Extracted from:

Rails, Angular, Postgres, and Bootstrap, Second Edition

Powerful, Effective, Efficient, Full-Stack Web Development

This PDF file contains pages extracted from *Rails, Angular, Postgres, and Bootstrap, Second Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

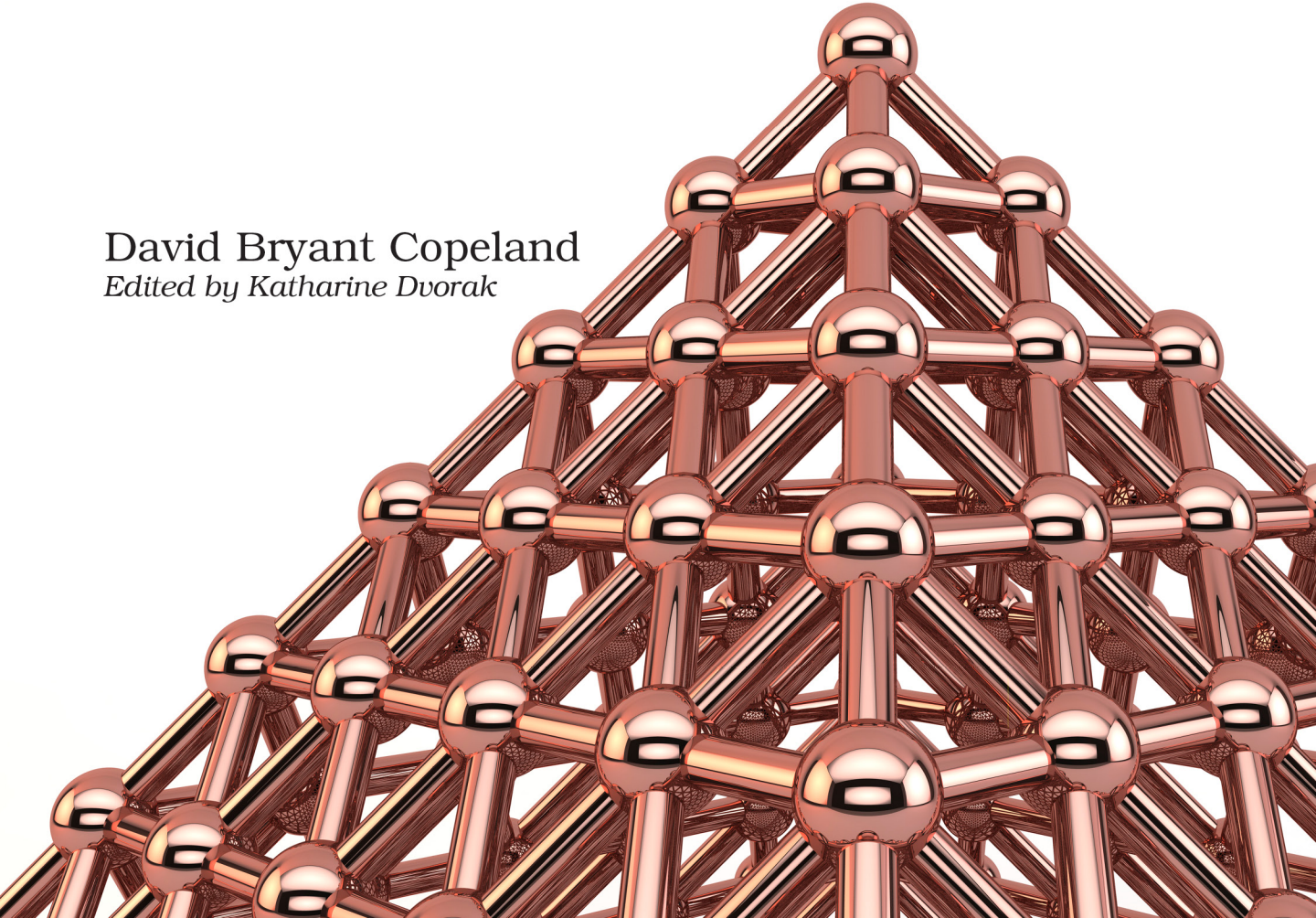
Raleigh, North Carolina

The Pragmatic Programmers

Rails, Angular, Postgres, and Bootstrap

Second Edition

Powerful, Effective, Efficient, Full-Stack Web Development



David Bryant Copeland

Edited by Katharine Dvorak

Rails, Angular, Postgres, and Bootstrap, Second Edition

Powerful, Effective, Efficient, Full-Stack Web Development

David Bryant Copeland

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
Layout: Gilson Graphics

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

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

Copyright © 2017 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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: P1.0—June 2017

Introduction

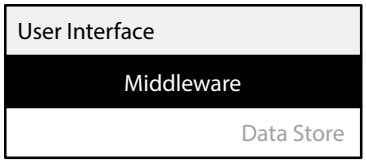
Think about what part of an application you're most comfortable working with. If you're a Rails developer, there's a good chance you prefer the back end, the Ruby code that powers the business logic of your application. What if you felt equally comfortable working with the database, such as tweaking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

