

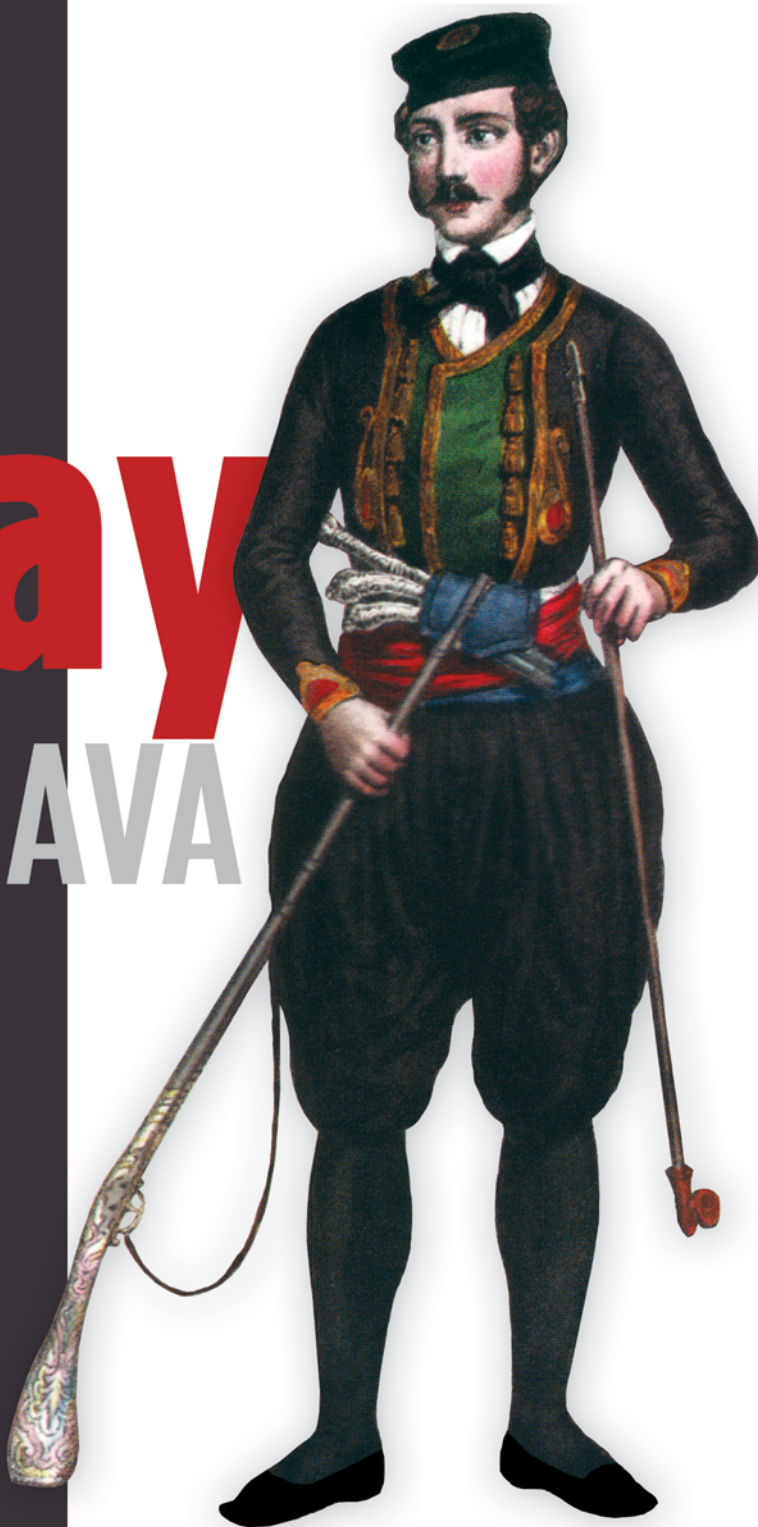
 MANNING

SAMPLE CHAPTERS

Play FOR JAVA

Nicolas Leroux
Sietse de Kaper

FOREWORD BY James Ward





Play for Java

by Nicolas Leroux

Sietse de Kaper

Chapters 3, 5, and 6

brief contents

PART 1 INTRODUCTION AND FIRST STEPS.....1

- 1 ■ An introduction to Play 3
- 2 ■ The parts of an application 21
- 3 ■ A basic CRUD application 37

PART 2 CORE FUNCTIONALITY.....57

- 4 ■ An enterprise app, Play-style 59
- 5 ■ Controllers—handling HTTP requests 72
- 6 ■ Handling user input 102
- 7 ■ Models and persistence 138
- 8 ■ Producing output with view templates 177

PART 3 ADVANCED TOPICS 205

- 9 ■ Asynchronous data 207
- 10 ■ Security 232
- 11 ■ Modules and deployment 249
- 12 ■ Testing your application 271

A basic CRUD application

This chapter covers

- An introduction to all major Play concepts
- Creating a small application

In the previous chapter, we introduced our example application: the paper clip warehouse management system. Now that we know what we're going to build, and have our IDE all set up, it's time to start coding.

In this chapter, we'll start implementing our proof-of-concept (POC) application. It will be a simple CRUD¹ application, with data stored in-memory rather than in a database. We'll evolve our POC application in later chapters, but this simple application will be our starting point.

We'll start by setting up a controller with some methods and linking some URLs to them.

¹ Create, Retrieve, Update, Delete

3.1 Adding a controller and actions

In chapter 1 we edited the `Application` class, changing the default output and adding custom operations. The `Application` class is an example of a *controller* class. Controller classes are used as a home for action methods, which are the entry points of your web application. Whenever an HTTP request reaches the Play server, it is evaluated against a collection of rules in order to determine what action method will handle this request. This is called *routing* the request, which is handled by something aptly named the *router*, which, in turn, is configured using the `conf/routes` file, as described in section 2.3. After the correct action method is selected, that method executes the logic necessary to come up with a *result* to return to the client in response to the request. This process is illustrated in figure 3.1.

Let's create a controller for the new warehouse application that we created in the previous chapter. The only requirement of a controller class is that it extends `play.mvc.Controller`. It does *not* have to be part of a specific package, although it is convention to put controllers in the `controllers` package. Let's create one for our product catalog. Because we're dealing with products, we'll call it `Products`. Create the `Products` class under the `controllers` package (that means the file is named `/app/controllers/Products.java`). Have this class extend `Controller`, like so:

```
package controllers;

import play.mvc.Controller;

public class Products extends Controller {

}
```

An empty controller doesn't do anything. The whole purpose of controllers is to provide action methods. An action method has to conform to the following requirements:

- It has to be *public*.
- It has to be *static*.
- It has to have a *return type* (a subclass) of *Result*.

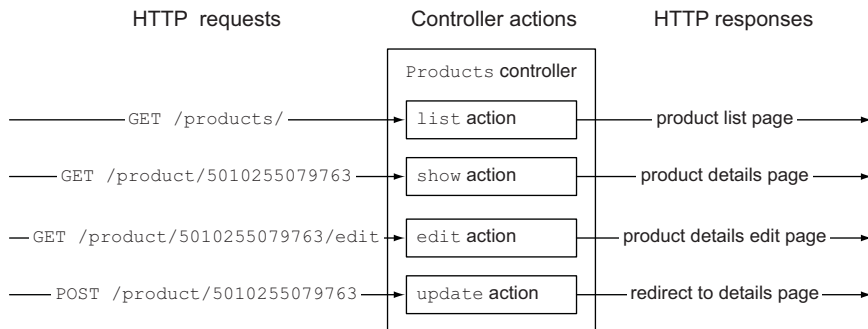


Figure 3.1 Requests routed to actions

Let's add some action methods to our controller. For our proof-of-concept application, we'll want to *list* products in our catalog, show *details* for an individual product, and *save* new and updated products. We'll add actions for these operations later, but for now we'll make them return a special type of result: `TODO`. A `TODO` result signifies that the method is yet to be implemented. Add the corresponding actions, as shown in the following listing.

Listing 3.1 Adding action methods

```
package controllers;

import play.mvc.Controller;
import play.mvc.Result;

public class Products extends Controller {

    public static Result list() {                                ← List all products
        return TODO;
    }

    public static Result newProduct() {                          ← Show a blank product form
        return TODO;
    }

    public static Result details(String ean) {                   ← Show a product edit form
        return TODO;
    }

    public static Result save() {                                ← Save a product
        return TODO;
    }
}
```

Now that we have some action methods, let's give them URLs so that we can reach them.

3.2 Mapping URLs to action methods using routes

In order to determine which action method will handle a given HTTP request, Play takes the properties of that request, such as its method, URL, and parameters, and does a lookup on a set of mappings called *routes*. Like we saw before, in section 2.3, routes are configured in the `routes` file in your application's `conf` directory. Add routes for our new operation, as shown in the following listing.

Listing 3.2 Adding routes for our product catalog

```
GET    /products/                controllers.Products.list()
GET    /products/new            controllers.Products.newProduct()
GET    /products/:ean          controllers.Products.details(ean: String)
POST   /products/                controllers.Products.save()
```

Now that we have some routes, let's try them out. Start your application if it's not already running, and point your browser at `http://localhost:9000/products/`. You should see the page shown in figure 3.2.

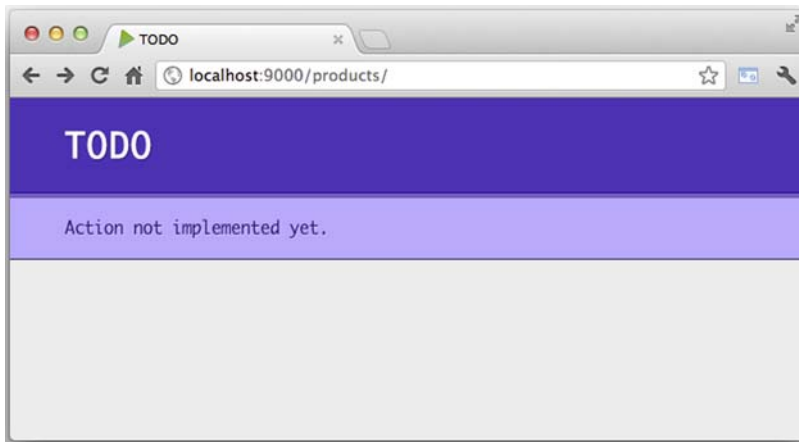


Figure 3.2 Play's TODO placeholder at /products

If you see the TODO placeholder page, that means the controller, action method, and route are all correctly set up. Time to add some functionality. The first step is adding a class that will model our products.

3.3 Adding a model and implementing functionality

In order to create a product catalog, we need a class to represent “a product” in our application. Such classes are called *model* classes, because they model real-world concepts.

3.3.1 Creating a model class

We'll keep our product model simple for now: an article number, name, and description will do. For the article number, we'll use an EAN code, which is a 13-digit internationally standardized code. Although the code consists of digits, we're not going to perform math on it, so we'll use `String` to represent the EAN code.

Create a class called `Product` under a new package called `models`. Again, there's nothing about Play that requires you to put model classes in the `models` package, but it's convention to do it that way. Add the properties we mentioned previously to your new class, and add a constructor that sets them on instantiation for convenience. In addition, add a default no-argument constructor, because we'll need that when we add database persistence later. The last thing we'll add is a `toString()` method, because that will make it easier for us to see what product object we have.

We end up with a class as shown in the following listing.

Listing 3.3 /app/models/Product.java

```
package models;

public class Product {

    public String ean;
    public String name;
```

```

public String description;

public Product() {}

public Product(String ean, String name, String description) {
    this.ean = ean;
    this.name = name;
    this.description = description;
}

public String toString() {
    return String.format("%s - %s", ean, name);
}
}

```

DON'T BE ALARMED BY PUBLIC PROPERTIES If you've been a Java developer for some time, you're probably surprised that we chose to use public properties. You're probably more used to making properties private and exposing them using getter and setter methods instead, creating a "Java Bean." Don't worry, we know what we're doing. For now, bear with us. Everything will be explained in detail in chapter 7.

We've created our first model class. In most cases, instances of these model classes are also stored in a database. To keep things simple, we'll fake this functionality for now by maintaining a static list of products. Now let's create some data.

3.4 Mocking some data

We'll mock data storage by using a static Set of Products on the Product model class, and we'll put some data in the class's static initializer, as shown in the following listing.

Listing 3.4 Adding some test data to /app/models/Product.java

```

import java.util.ArrayList;
import java.util.List;

public class Product {

    private static List<Product> products;

    static {
        products = new ArrayList<Product>();
        products.add(new Product("111111111111", "Paperclips 1",
            "Paperclips description 1"));
        products.add(new Product("222222222222", "Paperclips 2",
            "Paperclips description "));
        products.add(new Product("333333333333", "Paperclips 3",
            "Paperclips description 3"));
        products.add(new Product("444444444444", "Paperclips 4",
            "Paperclips description 4"));
        products.add(new Product("555555555555", "Paperclips 5",
            "Paperclips description 5"));
    }
    ...
}

```


NEVER DO THIS IN A REAL APPLICATION Although having a static property serve as a cache for data is convenient for this example, never do it in a real-world app. Because we'll only be using this `List` in dev-mode, which has only one thread running by default, we won't run into any serious trouble. But when you try this in any environment with multiple threads, or even multiple application instances, you'll run into all sorts of synchronization issues. Depending on the situation, either use Play's caching features, or use a database (see chapter 7).

Now that we have some data, let's also add some methods to manipulate the collection of `Products`. We'll need methods to retrieve the whole list, to find all products by EAN and (part of the) name, and to add and remove products. Add the methods shown in listing 3.5. We'll let their implementations speak for themselves.

Listing 3.5 Data access methods on the `Products` class

```
public class Product {
    ...
    public static List<Product> findAll() {
        return new ArrayList<Product>(products);
    }

    public static Product findByEan(String ean) {
        for (Product candidate : products) {
            if (candidate.ean.equals(ean)) {
                return candidate;
            }
        }
        return null;
    }

    public static List<Product> findByName(String term) {
        final List<Product> results = new ArrayList<Product>();
        for (Product candidate : products) {
            if (candidate.name.toLowerCase().contains(term.toLowerCase())) {
                results.add(candidate);
            }
        }

        return results;
    }

    public static boolean remove(Product product) {
        return products.remove(product);
    }

    public void save() {
        products.remove(findByEan(this.ean));
        products.add(this);
    }
}
```

Now that we have the plumbing for our products catalog, we can start implementing our action methods.

3.5 Implementing the list method

We'll start with the implementation for the `list` method. As we said before, an action method always returns a *result*. What that means is that it should return an object with a type that is a subclass of `play.mvc.Result`. Objects of that type can tell Play all that it needs to construct an HTTP response.

An HTTP response consists of a status code, a set of headers, and a body. The status codes indicate whether a result was successful and what the problem is if it wasn't. Play's Controller class has a lot of methods to generate these result objects. Let's go ahead and replace our `TODO` result with a code 200 result, which means "OK." To do this, use the `ok()` method to obtain a new OK result, like this:

```
public static Result list() {
    return ok();
}
```

If you were to try this out in a browser, you'd get an empty page. If you were to check the HTTP response,² you'd see that the response status code has changed from 501 - Not Implemented to 200 - OK. The reason why our browser shows an empty page is because our response has no *body* yet. That makes sense, because we didn't put one in yet. To generate our response body, we want to generate an HTML page. For this, we'll want to write a template file.

3.5.1 The list template

As we saw in the previous chapters, a Play template is a file containing some HTML and Scala code that Play will compile into a class that we can use to render an HTML page. Templates go in your application's `views` directory and, to keep things clean and separated by functionality, we'll create a `products` directory there. Next, create a file called `list.scala.html`, and add to it the contents shown in the following listing.

Listing 3.6 /app/views/products/list.scala.html

```
@(products: List[Product])
@main("Products catalogue") {
    <h2>All products</h2>

    <table class="table table-striped">
      <thead>
        <tr>
          <th>EAN</th>
          <th>Name</th>
          <th>Description</th>
        </tr>
      </thead>
      <tbody>
        @for(product <- products) {
```

² Your browser probably has tools to do that.

```

    <tr>
      <td><a href="@routes.Products.details(product.ean)">
        @product.ean
      </a></td>
      <td><a href="@routes.Products.details(product.ean)">
        @product.name</a></td>
    </tr>
  </tbody>
</table>
}

```

When rendered (we'll get to how to do that in a moment), this template will produce a page as seen in figure 3.3.

Don't worry, we'll make it look better in a bit. But first, without going into too much detail (see chapter 8 for more detail on templates), let's see what happens in the template.

HOW THE TEMPLATE WORKS

The first line of the list template is the *parameter list*:

```
@(products: List[Product])
```

With the parameter list, we define which parameters this template accepts. Every entry in the parameter list consists of a name, followed by a colon and the type of the parameter. In our example, we have one parameter called `name`, of type `List<Product>`,³ to represent the list of products we want to render. This parameter list will be part of the method definition for this template's `render` method, which is how Play achieves type safety for its templates.

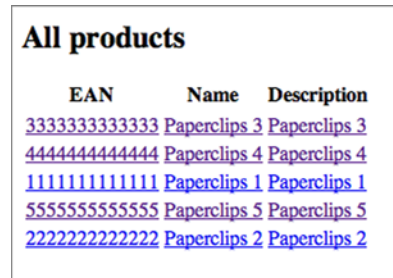
Let's take a look at the next line of code, which starts a block of code:

```

@main("Products catalogue") {
  ...
}

```

With this code we call another template, the one called `main`. This is the template at `/app/views/main.scala.html`, which Play created for us when we created the application. It contains some boilerplate HTML that we'll wrap around all of our pages, so we don't have to worry about that any more. The code we write in the block will end up in the `<body>` tag of our rendered HTML page. This is how you can compose templates in Play, and we'll see more of this in later chapters.



EAN	Name	Description
3333333333333	Paperclips 3	Paperclips 3
4444444444444	Paperclips 4	Paperclips 4
1111111111111	Paperclips 1	Paperclips 1
5555555555555	Paperclips 5	Paperclips 5
2222222222222	Paperclips 2	Paperclips 2

Figure 3.3 Our products listing

³ In Scala syntax, generic type arguments are indicated using square brackets instead of angle brackets as in Java.

The body of our code block is mainly HTML, which will be included in the rendered page verbatim. There's one bit of template code left—the bit that iterates over our products list:

```
@for(product <- products) {
  ...
}
```

This bit of code is comparable to a regular Java for-each loop: it iterates over a collection and repeats the code it wraps for every element in it, assigning the current element to a variable. In our example, it generates a pair of `<td>` elements for every `Product` in our products list. Listing 3.7 shows the full loop as a reminder.

Listing 3.7 for loop generating the product descriptions

```
@(products: List[Product])
<table class="table table-striped">
  <thead>
    <tr>
      <th>EAN</th>
      <th>Name</th>
      <th>Description</th>
    </tr>
  </thead>
  <tbody>
    @for(product <- products) {
      <tr>
        <td><a href="@routes.Products.details(product.ean)">
          @product.ean
        </a></td>
        <td><a href="@routes.Products.details(product.ean)">
@product.name</a></td>
        <td><a href="@routes.Products.details(product.ean)">
@product.name</a></td>
      </tr>
    }
  </tbody>
</table>
```

The pieces of code in the loop's body that start with an `@` are Scala *expressions*; the code that follows the `@` is evaluated, and the result is included in the output. In this case, we use it to print out properties of `product` and generate links to our action methods based on our routing configuration. For everything about routing, see chapter 5. We'll render our template soon, but first let's add some style.

ADDING BOOTSTRAP

During our examples, we focus more on functionality than styling; this is a book about Play, after all, and not about web design. But there is a way to make things look nicer with little effort: *Bootstrap*, by Twitter.

Bootstrap provides some CSS and image files that make HTML look good and maybe adding an HTML class here and there. It's easy to use Bootstrap in your Play applications. Here's how.

First, download the latest version of Bootstrap from the website at <http://getbootstrap.com>. Extract the contents of the zip file to a bootstrap directory under your application's public directory. This will make the files available from your application.

Next, we need to include the Bootstrap CSS in our templates. Because we're going to need it on all of our pages, the main template is the best place to do that. Open the file `/app/views/main.scala.html`, and add the following line below the existing `<title>` element, inside the `<head>` element:

```
<link href="@routes.Assets.at("bootstrap/css/bootstrap.min.css")"
      rel="stylesheet" media="screen">
```

This will allow your pages to be styled by Bootstrap, and, from now on, we'll use Bootstrap to make all of our examples look nicer. If you want to learn more about Bootstrap, check out the website at <http://getbootstrap.com>.

