SAMPLE CHAPTER

Sails.js IN ACTION

Mike McNeil
Irl Nathan

MANNING

Sails.js in Action

by Mike McNeil
Irl Nathan

Chapter 1

brief contents

1 ■ Getting started 1

2 ■ First steps 37

3 ■ Using static assets 55

4 ■ Using the blueprint API 88

5 ■ Custom backend code 112

6 ■ Using models 137

7 ■ Custom actions 171

8 ■ Server-rendered views 214

9 ■ Authentication and sessions 235

10 ■ Policies and access control 261

11 ■ Refactoring 282

12 ■ Embedded data and associations 318

13 ■ Ratings, followers, and search 345

14 ■ Realtime with WebSockets 376

15 ■ Deployment, testing, and security 405

1 Getting started

This chapter covers

- Reviewing modern web development
- Understanding the architecture of the Sails framework
- Positioning Sails in modern web development
- Installing the necessary components of the technical stack
- Setting up the tools of your development environment

Too often, backend programming is put on a pedestal, where only highly trained and disciplined experts are worthy. That's baloney. Backend programming isn't rocket science—but that doesn't mean it's easy. It means that for those new to it, you just need a healthy curiosity and a powerful framework like Sails to get started. If you already have experience with backend programming in a language other than JavaScript, the transition can also be frustrating. Shifting from synchronous to asynchronous patterns can take some time to master. Whether you're new or experienced, Sails will make this transition much easier. Our goal is to provide an entertaining, practical, gap-free path to understanding Sails as well as modern backend web development.

1.1 What is Sails?

Sails is a JavaScript backend framework that makes it easy to build custom, enterprise-grade Node.js apps. It's designed to emulate the familiar MVC pattern of frameworks like Ruby on Rails but with support for the requirements of modern apps: data-driven APIs with a scalable, service-oriented architecture. It's especially good for building chat, realtime dashboards, or multiplayer games, but you can use it for any web application project, top to bottom.

The book is targeted at two types of developers. First is a developer who has frontend experience and is looking to become a full-stack programmer using JavaScript, a language they already know. Second is a developer who has backend experience in a language other than JavaScript and is looking to expand their knowledge to Node.js. In either case, familiarity with HTML, CSS, and JavaScript is expected, as well as experience with making AJAX requests. Most important is a curiosity about how to build a web application.

1.2 What can you build with Sails?

Whether you're a frontend developer seeking to expand your backend knowledge or a server-side developer unfamiliar with using Node and JavaScript on the backend, the common denominator we all share is a desire to create web applications. Sails is designed to be compatible with whatever strategy you have for building your frontend, whether it be Angular, Backbone, iOS/Objective-C, Android/Java, or even a "headless" app that just offers up a raw API to be used by another web service or your developer community. Sails is great for building everyday backend apps that handle HTTP requests and WebSockets. It isn't a good approach for building the client side of your application—that part is completely up to you. If you end up changing your approach (for example, switching from Backbone to Angular) or building a new frontend entirely (for example, building a Windows Phone native app), your Sails app will still work.

WARNING You're about to experience a buzzword bonanza. If you see a term you don't recognize, don't worry—we'll return to these concepts in detail later in the book.

What types of applications can you build? Sails excels at building these:

- *Hybrid web applications*—These applications combine a JSON API with server-rendered views; that is, in addition to an API, this type of application can serve dynamic (that is, personalized) HTML pages, making it suitable for use cases that demand search engine optimization (SEO). These applications often use a client-side JavaScript framework (for example, Angular, Ember, React, and so on), but they don't necessarily have to. Examples of hybrid web applications you might be familiar with are Twitter, GitHub, and Basecamp.

- *Pure APIs*—These applications fulfill requests from one or more independent frontend user interfaces. We say *independent* because the frontend doesn't have to be delivered by the same server that's providing the JSON API—or even by a server at all. This umbrella category includes single-page apps (SPAs), native mobile applications (for example, iOS and Android), native desktop applications (for example, OS X, Windows, Linux), and the venerated Internet of Things (IoT). Many mobile-first products (think Uber, Instagram, Snapchat) start off as pure APIs.

If you aren't sure which category your application falls into, don't worry: the concepts overlap. A pure API is to a hybrid web application as a square is to a rectangle. We'll spend the first half of this book building a pure API, and the remaining chapters extending and maintaining it as it transitions into a hybrid web application.

1.3 Why Sails?

Sails' ensemble of small modules works together to provide simplicity, maintainability, and structural conventions to Node.js apps. Sails is highly configurable, so you won't be forced into keeping functionality you don't need. But at the same time, it provides a lot of powerful features by default, so you can start developing your app without having to think about configuration. Here are some of the things Sails does right out of the box:

- *100% JavaScript*—Like other MVC frameworks, Sails is built with an emphasis on developer happiness and a convention-over-configuration philosophy. But Node.js takes this principle to the next level. Building on top of Sails means your app is written entirely in JavaScript, the language you and your team are already using in the browser. Because you spend less time shifting context, you're able to write code in a more consistent style, which makes development more productive and fun.

NOTE Both authors of this book can attest to how nice it is to work with *one language* instead of constantly switching back and forth between JavaScript and whatever backend language our company or customers are using. The best part? It means you get *really good* at it.

- *Rock-solid foundation*—Sails is built on Node.js, a popular, lightweight, server-side technology that allows developers to write blazing-fast, scalable network applications in JavaScript. It also uses Express for handling HTTP requests and Socket.IO for managing WebSockets. So if your app ever needs to get really low level, you can access the raw Express or Socket.IO objects. And there's another nice side effect: if you already have an Express app, your existing routes will work perfectly well in a Sails app, so migrating is a breeze.
- *Frontend agnostic*—Although the promise of "one language and/or framework to rule them all" is certainly enticing, it isn't always realistic. Different organizations, technologies, and personalities all have their preferred way of doing things.

It's because of this that Mike McNeil made Sails compatible with *any* frontend strategy, whether it's Angular, Backbone, iOS/Objective-C, Android/Java, Windows Phone, or something else that hasn't been invented yet. Plus, it's easy to serve up the same API to be consumed by another web service or community of developers.

- *Autogenerated REST APIs*—Sails comes with "blueprints" that help jumpstart your app's backend without writing any code. Just run `sails generate api dentist` and you'll get an API that lets you search, paginate, sort, filter, create, destroy, update, and associate dentists. Because these blueprint actions are built on the same underlying technology as Sails, they also work with WebSockets and any supported database out of the box.
- *Use any popular database*—Sails bundles a powerful object-relational mapping (ORM) tool, Waterline, which provides a simple data access layer that just works, no matter what database you're using. In addition to a plethora of community projects, officially supported adapters exist for MySQL, MongoDB, PostgreSQL, Redis, and local disk storage.
- *Powerful associations*—Sails offers a new take on the familiar relational model, aimed at making data modeling more practical. You can do all the same things you might be used to doing in an ORM (one-to-many, many-to-many), but you can also assign multiple *named* associations per model. For instance, a cake might have two collections of people: "havers" and "eaters." Better still, you can assign different models to different databases, and your associations/joins will still work—even across NoSQL and relational boundaries. Sails has no problem implicitly or automatically joining a MySQL table with a Mongo collection and vice versa.
- *Standardization*—When you build a Sails app, you're taking advantage of all sorts of open standards behind the scenes. Almost everything has a specification, from database and file upload adapters to hooks that make up the framework itself. Using the machine specification, you can even make *any function* in your app pluggable, making it easy to switch between different providers for services like email delivery and social authentication. Building on top of well-defined interfaces means that whenever you need to do something custom, your work is self-documenting, quick to implement, and simple to debug.
- *Node machine services*—The Machine Specification is an open standard for JavaScript functions. Each machine has a single, clear purpose—whether it be sending an email, translating a text file, or fetching a web page. Machines are self-documenting, quick to implement, and simple to debug.
- *Realtime with WebSockets*—Sails translates incoming socket messages for you, making them compatible with every route in your Sails app.
- *Reusable security policies*—Sails provides basic security and role-based access control by default.

- *Sails generators*—Sails provides a consistent way of creating projects using reasonable defaults. Sails also contains generators for automating many tasks like creating models and controllers. Generators are built on an extensible architecture, supported by a community of developers.
- *Flexible asset pipeline*—Sails ships with opinionated build scripts and a default directory structure for client-side assets. Out of the box, the asset pipeline provides support for LESS, CoffeeScript, precompiled client-side HTML templates, and production minification. This makes setting up new projects easy and consistent, but it does pose a problem when it comes time to tweak or completely redefine that tooling to fit your personal preferences or your organization's best practices. Fortunately, all the default automation in Sails is implemented as plugins for the Grunt task runner, which means your entire frontend asset workflow is completely customizable. It also means you can choose from the thousands of widely used, open source Grunt plugins already out there.

If you don't understand some of these bullet points, don't worry. Our goal isn't to teach you a bunch of jargon and acronyms. But by the end of the book, you'll have a firm conceptual grasp of each of these topics—and, more important, you'll be able to apply that understanding when building the backend for any of the different types of apps listed.

1.4 Fundamental concepts of a web application

Web application development is riddled with core concepts and terminology that may or may not be familiar to you. It's critical that we have a common frame of reference for them before we begin this extended journey together. This section is a jump-start to your understanding of an important core concept in backend development: the HTTP request/response protocol. If this seems like a review, feel free to skip to section 1.5, "Understanding databases."

1.4.1 Requests and responses

The heart of a web application is handling the conversations made through requests sent by the frontend and responses sent by the backend. We'll ease into this discussion using a tool we're all familiar with: the browser. To start with, let's take a look at the completed version of Brushfire, the application we'll build together throughout the rest of this book. Navigate your browser to https://brushfire.io, as shown in figure 1.1.

The browser just made a *request* on your behalf and the Sails server *responded* with the contents of the home page you now see displayed, as shown in figure 1.2.

When you're talking with a waiter, you might use a *protocol* such as English or Spanish to make a request ("Could I have a glass of water?") and receive a response ("Certainly!"). The same kind of conversation exists between a frontend and your Sails application, but because computers don't have the kinds of mouths, ears, or brains fit

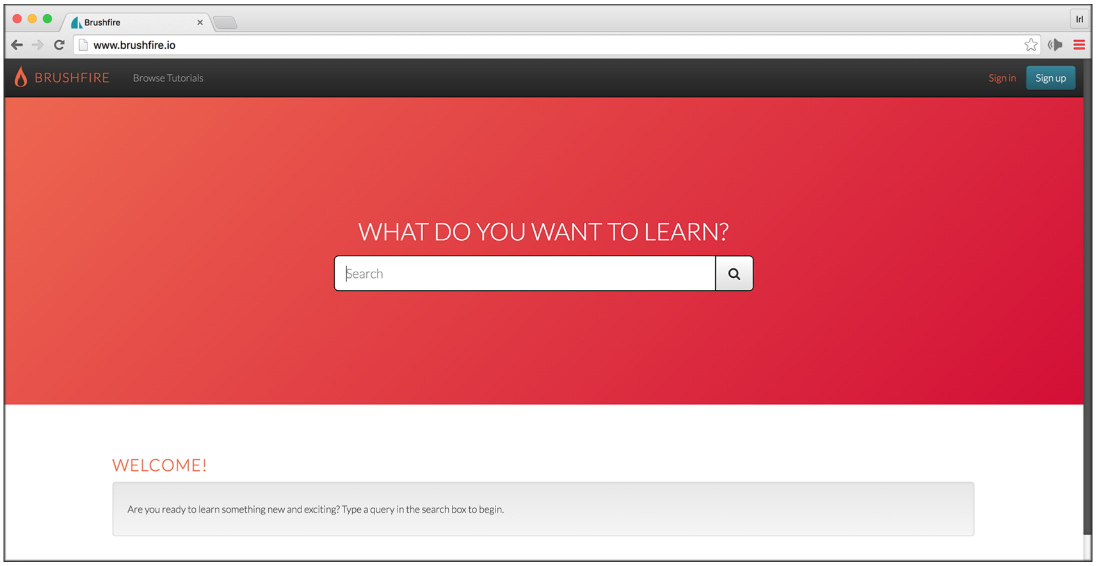

Brushfire
www.brushfire.io
BRUSHFIRE
Browse Tutorials
Sign in
Sign up
WHAT DO YOU WANT TO LEARN?
Search
WELCOME!
Are you ready to learn something new and exciting? Type a query in the search box to begin.