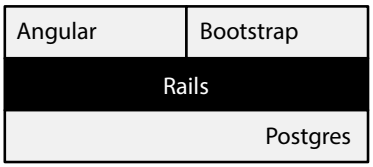
You'll learn *PostgreSQL*, *Angular*, and *Bootstrap*, but you can apply many of the lessons here to other data stores, JavaScript libraries, and CSS frameworks. In addition to seeing just how powerful these specific tools can be, you're going to be emboldened to think about writing software beyond what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

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

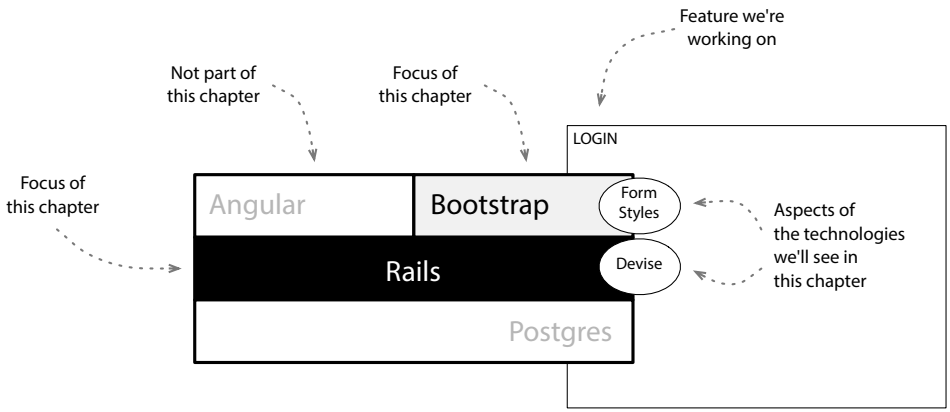
If all you've done with your database is create tables, insert data, and query it, you're going to be excited when you see what else you can do. Similarly, if all you've done with your web views is sprinkle some jQuery calls to your server-rendered HTML, you'll be amazed at what you can do with very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