Now that we have our templates ready, let's see how to render them.

RENDERING THE TEMPLATE

Now that we have a template, all that's left for us to do is to gather a list of products and render the template in our `list` action method. The following listing shows how.

Listing 3.8 Rendering the `list` template

```
...
import views.html.products.list;

public class Products extends Controller {

    public static Result list() {
        List<Product> products = Product.findAll();
        return ok(list.render(products));
    }

    ...
}
```

As you can tell from the import in this example, the template `/views/products/list.scala.html` results in a class called `views.html.products.list`. This `list` class has a static method called `render`, which, as your IDE can tell you, takes one parameter of type `List<Product>` and returns an object of type `Html`. The parameter is the one we defined at the top of our template, whereas the return type is determined by the `.html` extension of the template filename.

The `render` method on the template results in an HTML page, which we want to return to the client in the body HTTP response. To do this, we wrap it in a `Result` object by passing it to the `ok` method.

All products		
EAN	Name	Description
3333333333333	Paperclips 3	Paperclips 3
5555555555555	Paperclips 5	Paperclips 5
2222222222222	Paperclips 2	Paperclips 2
1111111111111	Paperclips 1	Paperclips 1
4444444444444	Paperclips 4	Paperclips 4
<input type="button" value="+ New product"/>		

Figure 3.4 Our products listing

Time to try out our code. Navigate to `http://localhost:9000/products/`, and you should see a list as in figure 3.4.

Now that we can see our list of products, let's continue implementing features.

3.6 Adding the product form

A static product catalog isn't useful. We want to be able to add products to the list. We'll need a form for that, so create a new template called `details.scala.html` at `/app/views/products`. We'll create a form that will work both for creating new products and editing existing ones. The template is shown in the following listing.

Listing 3.9 Product form `/app/views/products/details.scala.html`

```
@(productForm: Form[Product])
@import helper._
@import helper.twitterBootstrap._

@main("Product form") {
  <h1>Product form</h1>
  @helper.form(action = routes.Products.save()) {
    <fieldset>
    <legend>Product (@productForm("name").valueOr("New"))</legend>
    @helper.inputText(productForm("ean"), '_label -> "EAN")
    @helper.inputText(productForm("name"), '_label -> "Name")
    @helper.textarea(productForm("description"), '_label -> "Description")
    </fieldset>
    <input type="submit" class="btn btn-primary" value="Save">
    <a class="btn" href="@routes.Products.index()">Cancel</a>
  }
}
```

As you can see in the first line of the template, this template takes a `Form<Product>` parameter, like our list template took a `List<Product>` parameter. But what's this `Form` class? `Form` is what Play uses to represent HTML forms. It represents name/value

pairs that can be used to build an HTML form, but it also has features for input validation, error reporting, and data binding. Data binding is what makes it possible to convert between HTTP (form) parameters and Java objects and vice versa.

3.6.1 Constructing the form object

Let's see how these forms work. First, we need to create one to pass to the template. That's as easy as calling the `play.data.Form.form()` method in our action method. The `form` method takes a class as a parameter, to tell it what kind of object the form is for. Because a product form is always the same, and we're going to use it in a few places in the `Products` controller, we might as well create a constant for it in the class, like so:

```
private static final Form<Product> productForm = Form
    .form(Product.class);
```

Now that we have an empty form, it's easy to pass it to the template. Implement the `newProduct` action method as shown here:

```
public static Result newProduct(){
    return ok(details.render(productForm));
}
```

With this action method implemented, you can see the form at `http://localhost:9000/products/new`. It should look like figure 3.5.

Let's see how to create the form.

3.6.2 Rendering the HTML form

Let's see how we make an HTML form from our `Form` object. At the top of the template, you can see how we import two helpers:

```
@import helper._
@import helper.twitterBootstrap._
```

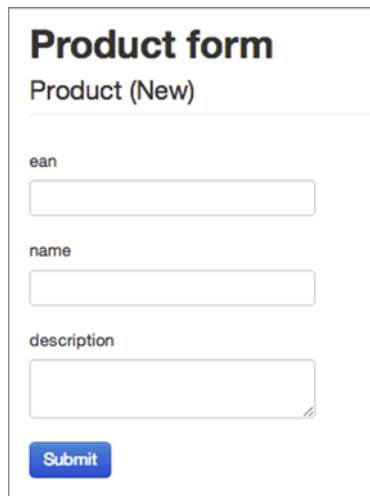
A screenshot of a web form titled "Product form" with a subtitle "Product (New)". The form contains three input fields: "ean", "name", and "description". Below the fields is a blue "Submit" button. The form is styled with a light gray border and a white background.

Figure 3.5 The product form

These helpers are there to help us generate HTML. The first one imports generic HTML helpers, and the second one makes the generated HTML fit the Twitter Bootstrap layout. We first use one of these helpers when we start the form:

```
@helper.form(action = routes.Products.save()) {
  ...
}
```

The form helper generates an HTML `<form>` element. The `action` parameter tells it where the form should be submitted to. In our case, that's the `save` method on our `Products` controller. Play will turn this into an `action` attribute with the correct URL value for us.

A form is not much use without any fields. Let's see how those are constructed.

3.6.3 Rendering input fields

Our form contains a single fieldset, which is created using regular HTML. The value for the fieldset's legend element is interesting enough to take a closer look at. It starts off with regular text, "Product," but then we use the form object to construct the rest of the value:

```
@productForm("name").valueOr("New")
```

Here, we request the form field name by calling `productForm("name")`.⁴ This object is of type `Form.Field`, and it represents the form field for the `name` property of the form. To get the value, we could call the `value` method on the field. But because we don't know *if* there is a value for this field, we use the `valueOr` method, which allows us to specify a default value to use in case the field has no value. This means we don't need to check for a value manually, saving us from a lot of messy, verbose code in our template.

The next few lines in our template render input elements—one for each property of our `Product` class:

```
@helper.inputText(productForm("ean"))
@helper.inputText(productForm("name"))
@helper.textarea(productForm("description"))
```

When our template is rendered, these lines are rendered as shown in the following listing.

Listing 3.10 Rendered input elements

```
<div class="clearfix" id="ean_field">
  <label for="ean">ean</label>
  <div class="input">
    <input type="text" id="ean" name="ean" value="" >
      <span class="help-inline"></span>
      <span class="help-block"></span>
  </div>
</div>
```

⁴ `productForm("name")` is short for `productForm.field("name")`.


```

<div class="clearfix" id="name_field">
  <label for="name">name</label>
  <div class="input">

    <input type="text" id="name" name="name" value="" >

    <span class="help-inline"></span>
    <span class="help-block"></span>
  </div>
</div>

<div class="clearfix" id="description_field">
  <label for="description">description</label>
  <div class="input">

    <textarea id="description" name="description" ></textarea>

    <span class="help-inline"></span>
    <span class="help-block"></span>
  </div>
</div>

```

With three simple lines of code, we've generated all that HTML! And because of the Bootstrap helper, it doesn't look bad, either.

The final line of our template's form adds a regular HTML Submit button, and with that, our form is ready. When you try it out, the form will submit to our unimplemented save method, so it doesn't do much yet. Let's take care of that now.

3.7 Handling the form submission

When you submit the product form in the browser, the form gets submitted to the URL specified in the action attribute of the HTML `<form>` element, which, in our case, ends up at our application's `Products.save` action method. It's now up to us to transform those HTML form parameters into a `Product` instance, and add it to the product catalog. Luckily, Play has some tools to make this job easy.

When we created the `Form` object in the previous section, we used it to create an HTML form based on the `Product` class. But Play's Forms work the other way around, too. This reverse process is called *binding*.

Play can bind a set of name/value combinations, such as a `Map`, to a class that has properties with the same names. In this case, we don't want to bind a map, but we do want to bind values from the request. Although we could obtain a `Map` of the name/value pairs from the HTTP request, this situation is so common that the `Form` class has a method to do this: `bindFromRequest`. This will return a new `Form` object, with the values populated from the request parameters. To obtain a `Product` from our form submission and add it to the catalog, we can write the following code:

Listing 3.11 Product binding

```

public class Products extends Controller {
  ...
  public static Result save() {
    Form<Product> boundForm = productForm.bindFromRequest();
    Product product = boundForm.get();
  }
}

```

```

        product.save();
        return ok(String.format("Saved product %s", product));
    }
}

```

When you try out the form now, you'll get a simple text message informing you of the successful addition of the product. If you then check the catalog listing we made in section 3.5, you can verify that it worked.

But our current implementation isn't particularly nice. The user is free to omit the EAN code and product name, for example; at the moment this will work, but it's not something that we want. Also, the text message reporting the result isn't great. It would be a lot nicer to rerender the form with an error message on failure, and show the product listing with a success message if everything was correct.

First, let's tell Play that the `ean` and `name` fields are required. We'll leave the `description` optional.

We can make those fields required by using an annotation, `play.data.validation.Constraints.Required`. Play will check for those annotations and report errors accordingly. The following listing shows the constraint added.

Listing 3.12 Adding a pattern constraint

```

...
import play.data.validation.Constraints;

public class Product {
    ...
    @Constraints.Required
    public String ean;
    @Constraints.Required
    public String name;
    public String description;
    ...
}

```

What we need to do now is perform the validation in our controller and show an error or success message accordingly. The following listing shows a different version of `save()` that has that functionality.

Listing 3.13 A better save implementation

```

public static Result save() {
    Form<Product> boundForm = productForm.bindFromRequest();
    if(boundForm.hasErrors()) {
        flash("error", "Please correct the form below.");
        return badRequest(details.render(boundForm));
    }

    Product product = boundForm.get();
    product.save();
    flash("success",
        String.format("Successfully added product %s", product));

    return redirect(routes.Products.list());
}

```

In this version of our implementation, we use the *validation* functionality of Play's forms. On the second line of our method, we ask the `Form` if there are any errors, and, if there are, we add an error message and rerender the page. If there are no errors, we add a success message and redirect to the products list.

The error and success messages aren't visible yet. We've added them to something called the *flash scope*. Flash scope is a place where we can store variables between requests. Everything in flash scope is there until the following request, at which point it's deleted. It's ideal for success and error messages like this, but we still need to render these messages.

Because messages like these are useful throughout the application, let's add them to the main template, because that's what every page extends. That way, every page will automatically display any messages we put in flash scope. Add the lines shown in listing 3.14 to the start of the `<body>` element in `app/views/main.scala.html`.

Listing 3.14 Displaying flash success and error messages

```
@if(flash.containsKey("success")){
  <div class="alert alert-success">
    @flash.get("success")
  </div>
}

@if(flash.containsKey("error")){
  <div class="alert alert-error">
    @flash.get("error")
  </div>
}
```

Now try out the form. Load the form at `http://localhost:9000/products/new`, and try to submit the form while leaving the EAN field blank. You should see a page as in figure 3.6.

A lot more is possible using form validation, but for now this is enough. You can learn all about forms and validation in chapter 6.

Now that we have our form working, we can use it to edit existing products. To do this, we need to implement the `details` method as in the following listing.

Listing 3.15 Implementing the details method

```
public class Products extends Controller {
  ...
  public static Result details(String ean) {
    final Product product = Product.findByEan(ean);
    if (product == null) {
      return notFound(String.format("Product %s does not exist.", ean));
    }

    Form<Product> filledForm = productForm.fill(product);
    return ok(details.render(filledForm));
  }
  ...
}
```

Please correct the form below.

Product form

Product (Paperclips 6)

ean

This field is required

Required

name

Required

description

Submit

Figure 3.6 Validation errors in our form

As you can see, it doesn't take a whole lot of code to turn a "new product" form into a "product edit" form. This method takes an EAN code as a parameter from the URL, as we defined in the `routes` file in section 3.2. We then look up the product based on the EAN. If there's no product with that EAN, we return a 404 - Not Found error.

If we *do* find a product, we create a new `Form` object, prefilled with the data from the product we found. We use the `fill` method on our existing empty form object for that. It's important to note that this does *not* fill in the existing form, but it creates a new form object *based on* the existing form.

Once we have the form, all that remains is to render the template and return the "ok" result, as in `newProduct`.

This action method is complete, and now the links in the product listing all work correctly. There's one more step left to complete our CRUD functionality: we need to implement delete functionality.

3.8 Adding a delete button

Let's start by adding a `delete()` method to our `Products` controller. The functionality is largely similar to the `details()` method; we take an EAN parameter, search for a corresponding `Product`, and return a 404 error if we can't find one. Once we have the `Product`, we delete it and redirect back to the `list()` method. Listing 3.16 shows the method.

Listing 3.16 The delete() action method

```
public static Result delete(String ean) {
    final Product product = Product.findByEan(ean);
    if(product == null) {
        return notFound(String.format("Product %s does not exists.", ean));
    }
    Product.remove(product);
    return redirect(routes.Products.list());
}
```

Now we need to add a route for this method in order to make it callable from the web. Because this is a method that changes something, we can't make this a GET operation. With a RESTful interface, we have to make this a DELETE operation. To do so, we'll use a bit of JavaScript to send a DELETE request, because we can't use a simple link (that would issue a GET operation). This is simple; the following code instructs your browser to issue a DELETE request to the server:

```
<script>
    function del(urlToDelete) {
        $.ajax({
            url: urlToDelete,
            type: 'DELETE',
            success: function(results) {
                // Refresh the page
                location.reload();
            }
        });
    }
</script>
```

Now, let's change our route to add a DELETE route, as shown here:

```
DELETE /products/:ean controllers.Products.delete(ean: String)
```

It's now time to add the user interface for our delete operation: the *Delete* button. Because the delete operation requires an HTTP DELETE call, we add a simple link that calls our JavaScript del method, which in turn calls the server and refreshes the page. We add a simple link with an onclick action handler that calls our JavaScript, and we're done. The following listing shows the updated list template.

Listing 3.17 Updated template—app/views/products/list.scala.html

```
@(products: List[Product])
@main("Products catalogue") {

    <h2>All products</h2>

    <script>
        function del(urlToDelete) {
            $.ajax({
                url: urlToDelete,
                type: 'DELETE',
                success: function(results) {
```

```

        // Refresh the page
        location.reload();
    }
    });
}
</script>

<table class="table table-striped">
  <thead>
    <tr>
      <th>EAN</th>
      <th>Name</th>
      <th>Description</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    @for(product <- products) {
      <tr>
        <td><a href="@routes.Products.details(product.ean)">
          @product.ean
        </a></td>
        <td><a href="@routes.Products.details(product.ean)">
@product.name</a></td>
        <td><a href="@routes.Products.details(product.ean)">
@product.name</a></td>
        <td>
          <a href="@routes.Products.details(product.ean)">
<i class="icon icon-pencil"></i></a>
          <a onclick="del('@routes.Products.delete(product.ean)')">
<i class="icon icon-trash"></i></a>
        </td>
      </tr>
    }
  </tbody>
</table>

<a href="@routes.Products.newProduct()" class="btn">
  <i class="icon-plus"></i> New product</a>
}

```

← The link calls our JavaScript del method, which in turn issues a request to the server

Go ahead and test it out. You should be able to delete products from the list page now.

With the delete functionality added, the functionality for our proof-of-concept application is now complete.

3.9 Summary

In this chapter, we implemented a simple proof-of-concept application. We added all the CRUD functionality, with a datastore in memory. We started with a *controller* with some basic *action methods* and linked them to URLs by setting up Play's *routing* system. We then introduced some *view templates* and added some *forms* with *validation*. Finally, we added the delete functionality by adding a DELETE action and a corresponding form.

This chapter was a quick introduction to all the core concepts of Play. All the topics in this chapter will be explained in detail in later chapters, but now you have the general idea of the most important concepts. You've also had a taste of what it means that Play is type-safe. If you followed along with the exercises, and you made an occasional mistake, you've probably also seen how soon mistakes are spotted because of the type safety, and how useful Play's error messages are when a problem is found.

In the next chapter, we're going to see how Play 2 fits in an enterprise environment and the enterprise challenges Play 2 is trying to solve.

5

Controllers— handling HTTP requests

This chapter covers

- How to use controllers
- Using action methods and results
- Using routing to wire URLs to action methods
- Composing actions with interceptors
- Using the different scopes

In this chapter we'll explain in detail one of the key concepts of the Play framework MVC paradigm: the controller. We'll take a closer look at our warehouse web application and at the same time explain how interaction with a web client works in Play.

We'll start by explaining controllers, and from there we'll examine action methods and how we can return results to web clients. We'll then see how to use routes to link HTTP requests to a controller's action method. After that, we'll look at what interceptors are and talk about what scopes are available in Play. All of these concepts are important when processing and responding to client requests.

Let's see how we can accept and process a request from a client. First, we'll introduce the concepts of *controllers* and talk some more about *action methods*.

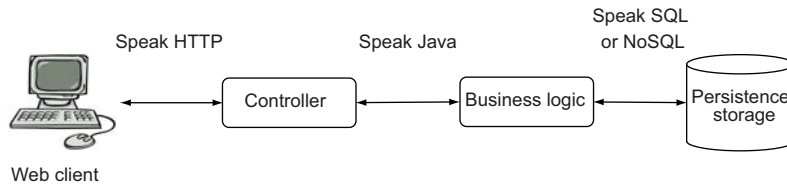


Figure 5.1 Role of the controller

5.1 Controllers and action methods

Business data is often stored in a relational database. That means that if you wanted to add a new product to the catalog, you'd have to write some SQL. But that wouldn't be very practical, would it? This is where web applications come to the rescue. Using a web application, the warehouse manager can interact with the database stock level. This is an easier and cheaper solution than learning SQL or having an on-site SQL expert.

But there's still a problem: the web browser can't directly access the data because it doesn't speak SQL. The web browser speaks HTTP. Moreover, the model layer is usually fine-grained, whereas the user usually wants to execute a series of actions in one go. For example, when you add a product, you want to make sure the data describes a valid product and that the product wasn't already created.

This is precisely the role the *controller* plays. The controller connects HTTP requests to the server business logic. It acts as glue between domain model objects and transport layer events. A controller exposes some application functionality at a given address or URI (unique resource identifier). In fact, it exposes the application business logic at a given address: it's our web application's public API. Like the HTTP interface, controllers are procedural and request/response oriented.

Figure 5.1 illustrates the role of the controller in a typical web application.

A controller is one of the central points in Play, as in any MVC framework. It is also the application entry point for you as a developer. As soon as a client (for example, a web browser) issues a request, Play will accept this request and delegate the processing of it to a controller. This is usually where your code comes into action. Figure 5.2 illustrates this lifecycle.

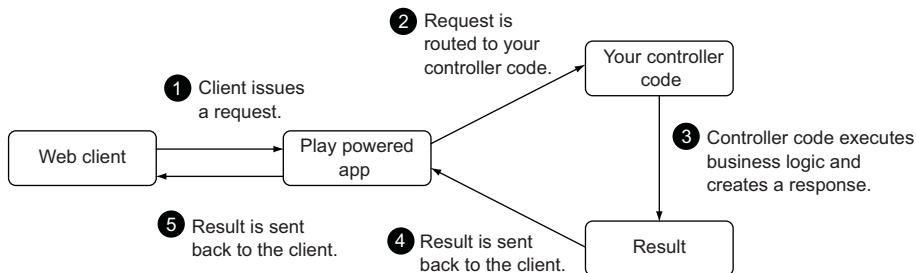


Figure 5.2 Controller lifecycle

While the controller is a central point in any Play application, its actual code resides in its action methods.

5.1.1 *Action methods*

In Play, a controller is an aggregation of action methods. An action method is a static method that takes some parameters as input and always returns an instance of (a subclass of) the `Result` class. In other words, an action method always takes the following form:

```
public static play.mvc.Result methodName(params...)
```

Note that action methods are static. This is because controllers are effectively singletons, which fits the HTTP approach. Because HTTP doesn't have the concept of an object but is mainly procedural, we shouldn't try to map those object concepts in our controller. You should really think of your action methods as the entry point from HTTP to your application. You could compare them to the static `main()` method of a Java program.