Figure 1.1 The contents of a response from an initial request by your browser to https://brushfire.io

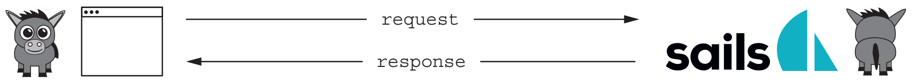


Figure 1.2 The frontend makes a request and the backend makes a response.

for processing human language, the client and server communicate by sending specially formatted strings back and forth. This conversation is called the *Hypertext Transfer Protocol (HTTP)*.

> **TIP** "Wait, no one said anything about learning protocols!" HTTP is just an agreed-upon set of rules not unlike a rudimentary language. And it is this language that enables different devices that know how to speak HTTP to talk to each other. For the adventurous who want a low-level explanation, check out the Request For Comments (RFC) pages for HTTP found here, http://tools.ietf.org/html/rfc7230#page-5, which are surprisingly readable.

Requests and responses sent back and forth using the protocol comprise the underlying communication bridge between our frontend client and backend Sails server.

Let's take a closer look at the actual request and the response. Click the Sign Up button in the upper navigation bar of the homepage, and you should see the signup page, as illustrated in figure 1.3.

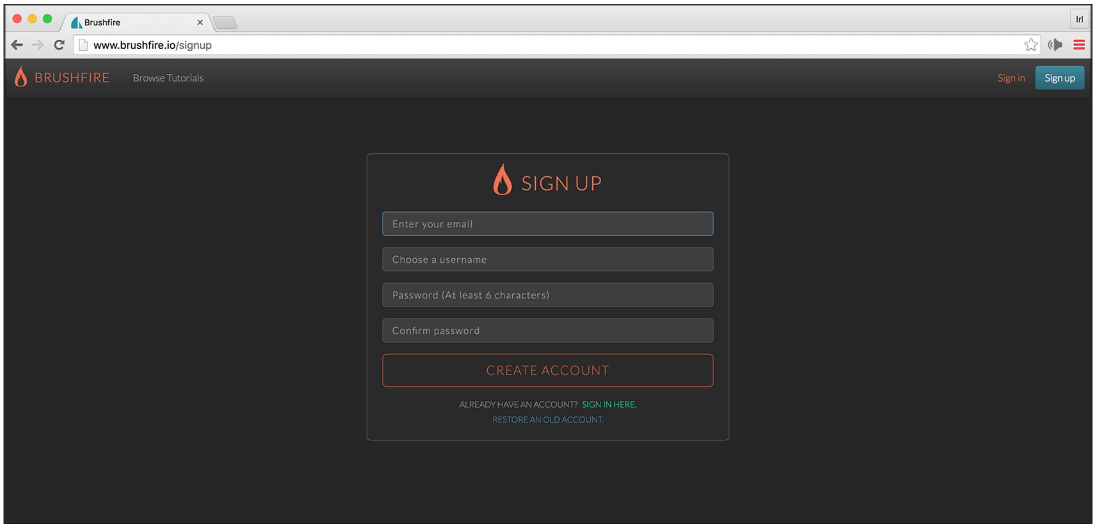


Figure 1.3 Clicking the Sign Up button generates a request to the Sails backend, which responds with the signup page.

Once again, the browser makes an HTTP request on your behalf and the Sails server responds, in this case with a string of HTML markup representing the signup page. Figure 1.4 displays an overview of the steps that culminate in the rendered signup page.

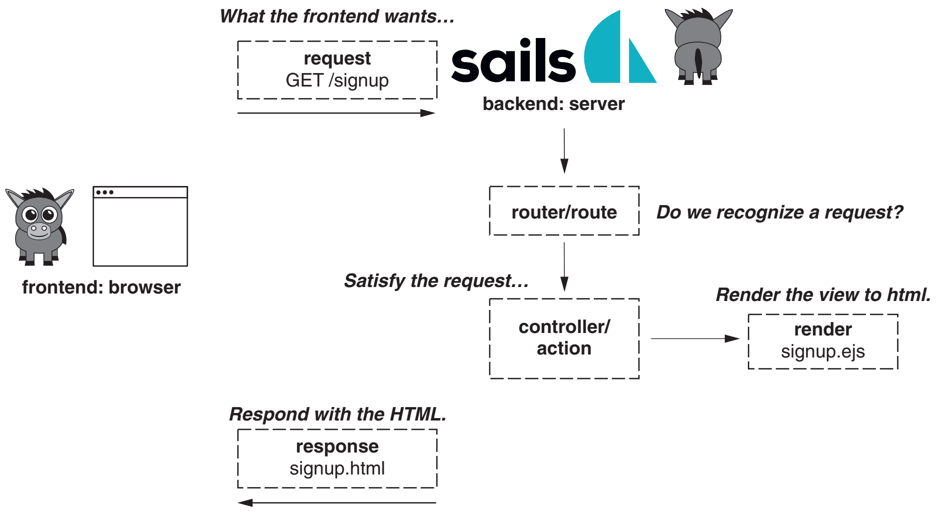


Figure 1.4 The components necessary for the backend to satisfy the request and deliver the signup page to the frontend

At a high level, figure 1.4 shows an often-repeated pattern of Sails components you'll use to build the backend. We'll focus on the details of each of these components in chapters 3 through 15. For now, let's concentrate on the request. A portion of the raw request string that was sent from your browser to Sails looks like this:

```
GET /signup HTTP/1.1
```

This is technically called the *request line* or *start line*, but what matters is that it consists of the *method* (`GET`) and the *path* (`/signup`).

> **NOTE** For our purposes, the protocol version (HTTP/1.1) can be ignored—we're interested in just the method and the path. The request contains two other things we care about: *request headers* and a *request body*. We'll discuss them a little later in the book.

Next, let's move over to the response. The Sails server received the `GET` request to `/signup` and determined that the intent of the request was to receive a response containing the signup page. The first piece of the raw response string sent from the Sails server to your browser looks like this:

```
HTTP/1.1 200 OK
```

This portion of the response message is called the *response line* or *start line* and consists of the *protocol version*, `HTTP/1.1`, the *server status code*, `200`, and something called the *textual reason phrase*, `OK`.

> **NOTE** Naming stuff is probably the hardest thing to do in programming. It's so hard that we get names like *textual reason phrase*.

The important part is the server status code (`200`), a special number that indicates the status or outcome of the request, like how the code exited. In addition to the status code, the response also contains the HTML of the startup page in a part of the response called the *response body*.

> **NOTE** The complete response message also contained *response headers*, which aren't part of our example, so we'll postpone discussing them.

Now for some good news! Because requests originate in a limited number of ways, you'll rarely have to work with a raw request or a raw response. Instead, outgoing requests will be generated by one of the approaches in table 1.1.

Table 1.1 Sources of HTTP requests

Approach	Example
A browser URL bar	http://www.myApp.com/signup
An anchor tag	`<a href="http://www.myApp.com/signup"></a>`

Table 1.1 Sources of HTTP requests *(continued)*

Approach	Example
The browser's location property on the `window` dictionary	`window.location = "http://www.myApp.com/signup"`
The browser `window` dictionary `open` method	`window.open("http://www.myApp.com/signup")`

An AJAX request	
	```
	$.ajax({
	  url: '/signup',
	  type: 'GET',
	  success: function(result){
	    console.log('result: ', result);
	  },
	  error: function(xhr, status, err){
	    console.log(err);
	  }
	});
	```

Via an HTTP library (Android example)	
	```
	// Instantiate the RequestQueue.
	RequestQueue queue = Volley.newRequestQueue(this);
	String url ="http://www.google.com";

	// Request a string response from the provided URL.
	StringRequest stringRequest = new
	StringRequest(Request.Method.GET, url,
	           new Response.Listener<String>() {
	    @Override
	    public void onResponse(String response) {
	        // Display the first 500 characters of the
	response string.
	        mTextView.setText("Response is: "+
	response.substring(0,500));
	    }
	}, new Response.ErrorListener() {
	    @Override
	    public void onErrorResponse(VolleyError error) {
	        mTextView.setText("That didn't work!");
	    }
	});
	// Add the request to the RequestQueue.
	queue.add(stringRequest);
	```

Incoming raw requests to the backend are parsed and transformed by Sails into dictionaries with properties you can easily access in your backend code.

DEFINITION What—JavaScript has dictionaries? Because the word *object* is used ubiquitously in JavaScript to describe almost everything, we use the term *dictionary* to refer to an *object* that's declared using {} curly braces. For example, { `foo: 'bar'` } is a dictionary.

For outgoing responses, you'll rely on Sails' built-in methods for responding to a request with JSON or a dynamic HTML web page. This allows you to focus on how your application is supposed to work, instead of the detailed minutiae of HTTP.

> **NOTE** For 99% of use cases, this level of abstraction is more than flexible enough. But if you ever need lower-level access, don't worry. Sails and Node.js allow you to work directly with the underlying HTTP request and response streams on a case-by-case basis.

Now that you know a bit more about the request and response, let's explore how they're used to successfully fulfill the requirements of our application.

1.4.2 HTTP methods

In section 1.4.1, we introduced HTTP as a way for the frontend and backend to send data back and forth. But requests are useful for more than just transporting data: they also convey intent. The Sails server must interpret that intent and respond with something that fulfills the requirements of the initial request. Let's examine how this communication is accomplished using the signup page as a real-world example. In your browser, fill out the Brushfire signup form you navigated to earlier, and click Create Account. This triggers an AJAX request from the browser. An overview of this request/response can be found in figure 1.5.

