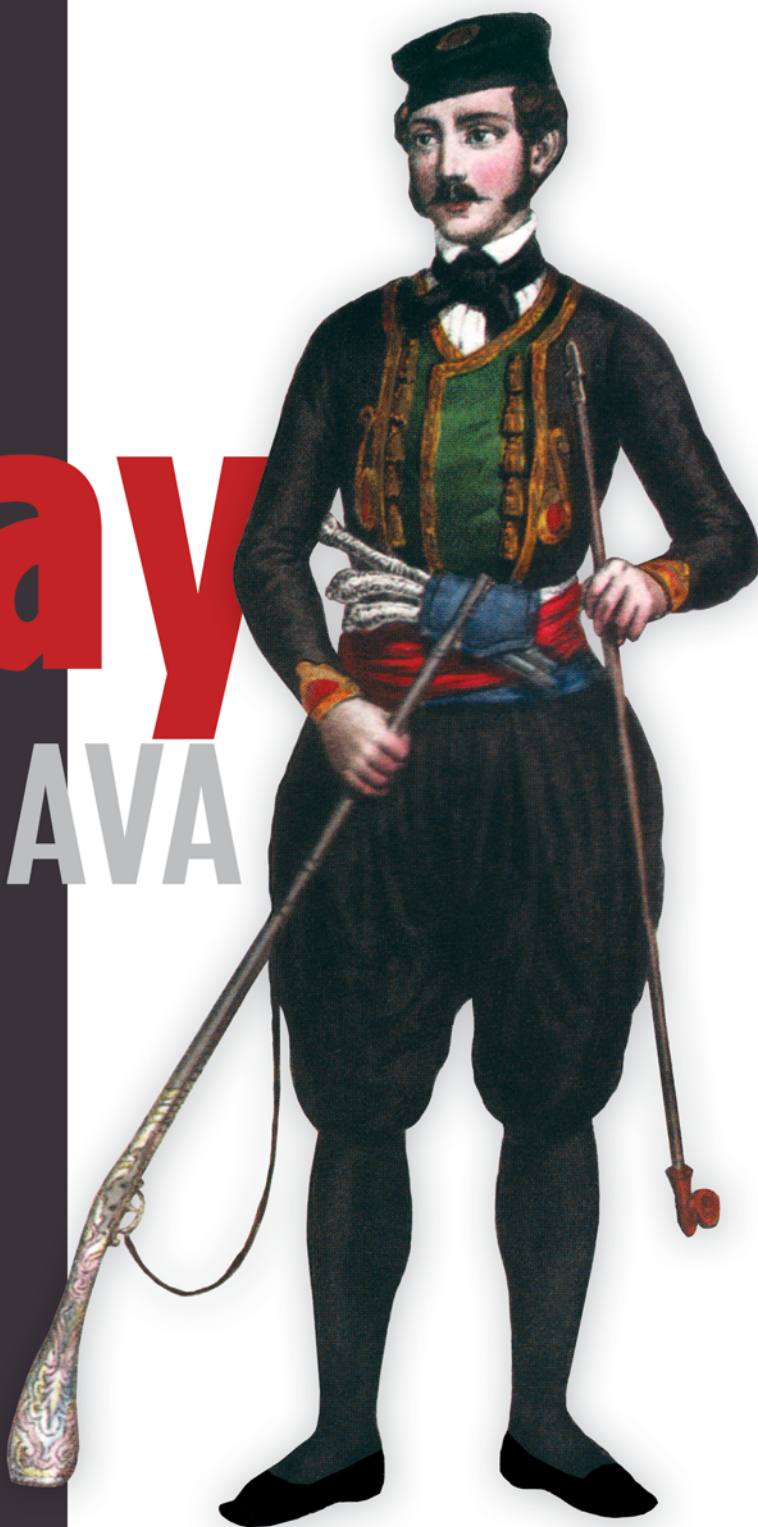


Play FOR JAVA

Nicolas Leroux
Sietse de Kaper

FOREWORD BY James Ward





Play for Java
by Nicolas Leroux
Sietse de Kaper

Chapter 3

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*.

