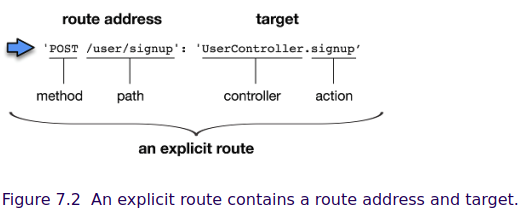
**CHAPTER 7**

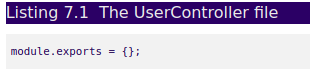
# **Custom Actions**

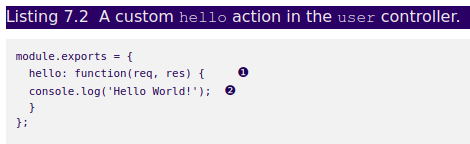
In this chapter we’ll connect front-end mockup requests to custom controller actions that use model methods to allow a user to create, modify and view their identity. Recall that the components of user identity management correlate to the front-end mockups outlined in table 7.1.

**7.1 Demystifying routes and actions**



The controller is a name we give the dictionary that aggregates ***actions*** under a common resource. Here we named the controller **user** because the ***actions*** will all concern a **user**. We define custom actions in controller files located in the **brushfire/api/controllers/** folder. In Sublime open **brushfire/api/controllers/UserController.js** similar to listing 7.1.

  
This is the controller that was generated when we created the **user** API in chapter 6 via **sails generate api user**. We’ve already demonstrated what we can do with an empty controller and empty model through blueprint routes and actions. Now, let’s add a custom action where we dictate what will occur instead of relying upon the fixed set of features in a blueprint action. Head back to Sublime and add a new custom action named **hello** in listing 7.2 to **brushfire/api/controllers/UserController.js**.



Restart the Sails server via **sails lift***.* In Postman make a **GET** request to **/user/hello** similar to figure 7.3.

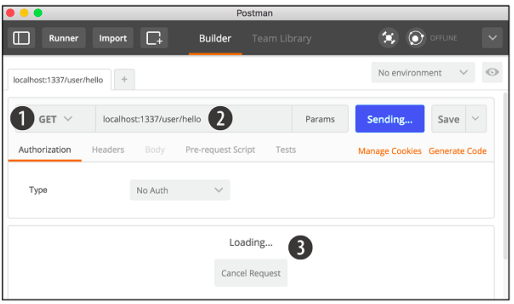
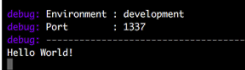


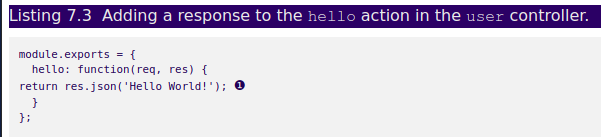
Figure 7.3 We made a ① **GET** request to ② **/user/hello** but there’s something wrong, however, with the action because the ③ loading message gets displayed and then times out.

Our ***Hello World!***message is logged to the console in figure 7.4.

We made a request but our action didn’t complete the transaction with a response. Let’s fix that.

### **7.1.1 Introducing res.json()**

The **res.json()** method responds with a **200** status code and whatever we provide as an argument to the method, formatted as **JSON** to the user-agent that made the request. In Sublime, add the following response to **brushfire/api/controllers/UserController.js** in listing 7.3.



Restart Sails using **sails lift** and make the same **GET** request to **/user/hello** in Postman

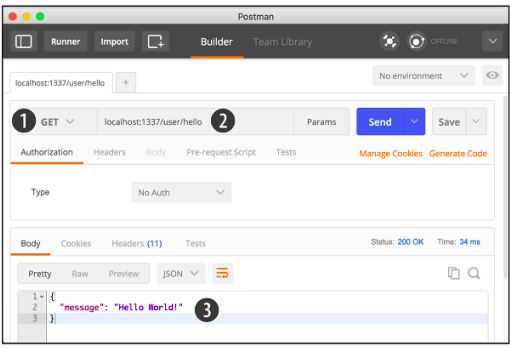


Figure 7.5 Again we’re making a ① **GET** request to ② **/user/hello**. Now that we’ve added a response in the action, Postman receives the response and ③ displays it.

Now when the request triggers the **hello** action a response is sent back with our Hello World! Message completing the request/response transaction. Notice that we also added **return** before **res.json**. This makes **res.json()**a terminal method.

A ***terminal method*** is generally the last line of code an action will execute for a given request. We use **return** as an indicator that nothing else will execute with respect to the request after this line of code.