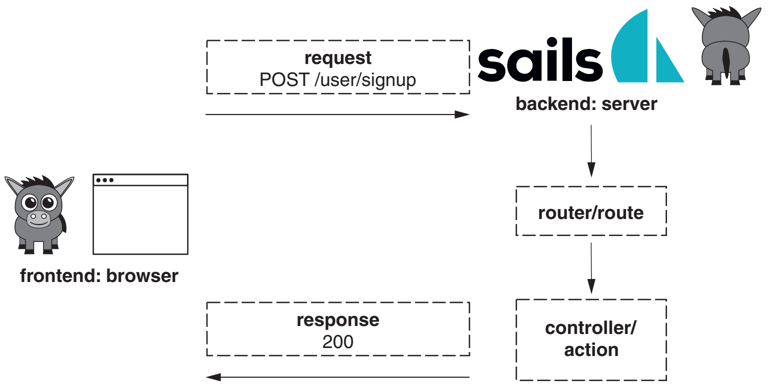


Figure 1.5 The components necessary for the backend to satisfy the request by creating a user and responding with the result

Once again, don't get overwhelmed by the details; we'll reinforce each component many times with examples. If you looked at the raw request string, it would look like this:

```
POST /user/signup HTTP/1.1
```

Because the request included the method, `POST`, and the path, `/user/signup`, in this example, you'd say that the browser sent a `POST` request to `/user/signup`. HTTP methods (also known as *verbs*) like POST, GET, PUT, and DELETE are simply labels that help indicate the intent of a request.

NOTE The request on the signup page is intended to create a new user. Shortly, you'll see that the frontend and backend have made an agreement that when the frontend makes a POST request to /user/signup, the backend Sails server will create the user. The request is the frontend's portion of that agreement.

The term *HTTP method* can be confusing because it gives the impression that each method has some inherent, specific purpose. The reality is that the method label, POST, doesn't inherently do anything. It can express your intent, but it's up to the backend to determine how to interpret that intent and respond.

For example, you could create a Sails app that interprets a GET request to the path /signup as a request to create a new user. Technically, this would work just fine, but it would be a bad idea, because it would violate a common convention.

DEFINITION We use the term *convention* to mean an informal agreement between programmers for how something is supposed to work. It's usually a bad idea to break conventions. Not only do they make it easier for developers to collaborate and get up to speed on a code base, but they also make it easier for you to remember how your app works as it matures.

The GET method, by convention, is used to indicate that an action is *cacheable* or *safe*, because nothing should change as a result of making a GET request. If your backend interpreted a GET request to the path /signup as a request to create a new user, adding the user would violate this convention. The conventional side effects of each HTTP method are listed in table 1.2.

Table 1.2 HTTP method conventions

Method	Side effects
GET	Should be *cacheable*; that is, sending a GET request shouldn't cause any side effects. Often used for fetching data.
POST	No guarantees. Any given POST request could cause side effects such as sending an email or creating a pet store in the database.
PUT	Should be *idempotent;* that is, sending the same PUT request over and over has the same side effects as sending it only once. Often associated with updating a resource.
DELETE	Should be idempotent (see the previous entry). Often associated with deleting a resource.

But we want to stress again that these are what the methods *should* do according to convention. It will be up to you to implement them in this way on the backend.

The other part of the request is the URL *path*, which looks like a file system path on your computer's local hard disk. Although a path can be anything, more often the path is simply a reference to a *resource* and *action*. For example, the path `/user/signup` consists of a *user* resource and a *signup* action.

TIP You can think of a *resource* as a label that groups related tasks together and an *action* as one of those tasks.

By combining the `POST` method with the path `/user/signup`, you convey the request's intent—to sign up or create a new user.

Next, let's move to the response. The Sails server received the `POST` request to `/user/signup`, interpreted its intent, and as you'll see later, created the user account before responding to the browser like this:

```
HTTP/1.1 200 OK
```

Here, the only part of the response line you're interested in is the status code.

DEFINITION The conventional meaning of a *status code* is even more ingrained than the conventional meaning behind HTTP method labels. You'll use status codes in your responses as a shorthand way to convey the status of a request. In this case, the Sails server responded with a status `200`, signaling that the creation of the user account was successful. For an exhaustive list of conventional status codes, see https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

In our previous example, you learned how the request can convey intent to create a user. But what interprets that intent on the backend? The short answer is that Sails matches the incoming request with a route using its built-in router. We'll explore routes and the router in the next section.

1.4.3 Routing

It's easier to examine how Sails interprets the intent of an incoming request using figure 1.6.

Recall that in the earlier example you made an AJAX `POST` request to `/user/signup`. Sails "heard" the request via a built-in module called the *router* ❶. The router recognized

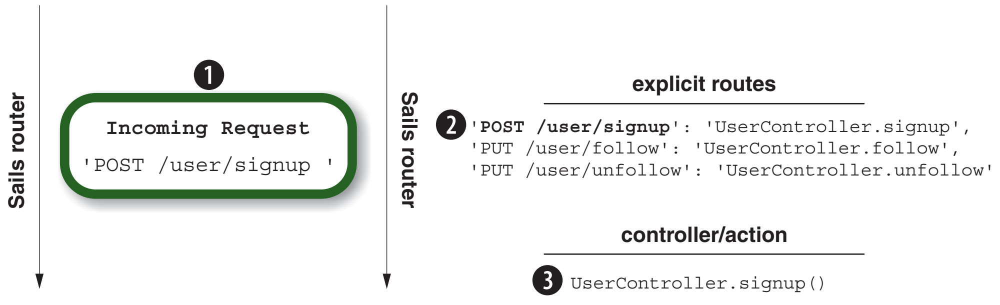

Sails router
1
Incoming Request
'POST /user/signup '
Sails router
explicit routes
2
'POST /user/signup': 'UserController.signup',
'PUT /user/follow': 'UserController.follow',
'PUT /user/unfollow': 'UserController.unfollow'
controller/action
3
UserController.signup()

Figure 1.6 Understanding how the Sails router matches the incoming request with routes to trigger an action

the particular request because the request matched ❷ the configured route address of an *explicit route*, and then that triggered an action ❸, as illustrated in figure 1.7.

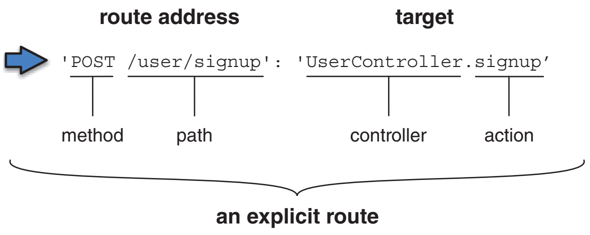


Figure 1.7 The router matched the request as part of an explicit route.

In Sails, explicit routes are configured in a JavaScript file, where they're written as a dictionary with a key for each unique route. The key `POST /user/signup` is called the *route address*, and it consists of the HTTP method and path. On the right side, every explicit route has a *route target*, special syntax that tells Sails what to do when it receives matching requests. In most cases, this route target consists of a controller, `UserController`, and an action, `signup`. When a request is made, the router looks for a matching route address, and if it finds one, it executes the corresponding controller action.

> **NOTE** The action is itself a JavaScript function, and the controller is a name we give the dictionary that aggregates actions under a common resource. So in the signup example, you named the controller `UserController` because the actions will all concern a user.

It's easy to get lost in all the new terminology, so let's compare a route to something you already understand, a jQuery click event, in figure 1.8.

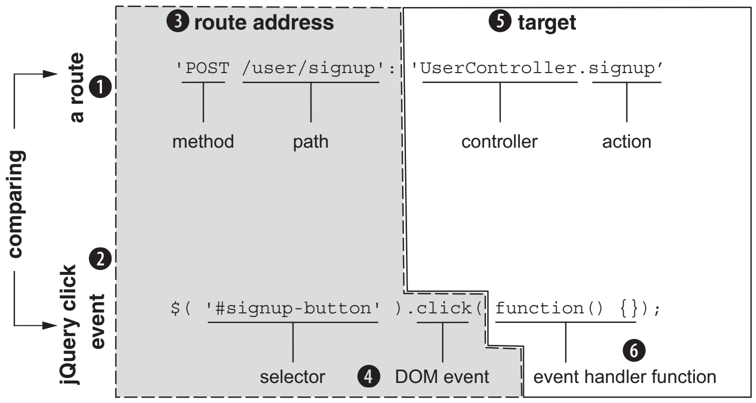


Figure 1.8 A route ❶ operates like a click event handler ❷. The route address ❸ serves a similar purpose to the combination of the DOM selector ❹ and DOM event. And the route target ❺ is similar to an event handler function ❻.

When the Sails router analyzes an incoming request, `POST /user/signup HTTP/1.1`, and it matches a route's method and path, `POST /user/signup`, it executes the controller action similarly to the way the browser analyzes an incoming DOM event, matches it against a particular selector, and executes an event handler function.

Now that you can convey intent from the frontend via a request and interpret that intent on the backend using a route and a router, let's explore how to fulfill the requirements of the request on the backend using controller actions.

1.4.4 Performing actions

To better understand controller actions (or simply *actions*), let's focus on an example: a signup form. When the user submits the form, a `POST` request is sent to `/user/signup`. When it arrives at the Sails backend, the request is automatically compared with your app's configured routes. The first matching route will be triggered, which then calls a controller action. It's the job of this controller action to fulfill the requirements of this request. Recall that actions are just JavaScript functions and that controllers are just dictionaries of related actions.

The requirements for the example endpoint (`'POST /user/signup'`) are to create a new user, store that user in your database, and respond with the status code of `200` to indicate success. If anything goes wrong, you'll want to respond with a different status code, depending on what issue or error occurred. These requirements seem simple, but they bring up some fundamental questions about Sails:

- How do you send the data harvested from your form input elements to the backend, for example, `email`, `username`, and `password`?
- How do you tell Sails where to put the new user's data? And how do you tell it which database to use?
- Speaking of that, what code do you write to store the properties of a user in the database? And where should you write it?
- How do you tell Sails you're finished—in other words, that you'd like to send a response to the requesting user-agent (your form)? And how do you tell Sails what status code and/or data to use?

A significant portion of the book is devoted to answering these questions in detail, so we mustn't get ahead of ourselves. But the least we can do is take a first step toward explaining the answers to these questions right away.

First, a bit about actions: Because you already know actions are JavaScript functions, it probably won't come as a shock that they also receive arguments. The first of these arguments (`req`) represents the incoming request, and the second (`res`) represents the eventual outgoing response. Both `req` and `res` are special objects called *streams* that come from the depths of Node.js Core. Fortunately, you rarely (if ever) have to think about them that way, because by the time you get hold of `req` and `res` in your Sails actions, they've been loaded up with a ton of useful properties and convenient functions that make your life much simpler.

The flexibility of req and res

One of the great things about Node.js is that even when you hide away complexity with helper methods, all the advanced and powerful features are still there, working their magic behind the scenes. Because `req` and `res` are still technically Node.js streams, you have as much flexibility as you would with Node.js out of the box. Imagine some ridiculously specific use case; perhaps you need to handle strange requests from a legacy point-of-sale system (read: broken-down cash register) in a small fish bait shop. And maybe that PoS system doesn't expect a normal response—instead it expects your server to slowly drip-drop each letter of the alphabet, one every second over the course of two long, excruciating minutes. No problem! You'll write your code to handle the incoming requests from that cash register in the same place you'd write any of your other request-handling code in Sails: an action.

