Beginning Web Development with Node.js

Andrew Patzer

Beginning Web Development with Node.js

Andrew Patzer

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

This version was published on 2013-10-18

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Andrew Patzer by spreading the word about this book on Twitter!

The suggested hashtag for this book is #webdevnode.

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

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

Contents

Views

There are a lot of different ways to present your application to the user and handle the interactions between the user and the backend. At it's simplest, the view layer of a web application presents content to the user through an HTML page, rendered by the browser, with some styling instructions from a CSS file (Cascading Style Sheets). If you'd like to collect data from the user, you could add some form handling code, as well as some input validation logic, either on the server or in the browser. You could also manage data between page views in session objects. All of these functions make up the view layer of a modern web application.

This chapter will focus on building a basic view using a templating language called Jade, organizing your code with a master layout file, and styling your content with a CSS framework called Twitter Bootstrap. Later chapters will cover forms, input validation, and session management.

View Templating

In the early days of the web, the large majority of web pages were written in clunky HTML code, with all kinds of brackets and gnarly layout instructions strewn throughout. Modern web applications tend to use what are referred to as **templating** languages. These are languages that are more developer-friendly, simplifying the task of laying out your user interface. Ultimately, the template file is pre-processed into HTML, as that is what the browser understands. The templating language is simply there for the developer.

As with just about everything Node related, you have many different options for a templating language. As you might recall from the first chapter, when we created our first Express application, the default templating language is Jade. For this tutorial, I chose to stick with Jade, as it is very popular, easy to understand, and quite versatile. As you move on beyond this tutorial, feel free to go check out some of the other templating options and see if there's something you like better.

Jade

Jade[1] was designed specifically for Node, although it may be used with other languages. We'll explore several features of Jade over the remaining chapters, but for now let's cover the basics. Jade is essentially HTML without the need for brackets and closing tags. As such, you'll find that spacing is very important. Nesting of tags is accomplished through indentation. It is important to be consistent in the level of each indentation, and to use either spaces *or* tabs, but not both.

Here is an example that does some simple nesting of tags in HTML, and then the same written as a Jade template.

[1] http://jade-lang.com/

Jade fragment

```
h1 Welcome to Jade
table
  tr
    td Apples
    td $1.50
    td 10
  tr
    td Oranges
    td $1.25
    td 8
p Look ma, no brackets!
.footer The div tag is implicitly assumed, so we can just add a class name with a\
 dot
```

Equivalent HTML code

```
<h1>Welcome to Jade</h1>
<table>
  <tr>
    <td>Apples</td>
    <td>$1.50</td>
    <td>10</td>
  </tr>
  <tr>
    <td>Oranges</td>
    <td>$1.25</td>
    <td>8</td>
  </tr>
<p>Look ma, no brackets!</p>
<div class="footer">The div tag...</div>
```

Jade also can access local variables, and include Javascript code. We'll see examples of this as we go through the next sections. Now, let's see how Jade makes it easy for us to write certain parts of our content, such as headers and navigation code, in one place and reuse it throughout the application.

Layouts

When we create an Express application, it gives us a default layout file in the views folder called layout.jade. Let's take a look at it and see how it works.