Since an action method mainly serves as an entry point, you shouldn't put too much business logic into your controllers and action methods. Rather, you should do what is necessary to translate an incoming HTTP request to data your business logic understands, hand it off to that business logic, and translate the result into an HTTP response.

When coding static methods, thread safety is usually a concern, but Play will make sure all action methods are thread-safe. You will see that making action methods static has little impact on the way you write code.

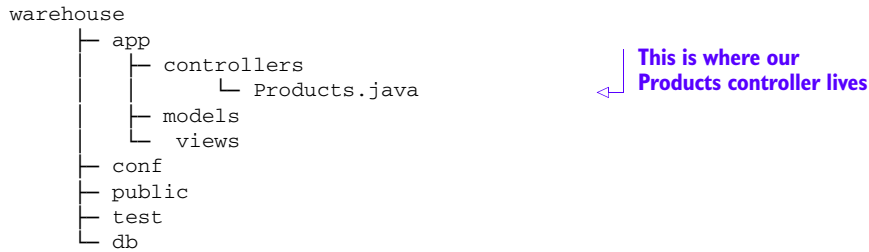
For now, imagine that Play queues the incoming requests for a controller in order to process them as fast as possible. Netty and Akka are used under the hood to dispatch and process users' requests as fast as possible in a thread-safe way. But the cool thing about Play is that all this complexity is handled for you, so don't have to worry about it.

We've already created our first controller, `Products`, in chapter 3. Let's examine it.

5.1.2 *Examining our controller*

In chapter 3, we created a controller class called `Products`. We put it in the `/app/controllers/` folder, which is the default location for all controllers in a Play application. You can change these defaults if you want to, but you should rarely need to do so.

Let's revisit the contents of the controller. In your favorite text editor or IDE, open the file called `Products.java` in the `app/controllers` directory, as shown in listing 5.1.

Listing 5.1 Project directory overview

We'll first pick apart the class definition. The following listing shows a reminder.

Listing 5.2 Products class definition

```

...
import play.mvc.Controller;
...

public class Products extends Controller {
    ...
}
  
```

The class definition tells us that we're extending the Controller class from Play. That's the only prerequisite for a controller; there is nothing else that makes a controller class "special." Let's move on to the action methods.

The first action method of this controller, `list()`, displays the product items in stock at the warehouse. The following listing shows a reminder.

Listing 5.3 The `list()` action method

```

public static Result list() {
    List<Product> products = Product.findAll();
    return ok(products.list.render(products));
}
  
```

In this example, the `ok` method constructs a 200 OK HTTP response containing a response body that is the result of rendering the `list` template. Note how little code this method contains. All it does is delegate data lookup to the model layer (the `models.Product` class) and data presentation to the view layer (the `list` template).

As you may recall from chapter 3, we can access the `list` method by requesting the `/products/` URL from our application. The reason why this works is because we've also configured a *route*. We'll learn everything about routing in section 5.3, but first, let's learn a little more about results.

5.2 Returning results from action methods

Besides the `list()` action methods, our `Products` controller contains several more action methods. Listing 5.4 shows an overview of their definitions.

Listing 5.4 The action methods in Products

```
public class Products extends Controller {  
    public static Result list() {  
        ...  
    }  
  
    public static Result newProduct() {  
        ...  
    }  
  
    public static Result details(String ean ) {  
        ...  
    }  
  
    public static Result save() {  
        ...  
    }  
}
```

An action method is a Java method that processes the request parameters and produces a result to be sent to the client. The action method is where the response is processed. Each action method returns a result, represented by a `play.mvc.Result` value, which represents an HTTP response.

5.2.1 Results

Let's take a closer look at what our action method returns: a `Result` object. A result is a response to a client request. Since we're creating a web application, it's always an HTTP response. It can be an OK with some text body, an error with an HTML error message, a redirect to another page, a 404 page, and so on.

The `Controller` class contains several static methods that generate `Results`. These methods all correspond to an HTTP status code, and they wrap an object that represents the body for the request. An example of this is the `ok()` method that we used to create the 200 - OK response code. We supplied it with HTML contents from the `list` template, thus generating the `Result` that we returned. Try to find the best response code for any situation.

For example, if a user enters an unknown EAN number as a parameter for the `show` method, we could return:

```
return badRequest("Incorrect EAN: " + ean)
```

This code returns an HTTP error code 400 with the text "Incorrect EAN: x" as content.

A more appropriate response would be to answer that we didn't find the product:

```
return notFound("No product with EAN " + ean);
```

This code returns an HTTP 404 error code with the text "No product with EAN: x" as content.

5.2.2 Redirect result

Another useful `Result` object is the `Redirect` result object. As its name suggests, it redirects to another action method. For example, using the `Redirect` result, we can redirect the user from the `index` method to the `list` method, allowing our users to see a list of products on the main page.

```
public static Result index() {
    return redirect(routes.Products.list());
}
```

Redirect to the product
list action method

The `routes` object is a class that's generated by Play 2 based on your routes file. The class only contains static members that allow you to access your controller methods. For now, remember that the `Redirect` result redirects from one action method to another action method. We'll go into more detail in the next section.

THE TODO RESULT There is also a useful `Result` object called `TODO`. As you might have guessed, this `Result` indicates that the action method has not been implemented yet. This is useful when you're developing your application and don't have your action method implementation finished, but still need to return a result. For example, if we have the following action method but our implementation is not ready yet, we can return a `TODO` result:

```
public static Result items() {
    return TODO;
}
```

When an action method returns a `TODO` result, the client will receive a 501 - Not Implemented HTTP response.

5.2.3 Using results

We are now able to control what status codes we return. But what about the response body? And how does Play know how to set the correct content-type header?

All result methods let you pass a `Content` object as a parameter. The type of content that object contains tells Play what kind of data it is. As it turns out, templates in Play also return `Content`.

In chapter 3, we built our first template, `products/list.scala.html`. The following listing reminds you of what it does.

Listing 5.5 Displaying our stock items

```
@(products: List[Product])
@main("Products catalogue") {

    <h2>All products</h2>

    <table class="table table-striped">
    <thead>
    <tr>
        <th>EAN</th>
        <th>Name</th>
        <th>Description</th>
    </tr>
```

```

</thead>
<tbody>
  @for(product <- products) {
    <tr>
      <td><a href="@routes.Products.details(product.ean)">
        @product.ean
      </a></td>
      <td><a href="@routes.Products.details(product.ean)">
        @product.name</a></td>
      <td><a href="@routes.Products.details(product.ean)">
        @product.name</a></td>
    </tr>
  }
</tbody>
</table>

<a href="@routes.Products.newProduct()" class="btn">
  <i class="icon-plus"></i> New product</a>
}

```

This template prints a list of products and their descriptions. What you need to know is that Play automatically compiles the template when it sees it. The result of this compilation step is a Java class with a `render()` method. By convention, it's compiled into a class with the name `views.html.{name of the template}`. In this case, since we created a template called `list` under the `views/products/` directory, the corresponding class is `views.html.products.list`.

When we render the template, the generated classes contain a render method. In Play, all templates are type-safe, which means the `render()` method expects a certain number of parameters of the correct types. In this case, we have to pass the `products` list as a parameter. All this will be explained in more detail in chapter 8.

What matters to us at the moment is to understand how we send HTML from our controller. In chapter 3, we rendered the template and immediately passed it the `ok()` method, like so:

```

public static Result list() {
  ...
  return ok(list.render(products));
}

```

This is the most concise way to render a template and return a `Result` that wraps the rendered content, but we could have also used an intermediary variable, like so:

```

public static Result list() {
  ...
  Html renderedTemplate = list.render(products);
  return ok(renderedTemplate);
}

```

As you can see, the template rendering doesn't simply return a string containing HTML; it actually returns an object of type `play.api.templates.Html`. That class is a subtype of `play.mvc.Content`, which is a type that can be used as a parameter for the `ok()` method (and all result methods, for that matter).

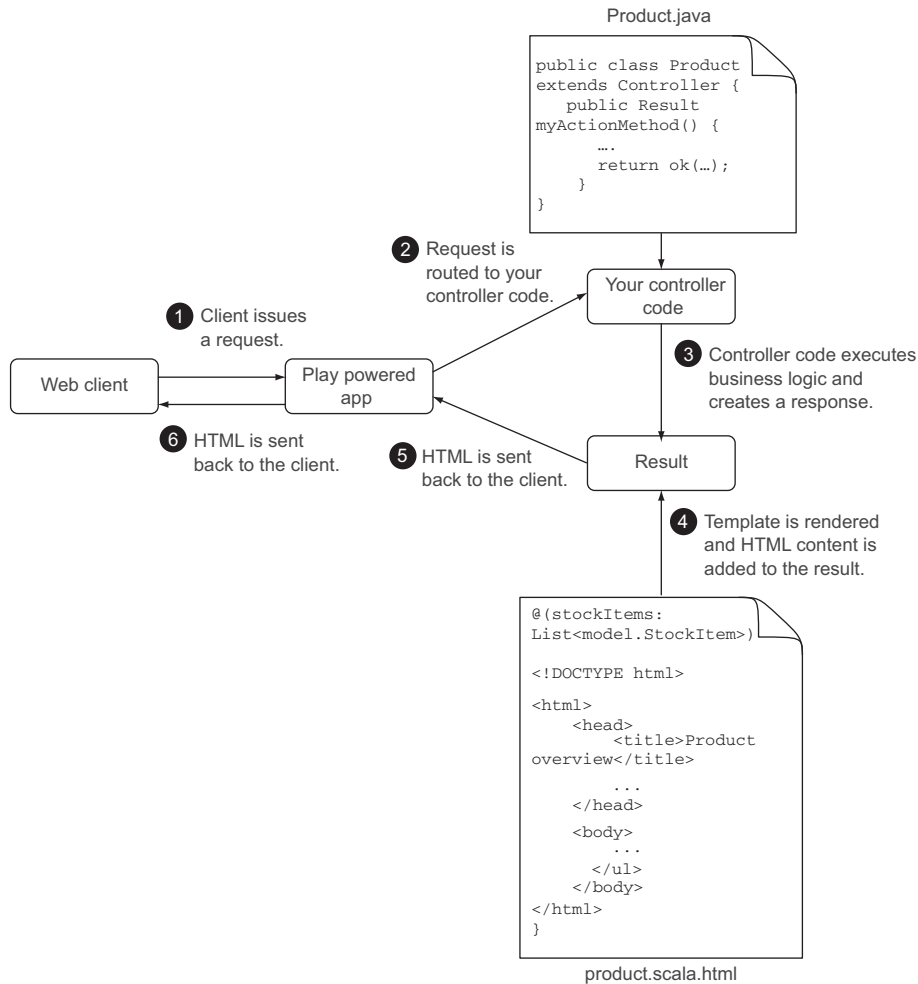


Figure 5.3 Detailed view of the controller lifecycle

You should now have an idea of how templates and results fit into the bigger picture of a Play application. Figure 5.3 shows the place of the controllers and products in the lifecycle diagram.

We've displayed some HTML without much effort; we only changed the provided `Result` object in our action method with some `Content`. Let's do something extra. Let's serve the client with the type of result they're asking for. If the web client asks for text, we'll return some text; if it asks for HTML, we'll return some HTML. This is easy to do with Play:

```
if (request().accept("text/plain")) {
  return ok(StringUtils.join(products, "\n"));
}
return ok(list.render(products));
```

request() method accesses the current request

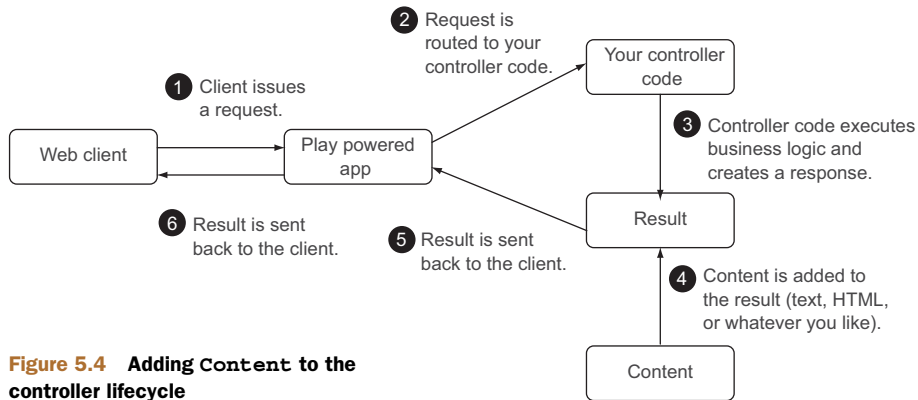


Figure 5.4 Adding Content to the controller lifecycle

We are now able to serve content according to the client’s wishes. We added our `Content` object to the diagram to explain the complete request/response lifecycle in Play, as shown in figure 5.4.

We are now able to respond to a request with some content. Let’s take a look at how our code is actually called in response to a request.

5.3 Using routing to wire URLs to action methods

We now know how we can render some content and how to execute business logic. But how do we link the controller method to be executed to the URL that the client invokes?

5.3.1 Translating HTTP to Java code

Remember, our client only speaks HTTP. But the code that we’re writing is in Java. We therefore need to translate the HTTP “language” to the Java language. This is the role of the router: translating each incoming HTTP request to an action method call. This way, it exposes the controller’s action methods to the client. An HTTP request can be seen as an event, from Play’s point of view. The router’s role is to coordinate a reaction to such an event (figure 5.5).

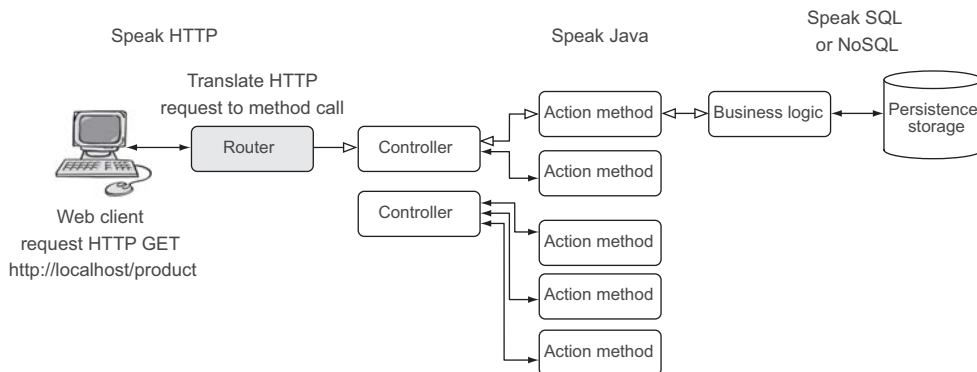


Figure 5.5 Role of the router

Two major pieces of information are contained in the request:

- The request path (such as `/clients /1542, /photos/list`), including the query string (such as `id=2`)
- The HTTP method (`GET, POST, ...`)

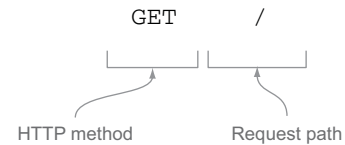


Figure 5.6 Request

For example, when you enter the URL `http://localhost/` in your browser, let's say to view the home page, it issues a request to the `localhost` server. On the server, the request is then decomposed as shown in figure 5.6.

Let's take a look at another example. To display the first page of a product listing, a URL could be `http://localhost/product?page=1`. It would be decomposed as shown in figure 5.7.

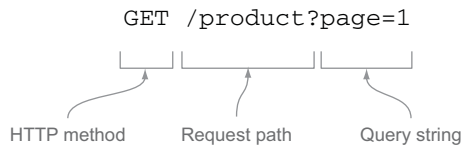


Figure 5.7 Request with query string

The HTTP method can be any of the valid methods supported by HTTP (`GET, POST, PUT, DELETE, or HEAD`). The request path identifies the resource we're trying to serve. Query strings are optional and are used to provide dynamic parameters. A query string is specified after the `?` sign and is always of the form `name=value`. We assume you know all about what a URL is and what HTTP methods are. If not, please read about them in chapter 3.

Let's get back to our application. We'd like to provide certain functionality, and therefore we should be able to respond to the requests shown in table 5.1.

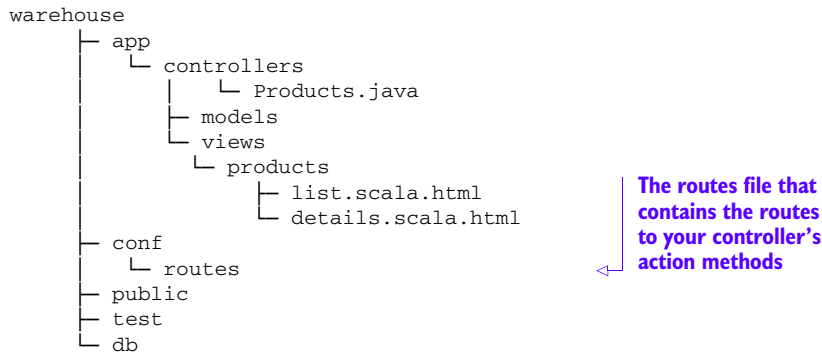
As you might have guessed, we need to translate each of these requests to a controller and action method. This way, we're translating from an HTTP request to a Java method call. This translation is what we call a *route definition*. Route definitions are contained inside a *routes file*. For our application, the routes file will expose the application functionality just listed.

Table 5.1 List of requests

Method	Request path	Description
GET	/	Home page
GET	/products/	Product list
GET	/products/?page=2	The products list's second page
GET	/product/5010255079763	The products with the given EAN
POST	/product/5010255079763	Update the product details

Routes are defined in the `conf/routes` file, as shown in the following listing.

Listing 5.6 Project directory structure



The routes file isn't just a text-based configuration file; it's actually code that will be compiled into a Java object. The object is accessible in our controllers and is called `routes`. This means that you'll see compile-time errors if a route definition is not valid or if a requested URL doesn't exist in your application, as shown in figure 5.8.

This is convenient, as you know immediately that something is wrong in your application. This is another example of Play 2 employing type safety to make your application more robust. Let's see what other benefits we get from that.

Compilation error

value home is not a member of object controllers.Application

In /Users/nicolasleroux/Projects/Personal/book/conf/routes at line 6.

2	# This file defines all application routes (Higher priority routes first)	
3	# ~~~~	
4		
5	# Home page	
6	GET / controllers.Application.home()	
7	GET /:id controllers.Application.index(id:Long)	
8		
9	# Map static resources from the /public folder to the /assets URL path	
10	GET /assets/*file controllers.Assets.at(path="/public", file)	

Figure 5.8 An error in the routes file

TYPE SAFETY IN THE ROUTES FILE

Beside the obvious error reporting, another benefit of compilation is that the routes object is accessible from the controllers and templates. This comes in handy when you want to refer from one action method to another, or when you want to link to an action from a template.

An example of using the routes from a controller is redirection. Let's say we have an `index` method available at `http://localhost:9000/` that should redirect the client to the `product list` method available at `http://localhost:9000/product`. Using the routes object, you can simply refer to the `products` method, which is used to do something called *reverse routing*, which is explained in section 5.3.5. But for now, let's focus on the syntax of the routes file.

5.3.2 The routes files explained

So, what are routes, exactly? As explained previously, the routes file is where the translation between the HTTP request and your code is performed. Let's take a close look at the routes file syntax. The routes file lists all of the routes needed by the application, and each route consists of an HTTP method and a URI pattern associated with a call to an action method. This is what we call a *route definition*.

Let's see what a route definition looks like for the products home page:

```
GET    /                controllers.Products.index() ← Assume we have a
                                     Products controller with
                                     an index action method
```

This means that when a client issues an HTTP request `GET /`, the action method located in the `Products` class should be called. The `Products` class is our code; it's our entry point.

In the routes file, a route definition is composed of the following parts:

- The HTTP method
- The request path
- Optionally, the query string
- The call definition

Figure 5.9 breaks down our example route definition. Figure 5.10 breaks down a route definition that includes the optional query string.