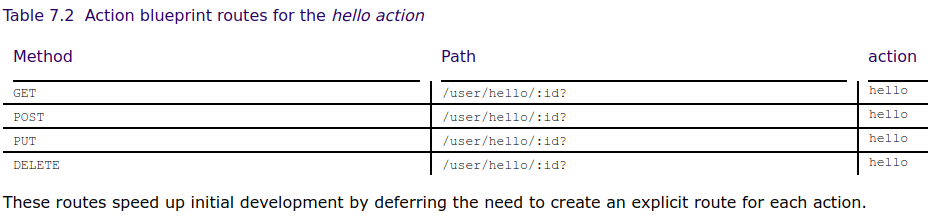
The **req** and **res** arguments in our **hello** action are JavaScript dictionaries provided by Sails. The details of how they’re created are not important. Instead it’s important to understand that, like all JavaScript dictionaries, they can have methods (functions) and properties. These methods and properties are tied to the request and response. For example, not surprisingly, in a **GET** request to **/user/hello***,* the **req** dictionary provides access to details about the request like the type of HTTP method used via **req.method**. In this case, the **req.method** property would return **GET.**

We have one more bit of unfinished business to cover. How was our custom ***hello*** action triggered without an explicit route? The answer lies in a third type of Blueprint: route called ***Blueprint: actions routes*.**

### **7.1.2 Automatic routing for custom actions**

Instead they’re available unless overridden using the same path in the **routes.js** file or disabled in **brushfire/config/blueprints.js**

So for our **hello** action, the following shadow routes are created in table 7.2 each time the Sails server starts.

Finally, we know that each request needs a corresponding response. Now that we have a solid foundation of how ***controller/actions*** work, let’s start identifying the custom actions we’ll need to fulfill the front-end mockup requests and requirements for user identity management.

## **7.2 Identifying the requirements for our custom actions**

### **7.2.1 Obtaining the example materials for this chapter**

If you haven’t completed chapter 6, you can clone the end of chapter 6Github repo at: <https://github.com/sailsinaction/brushfire-ch6-end.git> and start from there. Remember to use **npm install** in the terminal window from the root of the project after you clone the repo.

**If you do choose the cloning option**, don’t forget to add the **brushfire/config/local.js** file with your Google API key from chapter 5 (section 5.4.6) as well as starting your local PostgreSQL **brushfire** database from chapter 6 (section 6.4.2).

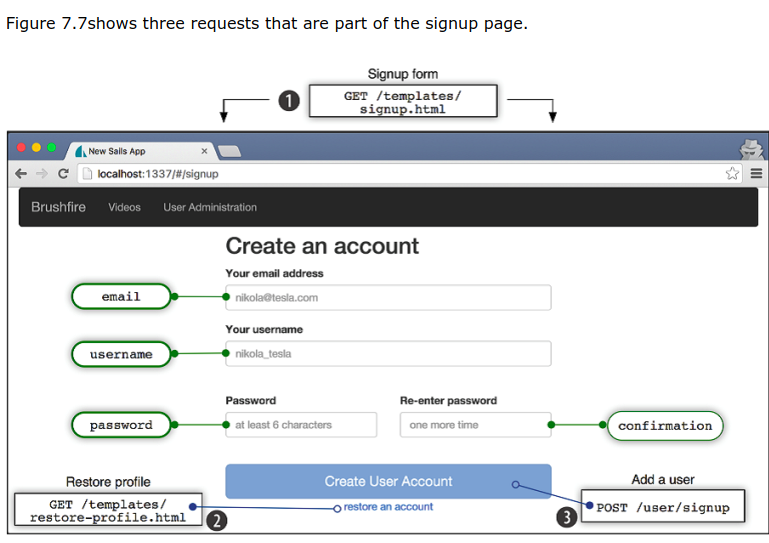
## **7.3 Handling a signup form**

For each mockup we’ll identify:

· all of the requests on the page including the request that initiated the display of the page itself.

· the inputs that will be sent with a request.

· the requirements and expectations of the response back to the front end.



**Figure 7.7** The signup page mockup contains three endpoints. ① The first request initiated the display of the **signup** page, which is handled by an asset route. ② The second request is a link to the **restore-profile** page that’s handled by an asset route and ③ is a **POST** request to **/user/signup** that will be handled by a custom controller action.

