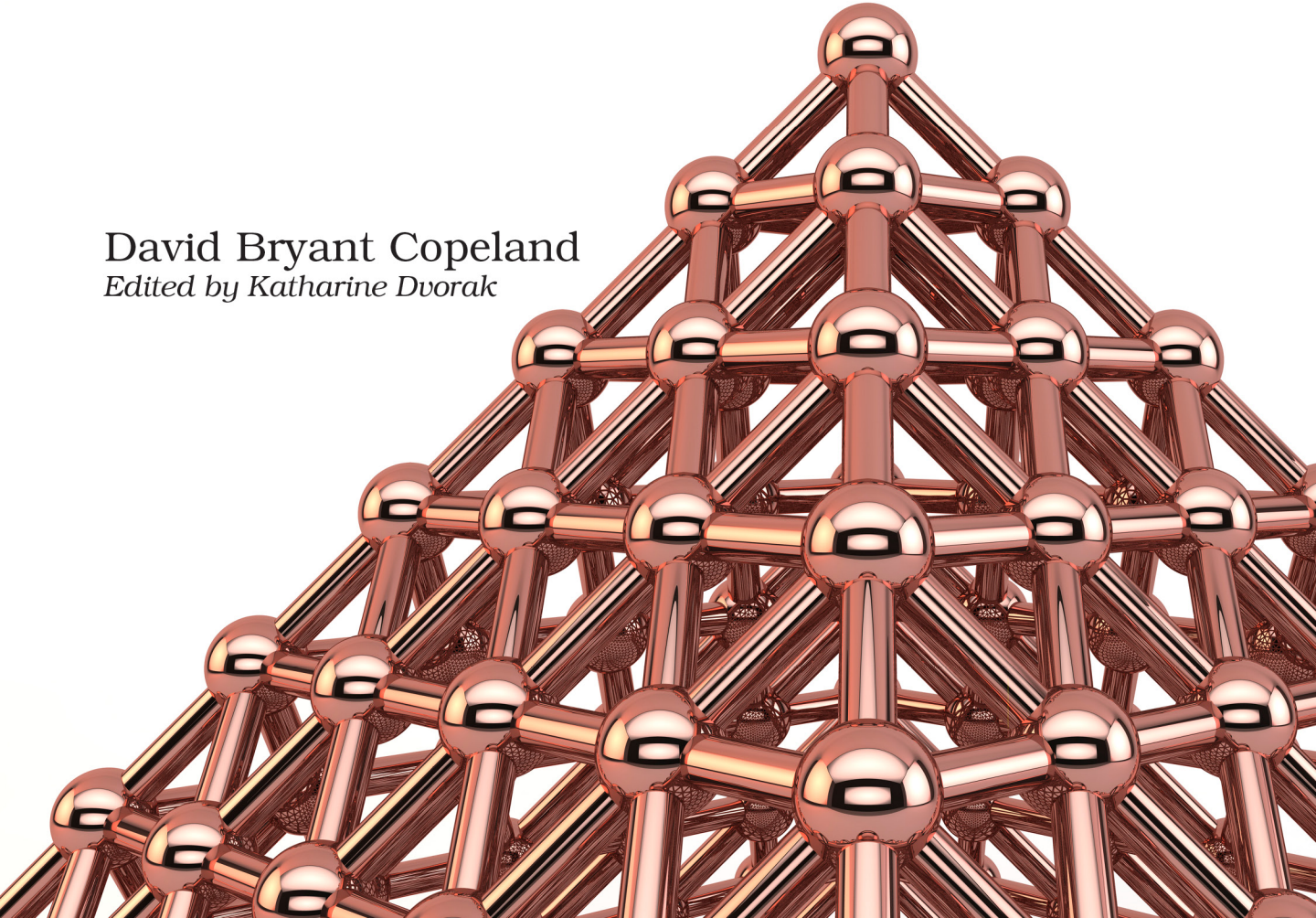
The Pragmatic Programmers

Rails, Angular, Postgres, and Bootstrap
Second Edition

Powerful, Effective, Efficient, Full-Stack Web Development



David Bryant Copeland
Edited by Katharine Dvorak

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/dcbang2/errata*, or by using the links at the bottom of each page.

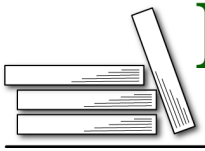
Thank you for being part of the Pragmatic community!

Andy

Rails, Angular, Postgres, and Bootstrap, Second Edition

David Bryant Copeland

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-220-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—October 19, 2016

Contents

Introduction

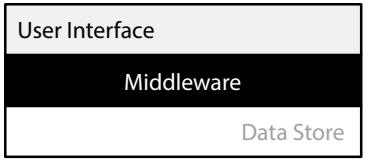
Think about what part of an application you're most comfortable working with. If you're a Rails developer, there's a good chance you prefer the back end, the Ruby code that powers the business logic of your application. What if you felt equally comfortable working with the database, such as tweaking 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.

The Rails framework encourages developers 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 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 ones used in this book) 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 (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 the organization's most important asset —its data. Even if the organization loses all of its source code, as long as it retains its data, it can still 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 mean 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 the 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 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.

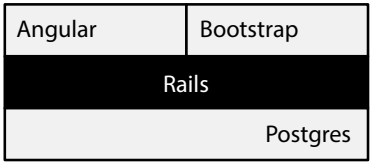
You'll learn *PostgreSQL*, *AngularJS*, and *Bootstrap*, but you can apply many of the lessons here to other data stores, JavaScript libraries, and CSS frameworks. In addition to seeing just how powerful these specific tools can be, you're going to be emboldened to think about writing software beyond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

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

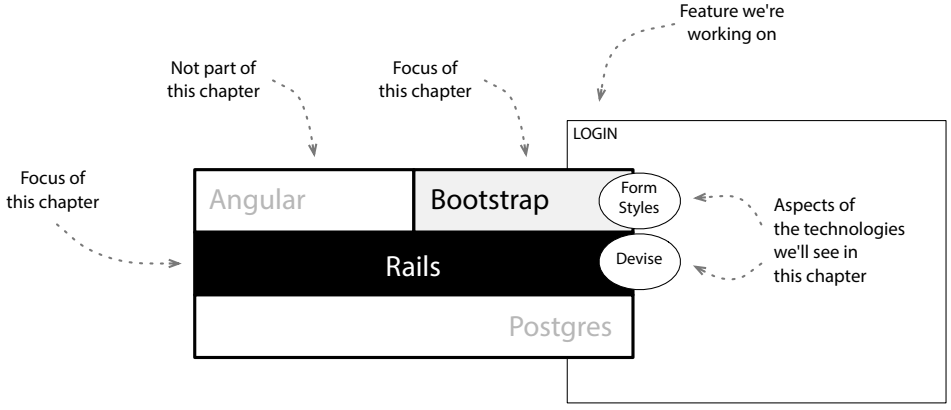
If all you've done with your database is create tables, insert data, and query it, you're going to be excited when you see what else you can do. Similarly, if all you've done with your web views is sprinkle some jQuery calls to your server-rendered HTML, you'll be amazed at what you can do with very 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 will look and feel so much better.

In this book, we focus on PostgreSQL (or simply *Postgres*) as our data store —the bottom of the stack—and AngularJS (or just *Angular*) with Bootstrap as our front end—the top of the stack. Each of these technologies is widely used and very powerful. You're likely to encounter them in the real world, and they each 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 this:



In each chapter, I highlight the parts of the stack you'll be focusing on and call out the various aspects of these technologies you'll be learning. Not every chapter focuses on all parts of the stack, so at the start of each chapter, you'll see a roadmap like this of what you'll be learning:



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

PostgreSQL

PostgreSQL[1] 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[2] or commercial databases such as Microsoft SQL Server.[3] Here are some of the features you'll learn about (and I'll show you 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, that the state in a U.S. address be written exactly as two uppercase 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 lowercased version of someone's name so that a case-insensitive search is just as fast as an exact-match search.

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 stored in an actual table—accessing a materialized view won't run the query again like it would in a normal view.

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

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

you'd otherwise have to use a different type of database for). And, by using the JSONB data type, you ensure that 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 them, and you certainly can't index them. JSONB fields can interoperate with many systems other than Rails, and they provide great performance.

AngularJS

AngularJS[4] is a JavaScript framework created and maintained by Google. Angular allows you to model your user interface as *components*, which combine a model, template, and code all into one self-contained class. This means your view is not a static bit of HTML, but a full-blown application. By adopting the mind-set that your front end is a dynamic, connected interface comprised of components, and not a set of static pages, you open up many new possibilities.

Angular provides powerful tools for organizing your code and lets you structure your markup to create intention-revealing, testable, manageable front-end code. It doesn't matter how small or large the task—as your UI gets more complex, Angular scales 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>
  <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.

4. https://angular.io

An Angular version of this might look like this:

```
var DetailsComponent = ng.core.Component({
  template: '\
    <section> \
      <p>You currently owe: $123.45</p> \
      <button on-click="toggleDetails()">Show/Hide Details</button> \
      <ul *ngIf="showDetails"> \
        <li>Base fee: $120.00</li> \
        <li>Taxes: $3.45</li> \
      </ul> \
    </section> \
    '
  }).Class({
    constructor: function() {
      this.showDetails = false;
    },
    toggleDetails: function() {
      this.showDetails = !this.showDetails;
    }
  });
```

It may seem oddly-shaped, but it should reveal its intent much more clearly than the jQuery version, despite the extra bits of code. Without knowing Angular at all, you can piece together that when the button is clicked, it calls the toggleDetails button, and if showDetails is true, we'll show the detailed information inside the ul. This all maps closely to the user's actions and intent. There's nothing in this code about locating DOM elements or handling browser events. And, when you need to do fancier or more complex interactions in your front end, writing code this way is still easy to manage.

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 covered in this book. We're using Angular for a few reasons. First, it's quite popular, which means you can find far more resources online for learning it, including deep dives beyond what is covered here. Second, it allows you to compose your front end similarly to how you compose your back end in Rails, but it's 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're 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 convenient.

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 a clear separation exists between view and code.

Huge ecosystem

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

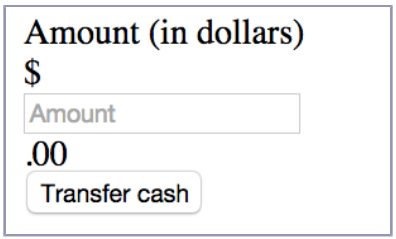
It's hard to fully appreciate the power of a JavaScript framework like Angular without using it, but we'll get there. We'll turn a run-of-the-mill search feature into a dynamic, asynchronous live search, with very little code.

Bootstrap

Bootstrap[5] 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 *design components*, which are classes that, when used on particular HTML elements in particular ways, produce a distinct visual artifact, like a form, a 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?

5. http://getbootstrap.com



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

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

Typography
: Just including Bootstrap in your application and using semantic HTML results in pleasing content with good general typography.

Grid
: Bootstrap's grid makes it easy to lay out complex, multicolumn 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, as mentioned earlier, 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 visual design, interaction design, or front-end implementation is crucial to the success of a project.

But for many apps, you don't need these specialists (they are very hard to find when you do). Bootstrap lets you produce a professional, appealing user interface without them. Bootstrap also lets you realize visual designs that might seem difficult to do with CSS.

Even if you have a designer or front-end specialist, the skills you'll learn by using Bootstrap will still apply—your front-end developer isn't going to write

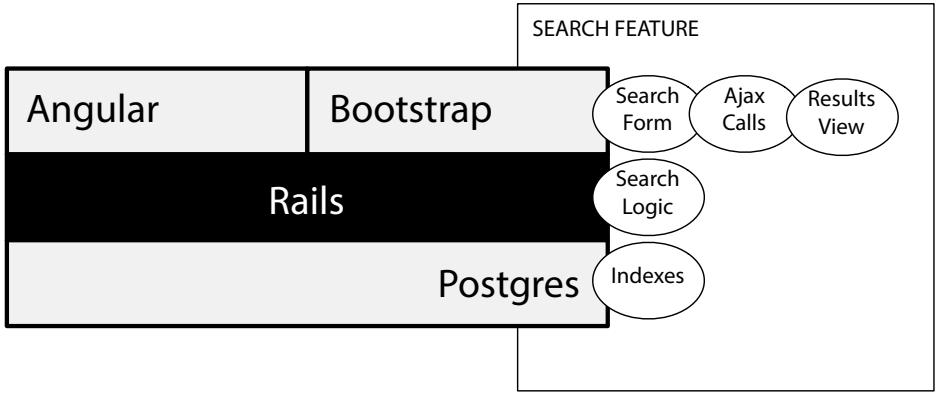
every line of markup and CSS. They are going to hand you a framework like Bootstrap that enables you to do many of the things we'll do in this book.

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

How to Read This Book

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 probably 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, as shown in the following diagram.



This holistic approach is how you're going to learn these technologies, and you'll have the most success reading the book in order. We'll build a Rails application together, adding features one at a time. These features will demonstrate various aspects of the technologies you're using.

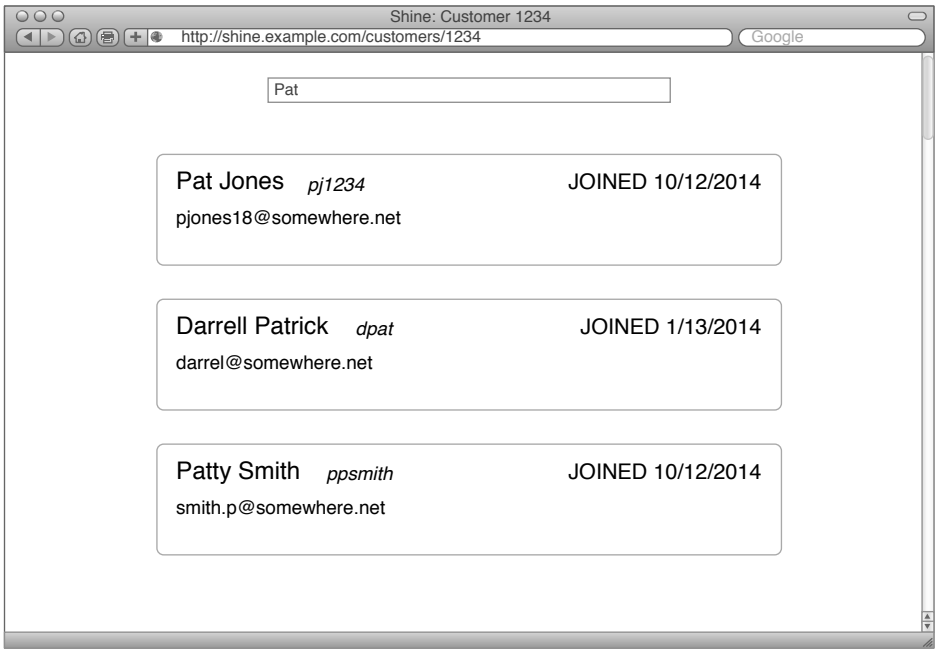
Shine, the Application We Build

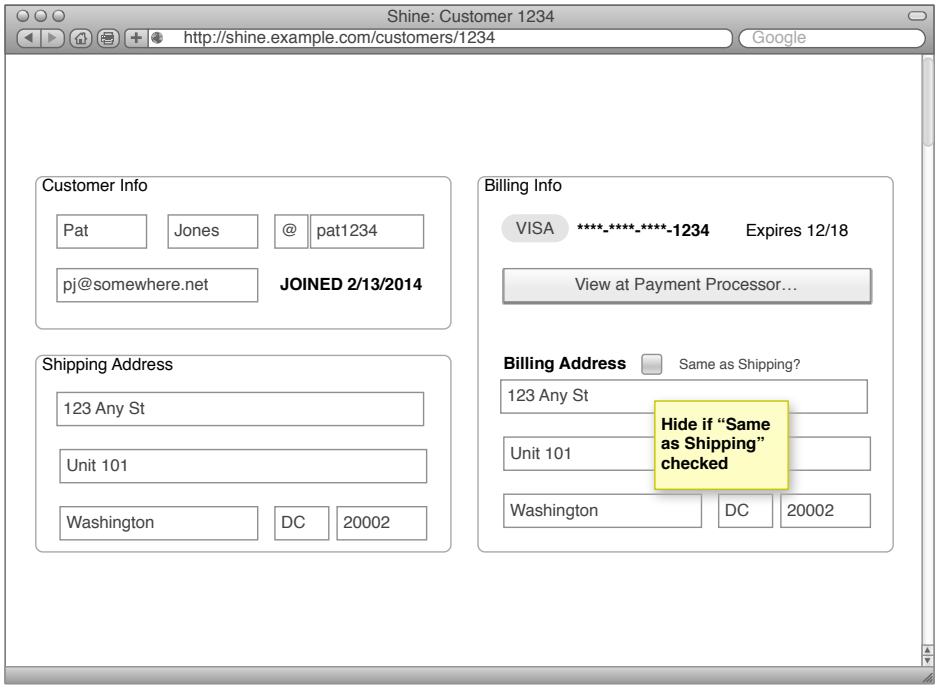
Throughout the chapters in this book we'll create and add features to a Rails application. We're creating this application for the customer service agents at the hypothetical company where we work. Our company has a public website that its customers use, but we want a separate application for the

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 involve searching for, viewing, and manipulating customer data.

For example, we'll allow the user to search for customers in a manner similar to what is shown in the following figures. They'll also be allowed to click through and view or edit a customer's data.





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

How We'll Build Shine

To keep things simple, each chapter focuses on a single technology, and you complete features over several chapters. We'll do as much setup as we can in Chapter 1, *Set Up the Environment*, on page 1. Because our goal is to bring together several technologies that weren't designed as a single package, we'll have to do a bit more configuration up front, but once that configuration is done, we'll be ready to add features and get the most out of everything.

The first feature we build is a registration and login system, which allows us to style the user interface with Bootstrap as well as secure the underlying database with Postgres. We 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 then tighten up the security by learning about *check constraints* in Chapter 3, *Secure the User Database*, on page 31.

We then move on to a customer search feature, which is a fertile ground for learning about full-stack development. In Chapter 4, *Perform Fast Queries with*, on page 39, we implement a basic fuzzy search, and you learn how to examine Postgres's *query plan* to understand why our search is slow. We then use Postgres's advanced indexing features to make it fast. In Chapter 5, *Create Clean Search Results*, on page 57, you learn how to use some of Bootstrap's built-in components and helper classes to create nontabular search results that look great.

Chapter 6, *Build a Dynamic UI with AngularJS*, on page 69, is an introduction to AngularJS, which we use to make our customer search much more dynamic. This chapter explores how to set up 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 pause in Chapter 7, *Test This Fancy New Code*, on page 95 to discuss how to write tests for everything you've learned. Testing has always been a big part of Rails, so whenever we veer off Rails's golden path, it's important to make sure we have a great testing experience.

Chapter 8, *Create a Single-Page App*, on page 131 is our first step in building a more complex feature that shows customer details. We 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 gives you an opportunity to learn about Angular's router and navigation features.

In Chapter 9, *Design Great UIs with*, on page 159, you learn about a powerful web design tool called *the grid* and how Bootstrap implements it. We use it to create a dense UI that's clean, clear, and usable. In Chapter 10, *Cache Complex Queries*, on page 175, we 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's *materialized views*.

In Chapter 11, *Asynchronously Load Data*, on page 197, you learn how Angular's asynchronous nature allows us to keep our front end simple, even when we need data from several sources. We finish off our customer detail page feature, as well as our in-depth look at these technologies, in Chapter 12, *Wrangle Forms and*, on page 199, by exploring Angular's data binding, which allows you 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 201, we'll survey some of the other features we don't have space to get to.

To help you keep track of where we are, each chapter starts with a diagram (like the one shown on page vii) that shows which parts of the stack we are focusing on, what feature we're building, and what aspects of each technology you are learning in that chapter.

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 be 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.

Example Code

The running examples in the book are extracted from fully tested source code that should work as shown, but you should download the sample code from the book's website at https://pragprog.com/titles/dcbang2/source_code. Each step of our journey through this topic has a different subdirectory, each containing an entire Rails application. While the book shows you only the changes you need to make, the downloadable code records a snapshot of the fully working application as of that point in the book.

Command-Line Execution

We'll be running a lot of commands in this book. Rails development is heavily command-line driven, and this book shows the commands you need to execute as we go. It's important to understand how to interpret the way we're using them in the book. Each time you see a command-line *session*, the text will first show how you call the command line, followed by the expected output.

In the following example, we invoke the command line `bundle exec rails generate devise:install`. This is the command you'd type into your terminal and press Return. The lines *following* the command line display expected or example output.

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

Sometimes I need to show a command-line invocation that won't fit on one line. In this case, I use backslashes to show the continuation of the command (which is how you'd actually run a multiline command in a shell like `bash`).

The last line *won't* have a backslash. For example, here we're typing everything from rails to shine.

```
$ rails new --skip-turbolinks \
            --skip-spring \
            --skip-test-unit \
            -d postgresql \
            shine
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
```

The sample output won't always match exactly what you see, but it should be close. It's included so you have a way to sanity-check what you're doing as you follow along.

A second form of command-line session is when we interact with Postgres. I'll indicate this by executing rails dbconsole first and then showing SQL commands inside the Postgres command-line interpreter. In the following listing I'm executing rails dbconsole; then I'm executing select count(*) from users; inside Postgres (note how the prompt changes slightly). After that you see the expected or sample results of the command.

```
$ rails dbconsole
postgres> select count(*) from users;
| count |
+-------+
|   100 |
+-------+
```

What You Need to Know

This book covers a lot of advanced topics in web development. However, my hope is that you can get a lot out of it regardless of your skill level. Nevertheless, the code and concepts are written assuming some basic exposure to the topics at hand:

Ruby and Rails

Much of the Rails content in the book is in configuration, specifically to get Rails to work with Angular, Bootstrap, and the ecosystem in which they live. If you've created a simple Rails app, and you know what controllers, models, views, migrations, and tests are, you should have no problem understanding the Rails code. If you're new to Rails, check out *Agile Web Development with Rails 4 [Rub13]*.

SQL

I also assume you know some SQL. If you know the basics of how to select, update, and insert data, along with how to do a join, you know everything you need to understand the Postgres parts of this book. For those of you new to SQL, there's an online course called "Learn SQL the Hard Way"[6] in development that will give you the necessary skills. As of this writing, Exercises 0–6 should give you a grounding in the basics, including joins.

JavaScript

You don't need to be a JavaScript expert, but you should know the basics of how JavaScript works as a language. *JavaScript, the Good Parts [Cro08]* is a quick read and should give you these basics if you need them. You don't have to know *any* Angular. I'll cover what you need and assume you've never seen any Angular code.

CSS

There's (almost) no actual CSS code in this book. You just need to know what CSS is and what it does. All the styling you'll do will be using Bootstrap, and I don't assume any prior knowledge.

I wrote the book with the following versions of the tools and libraries:

- Ruby 2.3
- Rails 5
- Bootstrap 3
- Angular 2
- Postgres 9.5
- Webpack 1.9
- Node 6.3.0
- Devise 4.2

For everything else you need, I'll show you how to set up 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 `package.json` that are included in the example code download. While I've tried to make the code future compatible, there's always a chance that a point release of a library breaks something.

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 https://pragprog.com/titles/dcbang2/errata. Think of it as a bug-reporting system for the book.

6. http://sql.learncodethehardway.org/book

For the former—questions about the material—you should visit the online forum on the book webpage at https://pragprog.com/titles/dcbang2. There, you'll be able to interact with other readers and me 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 doing all the up-front setup for our development environment.

CHAPTER 1

Set Up the Environment

Before you get into learning Postgres, Angular, and Bootstrap, there's a bit of setup to do first. Setup is never fun, and it's often difficult to figure out by just looking at documentation, especially when the tools aren't designed together. This isn't to say that Postgres, Angular, and Bootstrap aren't great tools that *can* work with Rails, but given what you're accustomed to with Rails, setting up these tools may be more work than you're used to.

Setup can also be distracting when trying to learn something new, despite being crucially important, since you have to be able to actually use the new tools you're learning. This is what you'll do in this chapter. By the end, you'll have a shell of the Rails application you'll be working on throughout the rest of the book. You'll have Postgres installed and set up, and you'll have configured Webpack to serve up Bootstrap, as well as the JavaScript you'll write in later chapters.

We'll do this in three steps. First, we'll install Ruby, Rails, and Postgres, which should be straightforward. Next, we'll create the Rails appplication we'll be building for the rest of the book using some particular options that aren't the default. Finally, we'll setup NPM and Webpack, which we'll be using to manage our CSS and JavaScript, and install Bootstrap to validate that they are working. I'll also explain why we're using Webpack at all.

Installing Ruby, Rails, and Postgres

Most of what we'll do in building our example Rails application, Shine, are commands you can execute or code you'll enter, but getting Ruby, Rails, and Postgres installed on your system is much more system-specific. Fortunately, the authors of these technologies have made the installation process as pain-free as possible. First, start with Ruby and Rails.

Ruby and Rails

If you don't have Ruby installed, follow the instructions on Ruby's website.[1] I recommend using an installer or manager, but as long as you have Ruby 2.3.1 or later 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
```

Be sure to get the latest version of Rails, which is 5.0.0.1 as of this writing.

Next, you'll need access to a Postgres database.

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.[2] Be sure you get at least version 9.5, as some of the features I'll discuss were introduced in that version. When we create our Rails application in the next section, we'll create a user to access the database.

An alternative is to use a free, hosted version of Postgres, such as Heroku Postgres.[3] You can sign up on their website, and they'll give you the credentials to access the hosted database from your computer. (You'll learn where to use them in the next section when we configure Rails to access Postgres.)

With Ruby, Rails, and Postgres installed, let's create the Rails application and get a basic view working.

Creating the Rails Application

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

First, we want to tell Rails to use Postgres as our database (it uses SQLite by default). We also don't want to use TurboLinks,[4] because it's going to clash with the JavaScript you'll be writing later on when you 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 in the

1. https://www.ruby-lang.org/en/downloads
2. http://www.postgresql.org/download
3. https://www.heroku.com/postgres
4. https://github.com/rails/turbolinks
5. https://github.com/rails/spring

book. Finally, we're skipping Test::Unit as our testing framework, because we're going to be using RSpec.[6] As you see in Chapter 7, *Test This Fancy New Code*, on page 95, we have good reasons to use RSpec, so bear with me for now.

Now that you've installed Rails, we can use the rails command-line app to create Shine:

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

If you get an error installing the pg gem, it could be that your Postgres install is in a nonstandard place. You may need to run gem install pg --with-pg-config=«path to your pg_config». (To locate *that* highly depends on how you installed Postgres. You want to find the executable named pg_config and this usually lives inside the bin directory of wherever you installed Postgres.)

The rails new command above creates the new application as you'd expect. Because almost all of our work will be in this Rails app, let's go ahead and change the current directory to the shine directory that contains the new Rails application:

```
$ cd shine
```

Before you run the Rails application, you'll need to set up your database. If you've installed Postgres locally, you'll need to create a user (if you're 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. You can create it using the command-line app installed with Postgres called createuser.

```
$ 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 log in 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). See *If the createuser Command Isn't Found on Your System*, on page 4 if your system doesn't have createuser available.

If the createuser Command Isn't Found on Your System

6. http://rspec.info

In some Linux installations, the createuser command isn't available. While you should consult the installation documentation for your operating system, you can create a user inside Postgres directly. You'll need to access the postgres schema, which you can usually do with psql like so:

```
$ psql postgres
```

Doing *this* could be tricky, as you may need to be the postgres user in your system. Again, the installation instructions for your operating system should show the way. Once you've done this, you can create a user inside psql:

```
$ psql postgres
postgres> CREATE USER shine PASSWORD 'shine';
```

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

```
default: &default
  adapter: postgresql
  encoding: unicode
  host: localhost
  username: shine
  password: shine
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>

development:
  <<: *default
  database: shine_development

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

If you're 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, let's set up the database:

```
$ bundle exec rails db:create
$ bundle exec rails db:migrate
```

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

```
$ bundle exec rails server
```

You can now visit http://localhost:3000 and see the new Rails 5 welcome page, as shown in the following screenshot.

Last, let's make a page you can use to see the entire Rails view-rendering life-cycle. This is needed to validate that you've installed Webpack properly, which we'll do after this. We'll call our page 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

```
Rails.application.routes.draw do
  root to: "dashboard#index"
end
```

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>
```

Restart your server, reload your app, and you should see the page we've created:

Now that you have a working Rails app, let's install Bootstrap. This requires first learning how to use NPM to install Webpack, which will serve up our CSS and JavaScript for the rest of the book. (I'll explain why we aren't using Rails' asset pipeline as well.)

Setting Up Bootstrap with NPM and Webpack

We want our users' experiences 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 that helps users get the most out of our software. A framework is a set of reusable 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 lay out effectively on the page.

Bootstrap[7] is one of the most popular and widely used CSS frameworks, and it gives programmers an immense amount of power to control the look and feel of their apps, without having to write any CSS themselves. Bootstrap is

7. http://getbootstrap.com

no replacement for an actual designer—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 install bootstrap, you're going to end up setting up NPM to manage Shine's dependency on Bootstrap's code, and Webpack to serve it up. Webpack will eventually serve up Angular as well as the JavaScript you'll eventually write.

Although you could reference Bootstrap from a CDN[8] host, this is not common for production applications. In most cases, you want to control exactly what is bundled in your application, so you want Bootstrap to be part of the application. You could use RubyGems to achieve this; however, not all assets and JavaScript libraries you are going to want are available or up-to-date (for example, at the time of this writing, Angular 2 is not available).

This means we need another way to download Bootstrap's code and include it into our application. I'd like to use a system like Bundler to manage it, but because Bundler only manages RubyGems, we'll turn to the package management system for JavaScript (and the rest of the front-end ecosystem): NPM.

NPM[9] is part of Node.js and is the Node equivalent of Bundler. Instead of a `Gemfile`, NPM uses a file named `package.json`. Just as running `bundle install` updates the Ruby gems in our application, `npm install` updates the front-end assets and libraries. This is the easy part. The hard part is figuring out how to serve up the assets downloaded with NPM.

In theory, the Rails Asset Pipeline, powered by Sprockets, should be able to do this. In practice, it's not easy making this happen. In addition to the configuration difficulties this would involve, Sprockets is also missing most modern features of front-end development. While we can do without many of them, when you start writing Angular code in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 69, you'll need a few features Sprockets simply doesn't provide.

So, while you might be able to get Bootstrap's CSS downloaded somewhere that Sprockets can serve it up, Sprockets won't work for code you'll write later on, and we don't want two systems to manage our assets—we just want one. That system is Webpack.[10]

Webpack is an asset pipeline that supports most modern features in front-end development, along with everything we're used to from Sprockets. It will

8. http://en.wikipedia.org/wiki/Content_delivery_network
9. https://www.npmjs.com
10. https://webpack.github.io

allow us to serve up Bootstrap (once installed) and handle all the JavaScript needs we'll have in later chapters. Unlike Sprockets, however, Webpack requires a substantial amount of up-front configuration to get working. That's what I'll cover in the following section, but to do that, we have to first install NPM.

Install NPM

NPM is part of Node.js, so installing NPM means installing Node. 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. The version of Node shouldn't matter too much, but this book was written using version 6.3.0.

Once you have Node installed, you'll have access to the `npm` command-line application. We'll use this later to install Bootstrap, but next we need to install Webpack.

Install and Configure Webpack

While you could install Webpack directly with NPM, there's a handy RubyGem called *webpack-rails*[13] that handles some Rails-specific setup. It provides some rake tasks to help with deployment but, more important, it provides a generator to set up a Webpack configuration that will work with Rails.

Add webpack-rails to your `Gemfile`:

setup/install-webpack/shine/Gemfile

```
gem 'rails', '~> 5.0.0'
gem 'pg', '~> 0.18'
gem 'puma', '~> 3.0'
gem 'sass-rails', '~> 5.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.2'

gem 'jquery-rails'
gem 'jbuilder', '~> 2.5'
➤ gem 'webpack-rails'
```

After running `bundle install` to install the gem, run the `webpack_rails:install` generator to create the necessary configuration files. Note that the generator will ask

11. http://nodejs.org
12. https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager
13. https://github.com/mipearson/webpack-rails

you two questions: if you want it to run npm install and if you want it to run bundle install. Say "yes" to both.

```
$ bundle exec rails generate webpack_rails:install
     gemfile  foreman
      create  Procfile
      create  package.json
      create  config/webpack.config.js
       exist  webpack
      create  webpack/application.js
      append  .gitignore
Would you like us to run 'npm install' for you? yes
         run  npm install from "."
npm WARN webpack-rails-example@0.0.1 No repository field.
Would you like us to run 'bundle install' for you? yes
         run  bundle install from "."
```

At the time of this writing, there are two issues you might run into. First, bundle install might not have actually been executed. To be safe, run bundle install again. Second, there is an incompatibility in some of Webpack's dependencies, so you'll need to add an additional line to package.json:

```
{
  "name": "shine",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    "stats-webpack-plugin": "^0.2.1",
    "webpack": "^1.9.11",
    "webpack-dev-server": "^1.9.0",
➤   "webpack-dev-middleware": "1.7.0"
  }
}
```

Then, re-run npm install.

```
> npm install
«ASCII-art animations of packages being checked and installed»
```

If you aren't familiar with Node or NPM, npm install downloaded all the necessary JavaScript libraries and placed them into node_modules. Generally, you don't want to check this into your version control system, but it depends on how you deploy. In Appendix 3, *How Webpack Affects Deployment*, on page 207 we'll discuss some of the implications of this setup on your deployment.

Also note that the generator ran bundle install. This was to bring in a new Ruby gem called *foreman*.[14] The reason you need *this* is because you now have to

14. https://github.com/ddollar/foreman

run two commands whenever you run the Rails application. You'll need to run rails server just as you would in any Rails application, but you now need to run webpack-dev-server, which will serve up our CSS and JavaScript assets in development (similar to what Sprockets does in a classic Rails application).

Because Webpack is not integrated into Rails, it has no way to automatically run when you type rails server, so you have to run it alongside our Rails application. While you could open up two terminal windows and run each command, this is cumbersome and error-prone. Instead, we'll list all the commands you need to run in a file called Procfile, which foreman can read. (Note that Procfile was created by the webpack-rails generator we jsut ran.) If you look at the Procfile that was generated, it should make sense:

```
# Run Rails & Webpack concurrently
# Example file from webpack-rails gem
rails: bundle exec rails server
webpack: ./node_modules/.bin/webpack-dev-server --config config/webpack.config.js
```

You can then use the foreman gem to run these commands together via foreman start. For the rest of the book, this is how we'll run Shine and we won't be doing bundle exec rails server any longer. Before we try it out, there is one more bit of configuration we need to do.

webpack-dev-server serves our assets during development. In production, you'll arrange for Webpack to create a bundle (similar to what is created by the Asset Pipeline during rails assets:compile), and that bundle will be served by Rails instead. (See Appendix 3, *How Webpack Affects Deployment*, on page 207 for a brief explanation.) In both cases, you have to tell the Rails application where those files are so it can generate the proper script tags in your markup.