Let's take a look at a more complicated route definition. If we want to have a route definition that displays product details based on the product EAN, we could put the following entries into our route files:

```
GET /products controllers.Products.list
```

Figure 5.9 Route definition

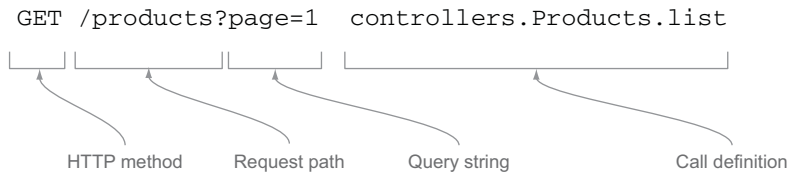


Figure 5.10
Route definition
with query string

```
GET /product/111111111111 controllers.Products.details("111111111111")
GET /product/222222222222 controllers.Products.details("222222222222")
GET /product/333333333333 controllers.Products.details("333333333333")
....
```

But that wouldn't scale very well (not to mention that we'd need to predict future products). We need to have a way to have a dynamic part in our route definition.

5.3.3 Dynamic path parts

Part of the path of our route can actually be used as a parameter for our action method. It would look like this:

```
GET /product/:ean controllers.Products.details(ean: String)
```

We replaced the part of the path that indicates the ID with a parameter name, indicated by the colon (:). We then reference that parameter in the action method call.

You'll notice that in the action method call, we added the parameter type after the parameter name. For parameters of type `String`, specifying the type of the parameter is optional, but it's required for every other type.

Every time we request `/product/111111111111` in our browser, the `details` method on the `Products` controller is called, with `111111111111` as the `ean` parameter. Since our parameter is a `String`, Play doesn't have to do much. But if our parameter's type had been a `Long`, for example, Play would make sure the parameter is transformed into a `Long`. Play is also able to convert to other types such as arrays and dates, and you can even add your own types as well, as we'll show you in chapter 6.

If Play can't convert to the required type, it means that the expected type is not the right one, and you're doing something you should not. You'll see an error screen like the one shown in figure 5.11.

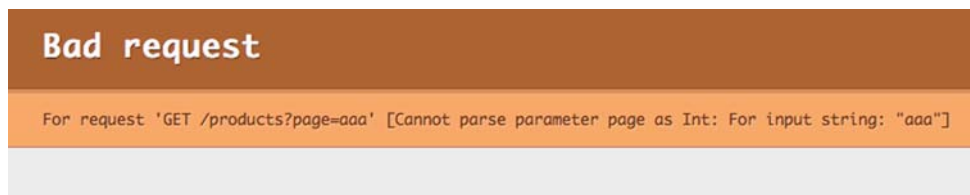


Figure 5.11 Bad request error screen

A word about simple data binding

Type conversion is handled automatically by Play, so you don't have to handle the conversion between Strings and other types. The automatic conversion is called *binding*. It's binding values from the HTTP requests (which are strings by definition) to a Java type. In chapter 6, we'll explain how Play can bind to other types. Binding is actually part of the translation process from HTTP to your Java code.

This is a really nice feature that allows us to concentrate on the problem at hand. If you were using Servlet or any other framework, you'd probably have to write something along the lines of the following listing.

Listing 5.7 A conventional Servlet method

```
public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    try {
        final String id = request.getStringParameter("id");
        final Long idCode = Long.parseLong(id);
        // Process request...
    } catch (NumberFormatException e) {
        final int status = HttpServletResponse.SC_BAD_REQUEST;
        response.sendError(status, e.getMessage());
    }
}
```

Play takes care of this for us. Note that you'll only see the message between brackets (in figure 5.11) while developing. In production, the detailed information is left out. Now let's get back to the discussion about dynamic route parts.

When using this syntax to define dynamic path parts, each parameter will match exactly one path part, which means each part between forward slashes (or the start of the path).

But you might sometimes want more flexibility. If you want a dynamic part to capture more than one request path segment, separated by forward slashes, you can define a dynamic part using the **id* syntax, which will use the rest of the path as the value for the parameter.

For example, let's say that we want to get the path to our product image. The route definition is as follows:

```
GET /product/image/*imagePath
    controllers.Products.downloadImage(imagePath: String)
```

If we issue a request like `GET /products/image/29929/paperclip.jpg`, the `imagePath` value will be `29929/paperclip.jpg`.

If you know what regular expressions are, you can also define your own regular expression for the dynamic parts using the `$id<regex>` syntax:

```
GET /product/$ean<[0-9]{13}>
    controllers.Products.details(ean: Long)
```

This route definition will only apply if the EAN consists of 13 digits, which is handy in our case, because we know that our product EAN codes consist of 13 digits. Play will return a not found error code if we enter alphanumeric characters as IDs.

As you know, you can also pass parameters with your URL. For example, `http://google.com?query=playframework` contains a parameter called *query*, with the value “playframework.” So how do you specify a parameter like that in your routes file so that you can access it from your action methods? Actually, you don’t have to declare it in your routes file at all. You can just use a parameter of the same name in your action method signature:

```
public static Result products(String filter) {
    ...
}
```

If the action method defines some parameters, *all of these parameter values will be searched for in the request path*. They will either be extracted from the request path as we saw before, or they will be extracted from the query string.

Let’s see a more detailed example. Let’s say we want to display a list of products. But that’s potentially a really large list, so we want to paginate it. To access the first 20 items, we will request the first page; for the next 20, we will request the second page, and so on. The following route definitions can be seen as equivalent, functionally:

```
GET /products/:page controllers.Products.list(page: Int)
GET /products/ controllers.Products.list(page: Int)
```

What is the difference between the two route definitions? The first route definition has the page parameter as part of the actual request path. The client requests the following URL to access the second product list page: `http://localhost/products/2`. The second route definition doesn’t require the page parameter to be part of the request URL. Instead, it’s a parameter that the user needs to provide. This is done by requesting the following URL: `http://localhost/products/?page=2`.

It’s interesting to note that `http://localhost/products/2` (where “2” is the page number) isn’t a good identifier for a resource. The product list for the second page is likely to change over time. Therefore the second definition is to be preferred. For more information about RESTful concepts like this, please refer to chapter 4.

Also notice that Play complains if we don’t specify the page parameter in the request URL, as shown in figure 5.12. The error is thrown because Play has no way to tell what the default value for the parameter is. There’s a way to find out, though: default values in the routes file.

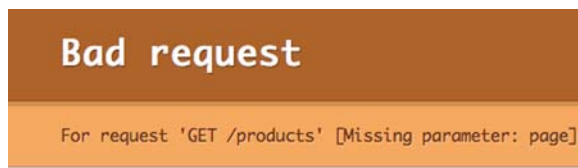


Figure 5.12 Bad request error screen for missing parameter

ROUTE WITH DEFAULT VALUE

We can (and should) choose a default value to use if none is specified in the request. For example, for our list of products with pagination, the following syntax will request the first page to be displayed if no first page is specified.

```
GET /products controllers.Products.list(page: Int ?= 0)
```

Using this syntax, it's impossible to get “Bad request” errors due to missing parameters. When the parameter *isn't* provided, the value specified (“1” in this case) will be used instead. Please note that we're using the `Int` keyword, as it is the Scala representation of an integer. This is the equivalent of the Java `Integer` type.

FIXED VALUE AS PARAMETER

Now let's say that we want the home page to display the first page of our product listing. We can do that using a fixed value as a parameter:

```
GET / controllers.Products.list(page: Int = 0)
```

The value of the `page` parameter will always be 1 on the home page, even if another value is provided using a query string parameter.

Using all these different syntaxes for route definition, it's very possible to construct multiple routes that will match the same URL. What happens then?

CONFLICTING ROUTES

Because many routes can match the same request, if there is a conflict, the first route (in declaration order) is used. For example, in our routes file we have the following route definitions:

```
GET /products/new controllers.Products.newProduct()
GET /products/:ean controllers.Products.details(ean: String)
```

The bottom line would *also* match on `/products/new`. But since there is a line matching that URL first, that is the one that is used. Therefore, calls to `/products/new` will be served by `newProduct()`, which is exactly what we want. If we were to switch the lines around, the same request would be handled by the `details()` method, with the value “new” used for the `ean` parameter, which would cause problems.

We now know enough about route definitions. Let's get back to our application.

5.3.4 Completing our routes file

With all we've learned about routes so far, we can finish the routes file for our application's product catalog. Edit the routes file so it contains the routes shown in the following listing.

Listing 5.8 Our current routes file

```
# Home page
GET / controllers.Products.index()
GET /products controllers.Products.list(page: Integer ?= 1)
POST /products/ controllers.Products.save()
```

These lines changed
from chapter 3

```
GET /products/new      controllers.Products.newProduct()
GET /products/:ean     controllers.Products.details(ean: String)
```

Compared to what we ended with in chapter 3, the first two lines have changed. The first line, which matches the root URL of our application, used to point to `Application.index()`, but it now points to the `index` method of our `Products` controller. Since we now no longer have any routes using the `Application` controller, feel free to delete that class.

The second route still points to our product catalog, but the call to the action method has gained a `page` parameter, which defaults to 1. To get the application to compile and run again, we need to change the action methods to match the routes. The first method, `index`, is new. We want it to show the first page of the product catalog, which we'll do by redirecting to it. Add the following action method to the `Products` class:

```
public static Result index() {
    return redirect(routes.Products.list(0));
}
```

The other change we made in our routes file was that we added the `page` parameter to the product listing. We need to change the action method to match the call in the routes file, or the routes file won't compile. Go ahead and add the parameter to the `list` method in the `Products` class, and fix the call to it on the last line of the `save()` method, like so:

```
public class Products extends Controller {
    public static Result list(Integer page) {
        ...
    }

    ...
    public static Result save() {
        ...
        return redirect(routes.Products.list(1));
    }
}
```

Don't worry about changing the method's implementation; we'll get to that later. For now, it's enough to get our routes compiling again.

So we now know how to link a URL to an action method. But what about the other way around? If we know the action, how do we get a corresponding URL? That is a process called *reverse routing*.

5.3.5 *Reverse routing*

The implementation of our `index` method from the previous section is interesting. It sends an HTTP response that redirects the user to the `list` method. To construct the URL for that method, it uses the `routes` object.

The `routes` object was generated by Play as a result of compiling the routes file. The `routes` object is a singleton object that contains only static methods that return

an object of type `Action`. It's used as a way to reference our action methods from the controller, but it's also used anywhere else we might need it (in our views, for example). Our action methods are added as methods to the object at compilation time, when Play generates the `routes` object.

The `routes` object provides what we call *reverse routing*.

Reverse routing, as the name implies, does the opposite from regular routing: it translates from Java to HTTP. Reverse routing is important, as it allows us to get an HTTP request for a given action method.

For example, say you want to be able to point your client to the `edit` method. Remember, your client only speaks HTTP. You need to return an HTTP call: an HTTP method and a URL. It's as simple as asking the `routes` object how to access the action method. The `routes` object returns a `play.mvc.Call` object. The `play.mvc.Call` defines an HTTP call, and provides both the HTTP method and the URL. It also makes sure the method call is correctly translated, especially when parameters are part of the action method. For example, the following call:

```
routes.Products.list(4)
```

is translated to:

```
GET /products?page=4
```

You now know everything about how to translate HTTP requests to action methods and vice versa with reverse routing. You should now have a complete picture of how Play operates when a request is received from the client and executed as Java code. Now it's time to see in more detail how to simplify some tedious operations in the controller.

5.4 Interceptors

Let's get back to our warehouse application. From time to time, an exception may occur. But if that happens, we can't spot it. It would be nice if we could just tell Play to send an email with the error whenever an exception occurs in specific controllers or action methods.

Action methods can be easily composed. This means that you can add extra behavior to action methods, and that's what we want to do: we want to catch any exceptions occurring in our action methods and send an email about them. Let's see how it works.

5.4.1 The `@With` annotation

Play provides an `@With` annotation, which allows you to compose an action. The `@With` annotation is used before an action method declaration. It can also be used on the class level—on the controller itself. Declaring the `@With` annotation on our action method tells Play that a certain action must be performed *around* each execution, meaning before and after. This is also called an *interceptor*, because it intercepts a call to the action method.

The `@With` annotation takes one parameter: the type of `Action` we want our code to be composed with. We will build a `CatchAction` class shortly.

For example, the following code tells Play it must execute the action method using the `CatchAction` action:

```
@With(CatchAction.class)
public static Result show(Long id) { .... }
```

Because we specify the annotation on the method level, Play will only use it for this specific action method. If we were to declare it at the class level, the `CatchAction` would be used for all the controller's action methods:

```
@With(CatchAction.class)
public class Products extends Controller { .... }
```

But what exactly is this `CatchAction` class? It's where we will put our code that will provide the added functionality. We need to build it. The `@With` annotation takes an `Action` object as a parameter. An action object must extend the abstract class `play.mvc.Action`, which means it must implement the following method:

```
public Result call(Http.Context ctx)
```

The `call` method is called before the action method execution. From there, the `call` method must actually call the action method using the `delegate` object. The `delegate` object is a reference to the action method of type `play.mvc.Action` that is marked with the `@With` annotation. In other words, it represents the original action method.

Let's just go ahead and code our exception interceptor. First, we'll create a fake `ExceptionMailer` class, to stand in for what would be an actual emailing class, which is not the point of this exercise. Create the file shown in the following listing.

Listing 5.9 `/app/utils/ExceptionMailer.java`

```
package utils;

public class ExceptionMailer {
    public static void send(Throwable e) {
        System.out.println("Sending email containing exception " + e);
    }
}
```

We'll create our interceptor action by extending the `Play.SimpleAction` abstract class that Play provides. The `Play.SimpleAction` class provides everything we need to get started. In the next listing, we'll define our `CatchAction`, which will catch any exceptions and email them.

Listing 5.10 /app/controllers/CatchAction.java

```
public class CatchAction extends Action.Simple {
    public F.Promise<SimpleResult> call(Http.Context ctx) {
        try {
            return delegate.call(ctx);
        } catch (Throwable e) {
            ExceptionMailer.send(e);
            throw new RuntimeException(e);
        }
    }
}
```

We execute
the action

This is the main method
we need to implement

We extend the
Action.Simple
class that extends
the abstract
Action class

Any exception is caught
and sent using our
ExceptionMailer class

We can now use our CatchAction on our Products controller, by annotating the controller with @With(CatchAction.class):

```
...
import play.mvc.With;

@With(CatchAction.class)
public class Products extends Controller {
    ..
}
```

Let's see how our implementation does its trick.

5.4.2 Explaining our CatchAction

The behavior of our custom Action is defined by our implementation of the call method. That method takes one parameter, the Play *context* object. The context object holds our session, request, response, and flash objects. We'll take a closer look at those concepts in the next section.

In our implementation, the actual call to the action method is done via the delegate object, which contains its own call method. In case an exception is triggered, we catch it, and our ExceptionMailer class sends an email with the exception stack trace.¹

Now that we know how actions work, let's see some other ways to use them.

5.4.3 Action composition

Once you have one interceptor, you're probably wondering how you can use more of them; for example, a LogAction that logs any access to our controllers. We can just add another parameter to the @With annotation:

```
@With(CatchAction.class, LogAction.class)
```

If you are familiar with annotations,² you can also define your own, to signify that a certain action should be added to an action method. This is a more readable notation,

¹ The ExceptionMailer code is not relevant at this moment, as the goal is to show you how interceptors work.

² Read http://en.wikipedia.org/wiki/Java_annotation if you're not.

and it allows reuse across multiple web applications. For example, an annotation for our `CatchAction` would be:

```
@With(CatchAction.class)
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Catch {
}
```

In this example, we define a new annotation called `Catch`, and use the `@With` annotation as usual to indicate which action class it should use. Using our newly defined annotation, we can now annotate our controller as follows:

```
@Catch
public class Products extends Controller { .... }
```

← Our newly defined annotation

We don't just use our own annotation is for readability purposes; we can also use it to pass extra configuration information. Using the previous example, we could specify if we want to send an email, or if we want to log the exception instead. Let's redefine our catch annotation:

```
@With(CatchAction.class)
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Catch {
    boolean send() default true;
}
```

← Add this line

We've added a `send` parameter to our annotation, with a default value of `true`, specifying whether we want to send an email or just log instead. The parameter value can be accessed via the configuration object in our action. The following listing shows how.

Listing 5.11 `CatchAction` using configuration

```
public class CatchAction extends Action<Catch> {
    public F.Promise<SimpleResult> call(Http.Context ctx) {
        try {
            return delegate.call(ctx);
        } catch (Throwable e) {
            if (configuration.send)
                ExceptionMailer.send(e);
            else
                e.printStackTrace();
        }
    }
}
```

1 We extend Action rather than Action.Simple

2 We read the configuration value

3 We either send an email...

4 ... or log the error

In this version of `CatchAction`, we extend the `Action` class directly, supplying a generic type parameter to indicate that we want to use our `Catch` annotation for configuration ①. The configuration object will now be an instance of our `Catch` annotation, meaning we can access the `send` parameter on it ②. We can then use a simple `if` statement to decide what we want to do ③ ④.

We are now able to be notified of any exceptions occurring in our code. This is a trivial example, but think about how you can use interceptors for transactions, security, and for a lot of other interesting examples you can think of.

There's one more important aspect of controllers that we haven't covered yet: scopes.

5.5 About scopes

We saw that by using the controllers and the routes, we're able to retrieve data from the clients and send data back to those clients. We didn't really talk about the lifetime and accessibility of that data. We assumed the data was transmitted each time a communication was made between the server and the client; that really is what's happening.

But as a developer, you don't really want to manage that dataflow. You want to store data for a certain period of time; for example, for the duration of a request or a browser session. You store that data in a certain *scope*. Play supports a number of scopes, for which it stores data for a certain lifetime. They're accessed in a similar way as a Map in Java; you store and retrieve values based on a key.

5.5.1 A bit of history about the scopes

Java EE traditionally defines these scopes:

- Application scope
- Session scope
- Request scope
- Page scope

The *application scope* has an application lifetime: you store data that will stay as long as your application is running.

The *session scope* has a session lifetime: you store data that will stay as long as your browser is open. This is traditionally the `JSESSIONID` parameter you sometimes see in URLs. The `JSESSIONID` is also stored in a cookie, so it's always available to the server on consecutive requests. This `JSESSIONID` is just an ID that points to some server-side storage space. The session scope is usually used to store information about your shopping basket or the fact that you're logged in.

The *request scope* defines data within the lifetime of request: the client makes an HTTP request, and with the request comes some data. That data is stored in the request scope and can only be accessed while that particular request is being processed.

The *page scope* defines data that is accessible in the view only: it can be accessed during the rendering phase. For example, you might want to store the current breadcrumb of your application in that scope, to display the current application path.

The different scopes can be conceptually viewed as shown in figure 5.13.

NOTE Though some Java EE frameworks introduce a *conversation scope*, we don't think this is relevant for our explanation and may be confusing more than anything else.

In a traditional Java EE environment, all the storage happens server-side. This means that each client has a unique ID, and the server uses that unique ID to retrieve the client storage space server-side.

Though this has the advantage of potentially using less bandwidth, it can be problematic when scaling up (adding more servers) since you need to synchronize all the client storage space between the different servers. Another disadvantage is that it makes the servers compute more operations, while the client stays idle. This used to be an advantage, but nowadays web clients are powerful beasts. An iPhone is more powerful, CPU-wise, than any computer older than five years. Moreover, web standards have evolved and added a lot of features to web clients, such as local storage and web workers.

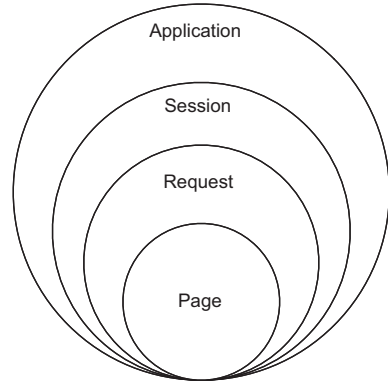


Figure 5.13 In JEE, application scope is the longest lived and page scope is the shortest lived.

5.5.2 Storing data with Play

Of course, as you may have guessed by now, Play is a bit different from Java EE frameworks. It's interesting to note that web frameworks from other languages, such as Django, Ruby on Rails, and Symphony, have been following an approach similar to that of Play.

Play defines four scopes:

- Session scope
- Flash scope
- Request scope
- Response scope

Before we start explaining those scopes, *there is a fundamental difference* between Play and the more traditional Java EE frameworks (read: servlet-based frameworks). *Play doesn't hold any data server-side. Data is held either in the client or in the database.* Because of that fact, scaling up is easy with Play. Just add a new server and a load balancer, and you're done. No server-side session replication is needed, so there's no risk of losing data on the server since there is no data to lose. But how does that work? What's Play's secret?