Angular

Angular[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 the following:

```
var DetailsComponent = 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

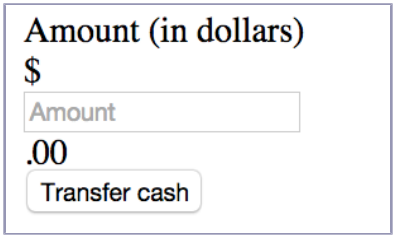
Because of its popularity, there's a large ecosystem of components and modules. Many common problems have a solution in Angular's ecosystem.

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?



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

5. http://getbootstrap.com

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.

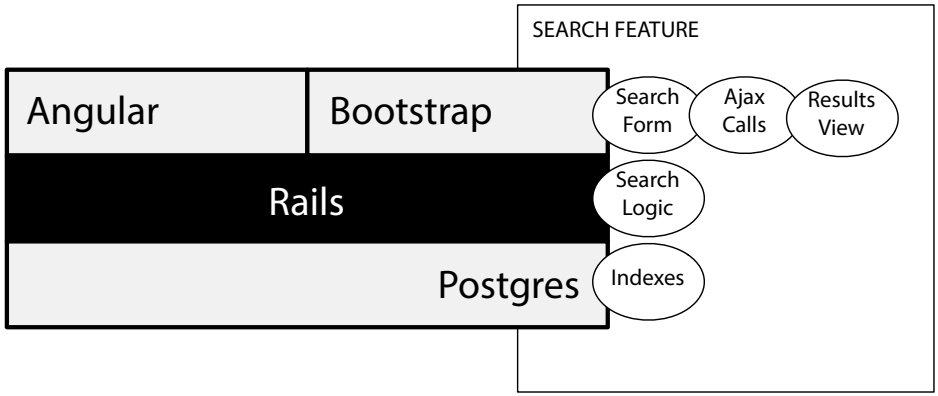
Now that you've gotten a taste of what we'll be covering, let's talk about how you're going to learn it.

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.



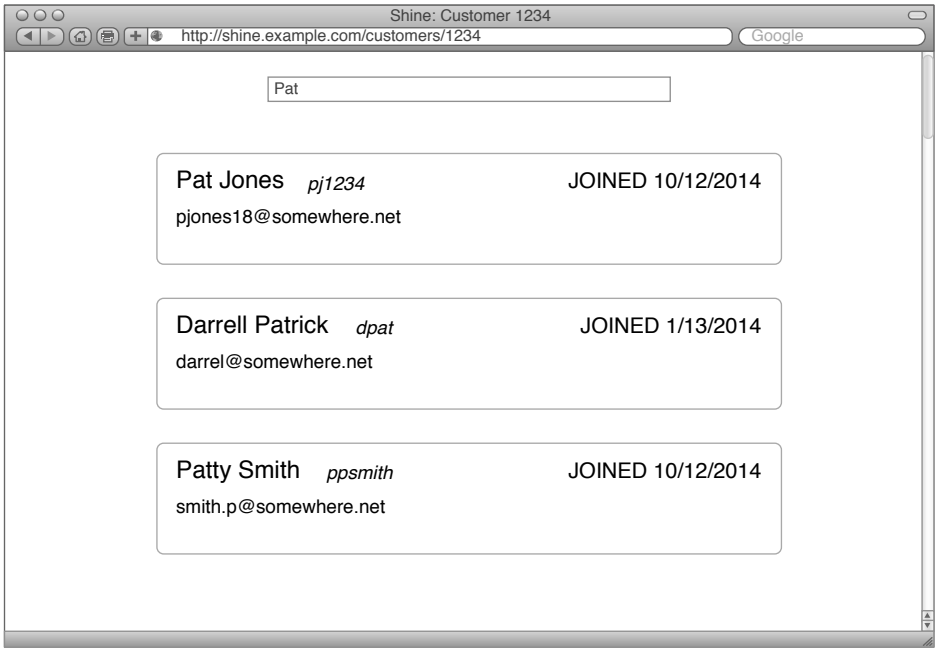
This holistic approach is how you're going to learn these technologies, and you'll have the most success reading the book in order. We'll build a Rails application together, adding features one at a time. These features will demonstrate various aspects of the technologies you're using.

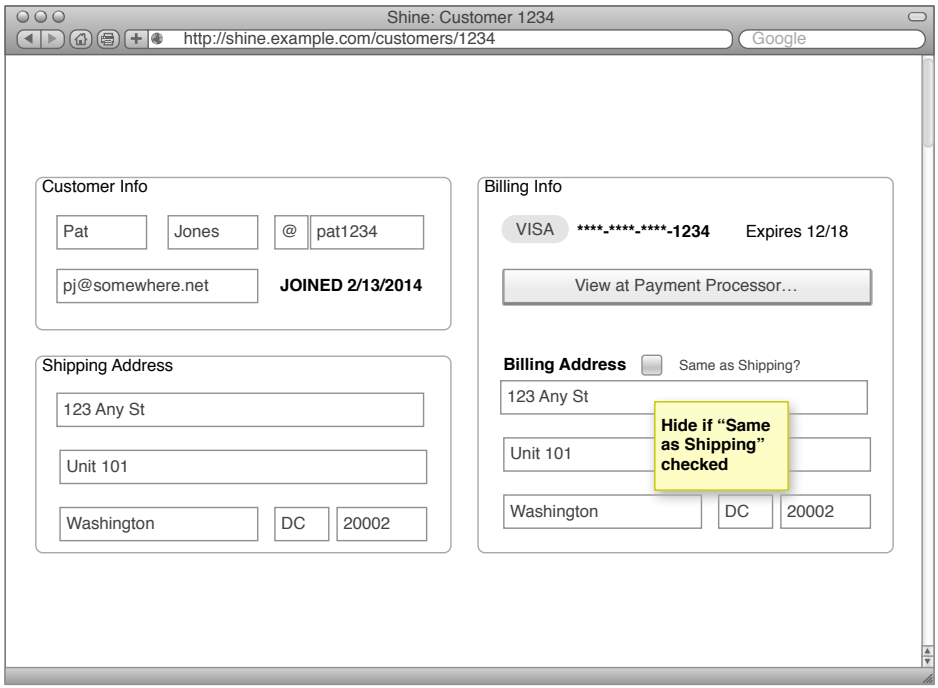
Shine, the Application We'll Build

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 figures on page xiv. Users will also be allowed to click through and view or edit a customer's data.


Shine: Customer 1234
http://shine.example.com/customers/1234
Google
Pat
Pat Jones pj1234
JOINED 10/12/2014
pjones18@somewhere.net
Darrell Patrick dpat
JOINED 1/13/2014
darrel@somewhere.net
Patty Smith ppsmith
JOINED 10/12/2014
smith.p@somewhere.net


Shine: Customer 1234
http://shine.example.com/customers/1234
Google
Customer Info
Pat
Jones
@
pat1234
pj@somewhere.net
JOINED 2/13/2014
Shipping Address
123 Any St
Unit 101
Washington
DC
20002
Billing Info
VISA
****-****-****-1234
Expires 12/18
View at Payment Processor…
Billing Address
Same as Shipping?
123 Any St
Unit 101
Washington
DC
20002
Hide if "Same as Shipping" checked

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 ?. 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'll style the login in Chapter 2, *Create a Great-Looking Login with Bootstrap and Devise*, on page ?, and tighten up the security by learning about *check constraints* in Chapter 3, *Secure the User Database*, on page ?.

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 ?, 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 ?, 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 Angular*, on page ?, is an introduction to Angular, which we use to make our customer search much more dynamic. This chapter explores how to set up and manage Angular code, 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 ? 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 ? 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 ?, 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 ?, 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 ?, 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, by exploring how to effectively use forms with Angular in Chapter 12, *Wrangle Forms and*, on page ?.

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 ?, 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
shine_development=> 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 5.1 [RC17]*.

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 are many online courses that can teach you the basics.

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 2.4.1
- Rails 5.1.1
- Bootstrap 3.3.7
- Angular 4.1.3
- Postgres 9.6
- Webpack 2.6.1
- Node 7.10.0
- Devise 4.3.0
- Karma 1.7.0

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.

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.