### **7.3.1   Naming custom actions**

We could use the traditional CRUD operation labels, in this case naming the action *create*. This would, however, overwrite the *blueprint: (create) action*. With all of that said, we think it’s more accurate to be descriptive in naming the action and will therefore use the name *signup*.

### **7.3.2 Creating an action**

To create the ***signup*** action, open **brushfire/api/controllers/UserController.js** in Sublime and add the following code in Listing 7.4.

So let’s check out the new action. Make sure Sails is running via ***sails lift*** and in POSTMAN make a **POST** request to **/user/signup** configured similarly to figure 7.9.

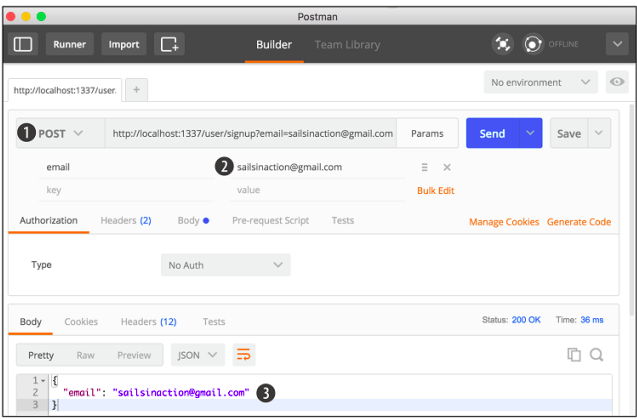
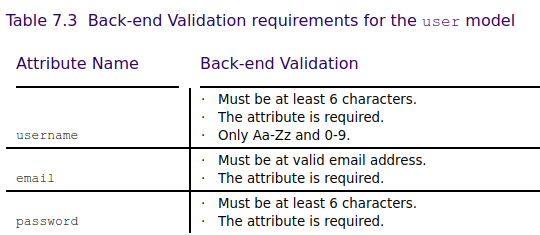


Figure 7.9 Using POSTMAN we’ll create a new user with the custom **signup** action. This will be a ① POST request to **localhost:1337/user/signup** with ② an email parameter set to **sailsinaction@gmail.com**. The **signup** action responds with the ③ email address as JSON and a **200** status code.

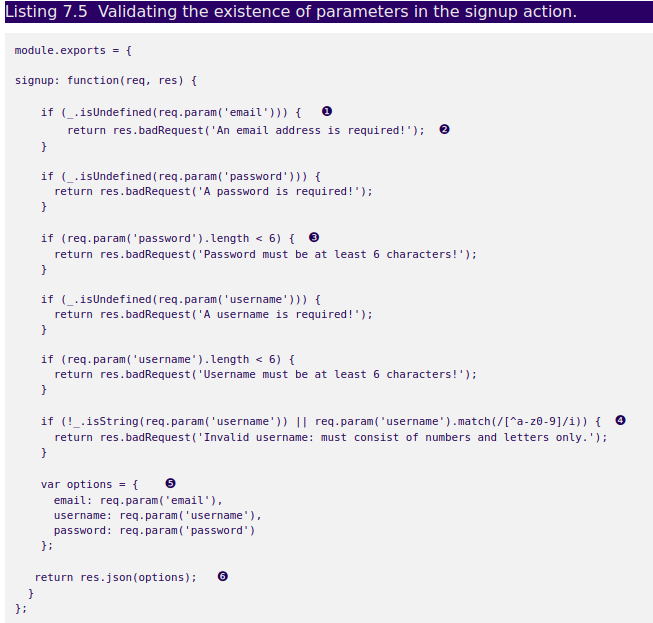
The **signup** action responds with the **email** as JSON and a **200** status code. Next, let’s take a look at the back-end validation requirements.

### **7.3.3 Introducing req.param()**

Reviewing the API Reference, specifically the Model Requirements tab, you’ll find the following Back-End Validation requirements in table 7.3.



Recall that we even though we validate user input on the front-end, we must also validate on the back end to protect against requests made outside of the browser. Head back to **brushfire/api/controllers/UserController.js** in Sublime and add the following validation code in listing 7.5.

❶ use a lodash **isUndefined()** method to determine whether a request parameter is undefined

❷ if a request parameter is undefined return a status code 400 bad request with a message

❸ check value length

❹ use a regular expression to assure only letters and numbers are used in the **username**

❺ build up a **user** dictionary that contains the parameters

❻ respond with the dictionary formatted as JSON

We’re using the lodash **\_.isUndefined()** method to check for the existence of form fields, now parameters.

lodash describes itself as a modern JavaScript utility library. The library is accessible via the global underscore **\_** symbol. As its name suggests checks if a value is **undefined.**

We’re also introducing another useful method, **req.param()***.*

**req.param()**searches the url path, query string, and body of the request for a specified parameter provided as an argument. For example, if a form field named **username**is sent in the request, **req.param(‘username’)**will return the value of the **username**field.