What is a cookie?

A cookie, also known as an HTTP cookie, web cookie, or browser cookie, is used by a web server to store data in the client's browser. The browser sends that information back to the server on every request. That information can be used, for example, for authentication, identification of a user session, user preferences, shopping cart contents, or anything else that can be accomplished through storing text data on the user's computer. A cookie can only be used to store text.

Well, really, there is no secret. Play stores the data client-side using cookies. It also encourages developers to think differently, to architect their application differently (following REST principles), and to fully embrace client-side technology.

While a Java EE developer would see the session as a giant cache in which everything is allowed, Play forces developers to think in terms of web development. And that is a good thing, since we happen to use Play mainly to develop web applications. So, no, you can't store your complete object tree in your session. And no, you can't save your last 10,000 database results. In fact, you can "only" store up to 4 KB (maximum cookie size), and you can only store string values. You might see that as a step back, but we'll attempt to explain to you that in fact it's a step forward. But first, let's go back and explain the four scopes in Play.

The four different scopes can be conceptually viewed as shown in figure 5.14.

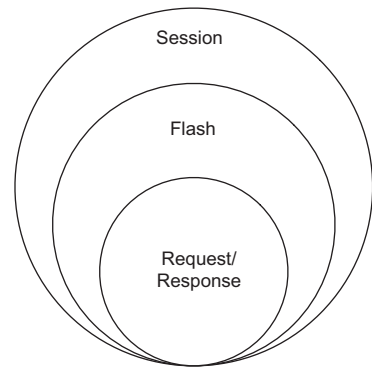


Figure 5.14 In Play, request scope is the shortest lived and session scope is the longest lived.

5.5.3 The context object

First of all, all accessible scopes in Play are stored in a Context object. The Context object is a final static class that contains the Request, Response, Session, and Flash objects. You can access it statically using the `current()` method. From there, you can access all the Play scopes:

```
Context ctx = Context.current();
Request request = ctx.request();
Response response = ctx.response();
Session session = ctx.session();
Flash flash = ctx.flash();
```

← Context object
 ← Request object
 ← Response object
 ← Session object
 ← Flash object

In your controllers, the request, response, session, and flash objects are also available directly, since they are part of the Controller class that any controller must extend.

Let's take a look at each of the scopes, starting with the request scope.

The context object and thread safety

If the context object is static, what does that mean for thread safety? The answer is easy. Each Context object is associated to the current thread using a ThreadLocal object. This ensures that there's no thread-safety problem and that your context object is really your context object and not the one from your neighbor (each thread holds an implicit reference to its copy of a thread-local variable). This ensures that our scope objects are *our* scope objects.

If you didn't follow all of that, just take our word for it: access to the context and scope objects is thread-safe.

5.5.4 The request scope

The request scope can't be used to store objects. It is used to access the data of the current request. For example, you can access the values submitted using an HTML form via the request scope.

5.5.5 The response scope

The response scope can't be used to store objects. It is used to set the response content type and any extra cookies to send extra information. For example, you can set the response content type to XML with the following code:

```
Context.current().response().setContentType("application/xml");
```

Charset

For text-based HTTP responses, it's important to set the character set (or character encoding) correctly when you set the response content type. Play handles that for you and uses UTF-8 by default.

The charset is used to both convert the text response to the corresponding bytes to send over the network socket, and modify the `ContentType` header with the proper `;charset=xxx` extension.

The charset can also be specified when you are generating the result value, in this case the `ok` result value:

```
public static Result index() {
    response().setContentType("text/html; charset=iso-8859-1");
    return ok("<h1>Hello World!</h1>", "iso-8859-1");
}
```

Set the charset (encoding)
on the response

Our result is using ISO-8859-1

Using the response object, we can also store extra data on the client, using a new cookie. For example, if we were to save the preferred theme for our application from our controller:

```
response().setCookie("theme", "blue");
```

Then, on a next request, from our controller's action method, we could check if the theme was set using the cookie method:

```
public static Result index() {
    if ("blue".equals(cookies("theme").value())) {
        // Do something
    }
    ....
}
```


You could also store more information by serializing your data into strings, but remember that you can only store up to 4 KB per cookie. Therefore, save only what you need, which is often entity IDs.

Finally, to discard a cookie, use `discardCookies`:

```
response().discardCookies("theme");
```

As you can see, the response object is rather straightforward.

5.5.6 The session scope

Objects stored in the session scope have a session lifetime: you can store data that will stay as long as the client's browser is open. It's important to understand that sessions are *not* stored on the server but are added to each subsequent HTTP request, using the cookie mechanism. So the data size is very limited (up to 4 KB), and you can only store string values. This means that the `Session` object *should not* be used as a cache! Play does offer a caching mechanism, which you can use instead.

To store a value in the session:

```
Context.current().session().put("x", "myvalue");
String value = Context.current().session().get("x");
System.out.println(value);
```

Print myvalue →

← **Store the value myvalue in the session with key x**

← **Retrieve the value from the session**

From a controller, you can also use the convenience methods provided by the controller:

```
session("x", "myvalue");
String value = session("x");
System.out.println(value);
```

Print myvalue →

← **Store the value myvalue in the session with key x**

← **Retrieve the value from the session**

Why can I only store strings in my Session?

You may argue that a `Session` object that can only store `String` objects is a regression; we feel that it's the way it should be. By disallowing the storing of complex objects, we're also removing a lot of associated problems of synchronizing the object states and/or any side effects. By storing immutable objects, like `Strings`, in our session, no side effects can occur.

For example, if we were to store an attached (marked as connected to the DB) database entity in our session, the entity might really well be detached (no longer connected to the DB) the next time we access the object, causing all sorts of troubles. Or the entity may have been modified by a third party since we last accessed it, causing unexpected behavior.

5.5.7 The flash scope

Objects stored in the flash scope have a lifetime of exactly two requests. It means that data in the flash scope will service one redirect. This is really useful when you want to retain data after a redirection. To understand this better, let's look at a concrete example.

The client issues the following request:

```
GET /
```

The server renders an HTML page. On the page, we have a form to input a telephone number and a *Submit* button. When the user submits the form, a POST request is made to the server to the following address:

```
POST /phonenumbers/
```

Now, once the controller receives the request, it attempts to validate the phone number. In case the validation fails, the server renders the same page with an error message. But if you try to refresh your browser, you're in for a surprise: your browser is asking you if you want to resubmit the data. Why is that? This is because, as far as your client is concerned, its last execution point matches POST /phonenumbers. And indeed, the URL that your browser shows is http://localhost:9000/phonenumbers.

Figure 5.15 illustrates the problem we just described.

A way to avoid bad surprises is to send a redirect instruction to your client, so it redirects to GET /. But of course, in the meantime, because of the redirect, you lost your error message. This is exactly the use case for which flash scope was invented. Storing the error messages in the flash scope allows you to still have access to your error messages after the redirect.

In Play, the objects in the flash scope are stored in a special cookie that's flushed after the second consecutive request.

Figure 5.16 illustrates the same problem, but uses the flash scope as a solution.

NOTE The flash scope has been introduced in Java EE 6 and defines the same lifetime as its Play equivalent, but it lives server-side. At the time of this writing, most Java EE web frameworks do not have flash scope.

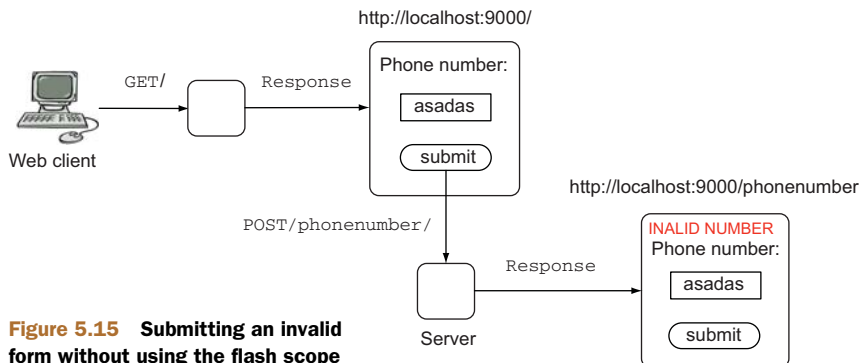


Figure 5.15 Submitting an invalid form without using the flash scope

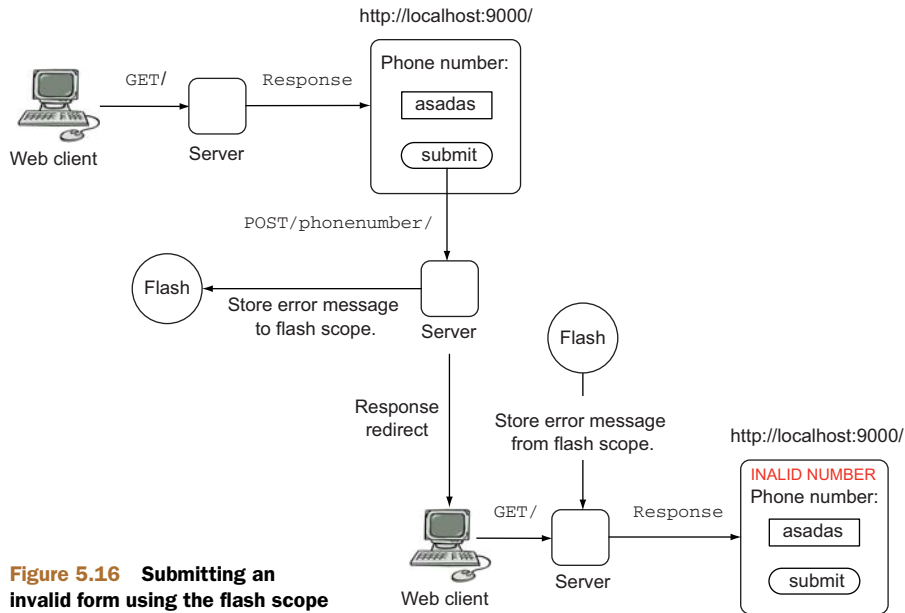


Figure 5.16 Submitting an invalid form using the flash scope

5.5.8 What about security?

Storing data client-side brings some security concerns with it. Because the user has access to this data, it can't be trusted without taking some additional measures. For example, when you store the user's username, there is nothing preventing the user from putting in another username and impersonating a different user.

For this reason, cookies are signed with a secret key so that the client can't modify the cookie data (if they do, the data will be invalidated).

The secret key used to sign the cookie is actually set in your `conf/application.conf` file (see following listing).

Listing 5.12 Project structure

```
warehouse
├── app
│   ├── controllers
│   │   └── Products.java
│   ├── models
│   └── views
├── conf
│   ├── routes
│   └── application.conf
├── public
├── test
└── db
```

application.conf contains the application configuration and the important secret key

If you open the `application.conf` file in an editor, you will find a line starting with `application.secret`:

```
application.secret="FuqsIcSJlLppP8s?UpVYb5CvX1v55PVgHQ1Pk"
```

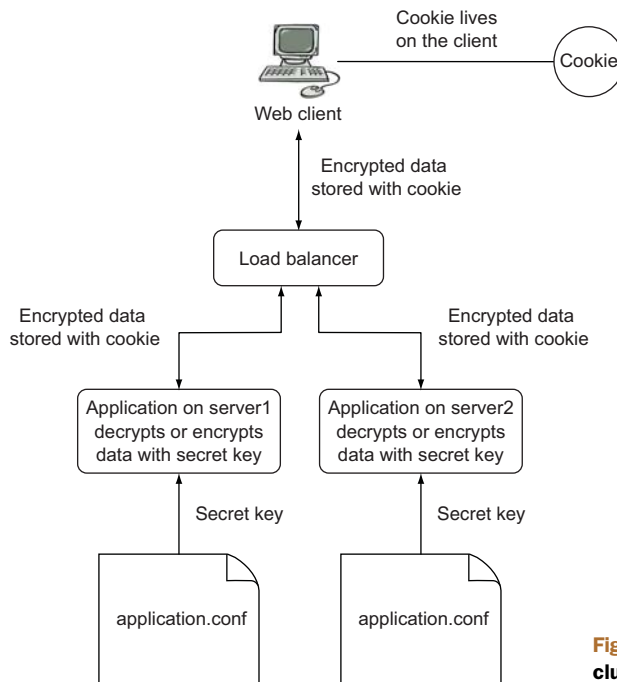


Figure 5.17 Secret key with a cluster of applications

This is the key we’re signing our cookies with. It’s essential to keep it secret, so be sure you don’t divulge your key. This also brings us to another important point: if you’re running multiple instances of your application, the secret key needs to be shared between the applications, or one instance won’t be able to verify the data that has been set by another instance.

Let’s imagine the following scenario. One load balancer and the same application are running on two different nodes: server1 and server2. Each client can be dispatched to either server1 or server2. If server1 and server2 don’t share the same key, the clients will need to communicate with the same server every time; otherwise the server can’t decipher the cookies. If both server1 and server2 have the same key, they can decipher the same cookies, and neither the client nor the load balancer will need to distinguish between server1 and server2.

Figure 5.17 illustrates the data encryption/decryption flow.

But what about my session timeout?

We can argue that session timeouts were introduced as a technical solution for server-side session storage rather than as a useful feature. Indeed, without this feature, user sessions could only grow on the server, resulting in memory deprivation. But developers are so used to session timeouts that they actually see them as useful functionality.

In Play, there is no technical timeout for the session. It expires when the user closes the web browser. If you need a functional timeout for a specific application, just store a time stamp in the user session and check it against your application needs (max session duration, max inactivity duration, and so forth).

5.6 Summary

We started this chapter by explaining what controllers are. We then looked into the specifics of Play controllers and more particularly the controller's action methods. We learned about routing our clients' requests to our action method code.

Let's pull out some of the key practices to take away from the chapter:

- Use flash scope. Flash scope is ideal for passing messages to the user (when a redirect is involved).
- Use action methods. This is the entry point for your business logic. Keep them short and delegate all business logic to your business models.
- Simple data binding is URL-centric data mapping to your action methods.

We covered how to use interceptors and why they're useful. We implemented a simple interceptor that catches all errors and sends an email.

We've learned a lot about the internals of controllers in this chapter, and in the next chapter we'll build on our knowledge by implementing some nice views to give our warehouse application some visual appeal.

Handling user input

This chapter covers

- Working with forms
- Data binding
- Using body parsers
- Validation
- Handling file uploads

In this chapter, we'll explain in detail how users can interact with our application. This is where we'll enable users to send data to our application. We'll see how to handle different kinds of data and how to customize Play to use our own data types. We'll also explain how to make sure the data sent is correct and, if it's not, how to alert our users.

6.1 Forms

Working with forms in a web application involves two operations: displaying the form and processing the form submission. Forms allow users to send data to the server (our application). In Play, forms are represented by objects that you can manipulate. Play provides useful helpers to handle form submission. Because we don't want users to send just any kind of data, Play makes sure the data is properly

formatted and processed. If data isn't well formatted, we need to tell the user what the problem is, so they can correct it and resubmit the form.

In our application, to submit a new product, we created two action methods: the first one, called `newProduct()`, displays the *create a new product* form, whereas the second, `save()`, handles the form submission. First we'll take a closer look at what exactly the action method that shows the empty form does.

As a reminder, figure 6.1 shows the form.

As we've seen, displaying a form involves the following steps:

- Creating an action method to display the form template
- Creating the form template
- Creating an action method to handle the form submission

Figure 6.2 illustrates the different steps.

Figure 6.1 Our “create new product” form

6.1.1 Displaying the new product form

To display a form, we create a `Form` object to represent it. In this case, we want our form to represent a `Product` instance, so we create a form based on the `Product` class (a `Form<Product>`). Our existing `newProduct()` action method does that:

```
...
import play.data.Form;

public class Products extends Controller {
    private static final Form<Product> productForm =
        Form.form(Product.class);
    ...
    public static Result newProduct() {
        return ok(details.render(productForm));
    }
    ...
}
```

← Create a new
form object based
on `Product`

← Render a template
with our `Form` as
argument

As you can see, we're creating a new `Form` object using the `Form.form()` method. The resulting object holds all the information about our product and can contain additional information that we can use while rendering the form, such as validation

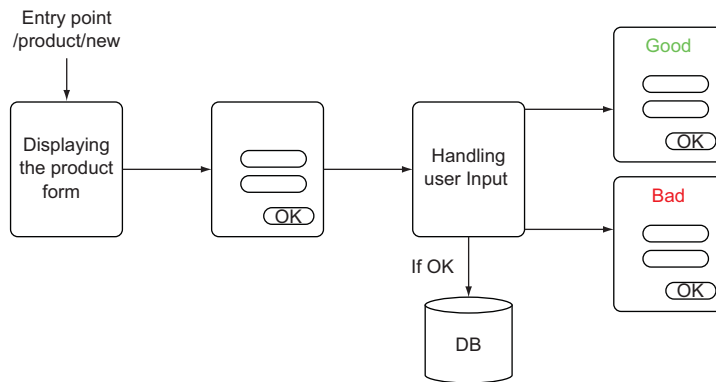


Figure 6.2 “Create product form” workflow

errors, prefilled values, and so on. We’ll examine the form object in more detail later in this chapter.

THE ROUTE TO OUR FORM

When we created the `newProduct()` action method back in chapter 3, we also set up a route to it. Here’s a reminder:

```
GET /products/new controllers.Product.newProduct()
```

This route makes the form available at `http://localhost:9000/products/new`.

ADDING THE VIEW

Also in chapter 3, we created the `product.scala.html` template that renders our HTML product form. The `newProduct()` action method uses it to render a blank product form.

The “new product” form template allows us to collect the information needed to create a new product. It uses HTML input tags to allow the user to input information.

The important thing to realize is that the HTML form is backed by a Play Form object. A form object is a *container* object. It contains information about our object model and, potentially, validation error information. In section 6.4 we’ll talk extensively about validation.

For now, it’s important to know you can access any form field value via a form object, and that you can use it to re-render the original value that was entered by the user if the action method detects a validation error. Our input helpers take care of that for us, so we don’t see the code that does it.

The following listing shows the full template for our form.

Listing 6.1 The “new product” form template

```
@(productForm: Form[Product])
@import helper._
@import helper.twitterBootstrap._

@main("Product form") {
  <h1>Product form</h1>
  @helper.form(action = routes.Products.save()) {
```

← Form object passed
into template by
`newInstance()`


```

<fieldset>
  <legend>Product (@productForm("name").valueOr("New"))</legend>
  @helper.inputText(productForm("ean"))
  @helper.inputText(productForm("name"))
  @helper.textarea(productForm("description"))
</fieldset>
<input type="submit" class="btn btn-primary">
}

```

Access form's name value

Access form's EAN value

Access form's description value

This HTML template is responsible for rendering the new product page (repeated in figure 6.3).

If your form looks different, please refer back to chapter 3, in which we showed you how to include Bootstrap to make things look a little nicer.

This form is for creating a *new* product and, therefore, it's empty. Let's see how the same template can serve as an *edit* form.

6.1.2 Displaying the edit product form

In our application, we also want to be able to edit a product. The “edit product form” is more or less the same form as the “create new product” form. The only difference is that we need to prefill some values and update an existing model object. But from a presentation point of view, the two forms are similar.

To turn the create form into an edit form, we perform the following actions:

- Prefill the edit form with the product value
- Display the edit form with the prefilled data
- Handle the user's input

Figure 6.4 illustrates the workflow.

Figure 6.3 The “create new product” form

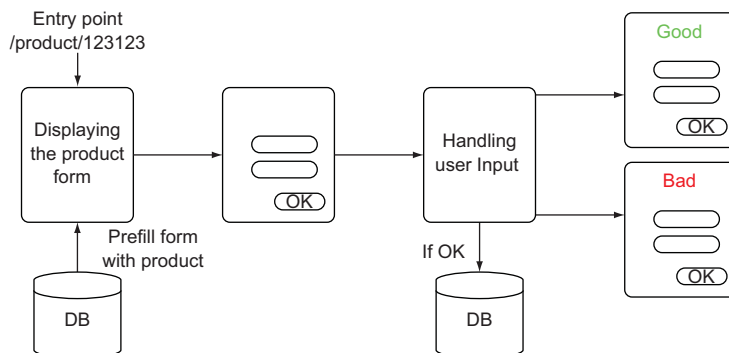


Figure 6.4 “Edit product” workflow

FILLING A FORM WITH INITIAL VALUES

Because we're using a form object in our view, we can decide to prefill the form with some default values. This is exactly what we need to do if we want to edit a product. When we know the unique identifier for the object we want to edit, we can then fetch it from our database so we can use the values in our HTML form. The action method we wrote to accomplish this is shown in the following listing.