From your action, you can access the data that the user originally typed into the form on your signup page by calling `req.param()`, one of the functions provided on the `req` dictionary. For example, when you call `req.param('username')`, it will return the value from the `username` input element in your form. This begs the question, though, how is the frontend sending these values (called *parameters*) to your action in the first place? If you were sending this request from a native iPhone app or your terminal, the way parameters are bundled would completely depend on the HTTP client library used to create the request. But in this example, because you use a web page as your frontend, you can narrow things down a bit. There are three common ways that parameters are included in a request from a web browser to the backend:

- When using a regular or traditional form submission, the contents of form input elements are included automatically as parameters in the request when the form is submitted. Depending on the method you put in your HTML, these parameters are bundled in either the request's body or in its *URL query string* (sometimes simply called the *query string*).
- When using an AJAX request, the parameters can be included in either the URL query string or in the body of a request.
- When navigating to a URL by pressing Enter in your browser's address bar, including parameters is as simple as typing out a URL query string by hand.

Remember the request line from an HTTP request we looked at earlier?

```
POST /user/signup HTTP/1.1
```

Well, the body is just another line like that in the HTTP request. It's used to transport stuff like the `email` and `password` parameters from your form. Don't overthink the term *body*. Even though it might seem foreign at first compared with something more familiar like a URL, it's just another way to stick data in a request.

The URL query string is similar in that it's another way to transport stuff inside a request, but luckily, it's even simpler to explain. You've probably seen query strings countless times already in your browser's address bar. This is because, as you can see from figure 1.9, the query string is just a part of the URL.

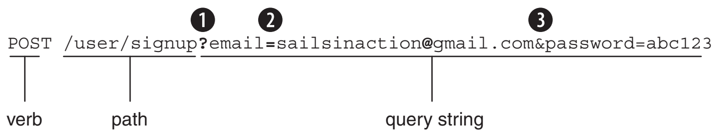

1
2
3
POST /user/signup?email=sailsinaction@gmail.com&password=abc123
verb
path
query string

Figure 1.9 An example of the URL query string that starts with a question mark, contains a key/value pair separated by an equals sign, and is separated by an ampersand

The URL query string begins with a question mark (`?`) ❶ followed by parameter key/value pairs, where the name of each parameter is separated from its value by an equals sign (`=`) ❷. The key/value pairs are separated from each other by ampersands (`&`) ❸.

When do you use the body and when do you use a query string? The short answer is most of the time the frontend framework or utility you're using makes the choice for you. For example, in jQuery if you use the `$.get()` syntax to send an AJAX request, the parameters will be transformed into a query string and tacked on at the end of the URL:

```
$.get('/dogs', {
  page: 4
}, function(data){ ... });
```

On the other hand, if you send a `POST` request using `$.post()` syntax, jQuery will bundle the parameters in the request's body:

```
$.post('/user/signup', {
email: 'sailsinaction@gmail.com',
password: 'abc123'
}, function(data){ ... });
```

So what's the difference? If the URL query string and the body are just two different ways to include parameters in a request, why use one over the other? The truth is that 99% of the time it doesn't have any practical impact on your code. A recurring philosophy in Sails is encapsulation; in other words, it shouldn't matter *how* you send parameters in your requests to the backend; what matters is *what* you send. That said, certain security considerations dictate when you can and can't safely use the URL query string, so we'll return briefly to this subject to cover best practices when we explore shortcut routes and the blueprint API in chapter 4.

We realize that it's a bit of a paradox for us to show you parts of the raw HTTP request but then go on to say that you'll rarely, if ever, interact with them in their raw state. You may be wondering, "Why do I care? You're not my algebra teacher! I don't need to know this!" Fair enough. On the frontend, we could have simply shown the syntax of how to send an AJAX request with jQuery, which demonstrates the verb and the path. We could have turned to the backend and showed the same verb and path in a route address. We could even have pointed you to a video with zooming cloud imagery and whooshing noises, to help you visualize the journey of a request in flight.

But that would be doing you a disservice. It's been the experience of both authors of this book that it was not until we *completely* demystified the raw HTTP request and response that we were able to intuitively understand how servers really work: by slurping up strings and spitting new strings back out. But enough didactics.

An important thing to remember is that you send requests to communicate intent and transport stuff, intent like "Enroll this new user, please" and stuff like `{ email: chad@hotmail.com }`. When you send a request from a browser or any other user-agent, you're simply generating a string called an HTTP request and blasting it out to the internet. Your request is just like any other string, except that it's specially formatted according to a well-defined standard called HTTP. That means it contains a method (a.k.a. verb), a URL, and maybe some headers and a body.

When your Sails server receives a request, it's parsed and routed to one of your controller actions automatically, at which point Sails runs your code. That backend code tells the server what to do next, whether that's sending an email, saving data, doing math, operating robot arms to play dueling banjos, or a combination of all these. Eventually, this backend code should always send a response; otherwise, the frontend would sit there waiting forever.

When the code in your controller action indicates that it's time to respond, Sails generates a string called an HTTP response and blasts it back out to the internet. This response is—you guessed it—also formatted according to the HTTP standard. It contains a status code and maybe some headers and a body of its own. The status code is used for specifying the outcome of the request, for example, to indicate that a new user was successfully created, or that the provided email address was already in use, or even that some other unexpected error occurred. The response body is used for transporting any relevant data back to the frontend, stuff like JSON data or an HTML web page.

Finally, back on the frontend, the user-agent (browser) receives and parses the response. Then it acts accordingly. For example, if you're using AJAX, jQuery triggers the callback function you provided. And that's it—back where you started!

Now that we've demystified the request and response a bit and set up the related terminology we'll use throughout the book, we're ready to explore what's going on in the backend code itself. We'll start with the most fundamental responsibility of any backend application: working with data.

1.5 Understanding databases

Although some experience with a database is helpful, it's not required for you to get through this book. In this section, we'll give a brief introduction to databases in the context of what you'll need to know about them while creating a Sails application.

Specifically, we'll talk about Sails models and the methods used to access various databases. We'll also take a deep dive into the subject of models in chapter 6. If you're already familiar with these concepts, feel free to skip to section 1.6.

A database can seem mysterious at first. But it's just another application: an application that stores an organized collection of data into records. In most cases, but not always, the database stores records in nonvolatile memory like your computer's hard drive. Or, infrequently, the records are stored using volatile memory like the RAM in your computer. A database even has its own API, similar to the one you'll design in the coming chapters. But unlike the web API you'll build in this book, which uses HTTP to communicate between the client and server, the underlying protocol you use to communicate between a Sails app and a database is abstracted away for you by a built-in component of Sails called *Waterline*.

TIP What's the difference between Sails and Waterline? Sails is composed of many Node.js modules that work together to help you build web applications. Waterline is one of those modules.

Waterline gives your Sails apps an abstraction layer on top of underlying databases like MongoDB or PostgreSQL, providing methods that allow you to easily query and manipulate data without writing PostgreSQL-specific integration code. Sails organizes these methods in a dictionary called a *model*.

1.5.1 What's in a Sails model?

A Sails model is a JavaScript dictionary representing a *resource* such as a MySQL table or a MongoDB collection. Every model contains attribute definitions, model methods, and other settings. When you start a Sails app, the framework automatically builds up model dictionaries from a variety of configuration files in your project, adding a whole suite of powerful methods. Your code can then use these methods to find and manipulate database records (a.k.a. rows). Let's look at the PostgreSQL database as an example and use the signup page frontend as a reference. You might define a model called `User` to store `username`, `email`, and `password` attributes, as displayed in figure 1.10.

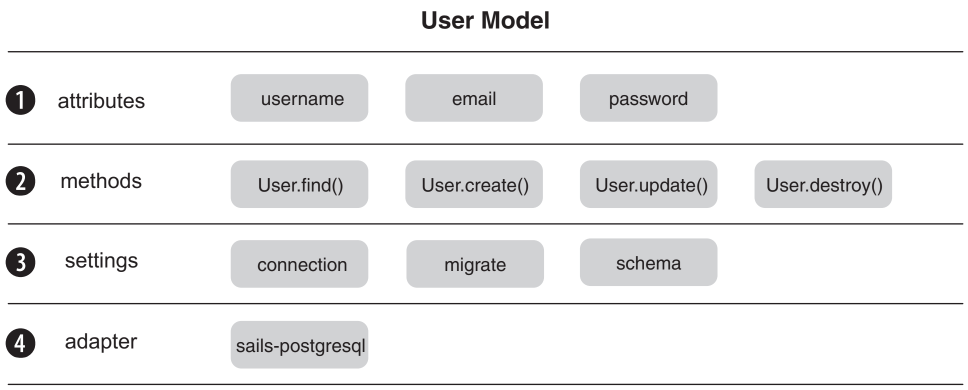


Figure 1.10 The components of a model include attributes, methods, and settings.

The attributes ❶ describe the properties of each `user` record that the database will be tasked with managing—in this case, `username`, `email`, and `password`.

> **NOTE** Attribute definitions are optional when working with some databases like MongoDB, whereas other databases like PostgreSQL require predefined attributes.

Model methods ❷ are the built-in set of functions provided by Sails that you use to find and manipulate records. Model settings ❸ include configurable properties like `connection`, `tableName`, `migrate`, and `schema`. Of particular importance is the `connection` setting, which describes the database the model methods will be run on.

> **NOTE** In Sails v1.0 and above, the `connection` setting for a model is referred to as its *datastore*. To make sure you're comfortable with both terms, we'll use them interchangeably throughout the book.

For example, if you use the `User.find()` method to find a particular record, this option tells Sails which database to search. The `connection` points to a dictionary that contains configuration information like the host, port, and credentials necessary to access the database. If any of that sounds unfamiliar, don't worry—we'll come back to it a few times throughout the book. Another model setting, `migrate`, designates how Sails should handle existing records in the database and whether or not to use automigration. As a final example, the `schema` setting allows you to enforce the use of a schema, even if the underlying database would allow you to proceed without one. This is particularly useful for schemaless databases like MongoDB.

The adapter ❹ is a Node.js module that allows your model to communicate with virtually any type of database, whether that's a traditional relational database like PostgreSQL or a non-relational database like MongoDB. As long as you install the adapter for a particular database, your app can talk to it using built-in model methods provided by Sails. Behind the scenes, the adapter takes care of translating code that uses model methods into the specific queries required by the underlying database system. The adapter to use for a particular model is determined by its `connection` setting.

1.5.2 Sails model methods

Earlier we briefly mentioned blueprint actions: `find`, `create`, `update`, and `destroy`. These built-in actions are provided by Sails, but, internally, they use functions we call *model methods* to fetch and manipulate records in a database. These are the same methods you'll call in your custom controller actions later in the book. Table 1.3 displays the most commonly used model methods provided by Sails.

Table 1.3 Common model methods

Method	Description
`.create()`	Creates a new record in the database
`.find()`	Finds and returns all records that match a certain criteria

Table 1.3 Common model methods *(continued)*

Method	Description
`.findOne()`	Attempts to find a particular record in your database that matches the given criteria
`.update()`	Updates existing records in the database that match the specified criteria
`.destroy()`	Destroys records in your database that match the given criteria
`.count()`	Returns the number of records in your database that meet the given search criteria

We'll start messing with databases in chapters 4 and 5 and get immersed in them in chapter 6.

NOTE In Sails, like most web frameworks, you can write code that works with the database that can be run from anywhere from tests to custom scripts. But for most apps, the overwhelming majority of the data-manipulation code you write will be triggered as the result of incoming web requests.

