Extracted from:

Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development

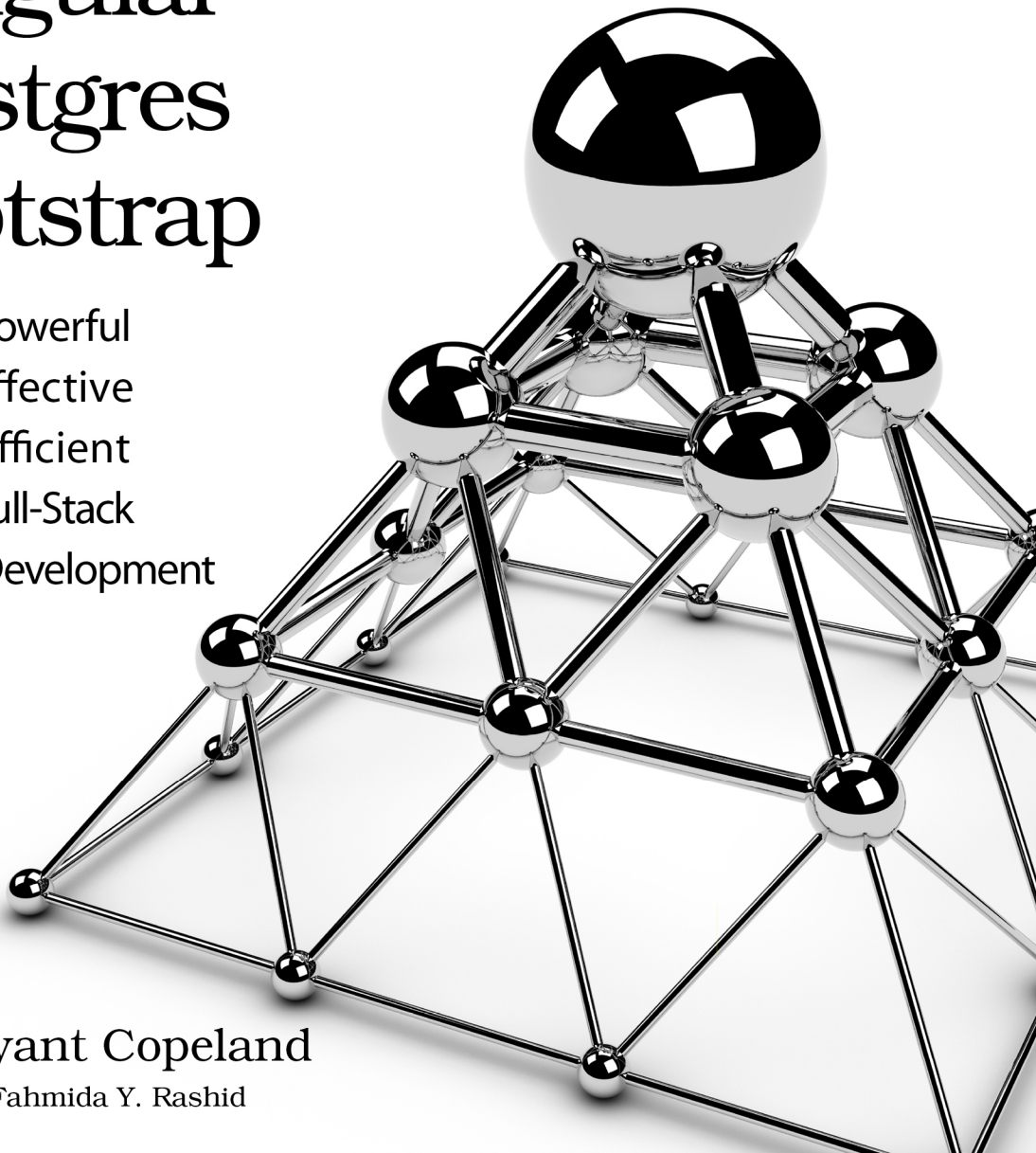
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Rails Angular Postgres Bootstrap

Powerful
Effective
Efficient
Full-Stack
Web Development



David Bryant Copeland
edited by Fahmida Y. Rashid

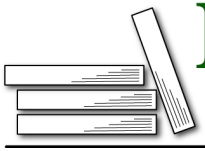
Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development

David Bryant Copeland

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-126-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—July 29, 2015

CHAPTER 1

Introduction

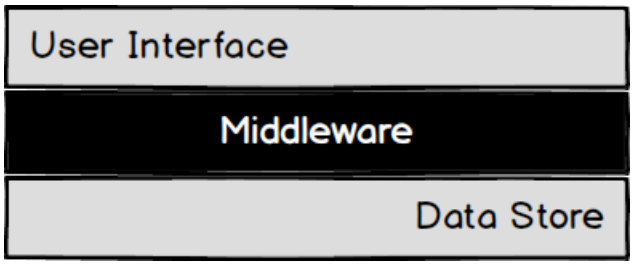
Think about what part of an application you are most comfortable working with. If you are a Rails developer, there's a good chance you prefer the back-end, the Ruby code that powers the business logic of your application. What if you felt equally comfortable working with the database, such as tweaking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