Listing 6.2 Displaying a form with preset values

```
public class Products extends Controller {
    private static final Form<Product> productForm =
        Form.form(Product.class);

    ...

    public static Result details(String ean) {
        final Product product = Product.findByEan(ean);
        if (product == null) {
            return notFound(String.format("Product %s does not exist.", ean));
        }

        Form<Product> filledForm = productForm.fill(product);
        return ok(details.render(filledForm));
    }
    ...
}
```

We also added the following route in our routes file. The route indicates that every time we access the `/products/xxx` page, we will in fact render an edit page for the product `xxx`, where `xxx` is the product's EAN number.

```
GET /products/:ean controllers.Product.edit(ean:Long)
```

Now let's see why the existing `products/details.scala.html` supports both creating and updating a product. Take a look at the following listing.

Listing 6.3 `/products/details.scala.html`

```
@(productForm: Form[Product])
@import helper._
@import helper.twitterBootstrap._

@main("Product form") {
    <h1>Product form</h1>
    @helper.form(action = routes.Products.save()) {
        <fieldset>
            <legend>Product (@productForm("name").valueOr("New"))</legend>
            @helper.inputText(productForm("ean"))
            @helper.inputText(productForm("name"))
            @helper.textarea(productForm("description"))
        </fieldset>
        <input type="submit" class="btn btn-primary">
    }
}
```

If the name field has a value, we show a different legend

The form helpers take care of rendering the value, if there is one

As you see, the edit product form and the create new product form can use the same template. If the form contains a value for the product's ID, then it's an update; otherwise, we're creating a new product.

We're now able to display a form to create or edit a product. Next, we need to process the data from the submitted form.

6.1.3 Processing form input

After the form is submitted, our action method that handles the form submission transforms the data the browser sent to the server into a `Product` instance and saves it. Finally, it sets a success message in the flash scope¹ and redirects to the page showing all products. The following listing shows `save()`, an action method that processes the form submission.

Listing 6.4 The `save()` action method

```
public class Products extends Controller {
    private static final Form<Product> productForm =
        Form.form(Product.class);

    ...
    public static Result save() {
        Form<Product> boundForm = productForm.bindFromRequest();
        if (boundForm.hasErrors()) {
            flash("error", "Please correct the form below.");
            return badRequest(details.render(boundForm));
        }

        Product product = boundForm.get();
        product.save();
        flash("success", String.format("Successfully added product %s", product));

        return redirect(routes.Products.list(1));
    }
}
```

① Create a Form object from the request

Detect errors (we'll cover this later)

② Extract a Product instance from the form

Save that instance

Redirect to the "view all products" page

First, we're creating the form object with all the information that we have received via HTTP ①. Then, we check whether the submitted information is valid, and display a message if it isn't.

If everything's in order, we ask the now-populated form object for the `Product` instance ② that it contains. The object instance will have its fields populated with the data extracted from the HTTP request. We then save the `Product` instance to our data store. Finally, we present another page to the user, using a redirect to the "all products" page.

Our `save()` action method is also represented in our routes file:

```
POST /products/ controllers.Product.save()
```

¹ We introduced the flash scope in the previous chapter, in which we also added the message to our main template.

We’ve seen how our form submission is processed. Once the form is displayed, the user enters the data and submits the form. The browser will then send a `POST` request with the form data to our application. The routes file determines that the processing is to be delegated to the `save()` action method.

We’re now able to add or edit a product, but let’s see in more detail how our action method managed to transform the HTTP form parameters into a `Form` object.

6.2 Data binding

Play transformed the submitted information into a Java object using a process called *data binding*. Data binding is the process of converting elements of a request into objects. There are three binding types in Play:

- Form binding, which we saw in the previous section
- URL query parameters binding
- URL path binding

Data binding is a simple principle. The HTTP parameters are mapped to their counterpart attributes on the Java object. The mapping is done using the parameter’s name. Each time a parameter name matches the name of an object attribute, the object attribute is set to the parameter’s value. Figure 6.5 illustrates this binding process.

In Play, *form binding* is primarily done with the help of a `Form` object, via the `bindFromRequest()` method. The binding processing is delegated to *Spring Data*, a library that can take a request’s body and convert it to the properties of a Java object.

URL query parameter binding consists of taking the URL’s parameters (for example, `?x=y`) and mapping them onto a Java object. The pattern is derived from the routes file.

For example, if our products had an ID, we could have a route like this:

```
GET /products Products.edit(id: Long)
```

Which would match the following URL:

```
http://localhost:9000/products?id=12
```

Play will convert the string “12” (query parameters are always strings) to the `Long 12`.

URL query path binding consists of mapping a query path (again, always a string), to another Java type or object. This also happens using the route definitions. For example, when you declare:

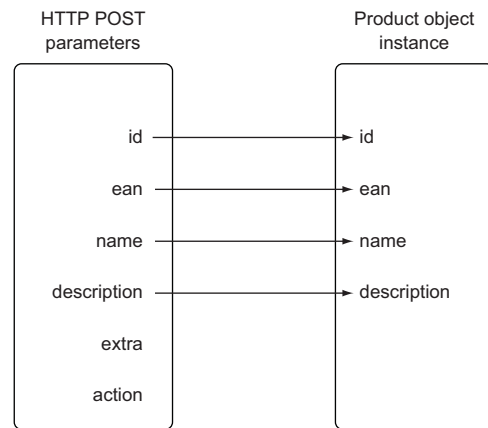


Figure 6.5 The binding process

```
GET /products/:id Products.edit(id: Long)
```

and you call:

```
http://localhost:9000/products/12
```

Play will convert the string “12” (the query path) to the Long 12.

Data *unbinding* is the reverse process. We have an object and we want to translate it into a valid URL or form representation. This is notably used in case of reverse routing. For more information about reverse routing, please refer to section 5.3.5.

It’s important to note that in Play, each type of binding can be customized and extended. But let’s take a step back first, and see how our product form is mapped.

6.2.1 Binding single values

Product is defined as the following class:

```
public class Product {
    public String ean;
    public String name;
    public String description;
}
```

To fully understand how binding works, we’ll cheat a little and respond to a GET HTTP request to trigger our `save()` action method, so that we can easily make requests using a browser.

Let’s edit our routes file, and add the following line:

```
GET /products/save controllers.Product.save()
```

We can now invoke our `save()` action method from the browser, and we should remove this route entry once we’re done experimenting (and use the POST `/products/` one).

```
http://localhost:9000/product/save?ean=1111111111111&
name=product&description=a%20description
```

The first line of the `save()` action method is the most important one:

```
Form<models.Product> productForm =
    form(models.Product.class).bindFromRequest();
```

This creates a new `Product` instance and then, for each HTTP parameter that matches a property by name, sets the value on that property. For this GET request, a new `Product` is created and the `ean` parameter is bound to the `ean` property, the `name` parameter is bound to the `name` property, and so on.

Our product form is filled with the HTTP parameter values from our request. We can then access the newly created `Product` instance with the following code:

```
models.Product product = productForm.get();
```

And the product object contains the following data:

```
ean: 1111111111111,
name: product,
description=a description
```

As you can see, it matches with the parameters we supplied in our URL. It's a simple and predictable mechanism. It's now trivial to map back and forth from HTTP parameters to objects.

It works well for simple types such as `String`, `Integer`, `Long`, and so forth. Play also provides support for more complex types such as `Date` and for multiple values that must be mapped to an array or `List` object.

For dates, you can annotate your model with the `@play.data.format.Formats.DateTime` annotation to indicate how to transform the HTTP parameter into a `Date` object:

```
@Formats.DateTime(pattern = "yyyy-MM-dd")
public Date date;
```

Here, if we use the date HTTP parameter with a year, month, and day date format, Play will be able to transform the string into a `Date` object using a form data binder called `Formatter`. For example, calling

```
http://localhost:9000/product/save?date=2021-10-02
```

results in a `Date` object when bound to a `Form` object.

Play provides some built-in `Formatters`, namely `DateFormatter`, `AnnotationDateFormatter`, and `AnnotationNonEmptyFormatter`. `DateFormatter` attempts to convert parameters to `Dates` with a date format of `yyyy-MM-dd` whenever a `Date` object is required. `AnnotationDateFormatter` is used in conjunction with the `@DateTime` annotation and allows you to specify a date format. The `AnnotationNonEmptyFormatter` is used in conjunction with the `@NonEmpty` annotation and prints an empty string instead of a null value.

All the built-in `Formatters` are located in the `play.mvc.data.Formats` class.

Next, let's see how binding multivalued parameters works.

6.2.2 Binding multiple values

This is a common use case: the user selects multiple values from a list of possible values, and we have to store them. To illustrate this use case, let's define a new scenario. We want to be able to tag our product with a certain label. For example, we can tag a product with the words *metal* or *plastic*. Each product can have zero or more tags, and each tag can be applied to multiple products. Figure 6.6 illustrates this relationship.

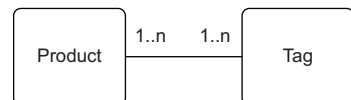
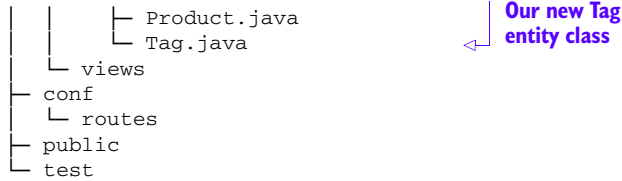


Figure 6.6 Product-Tag relationships

We first need to define our new model. Let's create a `Tag` model class. We create a new file in the `app/models` directory.

Listing 6.5 Project directory structure

```
warehouse
├── app
│   ├── controllers
│   │   └── Products.java
│   └── models
```



It's a relatively simple class; it holds a tag name. The following listing shows our new class.

Listing 6.6 /app/models/Tag.java

```

package models;

import play.data.validation.Constraints;
import java.util.*;

public class Tag {
    public Long id;
    @Constraints.Required
    public String name;
    public List<Product> products;

    public Tag(){
        // Left empty
    }

    public Tag(Long id, String name, Collection<Product> products) {
        this.id = id;
        this.name = name;
        this.products = new LinkedList<Product>(products);
        for (Product product : products) {
            product.tags.add(this);
        }
    }
}

```

We also need to modify our Product class to link the Tag class. We use a List so that each product can hold multiple tags. This is easily done by adding the following line to our Product class (/app/models/Product.java).

```
public List<Tag> tags = new LinkedList<Tag>();
```

We now need to create some tags. In order to do that, we'll fake some Tag objects in the same way that we created some test Products. We'll show you how to use a database as a backing store in chapter 7. For now, let's stick with a static list of tags. In the Tag class, add the lines shown in the following listing.

Listing 6.7 Adding mock data to Tag.java

```

public class Tag {

    private static List<Tag> tags = new LinkedList<Tag>();

    static {
        tags.add(new Tag(1L, "lightweight",
            Product.findByName("paperclips 1")));
    }
}

```

The lightweight tag is added to product names matching paperclips 1

```

tags.add(new Tag(2L, "metal",
    Product.findByName("paperclips")));
tags.add(new Tag(3L, "plastic",
    Product.findByName("paperclips")));
}

public static Tag findById(Long id) {
    for (Tag tag : tags) {
        if(tag.id == id) return tag;
    }
    return null;
}
...
}

```

← The plastic tag is added to all the products (they all match paperclips)

← The metal tag is added to all the products (they all match paperclips)

We now have our tags ready to be used. Next we need to modify our view so we can tag our products. Because we want to keep things simple, we'll use a predefined set of tags rather than dynamically generate the list of check boxes. Let's add the following code snippet to our `products/details.scala.html`.

```

<div class="control-group">
<div class="controls">
<input name="tags[0].id" value="1" type="checkbox" > lightweight
<input name="tags[1].id" value="2" type="checkbox" > metal
<input name="tags[2].id" value="3" type="checkbox" > plastic
</div>
</div>

```

We've added some HTML input elements of type checkbox. For the binding process, the important part of these input elements is the `id` attribute. The `[]` notation tells Play that we want to bind the check boxes to the product's tag collection (in our case, the `List<Tag>`). Once the form is submitted, Play will automatically create a new `List`, with new `Tag` objects for each checked box, and with the ID specified in the value attribute. For example, if we select "lightweight" and "plastic," Play will create a new `List` with two new `Tag` objects during the binding process. The first tag object will have its ID set to 2 and the second one to 3. Figure 6.7 illustrates exactly that.

Now that we have our form ready with the check boxes, let's see how our product form looks (figure 6.8).

We need to modify our `Products` controller's `save` method slightly, to take our tags into account. We need to save our relationship between the product and the tags. But

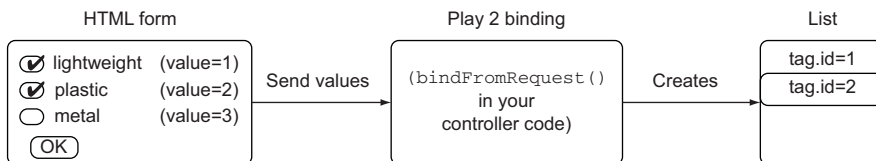


Figure 6.7 Binding HTML check boxes

our tag objects do not yet correspond to the Tag objects that we have in our datastore; they are new objects with only the ID set. We need to modify our action method to look up the tag, and set it. This can easily be done in our controller, as the following listing shows.

Listing 6.8 Product save method with tag relationship

```
public class Products extends Controller {
...
    public static Result save() {
        ... (binding and error handling)

        Product product = boundForm.get();

        List<Tag> tags = new ArrayList<Tag>();
        for (Tag tag : product.tags) {
            if (tag.id != null) {
                tags.add(Tag.findById(tag.id));
            }
        }
        product.tags = tags;
        product.save();
        ... (success message and redirect)
    }
}
```

We now need to display the tags when displaying a product. For this, we need to update our view so the correct tag check boxes are selected when a tag is present. We have already seen (in section 6.1.2) how our product prepopulates our product form:

```
productForm = productForm.fill(product);
```

This also loads the product's associated tags, so the action method doesn't need to be changed. Our goal can easily be achieved with a bit of code in our view. When rendering the view, we need to add a checked attribute to each check box if its associated tag is present in the productForm. The following line of code does exactly that:

```
@for(i <- 0 to 2) { @if(productForm("tags[" + i + "]").value!=null
    && productForm("tags[" + i + "]").value == "1") { checked }}
```

Because our product's associated tags might contain between zero and three tags, we need to iterate through them (the for loop in our code). If the tags contain the value

Figure 6.8 Edit product form with tags

1, then we preselect the check box. Don't worry if you don't fully understand this line; we'll take a closer look at template syntax in chapter 8.

We now need to apply this code for each check box we're displaying:

```
<div class="control-group">
  <div class="controls">
    <input name="tags[0].id" value="1" type="checkbox"
    @for(i <- 0 to 2) {
      @if(productForm("tags[" + i + "].id").value=="1"){ checked }
    }> lightweight
    <input name="tags[1].id" value="2" type="checkbox"
    @for(i <- 0 to 2) {
      @if(productForm("tags[" + i + "].id").value=="2"){ checked }
    }> metal
    <input name="tags[2].id" value="3" type="checkbox"
    @for(i <- 0 to 2) {
      @if(productForm("tags[" + i + "].id").value=="3"){ checked }
    }> plastic
  </div>
</div>
```

Again, in chapter 8, we'll see how we can refactor all this code to make it less repetitive and verbose.

We now can deal with related tags for our product. More importantly, we're now able to map back and forth from check boxes or select fields to our object model.

Let's see what happens if we want to bind a type that Play doesn't know about yet.

6.2.3 Custom data binders and formatters

As we have seen in the previous section, Play is able to bind most of the usual types automatically. But what about special types? Play allows you to define your own binder or your own formatter. There are three different ways of binding objects. You can bind objects via the URL or via URL parameters, and it's also possible to use a custom formatter that will transform your object when submitting a form.

PATH BINDERS

When requesting a URL with a named parameter in it, Play already performs some kind of binding. For example, in our application, we have:

```
/product/1111111111111
```

where 1111111111111 is our product EAN number. Internally, Play binds this value to the product's EAN number. This is done when declaring our URL in the route file (see chapter 5):

```
GET /product/:ean Product.details(ean: String)
```

Play looks for the `ean` type and converts it into the proper type.

In our case, it binds a string to another string. But Play can also deal with values such as primitives and primitive wrappers, based on information in the routes file. If we have the following route:

```
GET /product/:ean Product.details(ean: Int)
```

Play would make sure to convert

```
/product/1122334455
```

into an Integer with the value 1122334455. If we were to use aabbcc1122334455, Play would yield an error, because it can't convert aabbcc1122334455 into an Integer.

Therefore, there is a mechanism that allows Play to transform part of a URL (our URL parameter, a string) into an object. This means that we could, for example, automatically load and bind our Product objects based on the EAN number that is used in our URL.

Whenever we call

```
/product/1111111111111
```

we can have the matching product object passed to our controller, and we can have a details action method that looks like this:

```
public static Result details(Product product)
{
    Form<models.Product> productForm = form(models.Product.class);
    productForm = productForm.fill(product);
    return ok(edit.render(productForm));
}
```

The product should automatically be looked up using the product's EAN key

Figure 6.9 illustrates what we want to achieve: automatic binding of our Product object based on our EAN product number.

Play allows you to define your own binder for URL paths. This is called a *bindable path* in Play. In order to provide our own custom URL path binder, we need to create a new class that implements the play

.mvc.PathBindable interface, and our implementation must reference itself as a generic type argument:² if we create a class called ProductBinder, we have to implement play.mvc.PathBindable<ProductBinder>.

Because we want to bind the Product entity to the EAN number that is used in our URL, we need to implement the play.mvc.PathBindable interface on our Product class.

Open your editor and edit the Product.java class from the models directory.

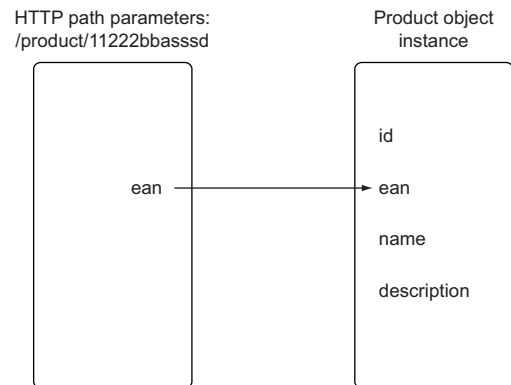


Figure 6.9 The binding process

² This might change in the next version of Play.

Listing 6.9 Project directory structure

```

warehouse
├── app
│   ├── controllers
│   ├── models
│   │   └── Product.java
│   └── views
├── conf
├── public
└── test

```

← The class that we need to edit

The code is simple; all we need to do is implement three methods. Like we said, our `Product` class implements the `play.mvc.PathBindable` interface and references itself (we have the class declaring a `Product`, and `play.mvc.PathBindable` references `Product` via the use of generics). The three methods we need to implement to satisfy the interface are:

- `T bind(String key, String txt)`
- `String unbind(String key)`
- `String javascriptUnbind()`

The `bind` method indicates how to bind our product from the URL. The `unbind` method tells Play what to display when a `Product` is referenced from the view, for example, in case of reverse routing (see chapter 5). In our case, we should display the product's EAN number. The method `javascriptUnbind` is used for Ajax calls. We also want to use the product's EAN number for that.

Let's add our three new methods. The `bind` method performs a lookup in the database, using the EAN number. The EAN number is passed via the `txt` parameter. That is the EAN number as it appears in the URL. The unbinding process is simple and consists of returning our `Product` instance's EAN number. The following listing shows our implementation.

Listing 6.10 Product class that is `PathBindable`-aware

```

...
import play.mvc.PathBindable;
import play.libs.F.Option;

public class Product implements PathBindable<Product> {
    ...

    @Override
    public Product bind (String key, String value) {
        return findByEan(value);
    }

    @Override
    public String unbind(String key) {
        return this.ean;
    }

    @Override

```

← We're telling Play that we're defining a new Binder to bind URL paths

← Binding—we're looking in the DB for a product with an EAN number equal to the one passed in our URL

← Unbinding—we're returning our raw value

```
public String javascriptUnbind() {
    return this.ean;
}
}
```

JavaScript unbinding—
we're returning our raw value

We now need to change our routes file to modify our `Product.edit()` route. Edit the routes file located in your `conf` directory.