Webpack-rails provides a webpack_asset_paths helper method you can drop into your application layout. Open up app/views/layouts/application.html.erb and add a call to javascript_include_tag using the new webpack_asset_paths helper, like so:

setup/install-webpack/shine/app/views/layouts/application.html.erb

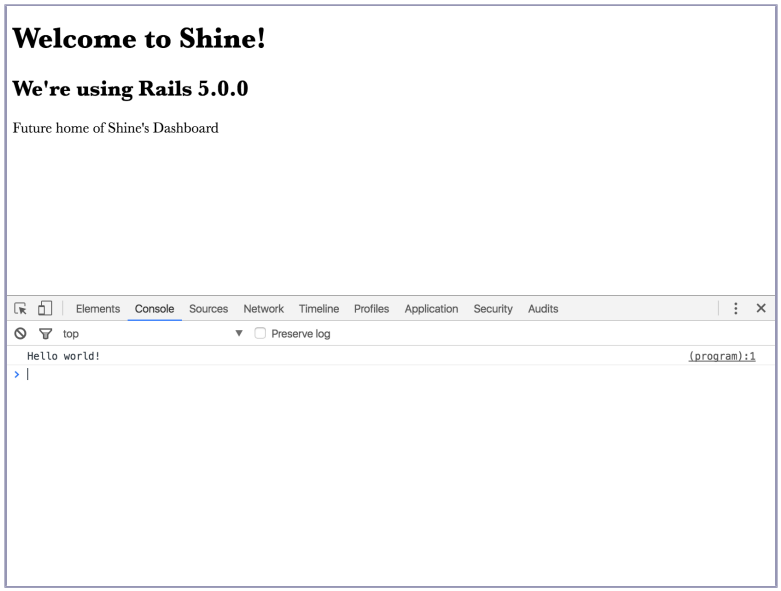
```
<head>
  <title>Shine</title>
  <%= csrf_meta_tags %>

  <%= stylesheet_link_tag    'application', media: 'all' %>
  <%= javascript_include_tag 'application' %>
➤ <%= javascript_include_tag *webpack_asset_paths("application") %>
</head>
```

Now, start Shine using foreman start (note that it will be running on port *5000* instead of 3000):

```
$ foreman start
```

```
08:29:05 rails.1   | started with pid 30480
08:29:05 webpack.1 | started with pid 30481
08:29:06 webpack.1 | http://localhost:3808/webpack-dev-server/
08:29:06 webpack.1 | webpack result is served from //localhost:3808/webpack/
08:29:06 webpack.1 | content is served from /Users/davec/Projects/rails-boo…
08:29:06 webpack.1 | Hash: 371845730ac40140dc34
08:29:06 webpack.1 | Version: webpack 1.13.1
08:29:06 webpack.1 | Time: 62ms
08:29:06 webpack.1 |          Asset     Size  Chunks             Chunk
08:29:06 webpack.1 | application.js  1.91 kB       0  [emitted]  applica…
08:29:06 webpack.1 |  manifest.json  0 bytes          [emitted]
08:29:06 webpack.1 | chunk    {0} application.js (application) 29 bytes
08:29:06 webpack.1 |     [0] ./webpack/application.js 29 bytes {0} [bui…
08:29:06 webpack.1 | webpack: bundle is now VALID.
08:29:08 rails.1   | => Booting Puma
08:29:08 rails.1   | => Rails 5.0.0 application starting in \
                       development on http://localhost:5000
08:29:08 rails.1   | => Run `rails server -h` for more startup options
08:29:08 rails.1   | Puma starting in single mode...
08:29:08 rails.1   | * Version 3.4.0 (ruby 2.3.1-p112), codename: Owl Bowl…
08:29:08 rails.1   | * Min threads: 5, max threads: 5
08:29:08 rails.1   | * Environment: development
08:29:08 rails.1   | * Listening on tcp://localhost:5000
08:29:08 rails.1   | Use Ctrl-C to stop
```

If you navigate to http://localhost:5000 and open the JavaScript console in your browser, you should see a “Hello world!” message similar to what is shown in the following figure.



This is coming from webpack/application.js, which was created by webpack-rails' generator:

```
console.log("Hello world!");
```

This validates that you've set up Webpack for our development environment; however, it's currently just configured to serve JavaScript. We're trying to serve up CSS, so there's a bit more work to do. Webpack is extremely unopinionated, and by default it doesn't handle CSS. It's also extremely flexible, so it can be made to serve up CSS by configuring some standard extensions.

These extensions are called *loaders* because they tell Webpack how to *load* particular file. In this case, Webpack views the problem of serving up CSS as being compsed of two smaller problems: how to read and interpret a CSS file, and how to serve it up to the front-end. The former is handled by Webpack's standard CSS loader, and the latter by its style loader (so named because it ultimately writes a <style> tag to our markup).

This works differently from what you're used to with Sprockets. With Sprockets, all CSS in app/assets/stylesheets is bundled up and served as a single .css file via a <link> tag. Webpack is going to use JavaScript to write out the CSS when the page loads, but that CSS is included in the JavaScript bundle. Yes, it's weird, but bear with me.

While we're setting this up, you also want to configure Webpack to serve Bootstrap's webfonts. You won't learn about those until Chapter 13, *Dig Deeper*, on page 201, but you should get the configuration sorted out now. Webfonts are handled by either a url loader or a file loader, depending on the file format of the webfont. In other words, we need both loaders because each browser uses a different file format for webfonts.

The four loaders you need aren't installed with Webpack—they are add-ons. This means you need to use NPM to bring them in. As mentioned earlier, the file package.json lists out the NPM modules our application needs. Webpack-rails' generator created one for us, which looks like so:

```
{
  "name": "webpack-rails-example",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    "stats-webpack-plugin": "^0.2.1",
    "webpack": "^1.9.11",
    "webpack-dev-server": "^1.9.0"
  }
}
```

We'll add the four loaders we need, which are named css-loader, style-loader, file-loader, and url-loader. We'll also change the name in package.json to "shine" instead of "webpack-rails-example."

```
{
  "name": "shine",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    "stats-webpack-plugin": "^0.2.1",
    "webpack": "^1.9.11",
    "webpack-dev-server": "^1.9.0",
    // START-HIGHLIGHT
    "css-loader": "^0.23.1",
    "file-loader": "^0.9.0",
    "style-loader": "^0.13.1",
    "url-loader": "^0.5.7"
    // END-HIGHLIGHT
  }
}
```

I've specified versions so that your experience matches mine. In the real world, you'd specify more open-ended versions, similar to the way you do it in a Gemfile. You'd also create a *shrinkwrap* verison of this file, which is NPM's equivalent to Gemfile.lock.

Once you install these with npm install, you now have to tell Webpack to use these loaders. You do this by giving Webpack regular expressions that match the files you want these loaders to handle, and then a magic string that indicates how to use the loaders.

Be warned; this configuration is arcane. In particular, the syntax needed to specify the loader is difficult to derive without explicit documentation. Fortunately, the loaders' documentation usually indicates the magic string you need.

setup/install-webpack/shine/config/webpack.config.js

```
var config = {
  entry: {
    // Sources are expected to live in $app_root/webpack
    'application': './webpack/application.js'
  },

  module: {
    loaders: [
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader'
      },
```

```
      {
        test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,
        loader: "file"
      },
      {
        test: /\.(woff|woff2)$/,
        loader:"url?prefix=font/&limit=5000"
      },
      {
        test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,
        loader: "url?limit=10000&mimetype=application/octet-stream"
      },
      {
        test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,
        loader: "url?limit=10000&mimetype=image/svg+xml"
      }
    ]
  },

  // rest of the configuration ...
```

The most relevant part of this to the task at hand is the first line inside loaders. It translates, roughly, as "for all CSS files, load them with the css-loader, however pass that result to the style-loader." The loader1!loader2 is somewhat like a UNIX pipe that runs backward. Like I said, it's arcane.

With this in place, we'll test out the CSS support by creating a simple CSS file that turns all text in Shine to Courier. Create webpack/application.css like so:

```
* {
  font-family: monospace;
}
```

You can now tell Webpack about this file by using the CommonJS require[15] function inside webpack/application.js:

```
require ("application.css");
console.log("Hello World");
```

If you restart your server and reload the page, all text will be in your configured monospaced font, thus demonstrating that our CSS is being loaded and served. If you inspect the markup, you'll see the contents of webpack/application.css inside a style tag in the head of the page.

Like I said, this is weird and verbose. Because Webpack is so flexible, you could configure it to serve CSS the way Sprockets does. There's not a huge reason for us to do that here, and you've already been through enough con-

15. http://www.commonjs.org/specs/modules/1.0

figuration for the time being, so let's get back to the problem at hand, which is installing Bootstrap.

Download and Set Up Bootstrap

With Webpack's initial setup out of the way, bringing in Bootstrap is relatively straightforward. First, add bootstrap to package.json:

```
{
  "name": "shine",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    "stats-webpack-plugin": "^0.2.1",
    "webpack": "^1.9.11",
    "webpack-dev-server": "^1.9.0",
    "css-loader": "^0.23.1",
    "file-loader": "^0.9.0",
    "style-loader": "^0.13.1",
    "url-loader": "^0.5.7",
➤   "bootstrap": "3.3.7"
  }
}
```

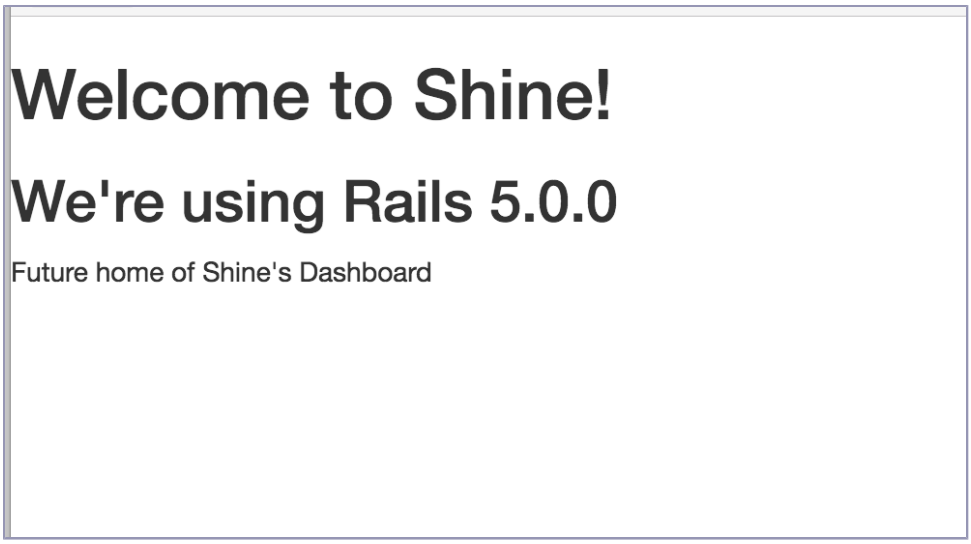
After running npm install to download bootstrap, change webpack/application.js to bring in Bootstrap's CSS:

setup/install-webpack/shine/webpack/application.js

```
require("application.css");
require("bootstrap/dist/css/bootstrap.css");
console.log("Hello world!");
```

You should also remove the contents of webpack/application.css as well so that the file is empty. (You don't want Shine to use a monospace font as its default font; that was just a test of our configuration.)

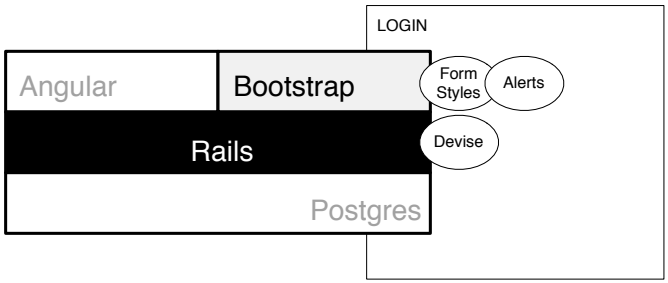
With all this done, restart your app, navigate to the home page, and you should see that the font is now rendering in Helvetica, which is Bootstrap's default.



Whew! That was a lot of setup and manual configuration, but it will all be worth it as you start adding features to Shine. Whenever you are trying to get new technologies to work together, it's worth spending time up-front on configuration, because you don't want to have to revisit it later. Webpack will be particularly helpful when our JavaScript code gets complex and we need to write tests for it.

Next: Authentication with Devise, Styled by Bootstrap

Now that you are set up and ready to go, let's dive into Shine by setting up a basic authentication system using Devise. Devise does all the hard stuff for us, but it doesn't create very usable or aesthetically pleasing views. That's where Bootstrap comes in.



CHAPTER 2

Create a Great-Looking Login with Bootstrap and Devise

Now that you have created and configured Shine, downloaded Bootstrap using NPM, and have Webpack set to serve it all up, let's start adding features to Shine. In this chapter you'll set up a simple authentication system using Devise. Devise does all the hard work around authentication, but its default user interface leaves much to be desired. Bootstrap makes short work of this, and by the end of the chapter you'll have a secure Rails application with user-friendly views in no time, all without writing any CSS.

Adding Authentication with Devise

An authentication system is a great first step into thinking about problems as a full-stack developer—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. We'll use the *Devise*[1] gem to handle the middleware bits of authentication.

Creating an authentication system from scratch is rarely a good idea. It's difficult to get every part of it correct, as security controls can be subverted in unusual and counterintuitive ways. Devise is tried-and-true and handles all of this for us. It's also quite flexible and will totally suit our needs. Here are the rules we want for our our authentication system:

- Employees who need to use our app will sign up on their own.
- Employees must use their company email addresses when signing up.

1. https://github.com/plataformatec/devise

- Employees' passwords must be at least 10 characters long.

Our app isn'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.[2]

First, add Devise to your Gemfile:

login/install-devise/shine/Gemfile
```
gem 'jquery-rails'
gem 'jbuilder', '~> 2.5'
gem 'devise'
```

Next, install it using Bundler:

```
$ bundle install
```

Devise includes several generators[3] you can use to simplify the setup and initial configuration. The devise:install generator is the first one you'll need to bootstrap Devise in the app.

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

This command outputs a fairly lengthy message about further actions to take to set up Devise. I'll address 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. Because 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 Active Record model and database table (called USERS) with the fields necessary for Devise to function.

2. http://guides.rubyonrails.org/action_mailer_basics.html#action-mailer-configuration
3. http://guides.rubyonrails.org/command_line.html#rails-generate

```
$ bundle exec rails generate devise user
    invoke  active_record
    create    db/migrate/20160714144446_devise_create_users.rb
    create    app/models/user.rb
    insert    app/models/user.rb
     route  devise_for :users
```

Have a look at what it created by examining the migration:

login/install-devise/shine/db/migrate/20160714144446_devise_create_users.rb

```
class DeviseCreateUsers < ActiveRecord::Migration[5.0]
  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

      ## Confirmable
      # t.string   :confirmation_token
      # t.datetime :confirmed_at
      # t.datetime :confirmation_sent_at
      # t.string   :unconfirmed_email # Only if using reconfirmable

      ## Lockable
      # t.integer  :failed_attempts, default: 0,
      #            null: false # Only if lock
      #                        # strategy is :failed_attempts
      # t.string   :unlock_token # Only if unlock
      #                          # strategy is :email or :both
      # t.datetime :locked_at

      t.timestamps null: false
    end

    add_index :users, :email,                unique: true
    add_index :users, :reset_password_token, unique: true
    # add_index :users, :confirmation_token,   unique: true
    # add_index :users, :unlock_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 you 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 you can use that in your 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 to be certain you've actually authenticated the user, let's show the email address on the dashboard. Devise provides a helper method called current_user, which returns the User instance of the currently authenticated user. Because it's a helper, you can use it directly in your 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
== 20160714144446 DeviseCreateUsers: migrating ================================
-- create_table(:users)
   -> 0.0321s
-- add_index(:users, :email, {:unique=>true})
   -> 0.0184s
-- add_index(:users, :reset_password_token, {:unique=>true})
```

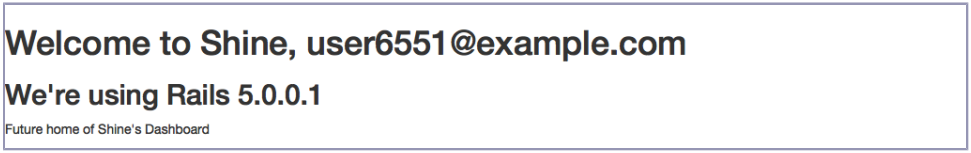
```
   -> 0.0070s
== 20160714144446 DeviseCreateUsers: migrated (0.0577s) =======================
```

```
$ bundle exec foreman start
```

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

Because 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.



You can also see that Devise has created an entry in the USERS table by going into the database directly. Use the Rails dbconsole[4] command to access the database so that you can examine the USERS table:

```
$ bundle exec rails dbconsole
postgres> \x on
Expanded display is on.
postgres> select * from users;
-[ RECORD 1 ]----------+--------------------------------------------------
id                     | 1
```

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

```
email                  | user7722@example.com
encrypted_password     | $2a$11$JiBGEx3/TCx6SsQPmSp8/uWPdQHqy/LlBaZQ8wL4d2
reset_password_token   |
reset_password_sent_at |
remember_created_at    |
sign_in_count          | 1
current_sign_in_at     | 2016-07-14 14:56:52.245834
last_sign_in_at        | 2016-07-14 14:56:52.245834
current_sign_in_ip     | ::1
last_sign_in_ip        | ::1
created_at             | 2016-07-14 14:56:52.242083
updated_at             | 2016-07-14 14:56:52.246722
```

Note that you can exit `dbconsole` by typing `\q` and hitting Return.

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

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

Styling the Login and Registration Forms

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

You might recall that you didn't write any markup for the login screens—they were all provided by Devise. Devise is packaged as a Rails Engine,[5] 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
  create    app/views/devise/passwords
  create    app/views/devise/passwords/edit.html.erb
  . . .
```

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

```
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 you can edit these files, you can use Bootstrap's CSS classes to make them look how you'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, you can log out to see the screens you're going to style. You're just going to be using the styles Bootstrap provides—you aren't writing any CSS yourself. You'll be amazed at how much better our screens are with just these simple changes.

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

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

```
➤ <body class="container">
    <%= yield %>
  </body>
```

Reloading the app, you can see that this class added some sensible margins and padding.

Let's start with the login screen.

Style the Login Screen

Because Devise uses Rails's 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[6] 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 the 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>
  <% 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.

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

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 add a few classes to the markup!

Before moving on to the registration screen, there's one more thing to fix here. If you submit the form without providing an email address or password, devise sets an error message in the Rails flash.[7] We're currently not displaying that anywhere.

We need to display it and style it in a way that allows users to easily see it. This will help users better understand when they've messed something up.

Style the Flash

In addition to classes designed to work with existing HTML entities like forms, Bootstrap provides *components*, which are a set of classes that, when applied to an element (or set of elements), create a particular effect. For the flash, Bootstrap provides a component called an alert.[8]

Let's display the flash using this 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

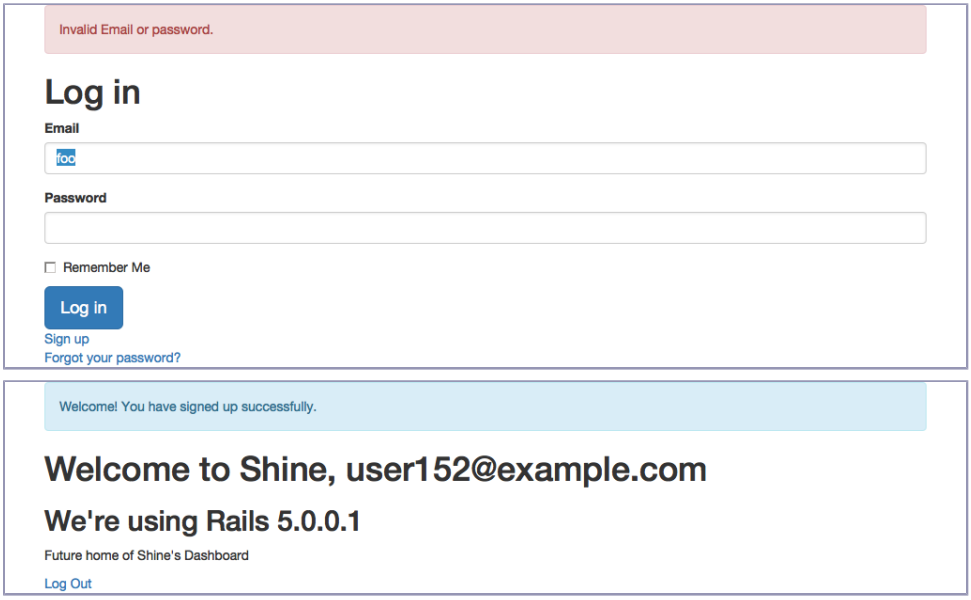
```
  <body class="container">
➤   <% if notice.present? %>
➤     <aside class="alert alert-info">
➤       <%= notice %>
➤     </aside>
➤   <% end %>
➤   <% if alert.present? %>
➤     <aside class="alert alert-danger">
➤       <%= alert %>
```

7. http://guides.rubyonrails.org/action_controller_overview.html#the-flash
8. http://getbootstrap.com/components/#alerts

```
➤        </aside>
➤      <% end %>
       <%= yield %>
    </body>
</html>
```

The markup got a bit more complex. We're using aside instead of div since 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 code to check whether 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, you can navigate to a page requiring login, provide incorrect login details, and see that the flash messages are styled appropriately. Users can now easily see their mistakes and understand their successes, as shown in the following screenshots.



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

Style 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 %>
    <%= 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 %>
    <%= f.password_field :password,
                         autocomplete: "off",
                         class: "form-control" %>
  </div>
  <div class="form-group">
    <%= f.label :password_confirmation %>
    <%= 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. You 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. I'll leave that as an exercise for you to style those pages, but it will be just as simple as what you'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 that aren't provided by Devise by default. In the next section, you'll see how to configure Devise to meet these requirements.

Validating Registration

If you look at the User model that Devise created, you 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. Note the :validatable symbol in the list.

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

```
class User < ApplicationRecord
  devise :database_authenticatable,
         :registerable,
         :recoverable,
         :rememberable,
         :trackable,
➤        :validatable
end
```

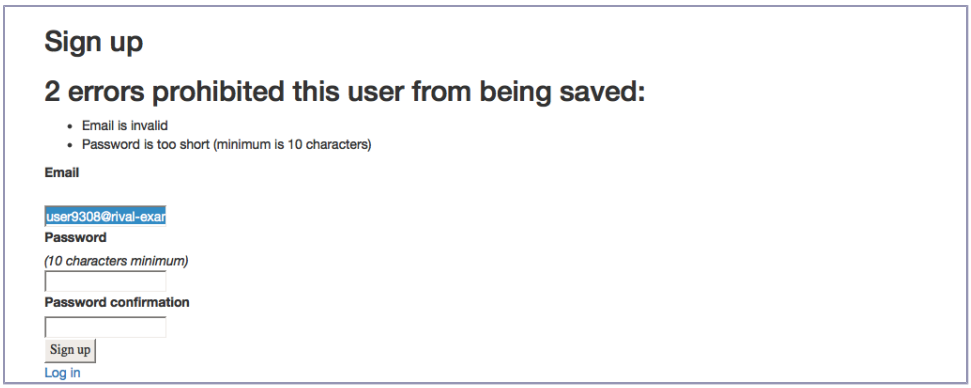
This :validatable *module* is what we're interested in. By using this in our model, Devise sets up various validations for the model, namely that the password is at least eight characters and that the email addresses looks like an email addresses. These defaults are set in the initializer Devise created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

The initializer, located in config/initializers/devise.rb, has copious documentation about all of Devise's configuration options. If you search for the string "validatable," you can find the options we want to change, which are password_length and email_regexp. We'll change the minimum password length to 10 characters 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/
```

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



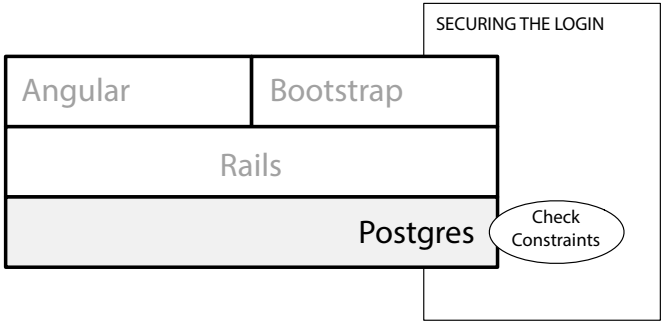
Note the unstyled error messages. These are produced by the helper method devise_error_messages!. You can override how this works by creating app/helpers/devise_helper.rb and implementing your own devise_error_messages! to output whatever markup you want. The details are on Devise's wiki[9] and, along with what you just learned about styling alerts with Bootstrap, you should be able to easily make those error messages look great.

This covers the user experience and, because this is implemented using Active Record validations, also covers most typical code paths that might modify the email column in the USERS table. Most, but not all.

Next: Using Postgres to Make Our Login More Secure

Devise cannot absolutely prevent users from being added into our database that do not meet our security criteria. For example, Active Record 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's *check constraints* can do this.

9. https://github.com/plataformatec/devise/wiki/Override-devise_error_messages!-for-views



CHAPTER 3

Secure the User 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 it can be. As you recall from the previous chapter, 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's 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 from 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 is about setting up a check constraint for Postgres as part of our Rails database migrations.

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

Exposing the Vulnerability Devise and Rails Leave 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` that 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 address 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 that 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 deactivating Sally's email account, she won't have access to any more internal systems. Aaron doesn't know that Sally was 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.*

Preventing 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`, but `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 you can create a constraint on your `email` field that requires its value to match the

same regular expression you used in our Rails code. This would prevent non-company email addresses from being inserted into the table entirely.

First, create a new migration where you can add this constraint:

```
$ bundle exec rails g migration add-email-constraint-to-users
    invoke  active_record
    create    db/migrate/20160718143725_add_email_constraint_to_users.rb
```

The *Domain-Specific Language* (DSL) for writing Rails migrations doesn't provide any means of creating this constraint, so you have to do it in straight SQL. Although Postgres *Data Definition Language* (DDL) looks different from what is normally used in 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 the following sidebar to learn why we're using the older `up` and `down` methods):

login/add-postgres-constraint/shine/db/migrate/20160718143725_add_email_constraint_to_users.rb

```
class AddEmailConstraintToUsers < ActiveRecord::Migration[5.0]
  def up
    execute %{
      ALTER TABLE
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '^[^@]+@example\\.com' )
    }
  end

  def down
    execute %{
      ALTER TABLE
        users
      DROP CONSTRAINT
        email_must_be_company_email
    }
  end
end
```

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.

1. http://www.postgresql.org/docs/9.5/static/ddl-constraints.html

Why Aren't We Using change in Our Rails Migrations?

Rails 3.1 introduced the concept of *reversible migrations* via the method `change` in the 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.

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[5.0]
  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 by first running the migrations.

```
$ bundle exec rails db:migrate
== 20160718143725 AddEmailConstraintToUsers: migrating =============
-- execute("
      ALTER TABLE
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '^[^@]+@example\\.com' )
    ")
   -> 0.0324s
== 20160718143725 AddEmailConstraintToUsers: migrated (0.0324s) ====
```

If you ran the migrations and saw something like the following error, 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, it's safe to manipulate the database or just blow it away. If you *are* seeing this issue, I recommend you just delete all the rows from the table as that's easy to do via delete from users;. If you were doing this to an active, production data set, you wouldn't have that luxury. You'd 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 from 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, for example, active = true AND email ~* '[A-Za-z0-9._%-]@example\.com'.

In any case, if you're 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, 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).
```

You can see that Postgres refuses to allow invalid data into the table (and you get a pretty useful error message as well). This means that a rogue application, bug in the 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's 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 use 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. You can do this by adding one line to config/application.rb.

login/add-postgres-constraint/shine/config/application.rb

```
module Shine
  class Application < Rails::Application
➤   config.active_record.schema_format = :sql
  end
end
```

You'll then need to remove the old db/schema.rb file, create db/structure.sql by running migrations, and finally reset your test database by dropping it and re-creating it. You can do all this with rake:

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

You May Get Churn in db/structures.sql

Because db/structure.sql is a Postgres-specific dump of the schema, certain aspects of it are dependent on the local environment. For example, if you use add_foreign_key, the names Postgres auto-generates might be different on different machines.

It's not a big problem for your application's behavior, since db/structure.sql is not used in production, but it can make for unnecessary churn in your version control history. You can combat this by tightly controlling the versions of Postgres each developer is using, providing explicit names for constraints and indexes, and not committing spurious changes to the file.

Why Use Rails Validations?

Given the power of check constraints, why would you 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.

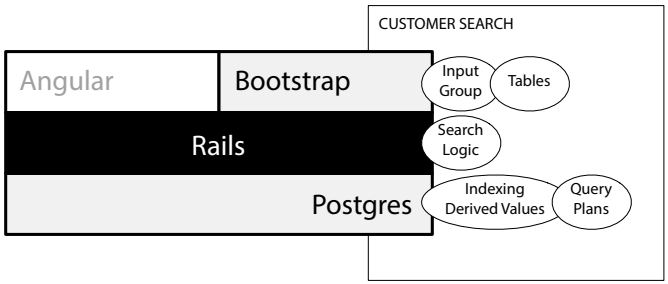
Rails validations are an elegant and powerful way to give users a great experience when providing data via web forms. The validations API is incredibly expressive, configurable, and extensible. If you remove the validations from our code and attempt to register with an invalid email, you'll get an exception—not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

This *does* result in some duplication, which is a chance for inconsistency to creep into our app, but we can fight this by writing tests for each part of the stack (which you do in Chapter 7, *Test This Fancy New Code*, on page 95).

Next: Using Postgres Indexes to Speed Up a Fuzzy Search

The registration and login feature is now secure *and* pleasant to use. By using the best of both Postgres and Bootstrap, you'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, you'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. You'll start this feature by implementing a naive fuzzy search that you can then examine and optimize using special indexes Postgres provides.



CHAPTER 4

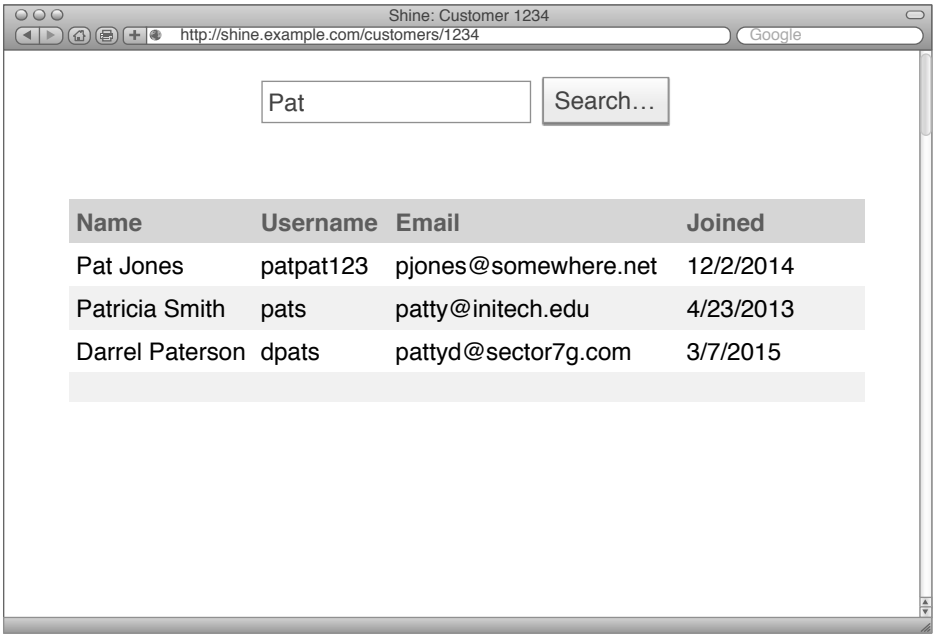
Perform Fast Queries with Advanced Postgres Indexes

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, you'll implement a customer search feature, digging deeper into Postgres and Bootstrap. You'll also start learning AngularJS in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 69.

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

Because this is the first bit of code you're writing for this feature, you'll also need some UI and middleware logic. Although this chapter is mostly about Postgres, you'll be working in all parts of the stack. Specifically, you'll write the basics of our search logic in Rails, and you'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*, on page 57, you'll spend more time on the search results themselves.

First, you'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 the following figure:



After that, you'll look at the performance of Active Record's SQL query using Postgres's *query plan*. You'll then use a special type of index on our tables to speed up our search, which you'll then verify by reexamining the new query plan. 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 get to it by implementing a naive version of the search.

Implementing a Basic Fuzzy Search with Rails

As mentioned in the *Introduction*, on page v, 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 the real world, our database would already exist and you'd hook up to it directly. Since that's not the case, you'll need to simulate its existence by creating the table in Shine's database. And, because you'll use Postgres query performance optimization, our table is going to need a lot of data in it.

Set Up the New Table and Data

If you were using an existing table, you wouldn't need a migration—you 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. It will ultimately look like the schema on page 42.

A customer has first and last names, an email address, a username, and creation and last update dates. None of the fields allow null and the data in the username and email fields must both be unique (that'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.

Because this table doesn't exist yet in our example, you can create it using Rails's model generator. This creates 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/20160718151402_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 and unique constraints. You can add those easily enough by opening up db/migrate/20160718151402_create_customers.rb.

search/setup-customer-data/shine/db/migrate/20160718151402_create_customers.rb

```
class CreateCustomers < ActiveRecord::Migration[5.0]
  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
```

The Final Customer Table Schema

```
                                    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)
```

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

```
$ bundle exec rails db:migrate
== 20160718151402 CreateCustomers: migrating ============
-- create_table(:customers)
   -> 0.0257s
-- add_index(:customers, :email, {:unique=>true})
   -> 0.0026s
-- add_index(:customers, :username, {:unique=>true})
   -> 0.0019s
== 20160718151402 CreateCustomers: migrated (0.0303s) ==
```

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

search/setup-customer-data/shine/Gemfile

```
gem 'devise'
gem 'bower-rails'
➤ gem 'faker'
```

Then, install it:

```
$ bundle install
Installing faker 1.6.5
```

You can now use faker to create real-looking data, which will make it much easier to use Shine in our development environment, since you'll have real-sounding names and email addresses. We'll create this data by writing a small

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

script in db/seeds.rb. Rails's seed data[2] feature is intended to prepopulate 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.to_s +
                "@#{Faker::Internet.domain_name}")
  print '.' if i % 1000 == 0
end
puts
```

The reason we're appending the index to the username and email is to ensure these values are unique. As you added 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, you'll ensure uniqueness with a number. Also note the final line in the loop where you use print if the current index is a multiplle of 1000. This will output a dot on the command line to give you a sense of progress. With the seed file created, we'll run it (this will take a while—possibly 30 minutes—depending on the power of your computer).