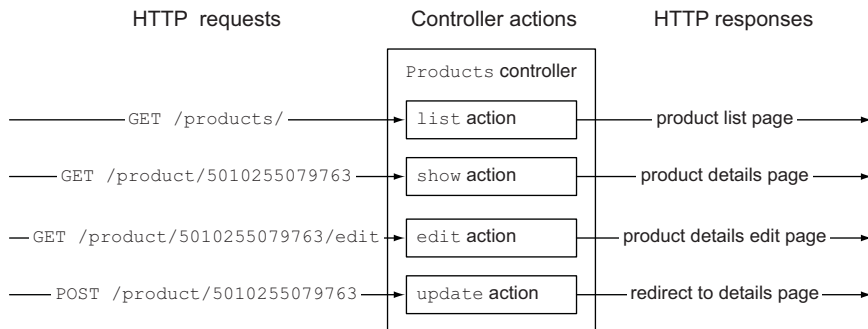


Figure 3.1 Requests routed to actions

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.

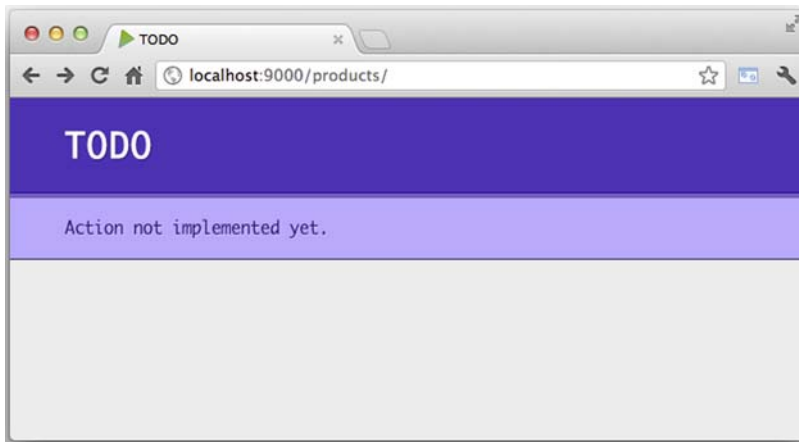


Figure 3.2 Play's TODO placeholder at /products

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("1111111111111", "Paperclips 1",
            "Paperclips description 1"));
        products.add(new Product("2222222222222", "Paperclips 2",
            "Paperclips description "));
        products.add(new Product("3333333333333", "Paperclips 3",
            "Paperclips description 3"));
        products.add(new Product("4444444444444", "Paperclips 4",
            "Paperclips description 4"));
        products.add(new Product("5555555555555", "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*:

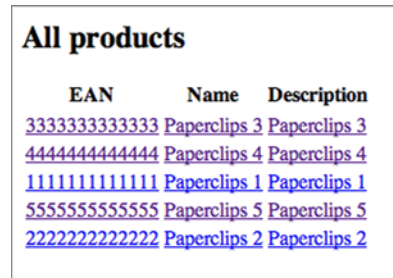
```
@(products: List[Product])
```

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._
```

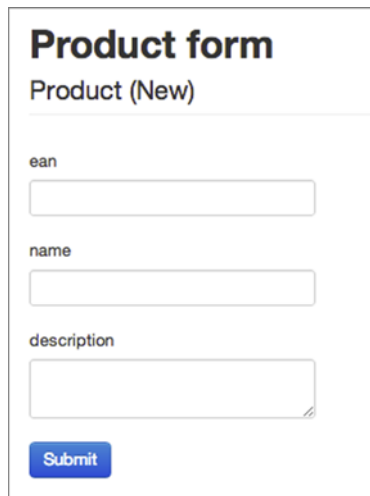
A screenshot of a web form titled "Product form" with the subtitle "Product (New)". The form contains three input fields: "ean", "name", and "description". Below the "description" field is a blue "Submit" button. The form is styled with a light gray border and a white background.

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.

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, XTI

“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