and feel cohesive and inviting to users.

Components Bootstrap also includes myriad *components*, which are CSS classes that, when applied to particular markup, generate a visual component, like a styled box or alert message. These components can be great inspiration for solving simple design problems.

It's important to note that Bootstrap is not a replacement for a designer, nor are all UIs created with Bootstrap inherently usable. There are times when a specialist in either visual design, interaction design, or front-end implementation is crucial to the success of a project.

But for many software problems, you don't need these specialists, and they are often incredibly hard to find when you do. Bootstrap will allow you to produce a professional, visually appealing user interface in their absence. Bootstrap will also allow you to realize visual designs that might seem difficult to do with CSS. In Chapter 5, *Create Clean Search Results with Bootstrap Components*, on page 69 and Chapter 9, *Design Great UIs With Bootstrap's Grid and Components*, on page 155 we'll see just how easy it is to create a customized UI without writing CSS, all thanks to Bootstrap.

Even if you have access to a designer or front-end specialists, the skills you'll learn by using Bootstrap will still apply—your front-end developer isn't going to write every line of markup and CSS. They are going to hand you a framework like Bootstrap that enables you to do many of the things we'll do in this book.

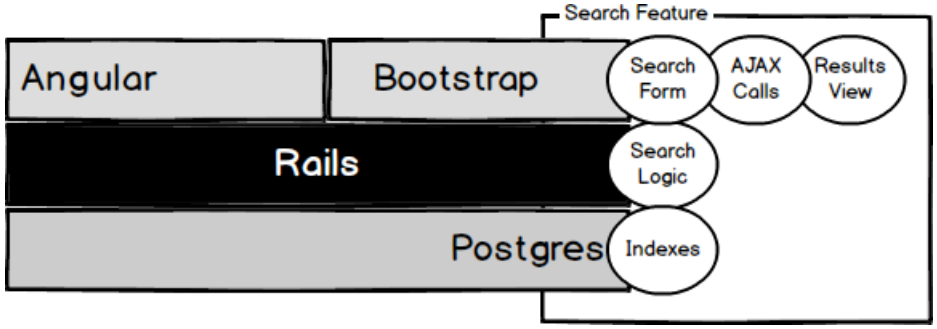
Now that we've gotten a taste of what we'll be learning, let's talk about how we're going to learn it.

Learning Postgres, Angular, and Bootstrap At The Same Time

If you've already looked at the table of contents, you'll see that this book isn't divided into three parts—one for Postgres, one for Angular, and one for

Bootstrap. That's not how a full-stack developer approaches development. A full-stack developer is given a problem to solve and is expected to bring all forces to bear in solving it.

For example, if you're implementing a search, and it's slow, you'll consider both creating an index in the database as well as performing the search with AJAX calls to create a more dynamic and snappy UI. You should use features at every level of the stack to get the job done.



This holistic approach is how we're going to learn these technologies. We're going to build a Rails application together, adding features one at a time. These features will demonstrate different aspects of the technologies we're using.

To keep things simple, each chapter will focus on one of these technologies, and we'll complete features over several chapters. For example, in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17, we'll set up a simple registration system for our application, and use Bootstrap to style the views. In Chapter 3, *Secure the Login Database with Postgres Constraints*, on page 41, we'll continue the feature, but focus on using Postgres to add extra security at the database layer. This will allow us to see a feature evolved and bring in every part of the application stack that's relevant. This will then give you the confidence to do the same for features that you build in your apps.

It's also worth emphasizing Rails' role in all of this. Although Rails doesn't have built-in APIs for using Postgres' advanced features, nor support for Angular's way of structuring code, it doesn't outright prevent our using them. And, Rails is a *great* middleware; probably one of the best.

So, in addition to learning Postgres, Angular, and Bootstrap, we're also going to learn how to get them working with Rails. We'll see not just how to create check constraints on columns in our tables, but how to do that from a Rails

migration. And we won't just learn how to style Angular components with Bootstrap, we'll do it using assets served up by the Asset Pipeline.

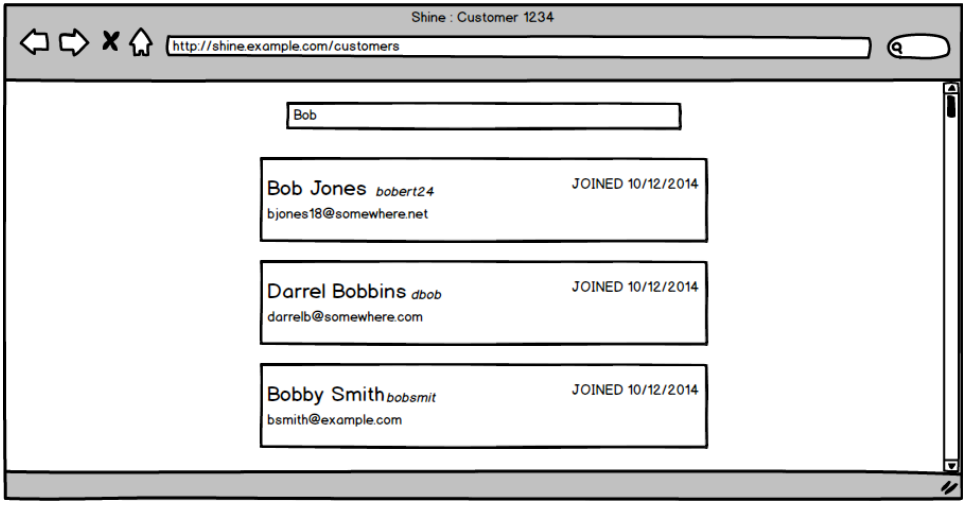
Let's learn about the Rails application that we'll be building throughout the book.

Shine, the Application We'll Build

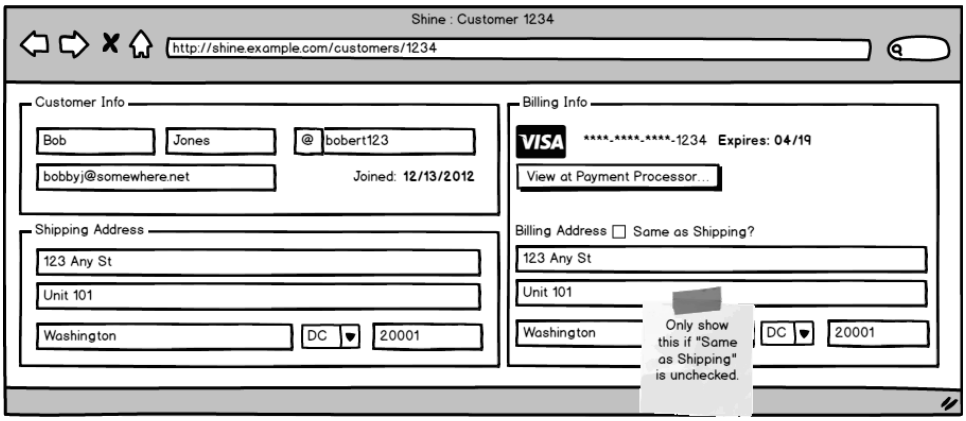
We'll create and add features to a Rails application over the chapters of this book. This application is going to be for the customer service agents at a hypothetical company where we work. Our company has a public website that its customers use, but we want a separate application for the customer service agents. You've probably seen or heard about internal-facing apps like this. Perhaps you've even worked on one (most software *is* internally-facing).

The application will be called *Shine* (since it allows our great customer service to *shine through* to our customers). The features that we'll build for this application in the book involve searching for, viewing, and manipulating customer data.

For example, we'll allow the user to search for customers.



And they can click through and view or edit a customer's data.



These features may seem simple on the surface, but there's hidden complexity that we'll be able to tame with Postgres, Angular, and Bootstrap. In each chapter, we'll make a little bit of progress on Shine, learning features of Postgres, Angular, Bootstrap, and Rails in the process.

How Each Chapter Will Work

As we mentioned, each chapter will focus on one part of our stack, as we build a part of a feature. To help us know where we are, each chapter will start with a diagram that shows which parts of the stack we'll be focusing on, what feature we're building, and what aspects of each technology we're going to be learning.

The first feature we'll build is a registration and login system, which will allow us to both style the user interface with Bootstrap, but also secure the underlying database with Postgres. We'll get our Rails application set up and style the login in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17. We'll then tighten up the security by learning about *check constraints* in Chapter 3, *Secure the Login Database with Postgres Constraints*, on page 41.

We'll then move onto a customer search feature, which is a fertile ground for learning about full-stack development. In Chapter 4, *Use Fast Queries with Advanced Postgres Indexes*, on page 49, we'll implement a naive fuzzy search, and learn how to examine Postgres' *query plan* to understand why our search is slow. We'll then use Postgres' advanced indexing features to make it fast. In Chapter 5, *Create Clean Search Results with Bootstrap Components*, on page 69, we'll learn how to use some of Bootstrap's built-in components and helper classes to create non-tabular search results that look great.

Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81, is an introduction to AngularJS, which we'll use to make our customer search much more dynamic. This chapter will allow us to learn how to setup and manage Angular as part of the Asset Pipeline, as well as how to read user input and do AJAX calls to our Rails application.

With a fully-implemented customer search, we'll take a pause at Chapter 7, *Test This Fancy New Code*, on page 103 to learn how to write tests for everything we've learned. Testing has always been a big part of Rails, so whenever we veer off of Rails' golden path, it's important to make sure we have a great testing experience.

Chapter 8, *Create a Single-Page App Using Angular's Router*, on page 139 will be our first step in building a more complex feature that shows customer details. We'll turn our customer search into a client-side, single-page application that allows the user to navigate from search results to customer details without reloading the page. This will let us learn about Angular's router and navigation features.

In Chapter 9, *Design Great UIs With Bootstrap's Grid and Components*, on page 155, we'll learn about a powerful web design tool called *the grid* and how Bootstrap implements it. We'll use it to create a dense UI that's clean, clear, and usable. In Chapter 10, *Cache Complex Queries Using Materialized Views*, on page 173, we'll implement the back-end of our customer details view by turning a query of highly complex joins into a simple auto-updated cache using Postgres' *materialized views*.

In Chapter 11, *Load Data from Many Sources Asynchronously*, on page 193, we'll learn how Angular's asynchronous nature allows us to keep our front-end simple, even when we need data from several sources. We'll finish off our customer detail page feature, as well as our in-depth look at these technologies, in Chapter 12, *Wrangle Forms and Validations with Angular*, on page 217, by learning about Angular's data binding, which will allow us to auto-save changes the user makes on the front-end.

All of this is just a small part of what you can do with Bootstrap, Angular, and Postgres, so in Chapter 13, *Dig Deeper*, on page 237, we'll survey some of the other features we don't have space to get to.

When it's all said and done, you'll have the confidence needed to solve problems by using every tool available in the application stack. You'll be just as comfortable creating an animated progress bar as you will setting up views and triggers in the database. Moreover, you'll see how you can use these sorts of features from the comfort of Rails.

Getting Set Up

To get ready to follow along, you don't need to do much to get yourself set up. You'll just need to install Ruby, Rails, and Postgres.

Ruby and Rails

If you don't have Ruby installed, you'll need to follow the instructions on Ruby's website[5]. I would recommend using an installer or manager, but as long as you have Ruby 2.2 installed and the ability to install gems you'll be good to go.

With Ruby installed, you'll need to install Rails. This is usually as simple as:

```
> gem install rails
```

You want to make sure to get the latest version of Rails, which is 4.2 at the time of this writing.

Postgres

Postgres is free and open-source, and you can install it locally by looking at the instructions for your operating system on their website[6]. Make sure you get version 9.4, as some of the features we'll discuss were introduced in that version. In Chapter 3, *Secure the Login Database with Postgres Constraints*, on page 41, we'll outline how to set up a user to access the databases.

An alternative is to use a free, hosted version of Postgres. Heroku Postgres[7] provides such a version, and will work great with this book. You can sign up on their website, and they'll give you the credentials to access the hosted database from your computer (we'll show you where to use them in the next chapter).

Everything Else

Everything else we'll need to install, we'll setup and install as we get to it. We'll be using a lot of third-party libraries and tools—integrating them together is what this book is about. Pay particular attention to the versions of libraries in `Gemfile` and `Bowerfile` that are included in the example code download. While we've tried to make the code future-compatible, there's always a chance that a point release of a library breaks something.

5. https://www.ruby-lang.org/en/downloads/
6. http://www.postgresql.org/download/
7. https://www.heroku.com/postgres

At a high level, the book was written with the following versions of the tools and libraries:

- Ruby 2.2
- Rails 4.2
- Bootstrap 3
- Angular 1.4
- Teaspoon 2
- Devise 3.5
- Postgres 9.4

We expect everything here to work with Rails 5 and Postgres 9.5. At the time of this writing, Angular 2 is set to be a ground-up rewrite—essentially a totally new framework—but 1.4 is expected to be supported for quite some time. Bootstrap 4 has no current release schedule, but it's expected to create breaking changes.

Example Code

The running examples in the book are extracted from full-tested source code that should work as shown, but you should download the sample code from http://pragprog.com. Each step of our journey through this topic has a different sudirectory, each containing an entire Rails application. While the book is only showing you the changes you need to make, we've tried to make sure that the downloadable code records a snapshot of the fully-working application as of that point in the book.

Online Forum and Errata

While reading through the book, you may have questions about the material, or you might find typos or mistakes. For the latter, you can add issues to the Errata for the book at http://www.pragprog.com. Think of it as a bug-reporting system for the book.

For the former—questions about the material—you should visit the online forum at http://www.pragprog.com. There, you'll be able to interact with me and other readers to get the most out of the material.

I hope you're ready to start your journey in full-stack application development! Let's kick it off by creating our new Rails application and setting up a great-looking, and secure, login system.

CHAPTER 2

Create a Great-Looking Login with Bootstrap and Devise

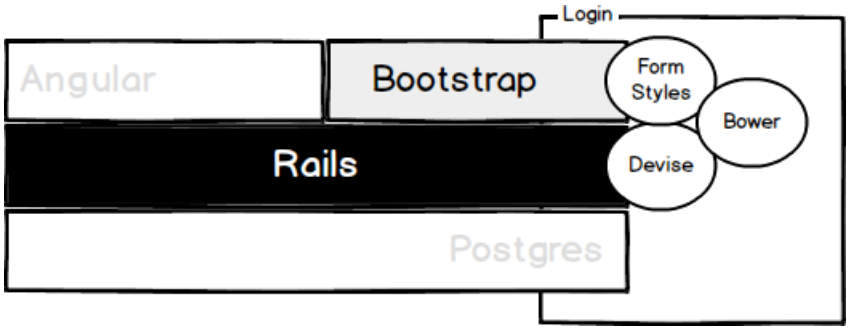
Applications aimed at a company's employees require some sort of restrictions around who can access them. These *internal apps* are instrumental in running the company's operations, so we want to be careful about who has access to them. This means we need some sort of authentication system.

An authentication system is a great first-step into thinking about problems as a full-stack developer, as we want the user experience to be great, but we also want the back-end to be secure, all the way down to the data layer. In this chapter, we'll use the *Devise* gem to handle the middleware bits of authentication, and use Bootstrap to make the user experience great.

Because this is the first chapter, we'll need to setup our fresh Rails application. But, we'll also need to get Bootstrap installed. Since Shine (the name of our Rails application) will eventually use AngularJS and other front-end components, we'll use the installation of Bootstrap as a way to learn about *Bower*, which is like Bundler[1], but for JavaScript and CSS libraries.

Here's where everything we'll learn sits in our application stack.

1. http://bundler.io/



Let's get to it, and set up our fresh Rails application with a simple login system powered by Devise.

Setting up Devise for Authentication

Creating an authentication system from scratch is rarely a good idea. It's difficult to get every part of it correct since security controls can be subverted in unusual and counter-intuitive ways. Because of this, we're going to use a tried-and-true Rails plugin called Devise[2].

Devise is built to handle almost any sort of authentication requirement, and will definitely suit our needs. Here's what we want our authentication system to do:

- Employees who need to use our app will sign up on their own.
- They must use their company email when signing up.
- Their password must be at least 10 characters long.

We aren't going to require users to validate their email addresses, mostly to keep things simple by avoiding email configuration. You should consider it for a real app, and Devise makes it easy to use once you've fully configured your mailers[3].

Since adding a login is the first feature we're going to implement, we'll start by setting up our Rails application. Note that while we're adding Devise to a brand-new Rails application, it's just as simple to add Devise to an already-in-development application.

2. https://github.com/plataformatec/devise
3. http://guides.rubyonrails.org/action_mailer_basics.html#action-mailer-configuration

Setting up Our Rails App

We really don't need much more than the basic Rails application we will generate with rails new, but since we know we're using Postgres now and later in the book we'll be using AngularJS, there are a few options we want to set now.

Let's call our new application shine (since it will allow our users to *shine* as they do their work, assisted by all the wonderful features we'll be able to add for them). It will use Postgres instead of the default SQLite database.

We don't want to use TurboLinks[4], because it's going to clash with the JavaScript we'll be writing later on when we start to use AngularJS. We're skipping Spring[5] as well, mostly because it isn't 100% reliable, and it could cause your experience with these examples to not mimic the one we're describing in the book. Finally, we're skipping Test::Unit as our testing framework, because we're going to be using RSpec. As we'll see in Chapter 7, *Test This Fancy New Code*, on page 103, we have good reasons to use RSpec, so bear with me for now.

Here's the command to create a Rails app according to our specifications (if you haven't installed Rails, you'll first need to gem install rails).

```
> rails new --skip-turbolinks \
            --skip-spring \
            --skip-test-unit \
            -d postgresql \
            shine
```

This will create our new application as you'd expect. Before we run it, we'll need to set up our database (if you don't have Postgres installed, review the setup instructions in *Postgres*, on page 14). If you've installed Postgres locally, you'll need to create a user. If you are using Postgres-as-a-service, you should have a user created already and should skip this step. Our user will be named shine and have the password shine.

```
> createuser --createdb --login -P shine
```

You'll be prompted for a password, so enter "shine" twice, as requested. --createdb tells Postgres that our user should be able to create databases (needed in a later step). The --login switch will allow our user to login to the database and -P means we want to set our new user's password right now (which is why you were prompted for a password).

4. https://github.com/rails/turbolinks
5. https://github.com/rails/spring

Next, modify config/database.yml so the app can connect to the database.

```
default: &default
  adapter: postgresql
  encoding: unicode
  username: shine
  password: shine
  pool: 5

development:
  <<: *default
  database: shine_development

test:
  <<: *default
  database: shine_test
```

Note that if you are using Postgres-as-a-service, you'll need to use the credentials you were given instead of what's shown here. Typically, you'll get a url, so you'll have to manually break it up into the pieces needed in config/database.yml. The URL is usually of the form postgres://some_user:their_password@some_host.com:PORT/database_name.

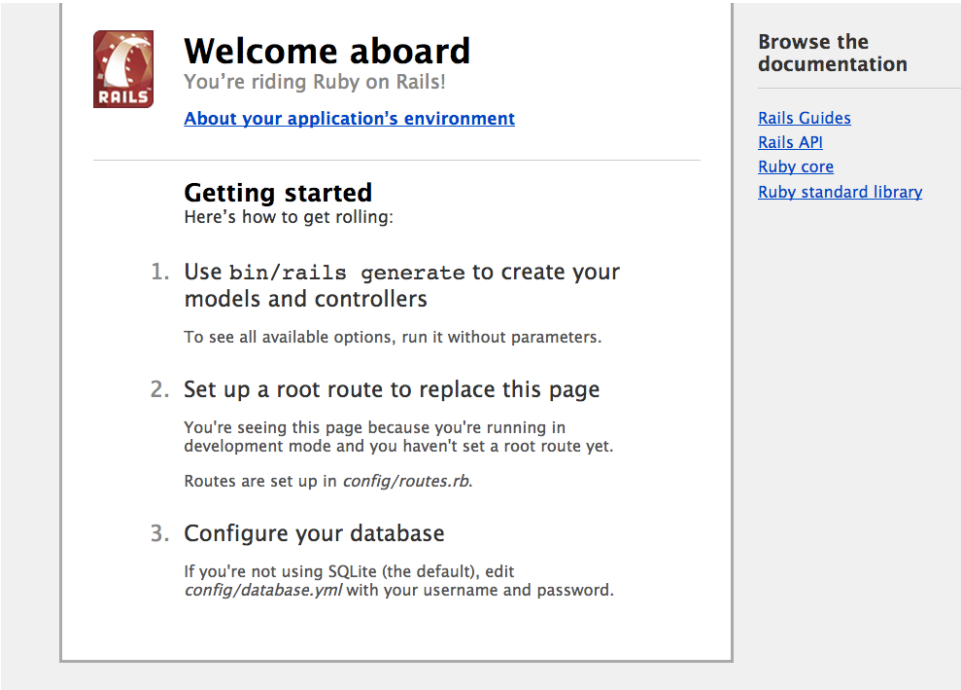
Next, we'll set up our database.

```
> cd shine
> bundle exec rake db:create
> bundle exec rake db:migrate
```

We can now start the app to verify that everything worked. Although we don't have any database tables, Rails should complain if the database configuration is wrong, so this is a decent test of our configuration.

```
> bundle exec rails server
```

You can now visit http://localhost:3000 and see the familiar Rails welcome page.



Lastly, we need to make a page in our app that will require authentication. We'll call this the *dashboard*, and our initial version will have a simple, static view.

Add the route to config/routes.rb.

login/create-dashboard/shine/config/routes.rb

```
root 'dashboard#index'
```

Next, create app/controllers/dashboard_controller.rb.

login/create-dashboard/shine/app/controllers/dashboard_controller.rb

```
class DashboardController < ApplicationController
  def index
  end
end
```

Finally, create app/views/dashboard/index.html.erb with some basic content.

login/create-dashboard/shine/app/views/dashboard/index.html.erb

```
<header>
  <h1>
    Welcome to Shine!
  </h1>
  <h2>
    We're using Rails <%= Rails.version %>
  </h2>
```

```
</header>
<section>
  <p>
    Future home of Shine's Dashboard
  </p>
</section>
```

Restarting your server, and reloading your app, you should see the page we've created.

Now that we have a working Rails app, we can install Devise to get our login going.

Installing Devise

First, we'll add Devise to our Gemfile.

login/install-devise/shine/Gemfile

```
gem 'devise'
```

Next, we'll install it using Bundler.

```
> bundle install
```

Devise includes several generators we can use to simplify the setup and initial configuration. The devise:install generator is the first one we'll need to bootstrap Devise in our app.

```
> bundle exec rails generate devise:install
      create  config/initializers/devise.rb
      create  config/locales/devise.en.yml
```

This command will also output a fairly lengthy message about further actions to take to set up Devise. We'll be addressing all of that advice in this section, so don't worry about it for now.

Next, we need to tell Devise what model and database table we'll use for authentication. Even though our company's public website has a user authentication mechanism for our *customers*, we want to use a separate system for our internal users. This allows both systems to vary as needed for different parts of the business. It also creates a much more explicit wall between customers and users, and prevents customers from having access to our internal systems.

Devise is part of that separate system, but we also need a separate database table and model. Since we refer to our customers as "customers", we'll refer to our internal users as simply "users". Devise can create that table for us, using a generator called devise. It will create a User ActiveRecord model and database table (called USERS) with the fields necessary for Devise to function.

```
> bundle exec rails generate devise user
    invoke  active_record
    create    db/migrate/20150228234349_devise_create_users.rb
    create    app/models/user.rb
    insert    app/models/user.rb
     route  devise_for :users
```

Let's have a look at what it created by examining the migration.

login/install-devise/shine/db/migrate//20150228234349_devise_create_users.rb

```
class DeviseCreateUsers < ActiveRecord::Migration
  def change
    create_table(:users) do |t|
      ## Database authenticatable
      t.string :email,              null: false, default: ""
      t.string :encrypted_password, null: false, default: ""

      ## Recoverable
      t.string   :reset_password_token
      t.datetime :reset_password_sent_at

      ## Rememberable
      t.datetime :remember_created_at

      ## Trackable
      t.integer  :sign_in_count, default: 0, null: false
      t.datetime :current_sign_in_at
      t.datetime :last_sign_in_at
      t.inet     :current_sign_in_ip
      t.inet     :last_sign_in_ip

      t.timestamps
    end

    add_index :users, :email,                unique: true
    add_index :users, :reset_password_token, unique: true
  end
end
```

Each section that's commented indicates which Devise modules the fields are relevant to. Don't worry about what those are for now. We also won't add any fields of our own at this point. If we need some later, we can always add them with a new migration.

There's one last step before we can finally see Devise in action. We need to indicate which controller actions require authentication. Without that, Devise won't do anything, as it will perceive all pages as being open to anonymous users.

Devise provides a controller filter called authenticate_user!, and we can use that in our ApplicationController, since we want *all* pages and actions to be restricted.

login/install-devise/shine/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
➤ before_action :authenticate_user!
end
```

As a way for us to be certain we've actually authenticated the user, let's show their email address on the dashboard. Devise provides a helper method called current_user, which returns the User instance of the currently-authenticated user. Since it's a helper, we can use it directly in our view.

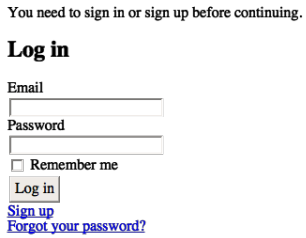
login/install-devise/shine/app/views/dashboard/index.html.erb

```
<header>
  <h1>
➤   Welcome to Shine, <%= current_user.email %>
  </h1>
  <h2>
    We're using Rails <%= Rails.version %>
  </h2>
</header>
<section>
  <p>
    Future home of Shine's Dashboard
  </p>
</section>
```

Now, we're ready to see it working. We'll need to run migrations and then start our server.

```
> bundle exec rake db:migrate
== 20150228234349 DeviseCreateUsers: migrating ================================
-- create_table(:users)
   -> 0.0538s
-- add_index(:users, :email, {:unique=>true})
   -> 0.0043s
-- add_index(:users, :reset_password_token, {:unique=>true})
   -> 0.0025s
== 20150228234349 DeviseCreateUsers: migrated (0.0608s) =======================
> bundle exec rails server
```

Navigating to http://localhost:3000 will no longer show the dashboard page, but will instead ask us to log in or sign up.



Since we don't have a user account yet, let's create one by clicking "Sign up".

If you fill in the fields, your account will be created and you'll be automatically logged in. You should be able to see your email address on the dashboard page, just as we wanted.

We can also see that Devise has created an entry in the `USERS` table by going into the database directly. We can do this using the `dbconsole`[6] command for Rails.

```
> bundle exec rails dbconsole
postgres> \x on
Extended display is on.
postgres> select * from users;
-[ RECORD 1 ]----------+----------------------------------------------------
id                     | 1
email                  | exampleuser@example.com
encrypted_password     | $2a$10$h6OFdaL5FsYwoRTwUF.hCrSgT0XEiMUt1x4Owk3Kqe
reset_password_token   |
reset_password_sent_at |
remember_created_at    |
sign_in_count          | 1
current_sign_in_at     | 2015-02-28 23:51:40.315758
last_sign_in_at        | 2015-02-28 23:51:40.315758
```

6. http://guides.rubyonrails.org/command_line.html#rails-dbconsole

```
current_sign_in_ip     | ::1
last_sign_in_ip        | ::1
created_at             | 2015-02-28 23:51:40.302823
updated_at             | 2015-02-28 23:51:40.322173
```

This is an amazing amount of functionality just for installing a gem and adding a few lines of code to our Rails application. And, since Devise is tried and tested, we know our authentication system is solid and dependable. But, it's ugly.

We could open up app/assets/stylesheets/application.css and start trying to make it look better, but we don't have to. Bootstrap provides a ton of styles we can apply to our markup that will make our login look great, and we won't have to write any CSS.

Installing Bootstrap with Bower

We want our user's experience with Shine to be good, but we don't have hours and hours to spend styling and perfecting it. We also might not even have the *expertise* to do a good job. The reality of software development, especially for internal tools, is that there's rarely enough time or people to work on a great design.

Fortunately, there are now many *CSS frameworks* available that can help us produce a *decent* design. A framework is a set of re-usable classes that we can apply—without writing any actual CSS—to style our markup. For example, a framework might set up a set of font sizes that work well when used together. It may also provide classes we can apply to form fields to make them layout effectively on the page.

Bootstrap[7] is one of the most popular and widely-used CSS frameworks, and it will give users an immense amount of power to control the look and feel of the app, without having to write any CSS ourselves. Bootstrap is no replacement for an actual designer as its default visual style won't win any design awards. But, for an internal application like Shine, it's perfect. It will make our app look good.

To turn our ugly log-in and sign-up forms into something we can be proud of, we need to install Bootstrap, configure it in our Rails application's Asset Pipeline[8], and then make a few changes to the markup Devise provides for those screens.

7. http://getbootstrap.com
8. http://guides.rubyonrails.org/asset_pipeline.html

Installing Bootstrap

There are three ways to install Bootstrap into a Rails application: as a Ruby Gem, by referencing a publicly-available version hosted on a CDN[9], or by downloading it and storing it within our application's source.

We're going to use the latter, assisted by a tool called Bower[10]. Later in the book, we'll need to use some front-end assets (CSS and JavaScript libraries) that aren't available as RubyGems, so Bower will become our one-stop shop for all things front-end (rather than having some assets managed via our Gemfile and some managed by Bower. See *Why not use RubyGems?*, on page 27 for more information about this decision).

Why not use RubyGems?

It may seem easier to just use the gemified versions of front-end assets instead of setting up Bower to download and manage everything. The problem is that RubyGems becomes a needless middleman, obscuring the actual versions of the assets you are using (as of this writing, the jquery-rails gem is at version 4.0.3, is compatible with Rails 4.2 and bundles jQuery 2.1.3, which I find incredibly confusing).

Further, you end up relying on others to package assets you need, so you have to hope that the asset you want is available, *and* that the latest version has been packaged by the gem's maintainer. If either of those is not the case, you have to find an alternate solution, which means either *you* become the gem's maintainer, or you just download the assets manually.

This is a very real problem, since the world of front-end libraries is much larger than Rails. The community around front-end asset frameworks like Bootstrap has chosen Bower as the means to manage these assets. Even though it's a bit of extra work to set up on our end, using Bower will give us flexibility and predictability in our toolchain.

To install Bootstrap using Bower, we'll first need to install Bower, then we'll instruct it to download and install Bootstrap and, finally, we'll configure the asset pipeline to make Bootstrap available.

Install Bower

Bower bills itself as a package manager for the web. It was created by Twitter and is analogous to RubyGems, but manages front-end assets (including CSS frameworks like Bootstrap and JavaScript libraries like Angular). Bower is

9. http://en.wikipedia.org/wiki/Content_delivery_network
10. http://http://bower.io

written in JavaScript, and so it requires a JavaScript runtime in order to work. That means you'll have to install Node.js.

To install Node[11], simply visit http://nodejs.org and follow the instructions for your operating system. In particular, you may want to check out the page on installing Node via package managers[12], as it has extensive documentation for various operating systems.

Once you have Node installed, you'll have access to the `npm` command-line application. This is a package manager for JavaScript, and we'll use that to install Bower.

```
> npm install -g bower
```

It may seem comical to install a package manager just to install another package manager that we'll use alongside our existing, Ruby-based package manager, but this is how it is. Bower is going to make our lives easier by giving us clear and complete control over our front-end assets. This initial set-up is going to be more than worth it.

With Bower installed, we *could* manage our dependencies using the `bower` command-line application. However there is a Ruby gem called "bower-rails" that makes our interaction with Bower a bit more "Rails-like", so let's install that as well by adding it to our `Gemfile`.

login/install-bootstrap/shine/Gemfile

```
gem 'bower-rails'
```

We can install it with Bundler.

```
> bundle install
```

The bower-rails gem does two things for us. First, it allows us to specify dependencies in a simple file called `Bowerfile`, which will be easier to work with than the JSON format[13] required by the `bower` command-line app. Bower-rails also provides rake tasks to run Bower for us.

```
> bundle exec rake -T bower
rake bower:cache:clean                    # Clear the bower cache
rake bower:clean                          # Attempt to keep only files...
rake bower:install[options]               # Install components from bower
rake bower:install:deployment[options]    # Install components from bower...
rake bower:install:development[options]   # Install both dependencies...
rake bower:install:production[options]    # Install only dependencies, exc...
```

11. http://nodejs.org
12. https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager
13. http://bower.io/docs/creating-packages/#bowerjson

```
rake bower:list                          # List bower components
rake bower:resolve                       # Resolve assets paths in bower...
rake bower:update[options]               # Update bower components
rake bower:update:prune[options]         # Update existing components....
```

We'll see how Bowerfile works in the next section, when we download Bootstrap into our application.

Download Bootstrap

Now that Bower is installed, we'll add our first front-end dependency, which is for Bootstrap. Bower-rails will look for Bowerfile to find our list of dependencies. It looks very similar to a Gemfile and exposes the method asset for specifying a front-end dependency. We'll create Bowerfile like so:

login/install-bootstrap/shine/Bowerfile

```
asset 'bootstrap-sass-official'
```

How did we know to use the string 'bootstrap-sass-official' to download Bootstrap? We found it by searching the Bower central registry. Bower packages are hosted in public git repositories (usually hosted on GitHub) and registered at http://bower.io/search, which is analogous to http://rubygems.org. You can search that registry for any package with the name "bootstrap" by running bower search boostrap.

```
> bower search bootstrap | head
Search results:

    bootstrap git://github.com/twbs/bootstrap.git
    angular-bootstrap git://github.com/angular-ui/bootstrap-bower.git
    bootstrap-sass-official git://github.com/twbs/bootstrap-sass.git
    sass-bootstrap git://github.com/jlong/sass-bootstrap.git
    bootstrap-datepicker git://github.com/eternicode/bootstrap-datepicker.git
    bootstrap-select git://github.com/silviomoreto/bootstrap-select.git
    angular-ui-bootstrap-bower git://github.com/angular-ui/bootstrap-bower
    angular-ui-bootstrap git://github.com/angular-ui/bootstrap.git
```

The list of results is *much* longer than what we've shown here (Bootstrap has a *ton* of add-ons in Bower's registry), but the very first result shows a URL to the official Bootstrap source code on GitHub. The string preceding that URL is the name you need to use in your Bowerfile to bring down that asset.

In our case, we don't want to use the first one, because that version is using LESS[14] as a CSS pre-processor. Rails uses SASS for CSS-preprocessing and, , we'll be using features of SASS to customize Bootstrap. So, we might as well use the SASS version now, which is available as "bootstrap-sass-official".

14. http://lesscss.org

With our Bowerfile ready to go, we'll use the bower:install bundle exec rake task, provided by bower-rails, to install Bootstrap.

```
> bundle exec rake bower:install
bower.js files generated
/usr/local/share/npm/bin/bower install -p
bower bootstrap-sass-official#* cached git://github.com/twbs/bootstrap-...
bower bootstrap-sass-official#*  validate 3.3.1 against git://github.co...
bower bootstrap-sass-official#*       new version for git://github.com/...
bower bootstrap-sass-official#*   resolve git://github.com/twbs/bootstr...
bower bootstrap-sass-official#*  download https://github.com/twbs/boots...
bower bootstrap-sass-official#*   extract archive.tar.gz
bower bootstrap-sass-official#*  resolved git://github.com/twbs/bootstr...
bower jquery#>= 1.9.0              cached git://github.com/jquery/jquer...
bower jquery#>= 1.9.0             validate 2.1.3 against git://github.co...
bower bootstrap-sass-official#*   install bootstrap-sass-official#3.3.3
bower jquery#>= 1.9.0              install jquery#2.1.3

bootstrap-sass-official#3.3.3 bower_components/bootstrap-sass-official
└── jquery#2.1.3

jquery#2.1.3 bower_components/jquery
```

By default, bower-rails places files in vendor/assets/bower_components and if you look in that directory, you'll see all of Bootstrap's JS and CSS files have been downloaded. You'll also notice that jQuery was installed. This is because the JavaScript parts of Bootstrap require jQuery. Bower detects this transitive dependency and handles downloading it, much as RubyGems would in a similar situation.

We won't actually be using Bootstrap's JavaScript, nor will we be using jQuery, but there isn't an easy way to avoid downloading them (as there is no "CSS-only" version of bootstrap-sass). Ultimately, it won't matter because we won't configure the asset pipeline to serve them—they'll essentially be inert files in our project.

Now that Bootstrap has been downloaded, we want to configure Rails to serve it up as part of our application's CSS assets.

Add Bootstrap to our Asset Pipeline

The Rails asset pipeline manages the deployment of front-end assets to a user's browser. Essentially, it allows us to organize our JavaScript and CSS however we'd like, but have all of it packaged up at runtime into only two files: one for CSS, one for JavaScript.

Rails' default configuration of the asset pipeline grabs all files in app/assets/stylesheets and app/assets/javascripts and packages them as application.css

and application.js, respectively. Since we've used Bower to bring in CSS (and, eventually, JavaScript) outside of app/assets, we'll need to add a bit more configuration for the asset pipeline to know about them.

Much like how we piece together a Ruby application by setting paths and using require, the asset pipeline (which is powered by sprockets[15]), is configured with *asset paths* which contain *directives* which describe all the files we want to serve as assets.

Bower-rails will automatically add vendor/assets/bower_components to the asset path for us, so there's no need to do any additional configuration. To use Bootstrap's CSS files, we'll use the require directive to tell it to bring them into our application. Because the default application layout references application.css we can add this directive in app/assets/stylesheets/application.css, which is where Rails places that file by default.

login/install-bootstrap/shine/app/assets/stylesheets/application.css

```
/*
 *= require_tree .
 *= require_self
 *= require 'bootstrap-sass-official'
 */
```

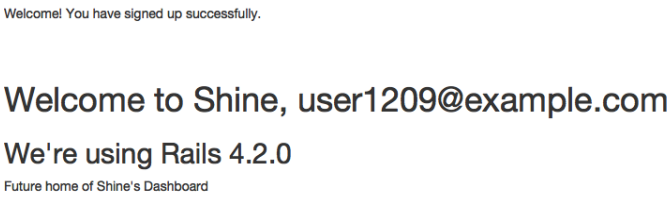
Generally, require will bring in a file named for the string following the directive, which would be bootstrap-sass-official.css in this case. If you look in vendor/assets/bower_components, you won't see that file, so how does this work?

Distributing assets for Bower requires creating a *manifest* file named bower.json that describes the package. Bootstrap is packaged this way and if you look at vendor/assets/bower_components/bootstrap-sass-official/bower.json, you can see its manifest. Sprockets actually knows to look for that file.

Sprockets sees that we've required bootstrap-sass-official, sees that there's a directory named that inside one of its asset paths (namely vendor/assets/bower_components), and sees that *that* directory contains a bower.json. It then reads that file to figure out the actual files to serve up as assets.

Let's see it in action. When you you navigate to the homepage, you'll see that the font has changed from your browser's default (likely Times New Roman) to Helvetica, which is what Bootstrap uses by default.

15. https://github.com/sstephenson/sprockets



If you view the source being served, you'll see that our application has picked up the assets in vendor/assets/bower_components/bootstrap-sass-official.

```
<link
 rel="stylesheet"
 media="all"
 href="/assets/application-78386373b678ffd4f96d8483bd55b01c.css?body=1" />

<link rel="stylesheet"
 media="all"
 href="/assets/bootstrap-sass-official/assets/stylesheets\
       /_bootstrap-242cc2b1c4514d91448d4c8fdeb4662b.css?body=1" />
```

This has been reformatted to fit the page here, but you can see that Rails has figured out from bower.json that the CSS files are in vendor/assets/bower-components/bootstrap-sass-official/assets/stylesheets.

With Bootstrap now installed, we can make use of the classes and components it provides to make our login experience look great.

Styling The Login and Registration Forms

Bootstrap doesn't do much to naked elements in our markup. It sets the default font and makes a few color changes, but most of what Bootstrap does requires us to add classes to certain elements in a particular way. This means we'll need access to the markup before we get started.

You might recall that we didn't write any markup for our login screens—they were all provided by Devise. Devise is packaged as a Rails Engine[16], so the gem itself contains the views. But, it also contains a generator called devise:views that will extract those views into our application, allowing us to modify them.

```
> bundle exec rails generate devise:views
  invoke  Devise::Generators::SharedViewsGenerator
  create    app/views/devise/shared
  create    app/views/devise/shared/_links.html.erb
  invoke  form_for
  create    app/views/devise/confirmations
  create    app/views/devise/confirmations/new.html.erb
```

16. http://guides.rubyonrails.org/engines.html

```
create    app/views/devise/passwords
create    app/views/devise/passwords/edit.html.erb
create    app/views/devise/passwords/new.html.erb
create    app/views/devise/registrations
create    app/views/devise/registrations/edit.html.erb
create    app/views/devise/registrations/new.html.erb
create    app/views/devise/sessions
create    app/views/devise/sessions/new.html.erb
create    app/views/devise/unlocks
create    app/views/devise/unlocks/new.html.erb
invoke  erb
create    app/views/devise/mailer
create    app/views/devise/mailer/confirmation_instructions.html.erb
create    app/views/devise/mailer/reset_password_instructions.html.erb
create    app/views/devise/mailer/unlock_instructions.html.erb
```

Now that we can edit these files, we can use Bootstrap's CSS classes to make them look how we'd like.

Since we'd like to style both the login screen *and* the registration screen, we need a way to log ourselves out so we can see them. Devise set up all the necessary routes for us, so we just need to create a link to the right path in app/views/dashboard/index.html.erb.

login/use-bootstrap/shine/app/views/dashboard/index.html.erb

```
<section>
  <p>
    Future home of Shine's Dashboard
  </p>
➤ <%= link_to "Log Out", destroy_user_session_path, method: :delete %>
</section>
```

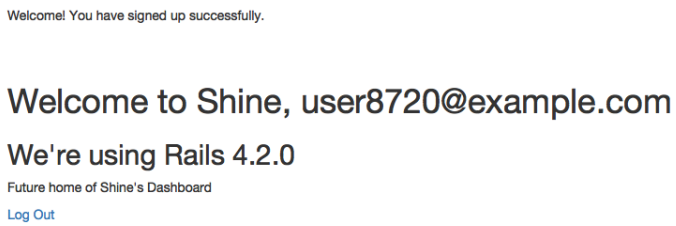
With that link in place, we can log out to see the screens we're going to style. We're just going to be using the styles Bootstrap provides—we aren't writing any CSS ourselves. You'll be amazed at how much better our screens are with just these simple changes.

First, we need to make sure all of our markup is in one of Bootstrap's "containers", which will "unlock" many of the features we need. We can apply this to the body element in our application layout.

login/use-bootstrap/shine/app/views/layouts/application.html.erb

```
➤ <body class="container">
  <p class="notice"><%= notice %></p>
  <p class="alert"><%= alert %></p>

<%= yield %>

</body>
```

Reloading our app, we can see that this class added some sensible margins and padding.



Let's start with the login screen.

Styling the Login Screen

Since Devise uses Rails' RESTful routing scheme, the "resource" around logging in is called a "user session". Therefore, the view for the login screen is in app/views/devise/sessions/new.html.erb.

For styling forms, Bootstrap's documentation[17] has several different options. We'll use the first, most basic, one, which is perfect for our needs.

We'll wrap each label and input element in a div with the class form-group and we'll add the class form-control to each control. The checkbox requires slightly different handling—we put it inside its own label which is inside an element with the class checkbox—and the submit tag will need some classes to make it look like a button.

Here's what our revised template looks like:

login/use-bootstrap/shine/app/views/devise/sessions/new.html.erb

```
<header>
<h1>Log in</h1>
</header>

<%= form_for(resource, as: resource_name,
                       url: session_path(resource_name)) do |f| %>
  <div class="form-group">
    <%= f.label :email %>
    <%= f.email_field :email, autofocus: true, class: "form-control" %>
  </div>

  <div class="form-group">
    <%= f.label :password %>
    <%= f.password_field :password, autocomplete: "off", class: "form-control" %>
  </div>
```

17. http://getbootstrap.com/css/#forms

```
  <% if devise_mapping.rememberable? -%>
    <div class="checkbox">
      <label>
        <%= f.check_box :remember_me %> Remember Me
      </label>
    </div>
  <% end -%>

  <%= f.submit "Log in", class: "btn btn-primary btn-lg" %>
<% end %>
<%= render "devise/shared/links" %>
```

If you reload your browser, the form now looks *a lot* better than before.

The spacing and "vertical rhythm" of the elements is more pleasing. The form controls feel more spacious and inviting. The "log in" button looks more clickable. You'll even notice a subtle animation and highlight when switching the active form field. And all we did was to add a few classes to our markup!

Before we move onto the registration screen, there's one more thing to fix here. If you submit the form without providing an email or password, you'll see the login form again and, up at the top, you'll see an error message: "Invalid email or password". It's hard to see.

This message is placed into the Rails flash[18] by Devise, and we should be styling it in a way that allows the user to more easily see it. This will help the user better understand when they've messed something up.

Styling The Flash

In addition to classes designed to work with existing HTML entities like forms, Bootstrap also provides "components" which are a set of classes that, when

18. http://guides.rubyonrails.org/action_controller_overview.html#the-flash

applied to an element (or set of elements), create a particular effect. For the flash, Bootstrap provides a component called an alert[19].

In our application's default layout file, app/views/layouts/application.html.erb, there are dummy placeholders for the flash messages.

```
<p class="notice"><%= notice %></p>
<p class="alert"><%= alert %></p>
```

Since there's no styling for the notice or alert class, this is why our flash messages look so bad. Let's recreate them using Bootstrap's alert component, which just requires using the class alert and then either alert-danger or alert-info, for the alert and notice flash messages, respectively.

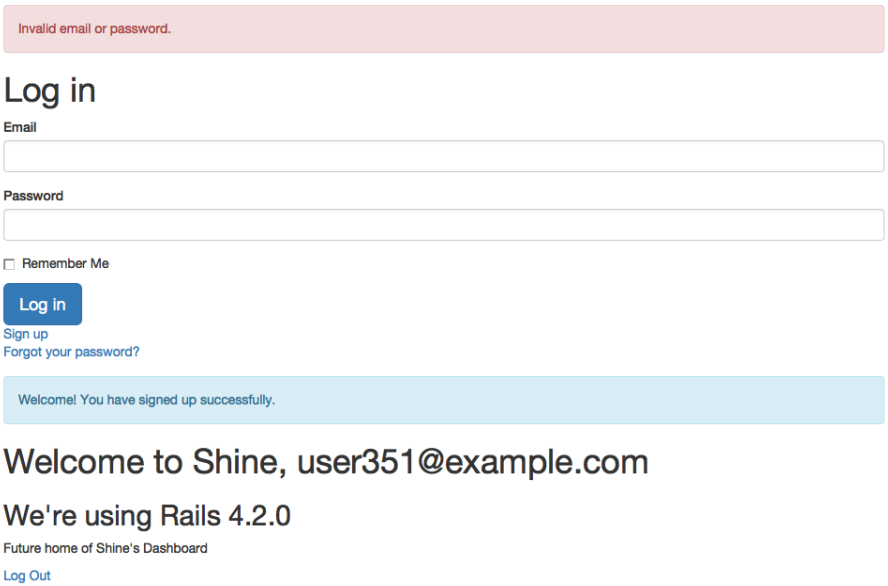
login/style-flash/shine/app/views/layouts/application.html.erb

```
<body class="container">
  <% if notice.present? %>
    <aside class="alert alert-info">
      <%= notice %>
    </aside>
  <% end %>
  <% if alert.present? %>
    <aside class="alert alert-danger">
      <%= alert %>
    </aside>
  <% end %>

<%= yield %>

</body>
```

Our markup did get a bit more complex. We're using aside instead of div as it's more semantically correct (Bootstrap doesn't generally care which type of element styles are applied to). We've also had to wrap each alert component in a check to see if that message was actually set. This is because even without content, the Bootstrap alert component will still show up visually and look strange.

With that done, we can navigate to a page requiring login, provide incorrect login details, and see that the flash messages are styled appropriately. The user can now easily see their mistakes, and understand their successes.

19. http://getbootstrap.com/components/#alerts



The only thing left to do is to style the registration page.

Styling the Registration Page

Devise refers to the resource for a user signing-up as a *registration*, so the registration form is located in app/views/devise/registrations/new.html.erb. We'll apply the same classes to this page that we did to the previous.

login/style-registration/shine/app/views/devise/registrations/new.html.erb

```
<h2>Sign up</h2>

<%= form_for(resource, as: resource_name,
                       url: registration_path(resource_name)) do |f| %>
  <%= devise_error_messages! %>

  <div class="form-group">
    <%= f.label :email %><br />
    <%= f.email_field :email, autofocus: true, class: "form-control" %>
  </div>

  <div class="form-group">
    <%= f.label :password %>
    <% if @validatable %>
    <em>(<%= @minimum_password_length %> characters minimum)</em>
    <% end %><br />
    <%= f.password_field :password, autocomplete: "off",
                                    class: "form-control" %>
  </div>

  <div class="form-group">
```

```
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation, autocomplete: "off",
                                                 class: "form-control" %>
  </div>

  <%= f.submit "Sign up", class: "btn btn-primary btn-lg" %>
<% end %>

<%= render "devise/shared/links" %>
```

Reload the page and navigate to the Sign Up screen. We can see it's now styled similarly to the login page.

Devise also provides screens for resetting your password and for editing your login details. We'll leave that as an exercise for you to style those pages, but it will be just as simple as what we've seen already.

We now have a secure login system that looks great, and we've hardly written any code at all. We still have a few login requirements left to implement which aren't provided by Devise by default. In the next section, we'll see how to configure Devise to meet these requirements.

Validating Registration

If we look at the `User` model that Devise created for us, we can see that a Devise-provided method named `devise` is being used. This is how you can control the behaviors Devise uses for registration and authentication on a per-model basis.

login/add-devise-validations/shine/app/models/user.rb

```
class User < ActiveRecord::Base
  devise :database_authenticatable,
         :registerable,
         :recoverable,
         :rememberable,
         :trackable,
         :validatable
```

```
end
```

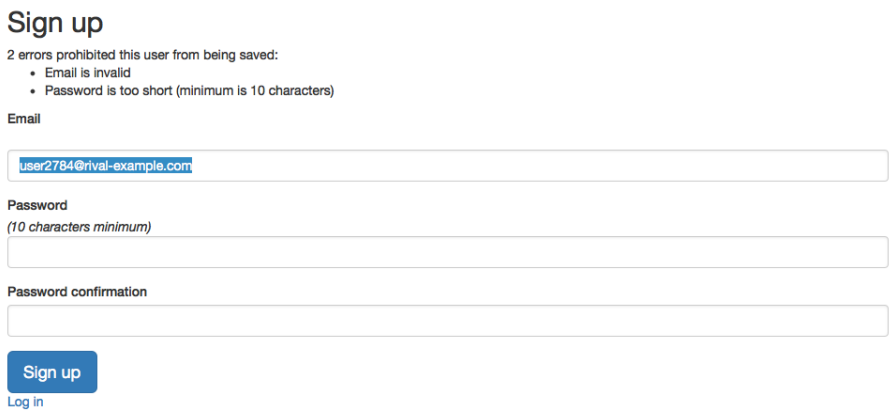
The :validatable *module* is what we're interested in. By using this in our model, Devise will set up various validations for the model, namely that the password is at least 8 characters and that the email address looks like an email address. These defaults are set in the initializer Devise created when we ran rails generate devise:install.

The initializer, located in config/initializers/devise.rb, has copious documentation about all of Devise's configuration options. If we search for the string "validatable", we can find the options we want to change, which are password_length and email_regexp. We'll change the minimum password length to 10 and require that emails end in our company's domain (which will be example.com)

login/add-devise-validations/shine/config/initializers/devise.rb

```
# ==> Configuration for :validatable
config.password_length = 10..128
config.email_regexp = /\A[^@]+@example.com\z/
```

Since we changed an initializer, we'll need to restart our server. Once we do that, if we try to register with a short password or an invalid email, we'll get an error message.



This covers the user experience and, because this is implemented using ActiveRecord validations, will also cover most typical codepaths that might change the user's email address. Most, but not all.

Next Up: Using Postgres to Make our Login More Secure

Devise cannot absolutely prevent users from getting into our database that do not meet our security criteria. For example, ActiveRecord provides the method update_attribute which skips validations and could be used to insert a

user with any email address into the USERS table. What we need is for the data layer itself to enforce our requirements.

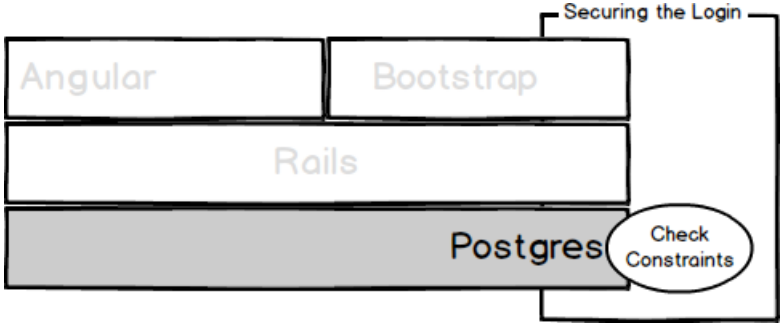
Postgres' *check constraints* can do this.

CHAPTER 3

Secure the Login Database with Postgres Constraints

Our registration and login system looks great, thanks to Bootstrap, and works great, thanks to Devise. But it's not as secure as we'd like. As you recall, we used validations to prevent users from registering with a non-company email address. Because this is done in Rails, it's easily circumventable using Rails' APIs or a direct database connection. Even something unintentional like bugs in our code could introduce vulnerabilities.

What we'd like is to prevent non-company email addresses getting into the database entirely. Most SQL databases do not have powerful features for preventing bad data. With Postgres, however, we can, by using a feature called *check constraints*. This chapter will be about setting up a check constraint for Postgres as part of our Rails database migrations.



Before we see how Postgres can solve our vulnerability, let's explore it briefly, so we know exactly what problem we're solving.

Exposing The Vulnerability Devise and Rails Leaves Open

You can easily verify the security hole in our application by creating a new user, signing out, changing that user's email in the database, and logging back in using the new email and previous password. This problem may seem academic, but it's more likely than you might think.

Even in a small company, there could be processes that access the database that aren't part of our application, and so won't benefit from the validations in our `User` model. Further, Rails itself provides methods like `update_attribute` which circumvent the validations, meaning a software bug could exist that used one of these methods and introduce a vulnerability.

How could this issue become a real problem? Consider a new employee named Sally. On Sally's first day, her company email wasn't set up properly, but she needed access to Shine. Sally was recruited by one of the engineers, Bob. Bob tries to help his friend Sally on her first day of work, and so creates a user for her using her personal email address, so she can start using Shine.

Months later, Sally leaves the company and Aaron in HR goes to deactivate her access to company systems. Aaron assumes that by de-activating Sally's email account, she won't have access to any more internal systems. Aaron doesn't know that Sally is using her personal email account to do that, so we are now in a situation where the company thinks Sally's access has been cut off, but it actually hasn't been.

Although this is all hypothetical, it now feels more possible than it might have seemed at first. When faced with security issues like this, you must weigh the cost of the security breach against the cost of preventing it. This means we need to figure out how much effort is required to prevent this vector of attack.

If preventing it required even a few days, it might not be worth it. Since we're using Postgres, it's a one-liner using *check constraints.*

Prevent Bad Data Using Check Constraints

If you've done any database work at all, you're no-doubt familiar with a "not null" constraint that prevents inserting `null` into a column of the database.