Also you may have noticed that if a value fails a validation test we respond with **res.badRequest().**

The **res.badRequest()** method is a default response that sends a **400** status code, which by convention means the request has some malformed syntax. We also pass a message that’s added to the error as an argument.

**NOW, I’M NOT UNDERSTANDING WHAT THE CODE IN LISTING 7.5 DO?**

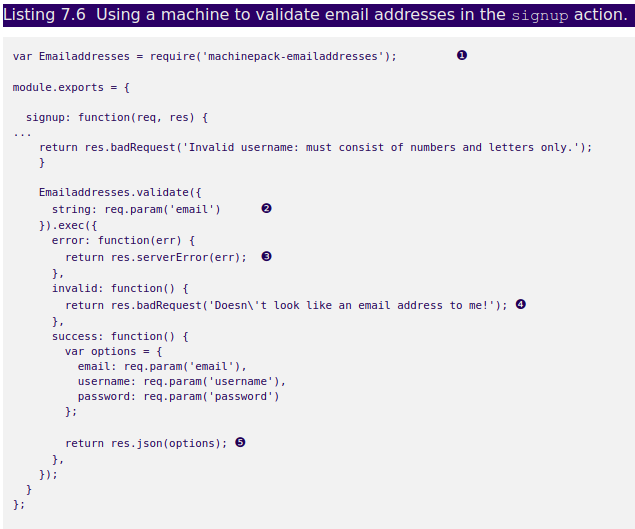
because the check status in **sigup page** be defined in **sigup.html,** not in **UserController.js**

**Does this code define in back-end??????????????**

Next, we need to assure the value for the email attribute has the proper syntax and we’ll use the validate machine to check the value.

### **7.3.4 Validating email addresses**

The **Emailaddresses.validate()** machine takes a string as input and determines whether that string uses valid email address syntax. If it does, the machine will execute the **success** exit, if it’s invalid the **invalid** exit will be executed and if there’s an error, the **error** exit will be executed. Let’s add the example code and then our own code to handle each exit. In Sublime, open **brushfire/api/controllers/UserController.js** and add the machine and custom handler code to the **signup** action of the **user** controller similar to listing 7.6.

❶ require the machinepack and assign it to the variable **Emailaddress**

❷ add the email address as input **via req.param('email')**

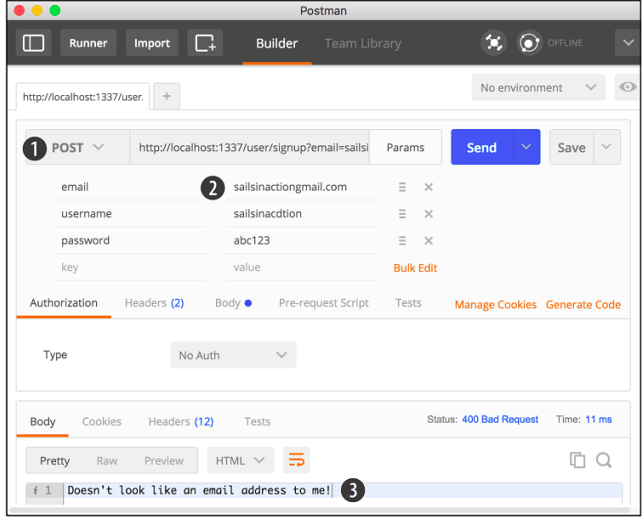
❸ return any errors via **res.serverError()**

❹ upon success respond with **res.json** returning the **user** dictionary

Next, we need to add the machinepack to the project. Head over to the terminal window and type:



Let’s see this in action. Restart Sails via **sails lift** and from within POSTMAN make another **POST** request to **/user/signup** similar to figure 7.11.



**Figure 7.11**  Using Postman, we’ll check the validity of the email attribute in the custom **signup** action. We’ll generate a ① **POST request to/user/signup** with ② an invalid email parameter, with the username and password parameters also provided. The signup action executed the invalid exit and responded with a **400** code and message.

The validate machine recognized that the email parameter used improper syntax and responded with an error message. With the email validation requirement fulfilled, let’s look at a transformation requirement-- encrypting the **password** (and yes we have a machine for that).