views/layout.jade

```
doctype 5
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

This is a simple Jade file, but it includes a couple of things we haven't introduced yet. First, notice the first line under the `head` element. We are adding the `title` tag and assigning to it the value of the local variable named `title`. This is something that we can pass in from our controller, as you'll see in a bit. Next, we create a link tag to reference a CSS stylesheet. The way this is done is by specifying attributes within a set of parentheses after the link tag. Finally, we include a line under our body element that defines a block labeled `content`. This is a placeholder for additional content.

Any Jade template that extends the layout.jade file, can add additional content to the layout by labeling it with the same block label. In this case, the label would be `content`. Let's have a look at the index.jade file that is given to us in the default Express application.

views/index.jade

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

The first line extends the layout file. This will give us all of the header content that is defined in the layout.jade file, specifically the `title` element and the `link` to our stylesheet. Next, the `block content` section is defined. Whatever is nested within this element will be substituted inside of the `block content` section of the layout file. Within the content, you'll notice that it uses the local variable `title` again. Let's see how we can pass a variable into the view through our controller.

Rendering a View

Okay, time to get back to our application and start creating the foundation for our user interface. To start, let's modify the index.jade file slightly to display a message that we pass in from the controller.

views/index.jade

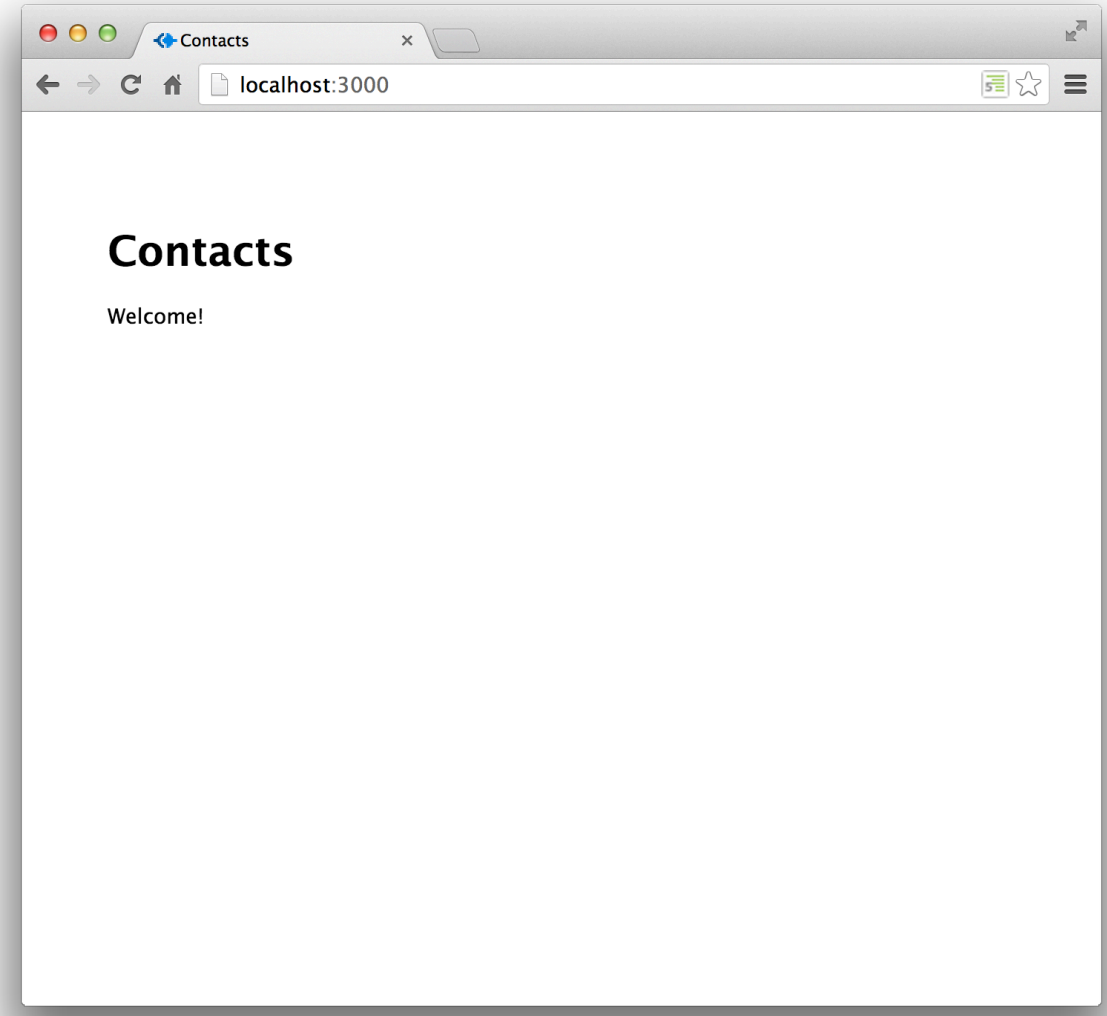
```
extends layout

block content
  h1= title
  p #{message}
```

The only thing different is how we display the 'Welcome!' message. Instead of being a static message, now we can pass anything we want into it via the `message` local variable. The final step to make all of this work, is it modify the `index` function of the controller file to render the index template and pass in the `title` and `message` variables.

controllers/staticController.js