```
CREATE TABLE people (
  id        INT          NOT NULL,
  name      VARCHAR(255) NOT NULL,
  birthdate DATE             NULL
);
```

In this table,id and name may not be NULL, however birthdate may be. Postgres takes the "null constraint" concept *much* further by allowing arbitrary constraints on fields. Postgres also has support for regular expressions. This means we can create a constraint on our email field that requires its value to match the same regular expression we used in our Rails code. This would prevent non-company email addresses from being inserted into the table entirely.

First, we'll create a new migration where we can add this constraint.

```
> bundle exec rails g migration add-email-constraint-to-users
    invoke  active_record
    create    db/migrate/20150303133619_add_email_constraint_to_users.rb
```

The DSL for writing Rails migrations doesn't provide any means of creating this constraint, so we have to do it in straight SQL. Although Postgres *Data Definition Language* (DDL) looks different from what we normally use in our migrations, it's still relatively straightforward and well documented online[1].

The basic structure of our constraint is that we want to "alter" the USERS to "add" a constraint that will "check" the email column for invalid values. Here's what our migration will look like (See *Why aren't we using change in our Rails migrations?*, on page 44 for why we are using the older up and down methods).

login/add-postgres-constraint/shine/db/migrate/20150303133619_add_email_constraint_to_users.rb

```
class AddEmailConstraintToUsers < ActiveRecord::Migration
  def up
    execute %{
      ALTER TABLE
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '[A-Za-z0-9._%-]+@example.com' )
    }
  end
  def down
    execute %{
      ALTER TABLE
        users
      DROP CONSTRAINT
        email_must_be_company_email
    }
  end
end
```

1. http://www.postgresql.org/docs/9.4/static/ddl-constraints.html

The ~* operator is how Postgres does regular expression matching. Therefore this code means that the email column's value must match the regular expression we've given or the insert or update command will fail. The regular expression is more or less identical to the one we used when configuring Devise.

Why aren't we using change in our Rails migrations?

Rails 3.1 introduced the concept of *reversible migrations* via the method change in migrations DSL. The Rails authors realized that most implementations of down were to reverse what was done inside up and Rails could figure out how to reverse the code in the up method automatically.

In order to make this work, programmers would need to constrain the contents of the change method to only those migration methods that Rails knows how to reverse, which are itemized in ActiveRecord::Migration::CommandRecorder[a].

In most of the migrations we'll write in this book, we aren't using those methods, and are typically just using execute, because we need to run Postgres-specific commands. We could work within the Reversible Migrations framework by using reversible, but the resulting code is somewhat clunky:

```
class AddEmailConstraintToUsers < ActiveRecord::Migration
  def change
    reversible do |direction|
      direction.up {
        execute %{
          ...
        }
      }
      direction.down {
        execute %{
          ...
        }
      }
    end
  end
end
```

Since up and down aren't deprecated, it ends up being easier to stick with the older syntax for the types of migrations we'll be writing.

a. http://api.rubyonrails.org/classes/ActiveRecord/Migration/CommandRecorder.html

Let's see it in action. First we'll run our migrations (if you experience a problem doing this see *Migrations Failing Because of Existing Data?*, on page 45).

```
> bundle exec rake db:migrate
== 20150303133619 AddEmailConstraintToUsers: migrating ==================
-- execute("
      ALTER TABLE
```

```
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '[A-Za-z0-9._%-]+@example.com' )
      ;
    ")
   -> 0.0012s
== 20150303133619 AddEmailConstraintToUsers: migrated (0.0013s) ========
```

Migrations Failing Because of Existing Data?

If you ran the migrations and saw something like the error below, you'll need to do a bit more work to apply this change.

```
> bundle exec rails db:migrate
ActiveRecord::StatementInvalid: PG::CheckViolation: ERROR:
  check constraint "email_must_be_company_email" is violated by some row:
      ALTER TABLE
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '[A-Za-z0-9._%-]+@example.com' )
      ;
```

This means that at least one row in your development database has a value for the email column that violates our new constraint. Postgres is refusing to apply the constraint because it doesn't know what to do.

In your development environment, you can easily change or remove those rows that violate the constraint. If you were doing this to an active, production dataset, you would not have that luxury. You would need to get more creative. There are several ways of handling this.

- Create a migration that deletes all users using a bad email address. This is drastic, but would work.
- Create a migration to assign bogus company email addresses to the existing bad accounts. This would prevent those users logging in but maintain their history. You could correct the accounts manually later on, but the constraint would be satisfied.
- You could also do something more complex where you demarcate active users with a new field, and prevent inactive users from logging in. Your check constraint could then only check for active users, e.g. active = true AND email ~* '[A-Za-z0-9._%-]@example.com'.

In any case, if you are adding constraints to a running, production system, you'll have to be more careful.

With the migration applied, let's see how it works. First, we'll insert a user whose email is on our company's domain.

```
> bundle exec rails dbconsole
shine_development> INSERT INTO
                     users (
                       email,
                       encrypted_password,
                       created_at,
                       updated_at
                     )
                     VALUES (
                       'foo@example.com',
                       '$abcd',
                       now(),
                       now()
                     );
INSERT 0 1
```

This works as expected. Now let's try to insert a user using a *different* domain.

```
shine_development> INSERT INTO
                     users (
                       email,
                       encrypted_password,
                       created_at,
                       updated_at
                     )
                     VALUES (
                       'foo@bar.com',
                       '$abcd',
                       now(),
                       now()
                       );
ERROR:  new row for relation "users" violates
        check constraint "email_must_be_company_email"
DETAIL: Failing row contains (4,
                              foo@bar.com,
                              $abcd,
                              null,
                              null,
                              null,
                              0,
                              null,
                              null,
                              null,
                              null,
                              '2015-03-03:12:12:14.000',
                              '2015-03-03:12:12:14.000'0).
```

We can see that Postgres will refuse to allow invalid data into the table (and that we get a pretty useful error message as well). This means that a rogue application, bug in our code, or even a developer at a production console will

not be able to allow access to any user who doesn't have a company email address.

Given how little effort this was, and the peace of mind it gives us, it's a no-brainer to add this level of security. Postgres makes it simple, meaning the cost of securing our website is low.

There is one last thing we'll need to change, because we're using a feature that's Postgres-specific. By default, Rails stores a snapshot of the database schema in db/schema.rb, which is a Ruby source file using the DSL for Rails migrations. Rails creates this by examining the database schema and creating what is essentially a single migration, in Ruby, to create the schema from scratch. This is what tests uses to create a fresh database.

The problem is that Rails doesn't know about check constraints, so the one we just added won't be present in db/schema.rb. This is easily remedied by telling Rails to use SQL, rather than Ruby, for storing the schema. We can do this by adding one line to config/application.rb

login/add-postgres-constraint/shine/config/application.rb
```
config.active_record.schema_format = :sql
```

We'll then need to remove the old db/schema.rb file, create db/structure.sql by running migrations and finally reset our test database by dropping it and recreating it. We can do all this with rake.

```
> rm db/schema.rb
> bundle exec rake db:migrate
> RAILS_ENV=test bundle exec rake db:drop
> RAILS_ENV=test bundle exec rake db:create
```

Why Use Rails Validations?

Given the power that check constraints give us, why would we bother with Rails validations at all? The answer is part of why taking a full-stack view of development is so important—the user experience.

Rails validations are an elegant and powerful way to provide users a great experience when providing data via web forms. The validations API is incredibly expressive, configurable, and extensible. If you remove the validations from our code, and attempt to register with an invalid email, you'll get an exception—not a good user experience.

This *does* result in some duplication, which is chance for inconsistency to creep into our app, but we can fight this by writing tests for each part of the stack (which we'll do in Chapter 7, *Test This Fancy New Code*, on page 103).

Next Up: Using Postgres Indexes to Speed Up a Fuzzy Search

Our registration and login feature is now secure *and* pleasant to use. By using the best of both Postgres and Bootstrap, we've gotten a good taste of using the full application stack to deliver a great feature. The power of these tools allowed us to tackle an important part of any application—authentication—easily and quickly, without sacrificing security or user experience.

In the next chapter, we'll start on a new feature: customer search. Search is a great way to learn about all aspects of full stack development. It's got everything: user input, complex output, and complex database queries. We'll start this feature by implementing a naive fuzzy search that we can then examine and optimize using special indexes that Postgres provides.

CHAPTER 4

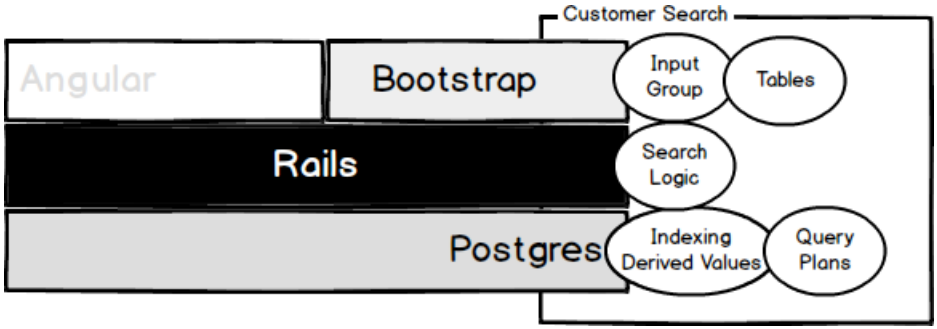
Use Fast Queries with Advanced Postgres Indexes

Our users can now securely log in to Shine using a well-designed login form, which was a great way to get a taste of what Postgres and Bootstrap can offer. In the next few chapters, we're going to implement a customer search feature, which will allow us to dig deeper into Postgres and Bootstrap, but also, in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81, AngularJS.

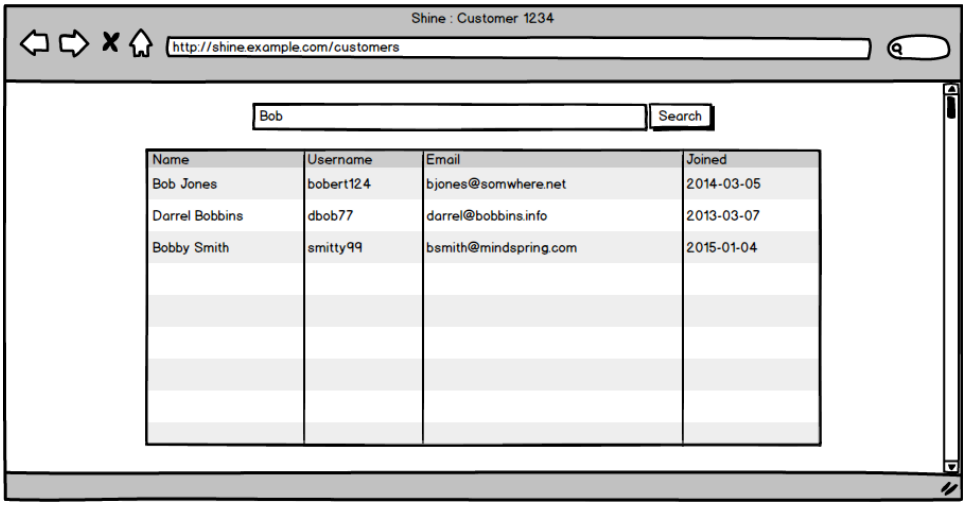
In this chapter, we'll implement the basics of our search, which will perform poorly. This will allow us to learn about the advanced indexing features of Postgres that will speed this search up without changing any code or setting up new infrastructure. We'll also learn how to understand query performance in Postgres, so we can be sure that the indexes we create have the performance improvements we want.

Because this is the first bit of code we're writing for this feature, we'll also need some UI and middleware logic. Although this chapter is mostly about Postgres, we'll be working in all parts of the stack.

Specifically, we'll write the basics of our search logic in Rails, and we'll learn how to style forms in Bootstrap using input groups, as well as how Bootstrap styles tables. In Chapter 5, *Create Clean Search Results with Bootstrap Components*, on page 69, we'll spend more time on the search results themselves.



First, we'll implement a naive fuzzy search that allows our users to locate customers based on first name, last name, or email. It'll look something like this:



After that, we'll examine the performance of the query ActiveRecord is executing by looking at Postgres' *query plan*. We'll then use a special type of index on our tables that should speed up our search, which we'll then verify by re-examining the query plan after adding the index. This all might feel really low-level, but you'll be surprised just how easy it is to get great performance out of Postgres with just a few lines of code.

Let's go to it by implementing a naive version of the search.

Implementing a Basic Fuzzy Search with Rails

As we mentioned in Chapter 1, *Introduction*, on page 1, Shine will be sharing a database with an existing customer-facing application. The customer-search feature we're building will search one of the tables in that database.

In a real-world scenario, the database we're going to share would already exist and we could just hook up to it directly. Since that's not the case here, we'll need to simulate its existence by creating the table in Shine's database. And, because we're going to learn about Postgres query performance optimization, our table is going to need a lot of data in it.

Setting up The New Table and Data

If we were actually using an existing table, we wouldn't need a migration—we could just create the Customer model and be done. That's not the case (since this is an example in a book), so we'll create the table ourselves. The table will ultimately look like this:

```
                                    Table "public.customers"
   Column   |            Type             |          Modifiers
------------+-----------------------------+-------------------------------
 id         | integer                     | not null default
                                          | nextval('customers_id_seq')
 first_name | character varying           | not null
 last_name  | character varying           | not null
 email      | character varying           | not null
 username   | character varying           | not null
 created_at | timestamp without time zone | not null
 updated_at | timestamp without time zone | not null
Indexes:
    "customers_pkey" PRIMARY KEY, btree (id)
    "index_customers_on_email" UNIQUE, btree (email)
    "index_customers_on_username" UNIQUE, btree (username)
```

We can see that a customer has a first and last name, an email address, a username, and two timestamp fields for creation date and last update date. None of the fields allow null and the data in the username and email fields must both be unique (if you haven't seen a database table description like this before, it's what the "index_customers_on_email" UNIQUE, btree (email) bit is telling us). This is more or less what we'd expect from a table that stores information about our customers.

Since this table doesn't exist yet in our example, we can recreate it using Rails' model generator. This will create both the database migration that will create this table as well as the Customer class that allows us to access it in our code.

```
> bundle exec rails g model customer first_name:string \
                         last_name:string \
                         email:string \
                         username:string
    invoke  active_record
```

```
    create    db/migrate/20150304140122_create_customers.rb
    create    app/models/customer.rb
```

The migration file Rails created will define our table, column names, and their types, but it won't include the not null or unique constraints. We can add those easily enough ourselves by opening up db/migrate/20150304140122_create_customers.rb.

search/setup-customer-data/shine/db/migrate/20150304140122_create_customers.rb

```
class CreateCustomers < ActiveRecord::Migration
  def change
    create_table :customers do |t|
➤     t.string :first_name, null: false
➤     t.string :last_name,  null: false
➤     t.string :email,      null: false
➤     t.string :username,   null: false

      t.timestamps null: false

    end
➤   add_index :customers, :email,    unique: true
➤   add_index :customers, :username, unique: true
  end
end
```

With that created, we can go ahead and run the migrations.

```
> bundle exec rake db:migrate
== 20150304140122 CreateCustomers: migrating ==============================
-- create_table(:customers)
   -> 0.0193s
-- add_index(:customers, :email, {:unique=>true})
   -> 0.0037s
-- add_index(:customers, :username, {:unique=>true})
   -> 0.0029s
== 20150304140122 CreateCustomers: migrated (0.0262s) =====================
```

Next, we'll need some customer data. Rather than provide you a download of a giant database dump to install (or include it in the code downloads for this book), we'll generate data algorithmically. We'll aim to make 350,000 rows of "real-looking" data. To help us, we'll use a gem called *faker*[1], which is typically used to create test data. First, we'll add that to our Gemfile.

search/setup-customer-data/shine/Gemfile

```
gem 'faker'
```

Then, we'll install it.

```
> bundle install
Installing faker 1.4.3
```

1. https://github.com/stympy/faker

We can now use faker to create real-looking data, which will make it much easier to use Shine in our development environment, since we'll have real-sounding names and email addresses. We'll create this data by writing a small script in db/seeds.rb. Rails seed data[2] feature is intended to pre-populate a fresh database with reference data, like a list of countries, but it'll work for creating our sample data.

search/setup-customer-data/shine/db/seeds.rb

```
350_000.times do |i|
  Customer.create!(
    first_name: Faker::Name.first_name,
     last_name: Faker::Name.last_name,
      username: "#{Faker::Internet.user_name}#{i}",
         email: "#{Faker::Internet.user_name}#{i}@#{Faker::Internet.domain_name}")
end
```

The reason we're appending the index to the username and email is to ensure these values are unique. Since we are adding unique constraints to those fields when creating the table, faker would have to have over 350,000 variations, selected with perfect random distribution. Rather than simply hope that's the case, we'll ensure uniqueness with a number. With our seed file created, we'll run it (this will take a while—possibly 30 minutes—depending on the power of your computer).

```
> bundle exec rake db:seed
```

With our table filled with data, we can now implement the basics of our search feature.

Building the Search UI

When starting a new feature, it's usually best to start from the user interface, especially if we didn't have a designer design it for us ahead of time. Recall that our requirements are to allow searching by first name, last name, and email address. Rather than require the user to specify which field they are searching by, we'll provide one search box and do an inclusive search of all fields on the back-end. We'll also display the results in a table, as that is fairly typical for search results.

The search will be implemented as the index action on the customers resource. First, we'll add a route to config/routes.rb.

search/search-ui/shine/config/routes.rb

```
resources :customers, only: [ :index ]
```

2. http://guides.rubyonrails.org/active_record_migrations.html#migrations-and-seed-data

That route will expect an index method on the CustomersController class, so let's create that next. We'll implement it to just grab the first 10 customers in the database, so we have some data we can use to style the view.

search/search-ui/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController
  def index
    @customers = Customer.all.limit(10)
  end
end
```

We can now start building the view in app/views/customers/index.html.erb. We'll need two main sections in our view: a search form, and a results table. We'll also want a header letting the user know what page they are on.

search/search-ui/shine/app/views/customers/index.html.erb

```
<header>
  <h1 class="h2">Customer Search</h1>
</header>
```

Next, we'll create the search form. Because there's just going to be one field, we don't need an explicit label (though we'll include markup for one that's only visible to screen readers). The design we want is a single row with both the field and the submit button, filling the horizontal width of the container (a large field will feel inviting and easy to use).

Bootstrap provides CSS for a component called an *input group*. An input group allows you to attach elements for form fields. In our case, we'll use it to attach the submit button to the right-side of the text field. This, along with the fact that Bootstrap styles input tags to have a width of 100% will give us what we want.

search/search-ui/shine/app/views/customers/index.html.erb

```
<section class="search-form">
  <%= form_for :customers, method: :get do |f| %>
    <div class="input-group input-group-lg">
      <%= label_tag :keywords, nil, class: "sr-only" %>
      <%= text_field_tag :keywords, nil,
            placeholder: "First Name, Last Name, or Email Address",
            class: "form-control input-lg" %>
      <span class="input-group-btn">
        <%= submit_tag "Find Customers",
              class: "btn btn-primary btn-lg" %>
      </span>
    </div>
  <% end %>
</section>
```

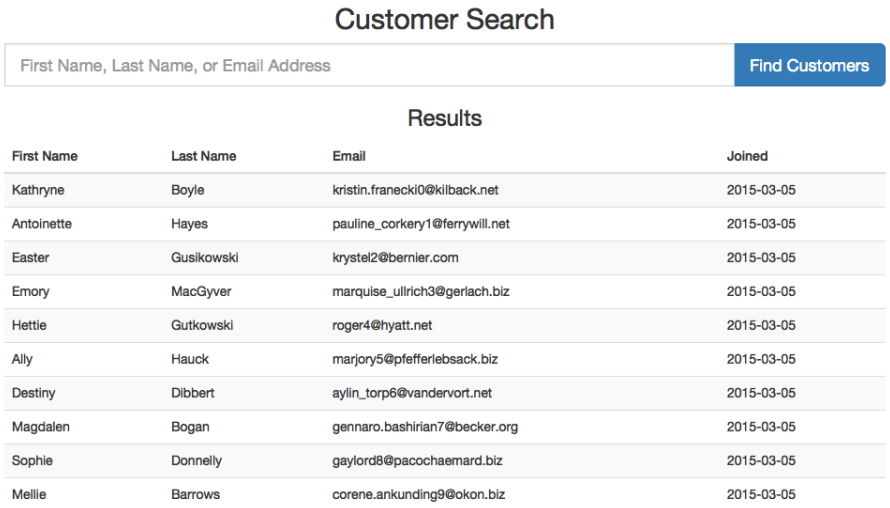
The sr-only class on our label is provided by Bootstrap and means "Screen Reader Only". You should use this on elements that are semantically "required" (like form labels) but that, for aesthetic purposes, you don't want to be visible. This makes your UI as inclusive as possible to users on all sorts of devices.

With our search form styled (we'll see what it looks like in a moment), we'll create a simple table for the results. Applying the class table to any table causes Bootstrap to style it appropriately for the table's contents. Adding the second class table-striped will create a "striped" effect where every other row has a light gray background. This can help users visually navigate a table with many rows.

search/search-ui/shine/app/views/customers/index.html.erb

```
<section class="search-results">
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <table class="table table-striped">
    <thead>
      <tr>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Email</th>
        <th>Joined</th>
      </tr>
    </thead>
    <tbody>
      <% @customers.each do |customer| %>
        <tr>
          <td><%= customer.first_name %></td>
          <td><%= customer.last_name %></td>
          <td><%= customer.email %></td>
          <td><%= l customer.created_at.to_date %></td>
        </tr>
      <% end %>
    </tbody>
  </table>
</section>
```

Now that we've got the table styled, we can see the entire view. It looks pretty good, and we still haven't had to write any actual CSS.



Now that we have our UI, all we need to do to implement our search is replace the implementation of `index` in `CustomersController` with the actual search.

Implementing the Search Logic

At a high level, our search should accept a string and do the right thing. Because our users are interacting with customers via email, we want to search by email, but because customers sometimes use multiple email addresses, we also want to search by first name and last name. To more strictly state our requirements:

- If the search term contains a "@" character, search email by that term.
- Use the *name* part of the email to search first and last name (e.g. we'd search for "bob" if given the term "bob123@example.com").
- If the search term does *not* contain a "@" character, don't search by email, but *do* search by first and last name.
- The search should be case-insensitive.
- The first and last name search should match names that start with the search term, so a search for "Bob" should match "Bobby".
- The results should be ordered so that exact email matches are listed first, and all other matches are sorted by last name.

This isn't the most amazing search algorithm, but it's sufficient for our purposes here, which is to implement the feature, but also demonstrate the performance problems present in an even moderately complex query.

There are two tricky things about the search we're running. The first is that we want case-insensitive matches, and ActiveRecord has no API to do that directly. The second is that we want exact email matches first. Fortunately, Postgres provides a means to do both of these things. We can use SQL like `lower(first_name) LIKE 'bob%'` and we can use complex expressions in the `order by` clause. Ultimately, we'll want a query that looks like so:

```
SELECT
  *
FROM
  customers
WHERE
  lower(first_name) LIKE 'bob%' OR
  lower(last_name)  LIKE 'bob%' OR
  lower(email)      =    'bob@example.com'
ORDER BY
  email = 'bob@example.com' DESC,
  last_name ASC
```

The `order by` in Postgres can take a wide variety of expressions. According to the documenation[3], it can "...be any expression that would be valid in the query's select list". In our case, we can order fields based on the results of matching the `email` column value to `bob@example.com`. This will evaluate to `true` or `false` for each row.

Because Postgres considers `false` less than `true`, an ascending sort would sort rows that *don't* match `bob@example.com` first, so we use `desc` to sort email matches first.

To execute this query using ActiveRecord, we'd need to write the following code:

```
Customer.where("lower(first_name) LIKE :first_name OR " +
               "lower(last_name)  LIKE :last_name  OR " +
               "lower(email)      =    :email", {
                 first_name: "bob%",
                 last_name: "bob%",
                 email: "bob@example.com"
               }).order("email = 'bob@example.com' desc, last_name asc")
```

Note that we are appending `%` to our name search term and using `like` so that we meet the "starts with" requirement. We have to do this because ActiveRecord has no direct API for doing this. To create this query in our code, let's create a class called `CustomerSearchTerm` that can parse `params[:keywords]` and produce the arguments we need to `where` and `order`.

3. http://www.postgresql.org/docs/9.4/static/queries-order.html

Our class will expose three attributes: where_clause, where_args, and order. These values will be different depending on the type of search being done. If the user's search term included a "@", we'll want to search the email column, in addition to last_name and first_name. If there's no "@", we'll just search first_name and last_name.

Let's assume that two private methods exist called build_for_email_search and build_for_name_search that will set the attributes appropriately, depending on the type of search as dictated by the search term. We'll see their implementation in a minute, but here's how we'll use them in the constructor of CustomerSearchTerm:

search/naive-search/shine/app/models/customer_search_term.rb

```
class CustomerSearchTerm
  attr_reader :where_clause, :where_args, :order
  def initialize(search_term)
    search_term = search_term.downcase
    @where_clause = ""
    @where_args = {}
    if search_term =~ /@/
      build_for_email_search(search_term)
    else
      build_for_name_search(search_term)
    end
  end
```

We're converting our term to lower case first, so that we don't have to do it later, and we're also initializing @where_clause and @where_args under the assumption that they will be modified by our private methods.

Let's implement build_for_name_search first. We'll create a helper method case_insensitive_search that will construct the SQL fragment we need and use that to build up @where_clause inside build_for_name_search. We'll also create a helper method called starts_with that handles appending the % to our search term.

search/naive-search/shine/app/models/customer_search_term.rb

```
def build_for_name_search(search_term)
  @where_clause << case_insensitive_search(:first_name)
  @where_args[:first_name]  = starts_with(search_term)

  @where_clause << " OR #{case_insensitive_search(:last_name)}"
  @where_args[:last_name] = starts_with(search_term)

  @order = "last_name asc"
end

def starts_with(search_term)
  search_term + "%"
```

```
end

def case_insensitive_search(field_name)
  "lower(#{field_name}) like :#{field_name}"
end
```

Next, we'll implement build_for_email_search, which is slightly more complex. Given a search term of "bob123@example.com" we want to use that exact term for the email part of our search, but we want rows where first_name or last_name starts with just "bob". To help us, we'll create a helper method called extract_name that uses regular expressions in gsub to remove everything after the @ as well as any digits.

search/naive-search/shine/app/models/customer_search_term.rb

```
def extract_name(email)
  email.gsub(/@.*$/,'').gsub(/[0-9]+/,'')
end
```

There's one last bit of complication, which is the ordering. To create the order by clause we want, it may seem we'd have to do something like this:

```
@order = "email = '#{search_term}' desc, last_name asc"
```

If building a SQL string like this concerns you, it should. Since search_term contains data provided by the user, it could create an attack vector via SQL Injection[4]. To prevent this, we need to SQL-escape search_term before we send it to Postgres for querying. ActiveRecord provides a method quote, available on ActiveRecord::Base's connection object.

Armed with this knowledge, as well as our helper method extract_name_from_email, we can now implement build_for_email_search.

search/naive-search/shine/app/models/customer_search_term.rb

```
def build_for_email_search(search_term)
  @where_clause << case_insensitive_search(:first_name)
  @where_args[:first_name] = starts_with(extract_name(search_term))

  @where_clause << " OR #{case_insensitive_search(:last_name)}"
  @where_args[:last_name] = starts_with(extract_name(search_term))

  @where_clause << " OR #{case_insensitive_search(:email)}"
  @where_args[:email] = search_term

  @order = "lower(email) = " +
    ActiveRecord::Base.connection.quote(search_term) +
    " desc, last_name asc"
end
```

4. http://en.wikipedia.org/wiki/SQL_injection

Note that we don't need to use quote when creating our SQL fragment in case_insensitive_search, because the strings involved there are from literals in our code and not user input. Therefore, we know they are safe.

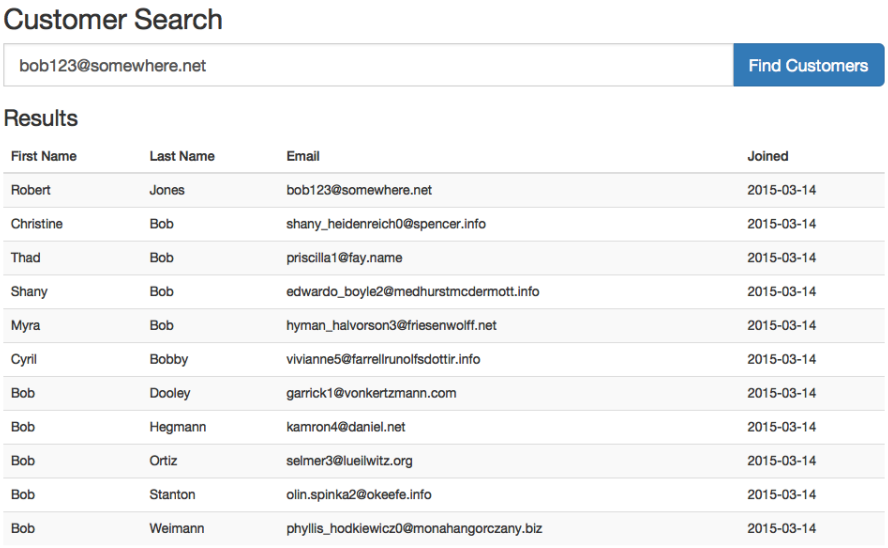
Now that CustomerSearchTerm is implemented, we can use it in CustomersController to implement the search.

search/naive-search/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController
  def index
    if params[:keywords].present?
      @keywords = params[:keywords]
      customer_search_term = CustomerSearchTerm.new(@keywords)
➤     @customers = Customer.where(
➤         customer_search_term.where_clause,
➤         customer_search_term.where_args).
➤       order(customer_search_term.order)
    else
      @customers = []
    end
  end
end
```

This may have seemed complex, but it's important to note that this search is quite simplified from what we might actually want. We might really need something more complex, for example a search term of "Bob Jones" would result in a first name search for "bob" and a last name search for "jones". The point is that our simplistic search is still too complex for ActiveRecord's API to handle. We had to create our own where clause and our own order by.

Now that our search is implemented, we can now see the results by starting our server (via rails server) and navigating to http://localhost:3000/customers. You'll see that the email match is listed first, while the remainder are sorted by last name.



Depending on the computer you are using, the search might seem fast enough. Or, it might seem a bit slow. If customers had more rows in it, or our database were under the real stress of production, the search might be unacceptably slow. It's popular to solve this problem by caching results in a NoSQL[5] database like Elasticsearch.

While there may be a case made for caching, Postgres gives more options than your average SQL database to speed up searches, which means we can get a lot more out of a straightforward implementation before complicating our architecture with additional data stores. In the next section, we'll learn about the powerful indexing features Postgres provides. We'll see that they are much more powerful than the indexes you get from most SQL databases.

Understanding Query Performance With the Query Plan

If you aren't familiar with database indexes, Wikipedia has a pretty good definition[6], but in essence, an index is a data structure created inside the database that speeds up query operations. Usually, databases use advanced data structures like B-trees to find the data you're looking for without examining every single row in a table.

If you *are* familiar with indexes, you might only be familiar with the type of indexes that can be created by ActiveRecord's Migrations API. This API provides

5. http://en.wikipedia.org/wiki/NoSQL
6. http://en.wikipedia.org/wiki/Database_index

a "lowest common denominator" approach. The best we can do is to create an index on last_name, first_name, and email. Doing so won't actually help us because of the search we are doing. We need to match values that *start* with the search term, ignore case.

Postgres allows much more sophisticated indexes to be created. To see how this helps, let's ask Postgres to tell us how our existing query will perform. This can be done by preceding a SQL statement with explain analyze. The output is somewhat opaque, but it's useful. We'll walk through it step-by-step.

```
> bundle exec rails dbconsole
shine_development=> EXPLAIN ANALYZE
                      SELECT
                        *
                      FROM
                        customers
                      WHERE
                        lower(first_name) like 'bob%' OR
                        lower(last_name)  like 'bob%' OR
                        lower(email)      = 'bob@example.com'
                      ORDER BY
                        email = 'bob@example.com' DESC,
                        last_name ASC ;

                        QUERY PLAN
------------------------------------------------------------------------------
1 Sort  (cost=15479.51..15494.00 rows=5797 width=79)
      (actual time=957.825..957.843 rows=234 loops=1)
   Sort Key: (((email)::text = 'bob@example.com'::text)), last_name
   Sort Method: quicksort  Memory: 57kB
2    -> Seq Scan on customers  (cost=0.00..15117.17 rows=5797 width=79)
                               (actual time=33.091..955.392 rows=234 loops=1)
3         Filter: ((lower((first_name)::text) ~~ 'bob%'::text) OR
                   (lower((last_name)::text) ~~ 'bob%'::text)  OR
                   (lower((email)::text) = 'bob@example.com'::text))
          Rows Removed by Filter: 388153
4  Total runtime: 957.945 ms
```

This gobbledegook is the *query plan* and is quite informative if you know how to interpret it. There are four parts to it that will help us understand how Postgres will execute our query.

❶ Here, Postgres is telling us that it's sorting the results, which makes sense since we are using an order by clause in our query. The details (e.g. cost=15479.51) are useful for fine-tuning queries, but we're not concerned with that right now. Just take from this that sorting is part of the query.

❷ This is this most important bit of information in *this* query plan. "Seq Scan on customers" means that Postgres has to examine every single row in the table to satisfy the query. This means that the bigger the table is, the more work Postgres has to do to search it. Queries that you run frequently should not require examining every row in the table for this reason.

❸ This shows us how Postgres has interpreted our `where` clause. It's more-or-less what was in our query, but Postgres has annotated it with the internal data types it's using to interpret the values.

❹ Finally, Postgres estimates the runtime of the query. In this case, it's almost a second. A second isn't much time to you or me, but to a database, it's an eternity.

Given all of this, it's clear that our query will perform poorly. It's likely that it performs poorly on our development machine, and will certainly not scale in a real-world scenario.

In most databases, because of the case-insensitive search and the use of `like`, there wouldn't be much we could do. Postgres, however, can create an index that accounts for this way of searching.

Indexing Derived and Partial Values

Postgres allows you to create an index on *transformed* values of a column. This means we can create an index on the lower-cased value for each of our three fields. Further, we can configure the index in a way that allows Postgres to optimize for the "starts with" search we are doing. The basic syntax is:

```
CREATE INDEX
  customers_lower_last_name
ON
  customers (lower(last_name) varchar_pattern_ops);
```

If you're familiar with creating indexes in general, the `varchar_pattern_ops` might look odd. This is a feature of Postgres called *operator classes*. Specifying an operator class isn't required, however the default operator class used by Postgres will only optimize the index for an exact match. Because we are using a `like` in our search, we need to use the non-standard operator class `varchar_pattern_ops`. You can read more about operator classes in Postgres' documentation[7].

Now that we've seen the SQL needed to create these indexes, we need to adapt them to a Rails migration. Rails doesn't provide a way to do this with

7. http://www.postgresql.org/docs/9.4/static/indexes-opclass.html

ActiveRecord's migrations API, but it *does* provide a method `execute` which will execute arbitrary SQL. Let's create the migration file using Rails' generator.

```
> bundle exec rails g migration add-lower-indexes-to-customers
    invoke  active_record
    create    db/migrate/20150308225243_add_lower_indexes_to_customers.rb
```

Next, we'll edit the migration to use `execute` to create our methods.

search/add-indexes/shine/db/migrate/20150308225243_add_lower_indexes_to_customers.rb

```
class AddLowerIndexesToCustomers < ActiveRecord::Migration
  def up
    execute %{
      CREATE INDEX
        customers_lower_last_name
      ON
        customers (lower(last_name) varchar_pattern_ops)
    }
    execute %{
      CREATE INDEX
        customers_lower_first_name
      ON
        customers (lower(first_name) varchar_pattern_ops)
    }
    execute %{
      CREATE INDEX
        customers_lower_email
      ON
        customers (lower(email))
    }
  end
  def down
    remove_index 'customers_lower_last_name'
    remove_index 'customers_lower_first_name'
    remove_index 'customers_lower_email'
  end
end
```

Note that we aren't using the operator class on the email index, since we'll always be doing an exact match. Sticking with the default operator class is recommended if we don't have a reason not to. Next, we'll run this migration (note that it may take over 30 seconds due to the volume of data being indexed).

```
> bundle exec rake db:migrate
== 20150308225243 AddLowerIndexesToCustomers: migrating =================
-- execute("CREATE INDEX
        customers_lower_last_name
      ON
        customers (lower(last_name) varchar_pattern_ops)
```

```
    ")
   -> 9.8050s
-- execute("CREATE INDEX
        customers_lower_first_name
      ON
        customers (lower(first_name) varchar_pattern_ops)
    ")
   -> 10.1730s
-- execute("CREATE INDEX
        customers_lower_email
      ON
        customers (lower(email))
    ")
   -> 13.5807s
== 20150308225243 AddLowerIndexesToCustomers: migrated (33.5590s) =======
```

Before we try our app, let's run the explain analyze again and see what it says. Note the highlighted lines.

```
> bundle exec rails dbconsole
shine_development=> EXPLAIN ANALYZE
                      SELECT
                        *
                      FROM
                        customers
                      WHERE
                        lower(first_name) = 'bob' OR
                        lower(last_name)  = 'bob' OR
                        lower(email)      = 'bob@example.com'
                      ORDER BY
                        email = 'bob@example.com' DESC,
                        last_name ASC
                      ;
                         QUERY PLAN
-----------------------------------------------------------------------------
Sort  (cost=6308.33..6322.83 rows=5797 width=79)
      (actual time=19.802..19.820 rows=234 loops=1)
 Sort Key: (((email)::text = 'bob@example.com'::text)), last_name
 Sort Method: quicksort  Memory: 57kB
  -> Bitmap Heap Scan on customers
       (cost=159.03..5945.99 rows=5797 width=79)
       (actual time=15.437..17.333 rows=234 loops=1)
   Recheck Cond: ((lower((first_name)::text) ~~ 'bob%'::text) OR
                  (lower((last_name)::text)  ~~ 'bob%'::text) OR
                  (lower((email)::text) = 'bob@example.com'::text))
     Filter: ((lower((first_name)::text) ~~ 'bob%'::text) OR
              (lower((last_name)::text)  ~~ 'bob%'::text) OR
              (lower((email)::text) = 'bob@example.com'::text))
     ->  BitmapOr  (cost=159.03..159.03 rows=5826 width=0)
                   (actual time=15.307..15.307 rows=0 loops=1)
         ->  Bitmap Index Scan on customers_lower_first_name
```

```
                  (cost=0.00..43.85 rows=1942 width=0)
                  (actual time=9.331..9.331 rows=234 loops=1)
               Index Cond: ((lower((first_name)::text) ~>=~ 'bob'::text) AND
                            (lower((first_name)::text) ~<~ 'boc'::text))
         ->  Bitmap Index Scan on customers_lower_last_name
                  (cost=0.00..43.85 rows=1942 width=0)
                  (actual time=4.851..4.851 rows=0 loops=1)
               Index Cond: ((lower((last_name)::text) ~>=~ 'bob'::text) AND
                            (lower((last_name)::text) ~<~ 'boc'::text))
         ->  Bitmap Index Scan on customers_lower_email
                  (cost=0.00..66.99 rows=1942 width=0)
                  (actual time=1.122..1.122 rows=0 loops=1)
                Index Cond: (lower((email)::text) = 'bob@example.com'::text)
Total runtime: 20.027 ms
```

This time, there is *more* gobbledegook, but if we look closely, Seq Scan on customers is gone, and we can see a lot of detail around our where clause. The highlighted lines indicate *index scans* which are contrates to the seq scan we saw before. And index scan is using our index and thus *not* examining each row in the table to find the correct results. We can see that it's doing three lookups, one for each field, using our indexes, and then or-ing the results together.

Setting aside the details of how Postgres does this, we can see that the results are about a 50x improvement—the query should complete in 20 milliseconds!

If we try our search in Shine now, the results come back almost instantly. We've improved the performance of our search by more than a factor of 50, all with just a few lines of SQL in a migration. *And* we didn't have to change a line of code in our Rails application. Can you imagine the work involved in setting up elasticsearch or some other system to make this search faster?

This sort of index is just the tip of the iceberg with some of the advanced features inside Postgres. Fortunately, Rails makes it easy for us to use these features via execute, even if they aren't baked directly into the ActiveRecord API.

With our search now performing better, let's take a final pass at the user interface. Bootstrap's default table styling made it a snap to create a reasonable user interface in no time. This then enabled us to focus on the Rails application's behavior and performance. If we stopped now, and shipped what we have, we'd be shipping a feature we could be proud of. But, since we haven't spent *that* much time on this feature, let's see if there's any way to make the UI better for our users.

Next Up: Better-looking Results with Bootstrap's List Group

We've got a solid back-end going for our search. It's now really fast and we didn't have to do anything other than add a few indexes to our database. The user interface actually isn't too bad, either, considering we didn't spend much time on it. But, it could be better.

The next chapter will bring us back to the front-end, as we redesign the results. We'll see that Bootstrap's many helper classes and components can make it easy to try out new designs. This means we can provide better software for our users without investing huge amounts of time in writing CSS.

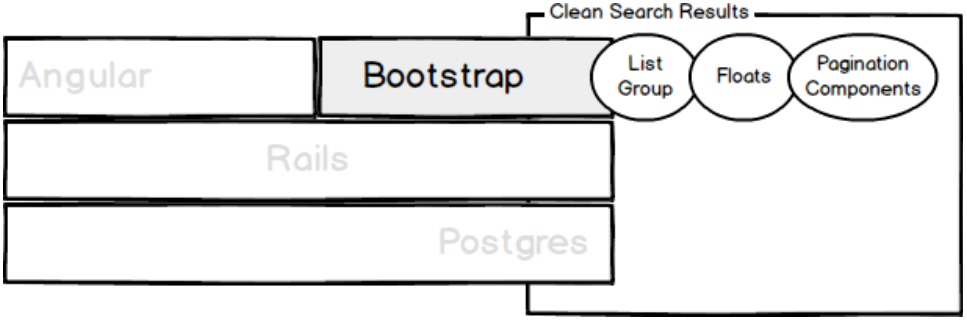
CHAPTER 5

Create Clean Search Results with Bootstrap Components

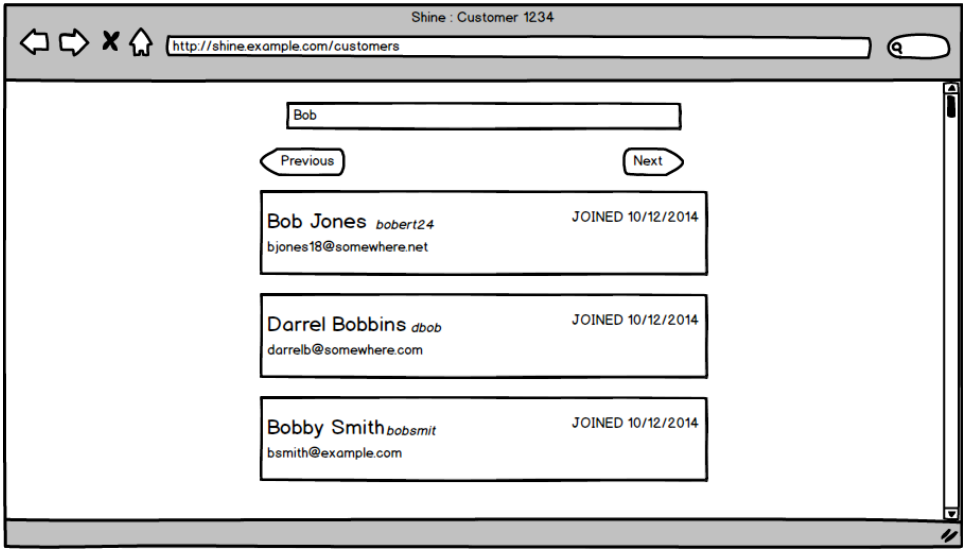
The customer search we've implemented is using tables to display the results. Tables are a common design component to use, but they aren't always the best. The reason they are so common isn't because the results are necessarily tabular, but because tables look decent by default. Everything lines up reasonably well, and the rows and columns nature of tables tends to be usable by default.

Bootstrap provides us a lot more options than tables to get great-looking results. That's what we'll learn in this chapter. We're going to re-style our results to do two things: first, we're going to get rid of the tables and create a more Google-like result that formats each customer in a *component* style, rather than as a row in a table.

This will demonstrate how easy it is to build out a seemingly complex design by using Bootstrap's *list group*, its typographic styles, and CSS floats. We're also going to paginate the results using one of Bootstrap's custom components. And we aren't going to write *any* CSS.



The UI we'll build will look like so:



Let's start by removing the table and replacing it with component-based results.

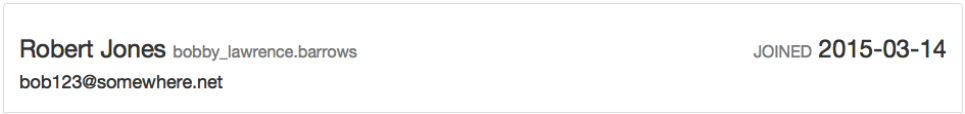
Creating Google-style Search Results without Tables

If we were styling this application on our own, the prospect of a customized, non-tabular search results page (like Google's, for example) would not be very appealing. We'd need to figure out the layout, design, and CSS to get it just right. Since we're always under pressure to ship our software and move onto the next feature, we might not be able to spend the time to give the user a better experience.

Bootstrap provides many components that make it easy to at least *try* something different, without a huge time investment. Let's try using the *list group* component. This component renders information in a list, but allows us to format what's in each list item with more flexibility than a table.

You'll recall the original motivation for this feature—users want to search customers by name or email to see if they signed up before a certain date. That means that the sign-up date is fairly important. It's also worth considering that our users will be getting emails from customers that will likely contain their full name written out, such as `Bob Jones`. Finally, our users might include their usernames in their email.

Perhaps the user interface would be better served with a mini component for each user, instead of a row in a table?



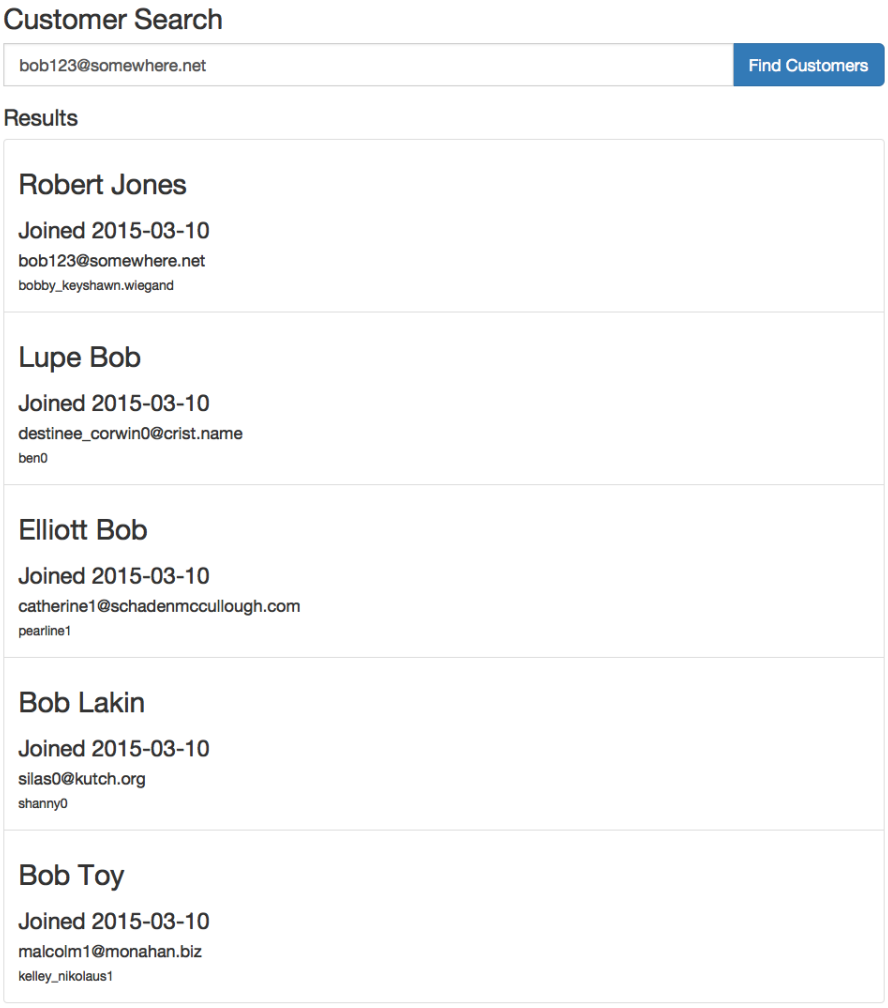
Unless you write a lot of CSS, this layout will appear to be somewhat tricky to pull off, especially if you consider how well-aligned all the sub-components are. Fortunately, Bootstrap makes this simple. We'll use three features of Bootstrap to do this: the list group component, the behavior of typography inside a small HTML element, and some floating-element helper classes.

The list group component styles a list of elements so that the contents of each element are set inside a bordered box, with appropriate spacing and padding to work well in a list of similar elements. To use it, we'll replace our table with an ordered list that has the class list-group and give each list item the class list-group-item. Inside each list element, we'll put each bit of information inside the appropriate "H" element, based on how important it is to the task at hand.

search/simple-list-group/shine/app/views/customers/index.html.erb

```
<section class="search-results">
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <ol class="list-group">
    <% @customers.each do |customer| %>
      <li class="list-group-item">
        <h2><%= customer.first_name %> <%= customer.last_name %></h2>
        <h3>Joined <%= l customer.created_at.to_date %></h3>
        <h4><%= customer.email %></h4>
        <h5><%= customer.username %></h5>
      </li>
    <% end %>
  </ol>
</section>
```

The results are a bit mixed, but we can see our design starting to form, since Bootstrap's list group does a reasonable job formatting the information.



Next, we want to change the position of the elements to match our earlier design. The main challenge is getting the customer's join date aligned to the right side of the component. To do that, we'll use some helper classes Bootstrap provides for floats.

Floats in CSS are a way to shift content to the right or left and allow other content to flow around it. For example, if we float some content to the left, the markup that follows that float will render to the right of the floated content. Floats are the basis of many advanced layout techniques in CSS.

Getting this working can be tricky, especially if you aren't familiar with how floats behave in various contexts. Bootstrap provides two classes that we'll

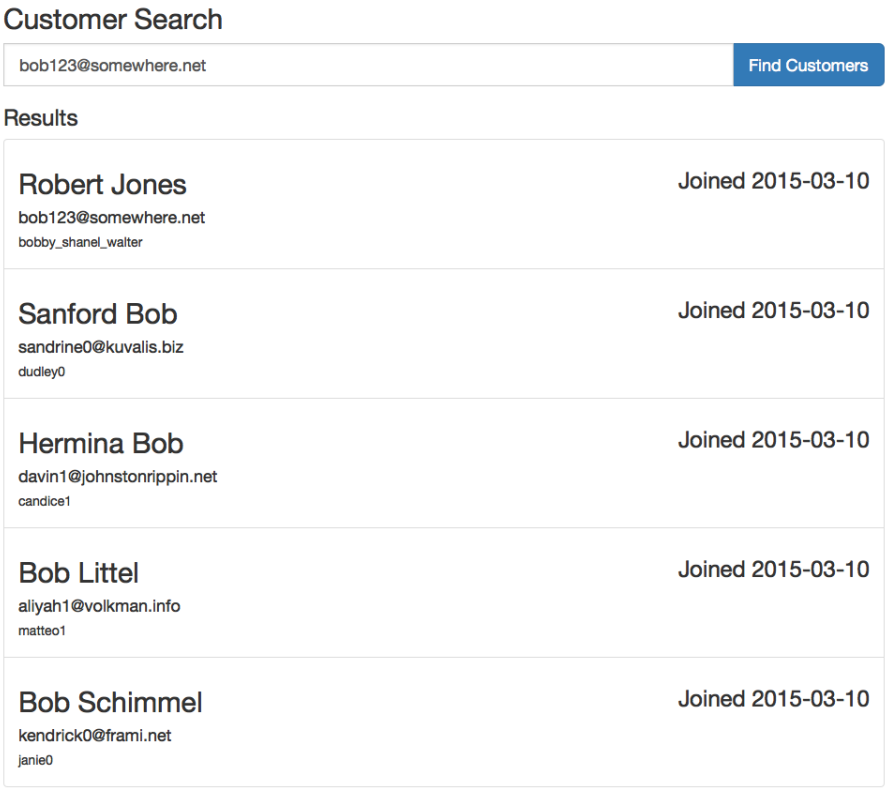
use to help achieve our design: pull-right, and clearfix (there's also a pull-left, but we don't need it for this design).

search/list-group-positioned/shine/app/views/customers/index.html.erb

```
<section class="search-results">
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <ol class="list-group">
    <% @customers.each do |customer| %>
      <li class="list-group-item clearfix">
        <h3 class="pull-right">Joined <%= l customer.created_at.to_date %></h3>
        <h2><%= customer.first_name %> <%= customer.last_name %></h2>
        <h4><%= customer.email %></h4>
        <h5><%= customer.username %></h5>
      </li>
    <% end %>
  </ol>
</section>
```

We moved the h3 that contains the join date above the h2 containing the customer's name because we want the name to flow to the left of the joined date. If we kept the join date in its original position, only the username and email would flow to the left.

The clearfix class is provided by Bootstrap to "reset" the floats. Because of the way floats are implemented, our page will explode to the right if we don't reset them (it's hard to explain, but try removing the clearfix class and see what happens). Now, our design is pretty close to what we want to achieve.



The last thing we need to do is to reduce the visual weight of both the username as well as the label "Joined". This allows the other information to be highlighted in a subtle way. To do that, we'll put both elements inside `small` tags which will trigger alternate typography from Bootstrap. We'll also use the class `text-uppercase` on the "Joined" label so that it has a subtle, yet distinct visual appearance from the more dynamic parts of our component.

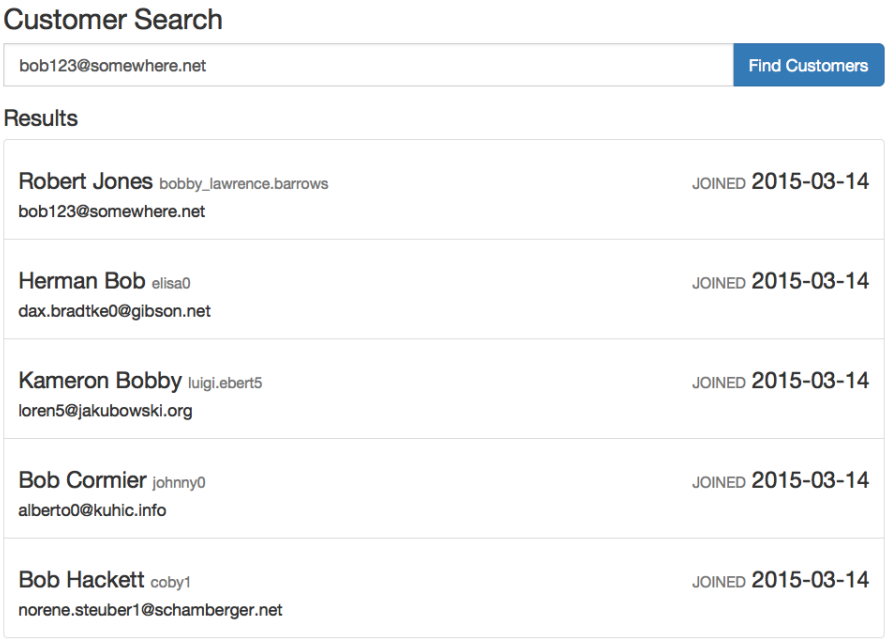
search/better-ui/shine/app/views/customers/index.html.erb

```
<section class="search-results">
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <ol class="list-group">
    <% @customers.each do |customer| %>
      <li class="list-group-item clearfix">
        <h3 class="pull-right">
          <small class="text-uppercase">Joined</small>
          <%= l customer.created_at.to_date %>
        </h3>
        <h2 class="h3">
          <%= customer.first_name %> <%= customer.last_name %>
```

```
            <small><%= customer.username %></small>
          </h2>
          <h4><%= customer.email %></h4>
        </li>
      <% end %>
    </ol>
</section>
```

Note that we've also added the `h3` class to the user's name. This will render it visually identical to the `h3` containing the join date. Doing this will ensure that both elements' text is horizontally aligned properly. It's a subtle difference, but polish like this will make Shine *feel* better to its users.

Now, the search looks pretty darn good!



This layout was much trickier than we've seen before, but Bootstrap made it simple to achieve, and we *still* haven't written any CSS. This is the power of a CSS framework like Bootstrap: if you have a design in mind, even if you just want to quickly try it out, Bootstrap provides a lot of tools, at every level of abstraction, to implement it.

There's one last thing we should take care of before moving on. Ordinary searches are returning a *lot* of results. This could be because our fake data only had so many fake usernames to choose from, but even in a real data set, common names could generate more results than a user will want to scroll

through. Let's paginate the results, so the user can see only ten results at at time, under the assumption the result they want is in the first ten.

Paginating the Results using Bootstrap's Components

Adding pagination can be done in just two steps: adjusting the query to find the right "page", and adding pagination controls to the view. There are several RubyGems out there that can help us, but it's actually not that much code to just do it ourselves. Since we'll be porting our view over to Angular in the next chapter, there's little benefit to integrating a gem at this point.

We'll take it one step at a time. First, we'll adjust the controller to handle pagination.

Handling Pagination in the Controller

For simplicity, we'll hard-code the size of a page to 10 results, and look for a new parameter, `:page` that indicates which page the user wants, with a default of 0.

search/pagination/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController
  PAGE_SIZE = 10

  def index
    @page = (params[:page] || 0).to_i

    # ...

  end
end
```

Next, we'll use both `PAGE_SIZE` and `@page` to construct parameters to ActiveRecord's `offset` and `limit` methods. Since our results are sorted, we can rely on these two methods to allow us to reliably page through the results without the order changing between pages.

search/pagination/shine/app/controllers/customers_controller.rb

```
@customers = Customer.where(
    customer_search_term.where_clause,
    customer_search_term.where_args).
  order(customer_search_term.order).
➤ offset(PAGE_SIZE * @page).limit(PAGE_SIZE)
```

That's all there is to our controller. Now, we'll adjust the view to allow paging.