```
$ bundle exec rails db:seed
```

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

Build the Search UI

When starting a new feature, it's usually best to start from the user interface, especially if you didn't have a designer design it for you ahead of time. Recall that our requirements are to allow searching by first name, last name, and email address. Rather than require users to specify which field they're 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, add a route to config/routes.rb:

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

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

```
root to: "dashboard#index"
resources :customers, only: [ :index ]
```

That route expects an index method on the CustomersController class, which doesn't exist yet. You could use a Rails generator, but it's just as easy to create the file from scratch. You'll create app/controllers/customers_controller.rb and add the definition of the CustomersController there. You'll implement the index method to just grab the first 10 customers in the database, so you have some data you 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
```

You can now start building the view in app/views/customers/index.html.erb (which won't exist yet). You'll need two main sections in your view: a search form and a results table. First, you should make a header letting users know what page they're on.

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

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

Next, create the search form. Because there's just going to be one field, you don't need an explicit label (though you'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 to 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>
```

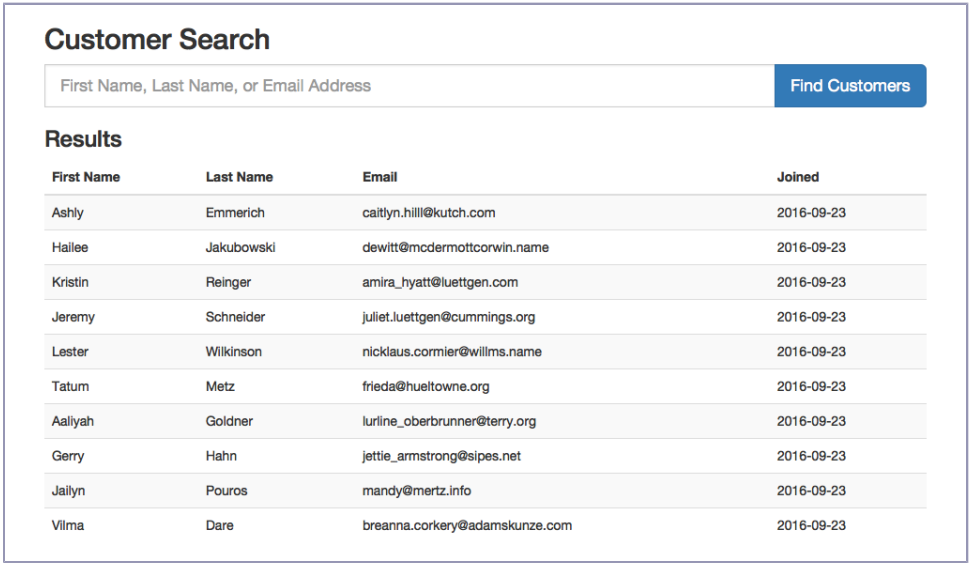
The sr-only class on our label is provided by Bootstrap and means "Screen Reader Only." You should use this on elements that are semantically required (like form labels) but that, for aesthetic purposes, you don't want to be visible. This makes your UI as inclusive as possible to users on all sorts of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

With our search form styled (you'll see what it looks like in a moment), we'll now 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 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 you've got the table styled, you can see the entire view in the following figure. It looks pretty good, and you still haven't had to write any actual CSS.



With the UI built, all you need to do to implement our search is replace the implementation of `index` in `CustomersController` with the actual search.

Implement 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 (for example, we'd search for "pat" if given the term "pat123@example.com").
- If the search term does *not* contain an "@" 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 "Pat" should match "Patty."

- 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 and 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 Active Record 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. You can use SQL like `lower(first_name) LIKE 'pat%'` and you can use complex expressions in the `order by` clause. Ultimately, you'll want a query that looks like this:

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

The `order by` in Postgres can take a wide variety of expressions. According to the documentation, it can "be any expression that would be valid in the query's select list."[3] In our case, we can order fields based on the results of matching the `email` column value to `pat@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 `pat@example.com` first, so we use `desc` to sort email matches first.

To execute this query using Active Record, you'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: "pat%",
                 last_name: "pat%",
                 email: "pat@example.com"
               }).order("email = 'pat@example.com' desc, last_name asc")
```

3. http://www.postgresql.org/docs/9.5/static/queries-order.html

Note that you're appending % to the name search term and using like so that you meet the "starts with" requirement. You have to do this because Active Record 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 you need to where and order.

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 an @, 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.

First, let's create app/models/customer_search_term.rb and add the three attributes as well as an initializer. 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 lowercase 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

```
private

  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 "pat123@example.com" you want to use that exact term for the email part of our search. But because we want rows where first_name or last_name starts with just "pat" 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 you'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. Because search_term contains data provided by the user, it could create an attack vector via SQL injection.[4] To prevent this, you need to SQL-escape search_term before you send it to Postgres for querying. Active Record 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, you 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)}"
```

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

```
  @where_args[:email] = search_term

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

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

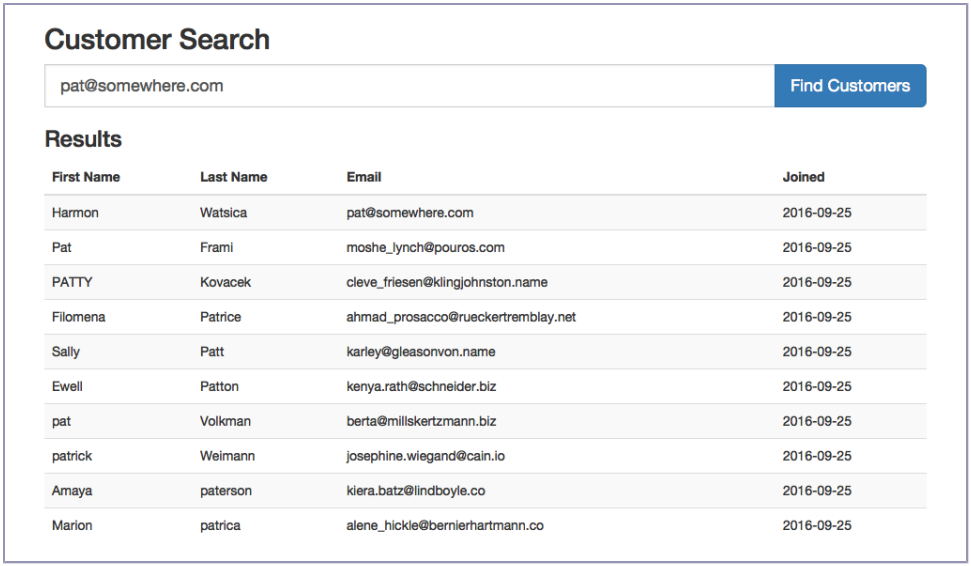
Now that CustomerSearchTerm is implemented, you can use it in CustomersController to implement the search.

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 you might actually want. You might really need something more complex; for example, a search term of "Pat Jones" would result in a first name search for "pat" and a last name search for "jones." The point is that our simplistic search is still too complex for Active Record's API to handle. You had to create your own where clause and your own order by.

Now that our search is implemented, you can see the results by starting our server (via foreman start) and navigating to http://localhost:3000/customers. As you see in the following figure, the email match is listed first, while the remaining ones are sorted by last name.



Depending on the computer you're 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, you'll learn about the powerful indexing features Postgres provides. You'll see that they're 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 Active Record's Migrations API. This API provides a "lowest common denominator" approach. The best we can do is create

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

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 and 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 'pat%' OR
                       lower(last_name)  like 'pat%' OR
                       lower(email)      = 'pat@example.com'
                     ORDER BY
                       email = 'pat@example.com' DESC,
                       last_name ASC ;

                      QUERY PLAN
------------------------------------------------------------------------------
1 Sort  (cost=13930.19..13943.25 rows=5225 width=79)
      (actual time=618.065..618.103 rows=704 loops=1)
   Sort Key: (((email)::text = 'pat@example.com'::text)) DESC, last_name
   Sort Method: quicksort  Memory: 124kB
2    ->  Seq Scan on customers  (cost=0.00..13607.51 rows=5225 width=79) #
                                (actual time=0.165..612.380 rows=704 loops=1)
3          Filter: ((lower((first_name)::text) ~~ 'pat%'::text) OR
                    (lower((last_name)::text) ~~ 'pat%'::text)  OR
                    (lower((email)::text) = 'pat@example.com'::text))
           Rows Removed by Filter: 349296
 Planning time: 1.223 ms
4 Execution time: 618.258 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 you understand how Postgres will execute our query.

❶ Here, Postgres is telling us that it's sorting the results, which makes sense since we're using an order by clause in our query. The details (for example, 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 the 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 more than half a second. That's not 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 you can create an index on the lowercased value for each of our three fields. Further, you can configure the index in a way that allows Postgres to optimize for the "starts with" search you are doing. Here's the basic syntax:

```
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 you're using a `like` in your search, you need to use the nonstandard operator class `varchar_pattern_ops`. You can read more about operator classes in Postgres's documentation.[7]

Now that you've seen the SQL needed to create these indexes, you need to adapt them to a Rails migration. Rails doesn't provide a way to do this with Active Record's migrations API, but it *does* provide a method, `execute`, which executes arbitrary SQL. Let's create the migration file using Rails's generator.

7. http://www.postgresql.org/docs/9.5/static/indexes-opclass.html

```
$ bundle exec rails g migration add-lower-indexes-to-customers
    invoke  active_record
    create    db/migrate/20160721030725_add_lower_indexes_to_customers.rb
```

Next, edit the migration to add the indexes. Rails 5 added the ability to create these Postgres-specific indexes using `add_index`. Previous versions of Rails required using `execute` and typing the `CREATE INDEX` SQL directly.

search/add-indexes/shine/db/migrate/20160721030725_add_lower_indexes_to_customers.rb

```
class AddLowerIndexesToCustomers < ActiveRecord::Migration[5.0]
  def change
    add_index :customers, "lower(last_name) varchar_pattern_ops"
    add_index :customers, "lower(first_name) varchar_pattern_ops"
    add_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, let's run this migration (it may take several seconds due to the volume of data being indexed).

```
$ bundle exec rails db:migrate
== 20160721030725 AddLowerIndexesToCustomers: migrating ===========
-- add_index(:customers, "lower(last_name) varchar_pattern_ops")
   -> 0.5506s
-- add_index(:customers, "lower(first_name) varchar_pattern_ops")
   -> 0.4963s
-- add_index(:customers, "lower(email)")
   -> 7.1292s
== 20160721030725 AddLowerIndexesToCustomers: migrated (8.1763s) ==
```

Before you try the 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) like 'pat%' OR
                       lower(last_name)  like 'pat%' OR
                       lower(email)      = 'pat@example.com'
                     ORDER BY
                       email = 'pat@example.com' DESC,
                       last_name ASC
                     ;
                        QUERY PLAN
------------------------------------------------------------------------
Sort  (cost=5666.10..5679.16 rows=5224 width=79)
```

```
   (actual time=14.467..14.537 rows=704 loops=1)
Sort Key: (((email)::text = 'pat@example.com'::text)) DESC, last_name
Sort Method: quicksort  Memory: 124kB
->  Bitmap Heap Scan on customers
      (cost=145.31..5343.49 rows=5224 width=79)
      (actual time=0.387..8.650 rows=704 loops=1)
    Recheck Cond: ((lower((first_name)::text) ~~ 'pat%'::text) OR
                   (lower((last_name)::text) ~~ 'pat%'::text)  OR
                   (lower((email)::text) = 'pat@example.com'::text))
      Filter: ((lower((first_name)::text) ~~ 'pat%'::text) OR
               (lower((last_name)::text) ~~ 'pat%'::text)  OR
               (lower((email)::text) = 'pat@example.com'::text))
      Heap Blocks: exact=655
      ->  BitmapOr  (cost=145.31..145.31 rows=5250 width=0)
                     (actual time=0.263..0.263 rows=0 loops=1)
          -> Bitmap Index Scan on
              index_customers_on_lower_first_name_varchar_pattern_ops
                (cost=0.00..41.92 rows=1750 width=0)
                (actual time=0.209..0.209 rows=704 loops=1)
                Index Cond: (
                  (lower((first_name)::text) ~>=~ 'pat'::text) AND
                  (lower((first_name)::text) ~<~ 'pau'::text))
          -> Bitmap Index Scan on
              index_customers_on_lower_last_name_varchar_pattern_ops
                (cost=0.00..41.92 rows=1750 width=0)
                (actual time=0.007..0.007 rows=0 loops=1)
                  Index Cond: (
                    (lower((last_name)::text) ~>=~ 'pat'::text) AND
                    (lower((last_name)::text) ~<~ 'pau'::text))
           ->  Bitmap Index Scan on index_customers_on_lower_email
                 (cost=0.00..57.55 rows=1750 width=0)
                 (actual time=0.046..0.046 rows=0 loops=1)
                   Index Cond: (
                     lower((email)::text) = 'pat@example.com'::text)
Planning time: 0.193 ms
Execution time: 14.732 ms
```

This time, there is *more* gobbledegook, but if you look closely, `Seq Scan on customers` is gone, and you can see a lot of detail around our `where` clause. The highlighted lines indicate *index scans*, in contrast to the Seq Scan you saw before. And the index scan is using our index and thus *not* examining each row in the table to find the correct results. You 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, you can see that the results are about 40 times faster—the query should complete in under 15 milliseconds!

If you 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 40, all with just a few lines of SQL in a migration. *And* you didn't have to change a line of code in the Rails application. If you were using a less powerful database, you'd need to set up new infrastructure for making this search fast, and that could have a significant cost to development, maintenance, and production support.

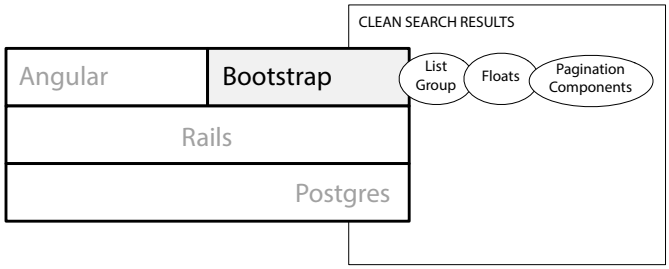
This sort of index is just the tip of the iceberg—Postgres has many advanced features.

With our search 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 you stopped now and shipped what you have, you'd be shipping a feature you could be proud of. But, because you 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: Better-Looking Results with Bootstrap's List Group

You've got a solid back end going for our search. It's now really fast and you didn't have to do anything other than add a few indexes to your 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 you back to the front end as you redesign the results. You'll see that Bootstrap's many helper classes and components can make it easy to try out new designs. This means you can provide better software for your users without investing huge amounts of time in writing CSS.



CHAPTER 5

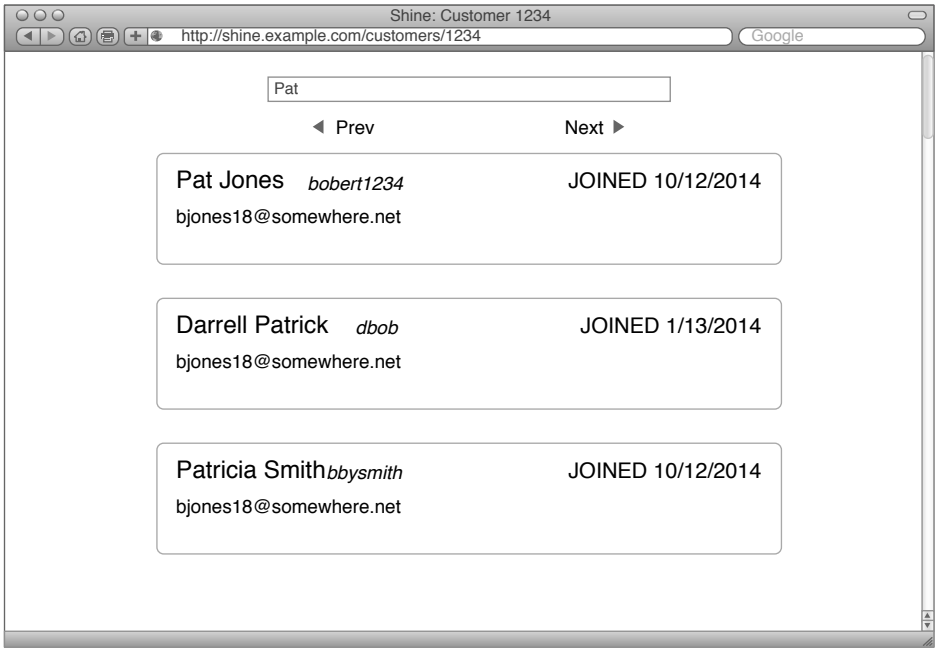
Create Clean Search Results with Bootstrap Components

The customer search you'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 a lot more options than tables to get great-looking results. That's what you'll learn in this chapter. You're going to re-style your results to do two things: first, you'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 a seemingly complex design by using Bootstrap's *list group*, its typographic styles, and CSS floats. You're also going to paginate the results using one of Bootstrap's custom components. And you aren't going to write *any* CSS.

The UI you'll build will look like the figure that follows. Let's start by removing the table and replacing it with component-based results.



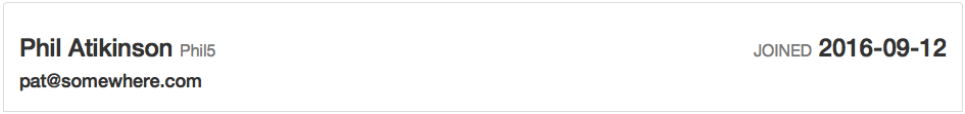
Creating Google-Style Search Results Without Tables

If you were styling this application on your own, the prospect of building a customized, nontabular search results page (like Google's, for example) would not be very appealing. You'd need to figure out the layout, design, and CSS to get it just right.But because you're probably always under pressure to ship your software and move on to the next feature, you 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 *attempt* something different, without a huge time investment. Let's try using Bootstrap's 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 Pat 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?



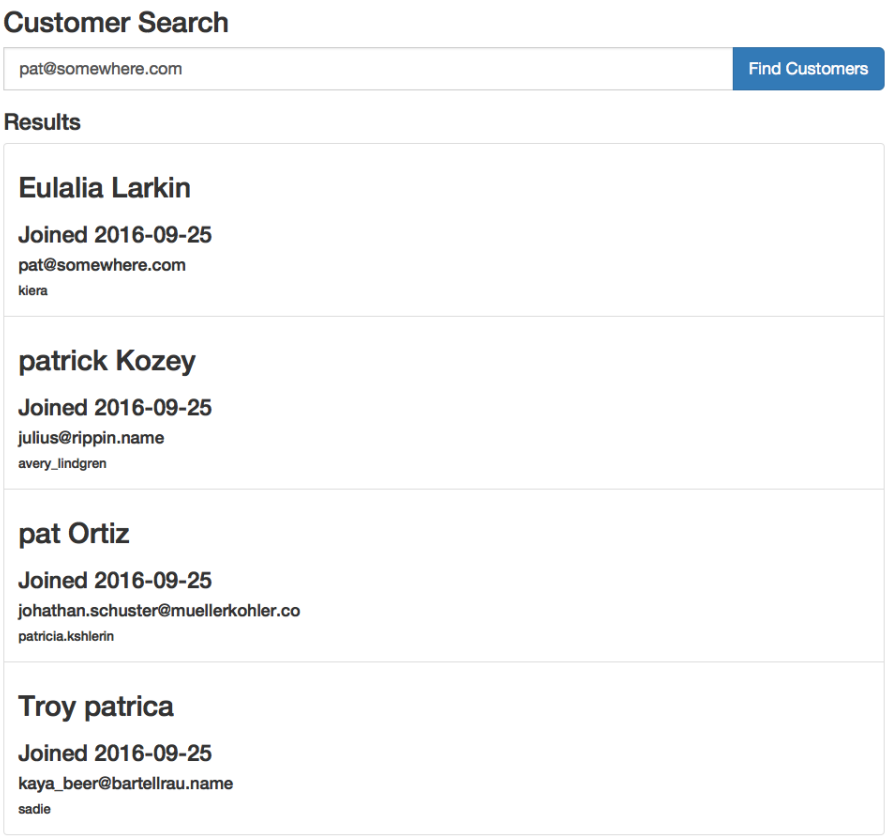
Unless you write a lot of CSS, this layout might appear to be somewhat tricky to pull off, especially if you consider how well aligned all the subcomponents are. Fortunately, Bootstrap makes this simple. You'll use three features of Bootstrap to do this: the list group component, the behavior of typography 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, you'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, you'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

```
  <!-- Existing header and search form -->

<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 as you can see in the following figure, our design is starting to form, as Bootstrap's list group does a reasonable job formatting the information.



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

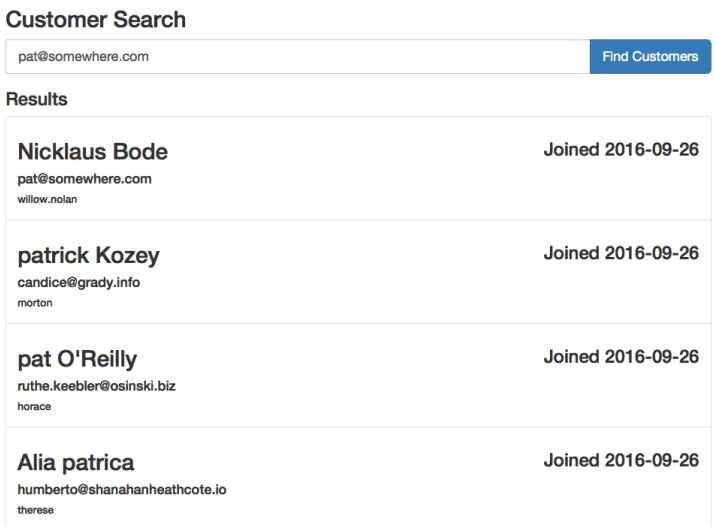
search/list-group-positioned/shine/app/views/customers/index.html.erb

```
  <!-- Existing header and search form -->

<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 you 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 you don't reset them (it's hard to explain, but try removing the clearfix class and see what happens). Now, take a look at the following figure. Our design is pretty close to what we want to achieve.



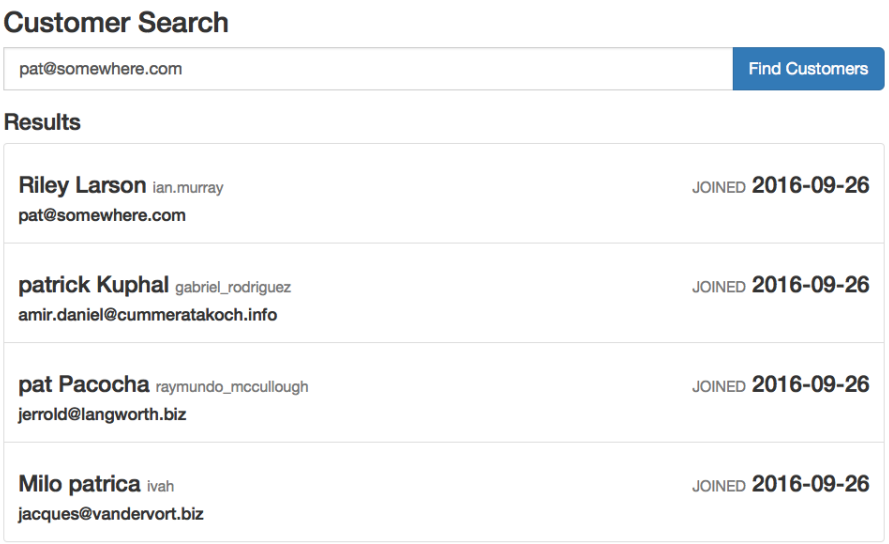
The last thing you need to do is reduce the visual weight of both the username as well as the label "Joined." This allows the other information to be highlighted in a subtle way. To do that, you'll put both elements inside `small` tags, which will trigger alternate typography from Bootstrap. You'll also 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

```
  <!-- Existing header and search form -->

<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 you'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 is much trickier than you've seen before, but Bootstrap made it simple to achieve, and you *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 you 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 our users can see only ten results at a 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 not that much code to just do it ourselves. Since you'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.

Handle Pagination in the Controller

For simplicity, let's hard-code the size of a page to ten 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

    # ...
```

Next, use both PAGE_SIZE and @page to construct parameters to Active Record's offset and limit methods. Because our results are sorted, you 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

```
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).
    offset(PAGE_SIZE * @page).limit(PAGE_SIZE)
else
  @customers = []
end
```

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

Add Pagination Controls to the View

To keep things simple, we'll go with previous/next pagination. This means you'll need two links on the page, which you 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 you can use, called a *pager*. Let's set it up in a partial, which you'll then use to place the pager before *and* after the results list. (This allows the user to always have the pager handy.) I've highlighted the markup and classes Bootstrap requires to style the pager. Note that I'm omitting the previous link if the user is on the first page. Bootstrap includes a disabled class you could use to render the button in a disabled state, but it requires more complexity on the Rails side to make it work, so I'll just omit the link entirely to make it simpler. (Omitting the

"Next" link when we've reached the end of the results is even more complex, since you have to do a seperate count of the number of results or otherwise pass a flag to the front-end to indicate that there aren't any more results.)

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

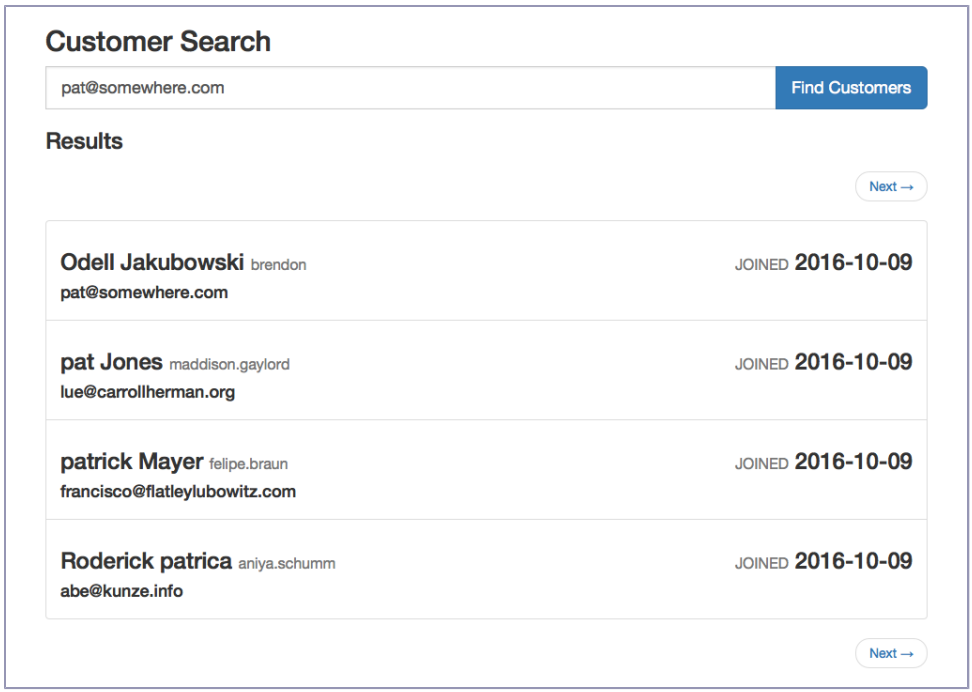
```
<nav>
  <ul class="pager">
    <% if page > 0 %>
      <li class="previous">
        <%= link_to "&larr; Previous".html_safe,
              customers_path(keywords: keywords, page: page - 1),
              title: "Previous Page" %>
      </li>
    <% end %>
    <li class="next">
      <%= link_to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

Now, let's include the partial in app/views/customers/index.html.erb.

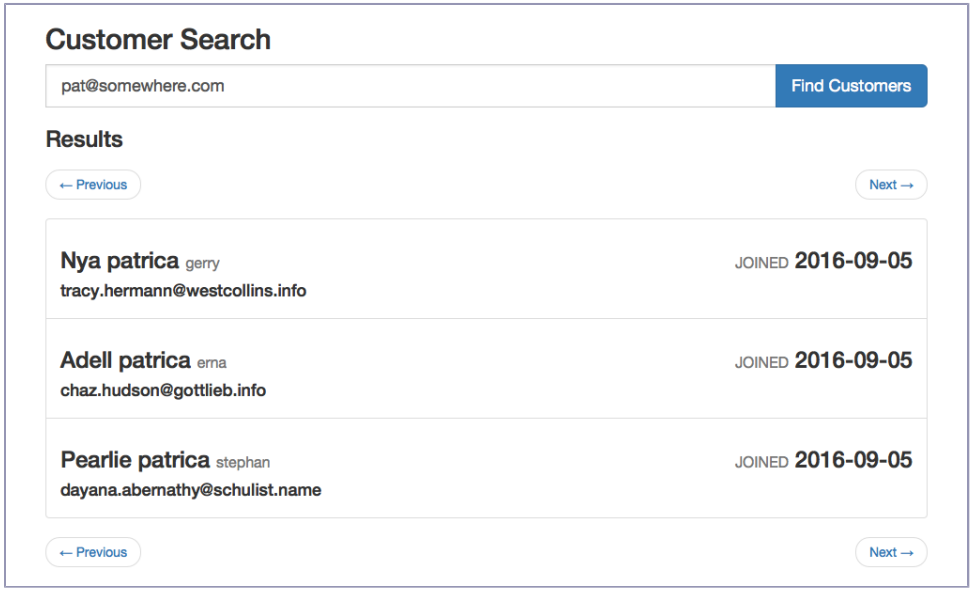
search/pagination/shine/app/views/customers/index.html.erb

```
  <header>
    <h1 class="h3">Results</h1>
  </header>
  <% if @customers.present? %>
    <%= render partial: "pager",
                locals: { keywords: @keywords, page: @page } %>
  <% end %>
  <ol class="list-group">

    <!-- ... -->
  </ol>
  <% if @customers.present? %>
    <%= render partial: "pager",
                locals: { keywords: @keywords, page: @page } %>
  <% end %>
</section>
```

If you start your server and search, you'll now only see ten results. You can see our pager control at the top and bottom and, because this is the first page, the "Prev" link is missing:



If you click "Next," you'll see the second page of results:



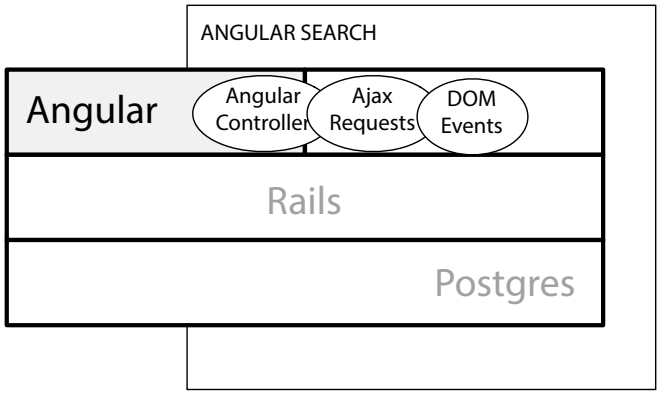
Because of our indexes, Postgres's powerful implementation of `order by`, and Bootstrap's premade components, you were able to add performant pagination in just a few lines of code.

Next: Angular!

In the next chapter, you'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, you'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.



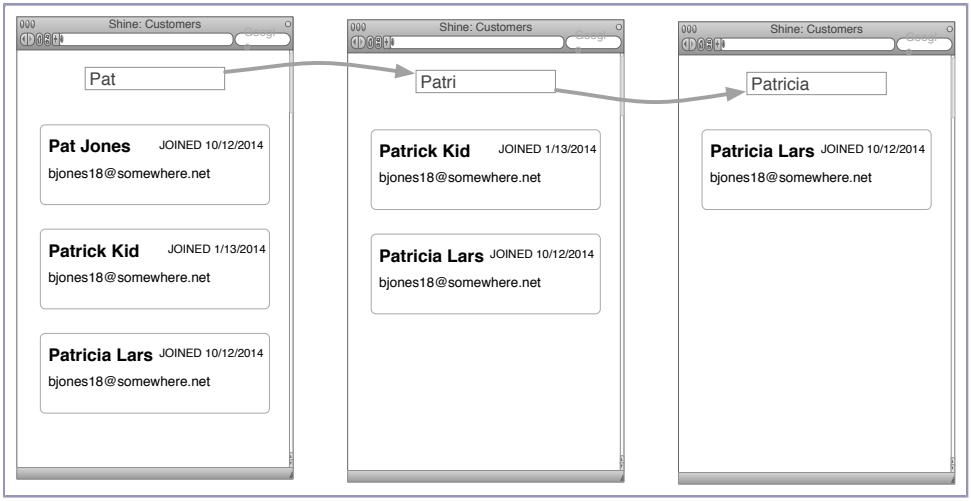
CHAPTER 6

Build a Dynamic UI with AngularJS

You'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.

In this chapter you're going to rework our search feature so that the system searches as our users type, dynamically changing the results as they type out a user's name. For example, if a user wants to find a customer named Patricia Smith, a search for just Pat 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 keep typing Patricia and potentially get the desired record right away, without having to wait for the browser to re-render the view. The figure that follows shows a mock-up of how it will work.

Angular makes it easy to implement this feature. Since you've already set up Webpack in *Setting Up Bootstrap with NPM and Webpack*, on page 6, we can focus here on getting Angular configured in Shine. Because this part of our Rails application is going outside what Rails gives us, it's important to take small, verifiable steps as we set everything up. That way, if something goes wrong, you'll know where.



You'll do this by creating a basic Angular *app* that does nothing more than validate our configuration. You'll then use Angular to power our existing search feature, maintaining the existing user experience of entering a search term and clicking a button. This allows you to learn some of Angular's concepts and features in isolation. Finally, you'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

As I mentioned earlier, using Angular 2 with the Rails Asset Pipeline is difficult. In the previous edition of this book, I used Bower to download Angular, and Sprockets (the library that powers Rails' Asset Pipeline) automatically brought it in. Angular 2 is not going to be available via Bower, so we have to use NPM (that's why we set it up in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page 17). Configuring the Asset Pipeline to use NPM-based libraries and packages is not simple, mainly due to the assumptions NPM modules make about the JavaScript environment they are running in.

Modern JavaScript is *modular*. This means that it's stored as a series of modules, located in various files, the same way you manage your Ruby code. Sprockets does not support modular JavaScript, so while you could coerce

Sprockets into loading a JavaScript file you downloaded with NPM, it won't understand the various module-based functions and conventions. What all this means is that we need another solution. In our case, that's Webpack, which you've already set up for just this purpose.

This is not ideal. It's obviously better to have an all-in-one solution to this provided by Rails. Fortunately, Rails design is convention *over* configuration, not convention *instead of* configuration, so you can make everything work with just a little bit of elbow grease. Part of being an effective full-stack developer is being able to use disparate technologies to achieve great things, and that's what we're going to do here. The setup *is* going to be complex, but once we do it, our day-to-day development will be simple and straightforward.

Install Angular

To install Angular, you'll need to add several packages to `package.json`. Note that you can't just add something like `"angular"` and call it a day. Angular is highly modular, and requires you to opt-in to almost every feature outside of what it considers its core. Further, Angular depends on some non-Angular libraries, which you'll also have to explicitly include because of how NPM handles transitive dependencies. Finally, note that you're setting the versions explicitly here. This is so your experience following along matches mine.

typeahead/install-angular/shine/package.json