1.6 Putting it all together in a backend API

Now that we've covered the fundamental pieces of any Sails application, let's take what you've learned so far and see how it all fits together. You saw how the frontend talks to your Sails app by sending HTTP requests and how your Sails app replies with HTTP responses. We looked at how every time your Sails app receives a request, it uses your configured routes to determine which controller action to run. And in the last section, we introduced model methods, which are just one example of the many Sails and Node.js library methods you can call from the backend code in your controller actions. But, in theory, you could create almost any imaginable server-side web application with routes and controller actions alone. Routes and controller actions are the fundamental pieces of any Sails application. In practice, controller actions usually leverage many additional library methods provided by Sails and Node.js.

Controller actions can be simple or complex. For example, in the same app you might write one controller action (`PageController.showHomePage`) that simply responds with an HTML web page and another (`CartController.checkout`) that uses model methods to fetch data, calls out to a service with custom business logic, contacts a third-party service to process a bitcoin transaction, and responds with a `200` status code to indicate success. Thinking about the different parts of your application this way can get very complicated very quickly—particularly as time passes and more hands touch the code.

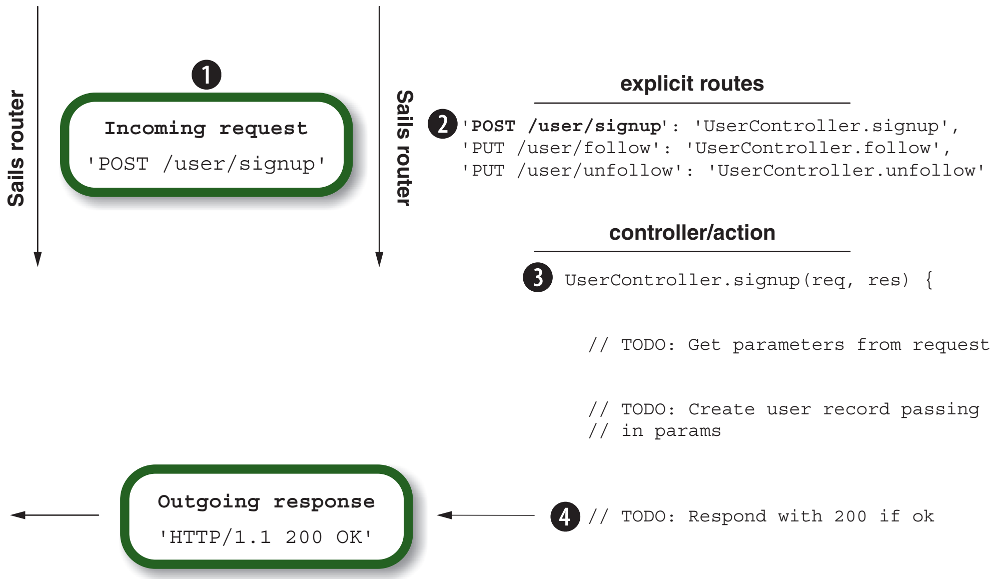
Luckily, there's another, simpler way to reason about the backend that's widely accepted by developers all over the world. Regardless of *what* a particular controller action (or endpoint) does, it's usually pretty easy to discuss *how it responds* and *why it ran* in the first place. Instead of focusing on the code inside the controller action, you can simply consider the request you need to send from the frontend to kick it off and the response you expect to receive in return. In the example of the complicated controller action we mentioned earlier (`CartController.checkout`), instead of thinking about the mechanics of working with the database and calling a third-party service, you can simply remember that to call the endpoint you need to send a `POST` request to `/checkout` and that you can expect a `200` status code in response (provided everything went according to plan).

Any abstraction that allows developers to think about what to call and what to expect in return (instead of having to be aware of the internals of how something works) is called an *application programming interface (API)*. More specifically, when talking about HTTP requests and responses, we call this a *backend API*.

DEFINITION At times you might also hear the backend API called a web API, cloud API, or even simply the API. No matter the variation in terminology, rest assured that this just refers to the interface exposed by the routes and actions in your Sails app.

Figure 1.11 provides a birds-eye view of how all the pieces we discussed earlier in this chapter work together in harmony to expose a backend API from your Sails app to the world. When Sails receives an incoming request ❶, it matches it against one of your app's routes ❷. Then, it runs the appropriate controller action ❸, whose job it is to send some sort of response ❹.

You'll see this pattern repeated throughout the book.



```
❷ 'POST /user/signup': 'UserController.signup',
  'PUT /user/follow': 'UserController.follow',
  'PUT /user/unfollow': 'UserController.unfollow'

❸ UserController.signup(req, res) {

    // TODO: Get parameters from request

    // TODO: Create user record passing
    // in params

❹  // TODO: Respond with 200 if ok
```

Figure 1.11 An endpoint from a backend API in action, processing a request from a signup form

1.6.1 Other types of routes

In addition to the explicit routes you define for serving web pages or working with the database, Sails includes some additional routes of its own, named *shadow routes*. Unlike explicit routes, which you write yourself, shadow routes are exposed automatically behind the scenes. Many web frameworks have a similar concept of automatic routing, specifically for assets. For example, adding a file called foo.jpg to the folder configured as the web root for an Apache web server implicitly causes `GET` requests to /foo.jpg to respond with the contents of that file. As you might expect, Sails provides a similar abstraction for static assets like images and client-side JavaScript files, sometimes called *asset routes*. These routes are exposed automatically and map directly to any files in the configured web root folder (.tmp/public/ by default). We'll examine asset routes extensively in chapter 3.

The framework also exposes a couple of other important shadow routes that we'll cover in this book:

- *Blueprint routes* automate the prototyping phase of backend development by providing an easy way to work with blueprint actions through a conventional API. We'll cover blueprints extensively in chapter 4.
- The *cross-site request forgery (CSRF) token route* is a built-in utility designed for use as part of an overall protection technique to prevent CSRF attacks. We'll cover this shadow route when we show how to secure your applications against CSRF vulnerabilities in chapter 15.

TIP Like any of the other "magic" features in Sails, you can use as many or as few of them as you like. Every shadow route can be disabled via configuration or overridden on a case-by-case basis by defining an explicit route with the same HTTP method and URL pattern.

1.7 Our backend design philosophy

Now that you have a better understanding of both the components of a backend API and how they function, it's worth spending a moment on the overall approach we'll take in this book. When you set out to build a web application, it's difficult to know exactly where to start. Conventional wisdom is mixed on the subject; some books suggest starting with UML diagrams and data modeling before you write a single line of code. More recently, the "Ship early, ship often" mantra (for example, Facebook's "Move fast and break things" motto) is becoming increasingly popular. This approach suggests getting to a first prototype as quickly as possible.

We've built many startup products and enterprise tools, and in every case we've found that the best place to begin is from the user's perspective. We call this a *frontend-first* approach to backend design; see figure 1.12.

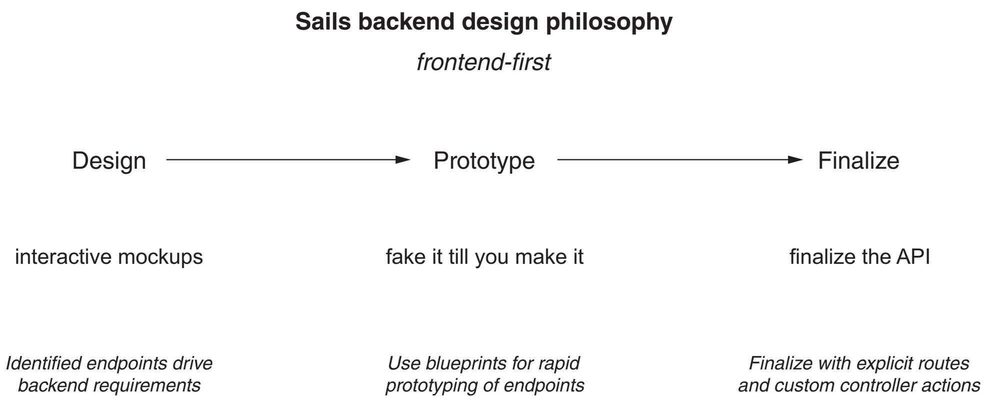

Sails backend design philosophy
frontend-first
Design
Prototype
Finalize
interactive mockups
fake it till you make it
finalize the API
Identified endpoints drive backend requirements
Use blueprints for rapid prototyping of endpoints
Finalize with explicit routes and custom controller actions

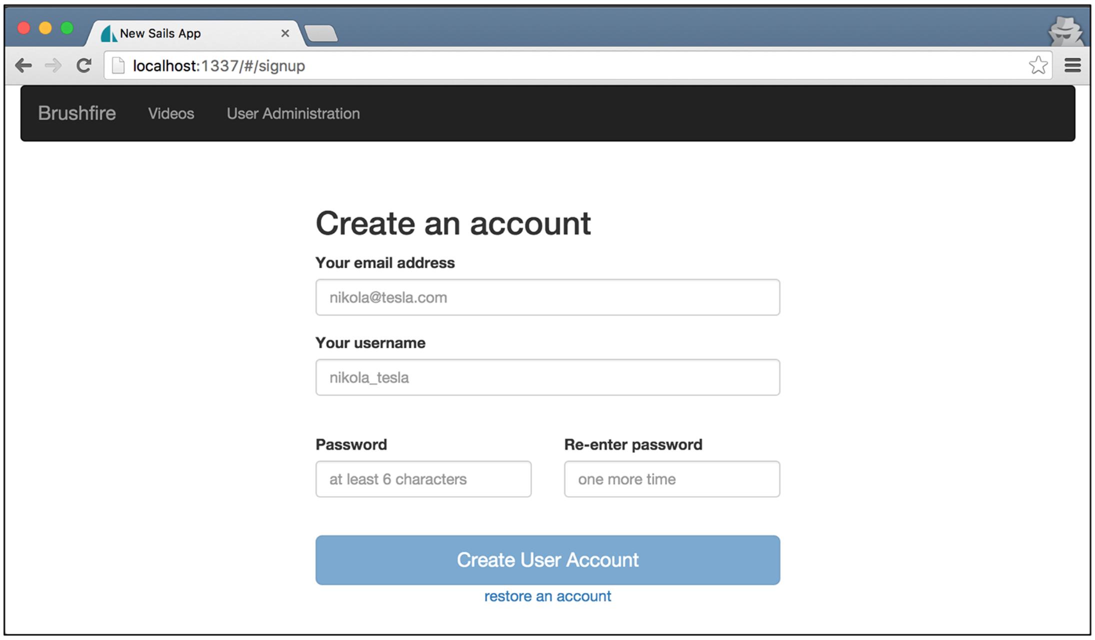
Figure 1.12 The frontend-first approach to backend development

Too often, development can get mired in what-if backend programming, that is, programming the backend to handle all of the things that the user *might* do rather than figuring out what the frontend will actually *allow* them to do and implementing only those features. Without direction, you can waste a lot of time creating things that are either unnecessary or aren't compatible with how the user will ultimately engage the frontend. Even worse, once created, backend code must be maintained—whether it's used or not! It's critical to spend the time necessary to identify the requirements for each of the API endpoints you build. Even if you think that an endpoint might be used in more ways down the road, the important thing is to optimize for the frontend you have today. It's always better to build the simplest, most specific API that meets your needs, even if it might need to be changed substantially someday as new features are added to the user interface.