Adding Pagination Controls to the View

To keep things simple, we'll go with a previous/next style of pagination. This means we'll need two links on the page, which we can create by adding or subtracting 1 to `@page` and passing that to the Rails-provided `customers_path` helper.

To style the links, Bootstrap provides a component we can use, called a *pager*. Let's set it up in a partial, which we'll then use to place the pager before *and* after the results list (this allows the user to always have the pager handy). We've highlighted the markup and classes Bootstrap requires to style the pager. Pay special attention to `disabled`, which will give our Previous button a disabled look if we're on the first page.

search/pagination/shine/app/views/customers/_pager.html.erb

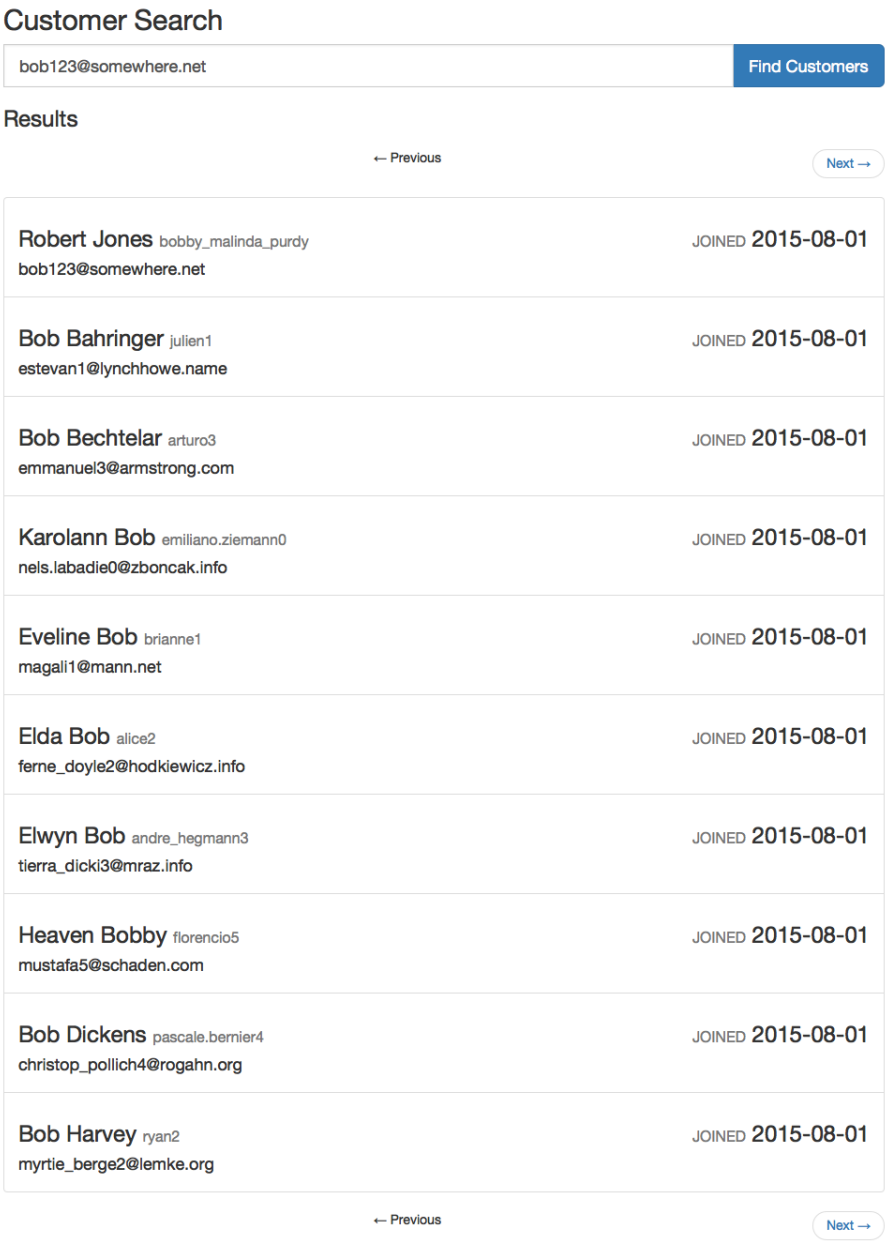
```
<nav>
  <ul class="pager">
    <li class="previous <%= page == 0 ? 'disabled' : '' %>">
      <%= link_to_if page > 0,
                  "&larr; Previous".html_safe,
                  customers_path(keywords: keywords, page: page - 1) %>
    </li>
    <li class="next">
      <%= link_to "Next &rarr;".html_safe,
                  customers_path(keywords: keywords, page: page + 1) %>
    </li>
  </ul>
</nav>
```

Now, we'll include the partial in `app/views/customers/index.html.erb`.

search/pagination/shine/app/views/customers/index.html.erb

```
<section class="search-results">
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <%= render partial: "pager",
             locals: { keywords: @keywords, page: @page } %>

  <ol class="list-group">

    <!-- ... -->

  </ol>
  <%= render partial: "pager",
             locals: { keywords: @keywords, page: @page } %>
</section>
```

If we start our server and search, we'll now only see 10 results, and our pager controllers work to allow us to step through them.



Because of our indexes, Postgres' powerful implementation of `order by`, and Bootstrap's pre-made components, we were able to add performant pagination in just a few lines of code.

Next Up: Angular!

In the next chapter, we'll add a third tool to our toolbox—AngularJS. AngularJS is a full-fledged Model-View-Controller (MVC) framework for JavaScript. Unlike libraries like jQuery, Angular provides a higher level of abstraction for designing interactive user interfaces.

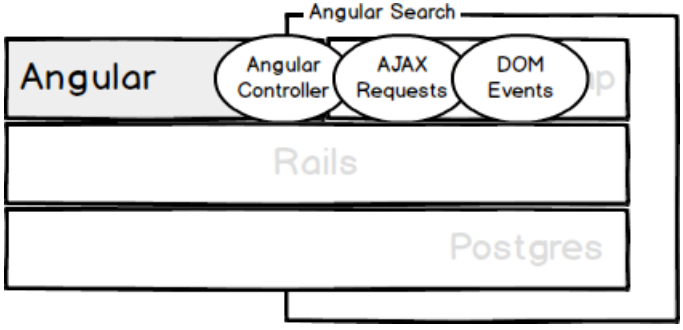
Even though Angular might feel heavyweight for the features Shine currently has, it's not. Angular can be applied lightly, on a screen-by-screen basis, to make interactive behavior far easier than it would using jQuery. Angular also scales with the complexity of your views—where your jQuery code would start to get messy, Angular keeps things simple.

Getting Angular working with Rails requires a bit more setup than simply installing a gem, so in the next chapter, we'll set up Angular and learn how it works by implementing a "typeahead" search. Instead of requiring our users to type a search term and click a search button, we'll fetch results in real-time, as they type. Because our search is so fast now—thanks to our Postgres-specific indexes—the user interface will feel snappy, and the code that powers it will be clean, clear, and maintainable.

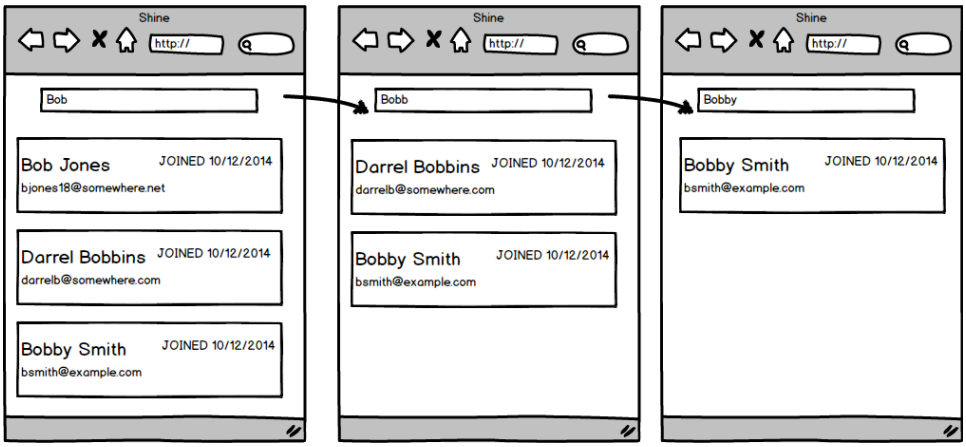
CHAPTER 6

Build a Dynamic UI with AngularJS

We've seen some of what Postgres and Bootstrap can do, but our user interface is still fairly static. It's now time to see what AngularJS is all about and how it can improve the experience for our users.



To do this, we're going to rework our search feature so that the system searches as the user types, dynamically changing the results as they type out a user's name. For example, if a user wants to find a customer named Bobby Smith, a search for just `Bob` may return more results than needed. The user would have to search again with a more refined query. If the results were visible while typing, the user could simply type `Bobby` and potentially get the desired record right away, without having to wait for the browser to rerender the view.



Angular makes it easy to implement this feature, although there's a fair bit of one-time setup we have to do to get there. That's what this chapter is about. We'll learn how to install and set up Angular in our Rails application. We'll use it to do an asynchronous search as the user types, by listening for the right DOM events and using Angular's AJAX features.

First, we'll get Angular set up in our application, validating that setup with a simple Angular *app*. We'll then use Angular to power our existing search feature, maintaining the existing user experience of entering a search term and clicking a button. Finally, we'll remove the button and make the search happen while the user is typing.

"App" vs. "Application"

One thing to keep in mind is that an Angular *app* is the front-end code and templates, whereas a Rails *application* refers to our entire Rails codebase, in this case Shine. Put another way, our Rails application can (and will) power multiple Angular apps. We'll stick to this convention throughout the book.

Configuring Rails and Angular

To start using Angular with Rails, we mostly just need to install Angular using Bower, and arrange for Angular's code to be served up by the asset pipeline. First, we'll add Angular to `Bowerfile`.

typeahead/install-angular/shine/Bowerfile

```
asset 'bootstrap-sass-official'
asset 'angular', '~> 1.4'
```

Next, we'll install it using the Rake task provided by bower-rails.

```
> bundle exec rake bower:install
```

```
bower.js files generated
/usr/local/share/npm/bin/bower install -p

bower angular#1.4.3  not-cached git://github.com/angular/...
bower angular#1.4.3     resolve git://github.com/angular/...
bower angular#1.4.3    download https://github.com/angul...
bower angular#1.4.3     extract archive.tar.gz
bower angular#1.4.3    resolved git://github.com/angular...
bower angular#1.4.3     install angular#1.4.3
```

With Angular installed, we'll need to bring it into the asset pipeline by adding it to app/assets/javascripts/application.js. This file is a manifest for all JavaScript in our application, just like app/assets/stylesheets/application.css (which we modified in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17 to bring in Bootstrap) is a manifest for CSS.

typeahead/install-angular/shine/app/assets/javascripts/application.js

```
//= require jquery
//= require jquery_ujs
//= require angular
//= require_tree .
```

As we mentioned, we want to validate this configuration is working before we get into really learning Angular. I find when learning a new technology, it's best to take very small steps to ensure that the setup and configuration is working, so if it isn't, I can easily narrow down where the problem is. In this case, we'll create the world's simplest Angular application. It will require a four line controller, a simple HTML template, and exactly one line of JavaScript. But, it will do the job: verifying that Angular is working in our application.

First up is angular_test, a basic controller and view. We add the route to config/routes.rb:

typeahead/install-angular/shine/config/routes.rb

```
get "angular_test", to: "angular_test#index"
```

Then we create AngularTestController to serve up the index action.

typeahead/install-angular/shine/app/controllers/angular_test_controller.rb

```
class AngularTestController < ApplicationController
  def index
  end
end
```

Next, we write our ERB template with certain custom HTML attributes to *activate* Angular. These attributes bind the value of our header text to the value inside an input field. Because of this binding, Angular will update the

header's value as we change the text in the input field. Don't worry too much about *how* this works for the moment—we're just testing our configuration.

Our template, in app/views/angular_test/index.html.erb looks like this:

typeahead/install-angular/shine/app/views/angular_test/index.html.erb

```
<article ng-app="angular_test">
  <header>
    <h1 ng-if="name">Hello, {{name}}</h1>
  </header>
  <section>
    <form class="form-inline">
      <div class="form-group">
        <label for="name">Name</label>
        <input class="form-control"
               name="name"
               type="text"
               placeholder="Enter your name"
               autofocus
               ng-model="name">
      </div>
    </form>
  </section>
</article>
```

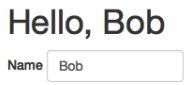
We'll explain more later, but the existence of the ng-app attribute will *Angularize* this markup. The app name we gave to that attribute, angular_test, must be defined somewhere. We'll create the file app/assets/javascripts/angular_test_app.js to define it (remember that the default configuration of Rails will pick up all files in app/assets/javascripts, so there's no need to add this file to application.js). Angular provides the function module that allows us to declare our app.

typeahead/install-angular/shine/app/assets/javascripts/angular_test_app.js

```
angular.module('angular_test',[ ]);
```

Now, let's start Shine with rails server and navigate to http://localhost:3000/angular_test. We should see something like this:

If we type a name, like Bob, into the text field, the header should appear and update live.



This verifies that Angular has been installed and is working with our Rails application. This tiny app that we've created also gives us a sneak peak into how our typeahead might work. You'll notice that we didn't write any real

code to make this work—Angular is detecting our keypresses and updating our UI automatically. We'll learn how that works when we implement the typeahead feature but, for now, let's re-implement our existing search as an Angular app.

Porting our Search to Angular

Although Angular works best when we have a dynamic user interface, it's often easier to introduce new technology by using it to solve an existing problem. That way, you aren't wrestling to understand both the new feature and technology. To that end, we'll rewrite our existing search feature in Shine using Angular. As before, the user will still type in a keyword and hit a submit button to perform the search. The difference is that the search will be powered by Angular, not by a browser submitting a form.

We'll write JavaScript code that grabs the search term the user entered, submits an AJAX request to the server, receives a JSON response, and updates the DOM with the results of the search, all without the page reloading.

This hopefully feels straightforward, at least conceptually. Again, this isn't the best demonstration of Angular's power, but it's simple enough to get our feet wet with some of Angular's concepts. We'll need this grounding to see more powerful features later. So, despite how simple this example seems, we're going to take things step-by-step.

First, we'll *Angularize* our view by removing all the Rails helpers and adding some Angular-specific attributes (similar to the ones we used when validating our configuration in the previous section). Next, we'll write some JavaScript that shows us how to respond to a click event. Then, we'll update that JavaScript to put canned data into the view when the user does a search—this will demonstrate how we can manipulate the DOM. Finally, we'll change our code to get real results from the server by making an AJAX call.

Angularizing Our View

When we were verifying that Angular had been installed and configured properly, we added the `ng-app` attribute to our HTML. In Angular parlance, `ng-app` is called a *directive*. Even though it's an attribute on a DOM element, using *directive* allows us to distinguish between special attributes that control our Angular app from plain attributes that are part of HTML.

The presence of this directive tells Angular to start managing the markup of all child elements of the element that contains the `ng-app` directive. *Managing* means two things. First, Angular creates a root scope to hold any data and

code that we write. Second, it compiles the DOM inside the element containing the ng-app directive (this is analogous to how Rails processes our ERB templates).

To make our existing search form ready to be used with Angular, we'll first need to wrap it in an element where we can place ng-app. Unlike the situation with our test app, *this* app will need some substantive code. Angular expects that code to be in a *controller*, which is similar to a Rails controller.

Unlike Rails—which can derive the name of the controller from a route—Angular requires that you explicitly name the controller you want to use. That can be done using the ng-controller directive. This directive will also be placed on the element we're using to wrap our markup.

We'll use an article element to wrap the markup in app/views/customers/index.html.erb and to hold the ng-app and ng-controller directives.

```
<article ng-app="customers" ng-controller="CustomerSearchController">
  <!-- rest of the existing markup -->
</article>
```

Before we see how to define CustomerSearchController, let's flesh out the rest of the search form. Previously, we were using the various Rails form helpers. Those wont' work with our Angular template, so we'll need convert the form back to HTML. There's a few new Angular directives in there which we'll explain in a minute, but let's see the code first.

typeahead/port-search-front-end/shine/app/views/customers/index.html.erb

```
<section class="search-form">
  <form>
    <div class="input-group input-group-lg">
      <label for="keywords" class="sr-only">Keywords</label>
      <input type="text"
             name="keywords"
             class="form-control input-lg"
             placeholder="First Name, Last Name, or Email Address"
             ng-model="keywords">
      <span class="input-group-btn">
        <button class="btn btn-primary btn-lg"
                ng-click="search(keywords)">
          Find Customers
        </button>
      </span>
    </div>
  </form>
</section>
```

You'll notice we're using ng-model on the input field. We saw that directive previously in our simple test app. This directive tells Angular to *bind* the symbol we've provided—in this case keywords—to the value of the form field. This variable lives in the scope Angular creates that we mentioned earlier (we'll see it manifested as an object later on).

The other directive we're using is one we haven't seen, called ng-click. Like ng-model it binds a DOM element to an object in code. In this case, however, it's binding a function and not a variable. Using ng-click like this tells Angular that when the user clicks the button, call the function named search that is defined in the scope that was set up for us passing the current value of keywords to it.

Now that we have our view ready for our Angular app, we need to define CustomerSearchController, which will contain an implementation of search.

Creating Our First Angular Controller

To keep things as simple as possible, we'll write just enough code to see something working. Our aim is to implement the search function so that it records the user's search term in a variable that we'll expose in the view. This way, we can see how to set up our controller and get it working, without worrying too much about the specifics of the search feature itself.

First, we'll need to define our Angular app using angular.module, much like we did before when we created our test app. We'll need to use the same string we gave to ng-app in our view (customers) and we'll put the code app/assets/javascripts/customers_app.js (see *Why Aren't We Using CoffeeScript?*, on page 88 for a brief detour explaining why we aren't using CoffeeScript). Our new file will be automatically picked up by the asset pipeline, due to being in app/assets/javascripts.

```
var app = angular.module('customers',[]);
```

Next we'll define our controller. Since JavaScript has no real concept of classes, Angular models everything as a function. Thus, our controller will be a function. Angular will call this function one time when bootstrapping the Angular app. Angular will pass the *scope* we've been referring to as an argument to our controller function.

That scope, is a Scope[1] and we'll name the argument $scope. Angular-provided objects are usually pre-pended with a $. We can then set properties on this object (both data and functions), and those properties will be available to the view.

1. https://docs.angularjs.org/api/ng/type/$rootScope.Scope

Why Aren't We Using CoffeeScript?

CoffeeScript[a] is the default setting in Rails for writing front-end code. If you aren't familiar with it, CoffeeScript is a language that can be translated into JavaScript. The theory is that CoffeeScript is easier to read and write, and makes it harder to make common mistakes developers make using JavaScript.

CoffeeScript is a fine language, and you can absolutely write your Angular apps using it. We aren't using it in this book for a couple of reasons. First, when you search online for help with Angular, you will almost always see code written in JavaScript. By learning Angular in JavaScript, it'll be easier for you starting out.

Second, JavaScript reads better in print. CoffeeScript relies heavily on whitespace for structure and meaning, and that makes it hard to format CoffeeScript code using the narrow margins imposed by print. With JavaScript, and all of its commas, parenthesis, and curly braces, we can format our code for print more easily.

a. http://coffeescript.org/

For right now, we want to assign the property search to our search function. Inside our search function, we'll assign the property searchedFor to the keywords *passed to* the search function.

```
var CustomerSearchController = function($scope) {
  $scope.search = function(searchTerm) {
    $scope.searchedFor = searchTerm;
  }
}
```

In our view we wrote ng-click="search(keywords)". When the user clicks that element, Angular will look in $scope to find a function named search, which we've just defined.

We aren't quite done, however. Although we declared a function named CustomerSearchController, this isn't sufficient for Angular to know that this function is what we are referring to when we wrote ng-controller="CustomerSearchController". Angular is not designed to look for variables of particular names, but instead looks at an internal registry.

So, we need to register our controller with Angular, so it can find it. The object returned from module (that represents our app, and was assigned to the variable app) is an angular.Module[2] and has a function named controller that will register our controller function.

2. https://docs.angularjs.org/api/ng/type/angular.Module

Before we see the actual call, I want to show you a simplified use of controller that demonstrates the concept, but that won't actually work in production.

```
// This code is conceptually correct, but won't work in production
app.controller("CustomerSearchController", CustomerSearchController);
```

We can see what controller does: it registers our CustomerSearchController function under the string CustomerSearchController. These two values don't have to be the same—we could've called our controller CustomerSearchCtrl. Angular doesn't care that the names match, only that they are registered properly. But why won't this work in production?

Angular is highly flexible. As such, there is no formal contract for what a controller function should accept as arguments. We wrote our controller expecting to be passed the scope we needed to expose values and functions to the view. Like everything in Angular, $scope is registered internally and mapped to the string $scope. And, in our development environment, Angular will actually inspect the *names* of our functions arguments, and use those to locate the registered objects.

This may seem strange, so let me give a little more detail. Angular will see that the name of the parameter we declared in our controller function is $scope. It will then look in its internal registry for an object mapped to the string $scope. Since Angular itself registered a Scope under that name, it will find it and pass it to our controller function.

You'll notice I said "in our development environment". In production, the Rails asset pipeline will *minify*[3] the Javascript. If you've ever looked at minified JavaScript, it's pretty unreadable. Among other things that minification does, it will change variable names to much shorter ones, all in an effort to save space and bandwidth when the user downloads the JavaScript in their browser. Our $scope argument might be re-named a during minification.

Thus, when Angular examines the arguments to our controller function after minification, the names of the arguments will be different. If $scope became a, Angular will look in its registry for the string a, find nothing, and generate an error.

So, we need explicitly tell Angular what objects we want passed to our function by putting that information somewhere where it won't get minified away. To do that, we can use an array as the second argument to controller. Above, we simply passed in our controller function. Now, we'll pass in an array that

3. http://en.wikipedia.org/wiki/Minification_%28programming%29

contains a list of the names of all objects we want passed to our controller function, with our controller function being the last element in that array.

```
app.controller("CustomerSearchController",
               [ "$scope", CustomerSearchController ]
);
```

Now, when this code gets minified, the string "$scope" in the array won't be touched, since minification won't change string literals. Angular will see that, no matter what the name of the argument to our controller function *actually* is, we want it to pass in the object mapped to the string "$scope" from its internal registry.

Like I said, this is strange. The reasons for this are partly due to JavaScript's overall lack of sophisticated language features, partly due to the minification issue, and partly a design decision by Angular to be as flexible as possible. All I can say is that you'll get used to it. In fact, you'll find it easier to declare your controllers like so:

typeahead/port-search-front-end/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerSearchController", [
          '$scope',
  function($scope) {
    $scope.search = function(searchTerm) {
      $scope.searchedFor = searchTerm;
    }
  }
]);
```

The alignment of '$scope' and scope in the function definition will underscore this strange-but-necessary way of setting up your code.

Now that we have our controller defined, let's add some code to the view so we can see the value of searchTerm. After our closing form tag, we'll add an h1 that displays the value of searchedFor, if it has a value.

typeahead/port-search-front-end/shine/app/views/customers/index.html.erb

```
</form>
<h1 class="searchedFor" ng-if="searchedFor">
  <small>Searched for:</small> {{searchedFor}}
</h1>
```

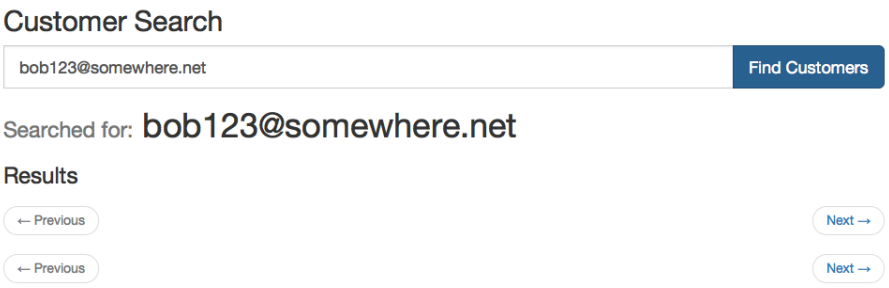
We've seen the curly-brace syntax before in our test app. This is Angular's templating system, and will substitute the value for searchedFor into the view, updating it as the value changes. It will expect to find the current value in $scope.searchedFor, which you'll recall is where we set the value in our controller, inside the search function.

The ng-if directive works as you might imagine: it renders the element it's applied to (the h1), and all its children, *only* if the searchedFor has a value.

This template is analogous to the following ERB:

```
<% if searchedFor.present? %>
  <h1>
    <small>Searched for: <%= searchedFor %></small>
  </h1>
<% end %>
```

Now, start your server, navigate to http://localhost:3000/customers, enter in a search term and click *Find Customers*. You should see the h1 appear with your search term.



What this demonstrates is that we can use Angular directives to access data from form elements, and execute code that we write using that data. Now, we need to learn how to render a list of results in our view. We'll do this by modifying search to return a canned set of results that look like what we'll expect from the server. This will allow us to learn a bit more without getting wrapped up in AJAX requests and JSON responses from the server.

Rendering Canned Search Results

Now that we have our search form in place, let's change the search method to populate the results section with some canned results. This will accomplish two things: it will allow us to understand how to render collections with Angular, but it will also motivate us to figure out what sort of data we'll need to get back from the server, when we finally implement the search itself.

First, let's change search to assign a list of results to a new object in $scope called customers. We'll use keys that map the fields of our ActiveRecord object, as this will make it a snap to implement the back-end later.

typeahead/canned-results/shine/app/assets/javascripts/customers_app.js

```
$scope.customers = [];
$scope.search = function(searchTerm) {
```

```
  $scope.customers = [
    {
      "first_name":"Schuyler",
      "last_name":"Cremin",
      "email":"giles0@macgyver.net",
      "username":"jillian0",
      "created_at":"2015-03-04",
    },
    {
      "first_name":"Derick",
      "last_name":"Ebert",
      "email":"lupe1@rennerfisher.org",
      "username":"ubaldo_kaulke1",
      "created_at":"2015-03-04",
    },
    {
      "first_name":"Derick",
      "last_name":"Johnsons",
      "email":"dj@somewhere.org",
      "username":"djj",
      "created_at":"2015-03-04",
    }
  ]

}
```

You'll notice on the first line of the listing that we've explicitly initialized `$scope.customers` to an empty array. While this isn't strictly necessary, it's good form as it lets future readers of the code know what values we're exposing to the view. Now we need to modify our template to render these canned results.

In an Angular template, to render a collection, we can use the `ng-repeat` directive. This will repeat the markup it's placed on (and its contents) once for each element in the collection. For example, our canned `customers` array has three elements in it. The following Angular template:

```
<div ng-repeat="customer in customers">
  {{customer.name}}
</div>
```

will render the following:

```
<div>
  Schuyler
</div>
<div>
  Derick
</div>
<div>
  Derick
```

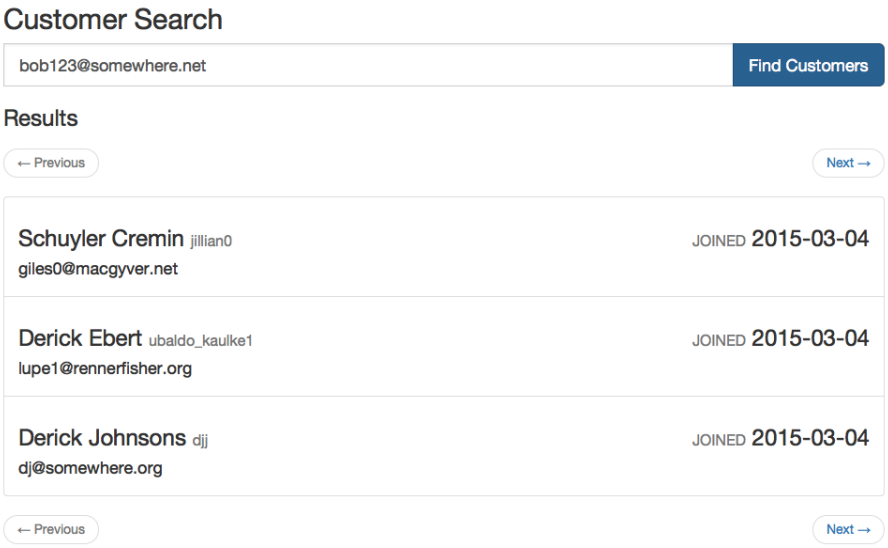
```
</div>
```

To convert our ERB template to Angular, we'll remove the `@customers.each` and replace it with `ng-repeat`, as well as to replace the `<@= .. %>` with Angular's curly-brace syntax.

typeahead/canned-results/shine/app/views/customers/index.html.erb

```
<section class="search-results">
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <%= render partial: "pager",
             locals: { keywords: @keywords, page: @page } %>

  <ol class="list-group">
    <li class="list-group-item clearfix"
        ng-repeat="customer in customers">
      <h3 class="pull-right">
        <small class="text-uppercase">Joined</small>
        {{ customer.created_at }}
      </h3>
      <h2 class="h3">
        {{ customer.first_name }} {{ customer.last_name }}
        <small>{{ customer.username }}</small>
      </h2>
      <h4>{{ customer.email }}</h4>
    </li>
  </ol>
  <%= render partial: "pager",
             locals: { keywords: @keywords, page: @page } %>
</section>
```

Now, when we click *Find Customers*, we can see our search results render on the page.



All that's left is to have `search` make an AJAX request to the back-end to get the real results.

Making an AJAX Request to Complete the Circle

Our Rails controller currently just handles regular browser-based requests. To serve results via an AJAX request, we'll need to use a more JavaScript-friendly format. JSON is the best fit, so we'll modify our controller to serve JSON if requested.

To do that, we'll just need to use Rails' `respond_to` method to indicate that we handle JSON and then use the `json` method to specify the JSON to return.

typeahead/angularized-search/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController
  PAGE_SIZE = 10

  def index

    # existing index method

    respond_to do |format|
      format.html {}
      format.json { render json: @customers }
    end
  end
end
```

Rails will handle converting our Customer instances into JSON[4], which will be in the format we used for our canned results, meaning we won't have to change our template once we start using this now-JSON-ified endpoint.

To get this data into our view, we need to modify search to use this endpoint. Angular provides a service we can use to make AJAX requests called $http[5]. Much like how $scope was registered under the name $scope, Angular also registers $http under the name $http. This means that if we want access to it, we add the string $http to our constructor initializer Array, and add a second argument to our initialization function.

typeahead/angularized-search/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerSearchController", [
          '$scope','$http',
  function($scope , $http) {
```

To make our AJAX request using $http, we'll use the get function it provides.

The get function takes a URL and some options (which is where we specify the request parameters) and returns a *promise* object. This object exposes two functions, success and error, that accept functions as arguments. Those functions allow us to specify what to do on a successful or unsuccessful AJAX request, respectively. These functions will be called whenever the AJAX request completes. It can be slightly confusing, but this is how almost all JavaScript code works—asynchronously. Let's see the code.

typeahead/angularized-search/shine/app/assets/javascripts/customers_app.js

```
  $scope.search = function(searchTerm) {
    $http.get("/customers.json",
              { "params": { "keywords": searchTerm } }
    ).success(
      function(data,status,headers,config) {
        $scope.customers = data;
    }).error(
      function(data,status,headers,config) {
        alert("There was a problem: " + status);
      });
  }
}
```

This will make the AJAX request to the server and, when we get a successful response, call the function passed to success. If the response is not successful (i.e. an HTTP error code of 400 or greater), error will be called instead.

4. http://guides.rubyonrails.org/v3.2.9/action_controller_overview.html#rendering-xml-and-json-data
5. https://docs.angularjs.org/api/ng/service/$http

The function we passed to success accepts four arguments, but we only care about the first: data. This will be an object representing the parsed JSON our server returned. Since that is in the exact format we used for our canned data, we can assign it to $scope.customers directly.

The function passed to error works similarly. Using alert to handle errors from the server isn't a great user experience, and we'll make this better later on, but for now this will suffice, especially since we aren't expecting any errors from the server.

If we start our server and navigate to http://localhost:3000/customers, we can now perform a search just as it worked before, but entirely powered by Angular!

One thing you'll notice is that created_at is a timestamp, not a date, so it's not rendering exactly how we'd like it. In our ERB template, we converted it to a date using the l helper method. Angular has a different way to handle this.

In Angular there are *filters* that work more like UNIX pipes than like method calls. In this case, Angular provides a filter named date that can be used inside the curly-brackets in our templates, like so:

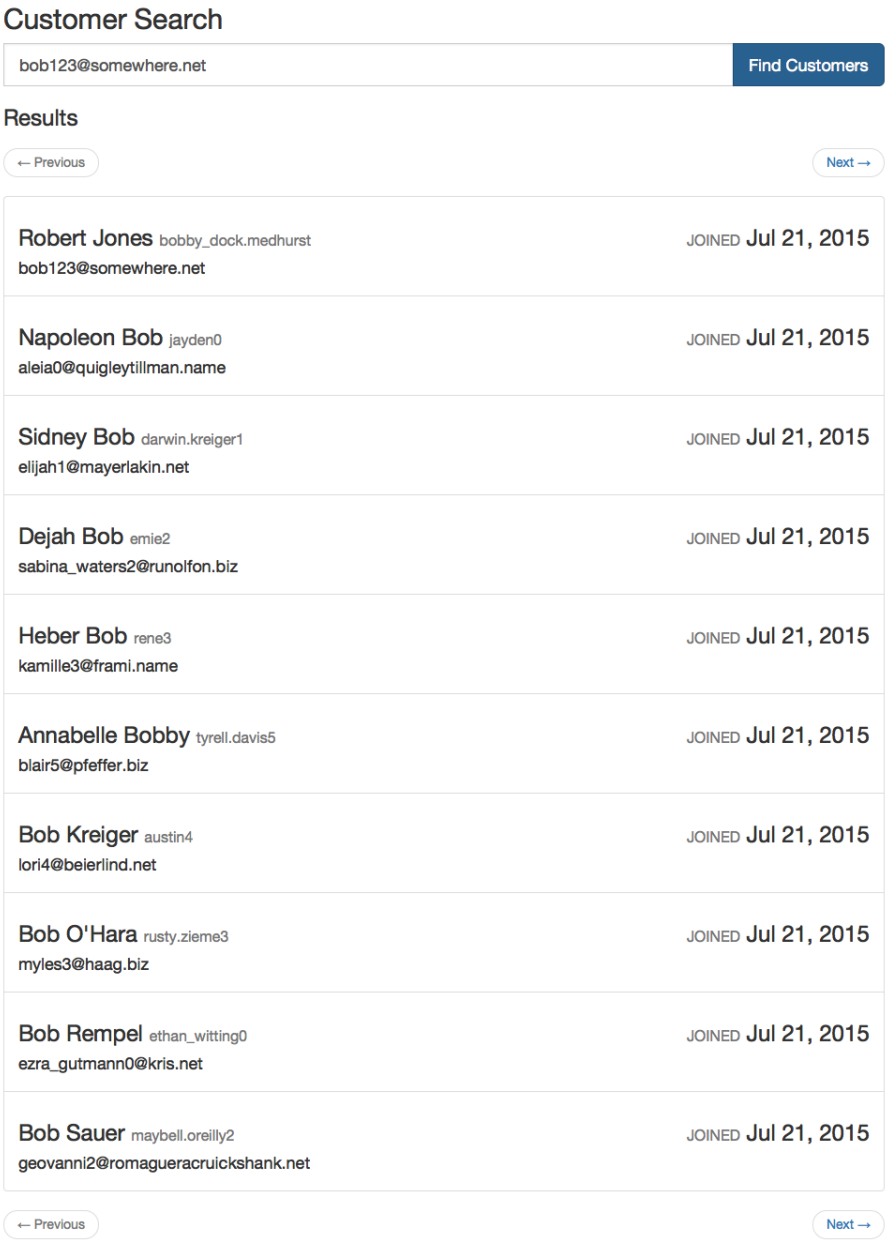
typeahead/angularized-search/shine/app/views/customers/index.html.erb

```
<li class="list-group-item clearfix"
    ng-repeat="customer in customers">
  <h3 class="pull-right">
    <small class="text-uppercase">Joined</small>
    {{ customer.created_at | date }}
  </h3>
  <h2 class="h3">
    {{ customer.first_name }} {{ customer.last_name }}
    <small>{{ customer.username }}</small>
  </h2>
  <h4>{{ customer.email }}</h4>
</li>
```

The date[6] filter can take a lot of options to customize how the timestamp is rendered, but for our purposes, the default behavior will work. Angular includes other filters[7], and it's straightforward to write you own (which we'll see later).

With that in place, our search looks just like it did before.

6. https://docs.angularjs.org/api/ng/filter/date
7. https://docs.angularjs.org/api/ng/filter



We still need to deal with the pagination, however.

Re-implementing the Pagination

In the original version of our search, our controller exposed `@page` to the view, so the view would know what page was being viewed. This allowed us to pass

back either `@page + 1` or `@page - 1` as the `page` parameter in our links. Our Angularized version is using AJAX, so we'll need to send the right page number with those requests.

To do this, we'll need to keep track of what page we are on, and use it in our call to `$http.get`. We'll also need to re-implement the pager buttons. First, let's modify the controller and AJAX request.

typeahead/angularized-pagination/shine/app/assets/javascripts/customers_app.js

```
function($scope , $http) {

  var page = 0;

  $scope.search = function(searchTerm) {
    $http.get("/customers.json",
              { "params": { "keywords": searchTerm, "page": page } }
    ).success(
```

Note that `page` is not part of `$scope`—it's local to our controller function. Since the view won't need access to this value, there's no reason to expose it. Because of JavaScript's scoping rules, `$scope.search` will have access to it, but the view won't.

Next, we'll modify our pager partial to use `ng-click` instead of Rails link helpers. We'll assume two functions on our controller named `nextPage` and `previousPage` exist. We'll see their implementation in a moment.

typeahead/angularized-pagination/shine/app/views/customers/_pager.html.erb

```
<nav>
  <ul class="pager">
    <li class="previous">
      <a href="" ng-click="previousPage()">&larr; Previous</a>
    </li>
    <li class="next">
      <a href="" ng-click="nextPage()">Next &rarr;</a>
    </li>
  </ul>
</nav>
```

Note how much simpler our template is. Instead of having links that we have to remember will change the current page of results on the server, we instead have shorter and more readable markup. Even if you didn't know Angular at all, you can immediately tell what the intention of this code is.

Now, let's see `previousPage`. To implement this, we'll need to explicitly grab the value of `keywords` from the view. Because we used `ng-model` on our `input` element, `$scope.keywords` will always have the value of in the field (which implies that we didn't really need to send it to `search`, but it will make testing this method

easier later on). So, we just need to decrement page (assuming it's not zero) and call search.

typeahead/angularized-pagination/shine/app/assets/javascripts/customers_app.js

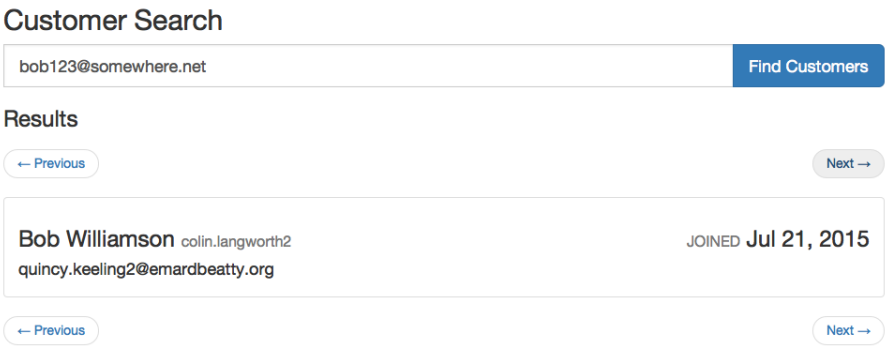
```
$scope.previousPage = function() {
  if (page > 0) {
    page = page - 1;
    $scope.search($scope.keywords);
  }
}
```

nextPage looks very similar.

typeahead/angularized-pagination/shine/app/assets/javascripts/customers_app.js

```
$scope.nextPage = function() {
  page = page + 1;
  $scope.search($scope.keywords);
}
```

Now, when we start our server and do a search, we can click *Next*, and the next page of results will be fetched and displayed in our view.



Converting our search to use Angular is just a step toward our goal of making the search feature work better for our users. By keeping the functionality the same while converting to Angular, we were able to just focus on the Angular-based aspects of the feature. Now that we've done that, we can change the search so that it searches as we type.

Changing our Search to Use Typeahead

Given everything we've done up to this point, changing the search from one where you must click a button to search to one where the search happens as you type will actually be fairly straightforward. Because Angular has allowed us to separate our concerns, we have all the code we need in place. We'll just need to connect it to the user interface in a different way.

First, let's modify search so it will only hit the server if the user has typed three or more characters. It's unlikely the user will make sense of results based on one or two characters, so we can save a trip the server by waiting until they type the third character.

typeahead/actual-typeahead/shine/app/assets/javascripts/customers_app.js

```
$scope.search = function(searchTerm) {
  if (searchTerm.length < 3) {
    return;
  }

  // ... rest of the function
```

Now, we need to arrange for search to be invoked whenever the user is typing and *not* when the user clicks the *Find Customers* button. Just as we used ng-click to bind the click event on the button to our search function, we can use the directive ng-change on the input field to invoke search when the text field contents changes.

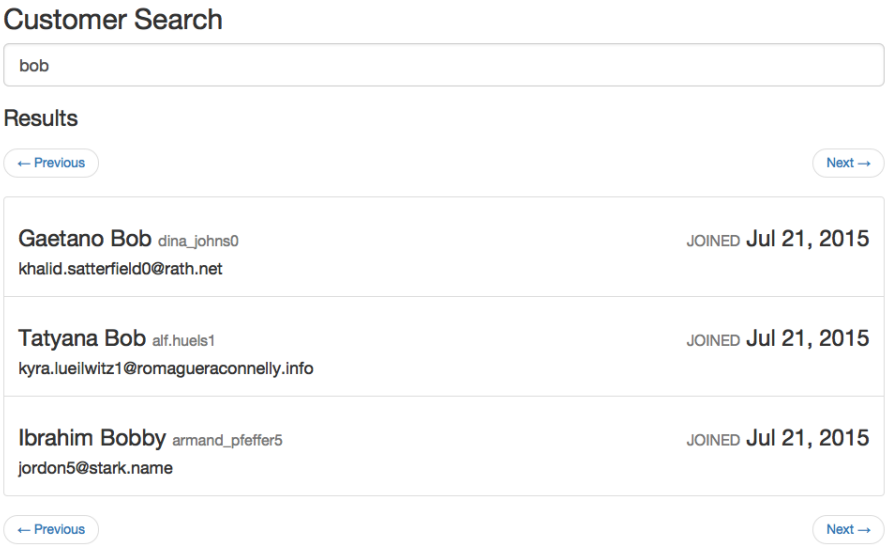
We'll also remove the search button, which makes our form much simpler, as we no longer need to put the text field and the submit button into a Bootstrap input-group.

typeahead/actual-typeahead/shine/app/views/customers/index.html.erb

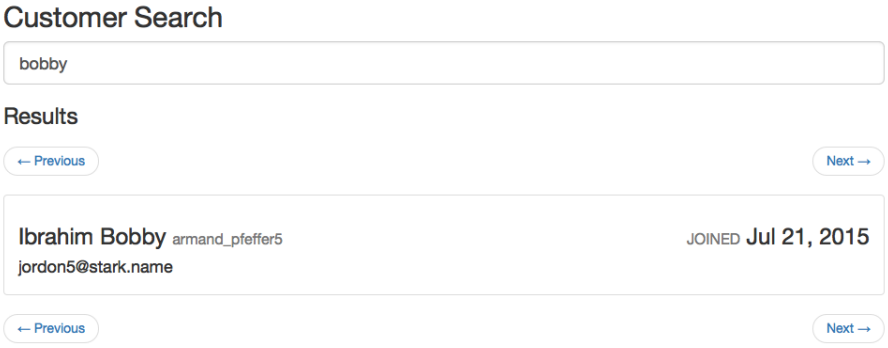
```
<section class="search-form">
  <form>
    <label for="search.keywords" class="sr-only">Keywords</label>
    <input type="text"
           name="keywords"
           class="form-control input-lg"
           placeholder="First Name, Last Name, or Email Address"
           ng-change="search(keywords)"
           ng-model="keywords">
  </form>
</section>
```

You'll note that we've highlighted the input tag, and you can see that we've used ng-change to arrange for search to be called with keywords whenever the typing changes.

Start the server and begin typing. In my test data, I have three users whose last names start with bob, one of which has the last name Bobby. Typing just bob, I see all three results.



If I keep typing out `bobby`, the results automatically reduce to only those that match.



The typeahead works! The entire feature required very little code, and instead of implementing typeahead with a special-purpose library, we have set up a framework for implement any user interface we might need. Because of how Angular works, we aren't wrestling with how to attach our JavaScript to our DOM elements or how to interact with the Backend. Because of how Rails works, our back-end is almost identical to the original back-end.

In other words, by using what Rails gives us, and using what Angular gives us, we were able to create a fairly sophisticated feature quickly, without a lot of code. And it's fast, thanks to Postgres' sophisticated indexing and ordering features.

Next Up: Testing

It's one thing to get code working a browser, but it's another to have the confidence in that code that you can ship it your users. To get that confidence, you need automated tests. We've completely avoided writing tests up to this point, because it would complicate the tasks of learning about Angular, Bootstrap, and Postgres.

Now that we've got a bit of confidence with these new technologies, we're at a point where we can turn our attention to tests. In the next chapter, we'll build on the testing tools that Rails provides, and learn how to test database constraints, write unit tests for our Angular code, and write acceptance tests that execute our Angular app in a real browser.

CHAPTER 7

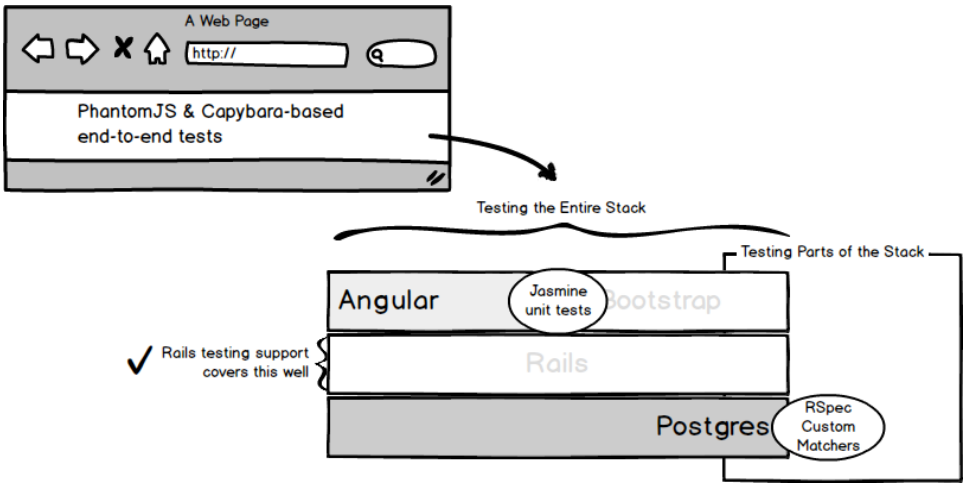
Test This Fancy New Code

To be confident with new libraries and technologies, we need to do more than just write working code—we have to be able to test it. Rails has a long history of supporting and encouraging testing, providing many useful features for testing every part of your application. We want the same experience testing Angular and database constraints that we get with Rails models and controllers. We also want a seamless experience testing end-to-end in a browser.

Rails has no built-in support for testing JavaScript, nor does it provide a direct way to test database constraints. And, there is no Rails Way for running an end-to-end test in a browser. But, Rails is configurable enough to allow us to set it up ourselves. That's what we'll learn here.

We'll learn how to write a clean and clear test of the database constraint we created in *Exposing The Vulnerability Devise and Rails Leaves Open*, on page 42. We'll then learn how to create acceptance tests that run in a real browser, executing our client-side code the same way a user's browser would, using Capybara, PhantomJS, and Poltergeist.

Finally, we'll see how to write a unit test for the Angular code we wrote in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81 using the JavaScript-based test library Jasmine. This will allow us to write small, focused tests for the individual pieces of our Angular app, without relying entirely on the browser-based acceptance tests.



For the tests we'll write in Ruby, we'll use RSpec instead of Test::Unit, and this requires some additional setup to our Rails application. So, before we get into our actual tests, let's get that set up, and understand *why* we want to use RSpec.

Installing RSpec for Testing

Rails ships with Test::Unit as the default testing framework. Test::Unit is a fine choice, and demonstrates the concepts of testing in Rails quite well. Despite that, RSpec is quite popular among Ruby developers. An annual survey conducted by Hampton Catlin[1] shows that, of the developers polled, 69.4% prefer RSpec for testing.

While this is a good reason to become familiar with RSpec, it's not the main reason we want to use it here. When we get to *Writing Unit Tests for Angular Components*, on page 123, we'll be using Jasmine for testing our JavaScript, and both Jasmine and RSpec share a similar syntax. Here's an RSpec test:

```
describe "a simple test" do
  it "should test something" do
    expect(number).to eq(10)
  end
end
```

Here is that same test in Jasmine:

```
describe("a simple test", function() {
  it("should test something", function() {
    expect(number).toEq(10);
```

1. http://www.askr.me/ruby

```
  });
});
```

As you can see, both tests have a similar shape and structure. This means that when you are bouncing between Ruby and JavaScript tests, the mental overload will be less, since you'll be looking at the same overall way of structuring and organizing your tests. So, while RSpec has many virtues, our reason for using it here is to reduce friction as we test throughout the stack of our application.

With that said, setting it up is easy. First, add *rspec-rails* to the Gemfile. This gem includes tight Rails integration for RSpec and will bring in the base RSpec gems as dependents.

testing/install-rspec-from-blank/shine/Gemfile

```
group :development, :test do
  gem "rspec-rails"
end
```

Now, we bundle install.

```
> bundle install
```

RSpec comes with a generator that will add the necessary configuration to get RSpec working.

```
> bundle exec rails g rspec:install
    create  .rspec
      exist  spec
     create  spec/spec_helper.rb
     create  spec/rails_helper.rb
```

The majority of RSpec's configuration is in spec/spec_helper.rb. It includes a set of defaults, commented-out, that it recommends you uncomment. We're going to uncomment most of them so we can use them, but also add a few non-default configuration options. Below is what the file should look like, with our additions highlighted.

testing/install-rspec-from-blank/shine/spec/spec_helper.rb

```
RSpec.configure do |config|
  config.expect_with :rspec do |expectations|
    expectations.include_chain_clauses_in_custom_matcher_descriptions = true
➤   expectations.syntax = [:expect]
  end

  config.mock_with :rspec do |mocks|
    mocks.verify_partial_doubles = true
➤   mocks.verify_doubled_constant_names = true
  end
```

```
  config.filter_run :focus
  config.run_all_when_everything_filtered = true

  config.disable_monkey_patching!
➤ config.expose_dsl_globally = true

  if config.files_to_run.one?
    config.default_formatter = 'doc'
  end

  config.profile_examples = 10

  config.order = :random

  Kernel.srand config.seed
end
```

We're explicitly requiring the expect(..) syntax, as we don't want to use the .should assertions as this would be counter to our desire for our Ruby and JavaScript tests to be similar. We've set verify_doubled_constant_names as an extra safety measure if we should need to mock class behavior (this warns us if we mock classes that don't exist). Finally we're setting expose_dsl_globally which will allow our tests to just use RSpec's DSL methods like describe and it without prefixing them with RSpec..

The Rails-specific configuration in spec/rails_helper.rb is fine as-is for now. Let's create a dummy spec file to verify everything's working in spec/dummy_spec.rb

testing/install-rspec-from-blank/shine/spec/dummy_spec.rb

```
require "rails_helper.rb"

describe "testing that rspec is configured" do
  it "should pass" do
    expect(true).to eq(true)
  end
  it "can fail" do
    expect(false).to eq(true)
  end
end
```

When we run rake, it will run this spec, and we should see one test pass and the other fail.