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

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

The Application Stack

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



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

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

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

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

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

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

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

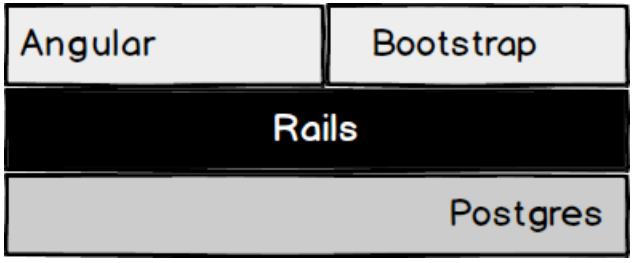
We'll learn *PostgreSQL*, *AngularJS* and *Bootstrap*, but you can apply many of the lessons here to other data stores, JavaScript libraries, and CSS frameworks. Outside of seeing just how powerful these specific tools can be, you're going to be emboldened to think about writing software beyond what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

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

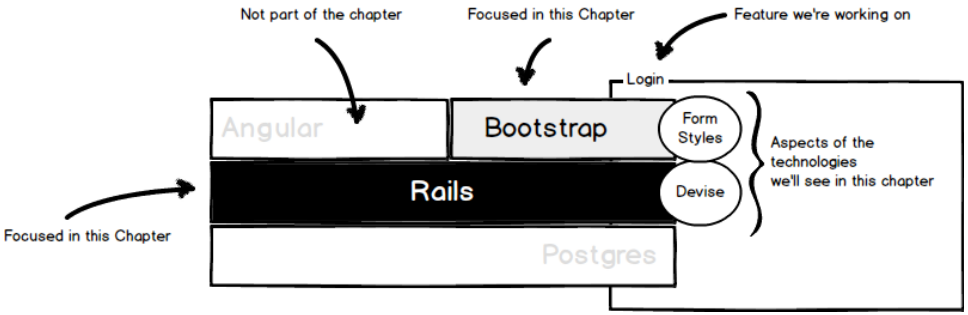
If all you've done with your database is create tables, insert data, and query it, you're going to be excited when you see what else you can do. Similarly, if all you've done with your web views is sprinkle some jQuery calls to your server-rendered HTML, you'll be amazed at what you can do with very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

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

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



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



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

PostgreSQL

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

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

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

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

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

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

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

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

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

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

AngularJS

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

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

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

```
<section>
  <p>You currently owe: $123.45</p>
  <button class="reveal-button">Show Details</button>
  <ul style="display: none" class="details">
```

3. https://angularjs.org

```
    <li>Base fee: $120.00</li>
    <li>Taxes: $3.45</li>
  </ul>
</section>
<script>
  $(".reveal-button").click(function($event) {
    $(".details).toggle();
  });
</script>
```

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

An Angular version of this might look like this:

angular_example.html

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

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

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

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

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

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

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

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

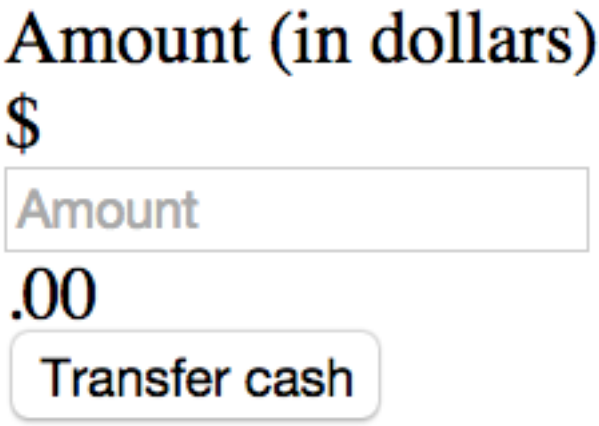
It's actually hard to fully appreciate the power of a JavaScript framework like Angular without using it, but we'll get there. We'll turn a run-of-the-mill search feature into a dynamic, asynchronous live search, with very little code.