Let's replace

```
GET /product/:ean controllers.Product.details(ean: String)
```

with

```
GET /products/:ean controllers.Products.details(ean: models.Product)
```

This is how we tell Play that we want to use our object. The automatic binding between the `Product` object and the EAN number in our URL path will be performed by the methods we have implemented.

We now need to change our `Products` controller with the following method:

```
public class Products extends Controller {
    ...
    public static Result details(Product product) {
        Form<Product> filledForm = productForm.fill(product);
        return ok(details.render(filledForm));
    }
    ...
}
```

The new edit method binds
directly to a product

Now that we've changed the method signature, we need to update its callers too. At this point, we only have one caller. In the products list template, `products/list.scala.html`, change the following line

```
<a href="@routes.Products.details(product.ean)">
```

to

```
<a href="@routes.Products.details(product)">
```

and we're done! We can now automatically bind a URL to an instance of `Product`. Calling `http://localhost:9000/products/1111111111` will show the product with EAN 1111111111. This simplifies the code for our controllers and centralizes the program logic that turns a path parameter into a `Product` instance. Any action method that requires a `Product` can now accept it as a parameter directly.

QUERY STRING BINDERS

Up to this point, we've seen that we can bind the URL path to our model objects. We can do exactly the same for URL parameters (the query string part of a URL). Let's take our product example. This time, we're going to bind to a `Product` object when we pass the product's EAN as a query parameter:

```
http://localhost:9000/products?ean=1
```

When Play encounters this URL, it should look up the `Product` in the database, based on the provided EAN. In order to achieve that, we have to implement the `play.mvc.QueryStringBindable` interface, similar to how we implemented `PathBindable` earlier. Because we want to look up `Product` objects, we'll implement the interface on our `Product` model class.

Edit the `Product` model class.

Listing 6.11 Project directory structure

```
warehouse
├── app
│   ├── controllers
│   ├── models
│   │   └── Product.java
│   └── views
├── conf
│   └── routes
├── public
└── test
```

The `Product` class we're going to edit

This time, we need to change it to implement the `play.mvc.QueryStringBindable` interface. Like `PathBindable`, this interface provides three methods we need to implement:

- `Option<T> bind(String key, Map<String,String[]> data)`
- `String unbind(String key)`
- `String javascriptUnbind()`

The `bind` method gives us access to the query parameters, both values and associated keys, and must return an `Option` object of type `T`. In our case, `T` is type `Product`. `Option` is the class that allows us to say that either we return something or we return nothing, giving a nicer alternative to returning `null`. The `data` parameter of the `bind` method gives us access to the query parameters. It's represented by a `Map` object that contains the query parameter names with their corresponding values.

Like before, `bind` indicates how to bind our product from the URL parameters, `unbind` tells Play what to display when a `Product` is referenced from the view, and `javascriptUnbind` is to support JavaScript. In our case, the logic is based on the `id` property of our `Product` class.

Let's implement the required methods in the `Product` model class. Please note that this is a simple example, and so it doesn't check for null values or other possible errors.

Listing 6.12 Product class that is `QueryStringBindable`-aware

```
...
import play.mvc.PathBindable;
import play.mvc.QueryStringBindable;

public class Product
    implements QueryStringBindable<Product> {
    ...
}
```

We're telling Play that we're defining a new Binder to bind query parameters

```

@Override
public Option<Product> bind(String key, Map<String, String[]> data) {
    return Option.Some(findByEan(data.get("ean")[0]));
}

@Override
public String unbind(String key) {
    return this.id;
}

@Override public String javascriptUnbind() {
    return this.id;
}
}

```

Binding on the product ID number—database lookup based on the product EAN

Unbinding—return our product EAN

JavaScript unbinding—return our product EAN

We need to declare a new route in our routes file that links the URL call to our controller. Open the routes file (located in your conf directory), and add the following route:

```
GET /products/ controllers.Product.details(ean: models.Product)
```

We're ready to accept URLs such as `/products?ean=1`. Modify the details method on the Products controller, if you haven't already done so in the previous section.

Listing 6.13 Project directory structure

```

warehouse
├── app
│   ├── controllers
│   │   └── Products.java
│   ├── models
│   │   └── Product.java
│   └── views
├── conf
│   └── routes
├── public
└── test

```

The Products controller we need to modify

The edit method becomes the same as when we used the path binder:

```

public static Result details(Product product){
    Form<Product> filledForm = productForm.fill(product);
    return ok(details.render(filledForm));
}

```

The new edit method binds directly to a product

As you can see, we can manipulate the `Product` object straight from our controller methods. This is quite nice if you want to reduce the size of your controller code. We can now automatically bind URL parameters (`id`, in our case) to an instance of `Product`. Calling `http://localhost:9000/products?ean=1` shows the product with `id` 1.

FORM FIELD BINDERS

Another form of binding occurs when a form is submitted or displayed. Values are mapped back and forth between the `Form` object and the forms in the views. Inside the form object, the data is stored as objects, but on the client side we can only display their string representations. This is why customized form field binders can be useful.

Form field binders in Play are also referred to as *formatters*, because they format data to and from controllers.

Formatters, both predefined and custom, are always used to map Strings to objects. You can register a Formatter using the `register()` method of `play.data.format.Formatters`. Let's do that now with a custom formatter for dates, by extending the `SimpleFormatter` class. We have to register the formatter at application startup in the `Global` object (or a class that extends `GlobalSettings` and overrides the `onStart` method). For that we need to create a `Global.java` file in the `app` folder. The `Global` object is part of Play and defines methods that are called during the application lifecycle (`onStart`, `onStop`, `onError`, and so forth). The `Global.java` file content is shown in the following listing.

Listing 6.14 Registering a DateFormatter class

```
import play.*;
import play.libs.*;
import java.util.*;
import models.*;
import play.data.format.Formatters;
import play.data.format.Formatters.*;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class Global extends GlobalSettings {

    public void onStart(Application app) {
        Formatters.register(Date.class,
            new SimpleFormatter<Date>() {
                private final static String PATTERN = "dd-MM-yyyy";

                public Date parse(String text, Locale locale)
                    throws java.text.ParseException {
                    if(text == null || text.trim().isEmpty()) {
                        return null;
                    }
                    SimpleDateFormat sdf =
                        new SimpleDateFormat(PATTERN, locale);
                    sdf.setLenient(false);
                    return sdf.parse(text);
                }

                public String print(Date value, Locale locale){
                    if(value == null) {
                        return "";
                    }
                    return new SimpleDateFormat(PATTERN, locale)
                        .format(value);
                }
            }
        );
    }
}
```

The `onStart` method is called when Play starts our application

Our `Global` object is part of Play's lifecycle

Register our new `Formatter`, extending the `SimpleFormatter` abstract class and specifying that the formatter applies to `Date` objects

Create a `Date` object based on our date-format-binding process

Create a string representation of our `Date` object based on our date-format-unbinding process

Because we're extending the `SimpleFormatter` class, we have to implement two methods: the `parse()` method and the `print()` method. The `parse()` method converts a `String` to a `Date` object (the binding process), whereas the `print()` method converts a `Date` to a `String` (the unbinding process). An incoming date string, such as 11-02-2008, will now be converted to a `Date` object representing that date. Similarly, an outgoing `Date` object will be converted to a "dd-MM-yyyy"-formatted string.

As you can see, this is straightforward to declare. Now every time we encounter a `Date` object, Play knows how to bind a date and send it back as `String`. But there are cases when you don't want the formatting to be global, or you want the date format to be enforced and/or different.

If we were to add a date to our product, the user would have to enter the date as 21-01-2012 in a product details input field using the formatter we defined earlier. Similarly, the date will be shown as 21-01-2012 in the details form. In our object model, however, it will be a `Date` and not a `String`; the formatter handles the conversion between `String` and `Date`.

For example, what if you want to have multiple date representations? For example, suppose you want the user to input 11-02-08 on the main page and 11/02/08 on the login screen. Play allows you to control the binding behavior with annotations, including custom annotations. Let's transform our date formatter into an annotation-based formatter by creating the following:

- An annotation
- A formatter that extends Play's `AnnotationFormatter`

Let's start with our annotation:

Listing 6.15 The `DateFormat` annotation, `/app/utils/DateFormat.java`

```
package utils;

import play.data.Form;
import java.lang.annotation.*;

@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Form.Display(name = "format.date", attributes = {"value"})
public @interface DateFormat {
    String value();
}
```

The value will be our date format

Indicates the annotation applies to fields only

The annotation is to be applied at runtime

This Play annotation indicates that the `value` parameter—this is used to display a validation error message in case of binding failure

The annotation is simple and is mainly a holder for our date format. The `Display` annotation defined on the `DateFormat` annotation holds metadata for Play. When a date can't be bound because of an incorrect date format, Play uses that metadata to display an error message on the form. The annotation tells Play to use the `format.date` message key and that value can be used as a parameter for the message.

Let's now define our formatter. We have to indicate that we want an annotation formatter. This is done by extending `Formatters.AnnotationFormatter`.

Listing 6.16 The `AnnotationDateFormat` class

```
public static class AnnotationDateFormat
    extends Formatters.AnnotationFormatter<DateFormat, Date> {

    public Date parse(DateFormat annotation, String text,
                     Locale locale)
        throws java.text.ParseException {
        if(text == null || text.trim().isEmpty()) {
            return null;
        }
        SimpleDateFormat sdf = new
            SimpleDateFormat(annotation.value(), locale);
        sdf.setLenient(false);
        return sdf.parse(text);
    }

    public String print(DateFormat annotation, Date value,
                       Locale locale) {
        if(value == null) {
            return "";
        }
        return new SimpleDateFormat (annotation.value(), locale)
            .format(value);
    }
}
```

Extends `Formatters.AnnotationFormatter` so we're annotation aware—the `Formatter` applies to `DateFormat` annotation and to `Date` object

Create a `Date` object based on our date format value annotation-binding process

We're taking the date format from our annotation

Create a string representation of our `Date` object based on our date format annotation-unbinding process

We're taking the date format from our annotation

Because we're extending the `Formatters.AnnotationFormatter` abstract class, as before with the `SimpleFormatter` class, we have to implement two methods: the `parse()` method and the `print()` method. And as before, the `parse()` method binds the date string representation to a concrete `Date` object, whereas the `print()` method does the exact opposite. But this time, the `Formatters.AnnotationFormatter` is annotation aware, and our `print()` and `parse()` methods take an extra parameter: our `DateFormat` annotation. We use it to access the configured pattern.

As we've seen in the previous section, we need to register our `AnnotationDateFormat` in our `Global.java` class. Listing 6.17 shows how to do that.

Listing 6.17 Registering our AnnotationDateFormatter in the Global.java file

```
public class Global extends GlobalSettings {
    public void onStart(Application app) {
        ...
        Formatters.register(Date.class, new AnnotationDateFormatter());
    }
}
```

Register our
AnnotationDateFormatter

Now we can now annotate any object property on a model class to indicate that we want to bind and unbind Date from a Form object using the following syntax:

Listing 6.18 Object model using the DateFormat annotation

```
public Person {
    @DateFormat("MM-dd-yyyy")
    public Date birthDate;
}
```

Now in our view, each Person's birth date will be displayed as MM-dd-yyyy. When submitting a form, the birthDate attribute will contain a Date object with the expected date. Note that in practice, the example we've seen is already part of Play. You don't need to code it and you can use the @DateTime annotation defined in the play.data.format.Formats class.

6.3 Body parsers

We've looked at request body data mapped to form objects and vice versa, but we haven't explored how the raw request bodies are processed. This step is the job of body parsers.

Each incoming HTTP POST and PUT request contains a body. This body may be single part or multipart, and may contain XML, JSON, binary data, or any other type as specified in the request's Content-Type header. A body parser will parse the body (hence the name) into a Java object. Further operations, such as formatting, can then take place. Figure 6.10 illustrates this process.

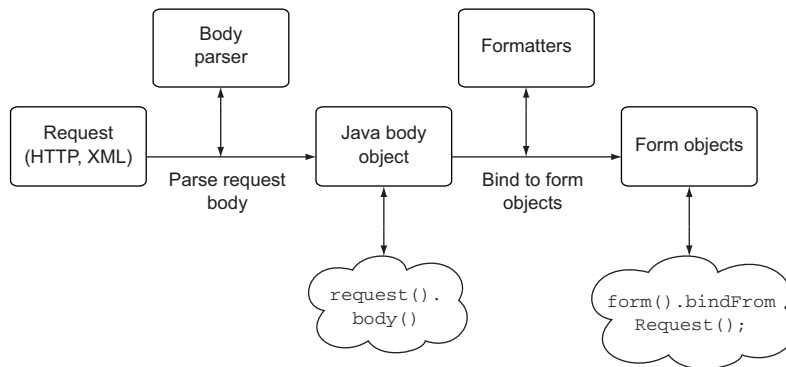


Figure 6.10 Body parser interaction with an incoming request

Body parsers “decode” the request and transform it into objects that can then be used by the other Play components. Because a JSON body is parsed differently to an XML body, Play uses pluggable body parsers. Different content types, therefore, have specific body parsers that can translate arbitrary incoming data into something Play can understand.

Using body parsers, it’s possible to directly manipulate the object obtained from a request. The interaction is done through a body-parser API.

6.3.1 *The body-parser API*

All body parsers must generate a `play.mvc.Http.RequestBody` value. This value can then be retrieved via `request().body()`. The following listing shows how to use the API.

Listing 6.19 Accessing the request body

```
public static Result index() {  
    RequestBody body = request().body();  
    return ok("Here is the body we received: " + body);  
}
```

← Accessing the request body

You can specify the `BodyParser` to use for a particular action using the `@BodyParser.Of` annotation. This means that we can tell Play to use a specific body parser to parse the request and then access the body request to read the value. Listing 6.20 shows an example in which we’re specifying a JSON body parser and reading the result via the `body.asJson()` method.

Listing 6.20 Specifying a specific body parser to use

```
@BodyParser.Of(BodyParser.Json.class)  
public static Result index() {  
    RequestBody body = request().body();  
    return ok("We expected to get json: " + body.asJson());  
}
```

← We output the JSON value on the console

All body parsers will give you a `play.mvc.Http.RequestBody` value. From this body object you can retrieve the request body content in the most appropriate type. In our previous example, we used the `asJson()` method. But if we’d used an XML body parser, we would’ve used the `asXML` method.

REQUESTBODY METHODS MAY RETURN NULL `RequestBody` methods such as `asText()` or `asJson()` will return `null` if the parser used can’t handle that content type. For example, in an action method annotated with `@BodyParser.Of(BodyParser.Json.class)`, calling `asXml()` on the generated body will return `null`.

Some parsers can provide a more specific type than `Http.RequestBody` (a subclass of `Http.RequestBody`). You can automatically cast the request body into another type using the `as()` helper method. An example is shown in the following listing.

Listing 6.21 Using the as() helper method

```
@BodyParser.Of(BodyLengthParser.class)
public static Result index() {
    BodyLength body = request().body().as(BodyLength.class);
    return ok("Request body length: " + body.getLength());
}
```

Using as() to automatically bind the request body to a BodyLength class

If you don't specify a body parser, Play will use the Content-Type header to determine which built-in parser to use:

- text/plain—String, accessible via asText()
- application/json—JsonNode, accessible via asJson()
- text/xml—org.w3c.Document, accessible via asXml()
- application/form-url-encoded—Map<String, String[]>, accessible via asFormUrlEncoded()
- multipart/form-data—Http.MultipartFormData, accessible via asMultipartFormData()
- *Any other content type*—Http.RawBuffer, accessible via asRaw()

Please note that if the requested body type isn't available, the method will return null. The following listing shows an example of how the asText() method can be used.

Listing 6.22 Using the body.asText() method

```
public static Result save() {
    RequestBody body = request().body();
    String textBody = body.asText();

    if(textBody != null) {
        return ok("Got: " + text);
    } else {
        return badRequest
            ("Expecting text/plain request body");
    }
}
```

Text-based body parsers (such as text, JSON-, XML-, or URL-encoded forms) use a maximum content length because they have to load all the content into memory. There is a default content length limit of 100 KB.

OVERRIDING THE DEFAULT MAX CONTENT-LENGTH The default content size can be defined in application.conf: `parsers.text.maxLength=128K`.

You can also specify a maximum content length via the @BodyParser.Of annotation, for example to only accept 10 KB of data:

```
@BodyParser.Of(value = BodyParser.Text.class, maxLength = 10 * 1024)
public static Result index() {
    ...
}
```

Custom body parsers must be written in Scala, because they process incoming data incrementally using Scala's Iteratee I/O.

But Play has body parsers for typical web content types such as JSON and binary data, and these can be reused to create body parsers in Java.

Now that we understand how to receive and transform data, we need to be able to ensure the data is valid.

6.4 Validation

We've seen how we can translate user data to our own object model and vice versa. But we also need to ensure the data is valid in the context of our application by meeting certain criteria. For example, some fields might be mandatory. The binding process can't guarantee that data is valid. Play allows you to define constraints to catch invalid submitted data. It also provides detailed feedback to the client in the case of invalid data. In this section we'll see how we can define constraints on the data the user inputs.

6.4.1 Using the built-in validators

Play validates the data once it is bound to the domain model. Play uses JSR-303³ and Hibernate validators for this step. Defining constraints is as simple as annotating the object model. For example, ensuring that the user inputs an EAN number in our product form is as simple as adding the `@Required` annotation on the `Product` object:

```
import play.data.validation.Constraints.*;
...

public Product extends Model {
    ...
    @Required
    public String ean;
}
```

← Indicates this field is required

To ensure the user input complies with the constraint defined by the annotation, we need to check the `hasError` method on our `Form` object. Validation occurs immediately after binding and registers errors in the form as they are found. The validation process sets any validation errors to the `Form` that was bound. Validation errors contain the relevant `i18n` error message key, the field name, and a potential parameter list to display with the error message. Listing 6.23 shows the `save` method seen in section 6.1.3, with the validation check added.

Listing 6.23 Save action method with validation

```
public static Result save() {
    Form<models.Product> productForm =
        form(models.Product.class).bindFromRequest();
    models.Product newProduct = productForm.get();
}
```

← Create a Form object from the current request

← Bind the Product object from the form

³ JSR-303 is a Java specification that defines a metadata model and API for JavaBean validation based on annotations; see <http://jcp.org/en/jsr/detail?id=303>.

```

If a validation error occurred → if (productForm.hasErrors()) {
    return ok(edit.render(productForm));
} else {
    newProduct.saveOrUpdate();
    return redirect(routes.Product.all());
}
                                ← Rerender our form
                                ← Save or update our current Product object
                                ← Redirect to our "view all products" page

```

As you can see, validating a form is straightforward.

Play comes with a lot of commonly used validations, and these are detailed in table 6.1.

Table 6.1 Built-in Play validation annotations

Name	Description
@Required	A Play-specific validation indicating the field must be non-null or, in the case of <code>Strings</code> and <code>Collections</code> , not null and not empty. Works with any object.
@Min	Indicates the minimum this number should be; for example, <code>@Min(1)</code> .
@Max	Indicates the maximum this number should be; for example, <code>@Max(2)</code> .
@MinLength	Defines a minimum length for a string field.
@MaxLength	Defines a maximum length for a string field.
@Pattern	Checks if the annotated string matches the regular expression pattern.
@ValidateWith	Uses a custom validator (see section 6.4.3).
@Email	Checks whether the specified string is a valid email address.

Play uses an implementation of the JSR-303 specification,⁴ better known as *Bean Validation*, to perform the validation. The standard also comes with its own set of validation annotations that covers most common use cases.

Table 6.2 shows some built-in annotations that come with Bean Validation. They can all be found in the `javax.validation` package.

Table 6.2 Built-in Hibernate validation annotations

Name	Description
@Null	Indicates that the field should be <code>null</code> (but it can be empty).
@NotNull	Indicates that the field cannot be <code>null</code> .
@NotEmpty	Checks if the string is not <code>null</code> nor empty.
@AssertTrue	Asserts that this field is a <code>Boolean</code> that resolves to <code>true</code> .