```
> bundle exec rake
testing that rspec is configured
  should pass
  can fail (FAILED - 1)

Failures:
```

```
  1) testing that rspec is configured can fail
     Failure/Error: expect(false).to eq(true)

       expected: true
            got: false

       (compared using ==)
     # ./spec/dummy_spec.rb:8:in `block (2 levels) in <top (required)>'

Finished in 0.00388 seconds (files took 4.06 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/dummy_spec.rb:7 # testing that rspec is configured can fail

Randomized with seed 7430
```

This looks good, and we're ready to learn about how to test the new features and technologies we've learned. We'll start by writing tests of our `User` model that exercise the database constraints we created in *Exposing The Vulnerability Devise and Rails Leaves Open*, on page 42.

Testing Database Constraints

When treating your SQL database as a "dumb store" (or when using an RDBMS that lacks the sophisticated features of Postgres), you would typically use various features of ActiveRecord to ensure database integrity and you'd naturally want to test that code. Although we are using constraints to enforce database integrity (like the `check` constraint from *Exposing The Vulnerability Devise and Rails Leaves Open*, on page 42), we'd still like to have test coverage that the constraint is doing what we want.

We can easily test this constraint in RSpec, but it requires a somewhat cumbersome assertion mechanism using exceptions. First, we'll see how this works, and then we'll learn how to create an RSpec matcher to abstract the awkward syntax away so our tests can be cleaner and clearer.

Asserting that Constraints Exist Using RSpec's Matchers

To test our database constraint, we'll need to force ActiveRecord to insert bad data into the database, so that *Postgres* is generating the error about bad data, not ActiveRecord. You'll recall that we added ActiveRecord validations to assist in the registration process, which means it will be difficult to using ActiveRecord to insert bad data into the database. Difficult, but not impossible.

The method update_attribute is available on all ActiveRecord instances and it circumvents validations. We can use this in our test to attempt to insert bad data and simulate a rogue agent that's not using ActiveRecord. It will attempt to write to the database immediately, so invoking it with a non-example.com email address should fail. Here's what it looks like in Rails console.

```
> bundle exec rails c
2.2.0 :002 > User.first.update_attribute(:email,"foo@somewhere.com")
  User Load (0.7ms)  SELECT  "users".* FROM "users"
                       ORDER BY "users"."id" ASC LIMIT 1
   (0.1ms)  BEGIN
  SQL (1.8ms)  UPDATE "users" SET "email" = $1,
                                  "updated_at" = $2
               WHERE "users"."id" = $3
                  [
                    ["email", "foo@somewhere.com"],
                    ["updated_at", "2015-03-18 15:33:08.091547"],
                    ["id", 2]
                  ]
➤ PG::CheckViolation: ERROR:  new row for relation "users" violates
➤                             check constraint "email_must_be_company_email"
➤ ActiveRecord::StatementInvalid: PG::CheckViolation: ERROR:
➤            new row for relation "users" violates
➤              check constraint "email_must_be_company_email"
```

I've omitted and reformatted the error produced to only show what we are interested in, but we can see that the internals of Rails raised a PG::CheckViolation which ActiveRecord wrapped inside a ActiveRecord::StatementInvalid exception. To test that this error occurs, we'll need to use RSpec's expect { ... }.to raise_error(...) form.

The raise_error matcher accepts two arguments: the exception we're expecting, and the message it should contain (or a regular expression it should match). Since ActiveRecord::StatementInvalid is so generic, if we just check that we've received that exception, our test might pass if there are different errors happening. We want the test to pass *only* when our constraint is violated. So, we'll expect both that a ActiveRecord::StatementInvalid is raised, and also that the error message names our constraint. This isn't as precise as we'd like, but it's the best we can do, and a reasonable compromise.

testing/test-postgres-constraint/shine/spec/models/user_spec.rb

```
require 'rails_helper'

describe User do
  describe "email" do
    let(:user) {
      User.create!(email: "foo@example.com",
                   password: "qwertyuiop",
```

```
                    password_confirmation: "qwertyuiop")
    }
    it "absolutely prevents invalid email addresses" do
      expect {
        user.update_attribute(:email, "foo@bar.com")
➤     }.to raise_error(ActiveRecord::StatementInvalid,
➤                      /email_must_be_company_email/i)
    end
  end
end
```

Note that we're using a regular expression as the second argument to raise_error so that we aren't too tightly coupled to the specific error message. Let's run our tests (note that we're using rspec to run a single tests—rake still works to run our entire test suite).

```
> rspec spec/models/user_spec.rb

Randomized with seed 16383

User
  email
    absolutely prevents invalid email addresses

Finished in 0.09785 seconds (files took 4.91 seconds to load)
1 example, 0 failures

Randomized with seed 16383
```

Our test passes. This gives us a way to drive the addition of sophisticated database constraints with tests. But, it's pretty ugly catching exceptions and asserting that their messages match a regular expression. Unfortunately, there's not a better way to do it, but we *can* make our test code a bit cleaner and more intention-revealing by creating a custom RSpec matcher.

Using RSpec Matchers to Make our Test Code Cleaner

Rspec uses the term *matcher* to describe the constructs it provides to evaluate assertions. In a line of code like expect(2 + 2).to eq(4), the method eq is a matcher. It's matching the result of 2 + 2 against the constant 4.

We can create our own custom matchers to test attributes of our code that are more particular to what we're doing. This saves us test code, and can make our tests more clear. Ideally, we'd be able to write our test like so:

testing/custom-rspec-matcher/shine/spec/models/user_spec.rb

```
it "absolutely prevents invalid email addresses" do
  expect {
    user.update_attribute(:email, "foo@bar.com")
```

```
➤   }.to violate_check_constraint(:email_must_be_company_email)
end
```

If we could create the matcher violate_check_constraint, it will not only make our tests more clear, but will also allow us to abstract the method we're using to test: catching an exception and checking its message. This means if we could devise a better way of testing the constraint, we only have to change it in one place—our matcher.

RSpec makes it easy to create such a matcher. The code to do so is quite dense, but once you see how it works, you'll find it's straightforward to create your own custom matchers.

We'll create our custom matcher in spec/support/violate_check_constraint_matcher.rb. It's customary to put code that supports your specs in spec/support. Naming the file violate_check_constraint_matcher.rb will make it easy to know what's in there and where to find it, since it uses the name of the matcher with a _matcher suffix.

Let's see the code, and then we can walk through it line-by-line.

testing/custom-rspec-matcher/shine/spec/support/violate_check_constraint_matcher.rb

```
1 RSpec::Matchers.define :violate_check_constraint do |constraint_name|
2   supports_block_expectations
3   match do |code_to_test|
      begin
4       code_to_test.()
5       false
6     rescue ActiveRecord::StatementInvalid => ex
7       ex.message =~ /#{constraint_name}/
      end
    end
end
```

Like I said, this code is dense, so we'll take it one step at a time.

❶ Here, we define our matcher and state its name. Since RSpec has an English-like syntax, you'll want your matcher to follow from the word "to". In this case we expect our code "to violate check constraint email_must_be_company_email". Any arguments given to the matcher are passed to the block as arguments. We've named the argument we're expecting constaint_name.

❷ By default, custom matchers don't support the block syntax we're using. In that case, the match method (below) would be given the result of the code under test. Since we need to actually *execute* the code under test ourselves—so we can detect the exception that was thrown—we need to

use the block syntax. The supports_block_expectations method tells RSpec that this is the case.

❸ This is where we define what passing or failing means. match takes a block that is expected to evaluate to true or false if the actual value matches the expected one, or not, respectively. Since we used supports_block_expectations, the argument passed is the block used, unexecuted. Our job is to execute it and see what happens.

❹ Here, we run the code under test.

❺ If we didn't get an exception, this is where the flow-of-control will end up. Since we *want* an exception, getting here means our test failed, so we return false.

❻ Here, we catch the exception we're expecting. If we get any other exception, the test will fail. Catching the exception is only part of the test.

❼ The final part of the test is to examine the message of the caught exception. Just as we did before, we'll simply assert that it contains the name of the constraint we're expecting should be violated.

We're almost ready to use our custom matcher. The last thing to do is to bring it into our spec file. While it's possible to configure RSpec to auto-require everything in spec/support, doing so can make your specs much harder to understand. Because RSpec plays so fast-an-loose with Ruby's syntax, it can be challenging to look at the use of a matcher and figure out where it's defined.

To that end, we'll explicitly require our customizations, like so:

testing/custom-rspec-matcher/shine/spec/models/user_spec.rb

```
require 'rails_helper'
➤ require 'support/violate_check_constraint_matcher'

describe User do
  describe "email" do

    # ... rest of the test

  end
end
```

Running our spec, we can see it still passes.

```
> rspec spec/models/user_spec.rb

Randomized with seed 2818

User
```

```
  email
    absolutely prevents invalid email addresses

Finished in 0.15076 seconds (files took 5.78 seconds to load)
1 example, 0 failures
```

We've now seen how RSpec can allow us to test our database constraints and, by using custom matchers, do so with clean and clear test code.

Now, let's head to the total opposite end of our application stack and learn how to write end-to-end acceptance tests that run in a real browser, thus executing the Angular code we've written and simulate user behavior.

Running Headless Acceptance Tests in PhantomJS

Acceptance Tests[2] are the way in which we assure that our application meets the needs of the users. In most Rails applications, an acceptance test performs a *black box*[3] test against the HTTP endpoints and routes.

When our application uses a lot of JavaScript—as our Angular-powered typeahead search does—it's often necessary for our acceptance tests to execute the downloaded HTML, CSS, and JavaScript in a running browser, so we can be sure that all of the DOM manipulation we are doing is actually working.

Typically, developers would use Selenium, which would launch an instrumented instance of Firefox, running it on your desktop during the acceptance testing phase. This is quite cumbersome and slow, and for running tests on remote continuous integration[4] servers, it required special configuration to allow a graphical app like Firefox to run.

Ideally, we'd want something that executes our front-end code in a real browser—complete with a JavaScript interpreter—but that can run *headless*, that is, without popping up a graphical application. *PhantomJS*[5] is such a browser.

PhantomJS describes itself as "a headless WebKit, scriptable with a JavaScript API". WebKit is the browser engine that powers Apple's Safari (and was the basis of Google's Chrome). The "scriptable JavaScript API" means that we can interact with it in our tests.

2. http://en.wikipedia.org/wiki/Acceptance_testing
3. http://en.wikipedia.org/wiki/Black-box_testing
4. http://en.wikipedia.org/wiki/Continuous_integration
5. http://phantomjs.org/

Most Rails acceptance tests use *Capybara*[6], which provides an API to interacting with such an instrumented browser. To allow Capybara to talk to PhantomJS, we're going to use *Poltergeist*[7], which is analogous to Selenium, if we were using Firefox.

This may sound like a *ton* of new technologies and buzzwords, but it's really not. All we need to do is add the PhantomJS and Poltergeist gems to our Gemfile, do a bit of configuration, and start writing acceptance tests as we normally would. Let's get to it.

Installing and Setting-Up PhantomJS and Poltergeist

First, you'll need to download and install PhantomJS. The specifics of this depend on your operating system, but the details for Mac, Windows, and Linux are on PhantomJS's download page[8].

You can verify your install by running phantomjs and issuing some basic JavaScript.

```
> phantomjs
phantomjs> console.log("HELLO!");
HELLO!
undefined
phantomjs>
```

This is the only time you'll need to interact with PhantomJS in this way, but it's enough to validate your install.

Now, we'll install Poltergeist, which is an adapter between the Ruby code we'll write for our acceptance tests and the "scriptable JavaScript API" PhantomJS provides. To this, add it to the testing group in your Gemfile and then do bundle install to install it.

testing/setup-poltergeist/shine/Gemfile

```
group :development, :test do

  # other gems...

  gem 'rspec-rails'
➤ gem 'poltergeist'
end
```

Installing Poltergeist will bring in Capybara as a dependency. If you aren't familiar with it, we'll explain more when we see the acceptance tests.

6. https://github.com/jnicklas/capybara
7. https://github.com/teampoltergeist/poltergeist
8. http://phantomjs.org/download.html

To use Poltergeist, now that it's installed, we need to do two things. First, we'll need to configure Capybara to use it during test runs. Secondly, we'll need to configure RSpec to handle the testing database differently for acceptance tests than for our unit tests. We'll explain more in a minute.

To connect Poltergeist and Capybara, we just need a few lines in spec/rails_helper.rb. We'll need to require Poltergeist and then set Capybara's drivers to use it. Capybara has two different drivers: one default and one for JavaScript. This is handy if we don't have a lot of JavaScript and want our acceptance tests to normally run using an special in-process driver that won't execute JavaScript on the page. That's not the case for Shine, so we'll use Poltergeist (which is powering PhantomJS) for all acceptance tests.

Here's the changes to spec/rails_helper.rb:

```
ENV['RAILS_ENV'] ||= 'test'
require 'spec_helper'
require File.expand_path('../../config/environment', __FILE__)
require 'rspec/rails'
➤ require 'capybara/poltergeist'

➤ Capybara.javascript_driver = :poltergeist
➤ Capybara.default_driver    = :poltergeist

ActiveRecord::Migration.maintain_test_schema!

RSpec.configure do |config|

  # rest of the file ...

end
```

Note that we're using spec/rails_helper.rb and not spec/spec_helper.rb because these tests require the full power of Rails to execute (namely, access to ActiveRecord and the path helpers).

Configuring Poltergeist is the easy part. The trickier part is how to deal with the testing database. In a normal Rails unit test, the testing database is maintained using *database transactions*[9]. At the start of a test run, Rails opens a new transaction. Our tests would then write data to the database to set up the test, run the test (which might make further changes to the database) and then assert the results, which often require querying the database. When the test is complete, Rails will *roll back* the transaction,

9. http://en.wikipedia.org/wiki/Database_transaction

effectively undoing all the changes we made, restoring the test database to a pristine state.

This works because the process that starts the transaction can see all of the changes made to the database inside that transaction, even though no other process can. Since Rails runs our tests in the same process that it uses to execute them, using transactions is a clever and efficient way to manage test data. But, our acceptance tests will actually run *two* processes: our application and our test code (which will use PhantomJS to access our application).

This means that if our tests are setting up the test database inside a transaction, our server won't be able to see that data and our tests won't work. What we need to do is *actually* write the data to our database and commit those changes permanently.

Doing *this* creates a new problem, which is that we now need a way to restore the test database to a pristine state between test runs. For example, if we are testing our search by populating the database with four users named "Bob", but we are also testing our registration by signing up a user named "Bob", our search test might fail if the registration test runs first, since there would be *five* users named "Bob".

Fortunately, this is a common problem and has a relatively simple solution: the *DatabaseCleaner*[10] gem.

DatabaseCleaner works with RSpec and Rails to reset the database to a pristine state without using transactions (although it can—it provides several strategies). RSpec allows us to customize the database setup and teardown by test type. This means we can keep the fast and efficient transaction-based approach for our unit tests, but use a different approach for our acceptance tests.

First, we'll add DatabaseCleaner to our Gemfile and bundle install

testing/setup-poltergeist/shine/Gemfile

```
group :development, :test do

  # other gems...

  gem 'database_cleaner'
end
```

Now, we'll configure it in spec/rails_helper.rb. To do this, we'll disable RSpec's built-in database handling code by setting use_transactional_fixtures to false (note

10. https://github.com/DatabaseCleaner/database_cleaner

that the generated rails_helper.rb will have it set to *true*). We'll then use RSpec's hooks[11] to allow DatabaseCleaner to handle the databases. By default, we'll use DatabaseCleaner's :transaction strategy, which works just like RSpec and Rails' default. But, for our acceptance tests (which RSpec calls "features"), we'll use :truncation, which means DatabaseCleaner will use the SQL truncate[12] keyword to purge data that's been committed to the database.

Here's what we'll add to spec/rails_helper.rb, with the most relevant parts highlighted.

testing/setup-poltergeist/shine/spec/rails_helper.rb

```
RSpec.configure do |config|
  config.fixture_path = "#{::Rails.root}/spec/fixtures"

➤   config.use_transactional_fixtures = false
➤   config.infer_spec_type_from_file_location!

  # rest of the file...


  config.before(:suite) do
    DatabaseCleaner.clean_with(:truncation)
  end

  config.before(:each) do
    DatabaseCleaner.strategy = :transaction
  end

➤   config.before(:each, :type => :feature) do
➤     DatabaseCleaner.strategy = :truncation
➤   end

  config.before(:each) do
    DatabaseCleaner.start
  end

  config.after(:each) do
    DatabaseCleaner.clean
  end
end
```

Now that we've set up PhantomJS, Poltergeist, and DatabaseCleaner, we're ready to write an acceptance test.

11. https://www.relishapp.com/rspec/rspec-core/v/3-2/docs/hooks/before-and-after-hooks
12. http://en.wikipedia.org/wiki/Truncate_%28SQL%29

Writing Our First Acceptance Test

To validate that our testing setup is working and that PhantomJS is properly executing our JavaScript in the browser, let's write a test for the test Angular App we created in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81. As you recall, this Angular app had a text field where we could type a name. As we typed, a heading would dynamically update with what we had entered.

Before we see the actual test, let's plan our how it will work. First, we have to log in, since every page in Shine requires a login. That means we'll need to set up a test user and fill in that user's name and password on the login screen. Then, we'll enter some text into the text field in our Angular test app. Finally, we'll assert that the DOM updated with what we typed.

Way back in Chapter 3, *Secure the Login Database with Postgres Constraints*, on page 41, we talked about how Devise properly secures our user information, including passwords. This means it will be very difficult for us to write a valid encrypted password from our tests. Fortunately, the additions Devise made to our User model allow us to do this directly.

```
User.create!(email: "bob@example.com"
             password: "password123",
             password_confirmation: "password123")
```

This is what happens when a user registers, so we can just call code like this in our test. RSpec provides the method before to allow us to run code before any tests runs (we saw something similar when setting up DatabaseCleaner). We'll also use the method feature (instead of describe) to indicate that this is an acceptance tests.

Finally, we want to avoid duplicating the test user's email and password, so we'll put those into variables using let. Here's an outline of our acceptance test so far, which is in spec/features/angular_test_app_spec.rb

testing/setup-poltergeist/shine/spec/features/angular_test_app_spec.rb

```
require 'rails_helper'

feature "angular test" do

  let(:email)    { "bob@example.com" }
  let(:password) { "password123" }

  before do
    User.create!(email: email,
                 password: password,
                 password_confirmation: password)
```

```
  end

  # tests will go here...

end
```

Our test itself will need to login, assert that we're on the Angular test app's page, fill in a name, and assert that the DOM was updated. This is where we'll see Capybara's DSL in action. A good reference for everything you can do can be found on Capybara's GitHub page[13], but we'll call out the methods we're using.

Primarily we'll simulate user behavior with visit (to navigate to a particular URL), fill_in (to enter data into a text field), and click_button (to, you guessed it, click a button). For asserting that our application is working, we'll use the have_content matcher, which checks for text within a given DOM element. By default, it checks the entire page. We'll also use within which will restrict the part of the page where we're asserting content.

Let's see the test.

testing/setup-poltergeist/shine/spec/features/angular_test_app_spec.rb

```
require 'rails_helper'

feature "angular test" do

  # setup from before...

  scenario "Our Angular Test App is Working" do
    visit "/angular_test"

    # Log In
    fill_in      "Email",    with: "bob@example.com"
    fill_in      "Password", with: "password123"
    click_button "Log in"

    # Check that we go to the right page
    expect(page).to have_content("Name")

    # Test the page
    fill_in "name", with: "Bob"
    within "header h1" do
      expect(page).to have_content("Hello, Bob")
    end
  end
```

13. https://github.com/jnicklas/capybara#the-dsl

Thanks to Capybara's DSL, the test is pretty readable. When we run it, it works, thus validating that PhantomJS is executing the Angular app on the page.

```
> rspec spec/features/angular_test_app_spec.rb
angular test
Our Angular Test App is Working

Finished in 2.05 seconds (files took 4.43 seconds to load)
1 example, 0 failures

Randomized with seed 20281
```

Before we move on, let's write a test of our typeahead feature from the previous chapter, as this will be a bit more involved and allow us to both test an *actual* feature of Shine, but also see how to use Capybara in light of a dynamic page with a heavier client-side code.

Testing the Typeahead Search

There are two parts of the typeahead search we can test. The first is that merely typing into the search field will perform the search. The second is that our results are ordered according to our original specification from Chapter 4, *Use Fast Queries with Advanced Postgres Indexes*, on page 49.

To test the search, we'll write two tests: one that searches by name, and a second that searches by email. This will allow us to validate that matching emails are listed first. Both tests will assert that merely typing in the search term returns results.

Unlike the test for our Angular test app, *this* test requires a bit more setup. Namely, we need to create customers in the database, create a test user, login as that user, and navigate to the customer search page.

To create customers, we're going to create them manually inside our test file. Although Rails provides *test fixtures*[14] to do this, we're not going to use them here. Because tests related to search require meticulous set-up of many different rows, we want that setup to be in our test file so that we (and future maintainers of our code) can clearly see what we're setting up for our test.

To help us create customers, we'll define a helper method, `create_customer`, that will allow us to specify only those fields of a customer we want, using Faker to fill in the rest of required fields.

14. http://guides.rubyonrails.org/testing.html#the-low-down-on-fixtures

testing/setup-poltergeist/shine/spec/features/customer_search_spec.rb
```
require 'rails_helper'

feature "Customer Search" do
  def create_customer(first_name: nil,
                      last_name: nil,
                      email: nil)

    first_name ||= Faker::Name.first_name
    last_name  ||= Faker::Name.last_name
    email      ||= "#{Faker::Internet.user_name}#{rand(1000)}@" +
                     "#{Faker::Internet.domain_name}"
    Customer.create!(
      first_name: first_name,
       last_name: last_name,
        username: "#{Faker::Internet.user_name}#{rand(1000)}",
           email: email
    )
  end
```

We'll also re-create the user and password let statements from our Angular test app test, since we'll need to create a user here and login before the test.

testing/setup-poltergeist/shine/spec/features/customer_search_spec.rb
```
let(:email)    { "bob@example.com" }
let(:password) { "password123" }
```

With create_customer, email, and password in place, we can now create the test data we need to run our tests. As before, we'll do this in a before block.

The before block is *also* useful for behavioral setup, like logging the user in before the test. In our Angular test app test, we logged the user in as *part* of the test. Since we aren't testing user login explicitly, this makes our test more verbose than it needs to be. We can test login elsewhere, so for *this* test, we'd like to avoid having login code in the actual test itself. But, we still need to be logged in to run our tests. This kind of *behavioral setup* can go in the before block.

Our before block now looks like so (we've highlighted the login code we copied from the Angular test app test):

testing/setup-poltergeist/shine/spec/features/customer_search_spec.rb
```
before do
  User.create!(email: email,
               password: password,
               password_confirmation: password)

  create_customer first_name: "Robert",
                  last_name: "Aaron"
```

```
    create_customer first_name: "Bob",
                    last_name: "Johnson"

    create_customer first_name: "JR",
                    last_name: "Bob"

    create_customer first_name: "Bobby",
                    last_name: "Dobbs"

    create_customer first_name: "Bob",
                    last_name: "Jones",
                        email: "bob123@somewhere.net"
➤   visit "/customers"
➤   fill_in      "Email",    with: "bob@example.com"
➤   fill_in      "Password", with: "password123"
➤   click_button "Log in"
  end
```

Now, we can start writing tests. Our first test will search by name. If we search for the string "bob", given our test data, we should expect to get four results back. Further, we should expect that the test user named "JR Bob" will be sorted first, while the test user "Bob Jones" will be last (since our search sorts by last name).

To assert this, we'll use the `all` method of the `page` object Capybara provides in our tests. `all` returns all DOM nodes on the page that match a given selector. In our case, we can use a CSS selector to count all list items with the class `list-group-item` (you'll recall from Chapter 5, *Create Clean Search Results with Bootstrap Components*, on page 69 that we designed our results using Bootstrap's List Group component). We can also dereference the value returned by `all` to make assertions about the content of a particular list item.

Here's what our test looks like (note the use of `scenario` instead of `it`—this is purely stylistic, but most RSpec acceptance tests use this for readability).

testing/setup-poltergeist/shine/spec/features/customer_search_spec.rb

```
scenario "Search by Name" do
  within "section.search-form" do
    fill_in "keywords", with: "bob"
  end
  within "section.search-results" do
    expect(page).to have_content("Results")
    expect(page.all("ol li.list-group-item").count).to eq(4)

    expect(page.all("ol li.list-group-item")[0]).to have_content("JR")
    expect(page.all("ol li.list-group-item")[0]).to have_content("Bob")

    expect(page.all("ol li.list-group-item")[3]).to have_content("Bob")
    expect(page.all("ol li.list-group-item")[3]).to have_content("Jones")
```

```
  end
end
```

Before we see the next test, I want to point out a useful feature of within. Although within allows us to scope the part of the page where we're making assertions, it serves a much more useful purpose when testing applications that make heavy use of JavaScript. You'll recall that our template used ng-if to show the results markup *only* if there were any results.

The documentation for ng-if[15] states:

> If the expression assigned to ng-if evaluates to a false value then the element is removed from the DOM…

This means that there could be a *race condition*[16] in our test. After we enter search results, the page is making an AJAX call to get the results. Only when that call completes will the page be updated with the results causing section.search-results to actually be in the DOM. If our test were to look it before that happens, it won't find those elements and the test will fail—even though our code is actually working.

This is a problem that within solves for us. Instead of requiring that the markup matched by the selector passed to within exist immediately, within will wait up to 3 seconds (which is configurable if needed). It does this to give all JavaScript on the page a chance to run and get the DOM into working order.

Because Shine will be using Angular, it means that our in-browser tests will need to make heavy use of within to avoid these race conditions.

Now that we understand the importance of within, let's see our test for searching by email. It will be structured similarly to our previous test, but we want to check that the user with the matching email is listed first. We'll then check that the remaining results are sorted by last name, using a similar technique to what we saw in our search-by-name test.

testing/setup-poltergeist/shine/spec/features/customer_search_spec.rb

```
scenario "Search by Email" do
  within "section.search-form" do
    fill_in "keywords", with: "bob123@somewhere.net"
  end
  within "section.search-results" do
    expect(page).to have_content("Results")
    expect(page.all("ol li.list-group-item").count).to eq(4)

    expect(page.all("ol li.list-group-item")[0]).to have_content("Bob")
```

15. https://docs.angularjs.org/api/ng/directive/ngIf
16. http://en.wikipedia.org/wiki/Race_condition

```
    expect(page.all("ol li.list-group-item")[0]).to have_content("Jones")

    expect(page.all("ol li.list-group-item")[1]).to have_content("JR")
    expect(page.all("ol li.list-group-item")[1]).to have_content("Bob")

    expect(page.all("ol li.list-group-item")[3]).to have_content("Bob")
    expect(page.all("ol li.list-group-item")[3]).to have_content("Johnson")
  end
end
```

Now, let's run our tests.

```
> rspec spec/features/customer_search_spec.rb

Customer Search
  Search by Email
  Search by Name

Finished in 3.63 seconds (files took 7.08 seconds to load)
2 examples, 0 failures

Randomized with seed 4873
```

They pass! We now have a way to test our features the way a user would use them: using a real browser. Our tests can properly handle our extensive use of JavaScript, but they don't need to pop up a web browser, which makes them easy to run in a continuous integration environment.

Of course, testing our Angular code purely in the browser is somewhat cumbersome, especially if it becomes complex with a lot of edge cases. To help us get good test coverage without always having to go through a browser, we need to be able to unit test our Angular code.

Writing Unit Tests for Angular Components

Browser-based acceptance are slow and brittle. Even though we've eschewed starting an actual browser for each test by using PhantomJS, we still have to start our server and have it serve pages to the headless browser. Further, our tests relay on DOM elements and CSS classes to locate content used to verify behavior. We may need (or want) to make changes to the view that don't break functionality, but break our tests.

Although we don't want to abandon acceptance tests—after all, they are the only tests we have that exercise the system end-to-end—we need a way to test isolated bits of functionality (commonly called *unit tests*). In Rails, we have model tests and controller tests to allow us to do that for our server-side

code. Unfortunately, Rails doesn't provide any help for unit testing our client-side code.

In a classic Rails app, there simply isn't much client-side code, so we are comfortable not explicitly testing it. In our more modern app, where a non-trivial amount of logic is written in JavaScript, the lack of unit-testing will be a problem. For example, in our Angular app that powers the typeahead search that we built in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81, the `previousPage` function has logic to prevent going to negative page numbers. That should be tested, and it makes a lot more sense to test it as a unit test than to set up an entire acceptance test to verify we can't go to a negative page.

In this section we'll set up a means of writing and running unit tests for our Angular code. We'll be using *Jasmine*, which is a commonly-used JavaScript testing framework. We'll be using *Teaspoon*, which is a Rails plugin to help integrate Jasmine into our workflow. With those in place, Angular provides everything else we need for testing (in fact, testing is the reason Angular does all the dependency injection we saw earlier). We'll start with a simple test of our code, and then see how to use *test spies* to do more sophisitcated tests.

First, let's get Jasmine and Teaspoon installed so we have a place to write unit tests for our Angular code.

Setting up Jasmine and Teaspoon

Jasmine[17] is a JavaScript testing framework similar to RSpec. It has nothing to do with Rails, and is designed to run either in the browser or on the command-line. *Teaspoon*[18] is a JavaScript test-runner for Rails. It isn't specific to Jasmine, but can run Jasmine tests. In this section we'll get them both working together in Shine so we can then write unit tests for our Angular code.

First, we'll add Teaspoon to our `Gemfile` and run `bundle install`. Note that we're adding the gem `teaspoon-jasmine` and not just `teaspoon`. Teaspoon has a core gem that requires a second gem specific to the test framework we're using. In our case, that's Jasmine, and `teaspoon-jasmine` depends on the `teaspoon` gem, so we just need to add `teaspoon-jasmine`.

testing/setup-jasmine-and-teaspoon/shine/Gemfile

```
group :development, :test do
```

17. http://jasmine.github.io/
18. https://github.com/modeset/teaspoon

```
  # rest of gems...
  gem 'teaspoon-jasmine'
end
```

After installing teaspoon-jasmine with bundle install, we can then use a Rails generator that teaspoon provides that we'll get us set up. We'll run it, telling it we don't want to use Coffeescript (see *Why Aren't We Using CoffeeScript?*, on page 88 for some details on why we aren't using Coffeescript).

```
> bundle exec rails generate teaspoon:install --no-coffee
      create  spec/teaspoon_env.rb
      create  spec/javascripts/support
      create  spec/javascripts/fixtures
      create  spec/javascripts/spec_helper.js
+======================================================================+
Congratulations!  Teaspoon was successfully installed. Documentation
and more can be found at: https://github.com/modeset/teaspoon
```

If you look at spec/javascripts/spec_helper.js, you can see a lot of comments in that file, as well as what appear to be Sprockets directives like //= require application. Teaspoon is using the asset pipeline to pull in our code. This is a good news for us Rails developers, because it means we don't have to learn a second means of bringing our JavaScript into scope to execute it and test it.

Note also that we don't need to explicitly install Jasmine—Teaspoon brings in the version it needs on its own.

To validate our setup we'll write a bare-bones test using Jasmine and make sure it runs. Jasmine's test syntax is similar to RSpec, in that we use describe to set up a test suite and it to write a single test. We'll make the simplest test we can in spec/javascripts/dummy_spec.js.

```
describe("Testing Jasmine", function() {
  it("can run a test", function() {
    expect(true).toBe(false);
  });
});
```

You'll notice we're expecting true to be false. This will let us see a test fail, so we can fix it and see it succeed.

To run our test, Teaspoon provides a rake task teaspoon.

```
> bundle exec rake teaspoon
Starting the Teaspoon server...
Puma 2.11.1 starting...
* Min threads: 0, max threads: 16
* Environment: test
* Listening on tcp://0.0.0.0:53430
```

```
Teaspoon running default suite at http://127.0.0.1:53430/teaspoon/default

F

Failures:

  1) Testing Jasmine can run a test
     Failure/Error: Expected true to be false.

Finished in 0.00600 seconds
1 example, 1 failure

Failed examples:

teaspoon -s default --filter="Testing Jasmine can run a test."
rake teaspoon failed
```

This is a good failure. This means that our test was executed and our assertion failed. We can fix the test by changing toBe(false) to toBe(true).

testing/setup-jasmine-and-teaspoon/shine/spec/javascripts/dummy_spec.js

```
describe("Testing Jasmine", function() {
  it("can run a test", function() {
➤   expect(true).toBe(true);
  });
});
```

Now, we can run our tests again and see success.

```
> bundle exec rake teaspoon
Starting the Teaspoon server...
Puma 2.11.1 starting...
* Min threads: 0, max threads: 16
* Environment: test
* Listening on tcp://0.0.0.0:53459
Teaspoon running default suite at http://127.0.0.1:53459/teaspoon/default
.
Finished in 0.00400 seconds
1 example, 0 failures
```

This validates our JavaScript unit-testing setup. Next, we'll use this setup to write a test of our Angular code.

Writing a Unit Test for our Angular Code

Testing Angular components is just like testing our Ruby code: we set up the conditions of a test, execute one of the publicly-exposed functions of our code, and check the results. In the case of our Angular code, the setup is a bit more complex, but once we start writing the tests, they will look familiar.

We'll approach our testing in three steps. In the first, we'll walk through the steps needed to set up our test. This is surprisingly involved, but allows our actual tests to be simple. In the second step, we'll test that `$scope.customers` is empty when the controller loads. That will let use examine the process for setting up the test, but without worrying about simulating the AJAX calls. Finally, we'll use a function provided by angular's *angular-mocks* package to simulate the AJAX calls.

Setting up An Angular Controller Test

If you recall from Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81, we mentioned that Angular stores our controller function in a repository, and that Angular will call it one time when starting up the app. Further, you'll recall that our controller function's arguments are expected to be passed by Angular during this process.

testing/angular-unit-test/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerSearchController", [
          '$scope','$http',
  function($scope , $http) {
```

In the test context, we won't be starting up the Angular app, but instead need to find our controller function and call it explicitly. Further, we need access to `$scope` in order to write our tests. `$scope` is where our public functions live, and where the values available to the view are available—it can be thought of as the public interface of our controller.

Before we can think about accessing our controller, we need to instruct Angular to load our app (which isn't the same as *starting* the app). Since all of our Angular code is in callback functions, none of them are executed by default. We can use the function `module` (provided by Angular) to load our app.

```
module("customers");
```

To create an instance of our controller, Angular provides a service function named `$controller`[19]. This function accepts two arguments: the name of the controller to instantiate, and an object of variables to be injected into the controller's constructor function. We'd use it like so:

```
module("customers");
var scope = {};
var controller = $controller("CustomerSearchController",
                             { "$scope": scope });
```

19. https://docs.angularjs.org/api/ng/service/$controller

This call will find our controller function and call it. For the first argument (named $scope), the local variable scope will be passed in. For the second argument (named $http), because we didn't specify it in the second argument to $controller, Angular will find the default implementation and pass it for us.

This is the mechanism we can use to exercise and examine the public members of controller. Because the values and functions get set on the passed-in $scope, and we've overridden the default behavior to pass in our own scope, we can then call functions like search, and examine values like customers.

That said, we don't want to pass in a vanilla object for $scope. We should pass in the same sort of object that Angular would use. This is because our controller might rely on getting an actual scope and not just an object. Passing in an object could cause our tests to fail in unexpected ways.

In order to do pass in a real scope, we need access to $rootScope[20], which exposes a function $new (note that preceding dollar-sign). When we call $new, we'll get back a new $rootScope.Scope[21], which is the same type that Angular passes in at runtime. The result will look something like this:

```
module("customers");
var scope = $rootScope.$new();
var controller = $controller("CustomerSearchController",
                             { "$scope": scope });
```

You might be wondering how we get access to the $controller and $rootScope functions. Here's where it starts to feel like Inception[22].

Our controller-declared arguments, along with the array of strings naming them, allow Angular to find those registered objects and pass them into our controller function. We can use this mechanism ourselves via the inject function. This function can be used to ask Angular to call a function with any objects in its internal registry. So, we'll ask Angular to pass both $controller and $rootScope into a function in which we'll set up our controller as outlined above.

This is Angular's flexibility rearing its head. It seems weird, and feels like work Angular should be doing for us. All I can say is that you'll get used to it. Here's what our code will look like.

```
module("customers");
inject(function($controller, $rootScope) {
```

20. https://docs.angularjs.org/api/ng/service/$rootScope
21. https://docs.angularjs.org/api/ng/type/$rootScope.Scope
22. http://www.imdb.com/title/tt1375666/

```
  var scope = $rootScope.$new();
  var controller = $controller("CustomerSearchController",
                               { "$scope": scope });
});
```

If you're paying close attention, you'll notice that our use of inject is a bit different than how we used controller in our production code, namely that we aren't using the array of strings argument. In *Creating Our First Angular Controller*, on page 87, we went into great detail about why we have to keep the names of arguments in a string so that Angular knew what to pass in at runtime. Since our tests won't be minified and served up through the asset pipeline, there's no concern that the arguments to inject will be minified away. Therefore, we just use plain syntax and allow Angular to examine the argument names directly.

A more pressing question might be where inject comes from. Do we have to call another function to get *that* passed into our test?

The answer is "no". inject is an alias to angular.mock.inject[23], which is provided by the *angular-mocks* module. This module adds inject to the global namespace, specifically so we can write the code we've seen.

To install angular-mocks, we'll need to add it to our Bowerfile and then run rake bower:install.

testing/angular-unit-test/shine/Bowerfile

```
asset 'angular-mocks'
```

Once it's installed, we need to bring it into our testing code using Sprockets require directive. Since we don't want it in our production code, we'll add the require directive to spec/javascripts/spec_helper.js (instead of app/assets/javascripts/application.js. Note that angular-mocks needs to be required *after* other Angular modules have been brought in, so we'll add it add the require after we've required application.js (which is what brings our production JavaScript into the tests):

testing/angular-unit-test/shine/spec/javascripts/spec_helper.js

```
//= require application
//= require angular-mocks
```

With that done, our call to inject will work as described. The next question is where to call it? Jasmine provide a function named beforeEach that works like before in RSpec. We'll also need to call module("customers") in a beforeEach block, so our app is always loaded.

23. https://docs.angularjs.org/api/ngMock/function/angular.mock.inject

Lastly, we want to hold a reference to our scope and controller so we can use them in our tests. To do that, we'll declare scope and controller outside our beforeEach functions. This means our set-up code will look like so:

testing/angular-unit-test/shine/spec/javascripts/customers_app/controllers/customer_search_controller_spec.js

```
describe("CustomerSearchController", function() {
  describe("Initialization", function() {

    var scope       = null,
        controller  = null;

    beforeEach(module("customers"));

    beforeEach(inject(function ($controller, $rootScope) {
      scope       = $rootScope.$new();
      controller  = $controller("CustomerSearchController", {
        $scope: scope
      });
    }));

    // tests go here...

  });
});
```

There's no particular location where the test files themselves should go, so we've defaulted to using a directory inside spec/javascripts based on our Angular app name (customers_app) and, inside there, a directory for the type of component (controllers). Further, we're naming our file using Rails' convention of underscorizing the class name (customer_search_controller_spec.rb). Angular, Jasmine, and Teaspoon don't care one bit about this, so you can use whatever makes sense to you.

Now that we've set everything up, writing the test is, as promised, straightforward.

Writing a Simple Test of Our Angular Code

For this test, we'll assert that the initial value of $scope.customers is the empty array. It might seem like you should write the test this way:

```
it("defaults to an empty customer list", function() {
  expect(scope.customers).toBe([]);
});
```

This will fail with an oh-so-helpful error message:

```
> bundle exec rake teaspoon
Teaspoon running default suite at http://127.0.0.1:65305/teaspoon/default
F.
```

```
Failures:

  1) CustomerSearchController Initialization defaults to an empty customer list
     Failure/Error: Expected [  ] to be [  ].

Finished in 0.01700 seconds
2 examples, 1 failure
```

I don't know about you, but [ ] sure looks equal to [ ], so what's going on? The issue is that toBe expects both objects being tested to be the exact same object, not just two objects that have the same value.

While this issue isn't a big deal for this particular test—we could just check that the length of scope.customers is 0—we'll write another test in a minute that will expect scope.customers to have data in it. Unfortunately, Jasmine doesn't provide matcher to compare the values inside arrays and objects. But, much like RSpec, Jasmine allows creating custom matchers. Further, Angular provides the function angular.equals[24] which can do the comparison we want, so all we need to do is create a custom Jasmine matcher using angular.equals.

Adding a custom matcher in Jasmine is a bit complex (see the documentation[25] for a great walkthrough), but it's not something we'll be doing a lot in the book, so we don't have to worry about the details. All we need to do is add this code to the end of spec/javascripts/spec_helper.js:

testing/angular-unit-test/shine/spec/javascripts/spec_helper.js

```
beforeEach(function(){
  jasmine.addMatchers({
    toEqualData: function(util,customEqualityTesters) {
      return {
        compare: function(actual,expected) {
          var result = {};
          result.pass = angular.equals(actual, expected);
          return result;
        }
      };
    }
  });
});
```

We've highlighted the meat of the matcher, which is the call to angular.equals. Jasmine looks for the returned result object's pass property to be true or false. We can now write our test like so:

24. https://docs.angularjs.org/api/ng/function/angular.equals
25. http://jasmine.github.io/2.3/custom_matcher.html

testing/angular-unit-test/shine/spec/javascripts/customers_app/controllers/customer_search_controller_spec.js
```
describe("CustomerSearchController", function() {
  describe("Initialization", function() {

    // setup code from before...

    it("defaults to an empty customer list", function() {
      expect(scope.customers).toEqualData([]);
    });
  });
});
```

And, our test passes!

```
> bundle exec rake teaspoon
Starting the Teaspoon server...
Puma 2.11.1 starting...
* Min threads: 0, max threads: 16
* Environment: test
* Listening on tcp://0.0.0.0:65382
Teaspoon running default suite at http://127.0.0.1:65382/teaspoon/default
..

Finished in 0.02500 seconds
2 examples, 0 failures
```

Despite how cumbersome the setup was, our actual test—the code that's most important—is straightforward and readable. Next, let's write a test for search. We'll want to see that calling search with a keyword makes an AJAX call and populates $scope.customers with the results. Because this is a unit test, we'll need to fake-out the HTTP call to the server.

Simulating AJAX Calls in a Unit Test

You'll recall that we injected $http into our Angular controller in order to make AJAX calls. We could pass in our own implementation of $http—much as we did with $scope—but this would be a lot of work. $http packs a lot of features and we'd need to mimic those in our test object to make sure the controller could function when executed from the test. Instead, angular-mocks provides a test-only version of $httpBackend, which is what $http uses under the covers.

The angular-mock version of $httpBackend[26] allows us to specify what AJAX requests might get made, and to control what gets returned. The function when can be used to configure an HTTP call, and the function respond controls the response, like so:

26. https://docs.angularjs.org/api/ngMock/service/$httpBackend

testing/angular-unit-test/shine/spec/javascripts/customers_app/controllers/customer_search_controller_spec.js

```
describe("Fetching Search Results", function() {
  beforeEach(module("customers"));

  beforeEach(function() {
    httpBackend.when('GET','/customers.json?keywords=bob&page=0').
                respond(serverResults);
  });
});
```

As before, we need Angular to inject $httpBackend into our test code so we can grab a reference to it. We can do this by adding $httpBackend to our call to inject when we set up our controller. While we're making changes, we'll also create a varaible serverResults that holds the simulated results passed-back from the server.

testing/angular-unit-test/shine/spec/javascripts/customers_app/controllers/customer_search_controller_spec.js

```
describe("Fetching Search Results", function() {
  var scope       = null,
      controller  = null,
      httpBackend = null,
      serverResults = [
        {
          id: 123,
          first_name: "Bob",
          last_name: "Jones",
          email: "bjones@foo.net",
          username: "jonesy"
        },
        {
          id: 456,
          first_name: "Bob",
          last_name: "Johnsons",
          email: "johnboy@bar.info",
          username: "bobbyj"
        }
      ];
  beforeEach(inject(function ($controller, $rootScope, $httpBackend) {
    scope       = $rootScope.$new();
    httpBackend = $httpBackend;
    controller  = $controller("CustomerSearchController", {
      $scope: scope
    });
  }));
```

With that in place, we can now write our test. All our test needs to do is call scope.search and then check that scope.customers has the right data in it. The only catch is that HTTP calls in JavaScript are asynchronous. Therefore, we need

a way to tell $httpBackend to call our callback functions registered in our controller. We can do that with the function flush.

Our test looks like so:

testing/angular-unit-test/shine/spec/javascripts/customers_app/controllers/customer_search_controller_spec.js

```
  // previous setup code

  it("populates the customer list with the results", function() {
    scope.search("bob");
    httpBackend.flush();
    expect(scope.customers).toEqualData(serverResults);
  });
});
```

If we run our tests again, we can see that our new test is passing.

```
> bundle exec rake teaspoon
Starting the Teaspoon server...
Puma 2.11.1 starting...
* Min threads: 0, max threads: 16
* Environment: test
* Listening on tcp://0.0.0.0:49187
Teaspoon running default suite at http://127.0.0.1:49187/teaspoon/default
...

Finished in 0.02400 seconds
3 examples, 0 failures
```

With these building blocks, we can test all sorts of server interactions. The respond function is highly flexible, allowing us to simulate different responses, including errors from the server. If you recall, our error-handling code pops up a JavaScript alert. While that's not the greatest user experience of all time, it's a) sufficient and b) what our code is currently doing. It would be ideal if we could test this. In the next section we'll learn how to do that using *Test Spies*.

Testing Browser-provided Features using Spies

If you've written a lot of tests for a Rails app, you've no-doubt used (or at least heard of) *mock objects*. A mock object is a stand-in for a real object that you use for test isolation. The $httpBackend variable we used in the previous section is an example. We don't want our test making real HTTP calls; we can assume that Angular's $http service is working and just focus on our code.

Jasmine provides a form of mock object called a *test spy* that allows us to record interactions with an object, and check those interactions later. It's

perfect for when we want to test the behavior of a function (as opposed to its return value).

Our error-handling code for our customer search provides an example.

testing/angular-unit-test/shine/app/assets/javascripts/customers_app.js

```
}).error(
  function(data,status,headers,config) {
    alert("There was a problem: " + status);
  });
```

Here, we're calling alert with a particular message. Because alert pops up a dialog, and our unit tests aren't running in a real browser, we'd like to *spy* on window (which is the global scope where alert is defined) and verify that, when we get an error, window.alert is called with the message we expect.

To test this, we'll need to configure $httpBackend to return an error, instead of a customers list. We'll also need to tell Jasmine to starting *spying on* window's alert function by using the spyOn function. spyOn takes two arguments: the object to spy on and the function to watch. Jasmine will capture any calls made to that function on the spied-upon object, allowing us to check later if the function was called and what arguments were passed.

testing/angular-unit-test/shine/spec/javascripts/customers_app/controllers/customer_search_controller_spec.js

```
describe("Error Handling", function() {

  // same setup as previous test...

  beforeEach(function() {
    httpBackend.when('GET','/customers.json?keywords=bob&page=0').
                respond(500,'Internal Server Error');
    spyOn(window, "alert");
  });
```

The respond function's arguments are quite flexible. Before, we passed an object as the first argument. In that case, $httpBackend interprets that as an HTTP 200 response, returning the given object as JSON. Here, we're using a number—500—which is interpreted as an HTTP status code. This should trigger the error callback in the search function in our controller.

To test that this is the case, we want to assert that our customers list is still empty *and* that window.alert was called with the string "There was a problem: 500". We can test the later with the matcher toHaveBeenCalledWith, which asserts that a given function was called and that it was called with particular arguments.

testing/angular-unit-test/shine/spec/javascripts/customers_app/controllers/customer_search_controller_spec.js

```
it("alerts the user on an error", function() {
```

```
  scope.search("bob");
  httpBackend.flush();
  expect(scope.customers).toEqualData([]);
  expect(window.alert).toHaveBeenCalledWith(
    "There was a problem: 500");
});
```

Running our test, we can see that it passes.

```
> bundle exec rake teaspoon
Starting the Teaspoon server...
Puma 2.11.1 starting...
* Min threads: 0, max threads: 16
* Environment: test
* Listening on tcp://0.0.0.0:64968
Teaspoon running default suite at http://127.0.0.1:64968/teaspoon/default
....

Finished in 0.02400 seconds
4 examples, 0 failures
```

For completeness, let's see it fail by expecting a different message.

```
it("alerts the user on an error", function() {
  scope.search("bob");
  httpBackend.flush();
  expect(scope.customers).toEqualData([]);
  expect(window.alert).toHaveBeenCalledWith("OH NOES! 500");
});
```

```
> bundle exec rake teaspoon
Teaspoon running default suite at http://127.0.0.1:64993/teaspoon/default
..F.

Failures:

  1) CustomerSearchController Error Handling alerts the user on an error
     Failure/Error: Expected spy alert to have been called with
         [ 'OH NOES!' ]
       but actual calls were
         [ 'There was a problem: 500' ]

Finished in 0.04000 seconds
4 examples, 1 failure
```

We've now seen how to set up unit test-running in our Rails app using Teaspoon and Jasmine. We've learned how to write a unit test for our Angular code, including using some mock objects provided by Angular. Finally, we've seen how to use Jasmine's test spies to assert our code is calling the external

functions we expect it to. These are the building-blocks you need to write tests for your front-end code.

Next Up: Level Up On Everything

We learned a ton in this chapter about testing our Rails application at every level of the stack. Now that we can test anything from database constraints, to JavaScript functions, to end-to-end user interactions, we're ready to move on to more complex features.

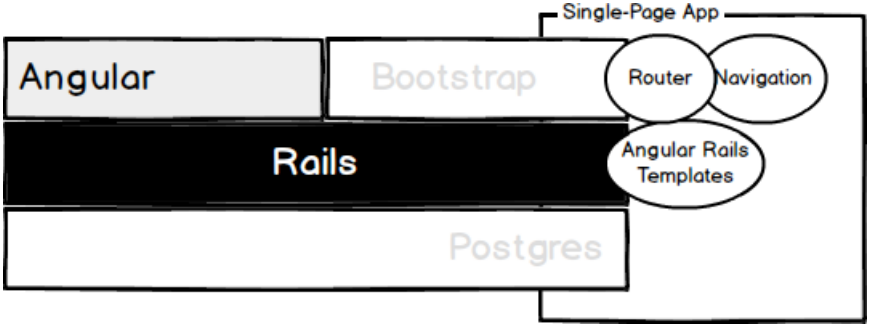
Now it's time to up our game on everything. Over the next several chapters, we'll build a complex customer detail view. This will be a great chance to learn how to design a dense UI with Bootstrap, wrangle multiple data sources with Angular, and optimize complex queries inside Postgres. But first, we need to turn our simple search screen into a *single-page app* by learning about Angular's router and navigation services.

CHAPTER 8

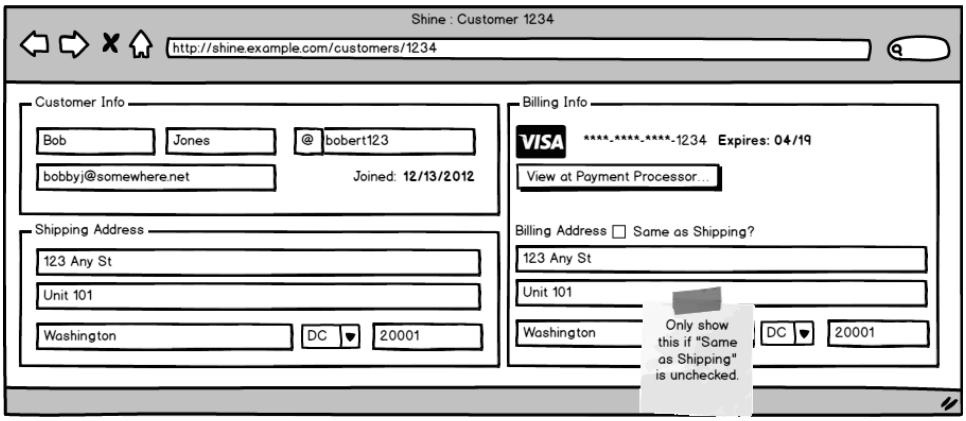
Create a Single-Page App Using Angular's Router

At this point, we are starting to feel confident. We experimented with some powerful features of Postgres, quickly made great-looking screens with Bootstrap, and created a dynamic user interface without a lot of code thanks to Angular. We also have rock-solid tests for every part of it. Now, it's time to level up.

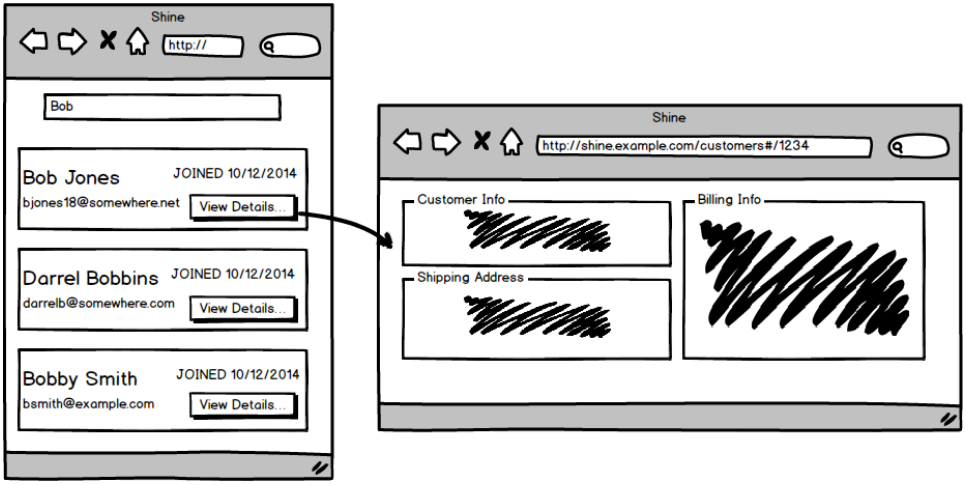
In this chapter, we're going to turn our simple customer search feature into a full-fledged *single-page app*, by learning about Angular's router, and how we can navigate users between pages, all within the browser. This will also allow us to learn how to manage our Angular views in different templates—just like we do in Rails—and have them play well with the Asset Pipeline.



We'll learn this by implementing the first bit of a feature we'll be building over the next few chapters. The feature is a detailed view of the customer's information, which includes more data than we saw on the result page, and requires pulling in data from many different sources, all viewable on one screen.



To get us started, we'll turn our existing Angular app into a *single-page application* that allows navigating from the search results to the detailed view (which will initially just be bare-bones).



Doing this will teach us about Angular's router—which is at the heart of any Angular app—as well as some Angular services for navigation and URL-parsing. We'll also learn how to serve up Angular templates using the Asset Pipeline. Once we've done that, we can add a second view to our Angular app, and set up navigation between the search results and the fledgling details view.

First, let's set up the router.

Using Angular's Router for User Navigation

Like Rails, Angular has a way to support navigation-by-URL. Because Angular's design ethos is based around flexibility, an Angular app isn't required

to use the router, though in most cases you would, and would set it up from the start. We didn't initially, simply to keep our intro to Angular as simple as possible and reduce the number of new concepts we had to learn.

Now, we'll need to convert our existing Angular app to use the router. This isn't a big task, so let's get to it.

Currently, our Angular app hard-codes both the view and the controller. The view is hard-coded by virtue of having all the HTML inside the <article ng-view='customers'>...</article> tags. The controller is hard-coded by the use of the ng-controller directive. We break that hard-coding with the router.

To accomplish this, we do three things. First, we install the *angular-route* module, which provides the router itself. Next, we'll configure it with a single route to match what we have now (i.e. that / renders the customer search view). Finally, we'll extract our existing view code into a standalone template, which will require using a Rails plugin called *angular-rails-templates* to help with Asset Pipeline integration.