```
{
  "name": "shine",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    "stats-webpack-plugin": "^0.2.1",
    "webpack": "^1.9.11",
    "webpack-dev-server": "^1.9.0",
    "css-loader": "^0.23.1",
    "file-loader": "^0.9.0",
    "style-loader": "^0.13.1",
    "url-loader": "^0.5.7",
    "bootstrap": "3.3.7",
    "@angular/common": "^2.0.0",
    "@angular/compiler": "^2.0.0",
    "@angular/core": "^2.0.0",
    "@angular/forms": "^2.0.0",
    "@angular/http": "^2.0.0",
    "@angular/platform-browser": "^2.0.0",
    "@angular/platform-browser-dynamic": "^2.0.0",
    "@angular/router": "^3.0.0",
    "systemjs": "0.19.27",
    "core-js": "^2.4.1",
```

```
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "^0.6.23"
  }
}
```

Once you've added all those dependencies, you can bring them in with `npm install`.

Next, let's validate that all this setup was done correctly before you start working on the typeahead feature. You can do that by creating a very simple Angular app that, if it works, means our configuration is good.

Validate the Angular Install with a Basic App

Before jumping into a new piece of technology, it's always a good idea to write something simple with it so that you can validate that all the configuration and plumbing is hooked up properly. Here, you'll create a simple page containing a text field that, when you type into it, reflects the value you've typed inside a header tag that's also on the page.

To do this, first create a new route to `/angular_test` inside `config/routes.rb`:

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

```
Rails.application.routes.draw do
  devise_for :users
  root to: "dashboard#index"
  resources :customers, only: [ :index ]
➤ get "angular_test" => "angular_test#index"
end
```

Then create a controller in `app/controllers/angular_test_controller.rb`:

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

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

Finally, you'll need a view that has enough markup that our Angular app can know where to insert itself. (I'll explain what this means a little bit later in this chapter.) Create the file `app/views/angular_test/index.html.erb` like so:

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

```
<section id="angular-test">
  <h1>Angular Test</h1>
  <shine-angular-test></shine-angular-test>
</section>
```

This markup looks a little strange. Namely, there's a non-standard tag called shine-angular-test. This is where Angular will render its view, and I'll show you later how that happens. Note also that you've given the top-most section an explicit id. You'll use this as a signal to Angular to bootstrap itself only when this Rails view is being shown.

Now, let's write the Angular code. You're not going to get too deep into what's going on here, as that's covered in the next section, so just view this is something to validate your setup before you start learning Angular.

All the code you're writing will go in webpack/application.js. After removing the call to console.log, you'll bring Angular and its dependencies into our file using the require function that Webpack provides. Note that you're putting all of Angular's libraries inside an object called ng. This isn't required, but a lot of documentation online assumes this is where the Angular libraries are, so it'll make it easier when you have to look something up.

typeahead/install-angular/shine/webpack/application.js

```
var coreJS          = require('core-js');
var zoneJS          = require('zone.js');
var reflectMetadata = require('reflect-metadata');
var ng              = {
  core:                   require("@angular/core"),
  common:                 require("@angular/common"),
  compiler:               require("@angular/compiler"),
  forms:                  require("@angular/forms"),
  platformBrowser:        require("@angular/platform-browser"),
  platformBrowserDynamic: require("@angular/platform-browser-dynamic"),
  router:                 require("@angular/router")
};
```

Also note that even though you aren't using many of these modules, the mere act of requiring them performs some necessary background configuration. This is not Rails-like. In Rails, you never have to do stuff like this, but the ecosystem for front-end development tends to favor explicit configuration over convention. There is some virtue in this, in that you tend to see everything that's happening in your app right there in your code or configuration.

Now, you can set up the code for our Angular test app. To do this, you need to do three things: (1) create a top-level component to render our app; (2) create a top-level module that references that component to represent and configure our app; and (3) bootstrap that module to start everything up. Let's look at the component first.

In Angular, a *component* can be thought as the model, view, and controller all wrapped up into one. A component holds the view template, the data that

template needs to render itself, and any functionality needed to respond to user actions.

In the case of our test app, our component, called AngularTestComponent is fairly small (and don't worry about some of the odd attributes like *ngIf you'll see in the template—I'll explain those in the next section).

typeahead/install-angular/shine/webpack/application.js

```
var AngularTestComponent = ng.core.Component({
  selector: "shine-angular-test",
  template: '\
  <h2 *ngIf="name">Hello {{name}}!</h2> \
  <form> \
    <div class="form-group"> \
      <label for="name">Name</label> \
      <input type="text" id="name" class="form-control" \
             name="name" bindon-ngModel="name"> \
    </div> \
  </form> \
  '
}).Class({
  constructor: function() {
    this.name = null;
  }
});
```

An Angular app also needs to be able to configure aspects of its behavior globally. Angular provides the ability to do this via the NgModule function. You'll learn more about this as you work, but for now you need to create our top-level module and connect it to our top-level component:

typeahead/install-angular/shine/webpack/application.js

```
var AngularTestAppModule = ng.core.NgModule({
  imports: [ ng.platformBrowser.BrowserModule, ng.forms.FormsModule ],
  declarations: [ AngularTestComponent ],
  bootstrap: [ AngularTestComponent ]
})
.Class({
  constructor: function() {}
});
```

Next, you need to tell Angular to start up our app by passing our top-level module to platformBrowserDynamic.platformBrowserDynamic().bootstrapModule(). (In Angular 1, you could do this using the ng-init attribute in our HTML, but this feature is not present in Angular 2.) Be careful in doing this, however. First, you need to wait for the DOM to load, which you can do by waiting for the DOMContent-

Loaded[1] event that the browser's JavaScript engine will fire. You also only want to bootstrap our Angular app if you're on the /angular_test page you set up in Rails. To do that, you'll check for the existence of the id you used, angular-test, like this:

typeahead/install-angular/shine/webpack/application.js

```
document.addEventListener('DOMContentLoaded', function() {
  var shouldBootstrap = document.getElementById("angular-test");
  if (shouldBootstrap) {
    ng.platformBrowserDynamic.
      platformBrowserDynamic().
      bootstrapModule(AngularTestAppModule);
  }
});
```

The reason you have to do this is that Angular does not assume it's running inside a browser. It's possible to run Angular on the server side to pre-render views (which I won't be doing in order to keep things simple). So, Angular needs to be told explicitly that you are running in a browser and to boot itself up for that environment.

All this was a one-time setup. You'll only need to revisit this type of code if you are bringing in new libraries or making major changes to your app's configuration.

Now, start up Shine with foreman start and navigate to http://localhost:5000/angular_test. You should see this:

Type "Pat" into the text field. You should see the header appear and update as you type, ultimately looking like so:

Whew! That was a lot of setup. From here on out, almost all of the JavaScript code you write will be Angular code for Shine. Let's get to that by porting our

1. https://developer.mozilla.org/en-US/docs/Web/Events/DOMContentLoaded

existing search functionality over to Angular. This will motivate us to learn exactly how Angular works and what some of its basic concepts are.

Porting Our Search to Angular

Although Angular works best when you 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 the 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 feels straightforward, at least conceptually. Again, this isn't the best demonstration of Angular's power, but it's simple enough to get your feet wet with some of Angular's concepts. You'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.

The most important concept in Angular is *components*. Previous versions of Angular (as well as many other front-end frameworks) view the JavaScript app as a series of models, views, and controllers (much like how a Rails application is organized). Angular 2, however, views the app as a series of components. A components can be thought of as a model, view, and controller all wrapped up into one. In Angular 2, a component is, at the very least, a view template and a class. The class contains data and functions available to the view template. The component can then be used anywhere its selector is written.

In the previous section, when you put `<shine-angular-test></shine-angular-test>` in your markup, that was using the component you'd created (called `AngularTestComponent`). Here, you'll create a component for our customer search.

First, you'll set up an empty component that just renders markup. Next, you'll write some JavaScript that shows us how to respond to a click event generated by clicking the search button. Then, you'll update that JavaScript to put canned data into the view when the user does a search, thus demonstrated how to manipulate the DOM. Finally, you'll change our code to get real results from the server by making an Ajax call.

Set Up the Empty Component

Start the same way you did with our Angular test app by replacing the contents of app/views/customers/index.html with the markup needed to bootstrap our Angular app. Assume that the top-level component you're creating will be called shine-customer-search, and use an id attribute so that you know when to bootstrap the new Angular app you're creating.

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

```
<section id="shine-customer-search">
  <shine-customer-search></shine-customer-search>
</section>
```

Prefixing selector names

Angular recommends all selector names have a prefix to ensure that there are no name clashes when you bring in other components or when the HTML spec evolves. We'll use shine- as the prefix to our selector names.

Next, create a shell for our new component in webpack/application.js called CustomerSearchComponent:

```
var CustomerSearchComponent = ng.core.Component({
  selector: "shine-customer-search",
}).Class({
  constructor: function() {
  }
});
```

Next, copy the markup from our Rails view into the template: key of the object passed to ng.core.Copmonent. Note that you've removed all Rails helpers and converted everything to plain HTML, hard-coding a single search result. You also removed the pagination, which, by the end of the chapter, you will be able to add back as an exercise if you like. Also note that each line ends with a backslash (\ ) so that you can have a multi-line string. (See *Why are we using ES5 and not TypeScript?*, on page 79 for some details about why this is.)

typeahead/angularized-search/shine/webpack/application.js

```
var CustomerSearchComponent = ng.core.Component({
  selector: "shine-customer-search",
  template: '\
<header> \
  <h1 class="h2">Customer Search</h1> \
</header> \
<section class="search-form"> \
  <form> \
    <div class="input-group input-group-lg"> \
```

```
      <label for="keywords" class="sr-only">Keywords></label> \
      <input type="text" id="keywords" name="keywords" \
             placeholder="First Name, Last Name, or Email Address"\
             class="form-control input-lg">\
      <span class="input-group-btn"> \
        <input type="submit" value="Find Customers"\
               class="btn btn-primary btn-lg">\
      </span> \
    </div> \
  </form> \
</section> \
<section class="search-results"> \
  <header> \
    <h1 class="h3">Results</h1> \
  </header> \
  <ol class="list-group"> \
    <li class="list-group-item clearfix"> \
      <h3 class="pull-right"> \
        <small class="text-uppercase">Joined</small> \
        2016-01-01\
      </h3> \
      <h2 class="h3"> \
        Pat Smith\
        <small>psmith34</small> \
      </h2> \
      <h4>pat.smith@example.com</h4> \
    </li> \
  </ol> \
</section> \
  '
}).Class({
  constructor: function() {
  }
});
```

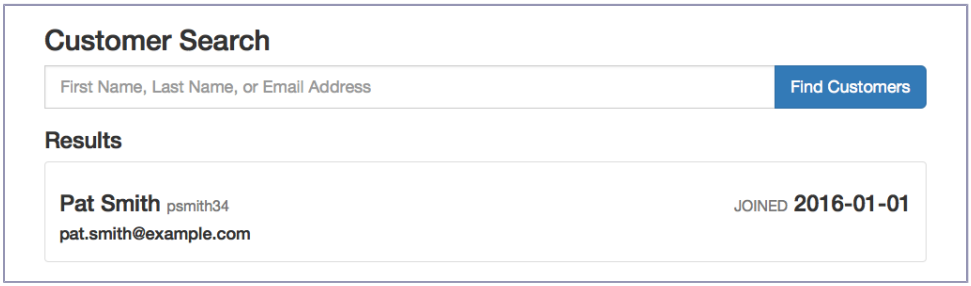
Finally, create our top-level module and bootstrap it using platformBrowserDynamic.platformBrowserDynamic().bootstrapModule():

typeahead/angularized-search/shine/webpack/application.js

```
var CustomerSearchAppModule = ng.core.NgModule({
  imports: [ ng.platformBrowser.BrowserModule, ng.forms.FormsModule ],
  declarations: [ CustomerSearchComponent ],
  bootstrap: [ CustomerSearchComponent ]
})
.Class({
  constructor: function() {}
});

document.addEventListener('DOMContentLoaded', function() {

  if (document.getElementById("shine-customer-search")) {
    ng.platformBrowserDynamic.
```

```
      platformBrowserDynamic().
      bootstrapModule(CustomerSearchAppModule);
  }
});
```

If you run our Rails application (remember to use foreman start) and navigate to http://localhost:5000/customers, you should now see our search form and one result, all rendered by Angular:



Now that you've got Angular rendering our markup, let's make it dynamic. You'll write code to respond to a click even in our search form and have that populate the results list with some canned results.

Why are we using ES5 and not TypeScript?

Angular 2 is written in *TypeScript*,[a] which is a superset of JavaScript that supports static typing. There are many advantages to this, and the toolchain we've set up using Webpack makes it easy for you to use TypeScript later, if you choose. However, the conceptual overhead of learning a new language on top of a new front-end framework is too much.

Fortunately, the Angular team plan to support JavaScript/ES5 (as well as Dart[b]), so there's no requirement you use TypeScript if you want all that Angular 2 has to offer. There is a trade-off to using plain JavaScript, however. Granted, you get to use a language you already are familiar with, but you also have to write a bit more code and deal with our templates in a less-than-ideal way.

In TypeScript, an Angular component can be created like so:

```
@Component({
  selector: "shine-angular-test"
  template: `
  <header>
    <h1>Angular Testing</h1>
  </header>
  `
}
export class AngularTestComponent {
}
```

To use ES5, you can't use the `@Component` annotation and must use `ng.core.Component`. You can't use the `class` keyword and must use the Angular-provided `Class` function. Worst of all, you can't use backticks to create a multi-line string for your template, and must use a trailing backslash on each line (you'll see later how to store our templates in external files, so this problem will go away).

a. https://www.typescriptlang.org
b. https://www.dartlang.org

Respond to Click Events

If you've written dynamic user interfaces with JavaScript before, the overall mechanics of what we're doing will be familiar. You'll add a click handler to the "Find Customers" button in our search form, and you'll then arrange to render the results template once for each result you get back. For now, the results will be a hard-coded array in our JavaScript code.

First, you need to set the click event. The way Angular does this (and, in fact, the way all of our interactions with the DOM work) is by providing a way to set a *property* on a DOM element. Many of these properties are part of the JavaScript specification and are implemented by your browser. Angular adds some of its own properties as well, but the overall model is the same: the markup in your view template uses attributes to set properties.

In the case of responding to clicks, you want to connect the standard `click`[2] property to a function you write that fetches the search results. Let's assume that function is called `search()`. You'd then modify the `<input type="submit" ... >` in our search form like so:

```
<input type="submit" value="Find Customers"\
       class="btn btn-primary btn-lg" \
       on-click="search()"> \
```

Note that you didn't write `click="search()"`, but instead used the prefix `on-`. This tells Angular exactly *how* you want your code connected to the standard `click` property. In this case, we want user actions to be sent back to our code.

This arrangement is called a *binding*, and this type of binding is called *one-way* from the "view target to data source," to use Angular's parlance.[3] Practi-

2. https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/click
3. https://angular.io/docs/ts/latest/guide/template-syntax.html#binding-syntax

cally speaking, this means that when the user clicks the submit button, Angular will call the search function (which you'll see in a moment).

Our search function can't work without the search term the user has entered, so you'll need to find a way to access that. Angular provides such a way via a custom property called ngModel. You need to bind that property to a value you have access to.

You can't write ngModel="keywords" as this would not establish a dynamic binding (instead this would render the initial value of keywords in the text field only). Further, on-ngModel also won't work, due to the implementation details of ngModel. Instead, you need a *two-way binding* so that our code is updated when the value changes, and the view updates if the code changes the value. You can do that by using bindon-ngModel like this:

```
<input type="text" id="keywords" name="keywords" \
       placeholder="First Name, Last Name, or Email Address"\
       class="form-control input-lg" \
       bindon-ngModel="keywords"> \
```

Now, our markup is in place to make the current value of the user's search term available via keywords and to call the function search whenever the "Find Customers" button is clicked. Let's see where those two things are defined.

If you recall when you set up our CustomerSearchComponent, you wrote this code using the Class function:

```
}).Class({
  constructor: function() {
  }
})
```

This created a class to go along with our component, and it's this class where properties and functions can be defined so that they can be referenced in the view template. In other words, this is where you define search as well as where you define keywords.

```
}).Class({
  constructor: function() {
    this.keywords = null;
  },
  search: function() {
    alert("Searched for: " + this.keywords);
  }
})
```

Because JavaScript has no official way to define a class, you're using the mechanism Angular provides. The previous code would be similar to the following Ruby code:

```
class CustomerSearchComponent
  def initialize
    @keywords = nil
  end

  def search
    puts "Searched for " + @keywords
  end
end
```

With this definition of `search`, you should be able to reload the app, type in a search term, click "Find Customers," and see a JavaScript alert with your search terms in them. All in all, this wasn't that much code to write. You specified the backing model for the text field and then wrote a small function to use that value.

All that's left is to populate the search results. Currently, you've hard-coded one result in our template. What we want is to render the search result markup (the content inside and include the `li`) once for each result. You saw that in Rails where you used a call to `each` inside our ERB template. In Angular, you do this with `ngFor`.

Populate the DOM Using ngFor

Manipulating the DOM for our results requires two things. First, you need to iterate over the results and render on `li` for each one. Second, you need to render properties of our customer objects inside our template. We'll start with iterating over the results.

The syntax you're about to see is quite verbose. Although you'll be using a much more compact version in the end, it's important to at least see what the full, explicit, and expanded version looks like. This will provide insight into how Angular works, which may help make future concepts seem less complicated.

Angular uses the standard `template` element[4] to implement control structures like loops. Inside the `template` tag, you'll set the `ngFor` property, which tells Angular the contents of the template should be rendered in a loop. You'll bind the `ngForOf` property to the list of results using a one-way binding from "data

4. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template

source to view target" (the opposite of what you did with the click property). This is done by prefixing the property with bind-:

```
<template ngFor bind-ngForOf="customers">
  <!-- template -->
</template>
```

The ngFor property tells Angular that you are creating a loop, and the bind-ngForOf tells it what you're looping over. Whatever markup is inside the template tags will be rendered once for each element in customers. Of course, you need access to the element during each iteration so you can render its data in the template. You can do that by telling Angular what name you'd like, much how you use the pipes in a call to each in Ruby:

```
@customers.each do |customer|
end
```

In Angular, you can do that by setting an attribute for our variable name, prefixed with let-. This is one of the special prefixes Angular looks for, much like the way you used bind-, on-, and bindon- earlier. The reason it's "let" is because this is the keyword[5] introduced in the latest version of JavaScript that declares a block-scoped variable (which is basically what you are doing here). Putting it all together, here is our loop:

```
➤ <template ngFor bind-ngForOf="customers" let-customer> \
    <!-- template --> \
  </template> \
```

As I said, this loop is verbose, but it's important to know how it actually works. Angular provides a much more compact syntax for this. Instead of using a templates element and several different properties, you can use this syntax:

```
<li *ngFor="let customer of customers" \
    class="list-group-item clearfix"> \
    \
    <!-- REST OF MARKUP --> \
    \
</li> \
```

The asterisk is what Angular uses to enable this syntactic sugar. Note that the contents of *ngFor are *not* a programming language, so what you are allowed to do in there is very limited. The documentation[6] has more details.

5. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
6. https://angular.io/docs/ts/latest/api/common/index/NgFor-directive.html

With our loop sorted out, you just need to render the various parts of each customer in the results template. In ERB, you would use something like <%= customer.first_name %>. In Angular, you use double-braces: {{customer.first_name}}. These work just like their counterparts in ERB, by evaluating the contents and converting the result to a string. Here is what our complete loop looks like:

```
<li *ngFor="let customer of customers" \
    class="list-group-item clearfix"> \
  <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> \
```

The last thing to do is implement search() so it sets the customers property, which will cause Angular to re-render the view and display the results. To do this, you need to establish that property in our class, and then set it inside search. You'll initialize customers to null in the constructor function, the same as you did with keywords. You'll implement search to set customers to some canned results if the search term is "pat" and an empty list otherwise.

typeahead/canned-results/shine/webpack/application.js

```
constructor: function() {
  this.customers = null;
  this.keywords  = "";
},
search: function() {
  if (this.keywords == "pat") {
    this.customers = RESULTS;
  }
  else {
    this.customers = [];
  }
}
```

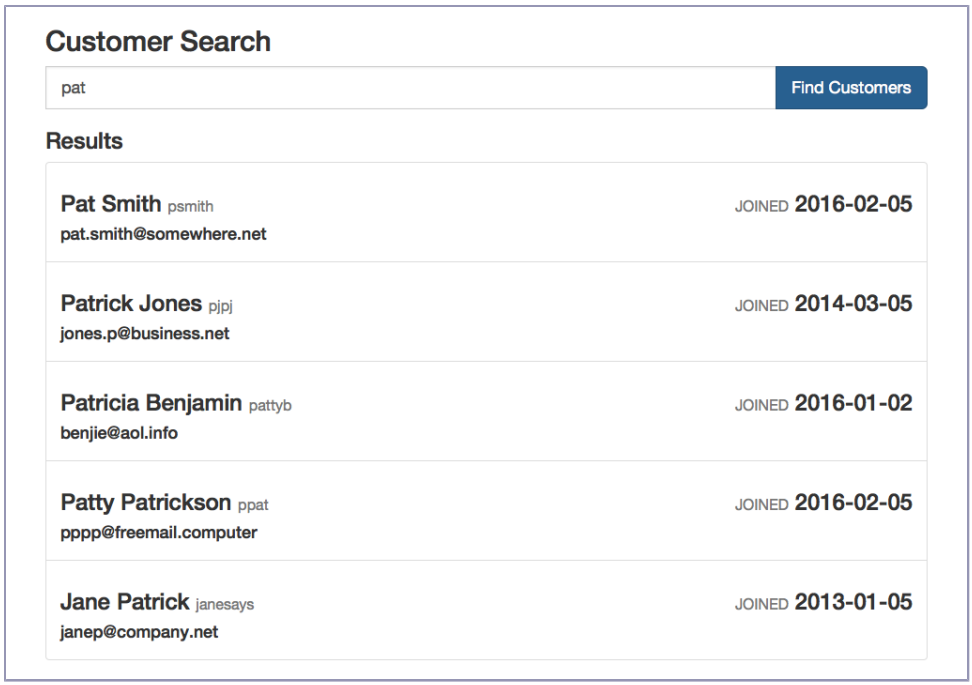
This references a hard-coded array of results called RESULTS:

typeahead/canned-results/shine/webpack/application.js

```
var RESULTS = [
  {
    first_name: "Pat",
    last_name: "Smith",
    username: "psmith",
```

```
    email: "pat.smith@somewhere.net",
    created_at: "2016-02-05",
  },
  {
    first_name: "Patrick",
    last_name: "Jones",
    username: "pjpj",
    email: "jones.p@business.net",
    created_at: "2014-03-05",
  },
  {
    first_name: "Patricia",
    last_name: "Benjamin",
    username: "pattyb",
    email: "benjie@aol.info",
    created_at: "2016-01-02",
  },
  {
    first_name: "Patty",
    last_name: "Patrickson",
    username: "ppat",
    email: "pppp@freemail.computer",
    created_at: "2016-02-05",
  },
  {
    first_name: "Jane",
    last_name: "Patrick",
    username: "janesays",
    email: "janep@company.net",
    created_at: "2013-01-05",
  },
];
```

Now, you can reload our app, type in “pat,” click the “Find Customers” button, and see our hard-coded results, as shown in the following figure.



With all the pieces of our Angular app wired together, you can re-implement search so that it gets real results from our Rails application, instead of the canned ones.

Get Data from the Back End

The logic for searching customers in CustomersController is sound; you just need it to return JSON. Rails makes that easy by using respond_to. We'll add this to the end of the index method:

typeahead/backend/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController

  PAGE_SIZE = 10

  def index

    # method as it was before

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

To call it from our Angular app, you'll use the Angular Http class. Like almost all JavaScript libraries that make network calls, Angular's Http operates asynchronously. This means you'll make an HTTP request to our Rails application and set up a function that will be called back when the HTTP request is complete.

In Angular, the mechanism to achieve this is via *Observables*, specifically the Reactive Extensions for JavaScript (RxJS) library (see *Reactive Programming with RxJS [Man15]* for a deep-dive on this library).[7] The way an observable works is that you *subscribe* to the *events* being observed by passing a function to the observable's subscribe function.

In our case, the event you want to observe is the HTTP request you'll make to our Rails application. When you get a response, you'll set the customers property to what you get back, and Angular will re-render our results. Making this work requires both writing the new search function to use Http, but also bringing in and setting up the Http module. In your day-to-day coding, you'd do the setup first, but this will require a slight digression into how Angular manages such classes, so let's see the code for search first:

typeahead/backend/shine/webpack/application.js

```
search: function() {
  var self = this;
  self.http.get(
    "/customers.json?keywords=" + self.keywords
  ).subscribe(
    function(response) {
      self.customers = response.json().customers;
    },
    function(response) {
      alert(response);
    }
  );
}
```

Let's go through each line.

1. Ultimately, you need to modify this.customers with the data you get back from our Rails application. Since you'll be doing that inside a function, the value of this in that function will be different, so if you wrote this.customers = «response»;, it wouldn't work. (Yehuda Katz wrote an excellent blog post[8] explaining this in detail.) For our purposes, we'll save it as self, a local

7. https://github.com/Reactive-Extensions/RxJS
8. http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this

variable we can refer to later (a common technique amongst JavaScript programmers).

❷ Here's where you set up the HTTP request to the Rails application. Note that you're assuming the existence of an `Http` object available via `self.http`. You'll see how that gets set up later. Also note that you are using `self` here instead of `this`. Because you've saved `this` in the variable `self`, it makes the code easier to follow if you consistently use `self` (even if `this` might work in some cases).

❸ This is the URL to our Rails controller you modified earlier. Note that you are using `.json` in the URL so that Rails knows to send us back JSON instead of rendering HTML.

❹ This is where you set up our function to be notified when the HTTP request is completed. The value passed to our function is a `Response`[9] object.

❺ Finally, you extract the results out of the response and set the value of `customers`. Note the use of our local variable `self` here. To re-iterate, if you had used `this` here, `this` would not have referred to the right object and our code wouldn't work.

❻ Last, you pass a second function to `subscribe` that will be called if there was an error talking to the server. You're simply alerting the user with whatever the response is for now. In a real production application, you'd need to make some design decisions around the user experience when an error happens and implement that here.

With `search` implemented, you just need to find out where `this.http` came from. This requires learning about how Angular does *dependency injection*.

Dependency injection refers to a way of structuring code so that when one piece of code depends on another, a third piece of code will wire the two together, *injecting* the first with the second. In our case, instead of creating an instance of `Http` ourselves, you want Angular to do that and then inject that instance into `CustomerSearchComponent`.

Angular is built entirely using the concept that code should be injected with its dependencies instead of creating them itself. This is the complete opposite of how Rails works and how you write Rails code. Part of why Angular does this is philosophical, but a practical upside is that it simplifies writing unit tests, which you'll see in Chapter 7, *Test This Fancy New Code*, on page 95.

9. https://angular.io/docs/ts/latest/api/http/index/Response-class.html

The way you'll set this up in your Angular app is to modify our constructor in a way that describes to Angular that you'd like an instance of Http passed when CustomerSearchComponent is first created. You do that by changing the value of the constructor property from a single function to an array with a special form.

The way this special array works is that the very last element is our constructor function, and it will take, as arguments, all the objects you want Angular to pass to it. The remaining elements are the classes representing the instances you need. For example, you'll pass ng.http.Http (the location of the Http class based on how you used function) as the first argument, and our constructor accepting an instance of that class as the second argument. (It's weird, but you get used to it.)

typeahead/backend/shine/webpack/application.js

```
constructor: [
  ng.http.Http,
  function(http) {
    this.customers = null;
    this.http      = http;
    this.keywords  = "";
  }
],
```

If you later need another object, say Angular's router, you'd add it to the array, and then add it to the constructor function:

```
constructor: [
  ng.http.Http,
  ng.router.Router,
  function(http, router) {
    this.customers = null;
    this.keywords  = null;
    this.http      = http;
    this.router    = router;
  }
],
```

Angular doesn't keep a list of every possible dependency that could be injected into other code, so you have to take one more step to tell Angular that you want Http to be injectable into your classes. We do this by adding ng.http.HttpModule to the import key of our NgModule we configured earlier.

typeahead/backend/shine/webpack/application.js

```
var CustomerSearchAppModule = ng.core.NgModule({
  imports: [
    ng.platformBrowser.BrowserModule,
    ng.forms.FormsModule,
```

```
    ng.http.HttpModule
  ],
  declarations: [ CustomerSearchComponent ],
  bootstrap: [ CustomerSearchComponent ]
})
.Class({
  constructor: function() {}
});
```

Understanding why you have to do this requires knowing a bit about Angular's internal design, which we won't get into here. Suffice to say, when you want access to classes provided by an Angular library, you need to take explicit steps to make those classes available—they won't just show up because you required the library. The high-level advantage of all this is that our code never has to create instances of Angular's classes. Because we write our code assuming Angular will hand them to us, when we go to test this code (in Chapter 7, *Test This Fancy New Code*, on page 95), it'll be much easier.

The last step is to bring in the HTTP module so that ng.http.Http and ng.http.HttpModule properly refer to Angular's Http class and HttpModule constant, respectively. You do this by adding a call to require at the top of our file:

typeahead/backend/shine/webpack/application.js

```
var ng                = {
  core:                    require("@angular/core"),
  common:                  require("@angular/common"),
  compiler:                require("@angular/compiler"),
  forms:                   require("@angular/forms"),
  platformBrowser:         require("@angular/platform-browser"),
  platformBrowserDynamic: require("@angular/platform-browser-dynamic"),
  router:                  require("@angular/router"),
  http:                    require("@angular/http")
};
```

Let's recap the steps. They are listed here in the order you'd normally do them once you understand the concepts in play.

1. Bring in the http module using require.
2. Configure our NgModule to import HttpModule to provide an implementation of Http.
3. Change the constructor to indicate that it's the piece of code that needs an instance of Http by using a specially-formed array.
4. Modify the constructor *function*—which is in the last position of the specially-formed array—to accept an argument that, at runtime, will be an instance of Http, configured by one of Angular's providers.

If you reload the page, enter a search term, and click "Find Customers," you can see real results come back. It will behave just as it did before, but now it's all in Angular!

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, you were able to just focus on the Angular-based aspects of the feature. Now that you've done that, you can change the search so that it searches as you type.

Changing Our Search to Use Typeahead

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

Currently, when the user modifies the contents of the text field, Angular updates the value of keywords in our CustomerSearchComponent class. You can't directly see it, but it does it as the user types. If you can hook into that behavior, you can perform your search as the user is typing.

Angular provides a way to do this by binding to the ngModelChange property. The code you have now is:

```
<input bindon-ngModel="keywords" ... >
```

Which is equivalent to this:

```
<input bind-ngModel="keywords"
         on-ngModelChange="keywords=$event" ... >
```

Recall that on- creates a one-way binding from the view to our code (you used this for the click event earlier in *Respond to Click Events*, on page 80). As part of sending the event back to our code for ngModelChange, Angular sets the global variable $event. This means that instead of assigning it to keywords, as happens by default, you can send $event to our search function:

```
<input type="text" id="keywords" name="keywords" \
       placeholder="First Name, Last Name, or Email Address"\
       class="form-control input-lg" \
       bind-ngModel="keywords" \
       on-ngModelChange="search($event)"> \
```

Note that because you've replaced Angular's default behavior, you need to set this.keywords inside search ourselves. Right after you do that, however, you

can use the updated value to perform the search just as before (though you're only going to do a search for three or more characters so you don't do too broad a search):

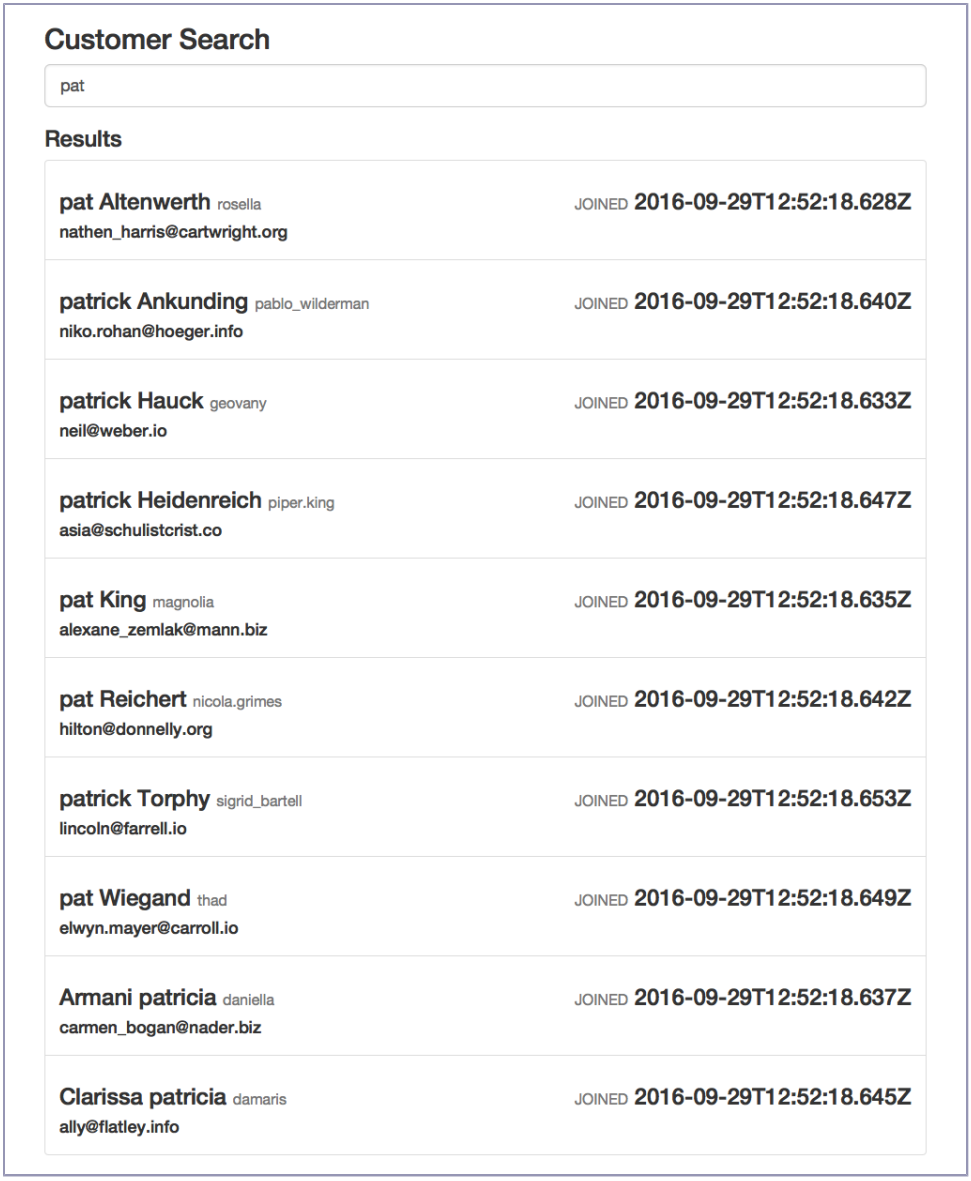
typeahead/actual-typeahead/shine/webpack/application.js