⁴ Hibernate Validator, to be precise.

Table 6.2 Built-in Hibernate validation annotations (*continued*)

Name	Description
<code>@AssertFalse</code>	Asserts that this field is a Boolean that resolves to <i>false</i> .
<code>@Min</code>	Indicates the minimum this number should be; for example, <code>@Min(1)</code> .
<code>@Max</code>	Indicates the maximum this number should be; for example, <code>@Max(2)</code> .
<code>@DecimalMin</code>	Indicates the minimum this decimal number should be; for example, <code>@Min(1.1)</code> .
<code>@DecimalMax</code>	Indicates the maximum this decimal number should be; for example, <code>@Max(4.2)</code> .
<code>@Size</code>	Indicates range this number should be in; for example, <code>@Size(min=2, max=4)</code> .
<code>@Digits(integer=, fraction=)</code>	Checks whether the property is a number, having up to <code>{integer}</code> digits and <code>{fraction}</code> fractional digits.
<code>@Past</code>	Checks whether the annotated date is in the past.
<code>@Future</code>	Checks whether the annotated date is in the future.
<code>@Pattern(regex=)</code>	Checks if the annotated string matches the <code>{regex}</code> regular expression pattern.
<code>@Valid</code>	Performs validation recursively on the associated object.
<code>@Email</code>	Checks whether the specified string is a valid email address.
<code>@Length(min=, max=)</code>	Validates that the annotated string is between <code>{min}</code> and <code>{max}</code> (inclusive).

These annotations are self-explanatory, but further information can be found in the documentation for Bean Validation (<http://beanvalidation.org>).

6.4.2 Partial validation

A common use case is having multiple validation constraints for the same object model. Because we're defining our constraint on the object model, it's normal to have multiple forms that refer to the same object model. But these forms might have different validation constraints. To illustrate this use case, we can imagine a simple wizard in which the user inputs a new product in two steps:

- 1 The user enters the product name and submits the form.
- 2 The user enters the product EAN number and the description.

We could validate the product's name during step 2, but displaying an error message for the product name at that point would be weird. Fortunately, Play allows you to perform partial validation. For each annotated value, we need to indicate at which step it

applies. We can do that with the help of the groups attribute from our annotations. Let's change our Product model class to do that:

```
public Product extends Model {
    public interface Step1{}           ← Define first step
    public interface Step2{}           ← Define second step

    @Required(groups = Step1.class)    ← This constraint only applies to first step
    public String name;
    @Required(groups = Step2.class)    ← This constraint only applies to second step
    public String ean;
}
```

We now need to indicate which step we're at. This is done when binding from the request, specifying Step1:

```
// We re//strict the validation to the Step1 "group"
Form<Product> productForm =
    form(Product.class, Product.Step1.class).bindFromRequest();
if(filledForm.hasErrors()) {
    ...
}
```

When binding, tell validator we're only interested in constraints that apply to first step

We can do the same for Step2. This is useful if the model object is used on different forms and has different validation constraints.

6.4.3 Creating a custom validator

Play gives you the ability to add your own validators. This is useful if you need to perform custom validation. You can implement your own validator in different ways:

- Using ad hoc validation—this method is the quickest and simplest one
- Using `@ValidateWith` and defining your own `Validator` class
- Defining a new JSR 303 annotation and defining a new `Validator` class

We'll take a closer look at each of these approaches, starting with ad hoc validation.

AD HOC VALIDATION

You can define an ad hoc validation by adding a `validate` method to your model object. Play will invoke the `validate` method of every object model. For example, to validate our EAN number, we can add a `validate` method on our `Product` object model. The `validate` method must return a `String` or `null`. If it returns a `String`, it must either contain the validation error message or the `IllegalArgumentException` key for one. A `null` return value indicates there are no errors.

An EAN is a 13-digit number. Our `validate` method must check that the `String` EAN variable contains exactly 13 digits. Listing 6.24 shows you how to do that.

Listing 6.24 EAN number ad hoc validation adding the validate method

```

public Product extends Model {
    ...
    public String ean; public String validate() {
        String pattern = "[0-9]{13}$";
        Pattern regex = Pattern.compile(pattern);
        return name !=null &&
            regex.matcher(name).matches()?null:"Invalid ean number";
    }
}

```

The validate method returns null in case of success, or the error message otherwise

This regex pattern means "a number character, 13 times"

Return the error message if the input doesn't match; null otherwise

The validate method is not practical if you have several attributes to validate, but it's the easiest and quickest way to add simple validation constraints. Let's take a look at how we can do more complex validation.

PLAY @VALIDATEWITH

Using `@ValidatWith`, we can have fine-grained control over validation. Let's use the EAN number as an example again. First, we need to annotate the `ean` attribute with the `@Constraints.ValidateWith.ValidateWith` annotation. The `@ValidatWith` method takes a class as parameter: the class that will do our custom validation.

We need to implement a class that extends the `Play Constraints.Validator` class. In our `Validator` class, we need to implement the `isValid` and `getErrorMessageKey` methods. `isValid()` tests the field's validity, and `getErrorMessageKey()` returns the `il8n` message key for the error message. The following listing shows how to validate the EAN number using `@ValidatWith`.

Listing 6.25 EAN number validation using the `@ValidatWith` annotation

```

package models;

import play.data.validation.Constraints;
...

public class Product implements PathBindable<Product>,
    QueryStringBindable<Product> {

    public static class EanValidator
        extends Constraints.Validator<String> {

        @Override
        public boolean isValid(String value) {
            String pattern = "[0-9]{13}$";
            return value != null && value.matches(pattern);
        }

        @Override
        public F.Tuple<String, Object[]> getErrorMessageKey() {
            return new F.Tuple<String, Object[]>("error.invalid.ean",

```

Our custom validator class must extend the Play abstract `Validator` class

Our implementation of the `isValid` method

Return true if the string has exactly 13 digits

```

        new Object[]{});
    }
}
...
@Constraints.Required
@Constraints.ValidateWith(value=EanValidator.class,
    message="must be 13 numbers")
public String ean;
...
}

```

← **getErrorMessageKey**
returns `il8` message key and
potential argument values

← **Indicates we want to
validate using our
custom validator class**

Using `@ValidateWith`, we now have a custom, fine-grained, reusable validation mechanism. This approach to validation is Play-specific, but using the JSR-303 standard, there is a way to specify validation logic in a more portable way. Let's see how that works.

JSR-303 CUSTOM ANNOTATION AND VALIDATOR

This approach is the JSR-303 standard way to add custom validation. It's not specific to Play and could be used in other, non-Play applications. This approach requires two steps:

- Define an annotation and reference a `Validator` class
- Define the `Validator` class that is referenced by our annotation

Let's define an `@EAN` annotation that will be used to make sure our EAN number has exactly 13 digits. We need to declare a new annotation and annotate that with the `@javax.validation.Constraint` annotation later. The `@Constraint` annotation references a `Validator` class that we need to implement next. The following listing shows you how to create the annotation.

Listing 6.26 Custom JSR-303 EAN annotation

```

@Constraint(validatedBy = EanValidator.class)
@Target( { FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface EAN {

    String message() default "error.invalid.ean";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}

```

← **Indicate that it is a validation
annotation that should use the
EanValidator class to validate the
value annotated by this annotation**

← **Our annotation targets the field value**

← **We always compute the
annotation at runtime**

← **The default error message—
can be overwritten when
setting the annotation**

← **The groups we want
to apply the
annotation to during
partial validation—
see section 6.4.2 for
more on partial
validation**

**Payload is a standard JSR-303
property currently not used by Play**

We now need to define a `Validator` that will hold the business logic to validate our EAN number. Our validator has to implement the JSR-303 `ConstraintValidator` interface, and we need to implement the `isValid` method. The `isValid` method returns a `Boolean` that indicates whether the value is valid. The following listing shows the `EanValidator` implementation.

Listing 6.27 Custom JSR-303 validator

```
public static class EanValidator extends Constraints.Validator<String>
    implements ConstraintValidator<EAN, String> {
    final static public String message = "error.invalid.ean";
    public EanValidator() {}

    @Override
    public void initialize(EAN constraintAnnotation) {}

    @Override
    public boolean isValid(String value) {
        String pattern = "[0-9]{13}$";
        return value != null && value.matches(pattern);
    }

    @Override
    public F.Tuple<String, Object[]> getErrorMessageKey() {
        return new F.Tuple<String, Object[]>(message,
            new Object[]{});
    }
}
```

The `ConstraintValidator` class identifies the validator; it extends Play's `Constraints.Validator` class.

This is called when the validator is initialized

The business logic—if regular expression doesn't match, it isn't valid

We can now annotate our `Product` object model with the `@EAN` annotation:

```
public Product extends Model {
    ...
    @EAN
    public String ean;
}
```

And our EAN field will automatically be validated. As you can see, it's easy to add new validation logic to our object models. Play allows you to validate your objects with your own custom validation code, so you can validate against a database if you wish to.

6.4.4 *Displaying the validation errors on the form*

Nice—we can now validate our forms. But we also need to present the cause of the validation errors to our users. This is trivial to do; the error messages are stored in our `Form` object. By passing the `Form` object to our views, we can easily display the error messages. For example, in our controller's `save()` action, we check the form for errors and pass it along to the view:

```
Form<Product> boundForm = productForm.bindFromRequest();
if (boundForm.hasErrors()) {
    return ok(details.render(productForm));
}
```

← We're passing our product form to the view

In our view, we can access the form object and display the errors. We can access the error message using the `errors()` method on the form or on a specified field. If you want to display all the validation errors, iterate over the form errors using the `errors()` method:

```
<p>Please fix the following errors<ul>
@for(error <- productForm.errors() ) {
    <li>@error.message</li>
}
<ul></p>
```

If you want to access the errors per field, then you can use the `errors()` method on the field value. For example, if we want to access the potential validation error for the product name field:

```
@productForm("name").errors()
```

This returns a list. But using the built-in `mkString` method to join strings, we can have a comma-separated list of errors per field using the following code:

```
@productForm("name").errors().mkString(", ")
```

For our current form, there's no need to use these methods, because the form helpers already include errors. But there's no magic to the form helpers; they render the errors using exactly this technique, as you can see in the following listing, which shows the template for the field helper.

Listing 6.28 Source code for the field helper template

```
@(elements: views.html.helper.FieldElements)

@import play.api.i18n._
@import views.html.helper._

@***** *
Generate input according twitter Bootstrap rules *
*****@

<div class="clearfix @elements.args.get('_class')
  @if(elements.hasErrors) {error}"
  id="@elements.args.get('_id').getOrElse(elements.id + "_field")">
  <label for="@elements.id">@elements.label(elements.lang)</label>
  <div class="input">
    @elements.input
    <span class="help-inline">
      @elements.errors(elements.lang).mkString(", ")
    </span>
    <span class="help-block">
```

```

        @elements.infos(elements.lang).mkString(", ")
      </span>
    </div>
  </div>

```

We now know how to validate and to report any validation errors back to the user. We're now able to submit any user inputs, process them, and reject any invalid data. Let's see about a more complicated type of input: file uploads.

6.5 File uploads

It's time to go back to our current application. There's something missing that could make our paperclips more identifiable; we need to visually recognize them. Let's add a picture. For that we need to modify the `Product` object model slightly: see the following listing.

Listing 6.29 Adding a picture to the `Product` object model

```

public class Product {
  ...
  @Constraints.Required
  @Constraints.ValidateWith(EanValidator.class)
  public String ean;
  @Constraints.Required
  public String name;
  public String description;
  public byte[] picture;
  ...
}

```

← Add this line—
a byte array will
hold our picture

We now need to give our users a chance to upload a picture for a product; we need to add an HTML input file element to our form. To do that, we need to edit the existing `product.scala.html` template that we created in section 6.1.1, add an HTML file input element, and display a picture if one exists using an HTML `img` element. We also need to change the form encoding type to send multipart data.

The form now looks like the one shown in the following listing.

Listing 6.30 “Product create” form with picture uploading

```

@(productForm: Form[Product])
@import helper._
@import helper.twitterBootstrap._

@main("Product form") {
  <h1>Product form</h1>
  @helper.form(action = routes.Products.save(),
    'enctype -> "multipart/form-data") {
    <fieldset>
      <legend>Product (@productForm("name").valueOr("New"))</legend>
      @helper.inputText(productForm("ean"))
      @helper.inputText(productForm("name"))
      @helper.textarea(productForm("description"))
      @helper.inputFile(productForm("picture"))
    </fieldset>
  }
}

```

← The HTML form is
now multipart

← Our input file
HTML element

```

@if(!productForm("picture").valueOr("").isEmpty()) {
<div class="control-group">
  <div class="controls">
    
    </div>
  </div>
}
...
}
}

```

← The **img HTML tag** used to display the picture

Once the form is submitted, we need to save the picture sent by the user. Unfortunately, because of the way HTTP file uploads work, Play is unable to bind the file directly to the picture field, so we'll need to modify our `save()` method on the `Product` controller.

In order to obtain the file that was uploaded by the user, we need to access the part of the HTTP body that holds the file. We've seen how we can use the `body-parser` API to do that in section 6.3.1; we call `request().body().asMultipartFormData()`.

Once we access the body, we can request the file and convert it to a byte array. Then we can save our product and the associated picture to the datastore. We could also access the filename and its content type through the `FilePart` object that we obtain from the multipart body, but we currently have no need for that in our application.

The following listing shows how to read and save the file.

Listing 6.31 Obtaining and saving uploaded files

```

import static play.mvc.Http.MultipartFormData;
...
public static Result save() {
  Form<Product> boundForm = productForm.bindFromRequest();
  if(boundForm.hasErrors()) {
    flash("error", "Please correct the form below.");
    return badRequest(details.render(boundForm));
  }
  Product product = boundForm.get();
  MultipartFormData body = request().body().asMultipartFormData();
  MultipartFormData.FilePart part = body.getFile("picture");

  if(part != null) {
    Get file → File picture = part.getFile();
    try {
      product.picture = Files.toByteArray(picture);
    } catch (IOException e) {
      return internalServerError("Error reading file upload");
    }
  }
}

```

← **Binds form as a multipart form so we can access submitted file**

← **Requests picture FilePart—this should match the name attribute of the input file in our form (for example, input name=picture)**

← **A utility method copies file contents to a byte[]**

```

List<Tag> tags = new ArrayList<Tag>();
for (Tag tag : product.tags) {
    if (tag.id != null) {
        tags.add(Tag.findById(tag.id));
    }
}
product.tags = tags;
product.save();
flash("success",
    String.format("Successfully added product %s", product));
return redirect(routes.Products.list(1));
}

```

Save product
as usual

We're now able to save our product and the associated picture to our datastore. If you wanted to store the file some other way, such as on the filesystem, you could easily write code that does that; once you have the `File` object, it handles like any other file.

Now, we still need a way to *display* the product picture on our form. This is easily done by adding a new method on our `Product` controller. Let's add a `picture()` method. This method takes an EAN number, uses it to look up the matching product in the database, and uses the image byte array to generate a `Result`.

The following listing shows how the `picture()` method is implemented.

Listing 6.32 Our picture method

```

public static Result picture(String ean) {
    final Product product = Product.findByIdEan(ean);
    if(product == null) return notFound();
    return ok(product.picture);
}

```

As usual when we implement a new action method, we need to edit our routes file to link the picture method to a URL. To request a picture for a product, we want to call the `/picture/ean` URL. We add the following line to our routes file:

```
GET /picture/:ean controllers.Products.picture(ean: String)
```

In our `product.html.scala` file, we already added the image:

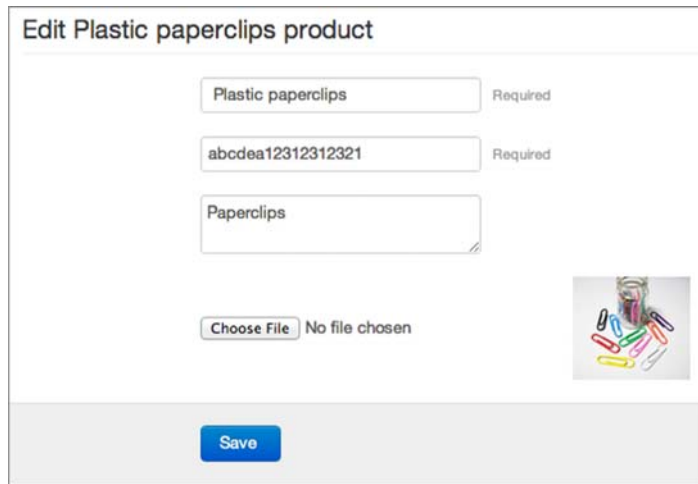
```

@if(!productForm("picture").valueOr("").isEmpty()) {
    
}

```

We're now able to display our product picture. Figure 6.11 shows what you should now see in the application.

Now that we're able to handle file uploads, we've covered all aspects of handling user input!



Edit Plastic paperclips product

Plastic paperclips Required

abcdea12312312321 Required

Paperclips

Choose File No file chosen

Save

Figure 6.11 Our paperclip picture

6.6 Summary

This chapter has been dense, and we saw a lot. We learned how to handle user input—from a simple form submission to a more complex file submission. We also learned how all this was possible through the use of body parsers. We saw in detail how the Play binding process transforms user-submitted data into data that our application can understand. We also learned how we could extend that binding process; we know how to customize and create our own binders.

Once we were ready to process user data, we saw how to validate that data. We also saw how we could create our own validator if needed. We then saw how to handle invalid data and to report those errors back to the user. We finished the chapter with a concrete example of how to handle file submission. We're now able to handle any data type, process it, and validate it when needed. In the next chapter, we'll see how to model our warehouse application.

RELATED MANNING TITLES



Play for Scala

Covers Play 2

by Peter Hilton, Erik Bakker, Francisco Canedo

ISBN: 978-1-617290-79-4

328 pages, \$49.99

October 2013



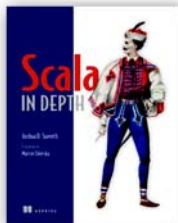
Functional Programming in Scala

by Paul Chiusano and Rúnar Bjarnason

ISBN: 978-1-617290-65-7

325 pages, \$44.99

June 2014



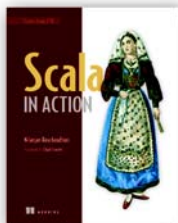
Scala in Depth

by Joshua D. Suereth

ISBN: 978-1-935182-70-2

304 pages, \$49.99

May 2012



Scala in Action

by Nilanjan Raychaudhuri

ISBN: 978-1-935182-75-7

416 pages, \$44.99

April 2013



Reactive Design Patterns

by Roland Kuhn and Jamie Allen

ISBN: 978-1-617291-80-7

325 pages, \$49.99

April 2015

For ordering information go to www.manning.com

Play FOR JAVA

Leroux • de Kaper

For a Java developer, the Play web application framework is a breath of fresh air. With Play you get the power of Scala's strong type system and functional programming model, and a rock-solid Java API that makes it a snap to create stateless, event-driven, browser-based applications ready to deploy against your existing infrastructure.

Play for Java teaches you to build Java-based web applications using Play 2. This book starts with an overview example and then explores each facet of a typical application by discussing simple snippets as they are added to a larger example. Along the way, you'll contrast Play and JEE patterns and learn how a stateless web application can fit seamlessly in an enterprise Java environment. You'll also learn how to develop asynchronous and reactive web applications.

What's Inside

- Build Play 2 applications using Java
- Leverage your JEE skills
- Work in an asynchronous way
- Secure and test your Play application

The book requires a background in Java. No knowledge of Play or of Scala is assumed.

Nicolas Leroux is a core developer of the Play framework.
Sietse de Kaper develops and deploys Java-based Play applications.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/PlayforJava



“Helps you transition to more productive ways to build modern web apps.”

—From the Foreword by James Ward, Typesafe

“The easiest way to learn the easiest web framework.”

—Franco Lombardo
Molteni Informatica

“The definitive guide to Play 2 for Java.”

—Ricky Yim, DiUS Computing

“A good cocktail of theory and practical information.”

—Jeroen Nouws, XTl

“An excellent tutorial on the Play 2 framework.”

—Lochana C. Menikarachchi
PhD, University of Connecticut

ISBN 13: 978-1-617290-90-9
ISBN 10: 1-61729-090-4



9781617290909