```
exports.index = function(req, res){
  res.render('index', { title: 'Contacts', message: 'Welcome!' });
};
```

Instead of sending content directly to the browser with `res.send()`, we call `res.render()`, passing it the name of the Jade template along with a set of variables to insert into the template. Try running your application and view it in the browser (if you use `foreman` to run it, remember to use port 5000 instead of 3000).



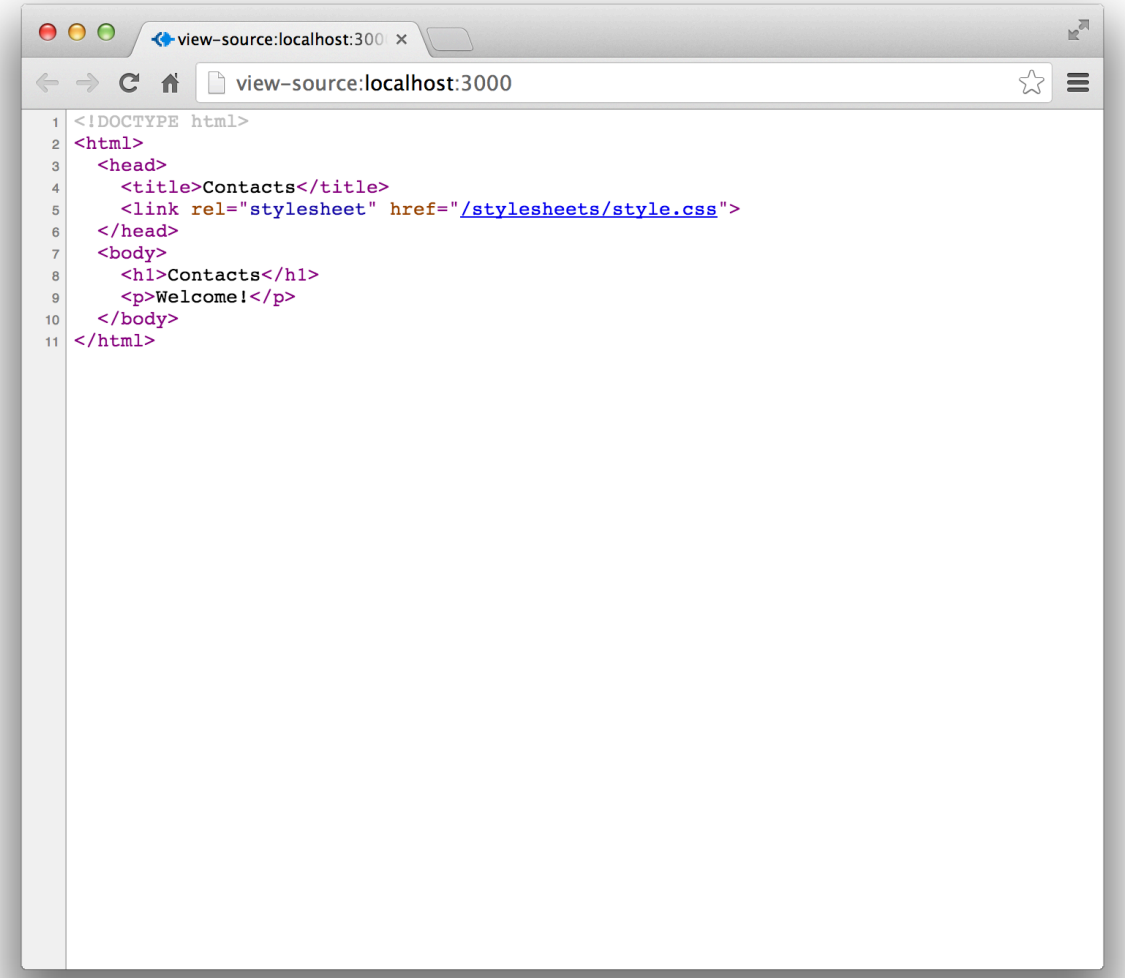
Main View Using Templates

As a debugging tool, especially when we move to a new templating language, it is useful to view the source code that is sent to the browser. If you were to right-click in your browser window and click "view source", you would get the raw HTML that was passed to the browser. It is displayed all in one continuous row though, which makes it difficult to sometimes make sense of what you are looking for. To fix this, go back to your app.js file and add the following line.