```
search: function($event) {
  var self = this;
  self.keywords = $event;
  if (self.keywords.length < 3) {
    return;
  }
  self.http.get(
    "/customers.json?keywords=" + self.keywords
  ).subscribe(
    function(response) {
      self.customers = response.json().customers;
    },
    function(response) {
      alert(response);
    }
  );
}
```

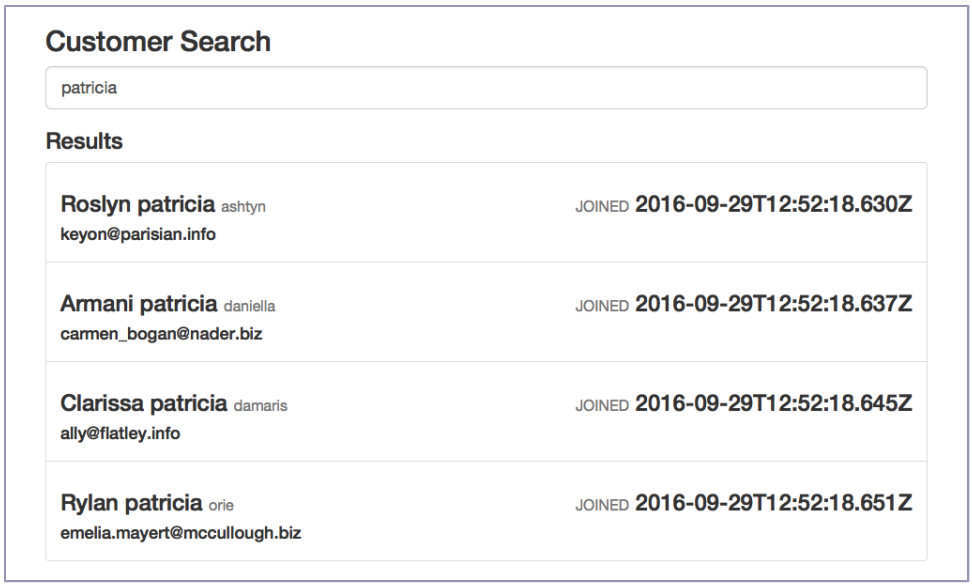
Finally, let's remove the "Find Customers" button as it's no longer needed. Removing this means you can remove the `span` surrounding the button as well as the `div` you used to make the button group. Our search form now looks like so:

```
<section class="search-form"> \
  <form> \
    <label for="keywords" class="sr-only">Keywords></label> \
    <input type="text" id="keywords" name="keywords" \
           placeholder="First Name, Last Name, or Email Address"\
           bind-ngModel="keywords" \
           on-ngModelChange="search($event)" \
           class="form-control input-lg">\
  </form> \
</section> \
```

Now, reload the page and type in "pat." You'll see some search results like those shown in the following figure.



If you keep typing out “patricia,” the results automatically reduce to only those that match, as shown in the next figure.



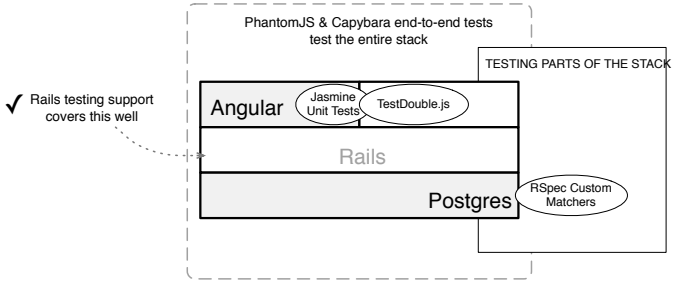
The typeahead works! The entire feature required very little code (once you installed and configured Angular—a one-time cost), and instead of implementing typeahead with a special-purpose library, you have set up a framework for implementing any user interface you might need. Because of how Angular works, you aren't wrestling with how to attach our JavaScript to our DOM elements or how to interact with the back end. 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, you were able to create a fairly sophisticated feature quickly and without a lot of code. And it's fast, thanks to Postgres's sophisticated indexing and ordering features.

Next: Testing

It's one thing to get code working in a browser, but it's another to have confidence in that code to ship it to 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. But now that you'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, you'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, you need to do more than just write working code—you 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. You want the same experience testing Angular and database constraints that you get with Rails models and controllers. You 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's 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 focus on in this chapter.

You'll learn how to write a clean and clear test of the database constraint you created in *Exposing the Vulnerability Devise and Rails Leave Open*, on page 31. You'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, I'll show you how to write a unit test for the Angular code you wrote in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 69 using the JavaScript-based test library Jasmine. This will allow you 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 you'll write in Ruby, you'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 you get to *Writing Unit Tests for Angular Components*, on page 117, you'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's that same test in Jasmine:

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

As you can see, both tests have a similar shape and structure. This means that when you're 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:

```
group :development, :test do
  gem "rspec-rails", '~> 3.0'
end
```

This gem includes tight Rails integration for RSpec and will bring in the base RSpec gems as dependents.

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

Now, 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 you can use them, but also add a few nondefault configuration options. Here's 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.shared_context_metadata_behavior = :apply_to_host_groups

➤ 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—you don't want to use the .should assertions because this would be counter to our desire for our Ruby and JavaScript tests to be similar. We're setting verify_doubled_constant_names as an extra safety measure if you should need to mock class behavior (this warns you if you 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 without prefixing them with RSpec.

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

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

```
require 'rails_helper'

describe "rspec is configured properly" do
  it "should pass" do
    expect(true).to eq(true)
  end

  it "can fail" do
#    expect(false).to eq(true)
  end
end
```

When you run bundle exec rails spec, it will run this spec, and you should see one test pass and the other fail.

```
$ bundle exec rails spec
rspec is configured properly
  should pass
  can fail (FAILED - 1)

Failures:

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

       expected: true
            got: false

       (compared using ==)
     # ./spec/canary_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/canary_spec.rb:7 # rspec is configured properly
```

This looks good, and you're ready to test the new features and technologies we've discussed. Note that we won't be going over the sorts of tests you'd typically write for a Rails app. This isn't because you shouldn't write them (you should), but because I want to focus on the new types of tests you need for what we're discussing. The Rails code you've seen *is* tested, and you can refer to the source included with the book if you want to see how I've done that.

Let's start by writing tests of our User model that exercise the database constraints you created in *Exposing the Vulnerability Devise and Rails Leave Open*, on page 31.

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'd typically use various features of Active Record to ensure database integrity and you'd naturally want to test that code. Although we're using constraints to enforce database integrity (like the `check` constraint from *Exposing the Vulnerability Devise and Rails Leave Open*, on page 31), we'd still like to have test coverage that the constraint is doing what we want.

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

Assert that Constraints Exist Using RSpec's Matchers

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

The method `update_attribute` is available on all Active Record instances and it circumvents validations. You can use this in our test to attempt to insert bad data and simulate a rogue agent that's not using Active Record. 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 the 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 reformatted the error message to show only what we're interested in, but you can see that the internals of Rails raised a PG::CheckViolation, which Active Record wrapped inside an ActiveRecord::StatementInvalid exception. To test that this error occurs, you'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 you just check that you'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 an ActiveRecord::StatementInvalid is raised and 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 it's a reasonable compromise.

Create the file spec/models/user_spec.rb and add the following code (the use of raise_error is highlighted):

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 you're using a regular expression as the second argument to raise_error so that you aren't too tightly coupled to the specific error message. Let's run our tests (note that you're using rspec to run a single test file—bundle exec rails spec 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
```

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 you *can* make the test code a bit cleaner and more intention revealing by creating a custom RSpec matcher.

Use RSpec Matchers to Make the 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`.

You can create your own custom matchers to test attributes of your code that are more particular to what you're doing. This saves us test code and can make our tests clearer. Ideally, you'd be able to write your 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 you could create the matcher `violate_check_constraint`, it would not only make your tests clearer, but also would allow you to abstract the method you're using to test: catching an exception and checking its message. This means if you could devise a better way of testing the constraint, you'd have to change it in only 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 look at the code:

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

```
RSpec::Matchers.define :violate_check_constraint do |constraint_name|
  supports_block_expectations
  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 let's take it one step at a time.

❶ Here, you define your 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 you're using. In that case, the match method (discussed next) would be given the result of the code under test. Since you need to actually *execute* the code under test ourselves—so you can detect the exception that was thrown—you must use the block syntax. The supports_block_expectations method tells RSpec that this is the case.

❸ This is where you 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 you used supports_block_expectations, the argument passed is the block used, unexecuted. Our job is to execute it and see what happens.

❹ Here, you run the code under test.

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

❻ Here, you catch the exception you're expecting. If you 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 you did before, you'll simply assert that it contains the name of the constraint you're expecting should be violated.

You're almost ready to use your 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 and 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 spec ...

  end
end
```

Running our spec, you 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
```

You've now seen how RSpec can allow you to test your 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 you've written and simulating user behavior.

Running Headless Acceptance Tests in PhantomJS

Acceptance tests 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 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 you can be sure that all of the DOM manipulation you 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 servers, it requires 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*[2] 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 you can interact with it in your tests.

Most Rails acceptance tests use *Capybara*,[3] which provides an API for interacting with such an instrumented browser. To allow Capybara to talk to PhantomJS, you're going to use *Poltergeist*,[4] which is analogous to Selenium if you were using Firefox.

This may sound like a *ton* of new technologies and buzzwords, but it's all worth it to get the kind of test coverage you need. You need to add the PhantomJS and Poltergeist gems to the `Gemfile`, do a bit of configuration, and start writing acceptance tests as you normally would. Let's get to it.

Install and Set 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.[5] You'll need the latest version, which is 2.1.1 at the time of this writing. Pay particular attention to this as version 1.9 is still in wide use and wil *not* work for our purposes here.

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

```
$ phantomjs --version
2.1.1
$ phantomjs
phantomjs> console.log("HELLO!");
HELLO!
undefined
phantomjs>
```

2. http://phantomjs.org
3. https://github.com/jnicklas/capybara
4. https://github.com/teampoltergeist/poltergeist
5. http://phantomjs.org/download.html

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 you'll write for our acceptance tests and the "scriptable JavaScript API" PhantomJS provides. To do 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, I'll explain more when we see the acceptance tests.

To use Poltergeist to run acceptance tests in a browser, you have to do three things: (1) you have to configure Capybara to use it during test runs; (2) you must configure RSpec to handle the testing database differently for acceptance tests than for our unit tests; and (3) you have to make sure webpack-dev-server is running so that our JavaScript and CSS are served up during the tests.

Configuring Capyabra

To connect Poltergeist and Capybara, you just need a few lines in spec/rails_helper.rb. You'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 you don't have a lot of JavaScript and want our acceptance tests to normally run using a 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 are 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 you'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 Active Record and the path helpers).

Next, we need to deal with how we manage our test data during acceptance test runs. We'll do that using a gem called *DatabaseCleaner*.

Using DatabaseCleaner to Manage Test Data

In a normal Rails unit test, the testing database is maintained using *database transactions*.[6] 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, effectively undoing all the changes you 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 you 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 "Pat", but we are also testing our registration by signing up a user named "Pat," our search test might fail if the registration test runs first, since there would be *five* users named "Pat."

Fortunately, this is a common problem and has a relatively simple solution: the *DatabaseCleaner*[7] gem.

6. http://en.wikipedia.org/wiki/Database_transaction
7. https://github.com/DatabaseCleaner/database_cleaner

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 you can keep the fast and efficient transaction-based approach for our unit tests, but use a different approach for our acceptance tests.

First, add DatabaseCleaner to the Gemfile and bundle install:

testing/setup-poltergeist/shine/Gemfile

```
group :development, :test do

  # other gems ...

  gem 'database_cleaner'
end
```

Now, configure it in spec/rails_helper.rb. To do this, disable RSpec's built-in database handling code by setting use_transactional_fixtures to false (note that the generated rails_helper.rb will have it set to true). Then use RSpec's hooks[8] to allow DatabaseCleaner to handle the databases. By default, use DatabaseCleaner's :transaction strategy, which works just like RSpec and Rails's default. But for our acceptance tests (which RSpec calls "features"), we'll use :truncation, which means DatabaseCleaner will use the SQL truncate 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.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
```

8. https://www.relishapp.com/rspec/rspec-core/v/3-2/docs/hooks/before-and-after-hooks

```
➤   end
➤   config.after(:each) do
➤     DatabaseCleaner.clean
➤   end
end
```

The last thing we have to do is to make sure our JavaScript and CSS is being served up to PhantomJS during our tests, meaning we have to make sure webpack-dev-server is running.

Running webpack-dev-server During Tests

In *Setting Up Bootstrap with NPM and Webpack*, on page 6, you set up Webpack as a replacement to Sprockets to manage our assets. A consequence of that was that you had to run webpack-dev-server alongside rails server in order to have the assets served. We made this easy by using Foreman, but we can't really do that when running our tests.

You *could* try to remember to run webpack-dev-server whenever you are running tests, but this is error-prone and difficult. You could also precompile all of our assets, the way you would on production, so that you wouldn't need to run webpack-dev-server. Webpack-rails provides the rake task webpack:compile, which will create a bundle of our assets and work during our tests without running webpack-dev-server. But, this is pretty slow and you don't want to have to do this every time you run a single test.

Instead, we'll use the RSpec hooks you saw earlier to arrange for the server to run before our tests, and shut down after. You'll need to learn a bit about UNIX processes to do this.

In broad strokes, you want to start a separate process running webpack-dev-server before any test runs and you want to save the process identifier (PID) of that process. When all of our tests have completed, you want to use that PID to shut down the process you started.

To start up a process and get its PID, you can use spawn[9], provided by the Kernel module that's available everywhere in Ruby. This method takes a string for the command to run, and returns its PID.

```
pid = spawn("webpack-dev-server")
```

To shut down a process in Ruby, you can use Process.kill. kill is so-named because the UNIX command-line app kill is the command used to ask (or force) process to shut down. kill takes two arguments: a signal to send the process

9. http://ruby-doc.org/core-2.3.1/Kernel.html#method-i-spawn

and the PID of that process. The reason it requires a signal is that you can both ask a process to shutdown gracefully, or force it. You don't want to force it by default, because the process might have resources it should release gracefully. But, if anything goes wrong, you do need to make sure the process ends, or the next time you run your tests, webpack-dev-server will be running in an unknown state.

To arrange for all this, you want to ask webpack-dev-server to shutdown gracefully by using the HUP signal (which means "hang up," as in a telephone line; yes, UNIX has been around for a while). You'll then need to wait a few seconds to see if webpack-dev-server exits and, if it hasn't, send the signal KILL to tell the operating system to stop the process. (This works 99.9999% of the time.) If you want to learn more about all this UNIX process stuff, *Working With UNIX Process [Sto12]* is a great reference for doing so with Ruby.

You can wait by using the Timeout.timeout method, which takes a block and, if that block has not completed in the given amount of time, raises a Timeout::Error, which you can then catch and use as your signal to forcibly kill webpack-dev-server.

```
Process.kill("HUP",pid)
begin
  Timeout.timeout(2) do
    Process.wait(pid,0)
  end
rescue Timeout::Error
  Process.kill("KILL",pid)
end
```

Now that you know the code to start and stop webpack-dev-server from within Ruby, you just need to make sure it runs at the right time. For staring it up, add this to the before(:suite) hook you already have. For the shutdown code, there is an analogous after(:suite) hook you can use.

testing/setup-poltergeist/shine/spec/rails_helper.rb

```
config.before(:suite) do
  $pid = spawn("./node_modules/.bin/webpack-dev-server " +
               "--config config/webpack.config.js --quiet")
  DatabaseCleaner.clean_with(:truncation)
end
config.after(:suite) do
    puts "Killing webpack-dev-server"
    Process.kill("HUP",$pid)
    begin
      Timeout.timeout(2) do
        Process.wait($pid,0)
      end
```

```
    rescue => Timeout::Error
      Process.kill(9,$pid)
    end
end
```

Now that you've set up PhantomJS, Poltergeist, and DatabaseCleaner, you're ready to write an acceptance test.

Write the 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 you created in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 69. As you recall, this Angular app had a text field where you could type a name. As you typed, a heading would dynamically update with what you had entered.

Before we see the actual test, let's plan how it will work. First, you have to log in, since every page in Shine requires a login. That means you'll need to set up a test user and fill in that user's name and password on the login screen. Then, you'll enter some text in the text field in our Angular test app. Finally, you'll assert that the DOM updated with what you typed.

Way back in Chapter 3, *Secure the User Database*, on page 31, we talked about how Devise properly secures our user information, including passwords. This means it will be difficult 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: "pat@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 test runs (you saw something similar when setting up DatabaseCleaner). We'll also use the method `feature` (instead of `describe`) to indicate that this is an acceptance test.

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)    { "pat@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 log in, 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 you'll see Capybara's DSL in action. A good reference for everything you can do can be found on Capybara's GitHub page,[10] but I'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: email
    fill_in      "Password", with: password
    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: "Pat"
    within "h2" do
      expect(page).to have_content("Hello Pat")
    end
```

10. https://github.com/jnicklas/capybara#the-dsl

```
  end
```

Thanks to Capybara's DSL, the test is pretty readable. When you 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
```

Before we move on, let's write a test of our typeahead feature from the previous chapter. Doing so will be a bit more involved and allow us to test an *actual* feature of Shine as well as see how to use Capybara in light of a dynamic page with a heavier client-side code.

Test the Typeahead Search

There are two parts of the typeahead search we can test. The first is that merely typing in the search field will perform the search. The second is that our results are ordered according to our original specification from Chapter 4, *Perform Fast Queries with*, on page 39.

To test the search, you'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. You need to create customers in the database, create a test user, log in 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*[11] to do this, we're not going to use them here. Because tests related to search require meticulous setup 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 create customers, you'll define a helper method, `create_customer`, that will allow you to specify only those fields of a customer you want, using Faker to fill in the remaining required fields.

testing/setup-poltergeist/shine/spec/features/customer_search_spec.rb

```
require 'rails_helper'
```

11. http://guides.rubyonrails.org/testing.html#the-low-down-on-fixtures

```
feature "Customer Search" do
  def create_customer(first_name:,
                      last_name:,
                      email: nil)

    username = "#{Faker::Internet.user_name}#{rand(1000)}"
    email  ||= "#{username}#{rand(1000)}@" +
               "#{Faker::Internet.domain_name}"

    Customer.create!(
      first_name: first_name,
       last_name: last_name,
        username: username,
           email: email
    )
  end
```

YOu'll also re-create the user and password let statements from our Angular test app test, since you'll need to create a user here and log in before the test.

testing/setup-poltergeist/shine/spec/features/customer_search_spec.rb

```
let(:email)    { "pat@example.com" }
let(:password) { "password123" }
```

With create_customer, email, and password in place, you can now create the test data you need to run the tests. As before, you'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. Because 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 (I've highlighted the login code 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: "Chris",
                  last_name: "Aaron"

  create_customer first_name: "Pat",
                  last_name: "Johnson"

  create_customer first_name: "I.T.",
                  last_name: "Pat"
```

```
  create_customer first_name: "Patricia",
                  last_name: "Dobbs"

  create_customer first_name: "Pat",
                  last_name: "Jones",
                      email: "pat123@somewhere.net"
➤ visit "/customers"
➤ fill_in      "Email",    with: email
➤ fill_in      "Password", with: password
➤ click_button "Log in"
end
```

Now we can start writing tests. Our first test will search by name. If we search for the string "pat", given our test data, we should expect to get four results back. Further, we should expect that the test user named "Patricia Dobbs" will be sorted first, whereas the test user "I.T. Pat" 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*, on page 57 that we designed our results using Bootstrap's List Group component). You 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: "pat"
  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("Patricia")
    expect(page.all("ol li.list-group-item")[0]).to have_content("Dobbs")

    expect(page.all("ol li.list-group-item")[3]).to have_content("I.T.")
    expect(page.all("ol li.list-group-item")[3]).to have_content("Pat")
  end
end
```

It's likely this test will not pass. Despite the fact that the test makes logical sense, if you look more closely, we are assuming that the back end will respond (and results will be rendered) between the time *after* the fill_in, but *before* the

first expect. This is likely not enough time, meaning we'll start expecting results in our test before they've been rendered in the browser (this is a form of *race condition*). Worse, if you try to debug it (see the following sidebar for some tips), the test will pass.

What you want to do is have the test wait for the back end to complete. There are many ways to do this, but the cleanest way, and the way Capybara is designed, is to write the markup and tests so that a change in the DOM signals the completion of the back end.

For the Capybara part, we are actually already set up to wait for the DOM. within is implemented by using find,[12] which is documented to wait a configured amount of time for an element to appear in the DOM—exactly what we want!

The problem is that the markup we're waiting on (<section class='search-results'>) is always shown, even before we have results. This means that the within won't have to wait, since the element is there, and it will proceed with the expectations, which fail. So, we'll change our template to only show this markup if there are search results by using *ngIf like so:

```
<section class="search-results" *ngIf="customers"> \
  <!-- Rest of the template -->
</section> \
```

*ngIf is similar to *ngFor in that the special leading asterisk is syntactic sugar for a much longer construct using the <template> element. I won't show the expansion here, since you learned how this concept generally works already in *Populate the DOM Using*, on page 82.

The way *ngIf works is to remove elements from the DOM if the expression given to it evaluates to false. This means that until you have results, there won't be any element that matches section.search-results, which means that within will wait on that element to appear. As long as our back end returns in three seconds (Capybara's default wait time), our test will pass. If you find that adding calls to sleep makes your tests pass, consider reexamining your use of within and find and see if you can change your markup to use this approach.

Debugging Browser-Based Tests

In a Selenium-based testing setup, you can observe the browser during the tests. This makes it possible to debug tests that are failing for features that are working

12. http://www.rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Finders#find-instance_method

when executed manually. For headless tests, it's much more difficult. Here are two techniques I've used to help debug these tests.

Printing the HTML Capybara's page object has a method called html that will dump the HTML of the current browser. A simple call to puts page.html right before a failing expectation can often be quite illuminating as to what the state of the page is. If you combine this with debugging information in your view code, the answer to your test woes often reveals itself.

Taking a Screenshot Capybara can also take screenshots via the save_screenshot[a] method (in fact, this feature is what has created most of the screenshots in this book). You can give it a filename and it will show what the browser would render at that moment—for example, save_screenshot("/tmp/screenshot.png").

a. http://www.rubydoc.info/github/jnicklas/capybara/master/Capybara/Session#save_screenshot-instance_method

Now that you 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 you 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: "pat123@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("Pat")
    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("Patricia")
    expect(page.all("ol li.list-group-item")[1]).to have_content("Dobbs")

    expect(page.all("ol li.list-group-item")[3]).to have_content("I.T.")
    expect(page.all("ol li.list-group-item")[3]).to have_content("Pat")
  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
```

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 Angular code purely in the browser is somewhat cumbersome, especially if it becomes complex with a lot of edge cases. To help get good test coverage without always having to go through a browser, you need to be able to unit test your Angular code.

Writing Unit Tests for Angular Components

Browser-based acceptance tests are slow and brittle. Even though you've eschewed starting an actual browser for each test by using PhantomJS, you still have to start our server and have it serve pages to the headless browser. Further, our tests rely on DOM elements and CSS classes to locate content used to verify behavior. You 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, you have model tests and controller tests that allow you 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 application, there simply isn't much client-side code, so we are comfortable not explicitly testing it. In our more modern app, where a nontrivial amount of logic is written in JavaScript, the lack of unit testing will be a problem. For example, in the Angular app that powers the typeahead search you built in Chapter 6, *Build a Dynamic UI with AngularJS*, on page 69, the `search` function has logic to prevent doing a search if the search term is fewer than three characters. It's simple code, but there should be a test for it.

In this section you'll set up a means of writing and running unit tests for our Angular code. You'll use *Jasmine*,[13] which is a commonly used JavaScript

13. https://jasmine.github.io

testing framework. You'll also use the *Testdouble.js*[14] library to stub the various functions and objects Angular is providing to our code so that you can test in isolation. Because of how Angular is structured, this will be surprisingly straightforward.

First, let's get set up for testing by installing Jasmine and Testdouble.js, and then arranging to run our unit tests with Rake.

Set Up Jasmine, Testdouble.js, and the Rake Tasks

We're going to add a new section to package.json called devDependencies. This is similar to the :development group in our Gemfile, and is how you can specify JavaScript libraries you need for testing, but not for production.

```
{
  "name": "shine",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    // Existing dependencies ...
  },
➤ "devDependencies": {
➤   "jasmine": "2.4.0",
➤   "jasmine-node": "1.11.0",
➤   "testdouble": "1.6.0"
➤ }
}
```

You can install these with npm install. Now, let's write a simple test to make sure things are working. Create the directory spec/javascripts and create a file there named canary.spec.js like so:

testing/angular-unit-testing/shine/spec/javascripts/canary.spec.js

```
var td = require("testdouble");

describe("JavaScript testing", function() {
  it("works as expected", function() {
    var mockFunction = td.function();

    td.when(mockFunction(42)).thenReturn("Function Called!");

    expect(mockFunction(42)).toBe("Function Called!");
  });
});
```

We'll learn about TestDouble once we start writing tests in *Using Test Doubles to Stub Dependencies*, on page 123, but basically this is bringing in the TestDouble library, using the describe and it functions provided by Jasmine, and

14. https://github.com/testdouble/testdouble.js

creating a simple test. You create a mock function that, when called with 42, returns the string "Function Called!" and then use expect to verify that this, in fact, happens.

Jasmine includes a test runner that was installed in node_modules/.bin called jasmine-node. We can invoke it like so to run our test (you may see a warning about util.print; dont' worry about it):

```
$ node_modules/.bin/jasmine-node spec/javascripts
.

Finished in 0.012 seconds
1 test, 1 assertion, 0 failures, 0 skipped
```

This is how we'll run our unit tests for our day-to-day work, but we also need them to get run automatically when someone types rake. The default task in any Rails application is to run all the tests, so we'll enhance the default rake task to run jasmine-node for us. To do that, create the file lib/tasks/jasmine.rake like so:

testing/angular-unit-testing/shine/lib/tasks/jasmine.rake

```
desc "Run Jasmine-based unit tests of JavaScript"
task :jasmine do
  root_dir = File.expand_path(File.join(File.dirname(__FILE__),"..","..",".."))
  sh("node_modules/.bin/jasmine-node #{root_dir}/spec/javascripts")
end

task :default => :jasmine
```

This locates our spec/javascripts directory and then feeds it to jasmine-node. You can try it by running rake jasmine and you can also see when you just run rake, our JavaScript unit tests get executed as well.

With our testing libraries installed, let's write a unit test of our existing Angular code.

Write a Unit Test for the Angular Code

We're going to write a complete unit test for CustomerSearchComponent, which will test its behavior when created and the behavior of the search function. Let's create spec/javascripts/CustomerSearchComponent.spec.js and just write out what we want to test using describe and it.

testing/angular-unit-testing/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
describe("CustomerSearchComponent", function() {
  describe("initial state", function() {
    it("sets customers to null");
    it("sets keywords to the empty string");
  });
```

```
  describe("search", function() {
    describe("A search for 'pa', less than three characters", function() {
      it("sets the keywords to be 'pa'");
      it("does not make an HTTP call");
    });
    describe("A search for 'pat', three or more characters", function() {
      describe("A successful search", function() {
        it("sets the keywords to be 'pat'");
        it("sets the customers to the results of the HTTP call");
      });
      describe("A search that fails on the back-end", function() {
        it("sets the keywords to be 'pat'");
        it("leaves customers as null");
        it("alerts the user with the response message");
      });
    });
  });
});
```

This roughly describes the behavior of CustomerSearchComponent, so all we need to do is fill in each of the it calls with our tests. To do *that*, we'll need access to CustomerSearchComponent, which is currently defined inside webpack/application.js. You could try to require that file in our tests, but doing so won't give us access to CustomerSearchComponent because it's not being exported. In JavaScript, unlike Ruby, you have to explicitly export anything you want visible outside a file.

webpack/application.js also has a ton of code we don't want to deal with for a unit test (our Angular test component is still hanging around in there). So let's extract the code for CustomerSearchComponent out of this file, so we can load only that class for our unit test. This is much more like how we manage our Ruby files and will seem natural. After we do that, we'll write our first tests, then learn about mocking CustomerSearchComponent's dependencies using TestDouble.js.

Extracting CustomerSearchComponent Into Its Own File

First, place all the code for CustomerSearchComponent into the file webpack/CustomerSearchComponent.js. That's the code we're assinging to CustomerSearchComponent in webpack/application.js. After all that code, we'll use the special module.exports property to indicate that CustomerSearchComponent is being exported to the outside world.

testing/angular-unit-testing/shine/webpack/CustomerSearchComponent.js

```
var CustomerSearchComponent = ng.core.Component({
  selector: "shine-customer-search",

  // Rest of the component ...

});
```

```
module.exports = CustomerSearchComponent;
```

Now, replace the code you extracted in webpack/application.js by using require:

testing/angular-unit-testing/shine/webpack/application.js

```
var CustomerSearchComponent = require("./CustomerSearchComponent");
```

While this cleans up our overall JavaScript codebase, it won't quite work. CustomerSearchComponent uses functions from Angular's Core and Http modules, and while we're using require to bring them into webpack/application.js, they won't be visible to the code you just extracted into webpack/CustomerSearchComponent.js. So, we'll need to require them explicitly in that file. We'll also need to require reflect-metadata, as ng.core.Class requires a global state that reflect-metadata sets up. This is a side effect of us using ES5 instead of TypeScript.

testing/angular-unit-testing/shine/webpack/CustomerSearchComponent.js

```
var reflectMetadata = require("reflect-metadata");
var ng = {
  core: require("@angular/core"),
  http: require("@angular/http")
};
var CustomerSearchComponent = ng.core.Component({
  selector: "shine-customer-search",

  // Rest of the component ...
```

This feels like duplication compared to how things work with Ruby, and it highlights a difference in how JavaScript treats modules. In JavaScript, everything is isolated to the file, and there isn't really any global state. This is great for encapsulation and code management, but it does require us to be expicit about what external libraries and functions any given file needs.

Now that we've extracted CustomerSearchComponent into its own class, we can run our browser-based tests to make sure we haven't broken anything and then move onto writing some tests for it.

Writing Simple Unit Tests

Let's start with the tests around the initial state of the component. That allows us to see how we create the component and inspect it. First, we'll use require to bring in CustomerSearchComponent:

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
var CustomerSearchComponent =
      require("../../webpack/CustomerSearchComponent");
```

Note that you have to use the relative path so that require can locate the file. Next, you need an instance of our component. I didn't talk much about what

the ng.core.Component and Class functions actually do. Under the covers, they create a JavaScript class, meaning you can create an instance of CustomerSearchComponent by using new, and anything you pass to the constructor will be sent along to the constructor function you defined.

Right now we don't need to worry about the constructor arguments, so we'll create our instance via new CustomerSearchComponent(). In order to only do that once, we'll use Jasmine's beforeEach function, which works just like RSpec's before method. Because of JavaScript's scoping rules, we need the variable for our component to be declared outside our describe function, so we'll declare it at the top of the file, and then assign it inside beforeEach:

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
var component = null;

describe("CustomerSearchComponent", function() {

  // existing test code ...

  beforeEach(function() {
    component = new CustomerSearchComponent();
  });
```

Now, we can fill in our two tests. All they'll do is access the customers and keywords properties of our instance and make sure they are set to their initial values:

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
describe("initial state", function() {
  it("sets customers to null", function() {
    expect(component.customers).toBe(null);
  });
  it("sets keywords to the empty string",function() {
    expect(component.keywords).toBe("");
  });
```

If you run jasmine-node, you'll see that our two assertions ran and our test passed:

```
> node_modules/.bin/jasmine-node spec/javascripts/CustomerSearchComponent.spec.js
.........

Finished in 0.01 seconds
9 tests, 2 assertions, 0 failures, 0 skipped
```

Now, let's test the search function of our component. This requires setting up some test doubles for Angular's Http library.

Using Test Doubles to Stub Dependencies

Because we're writing unit tests, we want to test in isolation, meaning we don't want to actualy make HTTP calls in our test. To avoid that, we'll stub out the Http class we're using an runtime. Because of how Angular components are designed, we can simply pass in our own implementation of Http to the constructor of CustomerSearchComponent. The question, then, is where do we get our test implementation?

Although Angular provides a complex system of mocking out HTTP calls, it is not well-documented, and will complicate our unit tests significantly. It requires specifying behavior at the network level, not at the class level, which results in hard-to-follow test setup. If we were using TypeScript (see *Why are we using ES5 and not TypeScript?*, on page 79 for why we aren't), this would all be slightly easier, but it turns out we can use a generic test double library to stub Angular's HTTP library.

The test double library is called Testdouble.js and you installed it earlier. It provides everything you need to make a mock HTTP object. Let's start with our first test of search which will do a search for "pa" and assert that we don't make any HTTP calls. In our case, our code would execute http.get() if it *did* perform a search, so we want a test double that responds to get() and a way to check that it wasn't called.

First, we'll bring in the Testdouble.js library:

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
var CustomerSearchComponent =
      require("../../webpack/CustomerSearchComponent");
var td = require("testdouble");

var component = null;
```

Next, we'll create a beforeEach block for our tests that creates a test double for Angular's HTTP and passes it into our component when we create it.

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
describe("search", function() {
  describe("A search for 'pa', less than three characters", function() {
    var mockHttp = null;
    beforeEach(function() {
      mockHttp = td.object(["get"]);
      component = new CustomerSearchComponent(mockHttp);
    });
```

The object method on td creates an object that can respond to the methods given in the array. In our case, the code td.object(["get"]) means "create an object that has a get function." In a later test, you'll use more functions of Testdou-

ble.js to specify the behavior of our mock object, but for now we just need it to exist.

Now, let's implement the first test that asserts that the keywords property gets set to our search string when we call search:

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js
```
it("sets the keywords to be 'pa'", function() {
  component.search("pa");
  expect(component.keywords).toBe("pa");
});
```

Next, we'll write a test that explicitly requires that the get method on http was not called. This is how we'll assert that a search for a string shorter than three characters doesn't hit the back-end.

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js
```
it("does not make an HTTP call", function() {
  component.search("pa");
  td.verify(mockHttp.get(), { times: 0 });
});
```

This is using the verify function of Testdouble, which is usually used to check that a method *was* called. In our case, because we want to make sure it wasn't, we use { times: 0 }.

Running our tests, they should pass. This wasn't too bad, but things are about to get a bit complex as we move onto the next test, where we need to arrange for our mock HTTP object to return results.

Stubbing Complex Interactions to Test HTTP

Let's write the test for a successful search of the term "pat." You can set up our beforeEach in much the same way, except now you have to provide an implementation of get.

If you look at the contract of the get function in Angular's HTTP library, it accepts a URL and returns an observable (which I talked about in *Get Data from the Back End, on page 86*). You then call subscribe on that observable, passing it two functions: one for a successful result and one for an error. The success function is expecting the data from the backend, and it's going to call json() on that data to parse it. Whew!

First, we'll create a test double for the response passed to our success function. We'll create it using object, just as we did with http, but we'll use td.when to specify the behavoior of the json function. Note this is all new code; we're highlighting the use of Testdouble.js.

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
describe("A search for 'pat', three or more characters", function() {
  var mockHttp = null;
  var customers = [
    {
      id: 1,
      created_at: (new Date()).toString(),
      first_name: "Pat",
      last_name: "Jones",
      username: "pj",
      email: "pjones@somewhere.net"
    },
    {
      id: 2,
      created_at: (new Date()).toString(),
      first_name: "Pat",
      last_name: "Jones",
      username: "pj",
      email: "pjones@somewhere.net"
    },
  ];
  beforeEach(function() {
➤   var response = td.object(["json"]);
➤   td.when(response.json()).thenReturn({ customers: customers });
    mockHttp = td.object(["get"]);
    component = new CustomerSearchComponent(mockHttp);
  });
```

We've declared customers outside beforeEach because we'll eventually assert that our component's customers property was assigned these customers. Next, we'll create a test double for our observable.