Installing Angular's Router

As Angular is highly configurable and flexible, there's no router installed by default. Instead, we'll need to install the module *angular-route*, which we can do by adding it to Bowerfile and running rake bower:install.

complex-views/setup-angular-router/shine/Bowerfile

```
asset 'angular-route'
```

We'll need to bring it into the Asset Pipeline by adding it to app/assets/javascripts/application.js.

complex-views/setup-angular-router/shine/app/assets/javascripts/application.js

```
//= require angular
//= require angular-route
```

Although we've installed angular-route and configured it to be available via the Asset Pipeline, our Angular app won't be able to just start using it. Our Angular app needs to explicitly bring the module in as a dependency. This is more of Angular's configurability and flexibility—even though most apps will use Angular's router, not *all* apps will, so Angular wants us to opt in to use the router.

Each Angular app, when declared, has a list of dependent modules. If you recall, when we declared our app using angular.module, the second argument was an empty array:

```
var app = angular.module('customers',[]);
```

That argument is our app's list of dependent modules. It's currently empty, because we haven't needed anything other than what's provided by Angular. Now, we'll need to add angular-route to this array.

Unfortunately, it's not as simple as simple as adding "angular-route" to the array. In Angular, the module name for declaring dependencies doesn't have to be the same as the name of the module we downloaded. For official Angular-provided modules this is unfortunately the case.

By convention, the name to use in code for an Angular module can be derived by replacing the angular- with ng and camel-casing the remaining module name. That means that angular-route becomes ngRoute and so "ngRoute" is the string to add to our list of dependencies.

complex-views/setup-angular-router/shine/app/assets/javascripts/customers_app.js

```
var app = angular.module(
  'customers',
  [
    'ngRoute',
  ]
);
```

Now that we've installed angular-route, the next step is to use the router to configure a single route.

Configuring the Router

At a high level, configuring Angular's router is similar to adding routes to Rails' config/routes.rb. But Angular's router could be thought of us one level lower in abstraction. Rails deals in resources and actions, deriving the names of the controllers, urls and views. Angular's router deals directly at that level, requiring you to map URLs to controllers and views.

The route our app is using now is effectively /, even though we haven't set it anywhere. If you expected me to say that our Angular app was using the route /customers, you are not alone in getting confused about Angular routes vs. Rails routes.

When talking about an Angular app with Rails, there are three routes in play:

- The full url that the user sees in their browser. In our development environment, that's http:/localhost:3000/customers.
- The Rails route, which Rails uses to figure out which controller and view to use. In our case, that's /customers.

- The Angular route, which is *relative to the url that rendered the ng-app directive*. In our case, that's /.

Because the Rails route /customers caused the CustomersController to render the index action, which, in turn, rendered app/views/customers/index.html.erb, which contains the ng-app directive, all routes visible to the *Angular app* are relative to /customers and thus our app's route is /.

This may seem confusing, but you'll get used to it. It will make a bit more sense when we add the route for our new view in the next section, but let's configure our existing view to use the router.

Converting our Existing View to an Angular Template

To configure routes using angular-route, we need access to the object $routeProvider. This object has a function named when that allows us to configure the controller and template to use for a given url.

Like $http and $scope in our Angular app, and $controller in our tests, we need to arrange for Angular to pass $routeProvider into a function that we've defined, so that we can call functions on it. Angular apps provide a function config that will do just that.

In app/assets/javascripts/customers_app.js, you'll remember we assigned our Angular app to the variable app. We can then call config on app, passing it an array similar to the one we used to define our controller in *Creating Our First Angular Controller*, on page 87. That array has two elements. The first is the string "$routeProvider", which is the name Angular uses to store the object $routeProvider in its internal registry. The second is our function that will be given the actual $routeProvider as an argument.

complex-views/setup-angular-router/shine/app/assets/javascripts/customers_app.js

```
app.config([
          "$routeProvider",
  function($routeProvider) {

    // configure our routes here...

  }
]);
```

As you do more Angular, you'll find that most of your setup and configuration of components happens inside the function you pass to config. That function is a general place for you to set-up anything you need before the Angular app starts.

Now that we've configured our function to get $routeProvider passed into it, we'll tell it that we'd like to use the controller CustomerSearchController for the url /. We also need to give it a view template, so we'll use the name "customer_search.html" (which we'll fill in later with our view code). We can register this route this using the when function.

complex-views/setup-angular-router/shine/app/assets/javascripts/customers_app.js

```
app.config([
          "$routeProvider",
  function($routeProvider) {

    $routeProvider.when("/", {
      controller: "CustomerSearchController",
      templateUrl: "customer_search.html"
    });
  }
]);
```

We're almost done. The last step is to move our view template code out of app/views/customers/index.html.erb and into the file specified in our routing config, customer_search.html. To do *that*, we need to know where that file goes, and how Angular will access it an runtime. This requires configuring the Asset Pipeline to serve Angular templates.

Serving Angular Templates from the Asset Pipeline

The way we've configured our routes, Angular will ask the server for the file /customer_search.html, which will result in a 404. We could place our view file in public, but most Rails deployments do not serve static assets from that directory, preferring to serve those from a CDN or a web server. Ideally, we want our templates managed the same as all other assets—through the Asset Pipeline.

That this is an HTML file poses a bit of a challenge, especially if we are using a CDN. Because our assets would be served from a different server on a different domain than our Rails application, most browsers won't allow Angular to fetch the HTML file without configuring the CDN server for Cross-Origin Resource Sharing (CORS). That can be complicated to do and hard to debug.

Instead, we'll arrange for our templates to be compiled into JavaScript, so they'll be bundled in our application's asset bundle, the same as all our other JavaScript. We can do this with the Rails plugin *angular-rails-templates*. This gem will handled compilation of our HTML templates and the necessary configuration within Angular to make it all work.

First, we'll add angular-rails-templates to the Gemfile:

complex-views/setup-angular-router/shine/Gemfile
```
gem "angular-rails-templates"
```

After we install it with bundle install, we'll need to add it to app/assets/javascripts/application.js. Angular-rails-templates assumes our HTML templates are in app/assets/javascripts/templates and requires us to add that directory to our application's JavaScripts bundle as well.

complex-views/setup-angular-router/shine/app/assets/javascripts/application.js
```
//= require angular-route
//= require angular-rails-templates
//= require_tree ./templates
//= require_tree .
```

app/assets/javascripts/templates may seem like an odd place for templates. Angular-rails-templates allows you configure this location, but it's usually easier to stick with defaults when learning something new. So, even though it doesn't feel like the right place, we'll stick with it for now.

The JavaScript code included with angular-rails-templates is an Angular module that our Angular app must depend on (much the way it had to depend on the router). The name of the included module is "templates", so we need to modify our Angular app's dependencies like so:

complex-views/setup-angular-router/shine/app/assets/javascripts/customers_app.js
```
var app = angular.module(
  'customers',
  [
    'ngRoute',
    'templates'
  ]
);
```

With that configuration, we can move all the markup inside the outermost <article> tags into app/assets/javascripts/templates/customer_search.html. We'll also remove the use of ng-controller from app/views/customers/index.html.erb. Lastly, we need to add an HTML element there with the ng-view directive. This allows Angular to know where to put the view. Ultimately, app/views/customers/index.html.erb will look like this:

complex-views/setup-angular-router/shine/app/views/customers/index.html.erb
```
<article ng-app="customers">
  <div ng-view></div>
</article>
```

With all that configure out of the way, Shine should work the same way it did before. We can verify this by running our test suite and verifying that nothing has broken.

```
> bundle exec rake
Randomized with seed 46987
.........................

Finished in 6.4 seconds (files took 2.59 seconds to load)
26 examples, 0 failures
> bundle exec rake teaspoon
Teaspoon running default suite at http://127.0.0.1:50159/teaspoon/default
....

Finished in 0.03000 seconds
4 examples, 0 failures
```

Now that we've refactored our Angular app to use the router, we can easily add our detail view and the necessary links to allow the user to navigate to it.

Adding a Second View and Controller to our Angular App

Now that we've installed and configured Angular's router, we can add the detail view. In this section, we'll set up some navigation from our existing search results view to the detail view, which will initially just be a bare-bones view to validate that the navigation and back-end are working.

Our initial back-end will just expose the data we already have and do so in a minimal way. We're just trying to get navigation working, so we don't want to get bogged down in UI design or complex back-end integrations—we'll see that in the chapters to come.

To make our bare-bones view, we'll need to do three things: add navigation to a new route that uses a new view and controller, design the bare-bones version of the view, and have its controller make an AJAX call to Rails to get the customer details.

Navigating from One View to Another

In Rails, we create navigation by using a helper like `link_to`, which creates an `a` element in HTML. We give `link_to` an argument, which is usually the result of calling the Rails URL helper of the route you want the user to navigate to, for example `customers_path`. In Angular, it's a bit different.

The Angular view is designed much more like a user interface than a document, so to link the user from one route to another, we bind the click action on an element to a function. We saw something similar in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81, where we created a binding between

the change event of our text field and the search method of our controller, using the ng-change directive.

Here, we'll use the ng-click directive on a new button we'll add to each search result. That directive will bind to the function viewDetails, which will handle the navigation. First, let's see how that looks, and then we'll see how to implement viewDetails, before finally adding our new route.

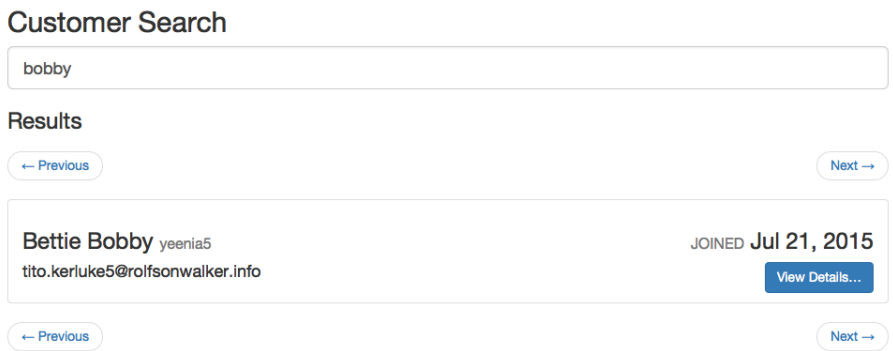
Adding the Navigation Button

We want the button that navigates the detail view to be named *View Details* and appear in the bottom right of the search result list view component. We can do that using the pull-right class, provided by Bootstrap, the same as we did for the *last login* field.

complex-views/navigation-to-new-view/shine/app/assets/javascripts/templates/customer_search.html

```
<ol class="list-group">
  <li class="list-group-item clearfix"
      ng-repeat="customer in customers">
    <div class="pull-right">
      <button class="btn btn-small btn-primary"
              ng-click="viewDetails(customer)">
        View Details…
      </button>
    </div>
    <h4>{{ customer.email }}</h4>
  </li>
</ol>
```

The search results now contain a button to allow viewing the customer details:



Note that we need to put the button *before* the email so that the email flows to the left of the button (since this is what pull-right does). Note also that the value for ng-click contains the call to viewDetails, including the argument customer.

When we click a button, Angular will pass in the right customer to our function, based on which result we clicked on.

Now, we'll see how to implement viewDetails by learning how to use an Angular service to navigate the user to a different route.

Navigating Between Routes

Angular has a service, named $location that allows us to change the route that the user is using. It has a function path, which takes, as an argument, the new route we want the user to navigate to.

Unlike with Rails, where the view ultimately contains the route embedded in an HTML attribute, Angular has a level of indirection between the user action and the routing, namely the function we gave to ng-click. It results in a bit more code, but keeps a better separation between the view and the code that responds to user actions.

Like the other Angular service we've seen—$http—$location can be passed to our controller function by placing the string "$location" in the argument list, and adding an argument to the function at the end of that list (see *Creating Our First Angular Controller*, on page 87 for a refresher on how this argument list works).

complex-views/navigation-to-new-view/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerSearchController", [
          '$scope','$http','$location',
  function($scope , $http , $location) {

   // rest of controller....
```

With the $location service getting passed in, we can now use it to route the user to the new view.

Routing Using the $location Service

As we mentioned, the function path on $location is the function we can use to route the user. path takes a string argument representing the path we want to route to. Because of Angular's flexibility, we can use any path we'd like.

Since we're getting a detail view of a single customer, the path should have the customer ID in it, so that the controller we'll write later can grab that ID out of the URL (just like we would in Rails), and request the details for the right customer. In Rails, we'd use the url /customers/1234 for the detail view of the customer with ID 1234.

Remember that the Rails portion of the URL for our Angular app is /customers, which is why the route for our customer search is just /. This makes sense, because our Angular app is just about customers. An additional /customers would be redundant. So, there's no sense in routing the user to the *Angular route* /customers/1234, because the URL in the user's browser would be http://localhost:3000/customers#/customers/1234. So, let's use /1234 as the route to our detail view.

Just as in Rails, Angular routes can have dynamic information in them. Angular even uses the same string format—:id. So, our route for the detail view will be /:id, and we'll use a new, unwritten, controller named CustomerDetailController.

complex-views/navigation-to-new-view/shine/app/assets/javascripts/customers_app.js

```
app.config([
          "$routeProvider",
  function($routeProvider) {
    $routeProvider.when("/", {
       controller: "CustomerSearchController",
      templateUrl: "customer_search.html"
    }).when("/:id",{
       controller: "CustomerDetailController",
      templateUrl: "customer_detail.html"
    });
  }
]);
```

Now, when we click the View Details button on one of our search results, say customer 1234, Angular will set the user's route to /1234 (which, remember, results in a complete url of http://localhost:3000/customers#/1234). Setting that route will cause the router to display the view in app/assets/javascripts/templates/customer_detail.html and use the controller CustomerDetailController.

Since neither of them exist yet, you'll see an error if you open the JavaScript console in your browser. In the next section, we'll create them.

Creating and Rendering the Bare-bones View

To close the loop on navigation and routing, we just want to create the most basic view we can of the customer data we have. We'll design the full detail view in the next section, and then learn how to retrieve the data it needs in the next chapter. For now, we'll just show the data we have.

To do that, we need to create the view markup itself, and then create a new controller to make the appropriate AJAX call to the server.

Creating the View Markup

Per our configuration of this route, the view markup should be in app/assets/javascripts/templates/customer_detail.html. Since we're going to re-work this view in the next section, we just need something to show dynamic data in it. So, we'll put each field we have on our customer inside the appropriate h tag.

complex-views/navigation-to-new-view/shine/app/assets/javascripts/templates/customer_detail.html

```
<article class="customer-details">
  <h1>Customer {{customer.id}}</h1>
  <h2>{{customer.first_name}} {{customer.last_name}}</h2>
  <h3>{{customer.email}}</h3>
  <h4>{{customer.username}}</h4>
  <h5>
    <small class="text-uppercase">Joined</small>
    {{customer.created_at | date}}
  </h5>
</article>
```

The way this view is written, it's expecting an object named customer to be exposed on the scope from the controller. Let's create the controller first, and then we'll make an AJAX call to the server to fill in $scope.customer.

Creating the Controller

Our controller ultimately needs to make an AJAX call to our Rails back-end, and to do that, it needs to extract the customer ID out of the route. Angular provides a service named $routeParams that can do just that. Since we'll be making an AJAX call, we know we'll also need the $http service, so we'll create our new controller, much as we did with CustomerSearchController, so that it accepts these as parameters.

complex-views/navigation-to-new-view/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerDetailController", [
          "$scope","$http","$routeParams",
  function($scope , $http , $routeParams) {

    // Make the AJAX call and set $scope.customer...

  }
]);
```

Our controller is all ready to be implemented.

Implementing the Detail View's Controller

To implement the controller we'll need to flesh out the function for our CustomerDetailController so that it makes an AJAX call to our Rails application, and we'll need to then implement the Rails side.

Since we know how to write unit tests for our controllers, let's use TDD to drive this test. This will demonstrate a difference between CustomerDetailController and CustomerSearchController, namely that CustomerDetailController makes the AJAX call when it loads, not when the user initiates an action.

First, we'll set up the same sort of boilerplate we saw before. This is mostly declaring variables we'll need in the test, but we'll also create the test customer we'll pretend is being sent to our controller from the back-end.

complex-views/navigation-to-new-view/shine/spec/javascripts/customers_app/controllers/customer_detail_controller_spec.js

```
describe("CustomerDetailController", function() {
  describe("Initialization", function() {
    var scope       = null,
        controller  = null,
        id          = 42,
        httpBackend = null,
        customer    = {
          id: id,
          first_name: "Bob",
          last_name: "Jones",
          username: "bob.jones",
          email: "bobbyj@somewhere.net",
          created_at: "2014-01-03T11:12:34"
        };

    beforeEach(module("customers"));
```

Next, we'll set up the controller. Much of this code will look similar to what we saw in Chapter 7, *Test This Fancy New Code*, on page 103, however there's a few differences that we've highlighted. First, you'll notice we're injecting $routeParams into our setup. This service allows our controller to access the dynamic parts of the route. So, when our route indicated :id as part of the route, the value at runtime will be available from $routeParams.id. So, we set that explicitly in our test.

You'll also notice that we've set our expectations on httpBackend *before* we call $controller. This is because calling $controller will execute the controller function, and we want that function to make the AJAX call automatically. So, we need to configure what call we're expecting and how it should respond first.

complex-views/navigation-to-new-view/shine/spec/javascripts/customers_app/controllers/customer_detail_controller_spec.js

```
beforeEach(inject(function ($controller,
                            $rootScope,
                            $routeParams,
                            $httpBackend) {
  scope       = $rootScope.$new();
```

```
    httpBackend = $httpBackend;

    $routeParams.id = id;

    httpBackend.when('GET','/customers/' + id + '.json').
                respond(customer);

    controller  = $controller("CustomerDetailController", {
      $scope: scope
    });
  }));
```

Lastly, we conduct our test. Our test is that our controller has set $scope.customer to have the same data as the customer we configured as a response to the AJAX call.

complex-views/navigation-to-new-view/shine/spec/javascripts/customers_app/controllers/customer_detail_controller_spec.js

```
it("fetches the customer from the back-end", function() {
  httpBackend.flush();
  expect(scope.customer).toEqualData(customer);
});
```

When we run our test via rake teaspoon, we should see a failure.

```
> bundle exec rake teaspoon
Starting the Teaspoon server...
* Listening on tcp://0.0.0.0:63241
Teaspoon running default suite at http://127.0.0.1:63241/teaspoon/default
F....

Failures:

  1) CustomerDetailController Initialization fetches the customer
     from the back-end
     Failure/Error: Error: No pending request to flush ! in
     http://127.0.0.1:63241/assets/angular-mocks/angular-mocks....

Finished in 0.03900 seconds
5 examples, 1 failure
```

This failure is expected, since we haven't implemented the controller yet.

To do that, we use $http in a fashion similar to what we saw in CustomerSearchController. Since we'll be hitting Rails to get the customer data, we'll want to trigger the show action of CustomersController, which means we want to build a url like /customers/1234.json.

As mentioned, $routeParams provides access to the specific ID from the Angular route, so we'll set that in the variable customerId. We then use that to build the

URL to pass to $http's get method and, inside the success callback, we set $scope.customer to the value we get (note also that we initialize it to {}, which is purely a stylistic choice to make it clear what we'll be exposing to the view).

complex-views/navigation-to-new-view/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerDetailController", [
          "$scope","$http","$routeParams",
  function($scope , $http , $routeParams) {
    var customerId = $routeParams.id;
    $scope.customer = {};

    $http.get(
      "/customers/" + customerId + ".json"
    ).success(function(data,status,headers,config) {
      $scope.customer = data;
    }).error(function(data,status,headers,config) {
      alert("There was a problem: " + status);
    });
  }
]);
```

With this code in place, our controller should now work as expected, when we run our test.

```
> bundle exec rake teaspoon
Starting the Teaspoon server...
Teaspoon running default suite at http://127.0.0.1:63375/teaspoon/default
.....

Finished in 0.03300 seconds
5 examples, 0 failures
```

The last thing to do is to implement the actual endpoint on the Rails side. Fortunately, this is Rails' bread-and-butter, so it's only a few lines of code.

First, we'll add the :show route in config/routes.rb

complex-views/navigation-to-new-view/shine/config/routes.rb

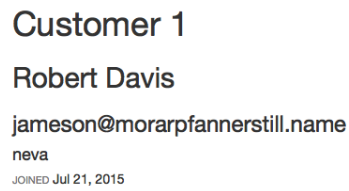
```
resources :customers, only: [ :index, :show ]
```

Then, we'll implement the show method in CustomersController.

complex-views/navigation-to-new-view/shine/app/controllers/customers_controller.rb

```
def show
  customer = Customer.find(params[:id])
  respond_to do |format|
    format.json { render json: customer }
  end
end
```

Now, when we run our app, search for users, and get the details of one of them, our bare-bones view works as expected.



Rails makes the middleware part of this dead simple, but we really didn't write much code at all. We had a few lines of configuration for the new route, a line of code to navigate there from the search results page, and a few more lines to fetch the customer details from the server. Angular provided all the heavy lifting with `$routeParams` and `$location`.

We've now learned the basics of making a single-page application using Angular. We can use Angular's built-in router to configure various routes, controllers, and views. We can serve those views up using the Asset Pipeline, and make use of built-in Angular services like `$routeParams` and `$location` to allow the user to navigate between them.

Next Up: Design Using Grids

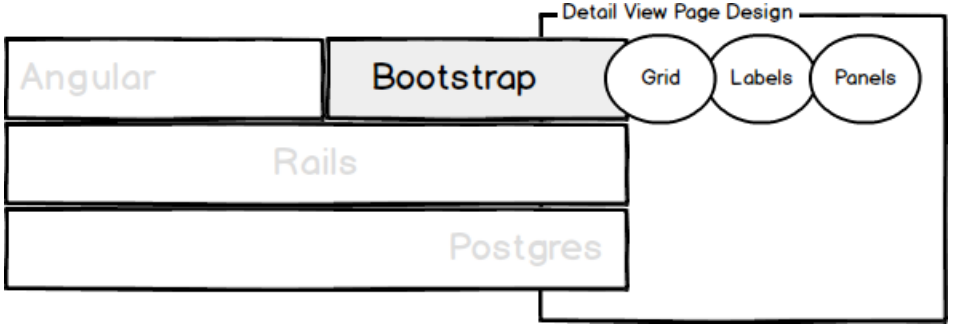
With this scaffolding set up, we can now set about the more difficult task of designing the actual detail view. In the next chapter, we'll learn about Bootstrap's *grid*, as well as various components Bootstrap provides.

These features of Bootstrap will unlock your inner web-designer and allow you to easily mock up, design, and implement complex user interfaces without writing any CSS.

CHAPTER 9

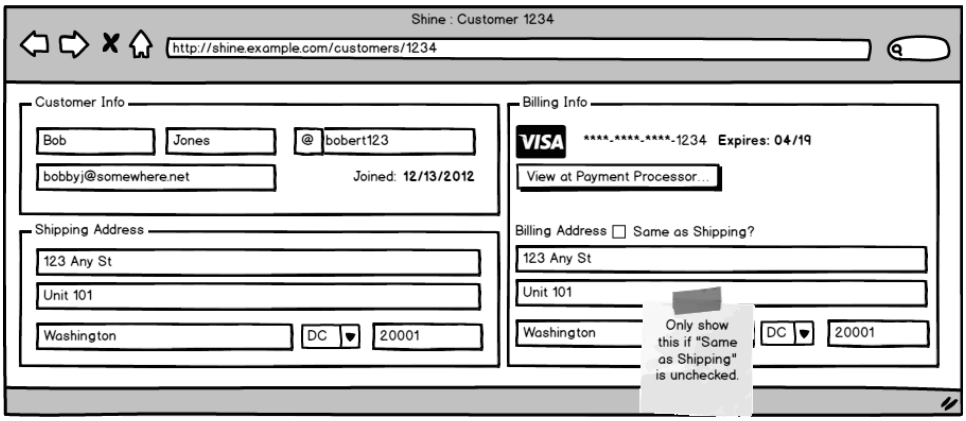
Design Great UIs With Bootstrap's Grid and Components

At the start of Chapter 8, *Create a Single-Page App Using Angular's Router*, on page 139, we saw a mock-up of the customer detail view. We're going to build this page now, and it's going to be a great way to learn about the true power of Bootstrap—its grid. We'll also learn about some of Bootstrap's many components, which will allow us to create a polished and visually-appealing user interface.



We'll learn this in three parts. First, we'll get some background on using grids in design. This will help us understand why Bootstrap is based on a grid, and how we can decompose any UI into grids, to make our work easier. Next, we'll lay out the customer detail screen we hinted at in the previous chapter, using Bootstrap's grid to make it easy. Finally, we'll use various Bootstrap components, such as panels and labels, to add some final polish to our UI.

We saw the mock-up in the previous chapter, but here it is again, so you know where we're headed:



By the end of the chapter we'll have a solid foundation in how to build user interfaces with Bootstrap, and even a bit of confidence in designing them ourselves. We'll sill be a far cry from being a real web designer, but we'll be able to do common, simple tasks on our own.

Let's learn about the grid and how it helps us create user interfaces.

The Grid: The Cornerstone of a Web Design

I don't know about you, but looking at a complex layout like the one we're going to build gives me a bit of anxiety. It's not just that CSS can be difficult to use, but it's also not immediately clear how to wrangle all the parts of this design.

Like functional decomposition in programming, a *grid* is how we can decompose a user interface into smaller parts. We can focus on each part of our design, and rely on the grid to keep everything looking visually cohesive.

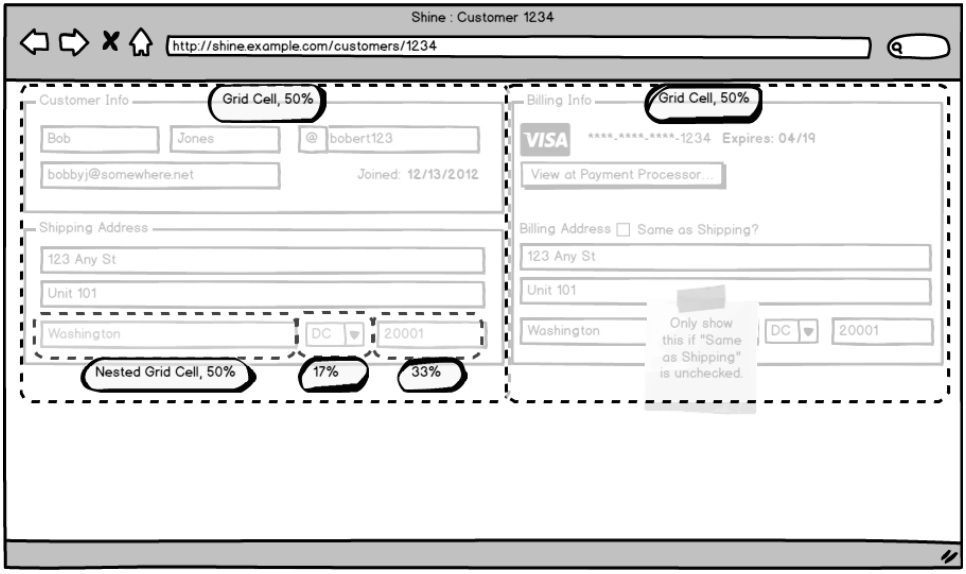
A grid is more or less what it sounds like—a means of aligning elements along a fixed horizontal and/or vertical axis. You might not have realized it, but you've been using a grid already. By just using Bootstrap's default styles and form classes, the forms we created in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17 (as well as the search results from Chapter 5, *Create Clean Search Results with Bootstrap Components*, on page 69 are using a *horizontal grid*. This means that each row of information is spaced in a particular way to make the text and other elements pleasing and orderly.

For the view here, we need a vertical grid. This is how we'll achieve most of the layout we want. Bootstrap provides a set of CSS classes that allow us to

create a grid. Under the covers, it uses CSS floats, which can get messy quickly, but Bootstrap's grid abstracts that away.

Bootstrap's grid has 12 columns. You can combine columns in any way you like to make larger columns, without disrupting the flow and spacing of the grid. For example, you could have a two column layout where the first column is 25% of the entire width, leaving the remaining 75% for the second column, or you could have three columns of equal size, each taking 33% of the available width.

If you think about your design in terms of rows and columns, you can start to see the grids pop out of our design. You can see two grid cells, each taking 50% of the available space, for the main columns of our design, but you can also see a grid nested in each form. The city/state/zip part of the shipping address could be thought of as a grid where the city takes 50% (six grid cells), the state takes 17% (or two grid cells), and the zip code takes the remaining 33% (or four grid cells).



What this means it that, if have sufficiently generic CSS classes that allow us to place content into grid cells, and to place those cells into rows, and to nest grids within each other, all with proper padding, spacing, and margins, we can break up any design into a series of grids.

This is exactly what Bootstrap's grid system will do.

Using Bootstrap's Grid

Bootstrap's grid is quite powerful, especially if you've never used one before. In this section, we'll build the layout for our view using Bootstrap's grid. As we saw in the previous section, our layout starts with two equal-sized grid cells: one that holds the customer info and shipping address, and the other that holds the billing information.

First, we'll create these cells, which will demonstrate the various CSS classes needed to enable Bootstrap's grid. Then, we'll see how the grid can nest within itself to lay out the customer info and shipping address as a grid-within-a-grid.

Laying out the Two Main Columns

The most obvious grid in our design is one that holds the two main columns, each taking half the available space. To do this, we'll create two nested `div` tags inside a parent `div`, giving each the appropriate CSS class—provided by Bootstrap—to layout it all out in a grid (see *Why are we sometimes using div and sometimes not?*, on page 158 for some detail on why we're using `div`s).

The outer *div* should have the class `row`, which tells Bootstrap we're going to place columns inside it. The `div`s inside the `row` should be given a class named `col-md-X` where `X` is the number of columns, out of 12, that this particular column should take up. In our case, we want two equal-sized columns. Since Bootstrap provides 12 total columns per row, we want each of *our* columns to take up six of Bootstrap's. Thus, each `div` will get the class `col-md-6` .

Why are we sometimes using div and sometimes not?

Before HTML 5, there were not a lot of standard elements we could use to describe our content. As a result, the `div` element came into favor as the way to organize content, particularly for targeting by CSS styling. With the advent of HTML5, there are now more meaningful elements available, such as `article`, `section`, `header`, and `footer`.

Because of this, the W3C recommends that the div be used only as a last resort[a], when no other elements are available.

What this means is that we want to use the right tags when describing our content, regardless of the visualization we are going for. We can then use `div` tags to achieve the layouts that we want. Since `div` is semantically meaningless, it allows anyone reading our view templates to see clearly what parts of the view are for styling and layout and what parts are for organizing the content.

So, the general rule of thumb is to use `div`s in cases where you need an element to style against, and *not* as a way to describe content.

a. http://www.w3.org/TR/html5/grouping-content.html#the-div-element

We can add this markup to app/assets/javascripts/templates/customer_detail.html, replacing the bare-bones markup we had there from the last section.

```
<form><div class="row">
  <div class="col-md-6">
    <h1>Customer</h1>
  </div>
  <div class="col-md-6">
    <h1>Billing Info</h1>
  </div>
</div></form>
```

If you bring this up in your browser, you'll see that our two headings are shown side-by-side.

Now, let's tackle the content *inside* these columns. As we saw above, we can think of each section of our page has having a nested grid inside this one. Bootstrap's grid works exactly this way.

Building Forms Using a Grid-within-a-Grid

Bootstrap's grid is not a fixed width, so whenever you write , Bootstrap will divide up the grid in that row based on the available space. This is a powerful feature of the grid system. Much like how we decompose complex objects into smaller ones to make our code easier to understand, we can decompose larger views into smaller ones using the grid.

By thinking of each page's component as a grid, we can design that component without worrying about where it is on the page. Bootstrap's grid components will make sure it works.

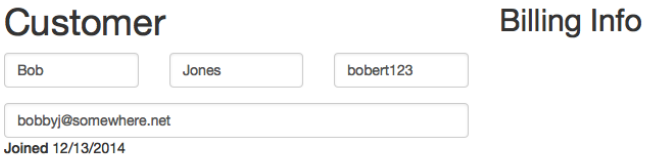
Let's style the customer info section using the grid. We can see from our mock-up that we have three rows, and the first row has three columns. Since the second and third rows just have one column that takes up the entire row, we don't need to use the grid markup for them. So, we just need to create a grid for the first row.

We'll be using the form classes we saw in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17, so hopefully this will look

familiar. Since the first name, last name, and username are all about the same size data-wise, we can create three equal-sized columns for them. Since Bootstrap's grid is 12 columns, we want each of our columns to take up four of Bootstrap's columns, so we'll use the class col-md-4 on each div.

```
<form><div class="row">
  <div class="col-md-6">
    <h1>Customer</h1>
    <div class="row">
      <div class="col-md-4">
        <div class="form-group">
          <label class="sr-only" for="first-name">First Name</label>
          <input type="text" class="form-control"
                 name="first-name" value="Bob">
        </div>
      </div>
      <div class="col-md-4">
        <div class="form-group">
          <label class="sr-only" for="last-name">Last Name</label>
          <input type="text" class="form-control"
                 name="last-name" value="Jones">
        </div>
      </div>
      <div class="col-md-4">
        <div class="form-group">
          <label class="sr-only" for="username">Username</label>
          <input type="text" class="form-control"
                 name="username" value="bobert123">
        </div>
      </div>
    </div>
    <div class="form-group">
      <label class="sr-only" for="email">Email</label>
      <input type="text" class="form-control"
             name="email" value="bobbyj@somewhere.net">
    </div>
    <label for="joined">Joined</label> 12/13/2014
    <h2>Shipping Address</h2>
```

Note that we used form-group on a different element as col-md-4. This isn't technically required, but is good to separate concerns. Generally, you want classes used for your grid to be separate from classes used for styling, so that you can be sure your grid not get messed up by styling classes, *and* allow us to add additional styling later on without worrying about how the grid will affect it. If we look at what we've done in our browser, we can see that it looks pretty good!



Shipping Address

Up to now, we've created grid cells that are all the same size. Let's lay out the shipping address part of our page, which requires that some of the grid cells be larger than others.

Using Grid Cells of Different Sizes

The main columns of our view, as well as the user info, all used grid cells of the same size. That won't work for the address views, since the city, state, and zipcode are all different sizes. It also won't work for the credit-card info view, because the card number and type can be quite long, but we still need room for the button that will (eventually) take the user to the payment processor's page for the customer's card.

In this section, we'll style both of these views using different grid sizes. The result will be a cohesive, well laid-out page, even though the grid cells aren't the same size.

First, we'll start with the addresses.

Laying out the Addresses

In a typical US address, the state code is very short—2 characters—and the zipcode is typically five or nine characters. So, let's make a column for the city—which is usually longer—that takes up half the available space. In the remaining half, we'll give the zipcode two-thirds of the remaining space, leaving the last third for the state code.

That works out to six columns for the city, two for the state code (since 6 ÷ 3 is 2), and the remaining four for the zipcode (the two street address lines can use up and entire row each, so we don't need the grid markup for them).

```
<h2>Shipping Address</h2>
<div class="form-group">
  <label class="sr-only" for="street-address">
    Street Address
  </label>
  <input type="text" class="form-control"
         name="street-address" value="123 Any St">
</div>
```

```
<div class="form-group">
  <label class="sr-only" for="street-address-extra">
    Street Address Extra
  </label>
  <input type="text" class="form-control"
         name="street-address-extra" value="Unit 101">
</div>
<div class="row">
  <div class="col-md-6">
    <div class="form-group">
      <label class="sr-only" for="city">City</label>
      <input type="text" class="form-control"
             name="city" value="Washington">
    </div>
  </div>
  <div class="col-md-2">
    <div class="form-group">
      <label class="sr-only" for="state">State</label>
      <input type="text" class="form-control"
             name="state" value="DC">
    </div>
  </div>
  <div class="col-md-4">
    <div class="form-group">
      <label class="sr-only" for="zip">Zip</label>
      <input type="text" class="form-control"
             name="zip" value="20001">
    </div>
  </div>
</div>
```

We can repeat this markup for the billing address, which just leaves us the credit card info to style.

Laying out the Credit Card Info

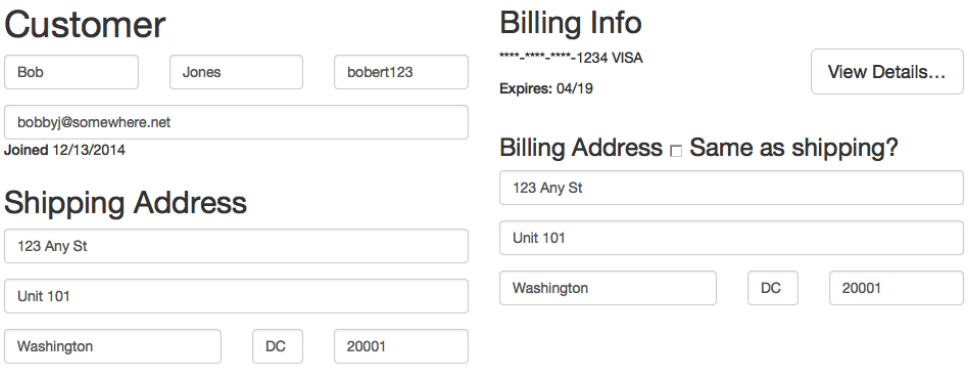
The credit card area has two distinct parts: the card info itself, and the button that will link the user to the payment processor's page for that card. We'll give the card info seven of the 12 columns, and use the remaining five for the button (these values might seem somewhat magic, and they were arrived at experimentally—feel free to change them and see how it affects the layout, making sure everything in the row adds up to 12).

```
<div class="col-md-6">
  <h2>Billing Info</h2>
  <div class="row">
    <div class="col-md-7">
      <p>
        ****-****-****-1234
        VISA
```

```
      </p>
      <p>
        <label>Expires:</label> 04/19
      </p>
    </div>
    <div class="col-md-5 text-right">
      <button class="btn btn-lg btn-default">
        View Details…
      </button>
    </div>
  </div>
  <h3>Billing Address <input type="checkbox"> Same as shipping?  </h3>
    <!-- Same markup as used for the shipping adddress -->
```

Note that we've used the helper class text-right on the button so that it aligns to the right side of the grid, and thus stands apart from the card info. Previously, we used pull-right to achieve this in our search results. Thinking back now, you might have more success using a grid for each result, rather than using floats. Fortunately, it's easy enough to try on your own!

Now that we've placed everything in a grid, we can see that the page is really starting to come together.



Bootstrap's grid is probably its single most useful feature. Before I knew about grids as design tools, and before I'd used one like Bootstrap's for creating them in CSS, a design like this would've taken me a very long time to create. Depending on the time pressure I was under, I might've opted with a different, less-optimal design that was easier to build, simply because my ability to create the right view was hampered by my lack of knowledge and lack of tools.

Now that the layout is solid, let's go through our view and polish up a few of the rough edges.

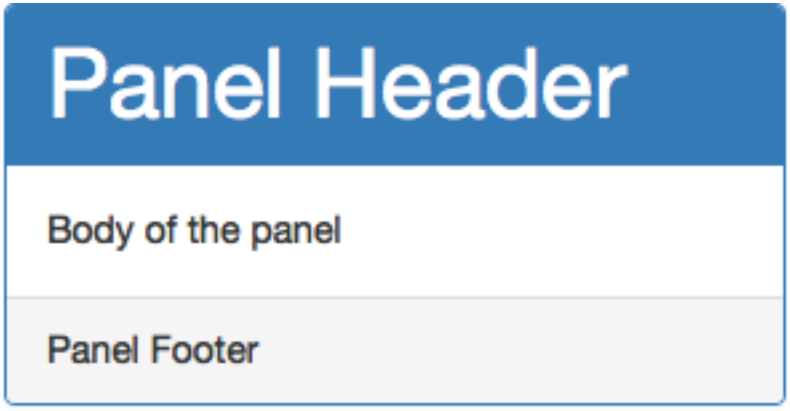
Adding Polish with Bootstrap Components

Our view looks pretty good—certainly better than what we might achieve in the same amount of time without Bootstrap—but it could be better. For example, the header text is a bit too large, there's not a clear distinction between the three sections of the view, and the credit-card info is a bit jumbled, since all the text uses the same size and weight font.

We'd like to distinguish parts of the view to make it easier for the user to visually navigate. If you look through Bootstrap's documentation, you can get some inspiration as to how we can do this. The trick with complex forms is allow the user to navigate all the data with their eye. We can get a long way with the *panel* component, which is a box surrounding our content along with a header and footer.

Using Panels

A panel looks like so



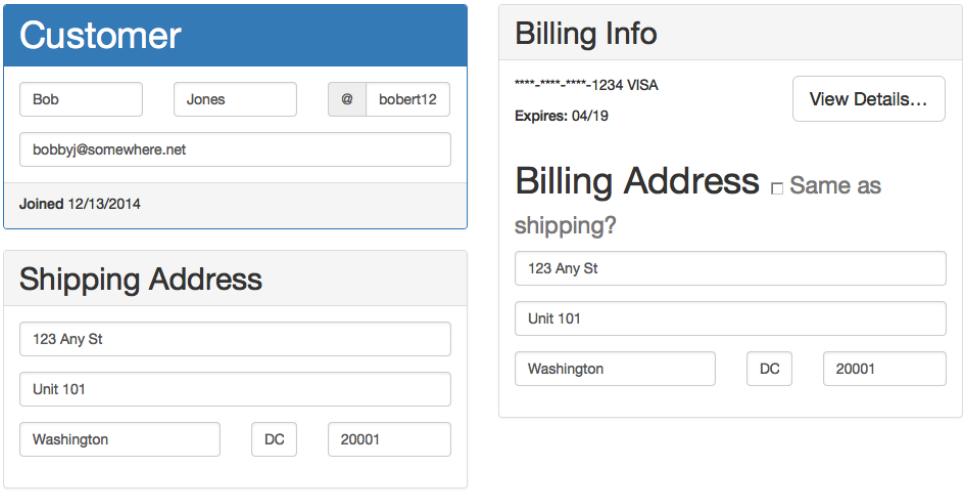
It can be created with Bootstrap with markup like so:

```
<article class="panel panel-primary">
<header class="panel-heading">
  <h1>Panel Header</h1>
</header>
<section class="panel-body">
  Body of the panel
</section>
<footer class="panel-footer">
  Panel Footer
</footer>
</article>
```

Let's put each of the three sections of our screen inside its own panel. Panels can be given different styles, so let's make the customer info panel styled differently from the other two, so it stands out more clearly. Each panel requires two classes: panel and then a second one that determines its style.

We'll use panel-primary for the customer info, which will use an inverse color scheme for the header, and panel-default for the other two. Finally, we'll move the *joined* field inside the customer panel's footer. As we'll see, this value won't be editable by the user, so moving it to the footer will reinforce this fact.

The result looks pretty good:



Next, let's improve the credit card info section. If we could have the card type more distinct from the card number and expiration, that would help users quickly distinguish this information. We can do that use *labels*.

Highlight Information with Labels

A *label* is a Bootstrap component that renders text inside a colored box with an inverse color-scheme (not to be confused with the HTML element label, which is used to label fields in a form). In lieu of finding and downloading images for each credit-card type, we can simply put the credit card type inside a label, and it'll stand out.

Labels, like panels, take two classes: a label class, and a decorative one that controls the color. We'll use label-success which will create a green label.

```
<div class="col-md-7">
<p>
  ****-****-****-1234
```

```
    <span class="label label-success">VISA</span>
  </p>
  <p>
    <label>Expires:</label> 04/19
  </p>
  </div>
```

With just this markup, the credit card type stands out pretty well.

Lastly, we'll make a few adjustments to the typography, as the headers are a bit too large.

Using h Classes To Tame Typography

The headers in our view are all a bit too large and, although our markup is semantically correct, some subheadings are larger than others. Further, the masked credit card number is a bit too small.

We can use the `h` classes provided by Bootstrap to manage the size of our headings (it may seem strange, but these classes allow us to keep the semantically correct element without inheriting their visual size). We can also use them on the `p` tag surrounding the credit card number to make it stand out a bit.

We'll see the entire markup in a moment, but here's an example of what we're talking about:

```
<article class="panel panel-default">
<header class="panel-heading">
  <h2 class="h4">
    Billing Info
  </h2>
</header>
<!-- ... -->
        <p class="h4">
          ****-****-****-1234
          <span class="label label-success">VISA</span>
        </p>
```

This sort of thing is a bit more art than science, so the values I've chosen here represent what looks right to me. The great thing about Bootstrap is that it's easy to play around with this stuff, and whatever you do will end up looking pretty decent, thanks to the horizontal grid the underlies all of the type.

One last bit of polish we'd like to add is to distinguish the username from the first and last names in the Customer Info section. You'll notice on our mockup, the username was preceded with a @ symbol. Bootstrap makes this easy using *form add-ons*.

Form Add-ons

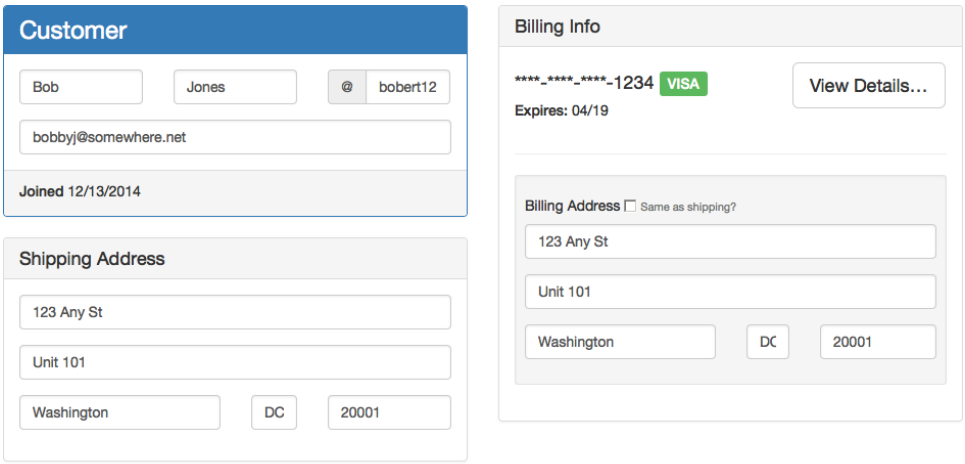
Often, a symbol prepended (or appended) to a value can give it enough context for users to understand what it means, without using a label. This can be handy on dense pages like our customer detail view. Since the first row of the customer info section is just three equal-sized strings, it might not be clear what they mean.

If we prepend the username with @ (similar to what Twitter does for mentioning someone in a tweet), that can be enough context for users to know that the third field is the username, and the first two are the first and last names, respectively.

Bootstrap provides the class `input-group-addon` that will do this in a pleasing way. We just surround the form element with an `input-group` and create an inner `div` with the class `input-group-addon` that contains the text we'd like prepended (you can place that `div` after the element to append it, instead).

```
<div class="input-group">
  <div class="input-group-addon">@</div>
  <input type="text" class="form-control"
         name="username" value="bobert123">
</div>
```

With all of these tweaks in place, the rendered form looks polished and professional, embodying the spirit of the mock-up.



HTML markup is quite verbose (especially since we have to heavily wrap it to fit the margins in this book), so we've only seen bits and pieces. The entire screen's markup is as follows:

complex-views/ui/shine/app/assets/javascripts/templates/customer_detail.html

```
<form>
  <div class="row">
    <div class="col-md-6">
      <article class="panel panel-primary">
        <header class="panel-heading">
          <h1 class="h3">
            Customer
          </h1>
        </header>
        <section class="panel-body">
          <div class="row">
            <div class="col-md-4">
              <div class="form-group">
                <label class="sr-only" for="first-name">
                  First Name
                </label>
                <input type="text" class="form-control"
                       name="first-name" value="Bob">
              </div>
            </div>
            <div class="col-md-4">
              <div class="form-group">
                <label class="sr-only" for="last-name">Last Name</label>
                <input type="text" class="form-control"
                       name="last-name" value="Jones">
              </div>
            </div>
            <div class="col-md-4">
```

```
          <div class="form-group">
            <label class="sr-only" for="username">Username</label>
            <div class="input-group">
              <div class="input-group-addon">@</div>
              <input type="text" class="form-control"
                     name="username" value="bobert123">
            </div>
          </div>
        </div>
      </div>
      <div class="form-group">
        <label class="sr-only" for="email">Email</label>
        <input type="text" class="form-control"
               name="email" value="bobbyj@somewhere.net">
      </div>
    </section>
    <footer class="panel-footer">
      <label for="joined">Joined</label> 12/13/2014
    </footer>
  </article>
  <article class="panel panel-default">
    <header class="panel-heading">
      <h2 class="h4">
        Shipping Address
      </h2>
    </header>
    <section class="panel-body">
      <div class="form-group">
        <label class="sr-only" for="street-address">
          Street Address
        </label>
        <input type="text" class="form-control"
               name="street-address" value="123 Any St">
      </div>
      <div class="row">
        <div class="col-md-6">
          <div class="form-group">
            <label class="sr-only" for="city">City</label>
            <input type="text" class="form-control"
                   name="city" value="Washington">
          </div>
        </div>
        <div class="col-md-2">
          <div class="form-group">
            <label class="sr-only" for="state">State</label>
            <input type="text" class="form-control"
                   name="state" value="DC">
          </div>
        </div>
        <div class="col-md-4">
```

```
          <div class="form-group">
            <label class="sr-only" for="zip">Zip</label>
            <input type="text" class="form-control"
                   name="zip" value="20001">
          </div>
        </div>
      </div>
    </section>
  </article>
</div>
<div class="col-md-6">
  <article class="panel panel-default">
    <header class="panel-heading">
      <h2 class="h4">
        Billing Info
      </h2>
    </header>
    <section class="panel-body">
      <article>
        <div class="row">
          <div class="col-md-7">
            <p class="h4">
              ****-****-****-1234
              <span class="label label-success">VISA</span>
            </p>
            <p class="h5">
              <label>Expires:</label> 04/19
            </p>
          </div>
          <div class="col-md-5 text-right">
            <button class="btn btn-lg btn-default">
              View Details…
            </button>
          </div>
        </div>
      </article>
      <hr>
      <article class="well well-sm">
        <header>
          <h1 class="h5">
            Billing Address
            <small>
              <input type="checkbox"> Same as shipping?
            </small>
          </h1>
        </header>
        <div class="form-group">
          <label class="sr-only" for="street-address">
            Street Address
          </label>
```

```
            <input type="text" class="form-control"
                   name="street-address" value="123 Any St">
          </div>
          <div class="row">
            <div class="col-md-6">
              <div class="form-group">
                <label class="sr-only" for="city">City</label>
                <input type="text" class="form-control"
                       name="city" value="Washington">
              </div>
            </div>
            <div class="col-md-2">
              <div class="form-group">
                <label class="sr-only" for="state">State</label>
                <input type="text" class="form-control"
                       name="state" value="DC">
              </div>
            </div>
            <div class="col-md-4">
              <div class="form-group">
                <label class="sr-only" for="zip">Zip</label>
                <input type="text" class="form-control"
                       name="zip" value="20001">
              </div>
            </div>
          </div>
        </article>
      </section>
    </article>
  </div>
</form>
```

Note that we *still* haven't written any CSS. We were able to create a highly complex form, displaying a lot of data, in a clean and easy-to-read way, using just a few simple classes, coupled with Bootstrap's grid system.

In this chapter, we saw several new features of Bootstrap, notably its grid system, but also some UI components that allowed us to polish our UI quickly and easily. What you should take away from this is what we mentioned at the start of the section: these are tools to allow you to design and build in the browser.

This takes away much of the friction in getting started on a new user interface. Armed with just Bootstrap, you can create complex interfaces quickly, and iterate on them as you find the most optimal design.

Next Up: Populating the View Easily and Efficiently

This chapter was focused on the top of the stack: the view. We learned how to configure our Angular app to allow for routing to different views, backed by different controllers. We also learned how easy it is to design a complex UI for our customer detail view.

In the next chapter, we'll see how to bring the actual data into the UI we've created. We'll use a feature of Postgres called *materialized views* to make querying the data from Rails very easy. We'll also see how Angular's asynchronous nature allows us to easily implement our UI using data from our database as well as from our third-party payment processor's system.

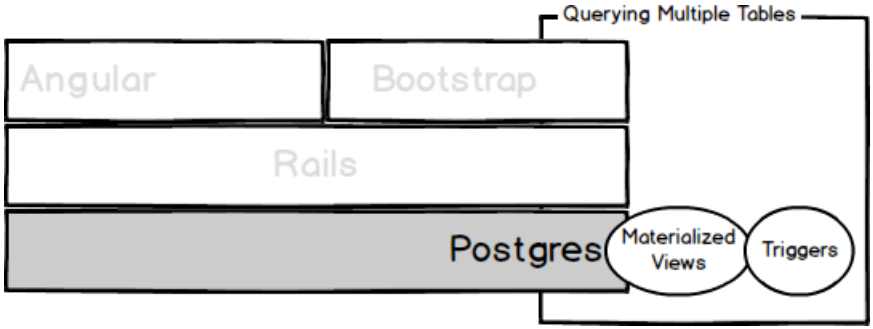
CHAPTER 10

Cache Complex Queries Using Materialized Views

When we need to query data stored in several tables, we often have to make a trade-off. Either we keep our code simple by using ActiveRecord—which makes several queries to the database—or we make our code more complex by, using a single, efficient query that's specific to our needs. In both cases, performance likely an issue, since we're pulling back a lot of data. Postgres provides a feature called *materialized views* that provides clean code, allows us to access our data with a single query, but exhibits high performance.

In this chapter, we'll continue our running example of displaying a customer's details. It will require us to fetch data from five different tables. We'll see how the idiomatic *Rails Way* of using ActiveRecord results in seven queries to the database, and how a single, more direct query, results in convoluted code, but potentially better performance.

We'll then create a materialized view of our single query. A materialized view is an abstraction Postgres provides that essentially creates a table with the results of our query, that we can access directly, as if it were a regular table in the database. This allows us to use ActiveRecord in our Rails code in an idiomatic way, but also get extremely high performance: the best of both worlds! We'll also see how we can use *database triggers* to keep the view up-to-date.



First, let's examine the performance characteristics of the two approaches—using ActiveRecord versus using a single, more complex query—by learning about the tables we need to access.

Understanding the Performance Impact of Complex Data

In Chapter 9, *Design Great UIs With Bootstrap's Grid and Components*, on page 155, we build the user interface for the data we'll be querying here. In addition to the data we've already seen in the `CUSTOMERS` table, we also need to display the customer's billing address, shipping address, and credit card information. The credit-card information is stored elsewhere (something we'll deal with in the next chapter), so what we're querying here will be the customer's billing and shipping addresses.

We've already seen the `CUSTOMERS` table, and it doesn't include either of these pieces of data. As you'll recall from Chapter 4, *Use Fast Queries with Advanced Postgres Indexes*, on page 49, our hypothetical company has tables in a shared database, and Shine will be able to access them. In this case, we'll assume that the tables we need to access billing and shipping addresses are available to us.

We're going to see how well both ActiveRecord and a single query perform when accessing these tables. First, let's look at their structure so we know what we're dealing with.

The Tables We'll Query

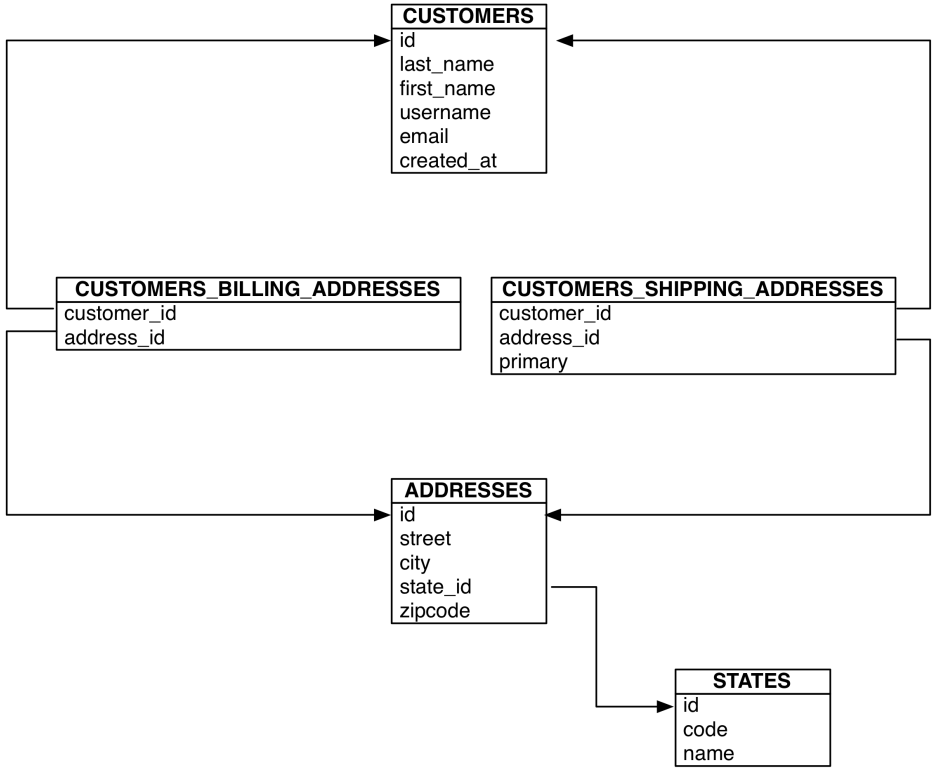
In our hypothetical system, all addresses are stored in the table `ADDRESSES`. This table contains fields you'd expect in a US-style address, however the code for the US state is not stored here. It's stored in `STATES`, and `ADDRESSES` references that table via the column `STATE_ID`.

Since all addresses are stored in `ADDRESSES`, we need to know which of those are for billing and which are for shipping. To determine that, we have two

join tables[1] called CUSTOMERS_BILLING_ADDRESSES and CUSTOMERS_SHIPPING_ADDRESSES. Each contains a column named CUSTOMER_ID, which identifies the customer, and a column ADDRESS_ID, identifying the address.

To further complicate things, the system allows a customer to have many shipping addresses, but the UI we've created only shows one. Fortunately, CUSTOMERS_SHIPPING_ADDRESSES contains a third field, PRIMARY, that is true for the *primary* shipping address for a customer. This is the one we'll use in our UI.

The tables and their relationships to one another are shown here:



There are two main ways to access this data: we can map these tables to ActiveRecord objects, model the relationships, and navigate them in our controller, or we can pull the data back with one big query. Let's examine the performance of each of these approaches.

1. https://en.wikipedia.org/wiki/Junction_table

Performance Using ActiveRecord

Since we're ultimately not going to use ActiveRecord to model this data, I'm going to skip the code you'd need to do so, but you can imagine how it would look. We would be able to write code like so to access all the data we need:

```
def show
  customer = Customer.find(params[:id])
  respond_to do |format|
    format.json do
      render json: {
                customer: customer,
        shipping_address: customer.shipping_address,
         billing_address: customer.billing_address,
      }
    end
  end
end
```

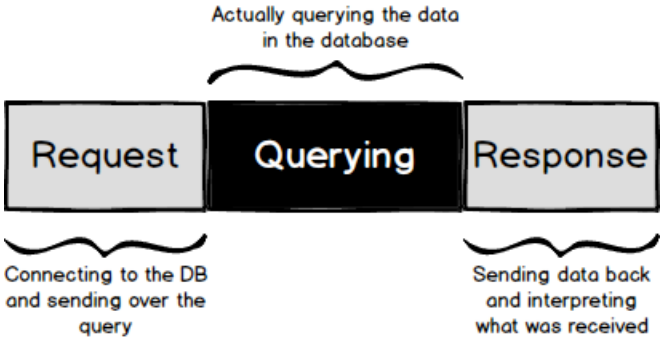
This code is going to run seven queries: one to pull back data from CUSTOMERS, one to query CUSTOMERS_BILLING_ADDRESSES, followed by a query to ADDRESSES and STATES. It will then query CUSTOMERS_SHIPPING_ADDRESSES, followed by second queries to ADDRESSES and STATES. Let's use EXPLAIN ANALYZE to see how well these will perform.