```
app.locals.pretty = true;
```

Now, when you right-click to view the source, you'll get a structured display of the HTML that was sent to the browser. This is useful when in development, but might not be needed for production.

The performance hit is minimal though, so I tend to leave it in for production, as it can be useful when investigating user issues.



Source HTML

Now that we've changed the structure of our view layer, let's run our test again to make sure it still passes.

```
$ mocha

  home page
    1) should show welcome: Zombie: Opened window http://localhost:3000/
    ✓ should show welcome
GET / 200 248ms - 162b
Zombie: GET http://localhost:3000/ => 200
Zombie: Loaded document http://localhost:3000/

Zombie: Event loop is empty

  1 passing (317ms)
```

Great, our test still passes! Next, we'll add some style to our pages using the Bootstrap framework.

Using Bootstrap

The Bootstrap framework[2] is a view framework that originated from Twitter and was given to the community. It is a set of CSS and JavaScript files that make it easy for anyone to create a modern looking web application that automatically adapts to a variety of devices. Bootstrap has really taken off this past year, with a large number of websites adopting the framework. There is even a market for custom-themed Bootstrap templates. My favorite is https://wrapbootstrap.com/[3]. For under $20, you can have a professional looking website, or web application, that looks like a designer created it.

To install Bootstrap, you need to download it from their website[4], unzip the file, and copy over the CSS, Javascript, and some font files to the public directory of your application.

```
$ unzip bootstrap-3.0.0.zip
$ cd bootstrap-3.0.0/dist/
$ cp -R fonts/ ~/Code/node/contacts/public/fonts/
$ cp css/bootstrap.min.css ~/Code/node/contacts/public/stylesheets/
$ cp css/bootstrap-theme.min.css ~/Code/node/contacts/public/stylesheets/
$ cp js/bootstrap.min.js ~/Code/node/contacts/public/javascripts/
```

Some of Bootstrap's features require JQuery, so let's go ahead and grab that and put it into our `public/javascripts` folder.

[2] http://getbootstrap.com/
[3] https://wrapbootstrap.com/
[4] http://getbootstrap.com/

```
$ cd ~/Code/node/contacts/public/javascripts/
$ curl -O http://code.jquery.com/jquery-2.0.3.min.js
```

Before modifying our view templates, we need to add one more piece to our view framework. Though not entirely necessary, we're going to add the Stylus module to simplify our CSS code. Stylus let's us get rid of braces and semi-colons. Similar to Jade, Stylus figures things out with proper indentation. Stylus works as a pre-processor, so it will convert your Stylus code into the proper CSS code before delivering it to the browser.

To use Stylus, just install the module through NPM.

```
$ npm install stylus
```

Then, add the following line to your app.js file to let your application know that you would like to add Stylus as a pre-processor for your CSS code. Be sure to add it just below the `app.use(app.router)` line.

```
app.use(require('stylus').middleware('./public'));
```

In your public/stylesheets folder, rename the style.css file to style.styl, remove the existing content, and add the following code to import the bootstrap CSS file.

public/stylesheets/style.styl

```
@import "bootstrap.min.css";
```

You are now ready to use Bootstrap in your templates. To get started, we'll modify the layout.jade file to import the necessary JavaScript files and set up a basic navigation template using the Bootstrap framework.

views/layout.jade