This is trickier because you need to define the subscribe function as "calls our success callback." You can do this by using td.callback along with td.when. Since subscribe takes two callbacks, and we only want one called, we'll use td.matchers.isA(Function) as the second argument. This tells Testdouble.js to not call the second callback, but to make sure it's a function.

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
var observable = td.object(["subscribe"]);
td.when(observable.subscribe(
  td.callback(response),
  td.matchers.isA(Function))).thenReturn();
```

This means that when our production code calls subscribe, the test double you've created will assume the first argument to that function is a callback, and it will call it with the response object. Lastly, we need to configure our mock HTTP object to return this mock observable when get is called:

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js
```
mockHttp = td.object(["get"]);

td.when(mockHttp.get("/customers.json?keywords=pat")).thenReturn(observable);
```

With all that setup, we can now write our two tests:

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js
```
describe("A successful search", function() {
  it("sets the keywords to be 'pat'",function() {
    component.search("pat");
    expect(component.keywords).toBe("pat");
  });
  it("sets the customers to the results of the HTTP call", function() {
    component.search("pat");
    expect(component.customers).toBe(customers);
  });
});
```

Running the tests, we should see that they pass. We've now successfully mocked the HTTP calls and can test in complete isolation! Also note that you didn't have to do anything Angular-specific. This is an interesting side-effect of the way Angular wants us to write code. *Our* code has no real ties to Angular, even though it uses some of Angular's classes. In the end, our code is just some JavaScript that you can execute.

Next, let's test the error case, which will create some complications around our use of window.

Testing the Error Case and Removing window

In *Get Data from the Back End*, on page 86, we decided to use window.alert as the means of letting the user know that an error had occured. As you'll see, this is going to be a slight problem. To see the problem, let's get our test setup. We'll set it up much as we did the success case, but we'll arrange to have Testdouble.js call the error callback instead. I've highlighted the differences in our setup.

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js
```
describe("A search that fails on the back-end", function() {
  beforeEach(function() {
➤   var response = "There was an error!";
    var observable = td.object(["subscribe"]);

➤   td.when(observable.subscribe(
➤     td.matchers.isA(Function),
➤     td.callback(response))).thenReturn();

    mockHttp = td.object(["get"]);
    td.when(mockHttp.get("/customers.json?keywords=pat")).thenReturn(observable);
    component = new CustomerSearchComponent(mockHttp);
```

```
  });
  it("sets the keywords to be 'pat'",function() {
    component.search("pat");
    expect(component.keywords).toBe("pat");
  });
  it("leaves customers as null", function() {
    component.search("pat");
    expect(component.customers).toBe(null);
  });
  it("alerts the user with the response message",function() {
    // ????
  });
```

You can already see what the problem will be, since it's not clear how to test that window.alert was called with our error message. Let's run the test anyway, and see what happens.

```
$ node_modules/.bin/jasmine-node \
  spec/javascripts/CustomerSearchComponent.spec.js
......FF.

Failures:

  1) CustomerSearchComponent search A search for 'pat',
       three or more characters
        A search that fails on the back-end
        sets the keywords to be 'pat'
   Message:
     TypeError: window.alert is not a function
   Stacktrace:
     TypeError: window.alert is not a function
     «stack trace omitted»

  2) CustomerSearchComponent search A search for 'pat',
       three or more characters
       A search that fails on the back-end
       leaves customers as null
   Message:
     TypeError: window.alert is not a function
   Stacktrace:
     TypeError: window.alert is not a function
     «stack trace omitted»

Finished in 0.033 seconds
9 tests, 7 assertions, 2 failures, 0 skipped
```

The problem is that window is only defined when running in a browser. You will have to define it and then set alert to be a doubled function on it. Fortunately, by declaring window *without* using var, it gets declared globally and can be accessed by our real CustomerSearchComponent.

testing/first-angular-unit-test/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
window = td.object(["alert"]);

describe("CustomerSearchComponent", function() {

  // existing test code ...

      describe("A search that fails on the back-end", function() {
        beforeEach(function() {

          // existing set up ...

          td.when(window.alert()).thenReturn();

          component = new CustomerSearchComponent(mockHttp);
        });
        it("alerts the user with the response message",function() {
          component.search("pat");
          td.verify(window.alert("There was an error!"));
        });
```

We're using the more typical form of td.verify, and this code does more or less what it appears to. It will fail the test if window.alert was called with an argument *other* than "There was an error!"

This is obviously brittle. Our test has to set global data read by our production code in order to work. Angular 1 provided an abstraction for us—$window—but this is not available in Angular 2. You could create a *service class* that handles the error notification, and inject that the same way you did with Http. It would probably be a better user experience to stop using window.alert and design a real alerting component using Bootstrap.

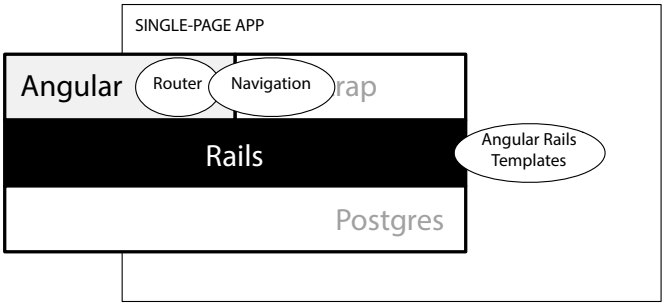
We won't do that here, as you now know enough to both design such a component and—now—write a test for it. For the latter, you could check that CustomerSearchComponent exposed the right error message to its view, much how we've been checking that customers has been set appropriately.

Now that you can successfully write unit tests using Jasmine, and mock out Angular-provided classes easily with Testdouble.js, you have the tools you need to test our JavaScript code. Our mocking of Http was complex enough that you should be able to tackle any test you need.

Next: Level Up on Everything

You learned a ton in this chapter about testing our Rails application at every level of the stack. Now that you can test anything from database constraints to JavaScript functions to end-to-end user interactions, you're ready to move on to more complex features.

Now it's time to up our game on everything. Over the next several chapters, you'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.



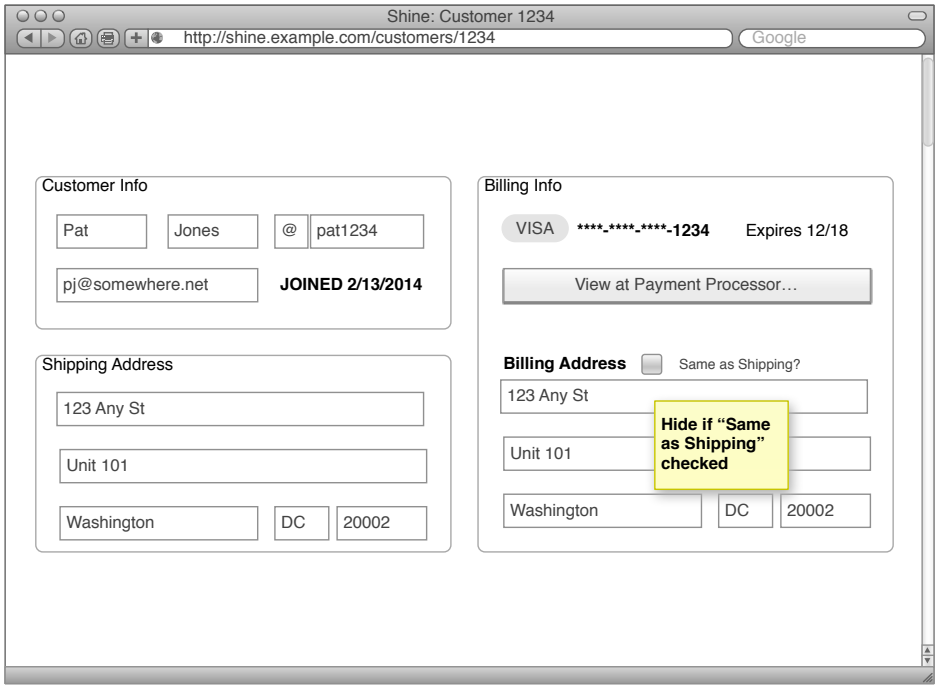
CHAPTER 8

Create a Single-Page App Using Angular's Router

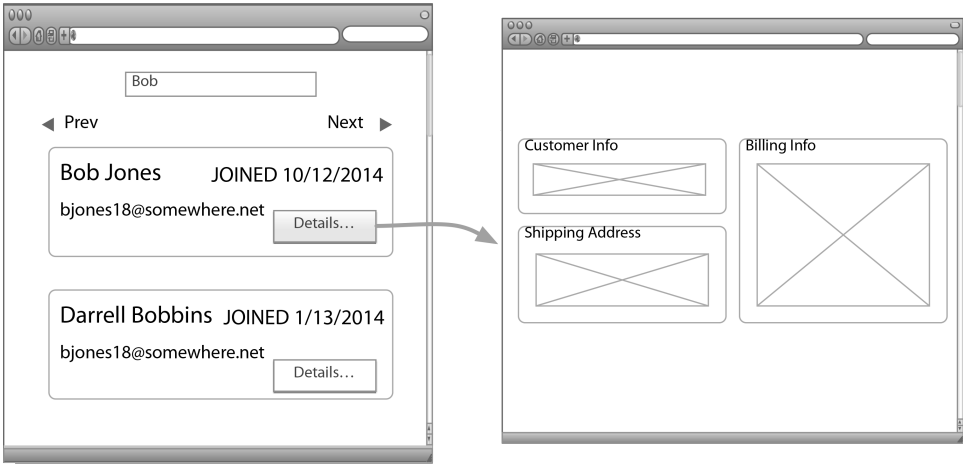
At this point, you're probably starting to feel confident. You experimented with some powerful features of Postgres, quickly made great-looking screens with Bootstrap, and created a dynamic user interface without a lot of code thanks to Angular. You also have rock-solid tests for every part of it. Now, it's time to level up.

In this chapter, we turn our simple customer search feature into a full-fledged *single-page app* using Angular's router and navigating users between pages—all within the browser. You learn how to put our view templates in separate files instead of string literals inside our JavaScript code, and see how to use test-driven development (TDD) to add features.

To learn all this, we'll start on a feature we'll build 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, similiar to what is shown in this figure:



First, we'll turn our existing Angular app into a single-page app that allows navigating from the search results to the detailed view (which will just be bare-bones, initially), like so:



To make this happen, we'll set up Angular's router, making sure our Angular routes play well with Rails' routes. We'll then add a second component to represent the detailed view, and write code to naivgate to it when the user clicks a new button that we'll add to the search results. This allows you to

learn about one of Angular's component lifecycle methods, which we'll use to parse Angular's route. Finally, we'll use TDD to connect our new component to the back-end in order to fetch customer details. This will give you an outline of how you can develop features using Angular and Rails, as well as serve as a second example of how to test this code.

But first, let's extract our templates out of string literals and into .html files, as this will make it much easier to manage our application in the face of multiple components.

Storing View Templates in HTML Files

Currently, the view templates are strings inside our JavaScript. This might be okay if we have a single small view, but as we create more components and have to manage more views, it will be increasingly difficult to keep them as strings. The trailing backslashes and general clunkiness of managing view code in JavaScript is something that's best to avoid.

Fortunately, Webpack makes this easy. With Angular1, which was used in the previous edition of this book, you had to install a special Rails plugin to convert our templates into something the Asset Pipeline could serve up. Now, you simply need a way to require an HTML file into a string and give that string to the template—key when declaring our component. You can do this with Webpack's raw loader.[1]

First, let's add it to package.json:

```
{
  "name": "shine",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    "stats-webpack-plugin": "^0.2.1",
    "webpack": "^1.9.11",

    // ...

    "bootstrap": "3.3.7",
    "raw-loader": "0.5.1"
  },
  "devDependencies": {
    // ...
  }
}
```

1. https://github.com/webpack/raw-loader

After installing the raw loader with npm install, we'll next modify config/webpack.config.js to tell Webpack that whenever we use require on an .html file, it should use the raw loader, which will have the effect of returning the HTML file as a string—exactly what we need!

complex-views/separate-template/shine/config/webpack.config.js

```
var config = {
  entry: {
    'application': './webpack/application.js'
  },

  module: {
    loaders: [

      // existing loader configuration ...

      {
        test: /\.html/,
        loader: 'raw'
      }
    ]
  },
```

Finally, you can move all the HTML from webpack/CustomerSearchComponent.js into webpack/CustomerSearchComponent.html (don't forget to remove all the trailing backslashes), and then bring it back in using require:

complex-views/separate-template/shine/webpack/CustomerSearchComponent.js

```
var CustomerSearchComponent = ng.core.Component({
  selector: "shine-customer-search",
  template: require("./CustomerSearchComponent.html")
}).Class({

  // rest of class as previously defined ...

});

module.exports = CustomerSearchComponent;
```

Because this didn't change any behavior, you can use our tests to make sure this change was good. If you run rake, you shouldn't see any test failures. However, you do. Although our acceptance tests are passing, indicating that Shine is still working as expected, we get a strange error from our JavaScript unit tests:

```
> node_modules/.bin/jasmine-node shine/spec/javascripts
Exception loading: shine/spec/javascripts/CustomerSearchComponent.spec.js
shine/webpack/CustomerSearchComponent.html:1
(function (exports, require, module, __filename, __dirname) { <header>
                                                              ^
SyntaxError: Unexpected token <
    at Object.exports.runInThisContext (vm.js:76:16)
```

```
    «stack trace»
```

Although Webpack is handling our production code, and it was configured to know how to interpret our HTML file, when we use Jasmine to run our unit tests, it uses a different implementation of require, and Webpack isn't involved at all. This means Jasmine thinks we're requiring a JavaScript file, and it's trying to interpret it. Because the file is HTML, it can't, and generates this error.

Configuring Webpack to work with our unit tests is extremely difficult. Because you don't actually need the HTML for our unit tests, instead of configuring Webpack for our unit tests, let's arrange for the HTML to not be interpretted instead. You can do that by mocking require itself. Using a library called *Proxyquire*, you can configure our test in a way that says "whenever CustomerSearchComponent.html is required, ignore it."

First, add proxyquire to package.json:

```
{
  "name": "shine",
  "version": "0.0.1",
  "license": "MIT",
  "dependencies": {
    // existing production dependencies...
  },
  "devDependencies": {
    "jasmine": "2.4.0",
    "jasmine-node": "1.11.0",
    "testdouble": "1.6.0",
➤   "proxyquire": "1.7.10"
  }
}
```

After it is installed with npm install, require it in our unit test as the very first line of the file:

complex-views/separate-template/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
var proxyquire = require("proxyquire");
```

The way Proxyquire works is intead of using require in your test, you use the function proxyquire, which uses require under the covers. It additionally allows you to control what happens when certain files are required. You configure this with a second argument to proxyquire, which is an object where the keys are the files being required and the values represent what we want Proxyquire to do when that file is required.

In our case, we want to do absolutely nothing when CustomerSearchComponent.html is required. Proxyquire refers to this concept as "no call through," and is configured like so (this replaces the existing call to require("./CustomerSearchComponent") in the test):

complex-views/separate-template/shine/spec/javascripts/CustomerSearchComponent.spec.js

```
var CustomerSearchComponent = proxyquire(
  "../../webpack/CustomerSearchComponent",
  {
    "./CustomerSearchComponent.html": {
      "@noCallThru": "true"
    }
  }
);
```

With this in place, when you run our tests via rake jasmine, you should see them passing again. This is an unfortunate thing to have to do; however, it *is* simpler than getting webpack configured with our tests. That would completely change all of our test configuration in an unpleasant way, and would result in an even more unusual setup than what we have now.

Now that we've cleaned up our code, let's install and configure Angular's router so we can detect when a user is navigating to a different URL and render a different view.

Configuring 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 you would set it up from the start. We didn't initially, just to keep our introduction to Angular as simple as possible and reduce the number of new concepts you had to absorb.

Now, we'll need to convert our existing Angular app to use the router.

Currently, our Angular app just renders the CustomerSearchComponent's view, since we're using its selector—shine-customer-search—in our Rails view. Instead, we're going to configure Angular's router, which allows us to specify which components are rendered for which URLs the user has navigated to.

To do this, you'll need to make a top-level component for the entire application, whose view will contain markup that the router can hook into. To validate this is working, you'll create a second component that renders a static view and see how changing URLs changes which component is being used. You'll also have to make a small change to our Rails routing configuration to make the URLs bookmarkable.

Create a Bare-Bones Second Component

Now that you know how to put our components and their templates in separate files, creating a shell for our second component is straightforward. First, we'll create a simple static view in webpack/CustomerDetailsComponent.html:

complex-views/separate-template/shine/webpack/CustomerDetailsComponent.html

```
<h1>Customer Details!</h1>
```

We'll require this view in our component's definition, which we'll create in webpack/CustomerDetailsComponent.js:

complex-views/separate-template/shine/webpack/CustomerDetailsComponent.js

```
var reflectMetadata = require("reflect-metadata");
var ng = {
  core: require("@angular/core")
};

var CustomerDetailsComponent = ng.core.Component({
  selector: "shine-customer-details",
  template: require("./CustomerDetailsComponent.html")
}).Class({
  constructor: [
    function() {
    }
  ]
});

module.exports = CustomerDetailsComponent;
```

Next, bring it into our main webpack/application.js:

complex-views/separate-template/shine/webpack/application.js

```
var CustomerDetailsComponent = require("./CustomerDetailsComponent");
```

Finally, pass it to NgModule (which you first learned about in *Validate the Angular Install with a Basic App*, on page 72):

```
var CustomerSearchAppModule = ng.core.NgModule({
  imports: [
    ng.platformBrowser.BrowserModule,
    ng.forms.FormsModule,
    ng.http.HttpModule
  ],
  declarations: [
    CustomerSearchComponent,
➤   CustomerDetailsComponent
  ],
  bootstrap: [ CustomerSearchComponent ]
})
.Class({
  constructor: function() {}
```

```
});
```

We'll make more changes to this code before our system is working again, so if you try the application in this state, it might not work as expected. Hold on as we complete the setup of router. The next step is to bring the router into our code and configure it.

Install and Configure Angular's Router

When we first set up Angular, we added a lot of dependencies to package.json. One was the router. We also are already requireing it in webpack/application.js. All that's left is to configure it. To do that, we'll pass an object describing our routes to the forRoot function of RouterModule, which we can access from ng.router.

complex-views/setup-angular-router/shine/webpack/application.js

```
var routing = ng.router.RouterModule.forRoot(
  [
    {
      path: "",
      component: CustomerSearchComponent
    },
    {
      path: ":id",
      component: CustomerDetailsComponent
    }
  ]
);
```

The paths are somewhat odd, in that they have no leading slashes, and one of them is just the empty string! These paths are considered relative to a base URL that we must specify using the base[2] element. This configuration means that a URL that is *just* the base URL will match the empty string path, and thus use our existing CustomerSearchComponent to handle the view. A URL with an identifier under the base element will match the second path (using a wildcard value similar to what Rails' router does) and use our newly-created CustomerDetailsComponent to handle the view.

I'll talk about the base URL in a moment, but before that, let's finish configuring the router inside our Angular app.

Given the configuration we just saw, you'll need to pass it to NgModule:

```
var CustomerSearchAppModule = ng.core.NgModule({
  imports: [
    ng.platformBrowser.BrowserModule,
    ng.forms.FormsModule,
```

2. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base

```
    ng.forms.HttpModule,
➤   routing
  ],
  declarations: [
    CustomerSearchComponent,
    CustomerDetailsComponent
  ],
  bootstrap: [ CustomerSearchComponent ]
})
.Class({
  constructor: function() {}
});
```

The mere presence of routing in the array given to imports is sufficient for Angular to know what to do and set up routing. Before you configure Rails and see this working, you need to create a top-level application component so that you have a place to tell the router where views should be rendered.

Wrap the App in a Top-Level Component

The way we've created our Angular app isn't common. Most Angular apps have several different components and tend to be designed with a single top-level component that wraps them all. Now that we have a second component (CustomerDetailsComponent), we should move to this design. It also makes the routing we are trying to achieve easier to set up.

First, create CustomerAppComponent by creating webpack/CustomerAppComponent.js like so:

complex-views/setup-angular-router/shine/webpack/CustomerAppComponent.js

```
var ng = {
  core: require("@angular/core")
};
var AppComponent = ng.core.Component({
  selector: "shine-customers-app",
  template: "<router-outlet></router-outlet>"
}).Class({
  constructor: [
    function() { }
  ]
});
module.exports = AppComponent;
```

Note that we've inlined the HTML as just using the element router-outlet. This element is special to Angular and tells it where to render components that are being changed due to routing changes. The reason we've inlined it is just for simplicity—we don't need any more markup in our top-level component.

Next, let's change our Rails view in app/views/customers/index.html.erb to use this new selector:

```
<section id="shine-customer-search">
  <shine-customers-app></shine-customers-app>
</section>
```

Last, we need to require CustomerAppComponent.js into our main application.js and then modify the call to NgModule to tell it about our new top-level component. This requires adding it to the declarations key, but also changing the bootstrap key from CustomerSearchComponent to our new CustomerAppComponent:

complex-views/setup-angular-router/shine/webpack/application.js

```
➤ var CustomerAppComponent = require("./CustomerAppComponent");
var CustomerSearchComponent = require("./CustomerSearchComponent");
var CustomerDetailsComponent = require("./CustomerDetailsComponent");

// router configuration ...

var CustomerSearchAppModule = ng.core.NgModule({
  imports: [
    ng.platformBrowser.BrowserModule,
    ng.forms.FormsModule,
    ng.http.HttpModule,
    routing
  ],
  declarations: [
    CustomerSearchComponent,
    CustomerDetailsComponent,
➤   CustomerAppComponent
  ],
➤ bootstrap: [ CustomerAppComponent ]
})
.Class({
  constructor: function() {}
});
```

This completes the Angular setup, but you still need to set the base element to the right value, as well as do a bit more Rails configuration.

Configure Rails to Work with Angular's Router

Angular makes no prescription about what the server-side should look like—that lack of direction is one of the main reasons for writing this book. The problem with being server-agnostic is we now have to make some complex decisions about how Rails routing and Angular's routing will interact. We can solve this problem by judiciously choosing what our base URL should be.

Suppose we choose the fairly obvious base URL of /customers. That's the URL we're using now to have Rails render the Angular app's initial view. This

means that, in our current configuration, if the user navigates to /customers/42, we'd expect that to render the CustomerDetailsComponent. (Later in this chapter, we'll add the show method to CustomersController so that our Angular app can fetch a single customer's information.) The idiomatic Rails configuration for that endpoint would be the URL /customers/:id, which clashes with our Angular configuration *assuming the base URL is* /customers.

It's possible to make that URL work for both our AJAX requests and for our Angular app, but it's pretty darn confusing. In Angular 1, the part of the path owned by Angular came after an anchor in the URL, so our Angular app's detail view would be available via /customers#/42, which has a *Rails route* of /customers. Angular 2 doesn't do this, so we need a convention where it's obvious what part of the URL is handled by Rails and what part is handled by Angular. In a sense, we need to recreate the delimiter that the octothorpe provided in Angular 1.

This is confusing. It has confused every developer I work with, and has confused many readers of the first edition. In an effort to make what's going on as explicit as possible, let's decide that our base URL is going to be /customers/ng. That means that /customers/ng will show our existing customer search feature, and /customers/ng/42 will show the new CustomerDetailsComponent. It's slightly icky, but it has the virtue of being more explicit.

To make this happen, you'll need to copy app/views/customers/index.html.erb to app/views/customers/ng.html.erb, implement the ng method in CustomersController, and modify our application layout to set the base URL.

Let's change config/routes.rb to account for this new setup. Note that you need two routes to make this work: one for just /customers/ng and a second that *globs* the remaining route. This is so *any* URL that starts with /customers/ng/ will be handled by our new ng method, which makes all of the URLs our Angular app will create bookmarkable.

complex-views/setup-angular-router/shine/config/routes.rb

```
Rails.application.routes.draw do
  devise_for :users
  root to: "dashboard#index"
  # These supercede other /customers routes, so must
  # come before resource :customers
  get "customers/ng", to: "customers#ng"
  get "customers/ng/*angular_route", to: "customers#ng"
  resources :customers, only: [ :index ]
  get "angular_test" => "angular_test#index"
end
```

If you haven't seen route-globbing in Rails before, the string after the asterisk can be anything, and is used to populate params in our controller. In this case, we could access params[:angular_route] to get the Angular-owned part of the route server-side. We won't need this, but using a descriptive name makes it clear to other developers what's going on.

Also note the ordering of our routes. In Rails, the routes are matched in the order listed in config/routes.rb. We want the route /customers/ng to have priority over a route like /customers/42, which we'll add later in this chapter to provide a way to access customer data via AJAX.

Next, let's implement the ng method in CustomersController. Note that we're setting the base URL as an ivar. (I'll explain that in a moment.)

complex-views/setup-angular-router/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController

  PAGE_SIZE = 10

➤  def ng
➤    @base_url = "/customers/ng"
➤  end

  def index

    # method as it was before

  end
end
```

The view is just a copy, but for completeness, this is what it looks like (note the new file name):

complex-views/setup-angular-router/shine/app/views/customers/ng.html.erb

```
<section id="shine-customer-search">
  <shine-customers-app></shine-customers-app>
</section>
```

Finally, you're ready to set the base URL in our application layout. You don't want to hard-code it as /customers/ng. It's possible (and a good architectural pattern) for our Rails application to host many small, single-purpose Angular apps. This design is a great way to scale an application's codebase, and you want to allow for that as Shine grows. So, we'll do something very simple, which is to check if the ivar @base_url has been set by the controller and, if it has, use it to set the base element in our head:

complex-views/setup-angular-router/shine/app/views/layouts/application.html.erb

```
<head>
  <title>Shine</title>
  <%= csrf_meta_tags %>
```

```
    <%= stylesheet_link_tag    'application', media: 'all' %>
    <%= javascript_include_tag 'application' %>
    <%= javascript_include_tag *webpack_asset_paths("application") %>
    <% if @base_url %>
      <base href="<%= @base_url %>">
    <% end %>
  </head>
```

You could create a helper to dynamically derive this and avoid the ivar, but this is a case where a few simple lines of explicit code is better than a highly complex dynamic solution.

You need to do one last thing, which is to redirect from /customers to /customers/ng. You should do this only when a browser requests that URL, which you can easily do inside the respond_to block in the index method:

complex-views/setup-angular-router/shine/app/controllers/customers_controller.rb

```
def index

  # method as it was before

  respond_to do |format|
    format.html {
      redirect_to "/customers/ng"
    }
    format.json {
      render json: { customers: @customers }
    }
  end
```

Now, any existing links to /customers will redirect to /customers/ng, which should render our existing Angular app.

After all this, you can restart Rails, navigate to http://localhost:5000/customers, see that we're redirected to http://localhost:5000/customers/ng, and see our existing customer search feature. You can *also* enter http://localhost:5000/customers/ng/42 into our browser and see our bare-bones customer details page:

You can also verify that things are working by running our tests. Because you haven't (yet) changed how our app functions, our existing tests should still pass. With the router configured, and our new component created, it's now time to add actual navigation between components.

Navigating the User Interface Client-side

With the router and our routes configured, we need to implement the navigation. We want to be able to do two things with what we've set up. First, we need to add a button to each search results that, when clicked, navigates from /customers/ng to /customers/ng/«customer id». Second, we need to extract that customer ID from the URL so we know which customer's details we want to view.

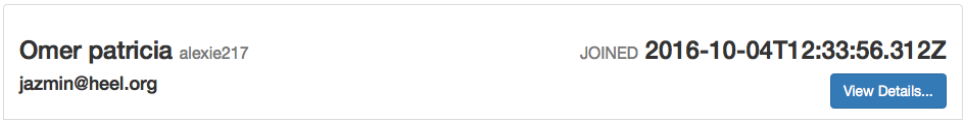
Add Navigation to the Detail View

In *Respond to Click Events*, on page 80, you saw how to respond to click events. What we want to do here is exactly the same thing. First, let's add a button to each search result that the user will use to view a single customer's details. We'll add it to the bottom right of our search result using Bootstrap's pull-right class:

complex-views/navigation-to-new-view/shine/webpack/CustomerSearchComponent.html

```
<ol class="list-group">
  <li *ngFor="let customer of customers"
      class="list-group-item clearfix">
    <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>
➤   <div class="pull-right">
➤     <button class="btn btn-small btn-primary"
➤             on-click="viewDetails(customer)">
➤       View Details...
➤     </button>
➤   </div>
    <h4>{{customer.email}}</h4>
  </li>
</ol>
```

When the page is reloaded, you should see the button, nicely located inside each search results:



The on-click attribute we used is assuming the existence of a function called viewDetails in our component, so let's add that next.

We want to navigate the user to /customers/ng/«customer.id» inside viewDetails. You'll recall that Rails owns the route /customers/ng, so we need to tell Angular's router to simply route to customer.id. That will trigger the second configured route (the one with the path ":id") and show CustomerDetailsComponent. To do this, we need access to a Router,[3] which is available via ng.router.

First, we need to require that inside CustomerSearchComponent:

complex-views/navigation-to-new-view/shine/webpack/CustomerSearchComponent.js

```
var reflectMetadata = require("reflect-metadata");
var ng = {
  core:   require("@angular/core"),
  http:   require("@angular/http"),
  router: require("@angular/router")
};
```

We next need to tell Angular to provide an instance of the router to our component by adding it to the special constructor object:

complex-views/navigation-to-new-view/shine/webpack/CustomerSearchComponent.js

```
constructor: [
  ng.http.Http,
  ng.router.Router,
  function(http,router) {
    this.customers = null;
    this.http      = http;
    this.router    = router;
    this.keywords  = "";
  }
],
```

Now, we can implement viewDetails. To do this, call the navigate method and pass it an array of the parts of the URL we want to send the user to. It uses an array so that we don't have build up a string, though in our case, we only have one part of our URL—the customer's ID.

complex-views/navigation-to-new-view/shine/webpack/CustomerSearchComponent.js

```
var CustomerSearchComponent = ng.core.Component({

  // rest of the component ...

  viewDetails: function(customer) {
    this.router.navigate(["/", customer.id]);
  },
```

3. https://angular.io/docs/ts/latest/api/router/index/Router-class.html

Now, when you reload the page, perform a search, and click on the new "View Details" button, we're navigated to CustomerDetailsComponent. You'll notice the URL contains the ID of the customer you clicked on and, if you look at the Rails logs, you'll see that we did *not* hit the server. The navigation was all performed client-side.

Let's build out the detail view just a little bit so we can see how to extract that ID from the URL. We'll use that ID to fetch the details for that customer from the server via AJAX in the final section of this chapter.

Extract Details from the Route

In a Rails application, every controller has access to the params hash that contains, among other things, the parameters extracted from the route. For example, if your route is /customers/42, you'll likely use params[:id] to get the 42 from that route. With Angular, it's slightly more complicated. This is partly due to the asynchronous nature of JavaScript, but also due to Angular's overall design philosophy of explicit configuration over implicit behavior through convention.

To get access to the route parameters, you need to ask Angular for an instance of the ActivatedRoute[4] for the route we've navigated the user to. An ActivatedRoute has a method params, which returns an Observable. That Observable will provide the routing parameters to us. You'll recall observables from *Get Data from the Back End*, on page 86 when we integrated our Angular app with our Rails back-end.

For now, we'll just extract the customer ID from the route and display that in the view to prove that everything's hooked up properly. This means we have to figure out where we can access the ActivatedRoute. It might seem intuitive that the constructor would be the place to examine the ActivatedRoute, but the constructor is only called once. You need a method that's called every time the component is about to be displayed. In Angular, that method is ngOnInit, which is part of the OnInit interface. Interfaces are a concept heavily used in TypeScript to describe "hook" methods we can implement to get certain behaviors. Because ES5 has no concept of interfaces, all you have to do is provide an implementation of ngOnInit and Angular will call it (this is part of many *lifecycle hooks*[5] Angular provides).

Let's see what this looks like. First, require Angular's router:

4. https://angular.io/docs/ts/latest/api/router/index/ActivatedRoute-interface.html
5. https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

complex-views/navigation-to-new-view/shine/webpack/CustomerDetailsComponent.js

```
var reflectMetadata = require("reflect-metadata");
var ng = {
  core:   require("@angular/core"),
  router: require("@angular/router")
};
```

Next, tell Angular that we want the ActivatedRoute given to us. We do that by adding it to the special constructor array:

complex-views/navigation-to-new-view/shine/webpack/CustomerDetailsComponent.js

```
var CustomerDetailsComponent = ng.core.Component({
  selector: "shine-customer-details",
  template: require("./CustomerDetailsComponent.html")
}).Class({
  constructor: [
    ng.router.ActivatedRoute,
    function(activatedRoute) {
      this.activatedRoute = activatedRoute;
      this.id             = null;
    }
  ],
});
```

Note that we've also initialized id to null. This is the property we'll set from the route and that we'll show in the view.

To set it, implement ngOnInit. This is less straightforward than you'd think because we have to deal with the observable. As before, use subscribe to configure the callback that should be given the results of the observable. In this case, it's an object called params. Also, note that you have to use the self trick you saw in *Get Data from the Back End*, on page 86 to ultimately set the id property to the value from the route.

complex-views/navigation-to-new-view/shine/webpack/CustomerDetailsComponent.js

```
var CustomerDetailsComponent = ng.core.Component({
  selector: "shine-customer-details",
  template: require("./CustomerDetailsComponent.html")
}).Class({

  // constructor ...

  ngOnInit: function() {
    var self = this;
    self.activatedRoute.params.subscribe(
      function(params) {
        var id  = +params['id'];
        self.id = id;
      }
    );
  }
```

```
});
```

Finally, let's modify webpack/CustomerDetailsComponent.html to show the customer id we've extracted from the route:

complex-views/navigation-to-new-view/shine/webpack/CustomerDetailsComponent.html

```
<section class="customer-details">
  <h1>Customer Details {{id}}</h1>
</section>
```

I'll explain the reasoning for the section tag in a moment. With all this done, you should be able to reload the customer search page, perform a search, click the "View Details" button, and see our customer details page showing that customer's ID:

We're almost done. We'd now like to use that ID to query the back-end and get the customer's details so that we can show them in our new component. We also want this new feature covered by tests. Let's take this opportunity to use TDD to add this final feature, so we can see what that workflow is like with all the new tools and libraries we've learned.