If you come from a design or user experience–design background, this may sound familiar. When designing user interfaces, we always prioritize the needs of human users before making decisions on the implementation. Similarly, as backend developers, it's our responsibility to make sure that user interaction drives backend functionality and not vice versa.

1.7.1 Starting with just the frontend code

The easiest way to make sure you build exactly the backend API you need is to build the frontend part of your app first. Until you add the real backend, this will feel more or less like a fake, interactive mockup. But it captures the basic functionality of the interface you're trying to build, and it ensures that you've taken all the requirements into account before you begin. For example, the signup form in figure 1.13 inspired the design of the `POST /signup` endpoint we showed previously in section 1.4.



Create an account

Your email address

nikola@tesla.com

Your username

nikola_tesla

Password

at least 6 characters

Re-enter password

one more time

Figure 1.13 An interactive mockup of a signup form. The fields in this form help determine the request parameters you should expect when designing the API endpoint to process it.

This interactive mockup consists of the code necessary to drive the frontend user experience. For websites and single-page apps, this is HTML, CSS, images, and client-side JavaScript files. For an iOS native app, it's the .nib files, Swift scripts, and other assets you need to compile your project in Xcode. The goal is to finalize the key pieces of the frontend of the user interface, because that will identify all the requests that will need to be made from a particular screen, as well as the requirements of each request. Figure 1.14 shows an annotated example of how you might design your API endpoints based on the requirements of this page.

Not only does this approach help you notice inconsistencies in requirements and catch gaps in the feature set early in the process, but it also allows you to punt on critical architectural decisions until you know more about how your application will work. Once you've created interactive mockups and used them to identify the requirements of your backend, you can use tools that Sails provides to quickly transform those interactive mockups into working prototypes.

1.7.2 Prototyping with blueprints

So far, we've focused on how you can create your own custom routes and controller actions to create backend APIs in Sails. Recall that in section 1.5, we showed how you might combine an explicit route and a custom action to expose an API endpoint for

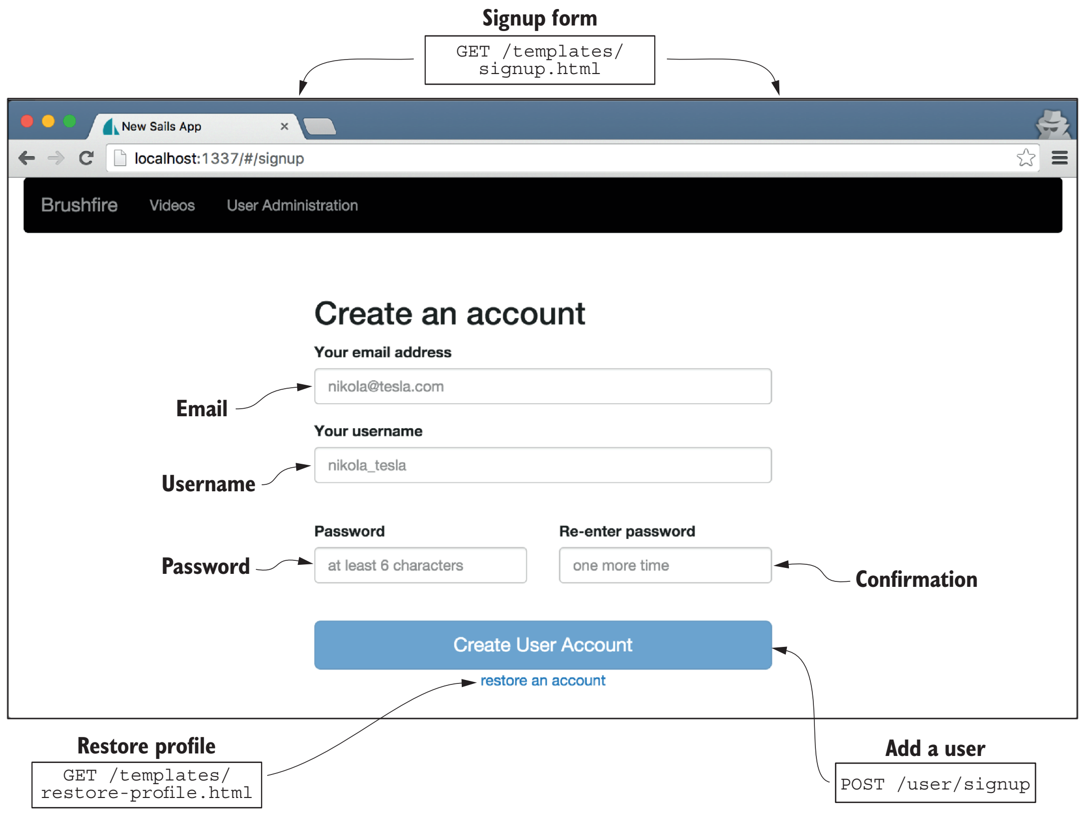


Figure 1.14 An annotated mockup showing the requests this user interface will send to your Sails app

handling new user signups. Under the covers, it's hard to say exactly what this API endpoint might need to do; you don't know enough details just by looking at the form in isolation. It might send a welcome email, encrypt a password, or even send a confirmation text message. But even without knowing all of the details, you can at least assume that it would need to create a new record in your database.

Traditionally, in this scenario, frontend developers were forced to use `setTimeouts` or to create a dictionary to fake a response with some JSON data. This allowed developers to test loading spinners and gave the user interface code some data to use temporarily until a backend API endpoint similar to the one for the signup page was available. Fortunately, for many use cases, Sails blueprints make this kind of temporary, throwaway code unnecessary. Instead, frontend developers can just set up a quick set of API endpoints (a JSON CRUD API) around a particular resource, such as a user. Those endpoints are then immediately available to use from frontend code, meaning that the frontend code can be finished and hooked up to the server ahead of any custom backend work.

NOTE JSON is called a lightweight data-interchange format. What that means for you is it's a way to safely transfer data from the client to the server and vice versa. In Node.js or the browser, you can take almost any JavaScript value stored in memory (for example, a variable containing a dictionary with an email address, password, and username) and stringify it, converting it into a specially encoded string. That string can then be transported over the network from the backend to the frontend or vice versa. On the receiving end, the JSON string is parsed back into the original JavaScript value.

Because you can create them very quickly, blueprints are incredibly useful during the prototyping phase. Instead of having to manually create the routes, controller actions, and model methods necessary to create an API before you even understand what it needs to do, you can use Sails' blueprint API to supply similar functionality. To set up blueprints for your signup example, you need only issue a single command in the terminal window:

```
~/sailsProject $ sails generate api user
```

Then, the next time you start the Sails server with `sails lift`, you'll have access to a JSON CRUD API around the `user` resource. Table 1.4 shows the shadow routes and built-in controller actions that this exposes automatically.

Table 1.4 Shadow routes and built-in controller actions exposed by the blueprint API

CRUD operation	Blueprint shortcut route			Blueprint RESTful route		
	Route address		Target	Route address		Target
	Verb	Path	Blueprint action	Verb	Path	Blueprint action
Read	`GET`	`/user/find`	`find`	`GET`	`/user`	`find`
Read	`GET`	`/user/find/:id`	`find`	`GET`	`/user/:id`	`find`
Create	`GET`	`/user/create`	`create`	`POST`	`/user`	`create`
Update	`GET`	`/user/update/:id`	`update`	`PUT`	`/user/:id`	`update`
Delete	`GET`	`/user/destroy/:id`	`destroy`	`DELETE`	`/user/:id`	`destroy`

In chapter 4, we'll examine what each blueprint action can do. For now, just note that each action corresponds to a CRUD operation. So, instead of creating a custom route and controller action to handle form submissions from the signup page, you just ran a command on the terminal, and Sails took care of setting all that up for you.

Why not use blueprints for everything? The truth is, for most applications, CRUD alone isn't enough, and you'll need to write a custom controller action for most if not all of your endpoints. For example, your signup endpoint will eventually need to encrypt the user's password, and as we mentioned earlier, you might also want it to send a welcome email (or someday, even a text message). Fortunately, when the time comes, overriding blueprint actions is just as easy as making your own custom controller action. And, in the meantime, your frontend code has gone from an interactive mockup to a full-fledged, server-driven prototype.

Shortcut blueprints

You might have noticed a subset of blueprint routes known as *shortcut blueprint routes* (or just *shortcut blueprints*). These are just more shadow routes that point to the same, built-in blueprint actions. The only difference is that you can access all of them from your browser's URL bar. Seems like a bad idea, right? That's why you should *never* enable shortcut blueprints in your production application.

What makes shortcut blueprints so insanely useful is that they allow you to quickly access and modify your data during development without needing to rely on a database-specific client like phpMyAdmin. As you build your application throughout this book, you'll take advantage of this Sails feature frequently.

1.7.3 Finalizing your API

There comes a point when blueprint actions alone are insufficient to meet the requirements of the frontend. Fortunately, transitioning to custom controller actions is easy: as we discussed earlier in this chapter, you just write code for the actions and then define explicit routes that point at them. As you can see in figure 1.15, the implementation of your Sails app doesn't affect the interface. In other words, as long as

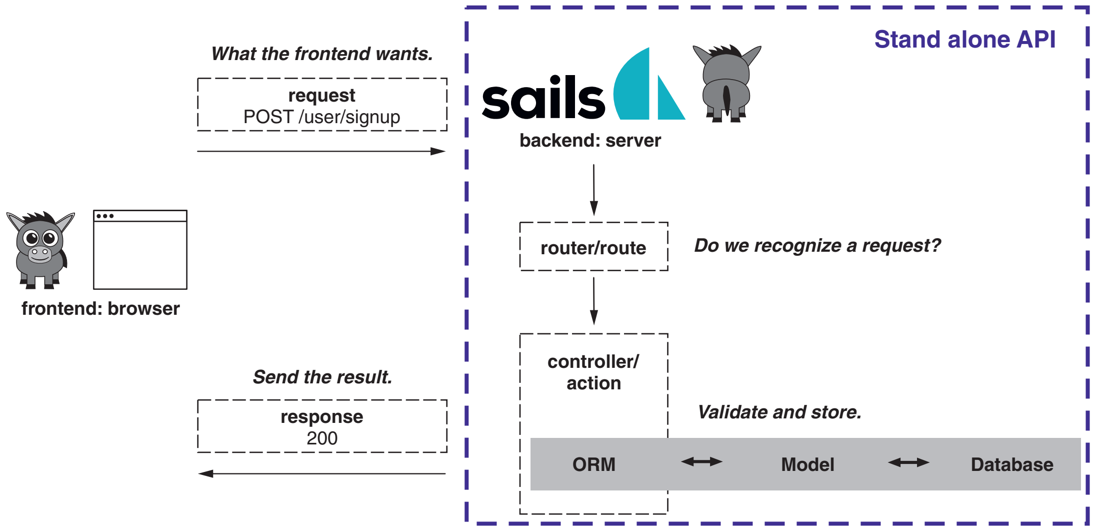


Figure 1.15 As long as the expected request and response for an API endpoint remain consistent, frontend code doesn't need to change.

your custom controller actions are written to be compatible with the requests that your frontend sends and the responses it expects, then your frontend code doesn't need to change at all.