```
doctype 5
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
    script(src='/javascripts/jquery-2.0.3.min.js')
    script(src='/javascripts/bootstrap.min.js')
  body
```

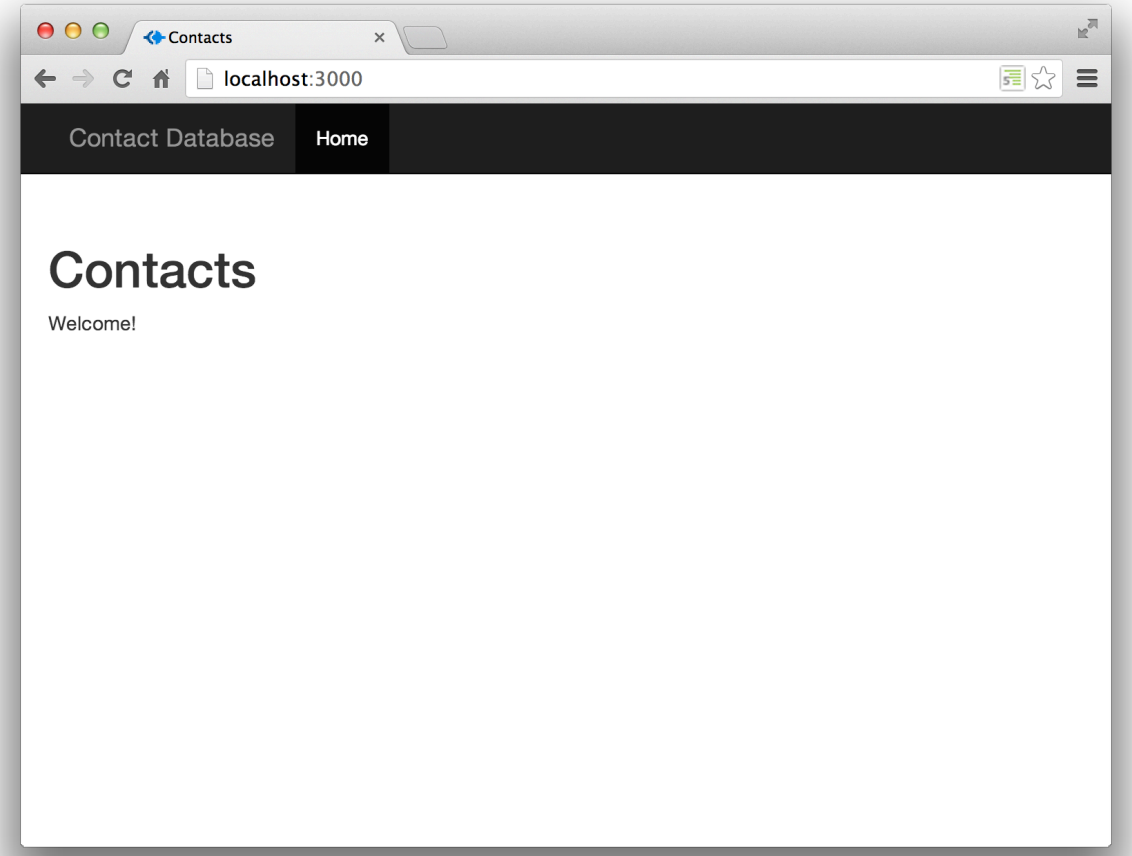
```
    .navbar.navbar-inverse.navbar-fixed-top
      .container
          .navbar-header
            button.navbar-toggle(type='button', data-toggle='collapse', data-targ\
et='.navbar-collapse')
              span.icon-bar
              span.icon-bar
              span.icon-bar
            a.navbar-brand(href='/') Contact Database
          .collapse.navbar-collapse
            ul.nav.navbar-nav
              li.active
                a(href='/') Home
    block content
```

Pay close attention to the nesting of elements here. We start by creating a navigation bar, using the appropriate Bootstrap classes, and then define a header and a collapsible menu with a single element. This creates a basic navigation bar for your application. When you run your application now, you'll see the navigation bar, but it runs over the content now. Let's fix that. Open up your style.styl file and modify it to add some space to account for the new navigation bar.

public/stylesheets/style.styl

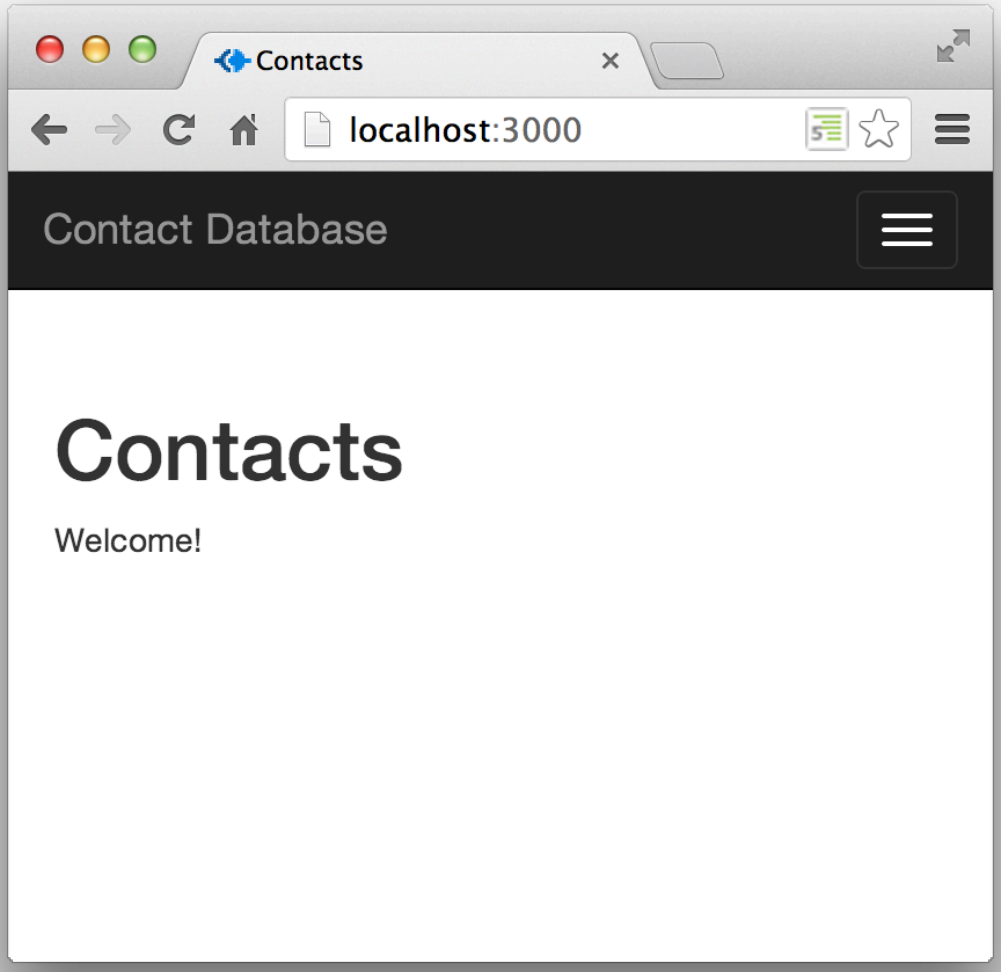
```
@import "bootstrap.min.css";