```
> explain analyze select * from customers where id = 2000;
QUERY PLAN
------------------------------------------------------------------------
 Index Scan using customers_pkey on customers
   (cost=0.42..8.44 rows=1 width=79)
   (actual time=1.298..1.299 rows=1 loops=1)
   Index Cond: (id = 2000)
 Planning time: 5.120 ms
 Execution time: 1.387 ms

> explain analyze select * from customers_billing_addresses
                  where customer_id = 2000;
QUERY PLAN
------------------------------------------------------------------------
  Index Scan using customers_billing_addresses_customer_id
                   on customers_billing_addresses
   (cost=0.42..8.44 rows=1 width=12)
   (actual time=0.105..0.106 rows=1 loops=1)
   Index Cond: (customer_id = 2000)
 Planning time: 0.241 ms
 Execution time: 0.130 ms

> explain analyze select * from customers_shipping_addresses
                  where customer_id = 2000;
QUERY PLAN
```

```
------------------------------------------------------------------------
 Index Scan using customers_shipping_addresses_customer_id
                  on customers_shipping_addresses
   (cost=0.42..8.48 rows=3 width=13)
   (actual time=0.118..0.119 rows=2 loops=1)
   Index Cond: (customer_id = 2000)
 Planning time: 0.519 ms
 Execution time: 0.171 ms

> explain analyze select * from addresses where id = 2000;
QUERY PLAN
------------------------------------------------------------------------
 Index Scan using addresses_pkey on addresses
   (cost=0.43..8.45 rows=1 width=47)
   (actual time=1.238..1.240 rows=1 loops=1)
   Index Cond: (id = 2000)
 Planning time: 0.984 ms
 Execution time: 1.267 ms

> explain analyze select * from states where id = 5;
QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on states
   (cost=0.00..7.48 rows=1 width=16)
   (actual time=0.038..0.114 rows=1 loops=1)
   Filter: (id = 5)
   Rows Removed by Filter: 357
 Planning time: 0.355 ms
 Execution time: 0.143 ms
```

All of these queries perform well, which isn't surprising as they are all querying against the primary key of the tables in question. We can see this by the information `Index Scan` in the query plan (though, interestingly, the queries against `STATES` use a full table scan—`Seq Scan`—presumably because this table is so small, it's faster than using the primary key's index). If we assume the timing reported by our query plans is a reasonable average, this means the total of these seven queries is about 13 milliseconds.

The query time isn't the only time it takes to access this data, however. Each request requires network time, as does each response. This is commonly called the *network round-trip*.



Since we're running seven queries, that's seven network round-trips. If our network is slow, this can have a significant impact on our overall response time.

If we could get the data in a single query, our performance would be less tied to the network's performance, and our system would be more predictable (and possibly faster, if our network is habitually slow). Let's craft that query and see how it performs.

Performance using SQL

To get this data using one query, we'll need to do a lot of joins. Let's build up the query first, and then see how it performs.

Crafting the Query

If you are comfortable with database joins, you can skip this section, but I've found that joins across many tables (especially when we need to join the same table twice, as we will for `ADDRESSES`) can be tricky, so it might help to see the query built piece by piece.

First, we need to itemize the fields we want back. In our case, we want all the fields from `CUSTOMERS` that we've been using, all the fields in `ADDRESSES` for both the billing and the shipping address, and the state codes from `STATES` for the same. The `SELECT` part of our query looks like this:

materialized-views/data-model/shine/db/customer_detail_view.sql

```
SELECT
  customers.id             AS customer_id,
  customers.first_name     AS first_name,
  customers.last_name      AS last_name,
  customers.email          AS email,
  customers.username       AS username,
  customers.created_at     AS joined_at,
```

```
  billing_address.id       AS billing_address_id,
  billing_address.street   AS billing_street,
  billing_address.city     AS billing_city,
  billing_state.code       AS billing_state,
  billing_address.zipcode  AS billing_zipcode,
  shipping_address.id      AS shipping_address_id,
  shipping_address.street  AS shipping_street,
  shipping_address.city    AS shipping_city,
  shipping_state.code      AS shipping_state,
  shipping_address.zipcode AS shipping_zipcode
FROM
  customers
```

Note a few things here. Because all addresses are stored in ADDRESSES, we'll need to join against that table twice (as we'll see). That means that we need to know which join we're referencing—the join that brings in the shipping address, or the join that brings in the billing address. To know *that*, we'll be aliasing[2] the ADDRESSES tables in each join to shipping_address and billing_address so we know what we're referring to.

Now that we know the data we're bringing back, we need to add the necessary joins to get it. First, we'll join CUSTOMERS against CUSTOMERS_BILLING_ADDRESSES, because that's how we'll eventually get to the actual address

materialized-views/data-model/shine/db/customer_detail_view.sql

```
JOIN customers_billing_addresses  ON
  customers.id = customers_billing_addresses.customer_id
```

Next, we'll join CUSTOMERS_BILLING_ADDRESSES to ADDRESSES. Note that this is where we alias ADDRESSES to billing_address.

materialized-views/data-model/shine/db/customer_detail_view.sql

```
JOIN addresses billing_address    ON
  billing_address.id = customers_billing_addresses.address_id
```

Then, we'll need to join ADDRESSES to STATES so we can get the state code.

materialized-views/data-model/shine/db/customer_detail_view.sql

```
JOIN states billing_state         ON
  billing_address.state_id = billing_state.id
```

Note again that we've had to alias STATES to billing_state.

Finally, we'll repeat this structure for CUSTOMERS_SHIPPING_ADDRESSES, with the addition of restricting by primary in our join.

materialized-views/data-model/shine/db/customer_detail_view.sql

```
JOIN customers_shipping_addresses ON
```

2. https://en.wikipedia.org/wiki/Alias_(SQL)

```
  customers.id = customers_shipping_addresses.customer_id
➤  AND customers_shipping_addresses.primary = true
JOIN addresses shipping_address   ON
  shipping_address.id = customers_shipping_addresses.address_id
JOIN states shipping_state        ON
  shipping_address.state_id = shipping_state.id
```

With our query constructed, let's see how it fairs.

Query Performance

We can use EXPLAIN ANALYZE on our query to see what sort of performance we might expect.

```
> EXPLAIN ANALYZE SELECT
  customers.id             AS customer_id,
  customers.first_name     AS first_name,
  customers.last_name      AS last_name,
  customers.email          AS email,me,
  customers.username       AS username,,
  customers.created_at     AS joined_at,dress_id,
  billing_address.id       AS billing_address_id,
  billing_address.street   AS billing_street,
  billing_address.city     AS billing_city,,
  billing_state.code       AS billing_state,e,
  billing_address.zipcode  AS billing_zipcode,_id,
  shipping_address.id      AS shipping_address_id,
  shipping_address.street  AS shipping_street,
  shipping_address.city    AS shipping_city,,
  shipping_state.code      AS shipping_state,e
  shipping_address.zipcode AS shipping_zipcode
FROM
  customers

«remainder of the query»

WHERE
  customers.id = 2000
;
QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=1.12..169.81 rows=1 width=157)
              (actual time=0.028..0.028 rows=0 loops=1)
   ->  Nested Loop  (cost=0.98..169.64 rows=1 width=158)
                    (actual time=0.027..0.027 rows=0 loops=1)

«Tons of query plan details omitted»

 Planning time: 20.192 ms
 Execution time: 5.724 ms
```

The planning time is a bit longer, but if we assume this is reasonable representative, at least as compared to the individual queries, it should take less around 26 milliseconds. This is still pretty fast, but it's twice as slow as all seven queries combined. But, this query only incurs one network round-trip. If we assume the network round-trip is 1ms, that means this query will take 27ms, and our seven queries will take 20ms.

What *this* means is that our complex-but-slower query might be reasonable to consider using, *especially* if we can speed that query up (which we'll do using a materialized view).

A Word on Optimizations

You should absolutely avoid optimizing your system like this until you know that what you are optimizing is actually a problem, based on your observations. The use of EXPLAIN ANALYZE us useful in explaining poor performance, not in identifying it.

This chapter is about teaching you a technique to deal with poor performance, and is not something you should use by default every time you need to query more than one table. Always measure your system's performance before optimizing it.

Using this Query in Rails

It will be difficult, if not impossible, to use ActiveRecord's API to produce this query. In these cases, it's easier to just use a string of SQL and execute the query. We can do that by using the method execute on the underlying connection object available via the connection method on ActiveRecord::Base.

```
class CustomerDetail
  QUERY = %{

    «The big query from before»

  }

  def self.find(customer_id)
    ActiveRecord::Base.connection.execute(
      QUERY + " WHERE customers.id = #{customer_id}"
    ).first
  end
end
```

This code is not ideal, and we'd like to avoid having code like this in our application. First, it contains a SQL injection vulnerability[3], since we are constructing SQL without escaping the value of customer_id. We can work around this using Postgres' API directly, which would make the code even more difficult to understand. Secondly, this code doesn't return a nice object, but instead returns a hash that we must reach into in order to access the data.

It's not the worst code in the world, but it's not idiomatic Rails. This code, along with any code that calls it, will stick out like a sore thumb, confusing everyone that looks at it.

It may seem like a minor thing, but this sort of unnecessary complexity can make a codebase hard to read, understand, and manage. Sometimes, we have to live with code like this, but it's always worth trying to find a better way. Even if the ActiveRecord version of the code executed more queries, and incured a higher penalty for network round-trips, it looked like idiomatic Rails code.

We've now seen that the default *Rails Way* of modeling our access to this data will result in seven queries each time, and that those queries perform reasonably well, but incur a penalty in network round-trips. We've also seen that a large single query using joins *might* perform better, but results in ugly, hard-to-maintain code.

Now, we're going to see the third alternative—materialized views.

Using Materialized Views for Better Performance

Materialized views are a special form of *database view* that performs much better. If you aren't familiar with views, they are a table-like construct that abstracts away a complex query. For example, we could create a view named ADDRESSES_WITH_STATES that abstracts away the need to join ADDRESSES and STATES together, like so:

```
CREATE VIEW addresses_with_states AS
  SELECT
    addresses.id,
    addresses.street,
    addresses.city,
    states.code AS state,
    addresses.zipcode
  FROM
    addresses
```

3. https://en.wikipedia.org/wiki/SQL_injection

```
  JOIN states ON states.id = addresses.state_id
```

Now, we can treat ADDRESSES_WITH_STATES just like a normal table for querying:

```
> select * from addresses_with_states where id = 12;
-[ RECORD 1 ]----------------
id      | 12
street  | 11828 Kuhn Turnpike
city    | Willmsmouth
state   | WA
zipcode | 46419-7547
```

We can *also* treat this like a regular table for mapping with ActiveRecord.

```
class AddressesWithState < ActiveRecord::Base
end

AddressesWithState.find(12).state
# => DC
```

But, a view is just a place to store the query. It would make our Rails code better-looking, but would still not necessarily out-perform the seven simpler queries, since we'd still be running the complex join underneath.

With most relational databases, we would not be able to use our database to solve this problem. We'd need to set up some sort of caching solution, like memcached[4] or Elasticsearch[5]. We'd run our expensive query offline and populate our cache with the data, then query that data at runtime.

Postgres provides this exact feature via *materialized views*. A materialized view is basically an actual table with the actual data from the underlying query. In effect, Postgres does what these alternative caching solutions do—stores the results of the query in another table that can be quickly searched.

The advantage is that, just like a normal view, we can access the materialized view as if it were a regular table using ActiveReecord, meaning our Rails code will still be idiomatic. But, since the materialized view isn't doing the expensive query each time, we can fetch the data very quickly.

To create a materialized view, we simply do CREATE MATERIALIZED VIEW instead of CREATE VIEW. Let's do that now by creating a new migration.

```
> bundle exec rails g migration create-customer-details-materialized-view
  invoke active_record
  create  db/migrate/20150625130204_create_customer_details_materialized_view.rb
```

4. http://memcached.org/
5. https://www.elastic.co/

Now, we'll use execute in our migration to create the materialized view using the large and complex query we were using before. Since a materialized view creates a table under the covers, we're also going to create an index on customer_id, since that's the field we'll be using the query the materialized view (it will also allow us to keep the view up-to-date, as we'll see).

materialized-views/actual-materialized-view/shine/db/migrate/20150625130204_create_customer_details_materialized_view.rb

```
class CreateCustomerDetailsMaterializedView < ActiveRecord::Migration
  def up
    execute %{
    CREATE MATERIALIZED VIEW customer_details AS
      SELECT
        customers.id             AS customer_id,
        customers.first_name     AS first_name,
        customers.last_name      AS last_name,
        customers.email          AS email,
        customers.username       AS username,
        customers.created_at     AS joined_at,
        billing_address.id       AS billing_address_id,
        billing_address.street   AS billing_street,
        billing_address.city     AS billing_city,
        billing_state.code       AS billing_state,
        billing_address.zipcode  AS billing_zipcode,
        shipping_address.id      AS shipping_address_id,
        shipping_address.street  AS shipping_street,
        shipping_address.city    AS shipping_city,
        shipping_state.code      AS shipping_state,
        shipping_address.zipcode AS shipping_zipcode
      FROM
        customers
      JOIN customers_billing_addresses  ON
        customers.id = customers_billing_addresses.customer_id
      JOIN addresses billing_address    ON
        billing_address.id = customers_billing_addresses.address_id
      JOIN states billing_state         ON
        billing_address.state_id = billing_state.id
      JOIN customers_shipping_addresses ON
        customers.id = customers_shipping_addresses.customer_id
        AND customers_shipping_addresses.primary = true
      JOIN addresses shipping_address   ON
        shipping_address.id = customers_shipping_addresses.address_id
      JOIN states shipping_state        ON
        shipping_address.state_id = shipping_state.id
    }
    execute %{
      CREATE UNIQUE INDEX
        customer_details_customer_id
      ON
        customer_details(customer_id)
```

```
    }
  end
  def down
    execute "DROP VIEW customer_details"
  end
end
```

Next, we run the migration with rake db:migrate. It will take a while, as it's basically running this query for every row of all of these tables. When it's done, we'll be able to query this data very quickly.

Let's do an EXPLAIN ANALYZE on our new materialized view.

```
> explain analyze
    select * from customer_details where customer_id = 2000;
QUERY PLAN
------------------------------------------------------------------------------
  Index Scan using customer_details_customer_id on customer_details
      (cost=0.42..8.44 rows=1 width=404)
      (actual time=0.125..0.125 rows=1 loops=1)
   Index Cond: (customer_id = 2000)
 Planning time: 1.750 ms
 Execution time: 0.196 ms
```

This is pretty darn good! We're pulling back all of the data we need in under 2 milliseconds. That is far faster that *both* the canonical Rails Way using ActiveRecord *and* our complex query.

We can now create a CustomerDetail class and query it just as we would any other ActiveRecord object, keeping our code clean and idiomatic, but it will blazingly fast.

materialized-views/actual-materialized-view/shine/app/models/customer_detail.rb

```
class CustomerDetail < ActiveRecord::Base
  self.primary_key = 'customer_id'
end
```

Note that since our materialized view doesn't have a field named ID, we need to use primary_key= to tell ActiveRecord what the primary key field is. In this case, it's CUSTOMER_ID. With this in place, our controller looks like a regular Rails controller.

materialized-views/simple-view/shine/app/controllers/customers_controller.rb

```
def show
  customer_detail = CustomerDetail.find(params[:id])
  #END_HIGHTLIGHT
  respond_to do |format|
    format.json { render json: customer_detail }
  end
end
```

It seems we've addressed our performance problem by creating a very fast way to get our data, but without having to write complex code that looks out-of-place or is hard to understand and maintain. Let's see what happens when we insert a new customer into our database.

```
> insert into customers(
    first_name,last_name,email,username,created_at,updated_at)
  values (
    'Dave','Copeland','dave@dave.dave','davetron5000',now(),now());
INSERT 0 1
> select id from customers where username = 'davetron5000';
   id   |
--------+
 388399 |
(1 row)

> insert into customers_billing_addresses(
    customer_id,address_id)
  values (388399,1);
INSERT 0 1
> insert into customers_shipping_addresses(
    customer_id,address_id,"primary")
  values (388399,1,true);
INSERT 0 1
```

We might assume that this new customer will come back when we query our materialized view.

```
> select * from customer_details where customer_id = 388399;
(No rows)
```

Oops, it looks like something's wrong. This is due to the implementation of materialized views. Like any caching solution, the cache (in our case, the materialized view) must be updated when data changes. This is the trade-off of a cache and is why it's able to be fast.

In the next section, we'll see how to set up our database to keep the materialized view updated.

Keeping Materialized Views Updated

The reason our materialized view is so much faster than the regular view is because it essentially caches the results of the backing query into a real table. The trade-off is that the contents of the table could lag behind what's in the tables that the backing query queries.

Postgres provides a way to refresh the view via `REFRESH MATERIALIZED VIEW`. Before Postgres 9.4, refreshing materialized views like this was a problem, because

it would lock the view while it was being refreshed. That meant that any application that wanted to query the view would have to wait until the update was completed. Since this could potentially be a long time, it meant that materialized views were mostly useless before 9.4.

As of Postgres 9.4, the refresh can be done concurrently in the background, allowing users of the table to continue querying old data until the refresh is complete. This is what we'll set up here, and requires running the command REFRESH MATERIALIZED VIEW CONCURRENTLY.

Let's try it out.

```
> refresh materialized view concurrently customer_details;
> select * from customer_details where customer_id = 388399;
-[ RECORD 1 ]---------------+-----------------------------
customer_id                 | 388399
first_name                  | Dave
last_name                   | Copeland
email                       | dave@dave.dave
username                    | davetron5000
joined_at                   | 2015-06-25 08:28:54.327645
billing_street              | 530 Nienow Stravenue
billing_city                | West Aniyah
billing_state               | RI
billing_zipcode             | 72842-8201
shipping_address_id         | 1
shipping_street             | 530 Nienow Stravenue
shipping_city               | West Aniyah
shipping_state              | RI
shipping_zipcode            | 72842-8201
shipping_address_created_at | 2015-06-20 16:51:06.891914-04
```

Now that we know how to refresh our view, the trick is *when* to do it. This highly depends on how often the underlying data changes and how important it is for us to see the most recent data in the view. We'll look at two techniques for doing that here. The first is to create a Rake task to refresh the view on a schedule. The second is to use *database triggers* to refresh the view whenever underlying data changes.

Refreshing the View on a Schedule

The simplest way to refresh the view is to create a Rake task, and then arrange for that task to be run on a regular schedule. We can do this by creating lib/tasks/refresh_materialized_views.rake and use the connection method on ActiveRecord::Base, which will allow us to execute arbitrary SQL.

materialized-views/actual-materialized-view/shine/lib/tasks/refresh_materialized_views.rake

```
desc "Refreshes materialied views"
```

```
task refresh_materialized_views: :environment do
  ActiveRecord::Base.connection.execute %{
    REFRESH MATERIALIZED VIEW CONCURRENTLY customer_details
  }
end
```

We can then run it on the command line via rake:

```
> bundle exec rake refresh_materialized_views
```

With this in place, we can then configure our production system to run this periodically, for example using cron[6]. How frequently to run it depends on how recent the data should be to users, as well as how long it takes to do the refresh. If users need the data to be fairly up-to-date, we could try running it every five minutes. If users can do their jobs without the absolute latest, we could run it every hour or even every day.

If the users need it to be absolutely up-to-date with the underlying tables, we can have the database itself refresh, whenever the underlying data changes by using *triggers*.

Refreshing the View with Triggers

A *database trigger* is similar to an ActiveRecord callback: it's code that runs when certain events occur. In our case, we'd want to refresh our materialized view whenever data in the tables that view is based on changes.

To do this, we'll create a database function that refreshes the materialized view, and then create several triggers that use that function when the data in the relevant tables changes. We can do this all in a Rails migration, so let's create one where we can put this code.

```
> bundle exec rails g migration trigger-refresh-customer-details
    invoke  active_record
    create    db/migrate/20150626120132_trigger_refresh_customer_details.rb
```

First, we'll create a function to refresh the materialized view. This requires using Postgres' PL/pgSQL[7] language. It looks fairly archaic, but we don't need to use much of it.

materialized-views/actual-materialized-view/shine/db/migrate/20150626120132_trigger_refresh_cus¬tomer_details.rb

```
execute %{
  CREATE OR REPLACE FUNCTION
    refresh_customer_details()
```

6. https://en.wikipedia.org/wiki/Cron
7. http://www.postgresql.org/docs/9.4/static/plpgsql.html

```
    RETURNS TRIGGER LANGUAGE PLPGSQL
  AS $$
  BEGIN
    REFRESH MATERIALIZED VIEW CONCURRENTLY customer_details;
    RETURN NULL;
  END $$;
}
```

The key part of this is RETURNS TRIGGER, which is what will allow us to use this function in the triggers we'll set up next.

The form of a trigger we want will look like so:

```
CREATE TRIGGER
  refresh_customer_details
AFTER
  INSERT OR
  UPDATE OR
  DELETE
ON
  customers
FOR EACH STATEMENT
  EXECUTE PROCEDURE refresh_customer_details();
```

The code for this trigger reads like what it does: any insert, update, or delete on the customers table will cause the database to run the function we defined, refresh_customer_details. So, we just need to set this up for each table that's relevant.

If we assume that the list of US states doesn't change, we can set up triggers for the other three tables: CUSTOMERS_SHIPPING_ADDRESSES, CUSTOMERS_BILLING_ADDRESSES, and ADDRESSES. Since the code is almost the same for each table, we'll loop over the table names and construct the SQL dynamically.

materialized-views/actual-materialized-view/shine/db/migrate/20150626120132_trigger_refresh_customer_details.rb

```
%w(customers
   customers_shipping_addresses
   customers_billing_addresses
   addresses).each do |table|
  execute %{
    CREATE TRIGGER refresh_customer_details
    AFTER
      INSERT OR
      UPDATE OR
      DELETE
    ON #{table}
      FOR EACH STATEMENT
        EXECUTE PROCEDURE
          refresh_customer_details()
```

```
  }
end
```

After we run rake db:migrate, we can insert new customers and see the view get refreshed automatically.

```
> insert into customers(
    first_name,last_name,email,username,created_at,updated_at)
  values (
    'Amy','Copeland','amy@amy.dave','amytron',now(),now());
INSERT 0 1
> insert into customers_shipping_addresses
    (customer_id,address_id,"primary")
  values
    (388400,1,true);
INSERT 0 1
> insert into customers_billing_addresses
    (customer_id,address_id)
  values
    (388400,1);
INSERT 0 1
> select * from customer_details where customer_id = 388400;
-[ RECORD 1 ]-------+--------------------------
customer_id         | 2
first_name          | Amy
last_name           | Copeland
email               | amy@amy.dave
username            | amytron
joined_at           | 2015-06-26 08:17:17.536305
billing_address_id  | 1
billing_street      | 123 any st
billing_city        | washington
billing_state       | DC
billing_zipcode     | 20001
shipping_address_id | 1
shipping_street     | 123 any st
shipping_city       | washington
shipping_state      | DC
shipping_zipcode    | 20001
```

Using triggers, you don't have to worry about setting up an external process to keep the materialized view refreshed. The downside is if your table will have a high volume of writes or updates, you'll be refreshing the view a lot, and this could slow down your database. You'll have to evaluate which technique will be best, based on your actual usage.

The path that lead us to materialized views was the promise of high performance and code simplicity. It may have felt circuitous, but it's a great demonstration of the type of power you have as a full-stack developer. By

understanding the breadth of tools available to you, and how to use them, you can create solutions that are simple.

Although we had to create a materialized view, and triggers to keep it updated, the system we've created is much simpler than what we'd have produced using a traditional caching system. Our Rails code looks like regular Rails code—we're using ActiveRecord to query data. We didn't have to set up any new infrastructure—everything is in Postgres. We were even able to use features built-in to Postgres—triggers—to keep our cached data updated.

Next Up: Combining the Data with a Second Source in Angular

We're about to complete our journey in creating a high-performing, usable and clean way for our users to see a customer's details. We have our UI designed and built, and our back-end is now clean, simple, and fast. We now need to bring them together in our Angular app.

With what we know about Angular, using this data in our view is pretty easy. We've seen how we can request information from the server user `$http` and we've seen how to show that data in our view using Angular's templating system, along with the `ng-model` directive.

One thing you might have noticed is that our `CustomerDetail` model doesn't expose some of the billing information, such as the last four digits and expiration data of the customer's credit card. We don't store this data in our database, but we need it in our view. In the next chapter we'll see how Angular manages that, by learning more about how its asynchronous nature works.

We'll show how we can pull data from two separate sources and have it display in the view, showing the data as it comes in, for the best user experience, all without a whole lot of code.

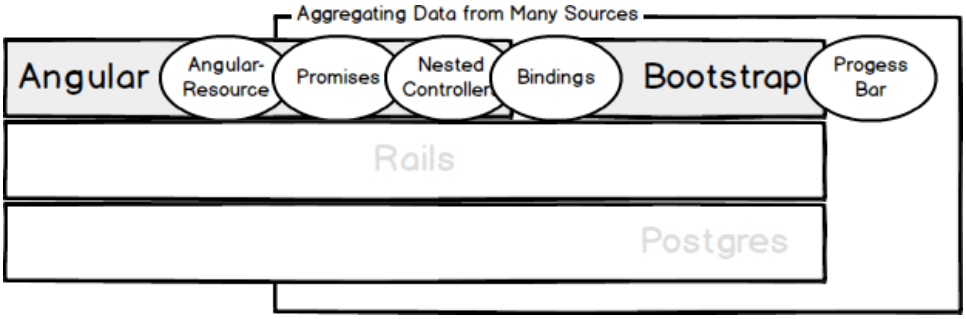
CHAPTER 11

Load Data from Many Sources Asynchronously

In the last chapter, we saw how using a materialized view made it easy to aggregate data from several different database tables. But what if data we need comes from more than one source? Do we collect data from all sources at the server, thus making the user wait for all of it to be available before seeing any of it? Or, is there a way to show the user data as it comes in, updating the UI along the way?

The latter is a better user experience, and that's what we're going to learn here. We'll use Angular's asynchronous design to show the user data as it comes in, rather than making them wait. We'll use Angular-Resource to keep our code clean and clear, and learn how to keep our code de-coupled by using multiple controllers for the view. We'll also use Bootstrap to make sure the user experience is acceptable while the user is waiting for data.

As you recall, our running example shows a detailed view of a customer's information. It includes the data we're pulling back from our materialized view, but also data about their credit card, which is located inside a third-party service (which we'll simulate). With what we'll learn in this chapter, we'll solve this problem in a clean way, maintaining the best user experience we can.



We're going to take this in small steps. First, we'll learn about *angular-resource*, which makes it easier to manage asynchronous code, and enables the user experience we want. We'll use this to fetch the customer details we made available in Chapter 10, *Cache Complex Queries Using Materialized Views*, on page 173. Next, we'll learn how to nest controllers in our view, so we can make a separate controller for fetching the credit card details.

With the structure of our user interface in place, we'll see how to use Bootstrap's progress component to show the user a loading animation that's connected to the AJAX call. With the user *experience* complete, we'll see how the two controllers can communicate so they can stay decoupled from one another. Finally, we'll revisit how to unit test our Angular controllers in light of this new information.

First, though, let's get an overview of how asynchronous code works in Angular. Since asynchronous code isn't something that is a part of Rails, you might not have a lot of experience with it conceptually, and this can make it hard to understand the code we're going to write. Let's step through the lifecycle of some asynchronous requests.

Understanding How Asynchronous Requests Work

If you've done any JavaScript programming, you are familiar with *callbacks*. These are functions that get called *later*. The simplest example is `setTimeout`, which takes a callback function and a number of milliseconds. After the given milliseconds have elapsed, the function is called.

In this code, we execute the function `errorMessage` one second later:

```
var errorMessage = function() {
  alert("OH NOES!");
};
setTimeout(errorMessage,1000);
```

We also saw this when using $http in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81, where we passed a callback to get that would execute our code once we got a response from the server. We didn't really talk about *why* $http works that way. Now we will.

Why Asynchronous?

Contacting the server takes time. Even with all the improvements we've made in the performance of our Rails controller, fetching data over the Internet is not instantaneous. If our JavaScript were to make an AJAX call and simply wait for a response (called a *synchronous request*), the entire browser would be hung while it waited on the network.

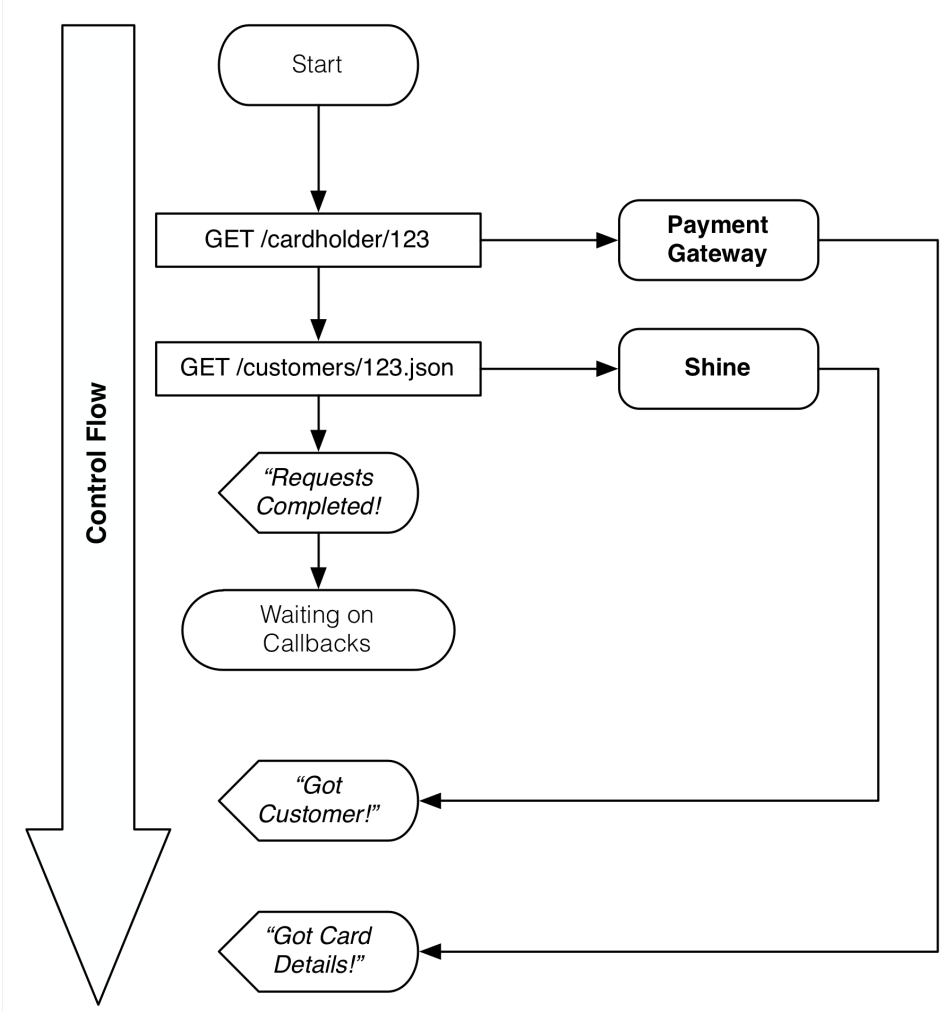
In order to prevent this, we want to make the call in the background and let our main bits of code (and the browser in general) continue to operate while the networking request is happening. In many programming languages, this is done with threads. JavaScript does not expose the concept of threads to the user, instead requiring programmers to use callbacks managed internally by the runtime.

This means our code is often organized in three chunks: setup to make our AJAX request, making the request, and code to run after the request has completed. Where this can get tricky is when we have more than one request. Consider this code:

```
Line 1 var base = "http://billing.example.com"
  2 $http.get(base + "/cardholder/123").success(function(data) {
  3   alert("Got card details!");
  4 });
  5 $http.get("/customer/123.json").success(function(data) {
  6   alert("Got customer!");
  7 });
  8 alert("Requests complete!");
```

Although this code looks like it runs top to bottom, it actually doesn't. Lines 1 and 2 execute first. While that AJAX call is happening, the code proceeds to line 5. After that, line 8 is called.

This means that lines 3 and 6 haven't executed yet! They'll execute when their respective AJAX calls complete, and that could happen in any order. This means that a big part of our code—handling the responses from the server—cannot rely on any particular ordering.



This can get confusing fast, especially if the code *does* need to run in a certain order. Consider if we needed a credit card holder ID from our customer details in order to fetch the customer's card information from our third-party billing service. We'd need to make our call to the billing service inside the callback that's called when our customer details are fetched, creatinga nested structure like so:

```
var base = "http://billing.example.com"
$http.get("/customer/123.json").success(function(data) {
  alert("Got customer!");
  var url = base = "/cardholder/" + data.cardholder_id
  $http.get(url).success(function(data) {
    alert("Got card details!");
```

```
  });
});
alert("Requests complete!");
```

This is called *callback hell* and can make your client-side code hard to deal with. Part of this is just the reality of programming with an asynchronous model. But, there are tools to help wrangle this complexity, called *promises*.

Promises

You'll notice that we aren't passing a function to $http.get like we did to setTimeout. Instead, we're calling the function success on whatever $http.get returns. What it's returning a called a *promise*.

A promise is an object that represents a background call that may or may not have completed. At its most basic, a promise contains a single function that accepts a function to be called when the underlying call has completed. When a promise is completed, it's said to be *resolved* and, when that happens, our function is called with the results.

All of Angular's asynchronous code is build on its $q library[1], which uses the function then:

```
var promise = $http.get("/customers/123.json");
promise.then(function(response) {
  // This is called when the AJAX call completed
  // regardless of the outcome.
});
```

Promises are a deep topic that we won't get too far into, but it's important to understand what they are. In the next section, we'll see how promises allow us to clean up our code quite a bit. The main thing you should take away is that there are objects that represent background work that can be accessed while the work is happening.

The code we've written thus-far (using $http) uses the promises returned by $http, but still is written in with a callback style. Now that we know about promises, we can learn about *angular-resource*, which makes more intelligent use of the underlying promise, providing an API that does *not* require callbacks.

1. https://docs.angularjs.org/api/ng/service/$q

Using Angular-Resource to Connect to Rails

The Rails way of designing HTTP endpoints (which is a *RESTful* style[2]) isn't particular to Rails. It's quite common outside the Rails world. Because of this, it's possible to use a higher-level of abstraction than what $http gives us. The Angular module *angular-resource* provides such an abstraction.

What it will allow us to do is to write code like so:

```
$scope.customer = Customer.get(customer_id);
```

Although there are no callbacks, this code is still asynchronous. Execution will proceed directly after the call and not wait for the AJAX call to complete. This is because the object returned by Customer.get is a promise that, when resolved, will behave like a customer object, exposing the various fields from the call.

What *this* allow us to do is to use customer in our view as if it were a populated object. When the promise is resolved, the view will automatically update. Without callbacks, our code is clean and easy to follow yet it won't on the remote HTTP call.

To be able to write our controller in this style, we'll need to install and configure Angular-Resource. It provides the $resource service, which we can use to create an object like Customer. Customer can then be used to make our AJAX calls as shown above.

Installing Angular-Resource

Much like how we installed angular-route in Chapter 8, *Create a Single-Page App Using Angular's Router*, on page 139, we'll do the same for Angular-Resource. First, we'll add it to Bowerfile and run rake bower:install to download it.

angular-async/start/shine/Bowerfile

```
asset 'bootstrap-sass-official'
asset 'angular', '~> 1.4'
asset 'angular-mocks'
asset 'angular-route'
➤ asset 'angular-resource'
```

Next, we'll add it to app/assets/javascripts/application.js.

angular-async/start/shine/app/assets/javascripts/application.js

```
//= require angular
//= require angular-route
➤ //= require angular-resource
```

2. https://en.wikipedia.org/wiki/Representational_state_transfer

```
//= require angular-rails-templates
//= require_tree ./templates
//= require_tree .
```

Finally, we'll add Angular-Resource as a dependent module when setting up our app. As before, we can't use the string angular-resource, but have to translate that to the module name ngResource.

angular-async/start/shine/app/assets/javascripts/customers_app.js

```
var app = angular.module(
  'customers',
  [
    'ngRoute',
    'ngResource',
    'templates'
  ]
);
```

All this set up will allow us access to the $resource service[3].

Using $resource

The $resource service is a *factory*[4] function that takes a URL fragment as a parameter and returns an object we can use to make AJAX calls to our Rails controller, without having to use callbacks.

First, we'll inject it into our CustomerDetailController by adding its name to the parameter list and adding the argument $resource to our function definition (see *Creating Our First Angular Controller*, on page 87 for a reminder on why we have to do this).

angular-async/start/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerDetailController", [
          "$scope","$routeParams","$resource",
  function($scope , $routeParams , $resource) {
]);
```

Now that it's available to our controller, we can replace the innards of our controller entirely with the following:

angular-async/start/shine/app/assets/javascripts/customers_app.js

```
var customerId = $routeParams.id;
var Customer = $resource('/customers/:customerId.json')

$scope.customer = Customer.get({ "customerId": customerId})
```

3. https://docs.angularjs.org/api/ngResource/service/$resource
4. https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)

This code is a lot cleaner and a lot clearer. It *looks* like a simple, synchronous call but it's still asynchronous. This is because the object we get back from Customer.get is a promise that, when resolved, will set properties on itself based on the results of the call.

We can see this in action, but first we need to use $scope.customer in our view.

Populating the View

Rather than revisit the entire view we designed in Chapter 9, *Design Great UIs With Bootstrap's Grid and Components*, on page 155, we'll just show a little bit of the changes we need to make. This shouldn't be anything new if you've been following along.

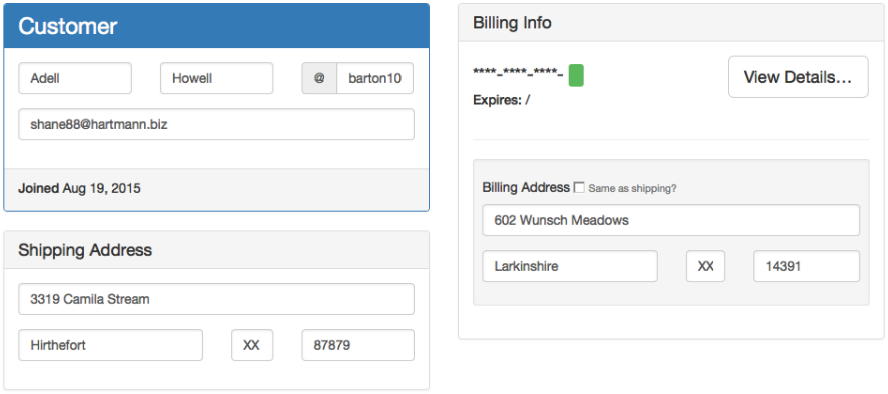
We'll use the ng-model directive to bind our form elements to the values we get back from the AJAX call.

angular-async/start/shine/app/assets/javascripts/templates/customer_detail.html

```
<article class="panel panel-primary">
  <header class="panel-heading">
    <h1 class="h3">
      Customer
    </h1>
  </header>
  <section class="panel-body">
      <div class="row">
        <div class="col-md-4">
          <div class="form-group ">
            <label class="sr-only" for="first-name">
              First Name
            </label>
➤           <input type="text" class="form-control"
➤                  name="first-name" ng-model="customer.first_name">
          </div>
        </div>
        <div class="col-md-4">
          <div class="form-group ">
            <label class="sr-only" for="last-name">Last Name</label>
➤           <input type="text" class="form-control"
➤                  name="last-name" ng-model="customer.last_name">
          </div>
        </div>
        <div class="col-md-4">
          <div class="form-group ">
            <label class="sr-only" for="username">Username</label>
➤           <div class="input-group">
➤             <div class="input-group-addon">@</div>
➤             <input type="text" class="form-control"
➤                    name="username" ng-model="customer.username">
```

```
            </div>
          </div>
        </div>
      </div>
      <div class="form-group">
        <label class="sr-only" for="email">Email</label>
        <input type="text" class="form-control"
               name="email" ng-model="customer.email">
      </div>
    </section>
    <footer class="panel-footer">
      <label for="joined">Joined</label>
      {{ customer.joined_at | date }}
    </footer>
  </article>
```

The rest of the view will be similar. We should now be able to navigate to our customer search, find a customer, click to view details and see the data populated into our form (save for the credit card info, which we'll get to in a moment).



We can see the asynchronous promise in action by adding a call to `sleep` to our controller method.

```
def show
  customer_detail = CustomerDetail.find(params[:id])
  sleep 5
  respond_to do |format|
    format.json { render json: customer_detail }
  end
end
```

This `sleep` call will make our controller artificially slow, which will make the AJAX call take a long time. We can also add a call to `alert` in our Angular

controller that will execute right after the AJAX call is made, but before it's completed:

```
var customerId = $routeParams.id;
var Customer = $resource('/customers/:customerId.json')

$scope.customer = Customer.get({ "customerId": customerId})
alert("AJAX Call Initiated!");
```

If you reload the detail page now, you should see the alert pop up. If you dismiss it quickly, the screen will show a blank form for a few seconds, and when the sleep in our controller finishes, the data will populate in the view automatically.

Angular knows to do this via *binding*. When we use ng-model or simply refer to values from the scope in our view via the {{value}} syntax, Angular *binds* those values to the view. It then begins to watch them for changes. When the AJAX call completes and sets values for properties like customer.first_name and customer.joined_at, Angular notices and updates any parts of the DOM affected by the new values.

We'll learn more about the bindings in Chapter 12, *Wrangle Forms and Validations with Angular*, on page 217, when we save the user's changes back to the server.

Now that we've simplified access to the back-end, and hooked everything up to our new detail view, we can start to think about how to get the credit card info into this view. Since that data will come from a different source than the data we've been using so far, it's worth considering if we can keep the code separate.

We'll see how we can do this by placing our new code in a different controller, and having that controller nested in our view, controlling only the part of it that needs its data.

Nesting Controllers to Organize Code

So far, we've had a setup of one controller per view. We could continue doing that here, and add the necessary code to CustomerDetailController to get the user's credit card info from the second source. Although for the task at hand, it might not be too complex to do so, we should learn how to do this a better way, especially if things get complex later. It's not much more code, and will make our Angular app easier to work.

We can achieve this separation by using a second, nested controller, attached to a subset of our view using ng-controller. We didn't need to use ng-controller after

we introduced Angular-Route, because the controller to use is configured in the routing configuration. But, we can still use ng-controller in our view to effectively override this configuration.

Implementing the controller itself will be straightforward: we just need know the URL from which to fetch the credit card info. Since this is just a simulation, we'll stand up an endpoint in our Rails app to act as the payment processor's website.

First, we'll add a route called *fake_billing* to config/routes.rb:

angular-async/start/shine/config/routes.rb

```
Rails.application.routes.draw do
  devise_for :users
  root 'dashboard#index'
  resources :customers, only: [ :index, :show ]

  get "angular_test", to: "angular_test#index"
➤ get "fake_billing", to: "fake_billing#show"
end
```

FakeBillingController will then use Faker to return canned data based on a *carholder ID*.

angular-async/start/shine/app/controllers/fake_billing_controller.rb

```
class FakeBillingController < ApplicationController
  skip_before_action :authenticate_user!
  def show
    if params[:cardholder_id]
      sleep 3
      render json: {
               lastFour: Faker::Business.credit_card_number[-4..-1],
               cardType: Faker::Business.credit_card_type,
        expirationMonth: Faker::Business.credit_card_expiry_date.month,
         expirationYear: Faker::Business.credit_card_expiry_date.year,
            detailsLink: Faker::Internet.url,
      }
    else
      head :not_found
    end
  end
end
```

We've used sleep to simulate a slow response from the payment processor. This will allow us to see the effects of how we've designed our user interface. We should see the customer details pop up quickly, followed later by the credit card info. This will be more what it would like in the real world, and also allow us to verify that our asynchronous handling of the various AJAX calls is happening as we expect.

With our simulated payment processor setup, we can write our nested Angular controller, which we'll call CustomerCreditCardController. It will work similarly to CustomerDetailController in that we'll use $resource to create the object CreditCardInfo, which will access the credit card endpoint we created above, then use it to fetch the info.

Despite the fact that we are simulating the third-payment processor with Rails, $resource isn't restricted to just Rails endpoints. It works with any HTTP endpoint that is RESTful. Were we integrating with a real payment processor, and their API was similar to how Rails works, our code would work just as it does here (of course $resource is highly flexible and can be configured to deal with just about anything).