Remember, controller actions are just JavaScript functions. This makes them incredibly powerful, because they can do anything that a JavaScript function can do. But with such great power comes great responsibility. You'll want to protect some of your actions, so that only certain logged-in users are allowed to run them. Fortunately, Sails provides a powerful feature for managing access to controller actions called *policies*. We'll explore and implement policies in chapter 10.

1.8 Delivering frontend assets

Now that you understand how clients and servers communicate and how to design and build backend APIs, we'll turn our attention to the frontend itself. *Wait a second, isn't Sails a backend framework?* It is! But for certain kinds of apps, the backend is responsible for *delivering* frontend assets. Whether that fits your Sails app depends on the types of frontends you're building or, more specifically, the types of *user-agents* your application will need to support.

DEFINITION A *user-agent* is any program that makes a request, such as browsers, spiders (web-based robots), command-line tools, custom applications, and mobile apps.

When we use the term *frontend*, we're talking about the user interface elements of your application. Figure 1.16 depicts the universe of common frontend user-agents for web applications.

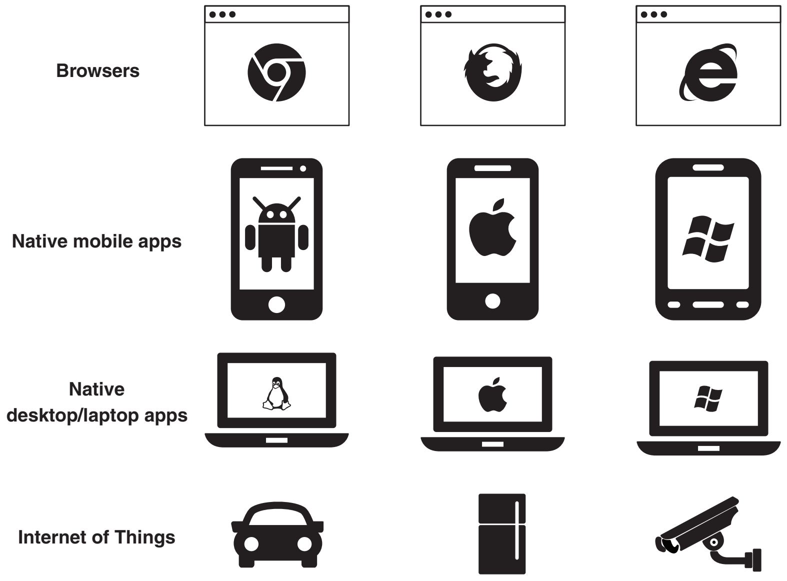


Figure 1.16 Examples of frontend user-agents used in web applications

If you were building a smart toaster or a native mobile or desktop application, you could skip ahead to chapter 4 and jump right into building and integrating a stellar, standalone API with Sails. Why? Because the frontend assets for Internet of Things (IoT), native mobile, and native desktop applications usually aren't distributed on the web. Instead, they're downloaded from an app store or bundled on a piece of hardware. Therefore, Sails can be blissfully unconcerned about their delivery. In that case, as shown in figure 1.17, all your Sails app has to worry about is requests for data (like a high score list) and behavior (like sending a text message or processing a signup).

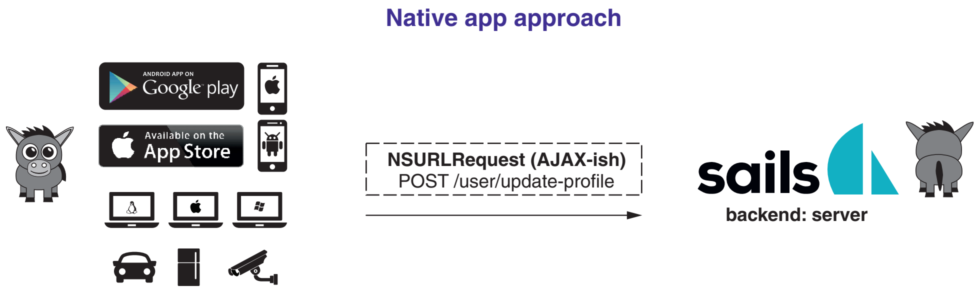

Native app approach
ANDROID APP ON Google play
Available on the App Store
NSURLRequest (AJAX-ish)
POST /user/update-profile
sails
backend: server

Figure 1.17 Sails used as a pure API with no responsibilities for delivering frontend assets

> **NOTE** There are two cases that may necessitate Sails delivering native app elements: for example, apps built using a frontend wrapper framework like PhoneGap or Electron. PhoneGap uses a browser within a native mobile app to display the UI. Some native app developers opt to deliver some or all frontend assets (for example, HTML, CSS, and JavaScript) via Sails because it allows for a greater degree of flexibility. In this case, treat the frontend like a browser user-agent and a single-page app.

Once installed, native and IoT applications make normal requests to Sails endpoints that fulfill backend requirements like storing and sharing data.

On the other hand, browser user-agents rely on some combination of HTML, CSS, and JavaScript for the frontend user interface. Instead of visiting an app store, users download the frontend app (or web page) by visiting a URL in their browser. If you plan to build an app that will support web browsers, then you need to decide how each page or view in the app will be delivered and ultimately rendered. There are two basic approaches: single-page apps (SPAs) and hybrid web applications. Figure 1.18 illustrates the two kinds of requests you can expect to see when building an SPA.

This approach is not too different from the approach for native apps because it relies on client-side rendering. The only real difference is that in addition to exposing endpoints for fetching data and triggering backend logic, Sails may also need to deliver static assets: files like images, HTML templates, client-side JavaScript, and style

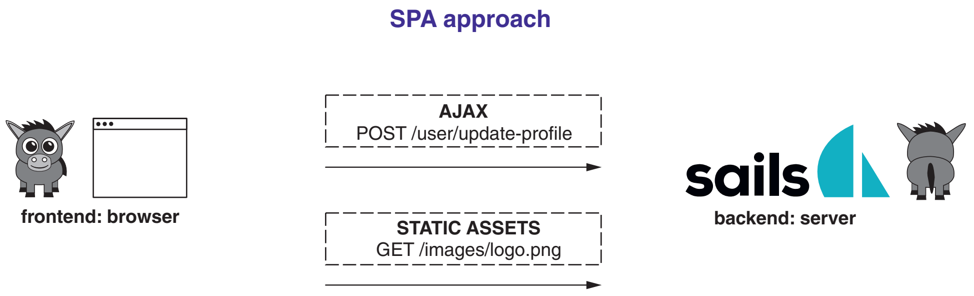


Figure 1.18 Typical frontend requests to the backend using the SPA approach

sheets. Using this approach, Sails delivers the initial HTML page as a static asset, and then client-side JavaScript (running in the browser) is responsible for making intermediate changes to the view via AJAX requests to the backend API. Wholesale page navigation, if any, is managed by special client-side JavaScript code (sometimes called a *client-side router*).

The second approach, a hybrid web application, relies (at least to some degree) on server-side rendering. That means Sails is responsible for preparing personalized, dynamic web pages on the backend from special templates called *views* and then delivering the personalized HTML to the browser. Figure 1.19 illustrates the kinds of requests you can expect to see if you're building a hybrid web application.

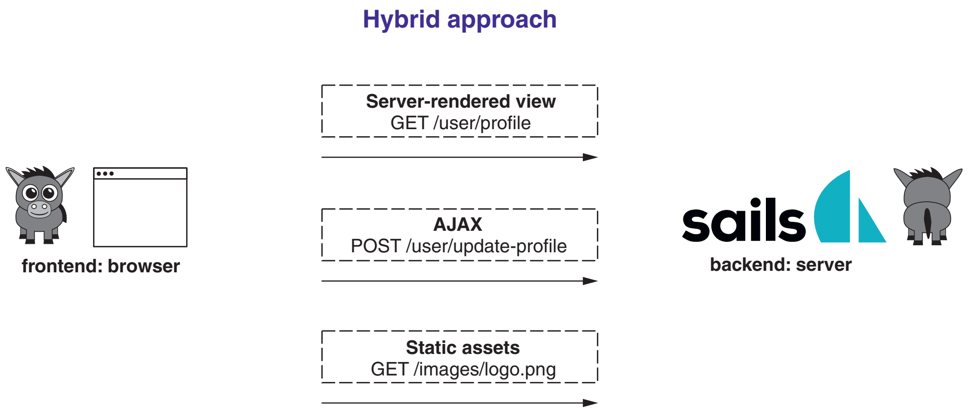


Figure 1.19 Delivering personalized HTML and static assets to a hybrid web application

Using this approach, Sails provides the initial server-rendered view for some or all of the pages on a website. Client-side JavaScript *might* also update the DOM by making calls to the Sails app, but most or all navigation between pages in a hybrid web application is handled by allowing users to navigate between different URLs in the browser, fetching freshly personalized HTML from the server each time.

Our experience, based on many client projects, has shown that when in doubt, the hybrid approach provides the best overall results. But in an effort to give you a broad knowledge base, we'll demonstrate both the SPA and hybrid approaches. We'll start by building an SPA in chapter 3. When it comes time to incorporate user authentication, access control, and SEO in chapter 8, we'll transition to the hybrid approach.

1.9 Frontend vs. backend validations

We'll address security throughout the book, with some extra emphasis on the subject in chapter 15. But in the meantime, we need to focus on an important security concept before you start building your application: whom can you trust? There are two basic realms in a web application: the frontend and the backend. Each of these realms guarantees a different level of trustworthiness and therefore requires a different degree of rigor when it comes to security, as depicted in figure 1.20.

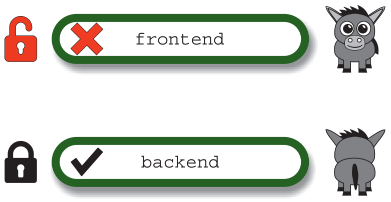


Figure 1.20 The two security realms of a web application

We'll address the implications of this security reality as they come up periodically throughout the book. For example, in chapter 7, we'll introduce frontend validations to restrict users from creating a password with fewer than six characters. But because you can't trust the frontend, it's important to be aware of the possibility that the same user could maliciously use a tool like Postman or cURL to make any conceivable request from outside the browser, thus completely bypassing whatever frontend validation you put in place.

> **NOTE** Another example of a security concern is a frontend that won't let the user submit a form until they fill out a required field. This is good UX, but your controller action on the backend still needs to do its own check, because it can't trust that the corresponding parameter will exist in the incoming request.

If you've done any sort of backend development, this concept might be old hat, but it's important enough that we wanted to address it up front. If this is a new concept for you, just remember this: you have to design your backend applications under the assumption that any given request *might* be malicious and could contain anything.

1.10 Realtime (WebSockets)

So far in this chapter, we've used HTTP to communicate between the user-agent (frontend) and the Sails server (backend). For most traditional web applications, this is all you need. The frontend always initiates requests, and whenever it receives a request, the backend responds. But for some apps that rely on features like chat (Slack), schedules (Nest thermostat), and realtime location tracking (Pokémon Go), this isn't enough.

Sails apps are capable of full-duplex realtime communication between the client and server. Instead of always having to initiate requests itself, client-side code can establish and maintain a persistent connection to a Sails server, allowing your backend code to send messages to individual clients or to broadcast messages to whole segments of your user base, at any time. In chapter 2, when you generate a new Sails app, start it up, and open your home page in the browser for the first time, you'll witness this behavior firsthand.