body
  padding-top: 80px
  padding-left: 20px
  padding-right: 20px
```

Now, things should look better.

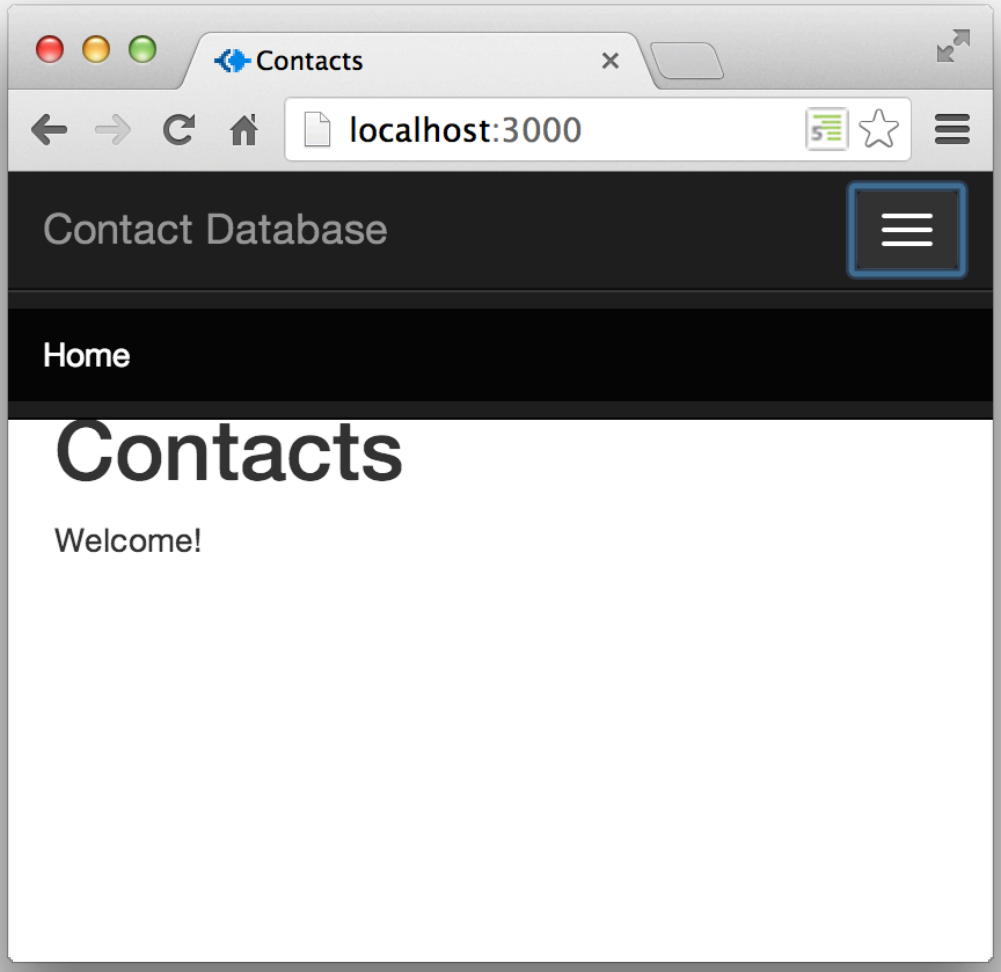


Bootstrap Template

It's a good time to play around with the adaptive layouts as well. Shrink your browser window until the menu goes away and is replaced by an icon in the right corner. This is what it might look like if you were viewing this on a mobile device. If you click the icon, you'll get a dropdown menu.



Bootstrap Template on a Small Screen



Bootstrap Template on a Small Screen with Menu

Fixing our Tests

As usual, we'd like to run our tests again to make sure that our recent changes haven't affected the behavior of our application. So, go ahead and run your test.