Implementing Back-End Integration Using TDD

You already know how to fetch the customer's details from the server—you'd do it just the way you did in *Get Data from the Back End*, on page 86. You've also seen how to write acceptance and unit tests for a feature like this, but another example could really help cement your understanding of these complex concepts. It's also worth seeing how a TDD-based flow could work, for those of you that prefer test-driven development.

In this section, we'll create an acceptance test that works with the client-side navigation we just implemented, then we'll create a unit test for ngOnInit that confirms that it's calling the back-end. When we write code in our Angular component to get that to pass, our acceptance test will pass as well. This should give you a basic understanding of how to use TDD to build additional features.

Acceptance Test for Navigation to the Details View

In *Running Headless Acceptance Tests in PhantomJS*, on page 103, you learned about writing acceptance tests using Capybara and PhantomJS. spec/features/customer_search_spec.rb is the result of our efforts. Here we'll augment the test named "Search by Email" to navigate to one customer's detail view.

After the existing code for this test, we'll want to find the first button labeled "View Details," click it, wait for the component to re-render, and fetch the customer details. Then, we'll make some expectations about what content we expect to see on the details page. You already have all the tools in your toolbox to do this, so let's see the code.

complex-views/fetch-one-customer/shine/spec/features/customer_search_spec.rb

```
require 'rails_helper'

feature "Customer Search" do

  # Existing test setup as before ...

  scenario "Search by Email" do
    within "section.search-form" do
      fill_in "keywords", with: "pat123@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("Pat")
      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("Patricia")
      expect(page.all("ol li.list-group-item")[1]).to have_content("Dobbs")

      expect(page.all("ol li.list-group-item")[3]).to have_content("I.T.")
      expect(page.all("ol li.list-group-item")[3]).to have_content("Pat")
    end

➤   click_on "View Details...", match: :first
➤
➤   customer = Customer.find_by!(email: "pat123@somewhere.net")
➤   within "section.customer-details" do
➤     expect(page).to have_content(customer.id)
➤     expect(page).to have_content(customer.first_name)
➤     expect(page).to have_content(customer.last_name)
➤     expect(page).to have_content(customer.email)
➤     expect(page).to have_content(customer.username)
➤   end
  end
end
```

When using click_on, Capybara throws an exception if there is more than one match. In our case, that will definitely be true, so we use match: :first to tell Capybara to click on the first button it finds. Because we did an email-based search, you know that the first result is the customer with the email pat123@somewhere.net, so after we click the button, we'll find that customer in our database.

To wait for the page to reload, we'll use within as we did before. This is why we used a section element earlier—it's something we can configure to only appear once we have our results. Finally, we'll simply assert that the customer's various bits of data are somewhere on the screen.

Running this, it will fail, but this is expected since we're doing TDD.

```
> rspec spec/features/customer_search_spec.rb

«Irrelevant output»

Failures:

  1) Customer Search Search by Email
     Failure/Error: expect(page).to have_content(customer.first_name)
       expected to find text "Pat" in "Customer Details 5"
     # ./spec/features/customer_search_spec.rb:95:in `block (3 levels)…
     # …gems/ruby-2.3.1@dcbang2/gems/capybara-2.7.1/lib/capybara/sessi…
     # …gems/ruby-2.3.1@dcbang2/gems/capybara-2.7.1/lib/capybara/dsl.r…
     # ./spec/features/customer_search_spec.rb:93:in `block (2 levels)…

Finished in 9.42 seconds (files took 2.13 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/features/customer_search_spec.rb:71 # Customer Search Sea…
```

When this passes, we'll know we've completed our task. Let's next write a unit test for ngOnInit, since that's doing all the hard work to make this feature happen.

Unit Test for ngOnInit

You should already know how to write this test, but there's a few gotchas that are hard to track down, related to the mocking you'll need to do. But first, we'll create spec/javascripts/CustomerDetailsComponent.spec.js in a similar fashion to our existing unit test in spec/javascripts/CustomerSearchComponent.spec.js:

complex-views/fetch-one-customer/shine/spec/javascripts/CustomerDetailsComponent.spec.js

```
var proxyquire = require("proxyquire");
var CustomerDetailsComponent = proxyquire(
  "../../webpack/CustomerDetailsComponent",
  {
    "./CustomerDetailsComponent.html": {
      "@noCallThru": "true"
    }
  }
);
var td = require("testdouble");

var component = null;
```

```
describe("CustomerDetailsComponentComponent", function() {
  describe("initial state", function() {
    beforeEach(function() {
      component = new CustomerDetailsComponent();
    });
    it("sets customer to null", function() {
      expect(component.customer).toBe(null);
    });
  });

  describe("ngOnInit", function() {

    // ...

  });
});
```

This is the same boilerplate you've seen before, plus a basic test of our constructor (namely, that it sets its customer property to null). Next, let's create the shell of our test for ngOnInit. The contract of ngOnInit is to set the property customer to an object it receives from the back-end. Let's assume two helper functions exist called createMockHttp and createMockRoute, which will provide the test doubles we can provide to CustomerDetailsComponent's constructor.

The basics of the test look like so:

complex-views/fetch-one-customer/shine/spec/javascripts/CustomerDetailsComponent.spec.js

```
describe("ngOnInit", function() {

  // ...

  var customer = {
    id: 1,
    created_at: (new Date()).toString(),
    first_name: "Pat",
    last_name: "Jones",
    username: "pj",
    email: "pjones@somewhere.net"
  }

  // more setup to come ...

  beforeEach(function() {
    var route = createMockRoute(customer.id);
    var http  = createMockHttp(customer);

    component = new CustomerDetailsComponent(route,http);
  });

  it("fetches the customer from the back-end", function() {
    component.ngOnInit();
    expect(component.customer).toBe(customer);
  });
});
```

The implementation of createMockHttp will look more or less the same as our mocking setup in spec/javascripts/CustomerSearchComponent.spec.js:

complex-views/fetch-one-customer/shine/spec/javascripts/CustomerDetailsComponent.spec.js

```
var createMockHttp = function(customer) {
  var response = td.object(["json"]);
  td.when(response.json()).thenReturn({ customer: customer });

  var observable = td.object(["subscribe"]);
  td.when(observable.subscribe(
    td.callback(response),
    td.matchers.isA(Function))).thenReturn();

  var mockHttp = td.object(["get"]);

  td.when(
    mockHttp.get("/customers/" + customer.id + ".json")
  ).thenReturn(observable);

  return mockHttp;
}
```

You could certainly generalize this and extract so both tests could use it, but I'll leave that an exercise for you to do on your own.

Finally, we need to implement createMockRoute. Our code accesses the params key on the ActivatedRoute, and is expecting the value to be an observable. So, createMockRoute will return a simple objects with a params key that maps to a test double observable that we can set up in a similar fashion to how we set it up for mocking Http.

Ultimately, we want our callback to subscribe to be given an object that maps id to the customer's ID.

complex-views/fetch-one-customer/shine/spec/javascripts/CustomerDetailsComponent.spec.js

```
var createMockRoute = function(id) {
  var observable  = td.object(["subscribe"]);
  var routeParams = { "id" : id };
  var mockActivatedRoute = { "params": observable };

  td.when(observable.subscribe(
    td.callback(routeParams),
    td.matchers.isA(Function)
  )).thenReturn();

  return mockActivatedRoute;
}
```

If you run these tests now, you should see some failures:

```
> rake jasmine
Failures:

 1) CustomerDetailsComponentComponent initial state sets customer to null
```

```
Message:
  Expected undefined to be null.
Stacktrace:
  Error: Expected undefined to be null.
at (shine/spec/javascripts/CustomerDetailsComponent.spec.js:21:34)

2) CustomerDetailsComponentComponent ngOnInit fetches the customer
                                     from the back-end
Message:
  Expected undefined to be { id : 1, created_at : 'Sun Sep 18 2016…
Stacktrace:
  Error: Expected undefined to be { id : 1, created_at : 'Sun Sep…
at (shine/spec/javascripts/CustomerDetailsComponent.spec.js:84:34)
```

Your output might not be exact, but it *is* important that your failures are the right ones. You want your test to fail at your test expectations. For the first test, we want to see a failure because customer *hasn't* been set to null. For the second test, we want it to fail because customer hasn't been set to the test customer we arranged Http.get to return.

Now that we've specified both the behavior of CustomerDetailsComponent as well as the behavior of Shine as a whole, let's write the code to finish our feature and make these tests pass.

Make our Tests Pass by Implementing the Back-End Integration

We'll start with CustomerDetailsComponent. Let's tackle the easy issue first and make our constructor test pass. We'll simply set customer to null as expected by the test:

complex-views/fetch-one-customer/shine/webpack/CustomerDetailsComponent.js

```
var CustomerDetailsComponent = ng.core.Component({
  selector: "shine-customer-details",
  template: require("./CustomerDetailsComponent.html")
}).Class({
  constructor: [
    ng.router.ActivatedRoute,
    function(activatedRoute,http) {
      this.activatedRoute = activatedRoute;
➤     this.customer = null;
    }
  ],
```

If you re-run the tests, you should see that the first one is now passing.

Next, we'll change ngOnInit to perform the HTTP request to get the customer's details. First, we'll need to require http and inject an instance of Http into our component:

complex-views/fetch-one-customer/shine/webpack/CustomerDetailsComponent.js

```
var reflectMetadata = require("reflect-metadata");
var ng = {
  core:   require("@angular/core"),
  http:   require("@angular/http"),
  router: require("@angular/router")
};

var CustomerDetailsComponent = ng.core.Component({
  selector: "shine-customer-details",
  template: require("./CustomerDetailsComponent.html")
}).Class({
  constructor: [
    ng.router.ActivatedRoute,
    ng.http.Http,
    function(activatedRoute,http) {
      this.activatedRoute = activatedRoute;
      this.http = http;
      this.customer = null;
    }
  ],
```

Now, we can use http to make our request of the server. This will look similar to how you fetched search results in *Get Data from the Back End*, on page 86, but we'll put this code inside the callback related to the route parameters. Because callbacks within callbacks is quite messy, let's pull everything into local functions. We'll start by holding a reference to this in self (which should be there already), and then definiting a generic error handling function to pass in as our error handler:

complex-views/fetch-one-customer/shine/webpack/CustomerDetailsComponent.js

```
ngOnInit: function() {
  var self = this;
  var observableFailed = function(response) {
    window.alert(response);
  }

  // more to come ...

}
```

Next, we'll define customerGetSuccess as the callback that will receive the response from the back-end, and routeSuccess as the callback that will be called with the route parameters:

complex-views/fetch-one-customer/shine/webpack/CustomerDetailsComponent.js

```
var customerGetSuccess = function(response) {
  self.customer = response.json().customer;
}
var routeSuccess = function(params) {
  self.http.get(
```

```
    "/customers/" + params['id'] + ".json"
  ).subscribe(
    customerGetSuccess,
    observableFailed
  );
}
```

Finally, we'll make our call to activatedRoute.params.subscribe:

```
self.activatedRoute.params.subscribe(routeSuccess);
```

Running our Jasmine tests will *still* show a failure, however. It's as if our mock observable callback isn't being called. If you were to pepper your code with calls to console.log or debug into your code, you'd see it's true—our callback isn't being called.

If you look closely at our test, we're telling Testdouble.js that subscribe is expecting two arguments: both functions. If you look at our actual code, we're only passing one argument to subscribe. Testdouble.js notices this and assumes we don't want *this* call to subscribe mocked. Oops!

I'm pointing this out as it's a very subtle failure on our part, and hard to track down (it took me over an hour to figure this out when I was first writing this example). It's fixed by passing our observableFailed function as the second argument:

```
self.activatedRoute.params.subscribe(routeSuccess,observableFailed);
```

The complete ngOnInit function now looks like so:

complex-views/fetch-one-customer/shine/webpack/CustomerDetailsComponent.js

```
ngOnInit: function() {
  var self = this;
  var observableFailed = function(response) {
    window.alert(response);
  }

  var customerGetSuccess = function(response) {
    self.customer = response.json().customer;
  }
  var routeSuccess = function(params) {
    self.http.get(
      "/customers/" + params['id'] + ".json"
    ).subscribe(
      customerGetSuccess,
      observableFailed
    );
  }

  self.activatedRoute.params.subscribe(routeSuccess,observableFailed);
}
```

And now, our Jasmine tests should pass:

```
> rake jasmine
node_modules/.bin/jasmine-node shine/spec/javascripts
............

Finished in 0.032 seconds
12 tests, 10 assertions, 0 failures, 0 skipped
```

If you re-run the acceptance test, it's still failing, but in a different way:

```
> rspec spec/features/customer_search_spec.rb

Randomized with seed 42474

Customer Search
http://localhost:3808/webpack-dev-server/
webpack result is served from //localhost:3808/webpack/
content is served from /Users/davec/Projects/rails-book/shine
Angular 2 is running in the development mode. Call enablePro…
  Search by Email (FAILED - 1)
Angular 2 is running in the development mode. Call enablePro…
  Search by Name
Killing webpack-dev-server

Failures:

  1) Customer Search Search by Email
 Failure/Error: raise ActionController::RoutingError,
 "No route matches [#{env['REQUEST_METHOD']}] #{env['PATH_INFO'].inspect}"

     ActionController::RoutingError:
       No route matches [GET] "/customers/5.json"

  «stack trace»

Failed examples:

rspec ./spec/features/customer_search_spec.rb:71
```

Because our code is now getting far enough to hit Rails, Rails is returning a 404, since there is no endpoint yet to find a customer. You can add it with a few lines of code.

First, add :show as an allowed action for our customers resource in config/routes.rb:

complex-views/fetch-one-customer/shine/config/routes.rb

```
Rails.application.routes.draw do
  devise_for :users
  root to: "dashboard#index"
  get "customers/ng",                to: "customers#ng"
  get "customers/ng/*angular_route", to: "customers#ng"
➤ resources :customers, only: [ :index, :show ]
  get "angular_test" => "angular_test#index"
end
```

Next, add a few lines of code to CustomersController to implement show and return the customer being requested:

complex-views/fetch-one-customer/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController
  def show
    customer = Customer.find(params[:id])
    respond_to do |format|
      format.json { render json: { customer: customer } }
    end
  end
```

If you re-run the tests now, they still fail, but we get a new failure around missing content:

```
> rspec spec/features/customer_search_spec.rb

«output»

Failures:

  1) Customer Search Search by Email
     Failure/Error: expect(page).to have_content(customer.id)
       expected to find text "5" in "Customer Details"

     «stack trace»

Failed examples:

rspec ./spec/features/customer_search_spec.rb:71 # Customer Search Search by Email
```

To fix this, we'll create a simple view that shows the content we're expecting in webpack/CustomerDetailsComponent.html:

complex-views/fetch-one-customer/shine/webpack/CustomerDetailsComponent.html

```
<section class="customer-details" *ngIf="customer">
  <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}}
  </h5>
</section>
```

Note the use of *ngIf on customer. This is how our acceptance test can successfully wait for the page to be rendered. Angular won't even show the section tag until customer has a value, so the within we're using will wait for it to appear. This prevents you from having flaky tests. We've also put the content in the appropriate h tags based on its importance. Now, when we run our acceptance test, it passes!

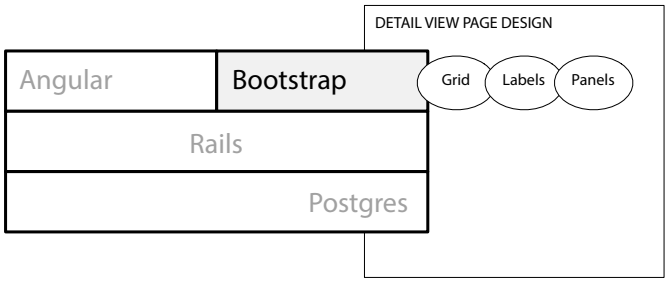
```
> rspec spec/features/customer_search_spec.rb

Randomized with seed 11944

Customer Search
http://localhost:3808/webpack-dev-server/
webpack result is served from //localhost:3808/webpack/
content is served from /Users/davec/Projects/rails-book/shine
  Search by Name
  Search by Email
Killing webpack-dev-server

Finished in 6.81 seconds (files took 1.89 seconds to load)
2 examples, 0 failures

Randomized with seed 11944
```

More important, when you try it in the browser, everything works great:

We've now turned our simple search into a small single-page application using Angular. You learned how to configure Angular's routing as well as a better way to manage your HTML templates. You also got a realistic view of how you can use TDD to drive out our features.

Next: Design Using Grids

The power we've just gained in using Angular is great, but our customer details view isn't that useful. It's now time to focus back on Bootstrap and use its *grid* in order to design a better details view. Grid-based design, and Bootstrap's implementation of it, will unlock your inner web designer and give you a great ability to mock up, design, and implement a complex user interface, without writing any CSS.



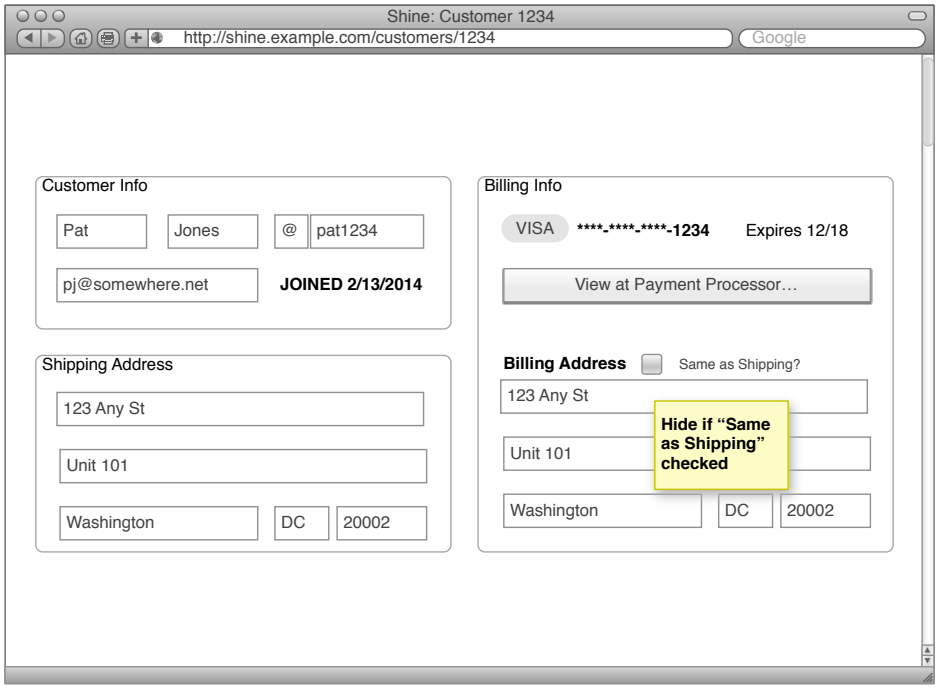
CHAPTER 9

Design Great UIs with Bootstrap's Grid and Components

At the start of Chapter 8, *Create a Single-Page App*, on page 131, you saw a mock-up of the customer detail view. We're going to build this page now to learn about the true power of Bootstrap—its grid. We'll also examine some of Bootstrap's many components, which will allow us to create a polished and visually appealing user interface.

We'll tackle this topic in three parts. First, I provide some background on grid-based design. This will help you understand why Bootstrap is based on a grid, and how you can break down any UI into grids to make your work easier. Next, you'll lay out the customer detail screen I hinted at in the previous chapter, using Bootstrap's grid to make it easy. Finally, you'll use various Bootstrap components, such as panels and labels, to polish up our UI. By the end of the chapter you'll have a solid foundation in building user interfaces with Bootstrap, and even a bit of confidence in designing them yourself. You'll still be a far cry from being a "real" web designer, but you'll be able to do common, simple tasks on your own.

You saw the mock-up in the previous chapter, but here it is again, in a slightly expanded form, so you know where we're headed:



Now, let's learn about the grid and how it helps 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.

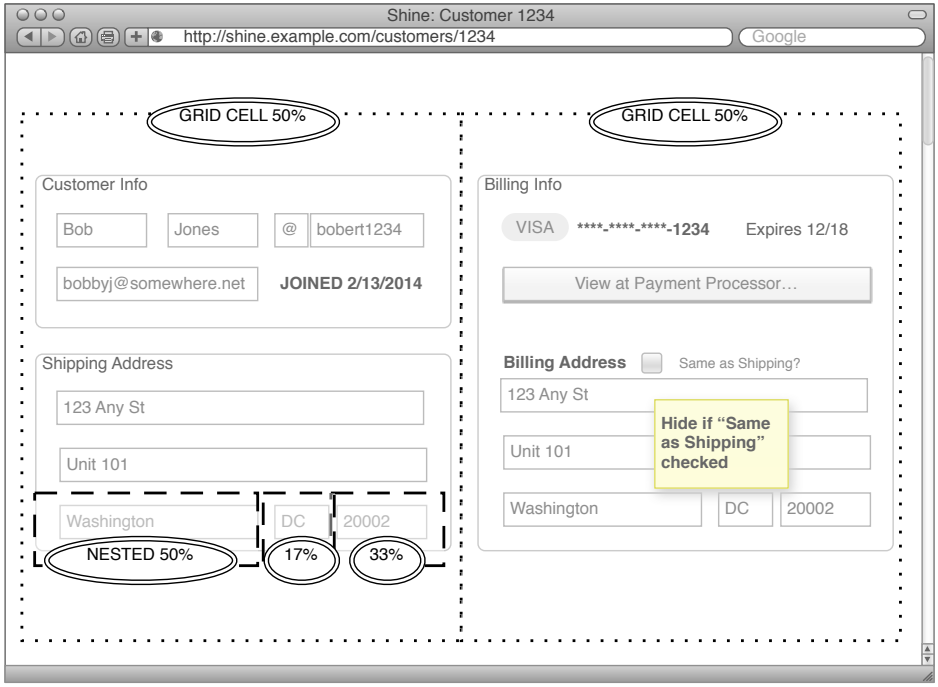
Like functional decomposition in programming, a *grid* is how you can break down a user interface into smaller parts. You can focus on each part of the design, and rely on the grid to keep everything looking visually cohesive.

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*, on page 57) use 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, which allows us to place content into side-by-side columns. 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. Take a look at the following figure.



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 is that, if we 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 information 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 information and shipping address as a grid-within-a-grid.

Lay 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 lay it all out in a grid (see the sidebar on page 163 for some details on why we're using divs).

The outer *div* has the class row, which tells Bootstrap we're going to place columns inside it. The divs inside the row has class col-md-*X* where *X* is the number of columns, out of 12, that this particular column should take up. As we want two equal-sized columns, we want each of *our* columns to take up six of Bootstrap's. Thus, each div will get the class col-md-6 (see Chapter 13, *Dig Deeper*, on page 201 for what the -md- means).

We can add this markup to webpack/CustomerDetailsComponent.html, replacing the bare-bones markup we had there from the last section.

```
<section class="customer-details" *ngIf="customer">
<form><div class="row">
  <div class="col-md-6">
    <h1>Customer</h1>
  </div>
  <div class="col-md-6">
    <h1>Billing Info</h1>
  </div>
```

Why Do We Sometimes Use div and Sometimes Not?

Before HTML5, there weren't a lot of standard elements you could use to describe your 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, more meaningful elements are available, such as article, section, header, and footer.

Because of this, the W3C recommends that div be used only as a last resort,[a] when no other elements are available.

What this means is that you want to use the right tags when describing your content, regardless of the visualization you are going for. You can then use div tags to achieve the layouts you want. Because div is semantically meaningless, it allows anyone reading your 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 divs 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

```
</div></form>
</section>
```

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 earlier, we can think of each section of our page as having a nested grid inside this one. Bootstrap's grid works exactly this way.

Build 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 information 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. The first name, last name, and username are all about the same size data-wise, so we can create three equal-sized columns for them. Because 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`.

```
<section class="customer-details" *ngIf="customer">
<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 commonly done 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 doesn't get messed up by styling classes. Also, we can add more styling later without worrying about how the grid will affect it. If you look at what we've done in our browser (take a look at the following figure), you can see that it looks pretty good!

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.

Use Grid Cells of Different Sizes

The main columns of our view, as well as the user information, all used grid cells of the same size. That won't work for the address views, as the city, state, and zip code are all different sizes. It also won't work for the credit card information 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—two characters—and the zip code 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 zip code 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 zip code (the two street address lines can use up an 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>
```

You can repeat this markup for the billing address, which just leaves us the credit card information section to style.

Laying Out the Credit Card Info

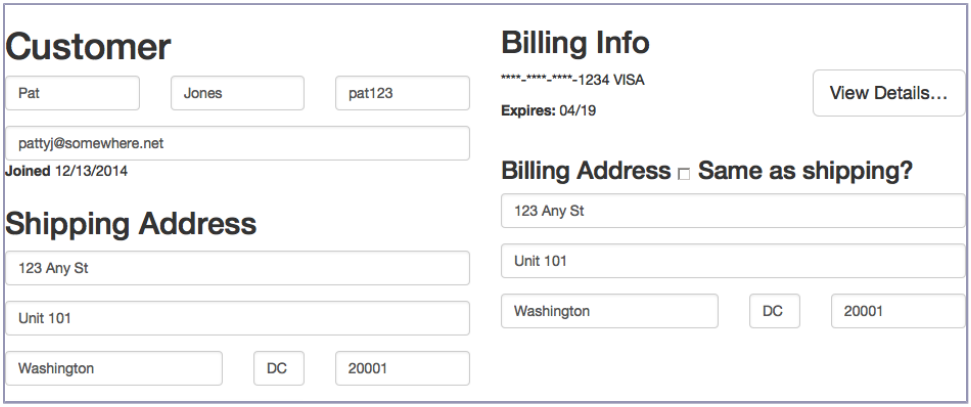
The credit card area has two distinct parts: the card information itself, and the button that will link the user to the payment processor's page for that

card. We'll give the card information seven of the twelve 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 address -->
```

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, you can see in the figure that follows 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 for 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.

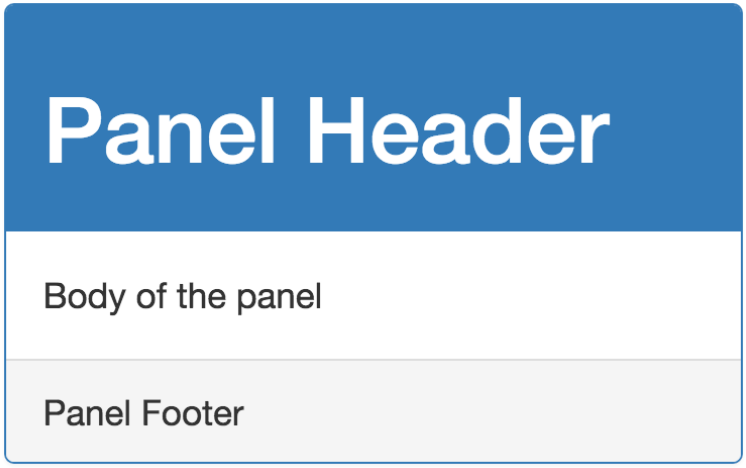
Adding Polish with Bootstrap Components

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, no clear distinction exists between the three sections of the view, and the credit card information is a bit jumbled, as 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 users to navigate all the data with their eyes. We can get a long way with the *panel* component, which is a box surrounding our content along with a header and footer.

Use Panels

A panel looks like so:



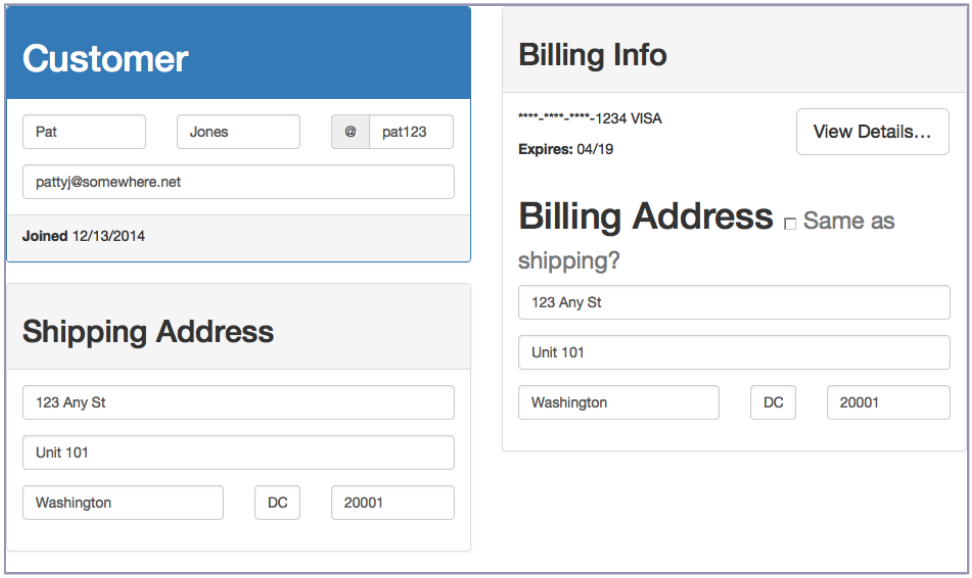
It can be created with Bootstrap with markup like this:

```
<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 information panel styled differently from the other two so that 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.

As shown in the following screen, the result looks pretty nice.



Next, let's improve the credit card information 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 using *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:

Last, we'll make a few adjustments to the typography—the headers are a bit too large.

Use 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.

You 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). You can also use them on the `p` tag surrounding the credit card number to make it stand out a bit.

You'll see the entire markup in a moment, but here's an example of what I'm 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 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 that underlies all of the type.

One last bit of polish I'd like to add is to distinguish the username from the first and last names in the Customer Information section. You'll notice on our mock-up, the username was preceded with an @ symbol. Bootstrap makes this easy using *form add-ons*.

Form Add-Ons

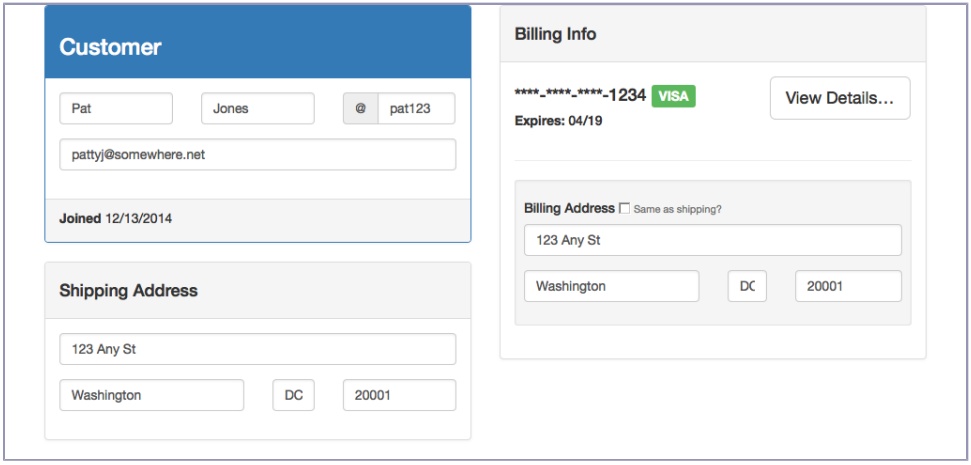
Often, a symbol prepended (or appended) to a value can give it enough context for users to understand what it means, without using a label. This can be handy on dense pages like our customer detail view. Because the first row of the customer information section is just three equal-sized strings, it might 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, as shown in the screen that follows, embodying the spirit of the mock-up.



HTML markup is quite verbose (especially because you have to heavily wrap it to fit the margins in this book), so you've only seen bits and pieces. The entire screen's markup can be seen in Appendix 1, *Full Listing of Customer Detail Page HTML*, on page 203, although it will be easier to examine it by downloading the sample code.[1]

Note that you *still* haven't written any CSS. We were able to create a highly complex form that displays 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, you 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 that allow you to design and build in the browser.

This eliminates 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.

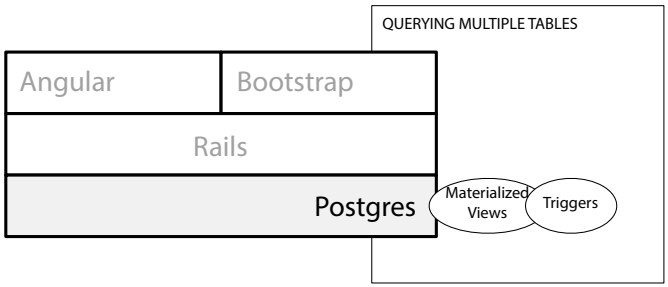
Next: Populating the View Easily and Efficiently

This chapter focused on the top of the stack: the view. You learned how to configure our Angular app to allow for routing to different views, backed by

1. https://pragprog.com/titles/dcbang2/source_code

different controllers. You saw how easy it is to design a complex UI for our customer detail view.

In the next chapter, you'll learn 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. You'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 you need to query data stored in several tables, there is a trade-off. Either you keep our code simple by using Active Record—which makes several queries to the database—or you make your code more complex by using a single, efficient query specific to your needs. Performance is an issue in both cases because you're pulling back a lot of data. Postgres solves this dilemma with its *materialized views* feature, which provides clean code, accesses data with a single query, and exhibits high performance.

In this chapter, we'll continue to build on our running example where we display a customer's details. To do this, we need to fetch data from five different tables. You'll see how the idiomatic *Rails Way*, using Active Record, actually results in our having to make seven queries to the database. In contrast, a single, more direct query results in convoluted code but with potentially better performance.

We'll then create a materialized view of our single query. A materialized view is part view and part table. Like a view, it's backed by a query that is the source of its data. Like a table, the data is stored on disk. The materialized view provides a way to update the data on disk by rerunning the backing query. This means you get to use Active Record in our Rails code in an idiomatic way but still get extremely high performance: the best of both worlds! You'll also see how we can use *database triggers* to automatically keep the view up-to-date.

First, let's look at the performance characteristics of the two approaches—Active Record 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*, on page 159, you built the user interface for the data we'll be querying here. In addition to the data you've already seen in the CUSTOMERS table, we need to display the customer's billing address, shipping address, and credit card information. The credit card information is stored elsewhere (we'll deal with that in the next chapter), so we're just querying the customer's billing and shipping addresses for now.

You've already seen the CUSTOMERS table, and you know it doesn't include either of these pieces of data. As you'll recall from Chapter 4, *Perform Fast Queries with*, on page 39, our hypothetical company has tables in a shared database, which Shine can access. In this case, we'll assume that the tables we need to access billing and shipping addresses are also available to us via the shared database.

Let's look at how well both Active Record and a single query perform when accessing all of these tables together. We start with the structure so we know what we're dealing with.

The Tables We'll Query

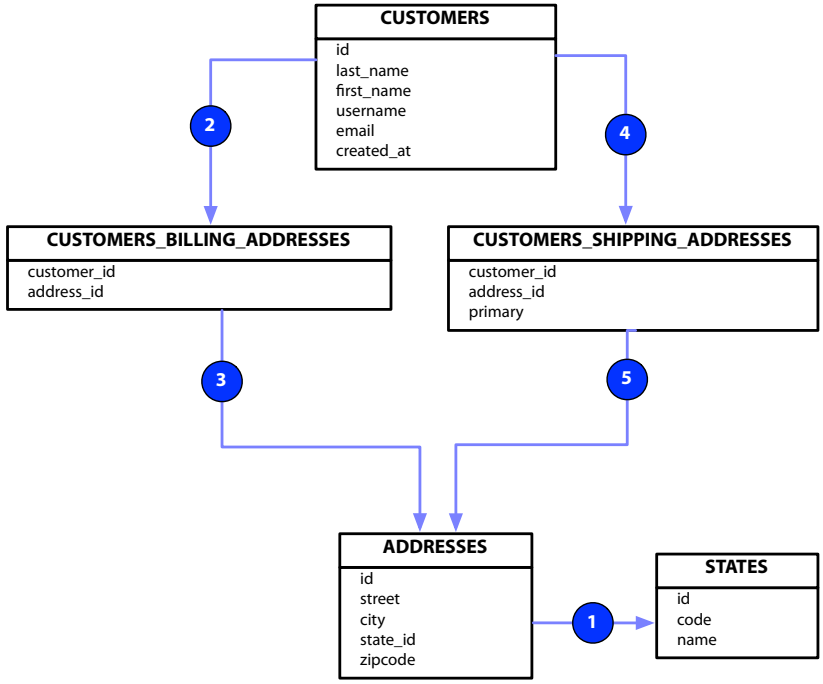
The diagram on page 177 shows the tables we'll be accessing and their relationships to one another.

In our hypothetical system, all addresses are stored in the table ADDRESSES. This table contains fields you'd expect in a U.S.-style address, except for the postal codes for U.S. states. The states are stored in STATES, and ADDRESSES references that table via the column STATE_ID (marked "1" on the diagram).

Because 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 called CUSTOMERS_BILLING_ADDRESSES and CUSTOMERS_SHIPPING_ADDRESSES.

In the diagram, you can see that the relationship marked "2" references a customer via CUSTOMER_ID, while the relationship marked "3" references an address via ADDRESS_ID. This is how you join the two tables together.

CUSTOMERS_SHIPPING_ADDRESSES uses a similar structure, with the relationships marked "4" and "5." You'll notice, however, that there's a column called PRIMARY. This is because, in our hypothetical system, a customer may have more than



one shipping address. PRIMARY denotes their primary shipping address, and this is the one we'll display in Shine.

Just as we did with CUSTOMERS in *Set Up the New Table and Data*, on page 40, we'll set up these new tables for our local development. In the real world, these would already exist, but we have to create them manually here in order to follow this example.

Let's do that and then populate these tables with sample data. First, create an empty migration via rails g migration add_addresses. Next, fill in that file like so:

materialized-views/data-model/shine/db/migrate/20161006115532_add_addresses.rb