Here's what CustomerCreditCardController will look like (we'll use a hard-coded cardholder ID for now and see how to get the real value in a later section of this chapter):

angular-async/start/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerCreditCardController", [
          "$scope","$resource",
  function($scope , $resource) {
    var CreditCardInfo = $resource('/fake_billing.json')
    $scope.creditCard = CreditCardInfo.get({ "cardholder_id": 1234})
  }
]);
```

This is enough for us to get things working in the UI. To do that, we'll use ng-controller on the element whose children should be managed by CustomerCreditCardController. Doing this will make anything placed into CustomerCreditCardController's $scope available to that markup, but *not* available outside of it, thus achieving the separation we were going for.

Here's our template, including how we are using the exposed creditCard object.

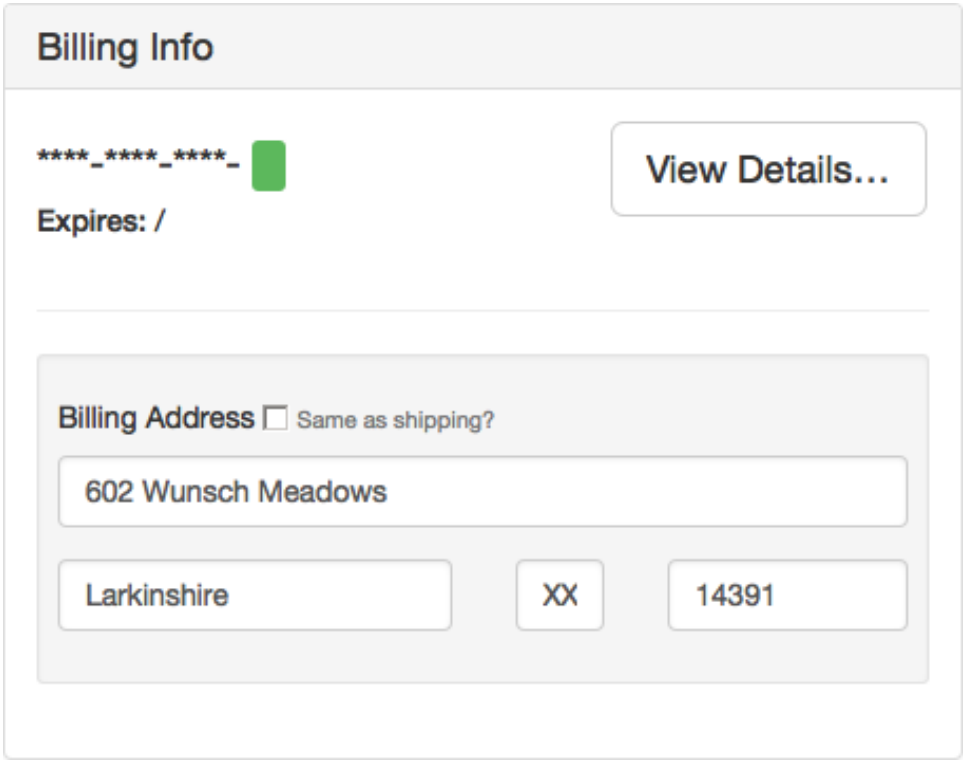
angular-async/start/shine/app/assets/javascripts/templates/customer_detail.html

```
  <div class="col-md-6">
    <article class="panel panel-default billing-info">
      <header class="panel-heading">
        <h2 class="h4">
          Billing Info
        </h2>
      </header>
      <section class="panel-body">
        <article ng-controller="CustomerCreditCardController">
          <div class="row">
            <div class="col-md-7">
              <p class="h4">
                ****-****-****-{{creditCard.lastFour}}
```

```
                <span class="label label-success">
                  {{creditCard.cardType}}
                </span>
              </p>
              <p class="h5">
            <label>Expires:</label>
            {{creditCard.expirationMonth}}/{{creditCard.expirationYear}}
              </p>
            </div>
            <div class="col-md-5 text-right">
              <a href="{{creditCard.detailsLink}}"
                 class="btn btn-lg btn-default">
                View Details…
              </a>
            </div>
          </div>
        </article>

        <!-- rest of the billing address markup... -->

        </article>
      </section>
    </article>
  </div>
</div></form>
```

With our UI connected to our nested controller, we can reload the page and see it all in action. Everything will look the same, but you should see that the customer details load quickly and, after a pause, the credit card info will appear. This demonstrates that we have the building-blocks to get the user the best experience we can *and* keep our code organized in doing so.

You'll note I said "building blocks". There are two problems with the current implementation. First is that while the request is happening, the app just sits there, providing no indication of what it's doing. The second is that while the user is waiting for the data, they see the credit card info in a weird empty state. For example, why should the "View Details" button even be displayed if we don't have the link available?



While we still need to get the actual value for `cardholder_id` into our controller, let's fix the UI issue first. We can do this by combining a property Angular exposes about the state of a promise's resolution, along with Bootstrap's *progress* component.

Using Bootstrap's Progress Bar When Data is Loading

In a traditional application, where the views are generated on the server, you get a reasonable user experience during slow-loading pages for free. Most browsers have some sort of loading animation or messaging while the page is being rendered. It's not great, but it's something.

When making heavy use of a front-end framework like Angular, we no longer get that bit of UI. As we are seeing with the customer detail page, there's no indication that anything is happening. It may appear to users that the app is broken or hung. We'd like to fix that.

There are two things we'll need to do. First, we need to know how to detect if the AJAX call is still in progress or if it's completed. We'll use that for the second step, which is to render a loading animation if the call is still in progress.

Dynamically Rendering Content Based on the AJAX Call's Status

We've seen how we can show or hide parts of the DOM before by using the ng-if directive. What we need to know is what expression to use to show or hide our UI while the AJAX call is happening. As you're recall, $resource creates a factory (in our case, CreditCardInfo) that returns a promise, meaning that $scope.creditCard is a promise that gets resolved when the AJAX call completes.

In the documentation for $resource you'll see that all such objects include the property $resolved, which will be false while the AJAX call is happening and true when it's done.

To use it, we'll create two divs. The first will be shown when $resolved is false, and will contain our loading animation. The second will contain the existing credit card info UI.

angular-async/progress/shine/app/assets/javascripts/templates/customer_detail.html

```
<article ng-controller="CustomerCreditCardController">
  <div class="row">
    <div ng-if="!creditCard.$resolved">

      <!-- loading animation will go here -->

    </div>
    <div ng-if="creditCard.$resolved">

      <!-- Credit Card info component -->

</article>
```

Because Angular is watching $resolved, when it changes from false to true, the loading animation will get hidden and the UI, populated with all the data we got back from the server, will be shown.

We just need to create a loading animation.

Using an Animated Progress Bar as a Loading Animation

There are countless loading animations across the web, some in pure CSS and some requiring JavaScript. To keep with our theme of not writing any CSS, we're going to use Bootstrap's progress bar component[5]. It has many features, including animation.

Although the progress of our AJAX call is indeterminate, Bootstrap's progress bar can function as an indeterminate progress indicator or loading animation. We'll simply create a progress bar that's at 100%, and animate it, overlaying

5. http://getbootstrap.com/components/#progress

the label “Loading”. With custom CSS, we could probably do better, but this will be pretty good, given the tools at hand.

We’d like to give the progress bar some space, so it doesn’t crowd the panel. We can do this using the grid we learned about in Chapter 9, *Design Great UIs With Bootstrap's Grid and Components*, on page 155. We’ll have the progress bar take up ten of the 12 available grid cells, and use a helper class, `col-md-offset-1`, which will offset the cell containing the progress bar by 1 cell, effectively centering it in the panel, with a padding of one cell on each side.

Inside our cell, we’ll put the progress bar’s code, which looks like so:

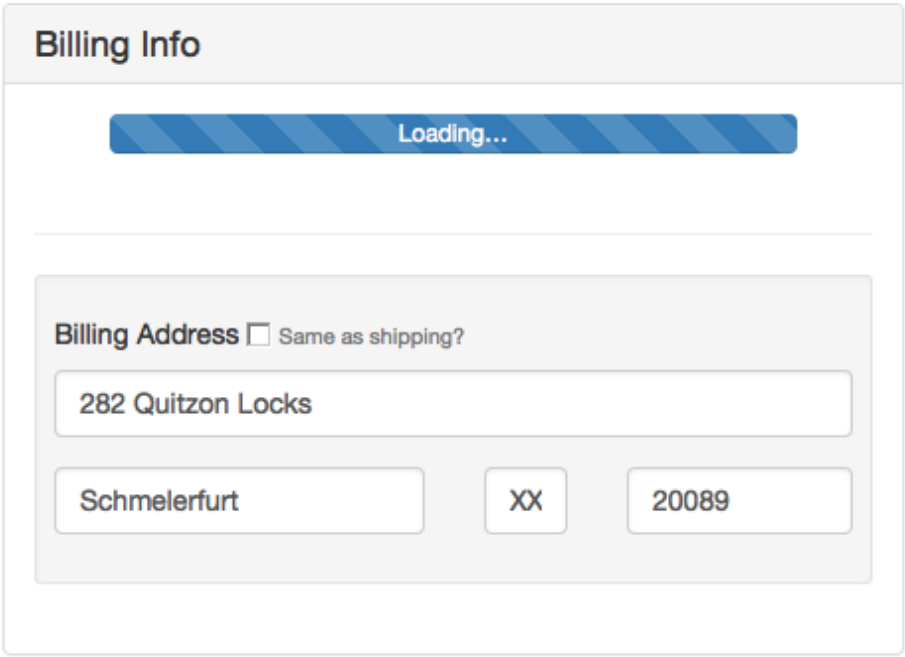
angular-async/progress/shine/app/assets/javascripts/templates/customer_detail.html

```
<div class="col-md-10 col-md-offset-1">
  <aside class="progress">
    <div class="progress-bar progress-bar-striped active"
         style="width: 100%">
      Loading…
    </div>
  </aside>
</div>
```

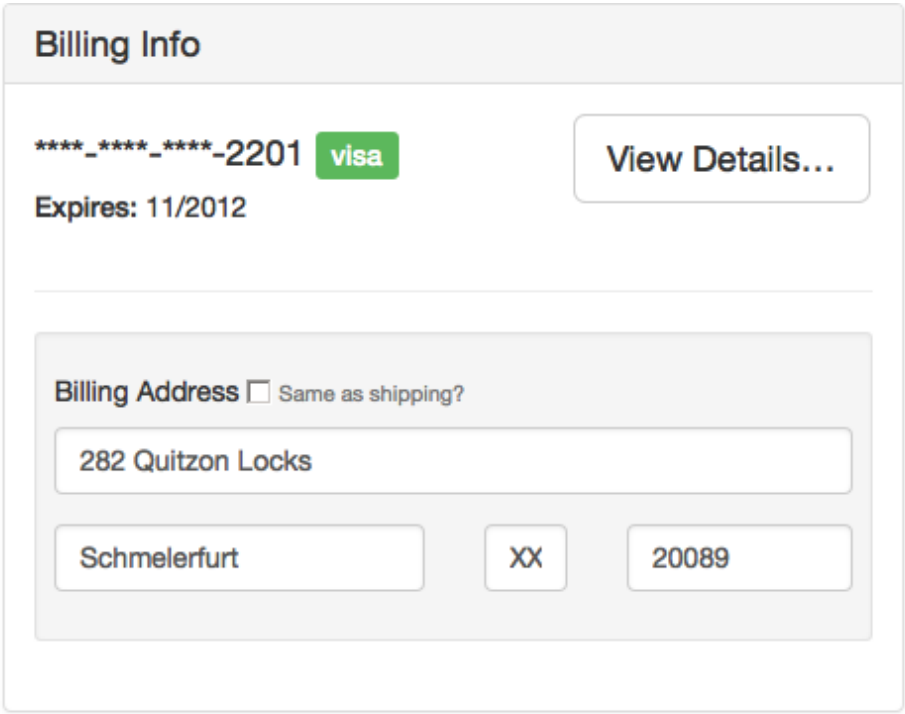
You’ll notice that we are somewhat cheating on our “no CSS” rule, because there is a `style` element on the inner `dive` of the progress bar. This is the way Bootstrap knows how much progress to render. It’s not very sophisticated, but it was designed before widespread use of the HTML5 `progress` element (which we can’t use, because it’s default styling conflicts with Bootstrap’s, and will eliminate the animation we need for this feature).

Nevertheless, it looks pretty good and solves our immediate problem. This is a great example of bending our tools to meet our needs without introducing complexity.

When we load the detail page now, you should see an animated progress bar:



After a few seconds, the AJAX call completes and you should see the credit card info UI, with all the data from the back-end:



It's pretty amazing what we've been able to accomplish for ourselves and our users with very few lines of code. We're fetching data from the backend, asynchronously, without callback hell, and we've been able to give the user a decent experience while a slow-loading resource becomes available. We've shown the user the customer details almost immediately, and used a loading animation to let them know the credit card info is coming.

With our user interface locked down, we now need to get the actual value for `cardholder_id`. We can do this by passing data from our outer controller to our inner, nested controller.

Passing Data Between Controllers

At this point, we're using a hard-coded value for `cardholder_id`, but we haven't explained what it is. In our simulated payment processor back-end, we need to pass an identifier of whose credit card information we want to show.

For the sake of simplicity, let's assume that, elsewhere in our hypothetical system, when we create credit card info with our payment processor, we explicitly state that the cardholder's ID is the same as our internal ID used

in the CUSTOMERS table. Thus, the value we get back from our contoller as customer_id should be the value to use for the cardholder_id.

Given that, how can we get the ID of the customer we're viewing inside CustomerCreditCardController? We could extract it from the route params like we do in CustomerDetailController, but this feels wrong, as it's copying code. It also makes our CustomerCreditCardController more tightly coupled to the view and routing configuration.

Another option is to reach into $scope. Because we've configured CustomerCreditCardController to be nested in the view that is powered by CustomerDetailController, Angular will pass in a scope that has the same data in it that CustomerDetailController has. This means that if we access $scope.customer inside CustomerCreditCardController, it will be the same object that CustomerDetailController is dealing with.

This approach isn't great because it maintains tight coupling between our two controllers. CustomerCreditCardController shouldn't require being nested inside a view related to CustomerDetailController if we can help it. The more CustomerCreditCardController can be decoupled the better, both for understanding the code and for testing it.

What we can do instead, is make the dependency between the two controllers more explicit. This allows CustomerCreditCardController and CustomerDetailController to not have to know about each other at all, which allows us to more easily work with both classes.

To do this, we'll use the directive ng-init. ng-init is used on the same element as ng-controller and allows you to execute code that has access to both scopes. In effect, we can use data exposed by CustomerDetailController and pass it to a function exposed by CustomerCreditCardController, like so:

```
<article ng-controller="CustomerCreditCardController"
         ng-init="setCardholderId(customer_id)">
```

It might seem odd to put this logic inside the view, and the Angular docs *do* warn against excessive use of this directive, but it's a great solution to our problem. It puts the code that couples the two controllers where the coupling actually exists: the view.

First, we'll create a function in CustomerCreditCardController that we can use in ng-init to set the customer id. This function will initiate the call to the payment processor back-end.

angular-async/ng-init/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerCreditCardController", [
          "$scope","$resource",
```

```
  function($scope , $resource) {
    var CreditCardInfo = $resource('/fake_billing.json')
    $scope.setCardholderId = function(cardholderId) {
      $scope.creditCard = CreditCardInfo.get(
        { "cardholder_id": cardholderId}
      )
    }
  }
]);
```

Notice how we use the name setCardholderId instead of setCustomerId. This reinforces the de-coupled nature of these two pieces of code—CustomerCreditCardController should not have to know that we are using the customer's ID as the cardholder ID in the payment processor's system.

It might seem like we could call this function in ng-init, like so:

```
<article
  ng-controller="CustomerCreditCardController"
  ng-init="setCardholderId(customer.customer_id)">
```

This won't work, however, because customer is a promise, and so won't initially have a value for customer_id. Fortunately, the customer's ID isn't a value we have to wait on from the AJAX response. Since it's part of the routeParams, CustomerDetailController already has access to it without hitting the back-ned. We can simply expose this value from CustomerDetailController and then use it in ng-init.

First, we'll expose the customer id to the view via $scope.

angular-async/ng-init/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerDetailController", [
          "$scope","$routeParams","$resource",
  function($scope , $routeParams , $resource) {
    $scope.customerId = $routeParams.id;
    var Customer = $resource('/customers/:customerId.json')

    $scope.customer = Customer.get({ "customerId": $scope.customerId})

    // rest of the controller...

  }
]);
```

Now, we can use it in ng-init:

angular-async/ng-init/shine/app/assets/javascripts/templates/customer_detail.html

```
<article
  ng-controller="CustomerCreditCardController"
  ng-init="setCardholderId(customerId)">
```

```
<div class="row">
  <div ng-if="!creditCard.$resolved">
```

With this in place, we're passing the cardholder ID to the fake payment processor, thus keeping all of our code separate and decoupled, but still able to handle getting data from all the right places. This wraps up our feature to view a customer's details, but it's worth re-visitng our testing setup in light of this new way of making AJAX calls.

Testing Controllers that Use Angular-resource

Before Chapter 7, *Test This Fancy New Code*, on page 103, our Angular code was using `$http` to make AJAX calls. Our tests used `$httpBackend` to mock out the HTTP interactions between our controller and the server. Will that still work, now that we are using Angular-Resource?

The short answer is "yes". Because Angular-Resource is using `$http` under the covers, the same techniques apply. In fact, you can use this information to refactor `CustomerSearchController` to use Angular-Resource.

Because Angular-Resource required us to understand promises, let's walk through a basic test of the `CustomerCreditCardController`, so it's clear how promises interact with our tests.

First, we'll set up our tests in the usual way:

angular-async/ng-init/shine/spec/javascripts/customers_app/controllers/customer_credit_card_controller_spec.js

```
describe("CustomerCreditCardController", function() {
  describe("Initialization", function() {
    var scope        = null,
        cardholderId = 42,
        cardInfo = {
               lastFour: '4321',
               cardType: 'visa',
        expirationMonth: 3,
         expirationYear: 2018,
            detailsLink: 'http://billing.example.com/foo'
        };

    beforeEach(module("customers"));

    beforeEach(inject(function ($controller,
                                $rootScope,
                                $httpBackend) {
      scope       = $rootScope.$new();
      httpBackend = $httpBackend;

➤     httpBackend.when(
```

```
➤          'GET',
➤          '/fake_billing.json?cardholder_id=' + cardholderId
➤        ).respond(cardInfo);

         controller  = $controller("CustomerCreditCardController", {
           $scope: scope
         });
       }));

       // tests will go here...

  });
});
```

You can see from the highlighted section that we're using $httpBackend the same way we did before, despite the fact that our controller isn't actually using $http.

To write the actual tests, we'll first write one that asserts that nothing happens when the controller is initialized, i.e. that no AJAX calls are initiated if setCardholderId has not been called.

angular-async/ng-init/shine/spec/javascripts/customers_app/controllers/customer_credit_card_con¬troller_spec.js

```
it("does not hit the backend initially", function() {
  expect(scope.creditCard).not.toBeDefined();
});
```

This test may seem spared, but it *will* fail if any AJAX calls are made during initialization of the controller. The next test is for the actual behavior of the controller, and this is where we'll walk through when the promises get resolved so it's clear how all this works.

angular-async/ng-init/shine/spec/javascripts/customers_app/controllers/customer_credit_card_con¬troller_spec.js

```
it("when setCardholderId is called, hits back-end", function() {
1  scope.setCardholderId(cardholderId);
2  expect(scope.creditCard).toBeDefined();
3  expect(scope.creditCard.lastFour).not.toBeDefined();
4  httpBackend.flush();
5  expect(scope.creditCard).toEqualData(cardInfo);
});
```

❶ Here we simulate what's going on in ng-init by calling the public function setCardholderId directly. After this call the AJAX call would be in-flight and the promise underlying $scope.creditCard would *not* be resolved.

❷ Although we wouldn't normally test Angular's underlying promise system, we can see it in action in our tests. Here we see scope.creditCard *is* defined.

❸ Here, we can see that although scope.creditCard is defined, it doesn't have any properties set on it. Again, you probably wouldn't test this for real, but it does demonstrate the order of operations.

❹ This line simulates the completion of the AJAX call by using httpBackend.flush() as we've seen before. At this point the promise *is* resolved, and we'd expect $scope.creditCard to have data in it.

❺ Finally, we can assert that we got data back from the server. scope.creditCard is now populate with all of the data we expect.

Promises can be somewhat confusing in your production code, and in test code it's even more confusing because you have to push each step through manually to see promises get created and then resolved.

It *is* worth pointing out how the separation of controllers made our tests simpler. If you imagine that we'd put all the code from CustomerCreditCardController in CustomerDetailController instead, our tests would be more complex. They would have to simulate two HTTP calls to make sure everything was working properly. By separating concerns and de-coupling, our controllers are easier to test.

Next Up: Sending Changes Back to the Server

In this chapter, we learned how to use Angular-Resource to easily fetch data from our Rails app (as well as a third-party payment processor), and to display it to the user as it became available using promises. But viewing data is only part of what we need to learn.

In the next chapter, we'll allow the user to edit some of the data and have it sent back to the server and updated automatically. We'll learn more about Angular's bindings and see how to handle the disparity between our materialized view of a customer's details and the actual database model that backs it.

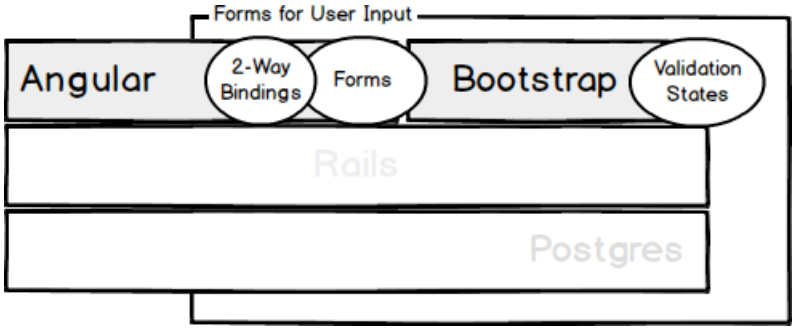
CHAPTER 12

Wrangle Forms and Validations with Angular

So far, we've treated our application as a producer of information, as it shows data quickly, easily, and in a usable way. But applications need to consume information as well, and that's what we're going to learn about in this chapter. So far, we've designed a complex form which shows the user some information about a customer, and the next step is to let the user modify that information.

A typical Rails application has access to various helpers, both in Active Record and in the view, to give the user a good experience. You can specify that a field is required or must match a certain format (for example, the validators we used with Devise in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17). Since we are relying on Angular to render our form, we don't have access to all of Rails' features to let users update the data.

We're going to learn how to get the same benefits and give the user the same great experience, but using Angular's form support, along with Bootstrap's styles for field validation. This will be a true test of our abilities as a full-stack developer, because Angular and Bootstrap were not designed to work in concert, so it'll be a bit tricky making it all work.



To understand how to do this, we'll first learn about *bindings*, which is the core of how Angular manages dynamic values and is the underpinning of `ng-model`. We'll then build on this to learn about Angular's form support. *Angular Forms* provide various hooks into the binding lifecycle, which will allow us to detect when values have changed, inject validations, and generally manage our code for the front-end.

From there, we'll learn how to use Angular Forms with Bootstrap's styles so that our form fields properly indicate their valid or invalid states to the user. Then, we'll see how to save the data back to the server. We'll also see how Rails' validators relate to all of this.

But first, let's learn about bindings.

Managing Client-side State with Bindings

In Chapter 11, *Load Data from Many Sources Asynchronously*, on page 193 we learned a bit about bindings, that they allow Angular to detect changes to our model so view can be updated. Angular's bindings are actually *two-way*, meaning that changes on the form are also reflected back on the model.

We actually saw this in action when we created our Angular test app in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81. If you recall, our app rendered whatever was typed in a text field as the value for an *h1* tag. It was Angular's bindings that made this happen.

Now that we are going to allow users to edit the fields in our form, we'll need to understand more details about how bindings work. On the surface, it may seem that when a user is editing a form field, the value of that field is copied back to the model. While true, there's much more to it.

Angular's bindings keep track of detailed states of the control:

Valid vs. Invalid Angular will keep track of the validity of a field, and will *not* sync invalid values back to the model. It can do this using HTML5 validations (like `required`[1]), or custom validators that you write.

Dirty vs. Pristine A control is *pristine* if the user has yet to interact with it and dirty otherwise. Angular tracks this, which you can use to avoid showing a user a fresh, blank form with a bunch of error flags for required fields.

Touched vs. Untouched A control has been *touched* if it's been blurred (e.g. the user has tabbed away from it) and untouched otherwise. Angular can track this, too (which you can use to only sync values or show errors after the user is done interacting with the control).

Asynchronous Validation Angular allows you to write validations that are the result of external, asynchronous calls. This can useful when you need to hit the server for validations.

This feature list should come as no surprise to you at this point, because it's complex, yet powerful—a great description of Angular in general. In the next section, we'll use these features by applying HTML5 validators to our markup. This will add Angular's validations to our form.

Validating User Input with Angular Forms

In addition to the properties of a binding we discussed above, Angular provides sophisticated features to interact with form data[2]. We can use these features to implement various validations on our input fields using HTML5's validations. For example, we can indicate a field is required using `required` like so:

```
<input type="text" name="first_name" required>
```

HTML5 validations, and thus Angular's form validations, can do more than just require values. Here's the validations we're going to implement, and it won't require very much code to make them work:

- First name, last name, and username are required.
- Email is required and should look like an email address.
- Address and city are required.
- State should be exactly two upper-case letters.

1. http://www.w3.org/TR/2011/WD-html5-20110525/common-input-element-attributes.html#attr-input-required
2. https://docs.angularjs.org/guide/forms

- Zip code should be exactly five digits.

To implement these validations, we'll first add the necessary markup to indicate what the validations on each field are. Next, we'll see exactly how this information is exposed to our JavaScript code, before finally adding a submit button that is only enabled if all fields are valid. This will teach us how the mechanics of Angular's form validation works, so we'll be ready to style our validated fields in the next section.

Adding Validation Markup

Angular's built-in validations are documented on each of the input components to which they apply[3]. Since we are using almost entirely text fields, we'll be using the validators for input type="text"[4].

Of course, our email field should use the new HTML5 input type="email", because that will trigger Angular's email pattern-matching validation. So, let's convert that field:

angular-forms/start/shine/app/assets/javascripts/templates/customer_detail.html

```
<div class="form-group">
  <label class="sr-only" for="email">Email</label>
➤ <input type="email" class="form-control" required
➤        name="email" ng-model="customer.email">
</div>
```

You'll also notice that we used the HTML5 required attribute. Angular will pick this up, too, and use it to validate that this field has a value.

Of course, using HTML5 validations creates a problem, because the browser will try to validate these values, too. Since Angular is going to handle that, we need to disable browser validation on the form element using the novalidate attribute:

```
➤ <form novalidate><div class="row">

    <!-- rest of the markup -->

  </div></form>
```

With that out of the way, we can proceed to add validations for the remaining fields. For first name, last name, username, the two address and two city fields, we just need to use required like so:

3. https://docs.angularjs.org/api/ng/input
4. https://docs.angularjs.org/api/ng/input/input%5Btext%5D

angular-forms/start/shine/app/assets/javascripts/templates/customer_detail.html
```
<div class="form-group" >
  <label class="sr-only" for="first-name">
    First Name
  </label>
  <input type="text" class="form-control" required
         name="first_name" ng-model="customer.first_name">
</div>
```

The markup should look similar for the other fields.

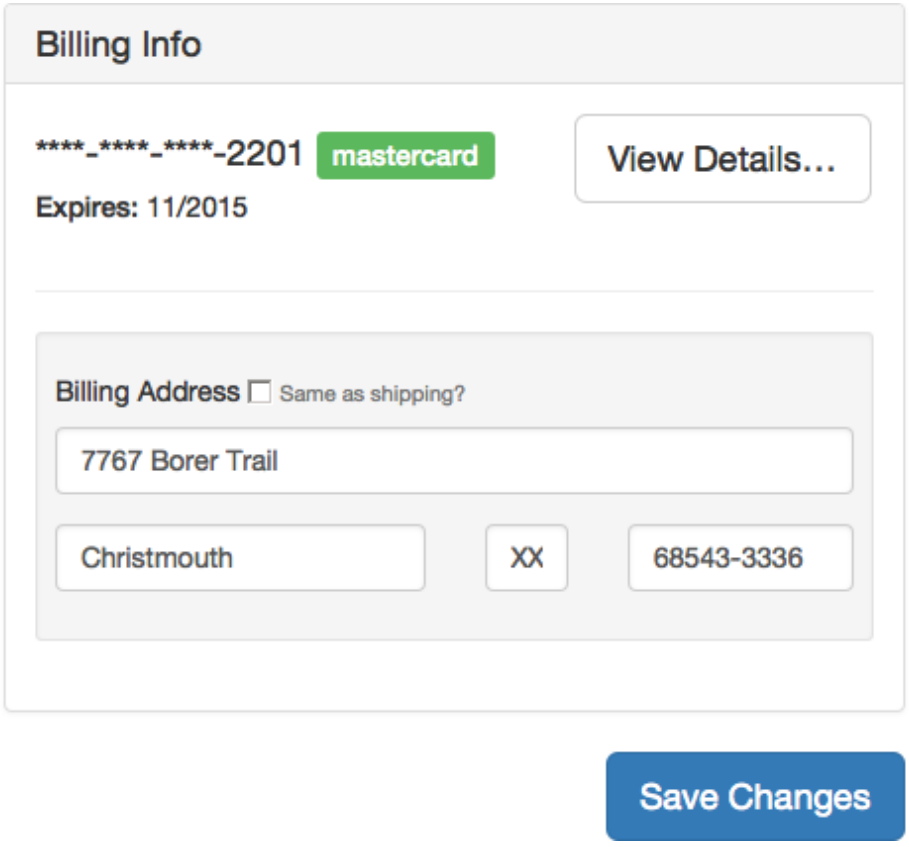
For the state and zipcode fields, we'll still use required, but we'll also use the HTML5 pattern attribute, which will allow us to specify a regular expression that the value of the field must match. For the two state fields, the regular expression [A-Z][A-Z] should suffice to capture a two-letter state code. For the zipcode fields, \d\d\d\d\d will capture the 5 digits required for a U.S. zipcode.

angular-forms/start/shine/app/assets/javascripts/templates/customer_detail.html
```
<div class="col-md-2">
  <div class="form-group">
    <label class="sr-only" for="state">State</label>
    <input type="text" class="form-control"
           required pattern="[A-Z][A-Z]"
           name="shippingState" ng-model="customer.shipping_state">
  </div>
</div>
<div class="col-md-4">
  <div class="form-group">
    <label class="sr-only" for="zip">Zip</label>
    <input type="text" class="form-control"
           required pattern="\d\d\d\d\d"
           name="shippingZip" ng-model="customer.shipping_zipcode">
  </div>
</div>
```

Now that we've added markup for validations to our fields, let's add a submit button for the user to use to save their changes. We'll add the button at the bottom of the grid cell that holds the billing info.

angular-forms/start/shine/app/assets/javascripts/templates/customer_detail.html
```
<form novalidate name="form"><div class="row">

    <!-- rest of the markup -->

  <div class="col-md-6">

    <!-- customer info & shipping markup -->

  </div>
  <div class="col-md-6 billing-info">
```

```
      <article class="panel panel-default">

        <!-- billing info markup -->

      </article>
      <div class="text-right">
        <button ng-click="save()" class="btn btn-lg btn-primary">Save Changes</button>
      </div>
    </div>
  </div></form>
```



You'll note that we used `ng-click` to bind this button to the `save` function. We haven't written that function yet, but we will in the next section, where we'll see how this validation works.

Accessing and Handling Validation Results

With the small changes we've made to the markup, we could prevent invalid submissions. When the value of a bound field is invalid, Angular will set that field's corresponding model property to `undefined`. We could check that all

properties of $scope.customer are defined and submit to the server only in this case.

This is bad for two reasons: it would be a lot of code that would need changing if we changed the form fields, and it wouldn't afford a great user experience. We'd like to show the user, on the form, where the problems are, as the user is making mistakes, as well as some messaging about what they need to do to fix them. Angular provides all of this for us, on an instance of a NgFormController.

The NgFormController can be queried at any time to get the current state of the form's validations (and other information). To get access to it, we just need to give our form element a name attribute:

angular-forms/start/shine/app/assets/javascripts/templates/customer_detail.html

```
<form novalidate name="form"><div class="row">

    <!-- rest of the markup -->

</div></form>
```

Now, we can access an object named $scope.form in our controller. We can ask it if the form is valid by calling the $valid function (note the leading $, which Angular uses frequently to indicate an Angular-provided function). This function is similar to the valid? method provided by ActiveRecord, with a small exception: $valid is a property that becomes true or false as the data becomes valid or invalid. As you'll recall, valid? is a method that checks for validity only when called.

Let's try it out. In app/assets/javascripts/customers_app.js, implement save like so:

```
$scope.save = function() {
  alert($scope.form.$valid);
}
```

If you search for a customer, and then click Save Changes, you should see a JavaScript alert with the message true. If you then modify the data to be invalid—based on our rules and markup—and click again, you'll see false. This is already a much better way to check for validity than iterating through each of the values looking for undefined.

The NgFormController gives us much more, however. Like ActiveRecord's Errors class, we can access detailed information about what's wrong with each field. For each field in our form, there is a corresponding object on the NgFormController. *Those* objects expose a lot of data about the field, include a property $valid,

indicating if the field's value is valid, and $error, which is an object describing the errors.

Let's see this action. We'll update our save function to show us information about the email field's validity.

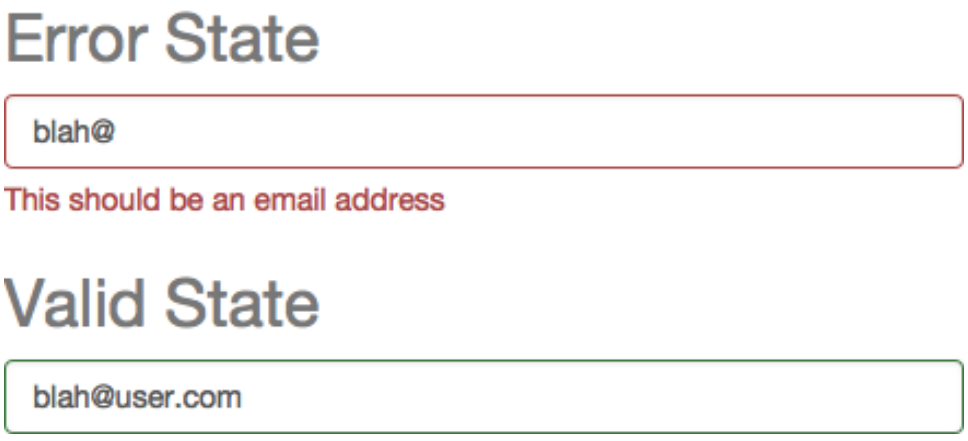
```
$scope.save = function() {
  if ($scope.form.email.$valid) {
    alert("Email is valid");
  } else if ($scope.form.email.$error.required) {
    alert("Email is required");
  } else if ($scope.form.email.$error.email) {
    alert("Email must look like an email");
  }
}
```

With this in place, let's play around with different values for the email field. What Angular is doing is setting keys in the $error object on our field that indicate what the problem is. We'll see how to use these in the next section.

In addition to showing the user the errors from $errors, we'll also make the user experience better using Bootstrap. The JavaScript alerts we're currently using aren't a great experience, so we'll make them better.

Styling Invalid Fields with Bootstrap

Now that we can detect which fields are invalid (and why), we want to show the user this information so they can correct the issue. We could just show a list of errors at the top of the screen, but we can do better. Bootstrap provides classes to allow us to highlight valid and invalid fields as well as to show error messages, like so:



Bootstrap refers to this as *feedback*. To show the error state above, we'd add the classes has-feedback and has-error to the div containing our form elements. To make sure the error message text picks up the same red styling, we'll put it inside a p that uses the help-block class:

```
<div class="form-group has-feedback has-error">
  <input type="email" class="form-control">
  <p class="help-block">
    This should be an email address
  </p>
</div>
```

For the valid state above, we'd use the classes has-feedback and has-success on the div:

```
<div class="form-group has-feedback has-success">
  <input type="email" class="form-control">
</div>
```

To use these features, we'll need to learn two things: where the messages will come from and how to conditionally style our form elements.

Displaying Error Messages

With the knowledge we currently have, we could conditionally display error messages like so:

```
<p class="help-block" ng-if="form.email.$invalid">
  <span ng-if="form.email.$error.required">Email is required</span>
  <span ng-if="form.email.$error.email">Email must look like an email</span>
</p>
```

Angular has a module that makes this a bit easier, called *Angular Messages*, and will allow us to write this:

```
<p class="help-block" ng-messages="form.email.$error">
  <span ng-message="required">Email is required</span>
  <span ng-message="email">Email must look like an email</span>
</p>
```

We'll install it by adding angular-messages to our Bowerfile:

angular-forms/bootstrap/shine/Bowerfile

```
asset 'bootstrap-sass-official'
asset 'angular', '~> 1.4'
asset 'angular-mocks'
asset 'angular-route'
asset 'angular-resource'
asset 'angular-messages'
```

We'll then install it with rake bower:install and add it to app/assets/javascripts/application.js:

angular-forms/bootstrap/shine/app/assets/javascripts/application.js

```
//= require angular
//= require angular-route
//= require angular-resource
//= require angular-messages
//= require angular-rails-templates
//= require_tree ./templates
//= require_tree .
```

As we learned in *Installing Angular's Router*, on page 141, we must translate the module's name so we can add it to our code to indicate it as a dependency of our Angular app. The translated name to use is ngMessages:

angular-forms/bootstrap/shine/app/assets/javascripts/customers_app.js

```
var app = angular.module(
  'customers',
  [
    'ngRoute',
    'ngResource',
    'ngMessages',
    'templates'
  ]
);
```

Now, we can add this code to each form element:

angular-forms/bootstrap/shine/app/assets/javascripts/templates/customer_detail.html

```
<label class="sr-only" for="email">Email</label>
<input type="email" class="form-control" required
       name="email" ng-model="customer.email">
<p class="help-block" ng-messages="form.email.$error">
  <span ng-message="required">Email is required</span>
  <span ng-message="email">Email must look like an email</span>
</p>
```

Note that we'll have to use different values for ng-message depending on the validations we added on that field. For example, the shipping zipcode field will look like this:

angular-forms/bootstrap/shine/app/assets/javascripts/templates/customer_detail.html

```
<label class="sr-only" for="shippingZip">Zip</label>
<input type="text" class="form-control"
       required pattern="\d\d\d\d\d"
       name="shippingZip" ng-model="customer.shipping_zipcode">
<p class="help-block" ng-messages="form.shippingZip.$error">
  <span ng-message="required">Zip is required</span>
  <span ng-message="pattern">Zip must be five digits</span>
</p>
```

Once we add this code to the rest of the form fields, the user will get great feedback on what they've done wrong. But, we'll want to visually highlight the errors as well, using the has-feedback classes we talked about earlier.

Conditionally Styling Form Elements

We'd like to show invalid fields in red, and valid fields in green, however we don't want to show any special highlighting for fields whose values haven't been touched or changed.

The reason for this is that for a split-second when our view template loads, there will be no values in the fields as the controller fetches them from the backend. Since we've configured most fields to be required, they would be considered invalid until the AJAX call completed. This is why $pristine exists.

Further, we don't want to show a field as green unless the user has actually done something to the field. This is where $touched comes into play.

This means that the logic for showing the error/invalid state would require that both form.email.$invalid *and* form.email.$dirty be true. Similarly, the logic for showing the green success state would require that both form.email.$valid *and* form.email.$touched be true.

So, how do we connect that logic to the classes Bootstrap provides? The answer is ng-class. The ng-class directive[5] allows us to conditionally add classes to an element.

The value to use for the directive depends on what you are trying to do. For simple uses, we can provide an expression that evaluates to a string. For example:

```
<div ng-class="'email-' + form.email.$valid">
</div>
```

The expression we gave to ng-class is 'email-' + form.email.$valid, which, if the email field is invalid, evaluates to 'email-false'. Thus, the browser will see this:

```
<div class="email-false">
</div>
```

Unfortunately, the Bootstrap feedback classes don't work in a way that allows us to build up a class name dynamically like this (or at least, not in a way that doesn't involve a lot of convoluted code). So, we'll use the second form that ng-class accepts, which is an object syntax.

5. https://docs.angularjs.org/api/ng/directive/ngClass

In this syntax, the value is a JavaScript object, where the keys represent different classes that might be applied to the element. For each key, if it's value evaluates to true, that key is applied as a class on the element. The above example using the object syntax would look like so:

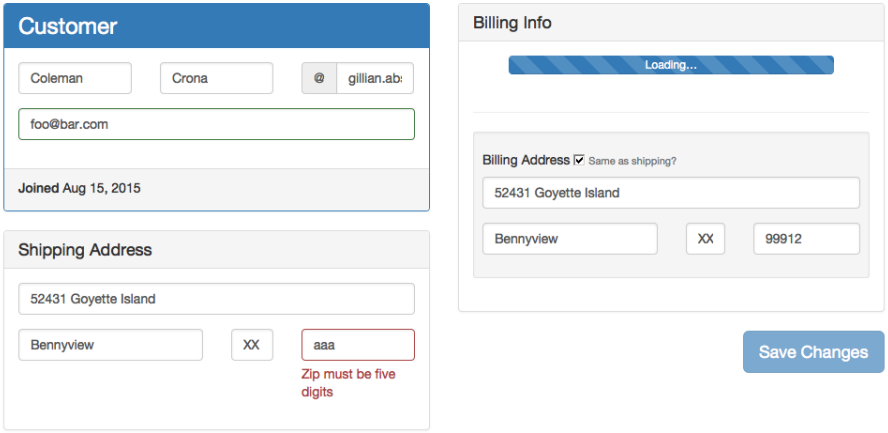
```
<div ng-class="{ 'email-false': form.email.$invalid,
                 'email-true': form.email.$valid }">
</div>
```

This syntax is much cleaner for our purposes. To mark up the email field, we'd do this:

angular-forms/bootstrap/shine/app/assets/javascripts/templates/customer_detail.html

```
<div class="form-group has-feedback" ng-class="{
    'has-error': form.email.$invalid && form.email.$dirty,
  'has-success': form.email.$valid   && form.email.$touched
}">

  <!-- form markup from before... -->

</div>
```

Try it out by clearing the email address, then changing the value to a non-email, then changing it to a valid email address. You'll see that the field will be red or green, depending on the value, and the error messages will change dynamically, depending on what the issue is.



This is exactly what we want! We'll need to add these lines of code to each field's form-group. It might feel a bit repetitive, and there *is* a way to extract this boilerplate using custom *directives*. This is an advanced topic that we'll go over briefly in Chapter 13, *Dig Deeper*, on page 237 but for now, we'll just deal with a bit of duplication.

The last thing we should do is to disable the Save button if the form is invalid.

Disabling the Save Button if Data is Invalid

As a last indicator to the user that something is wrong, we don't want them to be able to click the "Save Changes" button if there are any validation errors. We can do this in two steps. First, we'll disable the button using the ng-disabled directive. This will add disabled="disabled" to the button, which Bootstrap will see and render the button in a lighter disabled state.

angular-forms/bootstrap/shine/app/assets/javascripts/templates/customer_detail.html

```
<div class="text-right">
  <button ng-click="save()"
          class="btn btn-lg btn-primary"
          ng-disabled="form.$invalid || form.$pristine">
    Save Changes
  </button>
</div>
```

Note that we've also disabled it if the form is pristine. In this case no data has changed, and there's nothing to save to the server.

We'll also change our save function to no-op if the form is invalid.

angular-forms/bootstrap/shine/app/assets/javascripts/customers_app.js

```
$scope.save = function() {
  if ($scope.form.$valid) {
    alert("Save!");
  }
}
```

This isn't strictly necessary, because the browser should disable the button, but it can't hurt, and it prevents our controller code breaking because of changes in the view.

Our form now works and looks great. Users can see clear indicators when fields are valid or invalid, and get messages explaining the problem, right at the fields in question. The last thing we need to do is actually save the data back to the server.

Saving Data Back to the Server

Normally, saving back to the server wouldn't be terribly interesting, since we could just post a JSON blob that Rails expects to be given to an Active Record's update method. If you'll recall, our CustomerDetail class is backed by a materialized view and *not* a regular table. That means that we can't call update on it, since you can't update a materialized view.

That means we'll have to figure out how to get the data out of our de-normalized CustomerDetail and into the right places in the database. But first, let's get our Angular app and Rails backend setup to receive the data.

Sending Data Back to the Server

First, let's set up the update action for CustomersController, which is what will receive the payload from our Angular app to update the customer details. We'll modify the route in config/routes.rb like so:

angular-forms/save/shine/config/routes.rb

```
resources :customers, only: [ :index, :show, :update ]
# ------------------------------------------^^^^^^^
```

We'll also add a simple controller action that just prints out the parameters:

```
class CustomersController < ApplicationController

  # rest of the controller...

➤   def update
➤     puts params.inspect
➤   end
end
```

Lastly, we'll modify the save function in CustomerDetailController to post data to the server. As you recall, we used Angular Resource to fetch data from the back-end. The object we got back from Customer.get contains functions provided by Angular Resource to save to the server. They function we want for this case is $save(). Note the preceding $ that indicates this is an Angular-provided function.

Calling $save() will send the data inside $scope.customer to the server by POSTing it to /customers.json. This isn't quite what we want. The route we set up in our Rails application wants the data sent via PUT or PATCH, and it wants it sent to a url with the customer's ID in it, e.g. /customers.12345.json.

Fortunately, these are easy to handle by adjusting our call to $resource. Currently, this looks like so:

```
var Customer = $resource('/customers/:customerId.json');
```

When we called Customer.get, we supplied the value for :customerId as the second argument. Since we're calling $save() on $scope.customer, Angular can derive the proper value with a little help. The second argument to $resource is an object that describes how to do this.

That object's keys should be all the dynamic bits of the URL. The values are strings that explain how to get those values. If the values for the object's keys start with @ that tells Angular to look for that attribute on the object itself. So, { "customerId": "@customer_id" } instructs Angular to fill in :customerId with whatever the value of customer_id is on the object on which we are calling $save().

To handle the HTTP method, we can use the third optional argument to $resource. This object is highly flexible, allowing anything from customizing the HTTP method (what we need) to adding custom functions and actions. In our case, we just need to indicate that the *save* action uses the HTTP method *PUT*. We can do this with { "save": { "method": "PUT" }}. Bringing it all together looks like so:

angular-forms/save/shine/app/assets/javascripts/customers_app.js

```
app.controller("CustomerDetailController", [
          "$scope","$routeParams","$resource",
  function($scope , $routeParams , $resource) {
    $scope.customerId = $routeParams.id;
    var Customer = $resource('/customers/:customerId.json',
                             {"customerId": "@customer_id"},
                             { "save": { "method": "PUT" }});

    // rest of the controller...

  }
]);
```

Now, when we call $save() on $scope.customer, angular will look it $scope.customer.customer_id to create the URL to send back to the server, using the PUT HTTP method.

$save() takes two parameters: a success callback and an error callback. For now, let's keep them simple and just alert the user about what happened. We'll also call $setPristine and $setUntouched on $scope.form to reset it to its normal state. That will clear any green highlights due to successfully changing form values.

angular-forms/save/shine/app/assets/javascripts/customers_app.js

```
$scope.save = function() {
  if ($scope.form.$valid) {
    $scope.customer.$save(
      function() {
        $scope.form.$setPristine();
        $scope.form.$setUntouched();
        alert("Save Successful!");
      },
      function() {
        alert("Save Failed :(");
```

```
      }
    );
  }
}
```

We can restart our app, navigate to a detail page, modify one of the fields and click save. If we look at the Rails server log, we'll be somewhat disappointed in the results.

```
"PUT /customers/217689.json HTTP/1.1" 422 98873 0.2421
```

Instead of seeing the parameters printed out, we just get a lonely 422, which means "Unprocessable Entity". This is Rails Cross-Site Request Forgery protection[6] getting in our way. CSRF protection is a feature of Rails that prevents malicious entities from trying to hack our systems by exploiting non-GET endpoints. While we could integrate this into our Angular app[7], the Rails documentation for RequestForgeryProtection[8], recommends turning it off for AJAX requests like ours.

We can do this by conditionally skipping the request forgery protection in our ApplicationController:

angular-forms/save/shine/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  skip_before_action :verify_authenticity_token, if: :json_request?
  before_action :authenticate_user!

protected

  def json_request?
    request.format.json?
  end

end
```

If we try to save our changes again, we should see our log output the parameters we passed:

```
{ "customer_id"=>217689, "first_name"=>"Bobby", "last_name"=>"Abernathy", "e...
```

Now, we just need to save them to the database.

6. http://guides.rubyonrails.org/security.html#cross-site-request-forgery-csrf
7. http://stackoverflow.com/questions/14734243/rails-csrf-protection-angular-js-protect-from-forgery-makes-me-to-log-out-on
8. http://api.rubyonrails.org/classes/ActionController/RequestForgeryProtection.html

Saving Data from our Materialized View to the Database

As you recall, this data comes from the CustomerDetail class, which is backed by a materialized view. As such, we can't just call update as you can't update a materialized view using the SQL UPDATE command. But, we'd still like our controller code to look as Rails-like as possible. So, let's plan to override update on CustomerDetail to handle the necessary logic so our controller looks unsurprising.

angular-forms/save/shine/app/controllers/customers_controller.rb

```
def update
  customer_detail = CustomerDetail.find(params[:id])
  customer_detail.update(params)
  head :ok
end
```

Note that we're just returning an HTTP HEAD of 200 (:ok). There really isn't any data to report back to the caller, so this will be sufficient to let our Angular app know that everything was saved properly.

Now, we need to implement update in CustomerDetail. To keep this simple, we'll use the ids in our payload to find the Customer and two Address records and then use their update methods to update them.

We'll need a helper method to translate keys like billing_street to street, since both addresses are the same class.

angular-forms/save/shine/app/models/customer_detail.rb

```
class CustomerDetail < ActiveRecord::Base

  # rest of the class...

  def update(params)
    Customer.find(self.customer_id).update(
      params.permit(:first_name, :last_name, :username, :email))

    Address.find(self.billing_address_id).update(
      address_attributes(params,"billing"))

    Address.find(self.shipping_address_id).update(
      address_attributes(params, "shipping"))
  end

private

  def address_attributes(params, type)
    {
       street: params["#{type}_street"],
         city: params["#{type}_city"],
```

```
        state: State.find_by_code(params["#{type}_state"]),
      zipcode: params["#{type}_zipcode"],
    }
  end
end
```

We're not going to dwell too much on this, since this is simple Rails code that you are likely used to. It is worth pointing out that code like this is the downside to what we've done with the materialized view. This is the price we're paying to make our query super fast.

If we reload our page, modify some data, and click "Save Changes" we should see an alert indicating success, and our green highlights should disappear. If we reload the page again (forcing our Angular app to hit the server), we should see that the data saved successfully.

Our customer detail feature is done! But you may be wondering about Rails validators. We aren't using them at all and, in fact, have done all validation on the client side. Usually, you don't want the client to be the only place where you validate data, because it doesn't prevent malicious users (or buggy code) from submitting invalid data to the server.

Understanding the Role of Rails Validators

In a normal Rails application, the use of validators is crucial, as they integrate with Rails form helpers and provide a similar experience to the one we provided with Angular's form support. Given that we aren't using Rails views, where do validators fit it?

There are a few strategies and schools of thought here, and you will need to apply some combination of them to your day-to-day features.

One school of thought is that, given the power of Postgres, there's no need for Rails validators when using Angular for our front-end. We would implement all validation on the front-end, so the user gets a great experience and understands what the data should look like, and then we use Postgres check constraints (like the one we created in Chapter 3, *Secure the Login Database with Postgres Constraints*, on page 41) to ensure bad data never gets written to the database.

This is the most ideal situation because it's the least duplication of code and puts everything closest to where it matters: user-friendly validations are in the view, where they are helpful to users, and database constraints are in the database, making it difficult or impossible for anyone to put bad data there. Further, your database will most likely outlive your application code,

so having it be the canonical store of what is and isn't valid data makes the most sense.

That said, we aren't always starting from scratch, or might not be able to easily encode all of our constraints as database check constraints. For example, we might insist that the user's email address resides in a third-party system we use for managing marketing campaigns. The database can't check that. It could also be difficult to have our Angular app check that, so it would need to reside in our middleware, i.e. Rails.

In these cases, you can use Rails validators (or any middleware mechanism you want) by using the `$asyncValidators` attribute of your form. This object's keys are the names of custom validations you want to run against the server. Their value is a function that takes the new and old values and is expected to return a promise that, when resolved succeeds or fails if the validation passes or fails. The Angular docs for `NgFormController`[9] contain some good examples.

With this mechanism, you can have your server-side code do the complex checks for you.

A final school of thought is to just use Rails to do all validations. In this case, you'll need to do more work on the Angular app side. You'll need to implement a failure callback to `$save` that extracts the ActiveRecord error object out of the response and marks up each field with the errors that came back. This is probably not worth it unless you have a lot of Rails-powered views that will share this code.

Next Up: Everything Else

At this point, our work is done! We've createed a highly complex page with a clean design, performs well, and gives a great user experience. It took expertise in Bootstrap, Angular, and Postgres to make it happen. Take a moment to click around Shine and reflect on what we've done.

We've hardly written any code, yet every layer of our application is clean. We didn't write any CSS, and we've only written the JavaScript specific to the problems we're solving: no hacky `data-` attributes or code tightly-coupled to the DOM. Our Rails controller code looks like a regular Rails controller, even though its backed by a powerful, self-updating materialized view of our complex data.

9. https://docs.angularjs.org/api/ng/type/ngModel.NgModelController

There is so much more we could cover. We've only hit the tip of the iceberg with these technologies, but hopefully you are starting to see how employing Angular, Bootstrap, Postgres together with Rails allows you to do get a lot of great work done with little effort. You can keep leveling yourself up by opening the documentation and getting inspired.

But before I send you on your way, I want to take one last trip through these technologies to show some of the other possibilities and amazing things you can do with them. The next chapter is a grab bag of everything we didn't have space to get in to earlier.

CHAPTER 13

Dig Deeper

Throughout the book, we implemented a simple feature to search for, view, and edit customer details. This was a great framing example to learn a lot about full-stack development, from optimizing our database to creating a great user experience. Hopefully, this has emboldened you to dig deeper into the tools you are using, where you'll find easy solutions to the problems you face day-to-day.

And Angular, Postgres, and Bootstrap are *deep* tools. This section is going to expose you to some more features that you will find handy, but that we don't have space to delve into deeply. Unlike the previous chapters, we won't wind these into a tale of feature development, but rather show terse examples with links for more information. Think of this section as inspiration for the work you'll be doing.

Unlocking More of Postgres' Power

For most of my career, I viewed SQL databases as dumb stores for simplistic data. Postgres has shattered that view, and this section will do the same for you. We'll talk about many advanced column types, like JSON, Arrays and enumerated types. We'll then see that Postgres supports full-text search out of the box before finishing up with the most mundane yet frequently-needed tasks: CSV export.

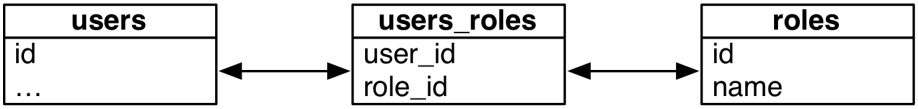
Modeling your Data with Advanced Column Types

Most databases store numbers and strings. For storing more advanced structures, like arrays or maps, you typically have to create them using tables. Postgres provides more advanced types to avoid doing that, and they can be a huge time-saver. Let's go over a few of them and show how they might be useful.

Arrays

Suppose we want to add role-based security to Shine. That is, a user can have zero or more roles, and each action in Shine requires that the user has a certain role. You might need the view_billing role to view a customer's billing info, or the edit_customer role to edit a customer's data.

In most SQL databases, you would need to create two new tables. The first would be the table of roles, and the second would be a join table that joins users to roles (USERS_ROLES).



In Postgres, you can avoid all this and just use an array on customers. This works pretty much like you'd expect. Active Record supports this datatype by using array: true when defining the column. Here's how we'd add this to our existing USERS table:

grab-bag/one/shine/db/migrate/20150820224152_add_roles.rb

```
class AddRoles < ActiveRecord::Migration
  def change
    add_column :users, :roles, :string, array: true, default: []
  end
end
```

We can then uses this like a normal array in Ruby:

```
> rails c
:001 > u = User.first
  User Load (0.8ms)  SELECT  "users".* FROM "users"  ORDER BY "users"."id"…
 => #<User id: 1, email: "dave@example.com", encrypted_password: "$2a$10$aa…
:002 > u.roles = [ "admin", "edit_customers" ]
 => ["admin", "edit_customers"]
:003 > u.save!
  SQL (0.5ms)  UPDATE "users" SET "roles" = $1, "updated_at" = $2 WHERE "us…
 => true
:004 > u.reload
  User Load (0.6ms)  SELECT  "users".* FROM "users"  ORDER BY "users"."id"…
 => #<User id: 1, email: "dave@example.com", encrypted_password: "$2a$10$aa…
:005 > u.roles
 => ["admin", "edit_customers"]
:006 > u.roles.include?("admin")
 => true
```

But that's not all. An array can be searched with SQL. Suppose we wanted to find all the users with the role edit_shipment. Postgres provides the @> operator to do just that.

```
SELECT
  *
FROM
  users
WHERE
  roles @> ARRAY['edit_shipment'::varchar];
```

The ::varchar is needed to allow Postgres to compare the values in our array with the literal 'edit_shipment' we used. Postgres treats string literals as the type TEXT, but Rails declared our array to be an array of VARCHAR.

This is already quite handy, but it gets better. We can *index* this array so that a query like the one above can be optimized. If we have a large USERS table, we'll see that the query is somewhat slow. If we do EXPLAIN ANALYZE on the query, we'll see the dreaded Seq Scan. As long as we are using the @> operator, however, we can improve the query by creating an index.

We can't just create a normal index, however. Postgres can only index arrays if you use a different index type. In Chapter 4, *Use Fast Queries with Advanced Postgres Indexes*, on page 49, we created an index using a special operator class. There is an additional dimension to indexes we can control in the same way called the *index type*[1]. The default, a B-Tree index, cannot index an Array value. But, the GIN type can. GIN[2] stands for General Inverted Index, and this index will take up more disk space and be slower to create than a B-Tree. But, it indexes arrays.

```
CREATE INDEX
  users_roles
ON
  users
USING GIN (roles)
```

The key part of this is USING GIN. With this in place, the query is now quite fast, and our EXPLAIN ANALYZE indicates the index is being used:

```
> EXPLAIN ANALYZE
  SELECT * FROM users
  WHERE roles @> ARRAY['edit_shipment'::varchar];
                            QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on admin_users  (cost=9.12..268.20 rows=145 width=…
```

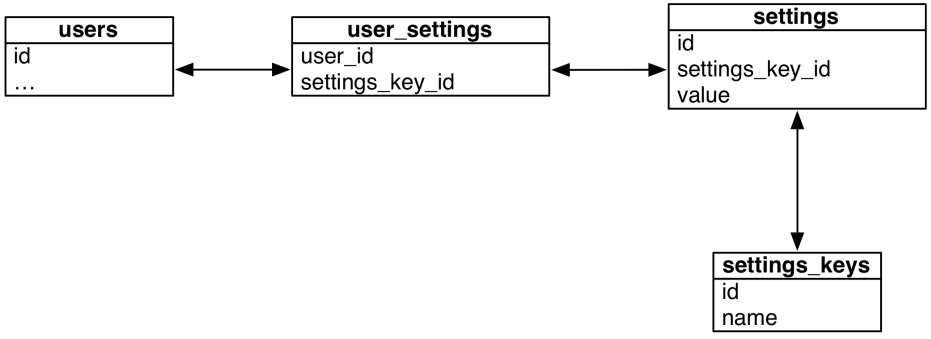
1. http://www.postgresql.org/docs/9.1/static/indexes-types.html
2. http://www.postgresql.org/docs/9.1/static/textsearch-indexes.html

```
                                (actual time=0.073..0.175 rows=145…
  Recheck Cond: (roles @> '{edit_shipment}'::character varying[])
  Heap Blocks: exact=93
  ->  Bitmap Index Scan on admin_users_roles
              (cost=0.00..9.09 rows=145 width=0)
              (actual time=0.056..0.056 rows=145 loops=1)
        Index Cond: (roles @> '{edit_shipment}'::character varying[…
Planning time: 0.235 ms
Execution time: 0.219 ms
```

This means that you can use an array on a very large table, and still get great performance out of it—performance that would certainly surpass using a join table.

HSTORE

Another common need we have when modeling data is to store key/value pairs. In most SQL databases, you would need to set up a generalized set of tables to do this, likely a table of possible keys, a table of possible values, and a join table to connect them to the table that needs them. An example might be user settings.



The HSTORE[3] type can solve this with a single field on USERS called settings. An HSTORE is a single-depth key/value store. Active Record supports it, and will treat it like a Ruby hash in your code.