```
$ mocha
  home page
    ◻ should show welcome: Zombie: Opened window http://localhost:3000/
    ✓ should show welcome
GET / 200 308ms - 911b
Zombie: GET http://localhost:3000/ => 200
Zombie: Loaded document http://localhost:3000/

  1 passing (386ms)
```

Everything is fine, or is it? Let's do a sanity check and add a `console.log(browser.html());` statement to our test like we discussed in the last chapter. When you run your test again, you'll see that it passes, but does not print out any HTML! What happened? Let's think about what we did since our last successful test. We've added the Bootstrap framework, but our test is just checking for the existence of the same text that was there before. It appears that it should still work. Look at our test code again and we'll see if we can figure out what is going wrong.

```
it('should show welcome', function(done) {
  browser.visit("/", function(e, b, status) {
    assert.ok(browser.success);
    assert.equal(browser.text('p'), 'Welcome!');
    console.log(browser.html());
  });
  done();
});
```

I think the key to our problem is in the `browser.visit()` function. We pass it a callback, only to be executed once the page is loaded. I suspect that something may be causing the browser to never execute the callback. The problem lies in the JavaScript that we are loading now for the Bootstrap framework. To make sure that our test executes immediately following page load, we can add a `wait()` function that tests for the existence of something we know will be on our page. In this case, we test for the existence of a `.container` element, as we know that will be part of the Bootstrap template. Once this is identified, then the callback function is called to perform our tests.

test/simpleTest.js

```
process.env.NODE_ENV = 'test';

var app = require('../app');
var assert = require('assert');
var Browser = require('zombie');

describe('home page', function() {
        var browser, server;

        before(function() {
                server = app.listen(3000);
                 browser = new Browser({site: 'http://localhost:3000', debug: true});
        });

        it('should show welcome', function(done) {

                // Wait until page is loaded
                function pageLoaded(window) {
                          return window.document.querySelector(".container");
                }

                browser.visit("/");
                browser.wait(pageLoaded, function() {
                        assert.ok(browser.success);
                        assert.equal(browser.text('p'), 'Welcome!');
                        console.log(browser.html());
                });
                done();
        });

        after(function(done) {
                server.close(done);
        });
});
```

Now, when you run the test, you'll see that it passes. This time, we're sure that it is actually running the tests, because we can view the source HTML in the console output.

Summary

Now is a good time to commit our code to Git and GitHub once again. We should also push it out to Heroku and verify that it still works in a hosted environment. This is especially important after making several modifications to the CSS and JavaScript resources, as I've found this to be problematic in some situations.

```
$ git add .
$ git commit -m "Added Bootstrap"
$ git push -u origin master
$ git push heroku master
$ heroku open
```

Your browser should now show your web application with the new Bootstrap template being served from Heroku. We tackled quite a bit in this chapter. You now have a framework for building modern web applications that are adaptive to various devices. We'll be extending this in the coming chapters as we add some forms and connect to a database backend.