Sails implements support for realtime messaging and persistent connections using Socket.IO, a popular MIT-licensed open-source tool that helps ensure a wide array of legacy browser support, including Internet Explorer 7 and up. We'll explore WebSockets extensively in chapter 14.

DEFINITION In this book, we'll use both the terms *sockets* and *WebSockets* to refer to a two-way, persistent communication channel between a Sails app and a client. Communicating with a Sails app via WebSockets is really a form of AJAX, in that it allows a web page to interact with the server without refreshing. But sockets differ from traditional AJAX in two important ways: First, a socket can stay connected to the server for as long as the web page is open, allowing it to maintain state. Traditional AJAX requests, like all HTTP requests, are stateless. Second, because of the always-on nature of the connection, a Sails app can send data down to a socket at any time, whereas AJAX allows the server to respond only when a request is made.

1.11 Asynchronous programming

One of the highest hurdles for most new Node.js developers is learning how to write asynchronous code. Even if you're already familiar with AJAX callbacks, timeouts, and event handlers from client-side JavaScript, the sheer number of nested callbacks that show up when writing JavaScript on the server can be a bit intimidating at first. There are also new patterns to learn: concepts like asynchronous loops (`async.each`), asynchronous recursion (imagine building Dropbox in Node.js), and asynchronous conditionals (`if/then/finally`); or doing something asynchronous under some conditions and something synchronous under others.

In this book, we don't expect you to have any past experience writing asynchronous functions. We'll cover that in depth throughout the coming chapters. But before you start, it's a good idea to get familiar with what it means to use an asynchronous function and what that looks like.

Asynchronous JavaScript programming is very similar to web programming on the frontend. In a browser, you might want to trigger a function each time a button on the page is clicked. So you bind an *event handler* (a.k.a. *event listener*), which is just a callback function that will be executed whenever the button is clicked. Let's look at an example using jQuery.

Listing 1.1 jQuery callback pattern

```
$('#my-button').click(function whenClicked (){
  $.get('some3rdpartyAPI', function(data) {
    $('.result').html(data);
  });
});
```

Sets up a callback function (whenClicked) that will run anytime the DOM element identified by #my-button is clicked

Listing 1.1 shows code that binds a callback as an event handler. Whenever the user clicks the specified button, the callback function (`whenClicked`) will run.

Now let's look on the backend for something similar. Let's say you want to create a user in a database. The time it takes to create the record in a database can vary, and you don't want every incoming request to your app to have to wait. Herein lies the beauty of Node.js, Sails, and server-side JavaScript in general: instead of blocking all incoming requests while the server communicates with the database, file system, or other third-party APIs, Node.js keeps working, allowing other requests to be processed while it waits, granting Node.js apps a huge scalability and performance boost.

But like everything in life, this comes with a price: instead of simply returning a value or throwing an error like normal code you might be used to, asynchronous function calls in Node.js expect you to provide a callback function. When Node.js hears back from the database, whether good news or bad, Node.js triggers the callback function you provided. If something goes wrong, the first argument (`err`) will be truthy. The pattern you'll see repeatedly is something like what's shown here.

Listing 1.2 A typical Node asynchronous callback pattern

```
User.create({name: nikola}).exec(function userCreated(err, newUser) {
  if (err) {
    console.log('the error is: ', err);
    return;
  }

  console.log("The result: ", newUser);
  return;

});
```

In this example, you want to create a user named `nikola`, and then once the `user` record has been created, you want Sails to log a message to the console. You provide a callback function, `userCreated`, that will be called once `User.create()` has finished. If anything goes wrong, your callback will receive a truthy `err`, which it will log to the console and then bail. Otherwise, everything works out, so a different message will be shown with the result from `User.create()`.

The important thing to recognize as a consumer of asynchronous functions in Node.js is that your later callback will always have at least one argument: `err`. And if the asynchronous function you're calling has output (as is the case with `.create()`), then you can expect a second argument: `newUser`. You can name these arguments whatever you want; it's often useful to name the second argument something that represents the expected result. By convention, the first argument is typically named `err` and it contains what you would think: a JavaScript error instance or at the very least some truthy value. This allows you to simply check `if (err) {…}` to find out if anything went wrong.

This pattern differs considerably from traditional synchronous programming, where you would do something like this:

```
var keys = Object.keys({name: 'nikola'});
```

In this example, when `Object.keys()` runs, the process is completely blocked until the JavaScript runtime can calculate an array consisting of all the keys from the specified dictionary. In the meantime, no other code runs, no callbacks are fired, and no new requests are handled. If everything works out, the synchronous instruction (a.k.a. function call), `Object.keys()`, returns the result (`['name']`). If something goes wrong (if this was `Object.keys(null)`, for example), then `Object.keys` will throw an error.

Handling uncaught exceptions

Possibly the most important thing to remember about writing code for Node.js is that throwing uncaught exceptions inside any callback from an asynchronous function *will cause your server to crash*. So it's imperative that, when writing code inside an asynchronous callback, you wrap anything that might throw in a `try/catch` block.

But don't worry! We'll reiterate this again and again throughout the book to help drive the point home. And once you've gotten used to this style of coding, you'll protect yourself by instinct. Eventually, you may even find, as we did, that writing code like this makes you a more efficient programmer (because it forces you to think about error conditions from the very beginning).

Finally, let's take a look at one last example that puts it all together. The next listing demonstrates what it looks like to use multiple asynchronous instructions (function calls) in a row.

Listing 1.3 Nesting other functions in an asynchronous function

```
Request.get('http://some3rdpartyAPI.com/user', function(err, response) {
  if (err) {
    console.log('the error is: ', err);
    return;
  }

  User.create({name: response.body.name}).exec(function(err, newUser) {
    if (err) {
      console.log('the error is: ', err);
      return;
    }
```

```
      console.log('The new user record: ', newUser);
      // All done!
      return;
    });//</after creating new user>
  });//</after receiving response to 3rd party request>

// No code should go down here!
```

Here, you're doing a `GET` request to some other API, `some3rdpartyAPI.com`. You don't know when the response will come back, so you provide a callback that will be triggered when the request is completed. Then (in *that* callback), you create the user based on the response you got back. Notice that instead of writing one line of code after another, when using asynchronous instructions, you'll want to nest whatever comes next within the callback of the previous instruction.

In Node.js, like in most programming languages, in synchronous instructions time flows from top to bottom. If you write two instructions, one on line 3 and one on line 4, then the instruction on line 3 will run first, followed by the instruction on line 4. But in asynchronous instructions, time flows from *left to right*. If you write two asynchronous instructions, then the second instruction must be nested inside the callback of the first.

New Node.js developers often refer to this as *callback hell*. Some developers find several strategies helpful when attempting to mitigate the amount of nesting in Node.js code (promises, fibers, `await`, and so on). There are also some trusted tricks and indispensable tools, such as an npm package called `async`. We'll cover some of our own tricks, as well as best practices for working with `async`, on a few occasions throughout the book.

For now, bear in mind that like most hells, callback hell is subjective. Asynchronous callbacks are a reality of Node.js. And until you've accumulated some serious experience working with them, they can feel a bit clumsy. But you may find that after a few months you feel just as comfortable using them as you do writing traditional synchronous code.

Mastering callbacks

We can't stress enough how important it is to master the basic use of callbacks *before* attempting to learn technologies like promises, `async/await`, or fibers. We've seen and dealt with countless timing issues and memory leaks introduced in Node.js apps. The vast majority of them could have been easily avoided by following this advice. So please learn callbacks first. It's far too easy to introduce bugs in a well-intentioned attempt to reduce the number of callbacks in your code.

The examples in this book are designed to give you plenty of reps with callbacks. If you follow along, you'll be more than prepared to make an informed decision about whether to use callbacks or promises in your own application.

Okay, enough asynchronous programming theory. Even if your head is swimming with all the new vocabulary, don't despair! We promise that in a few chapters you'll look at asynchronous functions and marvel at how much you know and how easy they are to use.

That about wraps up our primer. We're almost ready to start building stuff! But first, in the final section of this chapter, we'll outline the recurring scenario and example application that we'll use throughout the remainder of the book.

1.12 Meet Chad

This book would be boring if we just droned on and on about "feature this" and "feature that." So, to keep you on the edge of your seat (and to keep us motivated) we invented a fictional character—a friend named Chad. Likely, you're thinking: "Been there, done that. No more books about invisible friends." Don't worry. We won't make a habit of it.

Chad considers himself quite savvy in the ways of social and viral media. He explained to us that he has a vision: "I'm going to build the most virally adopted web app in history." Clearly, what Chad lacks in development experience, he makes up for in confidence. Normally, we avoid partnerships like these, but Chad is a nice guy. He even referred us to a couple of clients, and he *is* letting us sleep in his house for a couple of weeks. Long story short, we agreed to help build his vision. The only problem is that Chad's vision changes from week to week. Currently, the only thing he's sure about is that "the app *must* include YouTube video clips."

Armed with those detailed requirements, we're sure to build a prototype of something amazing. We'll pick back up on that in chapter 3 when we explore static assets. But before bringing Chad into the mix, you need to get your environment ready and take Sails for a quick spin.

1.13 Summary

- The heart of any web application backend is in handling incoming requests.
- The anatomy of a backend API includes its routes and controller actions, which deliver on the requirements of an incoming request.
- The ORM tool in Sails, called Waterline, allows you to communicate with databases like MySQL or MongoDB using JavaScript.
- Three common types of applications whose assets are delivered in different ways by Sails are native apps, SPAs, and hybrid web applications.

JAVASCRIPT/WEB DEVELOPMENT

Sails.js IN ACTION

McNeil • Nathan

Sails makes professional web development a breeze. This instantly familiar MVC framework automatically handles the tedious application boilerplate, so you can concentrate on developing features and creating business value. You get powerful tools for rapid API development, task automation, an ORM, and easy integration with any web, mobile, or IoT frontend. And because you're using Node.js, it's JavaScript all the way down.

Sails.js in Action is a comprehensive guide on how to build enterprise-capable web applications. Written by the creators of Sails.js, this book introduces each concept and technique with real-world examples and thorough explanations. As you read, you'll learn to build the backend of a typical web application while you explore real-time programming with WebSockets, security fundamentals, and best practices for building Sails/Node.js apps.

What's Inside

- Creating the backend for a web, mobile, or IoT app
- Real-time programming with WebSockets
- User management, authentication, and password recovery
- Using Sails to autogenerate REST APIs
- Custom backend development and third-party API integrations

Readers should be comfortable with JavaScript and frontend web development.

Mike McNeil is the creator of Sails.js. **Irl Nathan** is the producer of sailsCasts, a series focused on using Sails.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/sails-js-in-action

Free eBook
SEE INSERT

"Look no further—you've found the ultimate source."
—Damien White, Visoft

"This book is your path through the crazy jungle of JavaScript."
—Sam Kreter
Software Engineer, Microsoft

"Get up to speed quickly on full-stack web development using Sails.js."
—Alvin Raj, Oracle

"If you need to ship fast with Node.js, this book will definitely float your boat."
—Stephen Byrne, Dell

"Comprehensive ... equally relevant to both beginners and professionals."
—Damian Esteban, betterPT

ISBN-13: 978-1-61729-261-3
ISBN-10: 1-61729-261-3
54999
9 781617 292613

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]