```
class AddAddresses < ActiveRecord::Migration[5.0]
  def change
    create_table :states do |t|
      t.string     :code, size: 2, null: false
      t.string     :name,          null: false
    end
    create_table :addresses do |t|
      t.string     :street,        null: false
      t.string     :city,          null: false
      t.references :state,         null: false
      t.string     :zipcode,       null: false
    end
    create_table :customers_billing_addresses do |t|
```

```
      t.references :customer,      null: false
      t.references :address,       null: false
    end
    create_table :customers_shipping_addresses do |t|
      t.references :customer,      null: false
      t.references :address,       null: false
      t.boolean    :primary,       null: false, default: false
    end
  end
end
```

Finally, run rails db:migrate to create the tables. Although we won't ultimately be using the ActiveRecord models for these tables, let's create them anyway, as it'll make it easier to poke around in the Rails console to evaluate their performance. It will also make it easier to populate some seed data.

To do this, you'll need to create the files state.rb, address.rb, customers_billing_address.rb, and customers_shipping_address.rb in app/models like so:

materialized-views/actual-materialized-view/shine/app/models/address.rb

```
class Address < ActiveRecord::Base
  belongs_to :state
end
```

materialized-views/actual-materialized-view/shine/app/models/state.rb

```
class State < ActiveRecord::Base
  has_many :addresses
end
```

materialized-views/actual-materialized-view/shine/app/models/customers_billing_address.rb

```
class CustomersBillingAddress < ActiveRecord::Base
  belongs_to :address
  belongs_to :customer
end
```

materialized-views/actual-materialized-view/shine/app/models/customers_shipping_address.rb

```
class CustomersShippingAddress < ActiveRecord::Base
  belongs_to :address
  belongs_to :customer
end
```

You'll also need to modify app/models/customer.rb to add the Active Record relationships:

materialized-views/actual-materialized-view/shine/app/models/customer.rb

```
class Customer < ActiveRecord::Base
➤  has_many :customers_shipping_address
➤
➤  # Helper to get just the primary shipping address
➤  def primary_shipping_address
➤    self.customers_shipping_address.find_by(primary: true).address
```

```
➤   end
➤   has_one :customers_billing_address
➤   has_one :billing_address, through: :customers_billing_address,
➤                             source: :address
end
```

If you haven't used some of these features of Active Record (namely has_one :through), the Rails Guide on Active Record associations[1] is a great resource to see what you can do with mapping your database tables to models.

With our database tables and models set up, we'll need some data in the tables in order to examine the performance of our database. The code to do this is quite lengthy, so I recommend you copy db/seeds.rb from materialized-views/data-model/shine from the book's webpage.[2] For your reference, the code is also listed in Appendix 2, *Creating Customer Address Seed Data*, on page 205.

Once you have this file in your project, run rails db:seed to populate the data. It will take a long time, but it's worth it. This much data will stress our local database and demonstrate the power of materialized views to fetch data more quickly.

There are two main ways to access this data: you can use the Active Record relationships we've modeled and navigate them in our Ruby code, or you can pull the data back with one big query. Let's examine the performance of each of these approaches.

Performance Using Active Record

We aren't going to use Active Record relationthips to access our data in the end, but this is how it might look to send back a JSON blob of the customer's data and relevant addresses:

```
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
```

1. http://guides.rubyonrails.org/association_basics.html
2. https://pragprog.com/titles/dcbang2/source_code

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.

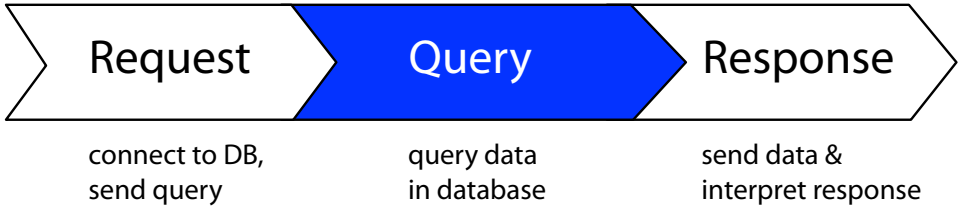
Remember to use rails dbconsole to get access to the Postgres console.

```
sql> 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=1.575..1.576 rows=1 loops=1)
   Index Cond: (id = 2000)
 Planning time: 0.515 ms
 Execution time: 1.590 ms

sql> EXPLAIN ANALYZE SELECT * FROM customers_billing_addresses
                  WHERE customer_id = 2000;
QUERY PLAN
------------------------------------------------------------------------
Index Scan using index_customers_billing_addresses_on_customer_id on
  customers_billing_addresses
  (cost=0.42..8.44 rows=1 width=12)
  (actual time=0.291..0.292 rows=1 loops=1)
   Index Cond: (customer_id = 2000)
 Planning time: 1.744 ms
 Execution time: 0.307 ms

sql> EXPLAIN ANALYZE SELECT * FROM customers_shipping_addresses
                  WHERE customer_id = 2000;
QUERY PLAN
------------------------------------------------------------------------
Index Scan using index_customers_shipping_addresses_on_customer_id on
  customers_shipping_addresses
  (cost=0.42..8.48 rows=3 width=13)
  (actual time=0.341..0.343 rows=4 loops=1)
   Index Cond: (customer_id = 2000)
 Planning time: 1.908 ms
 Execution time: 0.357 ms

sql> 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=0.101..0.102 rows=1 loops=1)
   Index Cond: (id = 2000)
 Planning time: 0.087 ms
 Execution time: 0.121 ms
```

```
sql> EXPLAIN ANALYZE select * FROM states WHERE id = 5;
QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on states
  (cost=0.00..1.64 rows=1 width=16)
  (actual time=0.015..0.018 rows=1 loops=1)
   Filter: (id = 5)
   Rows Removed by Filter: 50
 Planning time: 0.131 ms
 Execution time: 0.034 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 5 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*.



We're running seven queries: that's seven network round-trips. A slow network 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 you 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, you need to itemize the fields you 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, you'll need to join against that table twice (as you'll see). That means you need to know which join you're referencing—the join that brings in the shipping address, or the join that brings in the billing address. To know *that*, you'll be aliasing[3] the ADDRESSES tables in each join to shipping_address and billing_address so you know what you're referring to.

Now that you know the data you're bringing back, you 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
```

3. https://en.wikipedia.org/wiki/Alias_(SQL)

Next, we'll join CUSTOMERS_BILLING_ADDRESSES to ADDRESSES. Note that this is where you 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 you 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
  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 fares.

Query Performance

You can use EXPLAIN ANALYZE on our query to see what sort of performance we might expect.

```
sql> 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,
  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,
  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: 24.776 ms
 Execution time: 1.399 ms
```

The planning time is a bit longer, but if you assume this is reasonably representative, at least as compared to the individual queries, it should take less than around 26 milliseconds. This is still pretty fast, but it's certainly slower as all seven queries combined. But this query incurs only 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 12ms.

To me, this means that if there are *other* benefits to the single-query approach, it won't be at the cost of very much performance and, if you were experiencing performance issues, reducing these seven queries down to one is an avenue worth pursuing. Let's try to use this in Rails and see what it's like.

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` is 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 Active Record's API to produce this query. In these cases, it's easier to just use a string of SQL and execute the query. You 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, since we are constructing SQL without escaping the value of customer_id. You can work around this using Postgres's API directly, which would make the code even more difficult to understand. Second, this code doesn't return a nice object, but instead returns a hash that you 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 who 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, you have to live with code like this, but it's always worth trying to find a better way. Even if the Active Record version of the code executed more queries, and incurred a higher penalty for network round-trips, it looked like idiomatic Rails code.

You'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. You've also seen that a large single query using joins *might* perform better, but results in ugly, hard-to-maintain code.

Now, let's take a look at the third alternative: materialized views.

Using Materialized Views for Better Performance

Materialized views are a special form of *database view* that performs much better than a normal view. If you aren't familiar with views, they are a table-like construct that abstracts away a complex query. For example, you 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
  JOIN states ON states.id = addresses.state_id;
```

Now, we can treat ADDRESSES_WITH_STATES just like a normal table for querying:

```
sql> 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 Active Record:

```
class AddressesWithState < ActiveRecord::Base
end

AddressesWithState.find(12).state
# => WA
```

But a view is just a place to store the query. It would make our Rails code better-looking, but would still not necessarily outperform the seven simpler queries, as we'd still be running the complex join underneath.

If this join were the source of a performance problems, most relational database wouldn't be able to help us solve it. 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 from the secondary cache 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 with a normal view, you can access the materialized view as if it were a regular table using Active Record, meaning

4. http://memcached.org
5. https://www.elastic.co

our Rails code will still be idiomatic. But, because the materialized view isn't doing the expensive query each time, you can fetch the data quickly (though this does come at a cost of increased disk space, because the data for the materialized view is stored on disk).

To create a materialized view, simply use 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/20161007115613_create_customer_details_materialized_view.rb
```

Now, 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, because that's the field we'll be using to query the materialized view (it will also allow us to keep the view up-to-date, as you'll see later).

materialized-views/actual-materialized-view/shi ... 613_create_customer_details_materialized_view.rb

```
class CreateCustomerDetailsMaterializedView < ActiveRecord::Migration[5.0]
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 MATERIALIZED VIEW customer_details"
  end
end
```

Next, run the migration with rails db:migrate. It will take a while, as it's basically running this query for every row of all these tables. When it's done, you'll be able to query this data very quickly.

Let's do an EXPLAIN ANALYZE on our new materialized view:

```
sql> 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=163)
  (actual time=0.011..0.011 rows=1 loops=1)
   Index Cond: (customer_id = 2000)
 Planning time: 0.057 ms
 Execution time: 0.035 ms
```

This is pretty darn good! We're pulling back all of the data we need in under *one tenth* of a millisecond. That is far faster than *both* the canonical Rails Way using Active Record *and* our complex query.

We can now create a CustomerDetail class and query it just as we would any other Active Record object, keeping our code clean and idiomatic, but it will be blazingly fast.

materialized-views/actual-materialized-view/shine/app/models/customer_detail.rb

```
class CustomerDetail < ActiveRecord::Base
  self.primary_key = 'customer_id'
end
```

As our materialized view doesn't have a field named ID, we need to use primary_key= to tell Active Record to use CUSTOMER_ID. With this in place, our controller looks like a regular Rails controller.

materialized-views/actual-materialized-view/shine/app/controllers/customers_controller.rb

```
class CustomersController < ApplicationController
  def show
➤   customer_detail = CustomerDetail.find(params[:id])
    respond_to do |format|
➤     format.json { render json: { customer: 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:

```
sql> 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
```

Now, let's query our materialized view for this new customer:

```
sql> select * from customer_details where customer_id = 388399;
(No rows)
```

Oops. It looks like something's wrong. This is due to how materialized views are implemented by Postgres. 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, you'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.

```
sql> refresh materialized view concurrently customer_details;
sql> 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 you know how to refresh the view, the trick is *when* to do it. This highly depends on how often the underlying data changes and how important it is for you 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. You can do this by creating lib/tasks/refresh_materialized_views.rake and using 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 materialized views"
task refresh_materialized_views: :environment do
  ActiveRecord::Base.connection.execute %{
    REFRESH MATERIALIZED VIEW CONCURRENTLY customer_details
  }
end
```

You can then run it on the command line via rails:

```
$ bundle exec rails refresh_materialized_views
```

With this in place, you can then configure our production system to run this periodically, for example using cron. 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, you could try running it every five minutes. If users can do their jobs without the absolute latest, you could run it every hour or even every day.

If users need it to be absolutely up-to-date with the underlying tables, you 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 Active Record 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/20161007122606_trigger_refresh_customer_details.rb
```

First, we'll create a function to refresh the materialized view. This requires using Postgres's PL/pgSQL[6] language. It looks fairly archaic, but we don't need to use much of it.

materialized-views/actual-materialized-view/shi ... 161007122606_trigger_refresh_customer_details.rb

```
execute %{
  CREATE OR REPLACE FUNCTION
    refresh_customer_details()
    RETURNS TRIGGER LANGUAGE PLPGSQL
  AS $$
  BEGIN
    REFRESH MATERIALIZED VIEW CONCURRENTLY customer_details;
    RETURN NULL;
  EXCEPTION
    WHEN feature_not_supported THEN
      RETURN NULL;
  END $$;
}
```

The key part of this is `RETURNS TRIGGER`, which is what will allow you to use this function in the triggers you'll set up next. Also note the exception-handling clause that starts with `EXCEPTION`. This is similar to Ruby's `rescue` keyword and is a way to handle errors that happen at runtime. You can provide any number of `WHEN` clauses to indicate how to handle a particular exception. In this case, we're handling `feature_not_supported`, which is thrown if we run this function before the materialized view has been updated. In practice this won't happen, but in the our testing environment it can, since we are resetting the database during our tests.

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 reflects what it does: any insert, update, or delete on the `customers` table causes the database to run `refresh_customer_details`. So, we just need to set this up for each table that's relevant.

6. http://www.postgresql.org/docs/9.5/static/plpgsql.html

If we assume that the list of U.S. states doesn't change, we can set up triggers for the other three tables: ADDRESSES, CUSTOMERS_SHIPPING_ADDRESSES, and CUSTOMERS_BILLING_ADDRESSES. As 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/shi ... 161007122606_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 rails db:migrate, you can insert new customers and see the view get refreshed automatically:

```
sql> 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
sql> select id from customers where username = 'amytron';
   id
--------
 350002
sql> insert into customers_shipping_addresses
    (customer_id,address_id,"primary")
  values
    (350002,1,true);
INSERT 0 1
sql> insert into customers_billing_addresses
    (customer_id,address_id)
  values
    (350002,1);
INSERT 0 1
> select * from customer_details where customer_id = 350002;
-[ RECORD 1 ]-------+---------------------------
customer_id         | 350002
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
```

You'll notice that the inserts took a *lot* longer to execute than before. This is the downside of this technique. The materialized view is as up-to-date as it possibly can be; however, it updates very slowly. 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.

One thing you'll notice is that your tests are now failing. Let's fix those before moving on.

Altering Tests to Work with the Materialized View

The issue you should be seeing is that one of the tests in spec/features/customer_search_spec.rb is failing with a message similiar to Couldn't find CustomerDetail with 'customer_id'=11. This is because we've switched the controller's show method to use CustomerDetail, which is backed by our materialized view, and that view requires that a customer have a billing address and shipping address. It's using *inner joins* (the default when you just use JOIN).

An inner join is way to tell Postgres not to return data if there isn't data in a related table. In our case, if there is no row in customers_shipping_addresses or customers_billing_addresses for the given customer, the query backing our view won't return a row.

We could use an *outer join*, which would tell Postgres to return empty values when there isn't related data, but this should only be done if it's allowed by our business rules. In our case, it's not. Every customer must have a billing address and a primary shipping address, so instead of changing our query, we'll change our test data so it's creating valid data.

First, we need to modify the existing create_customer method you created in *Test the Typeahead Search, on page 112* to add the requisite addresses. We'll assume the existence of a method create_address, which you'll see in a moment.

materialized-views/actual-materialized-view/shine/spec/features/customer_search_spec.rb

```
require 'rails_helper'

feature "Customer Search" do

  def create_customer(first_name:,
                      last_name:,
                      email: nil)

    username = "#{Faker::Internet.user_name}#{rand(1000)}"
    email  ||= "#{username}#{rand(1000)}@" +
               "#{Faker::Internet.domain_name}"

    customer = Customer.create!(
      first_name: first_name,
       last_name: last_name,
        username: username,
           email: email
    )
    customer.create_customers_billing_address(address: create_address)
    customer.customers_shipping_address.create!(address: create_address,
                                                primary: true)
    customer

  end
```

If you aren't familiar with some of the methods Active Record creates for you, create_customers_billing_address is provided because we've configured the relationship using a has_one.[7] Similarly, you can call create! on the customers_shipping_address relation because we used has_many.[8]

Now, let's see create_address:

materialized-views/actual-materialized-view/shine/spec/features/customer_search_spec.rb

```
def create_address
  state = State.find_or_create_by!(
    code: Faker::Address.state_abbr,
    name: Faker::Address.state)

  Address.create!(
     street: Faker::Address.street_address,
       city: Faker::Address.city,
      state: state,
    zipcode: Faker::Address.zip)
end
```

7. http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html#method-i-has_one
8. http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html#method-i-has_many

With that in place, our tests should be passing again.

If you choose *not* to use triggers to maintain your materialized view, you will need to execute `REFRESH MATERIALIZED VIEW customer_details` in your tests. You could do this by using RSpec's hooks, which you learned about in *Using DatabaseCleaner to Manage Test Data*, on page 106.

The path that led 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 you had to create a materialized view and triggers to keep it updated, you were able to avoid setting up a new piece of infrastructure for caching and can get more out of the database system we already have in place. Our Rails code looks like regular Rails code (we're using Active Record to query data), and you didn't need to write a background process to keep our data updated. Eventually, you'll have complex enough query needs that you can't use this technique, but you're getting a lot further than you would with other RDBMSs.

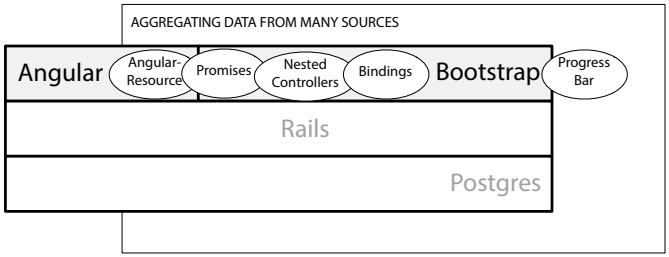
Next: Combining Data with a Second Source in Angular

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.

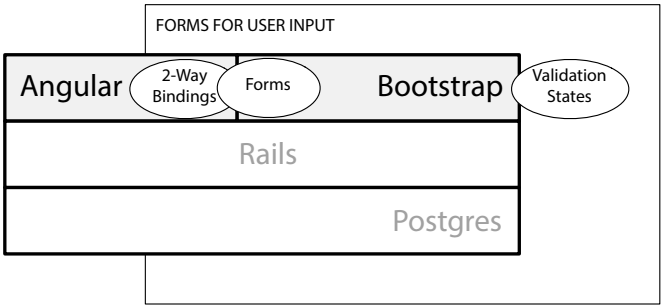
One thing you might have noticed is that our `CustomerDetail` model doesn't expose some of the billing information, such as the last four digits and expiration data of the customer's credit card. We don't store this data in our database, but we need it in our view. In the next chapter you'll see how Angular manages that by learning more about how its asynchronous nature works.

You'll learn how to 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

Asynchronously Load Data from Many Sources



CHAPTER 12

Wrangle Forms and Validations with Angular

CHAPTER 13

Dig Deeper

APPENDIX 1

Full Listing of Customer Detail Page HTML

APPENDIX 2

Creating Customer Address Seed Data

APPENDIX 3

How Webpack Affects Deployment

Bibliography

[Cro08] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly & Associates, Inc., Sebastopol, CA, 2008.

[Man15] Sergi Mansilla. *Reactive Programming with RxJS*. The Pragmatic Bookshelf, Raleigh, NC, 2015.

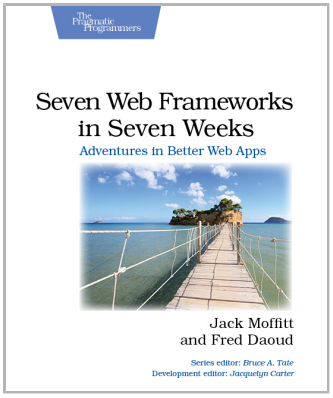
[Rub13] Sam Ruby. *Agile Web Development with Rails 4*. The Pragmatic Bookshelf, Raleigh, NC, 2013.

[Sto12] Jesse Storimer. *Working with Unix Processes*. The Pragmatic Bookshelf, Raleigh, NC, 2012.

Seven in Seven

From Web Frameworks to Concurrency Models, see what the rest of the world is doing with this introduction to seven different approaches.

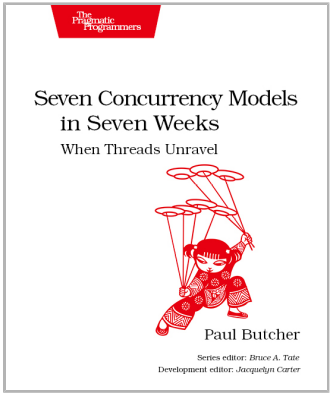
Seven Web Frameworks in Seven Weeks



Whether you need a new tool or just inspiration, *Seven Web Frameworks in Seven Weeks* explores modern options, giving you a taste of each with ideas that will help you create better apps. You'll see frameworks that leverage modern programming languages, employ unique architectures, live client-side instead of server-side, or embrace type systems. You'll see everything from familiar Ruby and JavaScript to the more exotic Erlang, Haskell, and Clojure.

Jack Moffitt, Fred Daoud
(302 pages) ISBN: 9781937785635. $38
https://pragprog.com/book/7web

Seven Concurrency Models in Seven Weeks



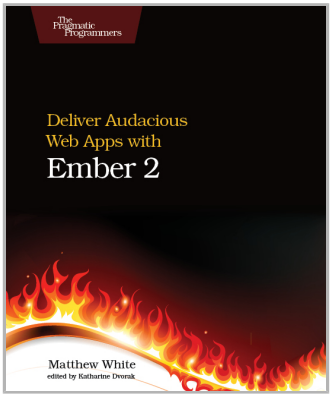
Your software needs to leverage multiple cores, handle thousands of users and terabytes of data, and continue working in the face of both hardware and software failure. Concurrency and parallelism are the keys, and *Seven Concurrency Models in Seven Weeks* equips you for this new world. See how emerging technologies such as actors and functional programming address issues with traditional threads and locks development. Learn how to exploit the parallelism in your computer's GPU and leverage clusters of machines with MapReduce and Stream Processing. And do it all with the confidence that comes from using tools that help you write crystal clear, high-quality code.

Paul Butcher
(296 pages) ISBN: 9781937785659. $38
https://pragprog.com/book/pb7con

The Modern Web

Get up to speed on the latest JavaScript techniques.

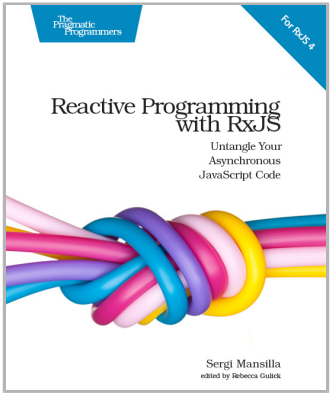
Deliver Audacious Web Apps with Ember 2



It's time for web development to be fun again, time to write engaging and attractive apps – fast – in this brisk tutorial. Build a complete user interface in a few lines of code, create reusable web components, access RESTful services and cache the results for performance, and use JavaScript modules to bring abstraction to your code. Find out how you can get your crucial app infrastructure up and running quickly, so you can spend your time on the stuff great apps are made of: features.

Matthew White
(154 pages) ISBN: 9781680500783. $24
https://pragprog.com/book/mwjsember

Reactive Programming with RxJS



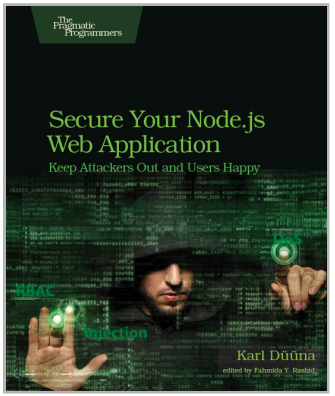
Reactive programming is revolutionary. It makes asynchronous programming clean, intuitive, and robust. Use the RxJS library to write complex programs in a simple way, unifying asynchronous mechanisms such as callbacks and promises into a powerful data type: the Observable. Learn to think about your programs as streams of data that you can transform by expressing *what* should happen, instead of having to painstakingly program *how* it should happen. Manage real-world concurrency and write complex flows of events in your applications with ease.

Sergi Mansilla
(142 pages) ISBN: 9781680501292. $18
https://pragprog.com/book/smreactjs

Secure and Better JavaScript

Secure your Node applications and make writing JavaScript easier and more productive.

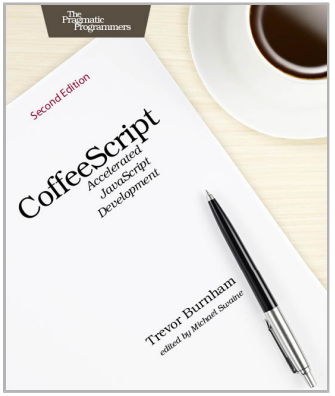
Secure Your Node.js Web Application



Cyber-criminals have your web applications in their crosshairs. They search for and exploit common security mistakes in your web application to steal user data. Learn how you can secure your Node.js applications, database and web server to avoid these security holes. Discover the primary attack vectors against web applications, and implement security best practices and effective countermeasures. Coding securely will make you a stronger web developer and analyst, and you'll protect your users.

Karl Düüna
(230 pages) ISBN: 9781680500851. $36
https://pragprog.com/book/kdnodesec

CoffeeScript



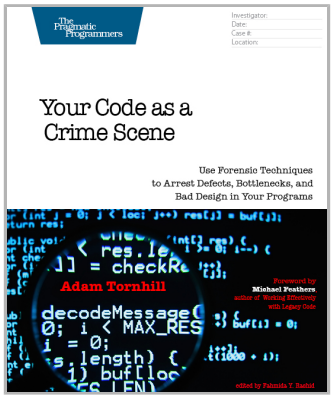
Over the last five years, CoffeeScript has taken the web development world by storm. With the humble motto "It's just JavaScript," CoffeeScript provides all the power of the JavaScript language in a friendly and elegant package. This extensively revised and updated new edition includes an all-new project to demonstrate CoffeeScript in action, both in the browser and on a Node.js server. There's no faster way to learn to write a modern web application.

Trevor Burnham
(124 pages) ISBN: 9781941222263. $29
https://pragprog.com/book/tbcoffee2

Pragmatic Programming

We'll show you how to be more pragmatic and effective, for new code and old.

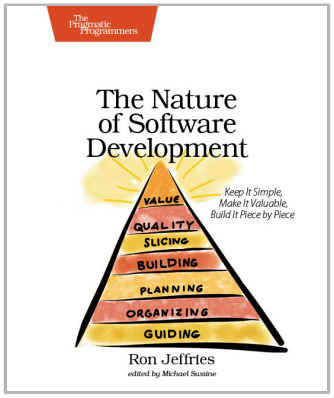
Your Code as a Crime Scene



Jack the Ripper and legacy codebases have more in common than you'd think. Inspired by forensic psychology methods, this book teaches you strategies to predict the future of your codebase, assess refactoring direction, and understand how your team influences the design. With its unique blend of forensic psychology and code analysis, this book arms you with the strategies you need, no matter what programming language you use.

Adam Tornhill
(218 pages) ISBN: 9781680500387. $36
https://pragprog.com/book/atcrime

The Nature of Software Development



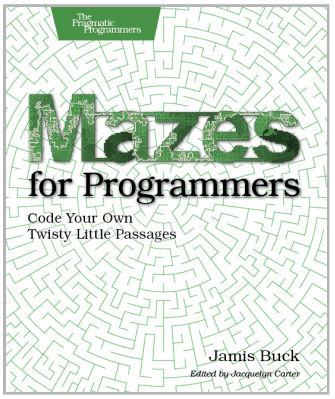
You need to get value from your software project. You need it "free, now, and perfect." We can't get you there, but we can help you get to "cheaper, sooner, and better." This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very beginning.

Ron Jeffries
(178 pages) ISBN: 9781941222379. $24
https://pragprog.com/book/rjnsd

The Joy of Mazes and Math

Rediscover the joy and fascinating weirdness of mazes and pure mathematics.

Mazes for Programmers



A book on mazes? Seriously?

Yes!

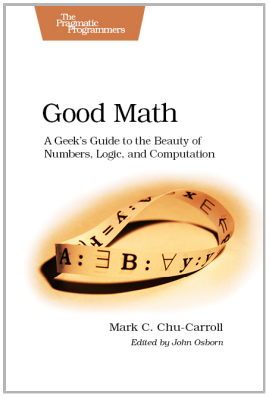
Not because you spend your day creating mazes, or because you particularly like solving mazes.

But because it's fun. Remember when programming used to be fun? This book takes you back to those days when you were starting to program, and you wanted to make your code do things, draw things, and solve puzzles. It's fun because it lets you explore and grow your code, and reminds you how it feels to just think.

Sometimes it feels like you live your life in a maze of twisty little passages, all alike. Now you can code your way out.

Jamis Buck
(286 pages) ISBN: 9781680500554. $38
https://pragprog.com/book/jbmaze

Good Math



Mathematics is beautiful—and it can be fun and exciting as well as practical. *Good Math* is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you've ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll
(282 pages) ISBN: 9781937785338. $34
https://pragprog.com/book/mcmath

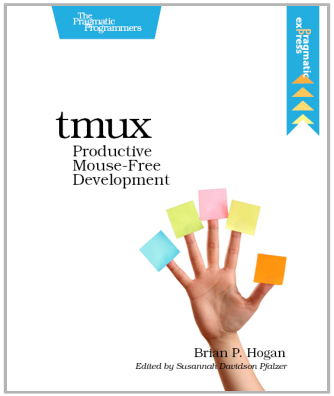
Long Live the Command Line!

Use tmux and Vim for incredible mouse-free productivity.

tmux

Your mouse is slowing you down. The time you spend context switching between your editor and your consoles eats away at your productivity. Take control of your environment with tmux, a terminal multiplexer that you can tailor to your workflow. Learn how to customize, script, and leverage tmux's unique abilities and keep your fingers on your keyboard's home row.

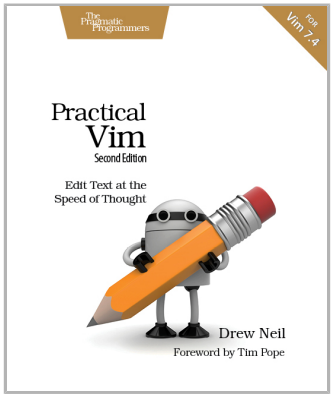
Brian P. Hogan
(88 pages) ISBN: 9781934356968. $16.25
https://pragprog.com/book/bhtmux



Practical Vim, Second Edition

Vim is a fast and efficient text editor that will make you a faster and more efficient developer. It's available on almost every OS, and if you master the techniques in this book, you'll never need another text editor. In more than 120 Vim tips, you'll quickly learn the editor's core functionality and tackle your trickiest editing and writing tasks. This beloved bestseller has been revised and updated to Vim 7.4 and includes three brand-new tips and five fully revised tips.

Drew Neil
(354 pages) ISBN: 9781680501278. $29
https://pragprog.com/book/dnvim2

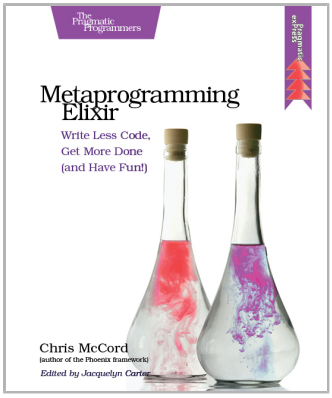


Put the “Fun” in Functional

Elixir puts the “fun” back into functional programming, on top of the robust, battle-tested, industrial-strength environment of Erlang.

Metaprogramming Elixir

Write code that writes code with Elixir macros. Macros make metaprogramming possible and define the language itself. In this book, you’ll learn how to use macros to extend the language with fast, maintainable code and share functionality in ways you never thought possible. You’ll discover how to extend Elixir with your own first-class features, optimize performance, and create domain-specific languages.



Chris McCord
(128 pages) ISBN: 9781680500417. $17
https://pragprog.com/book/cmelixir

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/dcbang2
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/dcbang2*

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