This type isn't available by default, but can be enabled via the command enable_extension, which we can do in our Rails migration via execute. Once we've done that, we can create the column using the :hstore type:

grab-bag/one/shine/db/migrate/20150820231159_add_settings.rb

```
class AddSettings < ActiveRecord::Migration
  def up
    enable_extension :hstore
```

3. http://www.postgresql.org/docs/9.4/static/hstore.html

```
    add_column :users, :settings, :hstore, default: {}
  end
  def down
    remove_column :users, :settings
    disable_extension :hstore
  end
end
```

Now, the settings attribute behaves like a Ruby hash:

```
> rails c
:001 > u = User.first
  User Load (2.2ms)  SELECT  "users".* FROM "users"  ORDER BY "users".…
 => #<User id: 1, email: "dave@example.com", encrypted_password: "$2a$…
:002 > u.settings[:page_size] = 20
 => 20
:003 > u.save!
  SQL (0.4ms)  UPDATE "users" SET "settings" = $1, "updated_at" = $2 W…
 => true
:004 > u.settings[:page_size]
 => 20
```

To query this in SQL, use the -> operator, which works like Ruby's square brackets.

```
>  select id,settings from users where (settings->'page_size')::int > 0;
-[ RECORD 1 ]----------+-----------------------------
id                     | 1
settings               | "page_size"=>"20"

> select * from users where (settings->'page_size')::int > 40;
(No rows)
```

Note that we need to cast the result of settings->'page_size' to an int, because Postgres stores the values as strings. You can also check if an HSTORE field contains a key via the ? operator:

```
>  select id,settings from users where (settings?'page_size');
-[ RECORD 1 ]----------+-----------------------------
id                     | 1
settings               | "page_size"=>"20"

> select * from users where (settings?'foo');
(No rows)
```

Indexing options for HSTOREs are limited. You can use a GIN index to assist with the ? operator, and you can use a B-tree index to help with testing for equality, but there isn't currently a way to index particular keys and values in an HSTORE

Still, this is a useful data type for storing key/value data, especially if you aren't sure what keys you might need.

JSON and JSONB

Sometimes, you need to store complex, structured data, but it won't fit into the relational model. Either it's hierarchical or doesn't have a small defined set of attributes. If you were using a normal SQL database, you would not be able to store this data in a useful way. You'd likely need to use a Document-oriented database[4] like CouchDB or MongoDB. Postgres, however, *can* store such data types using the JSON type className.

Although you can store JSON in a TEXT field (and even configure Rails to parse it for you), you can't really query a TEXT field in a useful way (for example, querying for rows that have a certain value for a JSON key). This is why many turn to document-oriented databases, because they can delve into the structure of the JSON document for querying.

Let's suppose we want to allow our data science team to store arbitrary insights about our customers in the database. They don't know what format they'll need, so we can just give them a JSON column. Active Record supports this via the :json type, so we can create the column as we normally would other columns.

grab-bag/one/shine/db/migrate/20150820233327_add_insights_to_customers.rb

```
class AddInsightsToCustomers < ActiveRecord::Migration
  def change
    add_column :customers, :insights, :json, default: {}
  end
end
```

You can interact with this field exactly as you'd expect:

```
> rails c
:001 > c = Customer.first
  Customer Load (2.3ms)  SELECT  "customers".* FROM "customers"  ORDER BY "custo…
 => #<Customer id: 1, first_name: "Toby", last_name: "McKenzie", email: "avery0@…
:002 > c.insights[:spendiness] = 4.5
 => 4.5
:003 > c.insights[:curiosity] = { shoes: 3, hats: 99, accessories: true }
 => {:shoes=>3, :hats=>99, :accessories=>true}
:004 > c.save!
  SQL (7.7ms)  UPDATE "customers" SET "insights" = $1, "updated_at" = $2 WHERE "…
 => true
:005 > c.insights
 => {:spendiness=>4.5, :curiosity=>{:shoes=>3, :hats=>99, :accessories=>true}}
```

4. https://en.wikipedia.org/wiki/Document-oriented_database

```
:006 > c.insights[:curiosity][:hats]
 => 99
```

JSON columsn support the -> operator (just like an HSTORE does), but this operator returns values that are also JSON types. This makes it hard to query for specific values, and requires complex casting. To deal with this, the ->> operator is also available, and this produces a TEXT value, which is much easier to deal with:

```
> select id,insights from customers where (insights->>'spendiness')::decimal > 4;
-[ RECORD 1 ]-----------------------------------------------------------------------
id         | 1
insights   | {"spendiness":4.5,"curiosity":{"shoes":3,"hats":99,"accessories":true}}

> select id,insights from customers
  where (insights->'curiosity'->>'shoes')::decimal > 2;
-[ RECORD 1 ]-----------------------------------------------------------------------
id         | 1
insights   | {"spendiness":4.5,"curiosity":{"shoes":3,"hats":99,"accessories":true}}
```

You can also query for equality, at a deeply nested level. Suppose we want to find all customers who are curious about accessories. We can use the @> operator which checks for containment.

```
> select id,insights from customers
where insights@>'{ "curiosity": { "accessories": true  }}';
-[ RECORD 1 ]-----------------------------------------------------------------------
id         | 1
insights   | {"spendiness":4.5,"curiosity":{"shoes":3,"hats":99,"accessories":true}}
```

This returns all rows where the insights contains a key curiosity which contains an object which contains a key accessories which has a value of true.

This is obviously pretty handy. But, it gets better. There is an alternate JSON type called JSONB that allows you to *index* the JSON. You can create a field like this in Active Record using :jsonb instead of :json. Doing that, we can now create a GIN index that will assist in some queries.

```
> EXPLAIN ANALYZE
  SELECT * FROM customers
  WHERE insights@>'{ "curiosity": { "accessories": true  }}';
                     QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..9586.00 rows=350 width=111) (actua
   Filter: (insights @> '{"curiosity": {"accessories": true}}'::jsonb)
   Rows Removed by Filter: 349999
 Planning time: 0.116 ms
 Execution time: 77.494 ms
(5 rows)
```

```
> create index on customers using GIN (insights);
CREATE INDEX

> EXPLAIN ANALYZE
  SELECT * FROM customers
  WHERE insights@>'{ "curiosity": { "accessories": true  }}';
                     QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on customers  (cost=74.71..1175.69 rows=350 width=11
   Recheck Cond: (insights @> '{"curiosity": {"accessories": true}}'::
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on customers_insights_idx  (cost=0.00..74.62
         Index Cond: (insights @> '{"curiosity": {"accessories": true}
 Planning time: 0.169 ms
 Execution time: 0.043 ms
(7 rows)
```

It's hard to overstate how powerful this is. This gives you the flexibility of a document-oriented database, but the performance of a relational database's queries. It also means that you can write features that store both relational and free-form data that take advantage of transactional integrity, which would be impossible if you were using multiple data stores. This means that you can use Postgres for a wide variety of data-storage applications.

It should be noted that while Active Record supports this type, you will still need to write SQL using these operators to do the advanced querying. For example, to perform the query we looked at above in Rails, you would need to do this:

```
Customer.where("insights @> ?", { curiosity: { accessories: true }}.to_json)
```

Since ery few databases support features like this, ActiveRecord has no native API for this. Fortunately, where is flexible enough for us to make it work.

Enums

Rails 4 added support for enums, which is short for *enumerated type*[5], which is essentially a field that has a small number of possible values. A common use for this is for status codes. For example, we might give our customers a status of "signed_up", "verified", or "inactive".

Before Rails' support for enums, you would need to use a string field in the database, and then something like validates_inclusion_of to make sure only the allowed types were being used. With Rails 4's enum support[6], you can now do this more explicitly:

5. https://en.wikipedia.org/wiki/Enumerated_type
6. http://api.rubyonrails.org/classes/ActiveRecord/Enum.html

```
class Customer < ActiveRecord::Base
  enum status: [ :signed_up, :verified, :inactive ]
end

c = Customer.first
c.status
# => signed_up
c.signed_up?
# => true
c.status = :foo
ArgumentError: "foo" is not a valid status
```

The problem with this is that, by default, Rails stores this in the database as a number corresponding to the index where the value falls in this array. Not only is this brittle, but it makes the data impossible to interpret without the Ruby code.

We can tell Rails to use strings instead:

```
class Customer < ActiveRecord::Base
  enum status: {
    signed_up: "signed_up",
     verified: "verified",
     inactive: "inactive",
  }
end
```

It's a bit repetitive, but it makes the data easier to understand. But, it only enforces valid values from within Rails. To enforce valid values at the database, we could use a check constraint:

```
ALTER TABLE
  customers
ADD CONSTRAINT
  allowed_statuses
CHECK
  (status in ('signed_up', 'verified', 'inactive'))
```

Postgres also has support for enumerated types[7], eliminating the need for check constraints like this. We can create a custom type named customer_status and Postgres will handle everything at the database layer, and it's all compatible with Rails

grab-bag/one/shine/db/migrate/20150821123732_add_status_to_customer.rb

```
class AddStatusToCustomer < ActiveRecord::Migration
  def up
    execute %{
      CREATE TYPE
```

7. http://www.postgresql.org/docs/9.4/static/datatype-enum.html

```
        customer_status
      AS ENUM
        ('signed_up', 'verified', 'inactive' )
    }
    add_column :customers, :status, "customer_status",
               default: "signed_up",
               null: false
  end
  def down
    remove_column :customers, :status
    execute %{
      DROP TYPE customer_status
    }
  end
end
```

The only downside is having to repeat the values in our Active Record model, but this is a small price to pay for an explicitly-modeled field in our code and database.

Postgres also has rich support for range types, various date and time types, geometric types and even IP addresses. The documentation[8] should provide inspiration for what you can store, and how you can index it, in your database.

Outside of advanced data types, we can also use Postgres a full-text search engine.

Searching free-form text

In Chapter 4, *Use Fast Queries with Advanced Postgres Indexes*, on page 49 we used LIKE to do a fuzzy search of a text field. Postgres actually supports a full-blown full-text search engine[9] that you can use for searching large swaths of text.

To do this, Postgres has two data types: tsvector, which represents a searchable *document* and tsquery, which represents a query of some document. To perform a full-text search, you use the @@ operator. The left-hand side should be a tsvector (you can turn any string type into one via the to_tsvector function) and the right-hand side is a tsquery (similarly, you can use to_tsquery to turn a string into a tsquery).

Here's an example that searches the given string to see if it contains both "perform" and "search".

```
> SELECT to_tsvector('Postgres can perform a full-text search') @@
```

8. http://www.postgresql.org/docs/9.4/static/datatype.html
9. http://www.postgresql.org/docs/9.4/static/textsearch-intro.html

```
        to_tsquery('perform & search');
 ?column?
----------
 t
(1 row)
```

You can also give to_tsquery a config name, which can help it better match the text to a query. A config is often a language, so we can use the config english and Postgres will know that if we search for performs instead of perform that the string still matches, since both words are the same *normalized lexeme*[10].

```
> SELECT to_tsvector('english','Postgres can perform a full-text search') @@
        to_tsquery('performs & search');
 ?column?
----------
 t
(1 row)
```

Suppose we want to allow customers to write an open-ended bio for themselves and then allow other customers to search those bios to find like-minded shoppers. We can add a new TEXT field to CUSTOMERS and create a special index on it.

The GIN index we talked about above is mostly intended for full-text search (it just happens to be useful for arrays and JSON). Let's see what it looks like.

grab-bag/one/shine/db/migrate/20150822194126_add_bio_to_customers.rb

```
class AddBioToCustomers < ActiveRecord::Migration
  def up
    add_column :customers, :bio, :text
    execute %{
      CREATE INDEX
        customers_bio_index ON customers
      USING
        gin(to_tsvector('english', bio));
    }
  end
  def down
    remove_column :customers, :bio
  end
end
```

With this in place, our full-text, fuzzy search of CUSTOMERS by the bio field will be indexed.

```
> explain analyze
  select * from customers
  where to_tsvector('english',bio) @@ to_tsquery('widgets');
```

10. https://en.wikipedia.org/wiki/Lexeme

```
                                  QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on customers  (cost=209.57..4015.82 rows=1750 width=…
   Recheck Cond: (to_tsvector('english'::regconfig, bio) @@ to_tsquery…
   -> Bitmap Index Scan on customers_bio_index  (cost=0.00..209.13 ro…
         Index Cond: (to_tsvector('english'::regconfig, bio) @@ to_tsq…
 Planning time: 0.131 ms
 Execution time: 0.036 ms
```

Note that if you omit the config from the call to to_tsvector, Postgres will not be able to use the index, since that's the config we used when setting it up.

```
> explain analyze
  select * from customers
  where to_tsvector(bio) @@ to_tsquery('widgets');
                                    QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..12033.00 rows=1750 width=147) (actu…
   Filter: (to_tsvector(bio) @@ to_tsquery('widgets'::text))
   Rows Removed by Filter: 350000
 Planning time: 0.136 ms
 Execution time: 529.359 ms
```

We can see here that it did a seq scan instead of using the index.

Full-text search in Postgres is extremely powerful. You can do much more than what we've just seen. You can create stop words that are not searched/indexed. You can do full text search on concatenated fields. You can create synonyms so that, for example, a search for "database" could find strings containing "data store". You can also control how search results are ranked. All within your database!

Exporting Data to the Outside World as CSV

It's often handy to query the database and produce a comma-separated values (CSV) version of the results. This can be useful for sharing the results with non-technical members of your team, or for pulling into a system that doesn't support SQL. Postgres makes it very simple to do this via the COPY[11] command.

Suppose we wanted to get the names, emails, and ids of all of our customers. We could do something like this:

```
> COPY (
    SELECT
      id, first_name, last_name, email
    FROM
      customers
```

11. http://www.postgresql.org/docs/9.4/static/sql-copy.html

```
) TO '/tmp/customers.csv' WITH CSV HEADER;
```

This will run the given query, format it as CSV, and save it to /tmp/customers.csv *on the server*. This isn't always convenient, but you can use \COPY to save it to the client. If we run the same command, but using \COPY (note the backslash), it will save the CSV locally:

```
> \COPY (
    SELECT
      id, first_name, last_name, email
    FROM
      customers
) TO '/tmp/customers.csv' WITH CSV HEADER;
```

Then /tmp/customers.csv on *our computer* will have the CSV.

This completes our whirlwind tour of some of Postgres other useful features. The documentation[12]contains more gems like these, and is quite readable, so be sure to check it out.

Leveling-up with Angular

Angular is a deep technology, and what you've mostly seen here is how to use it for tasks you'd normally use JQuery for, and how it makes that code simpler and cleaner. As you create more advanced user interfaces, you'll benefit from additional features Angular provides for organizing your code.

Creating re-usable markup with Directives

Let's suppose we wanted to add a header to our customer details page that looked like the search results component we created in Chapter 5, *Create Clean Search Results with Bootstrap Components*, on page 69. In Rails, we could accomplish this via a partial. In Angular, we can create our own directive that can act as a partial.

Angular directives are *highly* complex. They aren't just for sharing markup, but this is the simplest use-case that's useful in practice and demonstrates the concept. Let's see how it works.

First, we'll extract the markup to a new template customer_summary.html. Because it uses floats, we'll wrap the entire thing in a div that has the clearfix class on it.

```
<div class="clearfix">

  <!-- original markup -->
```

12. http://www.postgresql.org/docs/9.4/static/index.html

```
</div>
```

Our directive will be called *customerSummary* and we'll use it like so (note that our markup uses the dasherized version, cutomer-summary):

```
<customer-summary cust="customer" view-details-function="viewDetails">
</customer-summary>
```

You'll see that we have to pass in the two dynamic parts of the template, which is the customer object itself and the function to be called when "View Details" is clicked. We do this so that our component won't be tightly coupled to the code where it's used.

To make it work, we'll use the directive function on app that takes the name of our directive and a function. That function is expected to return an object that describes our directive.

Angular recognizes *many* attributes on the object our function returns, as a directive is highly flexible. In our case, we only need two attributes: scope and templateUrl. templateUrl is the name of our template file, and scope is the object containing what we want exposed to that template.

As a simple example, suppose our template just looked like this:

```
Hello {{some_name}}!
```

If our scope contained { "some_name": "Bob" }, then our template would work as expected and render Hello Bob!. The scope object can recognize special values to make things simpler. If the value is =, that tells Angular to grab the value from the attributes declared on the directive.

Let's see the code, as this will make it easier to understand.

grab-bag/one/shine/app/assets/javascripts/customers_app.js

```
app.directive("customerSummary", function() {
  return {
    "scope": {
      "cust": "=",
      "viewDetailsFunction": "="
    },
    "templateUrl": "customer_summary.html"
  }
});
```

Recall that we are passing the attributes cust and view-details-function to our directive. By indicating that the scope values for both cust and viewDetailsFunction are =, Angular will grab the values from what we specified.

In other words, cust will be the same as $scope.customer and viewDetailsFunction will be the same as $scope.viewDetails. This means that if we use our new customer-summary directive elsewhere, we can have control over the values we pass into the re-usable directive.

If we change our results code to use it:

grab-bag/one/shine/app/assets/javascripts/templates/customer_search.html

```
<ol class="list-group">
  <li class="list-group-item"
      ng-repeat="customer in customers">
    <customer-summary
      cust="customer"
      view-details-function="viewDetails">
    </customer-summary>
  </li>
</ol>
```

And try our search, it should look and work just the same as it did before.

Now, we can add it to our customer details page to create the header we want:

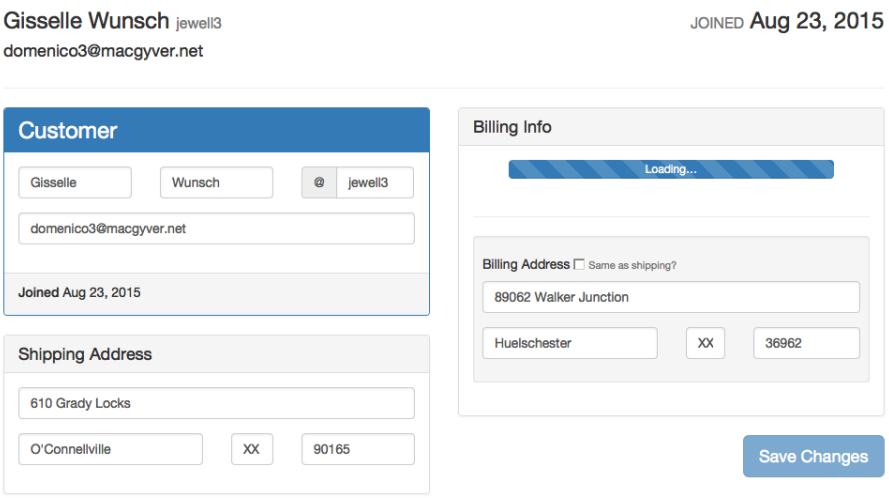
grab-bag/one/shine/app/assets/javascripts/templates/customer_detail.html

```
<header>
  <customer-summary cust="customer"></customer-summary>
  <hr>
</header>
<form novalidate name="form"><div class="row">

    <!-- rest of the markup -->

</div></form>
```

You'll note that we've omitted view-details-function. For this use-case, it's really not needed, so we'd like it not to show up if not specified. We can do that by using ng-if in customer_summary.html

```
<div class="pull-right" ng-if="viewDetailsFunction">
  <button class="btn btn-small btn-primary"
          ng-click="viewDetailsFunction(cust)">
    View Details…
  </button>
</div>
```

Now, our detail page has a header that uses the same markup as the results page!



This is the most bare-bones means of creating re-usable components using Angular directives. There is a lot more you can do with it to keep your code clean and reduce duplication. We'll see some open-source directives in the next chapter that are designed to work with Bootstrap. They are a great source of inspiration in terms of what you can do with directives.

Format View Content Using Filters

In Chapter 6, *Build a Dynamic UI with AngularJS*, on page 81, we used the `date` filter to format our timestamp as a human-readable date. As a reminder, filters are Angular's equivalent of Rails view helpers, but work more like UNIX pipes:

```
<span class="date">{{ customer.joined_at | date }}</span>
```

You can create filters yourself using the `filter` function available on Angular modules. Suppose that the names in our `CUSTOMERS` table aren't well normalized, and are a mix of all lower-case, all upper-case, and mixed case. We want to display those names properly capitalized, but *only* if the name is either all lower-case or all upper-case (so as to not mess up names like McAvoy or O'Drudy).

We'll create a filter called "name". We can do this by calling `filter` on `app` in `customers_app.js`.

grab-bag/one/shine/app/assets/javascripts/customers_app.js

```
app.filter("name", function() {
  return function(input) {
    if (!input) {
      return input;
```

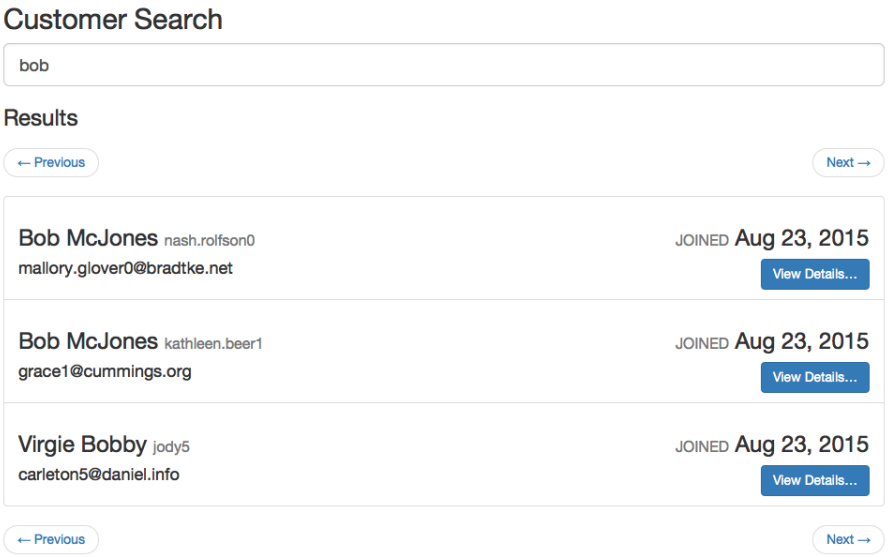
```
    }

    if ( (input.toLowerCase() === input) ||
         (input.toUpperCase() === input) ) {

      return input.charAt(0).toUpperCase() +
             input.slice(1).toLowerCase();
    }
    else {
      return input;
    }
  }
});
```

We can then use it like we did our date filter:

grab-bag/one/shine/app/assets/javascripts/templates/customer_summary.html

```
<h2 class="h3">
  {{ cust.first_name | name }} {{ cust.last_name | name }}
  <small>{{ cust.username }}</small>
</h2>
```

Now, when we search, we can see that user's names are formatted properly.



Filters are a great way to create view logic for formatting that you would normally use Rails helpers for.

Organize Code in Different Files

Up to this point, all of our Angular code has gone into customers_app.js. Angular has no opinions on what files code lives in, and there's no set standard for how to organize it. That said, having all code in one big file isn't recommended in *any* software system.

To make this happen in Rails requires two things. First, we have to move our code into different files, but, more importantly, we have to make sure the asset pipeline picks them in the right way.

As to what code goes in what files, you can essentially do whatever you want. I tend to follow the pattern Rails uses for the middleware code, and make a directory for each type of component. Therefore, I'll have, in app/assets/javascripts, controllers, directives, and filters. For Shine, that would mean we'd have controllers/customer_search_controller.js, controllers/customer_detail_controller.js, controllers/customer_credit_card_controller.js, filters/name_filter.js, and directives/customer_summary_directive.js. This would leave customers_app.js as just declaring our module dependencies and setting up the routes.

To make them work with the asset pipeline, we have to do two things. First, because Rails wraps all code in JavaScript inside an immediately invoked function[13], the app variable isn't available outside the file where it's declared. Think of our code in customers_app.js as actually being like this:

```
function() {
  var app = angular.module(
    'customers',
    [
      'ngRoute',
      'ngResource',
      'ngMessages',
      'templates'
    ]
  );
})();
```

This means that app is private to this file. So, in each file that contains our angular code, we have to look up our Angular app via angular.module. Here's how it looks for CustomerCreditCardInfoController's source:

grab-bag/two/shine/app/assets/javascripts/controllers/customer_credit_card_controller.js

```
var app = angular.module('customers');
app.controller("CustomerCreditCardController", [
          "$scope","$resource",
```

13. https://en.wikipedia.org/wiki/Immediately-invoked_function_expression

```
  function($scope , $resource) {
    var CreditCardInfo = $resource('/fake_billing.json')
    $scope.setCardholderId = function(cardholderId) {
      $scope.creditCard = CreditCardInfo.get(
        { "cardholder_id": cardholderId}
      )
    }
  }
]);
```

You'll note that we're using module with only one argument. Invoked this way, Angular will generate an error if the module we want doesn't exist (when using the two-arg version, as we do in customers_app.js, Angular will *create* the module if it doesn't exist, which we don't want to do in our controller files). This means that we must ensure the code in customers_app.js runs first. We can do *that* by changing the asset pipeline's configuration in application.js.

Our current version has, as its last line, //= require_tree .. The require_tree directive to the asset pipeline will tell it to recursively bring in all the JavaScript it finds. The documentation[14] says that the order in which files are required is unspecified and that we cannot rely on it.

So, we just need to explicitly require customers_app.js first, before the require_tree directive:

grab-bag/two/shine/app/assets/javascripts/application.js

```
//= require jquery
//= require jquery_ujs
//= require angular
//= require angular-route
//= require angular-resource
//= require angular-messages
//= require angular-rails-templates
➤ //= require customers_app
➤ //= require_tree ./templates
➤ //= require_tree .
```

Our app should still work the same, and all our tests will still pass (because teaspoon uses the asset pipeline to bring in files for testing).

Extract Re-usable Code into Services

In addition to breaking our code down into separate files, it might also be nice to extract logic out of our controllers. In Ruby and Rails we can easily extract classes to accomplish this. In Angular, we can do this by creating

14. http://guides.rubyonrails.org/asset_pipeline.html#manifest-files-and-directives

services. A service is a function defined like we've been defining controllers, but it has no special purpose other than to hold code we want to organize.

What makes an Angular service different from a regular JavaScript object is that we can arrange for them to have dependencies injected via Angular's dependency injection. For example, we could extract all the search logic out of CustomerSearchController into a service called customerSearch. We can then inject customerSearch into CustomerSearchController like so:

grab-bag/two/shine/app/assets/javascripts/controllers/customer_search_controller.js

```
var app = angular.module('customers');
app.controller("CustomerSearchController", [
          '$scope','$location', "customerSearch",
  function($scope , $location ,  customerSearch) {

    $scope.customers = [];
    customerSearch.successCallback(function(customers) {
      $scope.customers = customers;
    });

    $scope.search       = customerSearch.search
    $scope.previousPage = customerSearch.previousPage;
    $scope.nextPage     = customerSearch.nextPage;

    $scope.viewDetails = function(customer) {
      $location.path("/" + customer.id);
    }
  }
]);
```

We are assuming that customerSearch works as follows: we configure a success callback that will be given data based on a search term, and that it has three functions available: search, previousPage, and nextPage. search handles searching for customers and giving the results to our success callback declared with successCallback. Both previousPage and nextPage will paginate as before, but they will remember the search term last used, so we don't have to pass it in again.

Note how clean our controller is. All it's doing is surfacing data and functions to the view, but all actual logic has been extracted to the service. This is the sort of thing we strive for in our Rails controllers, so it's nice to be able to do it with our Angular controllers, too.

To set up a service to hold our search logic, we declare a function that will be given all the needed dependencies and is expected to return an object. That object is what is injected into CustomerSearchController as the object customerSearch. The basic outline looks like so:

grab-bag/two/shine/app/assets/javascripts/services/customer_search.js
```
var app = angular.module('customers');
app.factory("customerSearch", [
          "$http",
  function($http) {

    // ...

    return {
      "successCallback": successCallback,
      "search": search,
      "previousPage": previousPage,
      "nextPage": nextPage
    };
}]);
```

The returned object is the public interface to our service. All we have to do is implement these functions. First, let's declare some private variables to store the current page, most recent search term, and success callback:

grab-bag/two/shine/app/assets/javascripts/services/customer_search.js
```
var page = 0;
var mostRecentSearchTerm = undefined;
var success = function() {};
```

The definition of successCallback is straightforward:

grab-bag/two/shine/app/assets/javascripts/services/customer_search.js
```
var successCallback = function(newCallback) {
  success = newCallback;
};
```

The implementations of nextPage and previousPage look more or less like their old counterparts, however they use the mostRecentSearchTerm:

grab-bag/two/shine/app/assets/javascripts/services/customer_search.js
```
var previousPage = function() {
  if ( (page > 0) && mostRecentSearchTerm) {
    page = page - 1;
    search(mostRecentSearchTerm);
  }
};

var nextPage = function() {
  if (mostRecentSearchTerm) {
    page = page + 1;
    search(mostRecentSearchTerm);
  }
};
```

This leaves search, which looks similar to what it did before, however it defers to success:

grab-bag/two/shine/app/assets/javascripts/services/customer_search.js

```
var search = function(searchTerm) {
  if (searchTerm.length < 3) {
    return;
  }
  mostRecentSearchTerm = searchTerm;
  $http.get("/customers.json",
            { "params": { "keywords": searchTerm, "page": page } }
  ).success(success).error(
    function(data,status,headers,config) {
      alert("There was a problem: " + status);
    }
  );
};
```

Notice how the code in both places is fairly cohesive, and focuses on just one part of the whole. Also notice how our controller no longer depends on $http, because it's only needed to do the customer search.

At it's core, services in Angular are just how you decompose code into smaller bits. We can write tests for services, the same as we have for our controllers. This allows us to manage complexity in our front-end in the same way we would in the middleware.

As we've mentioned many times, Angular is a rich, deep, flexible, complex framework. It's popularity means that there are many add-ons and extensions to help you with common tasks. This popularity also means that Angular is highly *googleable*—you can find answers to common problems easily.

Getting Everything Out of Bootstrap

Unlike Angular and Postgres, Bootstrap is smaller and more focused in its scope. It's documentation is actually a great place for inspiration on how to solve common layout and design problems. One handy part of Bootstrap that we *can't* easily use is the section titled "JavaScript". This section contains more interactive components like modal dialogs and tooltips. These are invaluable tools for creating rich client-side applications, but because they are based on JQuery, it's difficult to use them with an Angular app. Fortunately, the Angular UI[15] project has re-implemented many of them as Angular components.

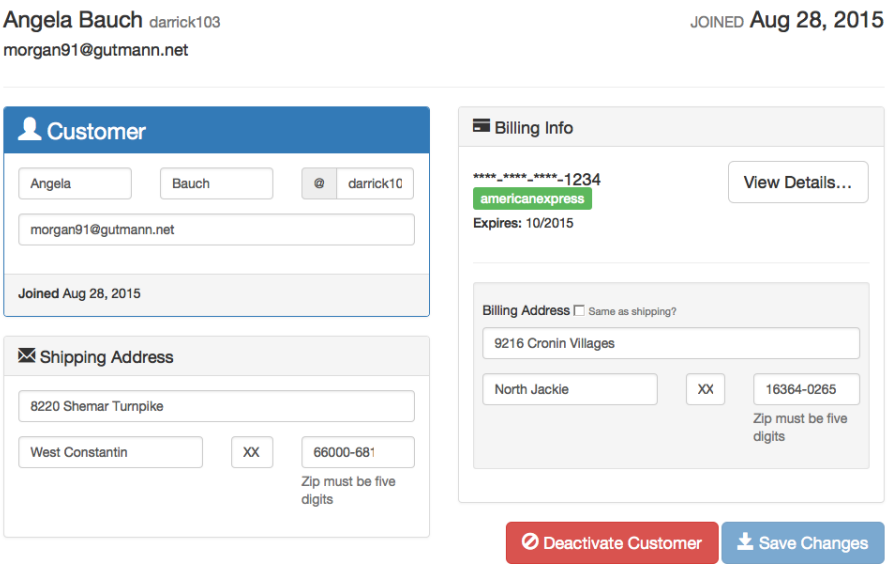
15. https://angular-ui.github.io/

But first, let's learn about another handy thing that comes with Bootstrap: icons.

Glyphicons

Bootstrap includes some of the icons that are part of the Glyphicons[16] icon font. This allows you to add icons to your UI by just applying CSS classes to empty elements. Bootstrap's documentation[17] lists the icons that are included.

Our customer detail page is complex, and we've used panels to create separation between the elements, but they could be made easier to visually navigate by adding icons to the panel titles. So, let's put a person icon next to the "Customer" title, an envelope icon next to "Shipping Address", a credit card icon next "Billing Info", and a save icon on the "Save Changes" button. It'll look like so:



To do this, we just create empty elements with the `glyphicon` class, plus a class for the icon we want.

```
<button ng-click="save()"
        class="btn btn-lg btn-primary"
        ng-disabled="form.$invalid || form.$pristine">
  <i class="glyphicon glyphicon-save" aria-hidden="true"></i>
  Save Changes
</button>
```

16. http://glyphicons.com/
17. http://getbootstrap.com/components/#glyphicons

Since the icons are purely for decoration, we use the i tag as well as aria-hidden="true" so that screen readers don't get confused and read this markup to users. The aria- tags are part of the Web Accessibility Initiative[18].

If you reload the page, you should see the icons. One thing worth pointing out is that icons are a tricky subject in user experience design. Most research indicates that icons without text are extremely difficult for users to understand. We're using them here as a demonstration but, as with all design choices, be cognizant of the problem you are solving when adding icons. Purely aesthetic reasons can be valid sometimes, but they are not necessarily paramount in making a great experience for your users.

Given that, there's a technical problem with our implementation, and it's a nasty one: the icons won't work outside our development environment. The Rails asset pipeline is notoriously opaque when things don't work. Fonts, in particular, are difficult to get working. You can see the troubles if you try to run a test that hits this page. You should get an error like No route matches [GET] "/fonts/bootstrap/glyphicons-halflings-regular.ttf".

To fix this, we need to do three things: change how the fonts are configured in CSS, add Bootstrap's font path as an asset path, and configure the asset pipeline to precompile all fonts. It's unclear why fonts are treated differently than CSS or JavaScript in this regard, but this is what we have to do.

First, we have to rename our application.css file to application.scss because we're going to need to use SASS[19]'s @import directive to bring in an auxiliary file that ships with Bootstrap that will fix how the fonts are referred to in our CSS.

By default, Bootstrap uses the path ../fonts. Relative paths like this don't work reliably in production. Rails provides a SASS function named font-path that dynamically figures out the right URL to use for referencing fonts at runtime. We can tell Bootstrap to use this by including the file bootstrap-sprockets that comes with Bootstrap.

First, we have to rename application.css so that the asset pipeline will run our newly-named application.scss through the SASS interpreter.

```
> mv app/assets/stylesheets/application.css app/assets/stylesheets/application.scss
```

Now, we need to remove the use of = require and replace it with @import:

grab-bag/three/shine/app/assets/stylesheets/application.scss

```
// START_HIGHLIGHT
```

18. https://en.wikipedia.org/wiki/WAI-ARIA
19. http://sass-lang.com/

```
@import "bootstrap-sass-official/assets/stylesheets/bootstrap-sprockets";
@import "bootstrap-sass-official/assets/stylesheets/bootstrap";
// END_HIGHLIGHT
/*
 *= require_tree .
 *= require_self
 */
```

What's going on here is that bootstrap-sprockets adds configuration to tell Bootstrap to use the Rails asset pipeline helpers, instead of hard-coding ../fonts. This helper tells Rails to look in app/assets/fonts for the fonts. But, that's not where they are. They are in vendor/bower_components/bootstrap-sass-official/assets/fonts/bootstrap.

So, in config/application.rb we have to do a few things. We must add that path to our asset paths, and we must also specify that all fonts must be pre-compiled. For whatever reason, any font not in app/assets/fonts just won't be served up properly unless it's pre-compiled.

grab-bag/three/shine/config/application.rb

```
module Shine
  class Application < Rails::Application
    config.assets.paths << Rails.root.join("vendor","assets","bower_components")
➤   config.assets.paths << Rails.root.join("vendor",
➤                                          "assets",
➤                                          "bower_components",
➤                                          "bootstrap-sass-official",
➤                                          "assets",
➤                                          "fonts")
➤   config.assets.precompile << /\.(?:svg|eot|woff|ttf|woff2)\z/
  end
end
```

Now we can confidently use glyphicons in our application! Note that if you use other icon fonts like Font Awesome[20], you'll need to do a similar configuration.

Design for Mobile Devices with Ease

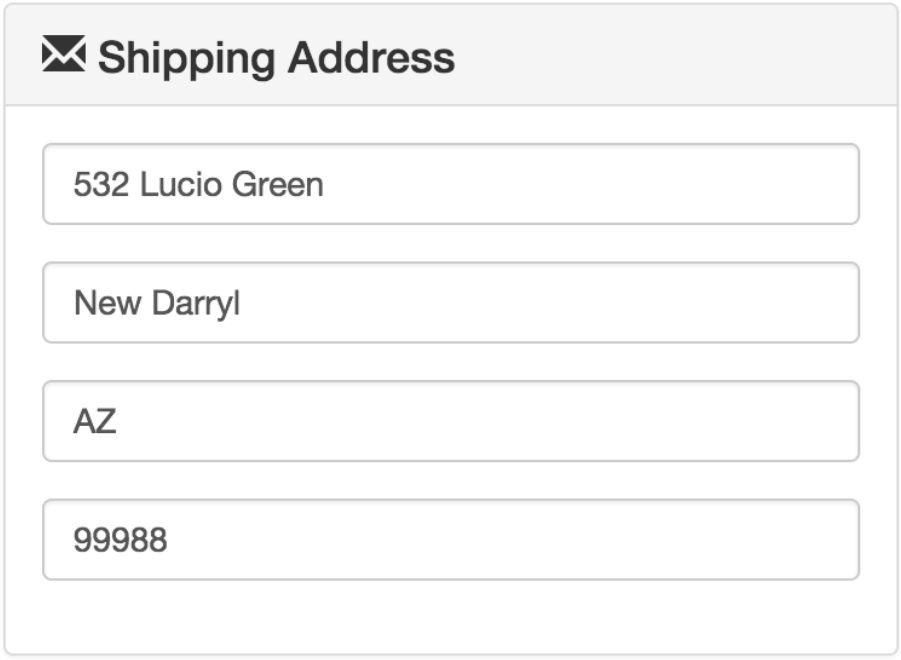
Responsive Web Design[21] is a technique for designing for many different screen sizes and devices. If you've been to a website on your mobile phone that looks great, but different from how it looks on your desktop computer, this is responsive design in action.

20. https://fortawesome.github.io/Font-Awesome/
21. https://en.wikipedia.org/wiki/Responsive_web_design

Bootstrap provides some basic tools to help with responsive design. If you recall from Chapter 9, *Design Great UIs With Bootstrap's Grid and Components*, on page 155, when we were learning about grid-based design, we used CSS classes that had `md` in them, such as `col-md-6`. That `md` means *this is the column size I want on medium-sized devices*. But, there are other options, as outlined in Bootstrap's grid options[22] documentation.

Let's suppose our complex customer detail view will be used on a mobile phone. We can quickly simulate this by resizing our browser to a narrow width. Because we used the `md` form of our grid, when the screen shrinks to below the medium-sized device width (992px), each grid cell takes up the full width, since this is the default.

Generally, this is fine, but for addresses, it's a bit extreme.



Suppose for mobile devices we are fine with city being 100% of the width, but we want state and zip code to be on the same line. With Bootstrap's responsive grid, making this happen is simple. In addition to the `col-md` classes we've used, we can add `col-xs` classes (`xs` for extra-small devices below 768px in width, i.e. most smartphones).

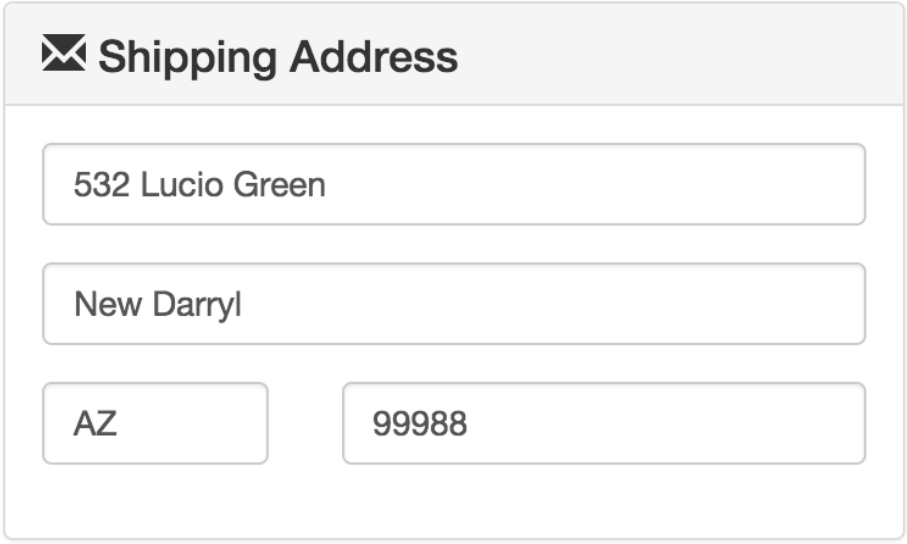
22. http://getbootstrap.com/css/#grid-options

For city, we'll use col-xs-12, meaning we want the city's grid to be full width (this is somewhat redundant, but it's good to be explicit about our design decisions, since we'll be using other xs classes). State will be given the col-xs-4 class, meaning it should take up four grid cells at the xs size, while zip gets the remaining 8 cells, which means we need the col-xs-8 class.

grab-bag/three/shine/app/assets/javascripts/templates/customer_detail.html

```
<div class="row">
  <div class="col-md-6 col-xs-12">

    <!-- city form markup -->

  </div>
  <div class="col-md-2 col-xs-4">

    <!-- state form markup -->

  </div>
  <div class="col-md-4 col-xs-8">

    <!-- zip form markup -->

  </div>
</div>
```

The result is as expected:



With just a few extra CSS classes, we've made drastically different layouts work on different screen sizes. You can even resize your browser to a larger width and watch the screen change layouts back to our original design.

In addition to a responsive grid, Bootstrap also has helper classes[23] for use when doing responsive design. For example, we can use a class like hidden-xs to hide elements on small screens. This could be useful for non-essential elements that can be omitted for mobile users (like icons :).

Responsive design is a deep topic, but Bootstrap gives you a few useful features to help do it more easily and without making radical changes to your markup.

Using Bootstrap-powered Components with Angular

Bootstrap has *components*, which are JavaScript-powered features such as modal dialogs, tab controls, etc. The problem is that they don't work well with an Angular app, since they rely on JQuery, which operates on the DOM differently than Angular and could create conflicts. Fortunately, the Angular UI project[24] includes most of these components in its Bootstrap sub-project[25].

These components are made to use Bootstrap's styles and templates, but use Angular instead of JQuery, so you can more easily integrate them into your Angular app. We'll look at two examples that will help you with the most common UI tasks. The first is alert messages, followed by modal dialogs, both of which will enhance Shine's ability to edit customer data.

First, we'll install it and get it all set up.

Install Angular UI Bootstrap

We've been through this a few times already, so it should be old hat by now. First we add asset 'angular-bootstrap' to Bowerfile and run rake bower:install. Then, we'll add //= require 'angular-bootstrap' to app/assets/javascripts/application.js. Finally, we'll add "ui.bootstrap to the list of dependent modules in app/assets/javascripts/customers_app.js:

grab-bag/three/shine/app/assets/javascripts/customers_app.js

```
var app = angular.module(
  'customers',
  [
    'ngRoute',
    'ngResource',
    'ngMessages',
```

23. http://getbootstrap.com/css/#responsive-utilities
24. https://angular-ui.github.io/
25. http://angular-ui.github.io/bootstrap/

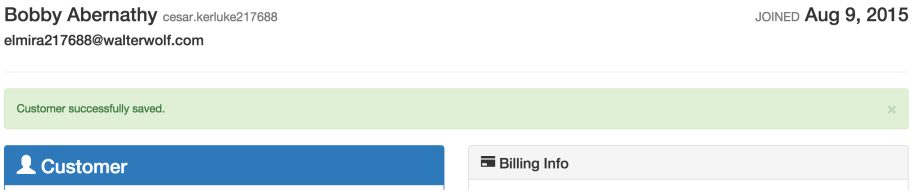
```
    'ui.bootstrap',
    'templates'
  ]
);
```

To know what value to use in the modules list, we have to look at the documentation[26] for Angular UI Bootstrap (flexibility comes at a price).

With that in place, let's add a dismissable alert.

Alerts

Bootstrap has support for creating static alert components[27], so we could hack up a basic alerting system with what we know right now. Bootstrap *also* includes a more dynamic alert component[28] that can be dismissed. We're going to use the Angular UI equivalent to create an alert after we successfully save a customer's information.



Angular UI Bootstrap includes the `alert` directive, which we can use as if it were an HTML element. We can use the attribute `type` to indicate what sort of alert it is (success vs. danger), and `close` to provide a function we'll implement that removes the alert when the user hits the close button.

grab-bag/three/shine/app/assets/javascripts/templates/customer_detail.html

```
<header>
  <customer-summary cust="customer"></customer-summary>
  <hr>
</header>
<section>
  <alert ng-show="alert"
         type="{{alert.type}}"
         close="closeAlert()">
    {{alert.message}}
  </alert>
</section>
<form novalidate name="form"><div class="row">

    <!-- rest of the markup -->
```

26. http://angular-ui.github.io/bootstrap/#/getting_started
27. http://getbootstrap.com/components/#alerts
28. http://getbootstrap.com/javascript/#alerts

To make it work, we just need to set $scope.alert to the alert we want to show the user, and implement closeAlert. Previously, our save function used the alert function to show the user the results of the save. We'll replace that and instead populate $scope.alert:

grab-bag/three/shine/app/assets/javascripts/controllers/customer_detail_controller.js

```
$scope.save = function() {
  if ($scope.form.$valid) {
    $scope.customer.$save(
      function() {
        $scope.form.$setPristine();
        $scope.form.$setUntouched();
        $scope.alert = {
          type: "success",
          message: "Customer successfully saved."
        };
      },
      function(data) {
        $scope.alert = {
          type: "danger",
          message: "Customer couldn't be saved"
        };
      }
    );
  }
}
```

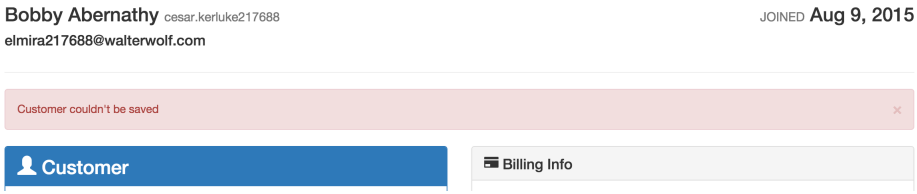
Because $scope.alert is bound to the alert component provided by Angular UI Bootstrap, just setting the value will cause it to show up to the user. We'll also need to clear $scope.alert inside closeAlert:

grab-bag/three/shine/app/assets/javascripts/controllers/customer_detail_controller.js

```
$scope.closeAlert = function(index) {
  $scope.alert = undefined;
}
```

Now, when we save the customer details, we see a pleasant message like we saw above telling us everything worked.

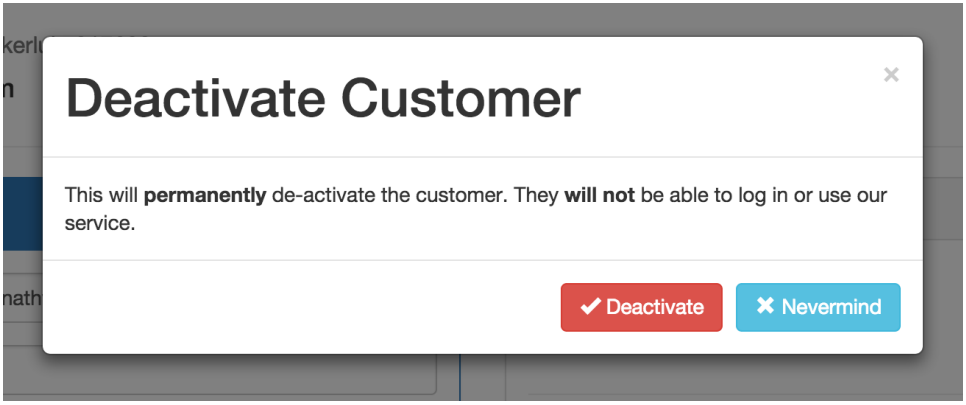
We can also see the error message by loading the page, making a change, stopping our server, and clicking "Save Changes". Since the server can't respond to our Angular app's AJAX request, the error callback will be called and we'll see an error message instead.



Modal Dialogs

Suppose we want to allow Shine's users to de-activate a customer (perhaps they are unhappy with the service and don't want to be a part of it any more). Since this is a destructive action, we'd want Shine's users (who you'll recall are CSRs) to confirm a customer deactivation. We could do this with the JavaScript `confirm` function to get a confirmation, but that control has some problems. First it's ugly. More importantly, it's inflexible. The buttons can only say "OK" and "Cancel", and we have limited options for formatting text.

Instead, let's create a modal dialog that fits with our design, and is easier for users to use.



Bootstrap provides the CSS to create this dialog, and Angular UI Bootstrap provides the code. We'll need to do a few things to make this work. First, we'll add a button to our view that will initiate the de-activation. Next, we'll inject `$modal` into `CustomerDetailController`, which is how we can launch and interact with the modal dialog. We can then use that to launch the modal inside our deactivation function. To make the modal itself work, we'll need a new view and controller for the modal.

Our deactivate button will live right next to our "Save Changes" button (notice we're using a Glyphicon as well as the `btn-danger` class to indicate that this action is more dangerous):

grab-bag/three/shine/app/assets/javascripts/templates/customer_detail.html

```
<div class="text-right">
  <button ng-click="deactivate()"
          class="btn btn-lg btn-danger">
    <i class="glyphicon glyphicon-ban-circle"></i>
    Deactivate Customer
  </button>
  <button ng-click="save()"
          class="btn btn-lg btn-primary"
          ng-disabled="form.$invalid || form.$pristine">
    <i class="glyphicon glyphicon-save"></i>
    Save Changes
  </button>
</div>
```

To implement the function we used for ng-click, deactivate, we'll need access to the $modal service provided by Angular UI Bootstrap. We inject that into our controller in the normal way:

grab-bag/three/shine/app/assets/javascripts/controllers/customer_detail_controller.js

```
var app = angular.module('customers');
app.controller("CustomerDetailController", [
          "$scope","$routeParams","$resource","$modal",
  function($scope , $routeParams , $resource , $modal) {
```

With this done, we can use the function open on $modal to open our modal dialog. This function returns a modal instance that is a promise. We'll call then on it with two parameters, both functions. The first is the function to run if the user confirmed the dialog. The second is the function to run if the user dismissed or canceled the dialog. In both cases, we'll just set an alert related to the results (actually building customer deactivation is out of scope for what we're learning here).

grab-bag/three/shine/app/assets/javascripts/controllers/customer_detail_controller.js

```
$scope.deactivate = function() {
  var modalInstance = $modal.open({
    templateUrl: 'confirm_deactivate.html',
    controller: 'ConfirmDeactivateController'
  });

  modalInstance.result.then(function () {
    $scope.alert = {
      type: "success",
      message: "Customer deactivated"
    }
  }, function (reason) {
    $scope.alert = {
      type: "warning",
      message: "Customer still active"
```

```
    }
  });
};
```

You'll note that open takes an object and we've defined two keys: templateUrl and controller. These indicate the view and controller for the modal.

For the view, we can use Bootstrap's modal markup[29] to create a modal specific to de-activation:

grab-bag/three/shine/app/assets/javascripts/templates/confirm_deactivate.html

```
<header class="modal-header">
  <button type="button" class="close" ng-click="nevermind()">
    <span aria-hidden="true">&times;</span>
  </button>
  <h1 class="modal-title">Deactivate Customer</h1>
</header>
<section class="modal-body">
  This will <strong>permanently</strong> de-activate the customer.
  They <strong>will not</strong> be able to log in or use our service.
</section>
<footer class="modal-footer">
  <button class="btn btn-danger" type="button" ng-click="deactivate()">
    <i class="glyphicon glyphicon-ok"></i>
    Deactivate
  </button>
  <button class="btn btn-info" type="button" ng-click="nevermind()">
    <i class="glyphicon glyphicon-remove"></i>
    Nevermind
  </button>
</footer>
```

You'll notice we have a close button in the header, as well as a button labeled "Nevermind" that both call the function nevermind when clicked. We also have a button labeled "Deactivate" that calls deactivate when clicked. These labels, along with the rich formatting available for the header and body text, are a big reason why we're going through all this to create a modal dialog instead of using JavaScript's confirm.

We can craft a message that helps the user understand what they are doing, and we can give the buttons semantically meaningful labels. Instead of the user having to mentally translate that clicking "OK" means "Deactivate", we can just put "Deactivate" right on the button. This makes the feature harder to misuse and easier to understand.

29. http://getbootstrap.com/javascript/#modals

With the view done, we just need to implement the controller for the modal. Since Angular UI Bootstrap is handling the mechanics of launching and hiding the modeal, as well as calling our callbacks, the modal's controller looks like a run-of-the-mill Angular controller. Which is great!

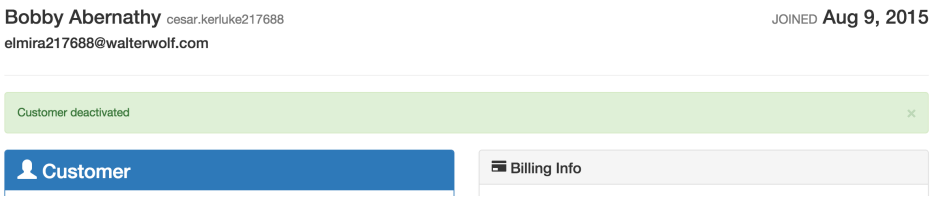
grab-bag/three/shine/app/assets/javascripts/controllers/confirm_deactivate_controller.js

```
var app = angular.module('customers');
app.controller("ConfirmDeactivateController", [
  "$scope","$modalInstance",
  function($scope , $modalInstance) {
    $scope.deactivate = function () {
      $modalInstance.close();
    };

    $scope.nevermind = function () {
      $modalInstance.dismiss('cancel');
    };
  }
]);
```

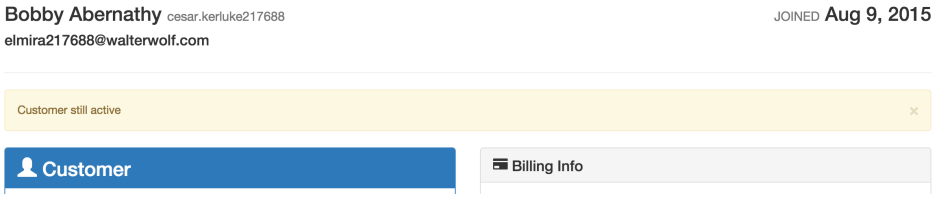
The only thing different is that we're injecting $modalInstance. This is the same value returned by $modal.open in CustomerDetailController. It exposes two functions, close and dismiss. When close is called, this resolves the promise, which will call the first function we passed to then. If we call dismiss, this rejects the promise, which will trigger the second function we passed to then.

With it all done, when we click the de-activate button, we get the nice-looking dialog we saw above.

If we click "Deactivate" we see a success message, using the alerting system we set up previously:



Clicking "Nevermind" shows us a warning message that the customer is still active:



These are just two of the many UI components available from Angular UI. As we've seen, they save us a lot of work in managing potentially complex UI interactions, but still afford us great flexibility in how we design for our users.

You can also create components like these yourself. The source code for Angular UI Bootstrap provides great examples of how to do this. Angular's directives documentation[30] is also a great place to start to get an overview.

This brings us to the end of our journey. Hopefully, you've learned not just how to use Angular, Postgres, and Bootstrap, but also the value of approaching each software problem with a holistic view of the tools you are using. Some problems are best solved with a better user experience, some with improved database performance, while some require bringing all parts of the stack together.

With the knowledge you've gained in this book, my hope is that it's given you the confidence to explore your toolset more deeply, and the curiosity to discover new and better ways to solve problems.

30. https://docs.angularjs.org/guide/directive

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/dcbang
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/dcbang*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764