Bootstrap

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

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

4. http://getbootstrap.com



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

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

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

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

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

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

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

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

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

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

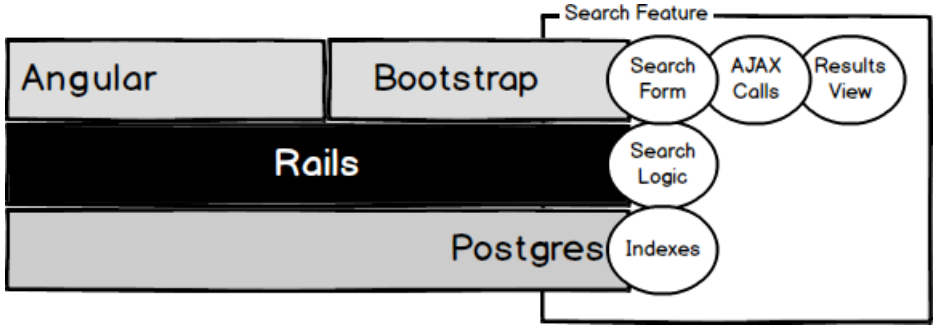
Now that we've gotten a taste of what we'll be learning, let's talk about how we're going to learn it.

Learning Postgres, Angular, and Bootstrap At The Same Time

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

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

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



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

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

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

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

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

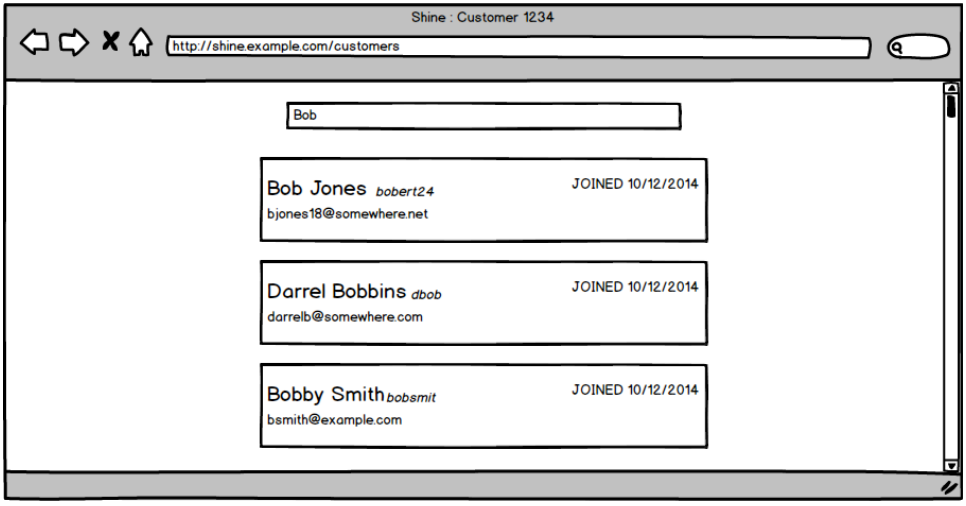
Let's learn about the Rails application that we'll be building throughout the book.

Shine, the Application We'll Build

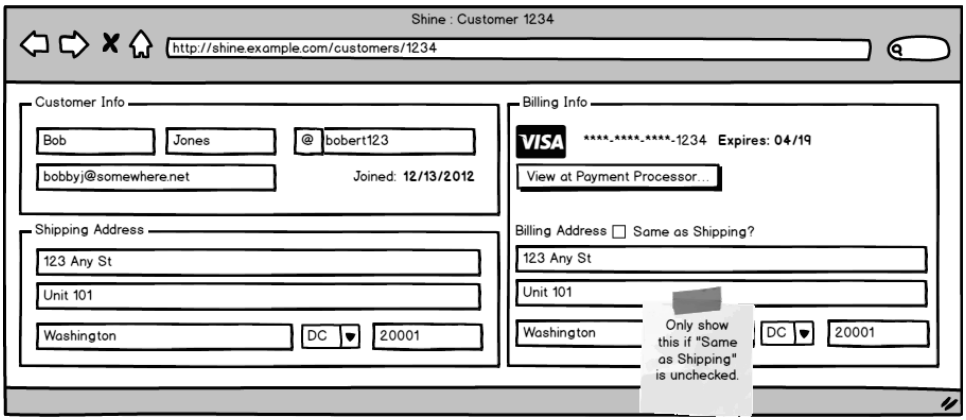
We'll create and add features to a Rails application over the chapters of this book. This application is going to be for the customer service agents at a hypothetical company where we work. Our company has a public website that its customers use, but we want a separate application for the customer service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

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

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



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



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

How Each Chapter Will Work

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

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

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

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

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

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

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

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

All of this is just a small part of what you can do with Bootstrap, Angular, and Postgres, so in Chapter 14, *Grab Bag: More Useful Tidbits*, on page ?, we'll survey some of the other features we don't have space to get to.

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

Getting Set Up

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

Ruby and Rails

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

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

```
> gem install rails
```

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

Postgres

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

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

Everything Else

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

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

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

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

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

Example Code

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

Online Forum and Errata

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

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

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