**Installing Play**

* Download zip file from the website http://www.playframework.org/.
* Unzip it to a location of your choice.
* Update PATH environment variable to point to the extracted folder, which contains the command-line tool: play!.
* Java 6 or a higher version is installed and available for use.

What's interesting at this point is that the play! tool is actually starting an SBT console.

Creating a brand new application in a directory,

* Use the play! Command-line tool with a parameter (named new) followed by the name of the new application (play-jbook).
* The tool will ask to specify whether our application is a Scala or Java application.
* At first, a new directory will be created with the name of the application, that is, play-jbook.

The whole structure will be inside this directory having following folders:

* **app:** root of all the server-side source files, whatever type they are (Java, Scala, server-side templates, compiled scripts, and so on).
  + At creation, only two subfolders will be created: **controllers** and **views**.
* **conf:** This folder is for the files meant to configure the application itself.
  + External services, or whatever the application could need at runtime.
* **project:** As SBT is used behind the curtains, the project folder is meant to contain all of the necessary files to configure this tool.
* **public:** Contain all of the external files that can be accessed by the **clients**.
  + Such as stylesheets, images, and JavaScript source files.
  + A dedicated folder has been created for each type as well.
* **test:** This folder contain all test files with some examples provided.

**To generate the Eclipse project configuration**

Play! has already configured everything in order to generate the project and module files required by the most popular IDEs. by simply invoking the eclipse:

C:\Project\passlite2\passcore>**play eclipse**

This will generate all the specific files necessary to configure an Eclipse project. Now we can open Eclipse and import the project into it.

**To run the project:**

Enter the play! console and type run[play run]

C:\Project\passlite2\passcore>play run

To open our browser and request this brand new server by going to the URL http://localhost:9000/.

An action in Play! Framework 2 is the business logic that defines an HTTP request.

It's a simple method that has a defined route declaring it as the code to be executed when a matching request arrives.

The action methods are declared in a container (a class in Java) that extends the Controller type (called as controller). Roughly, a controller is a bunch of methods that respond to the HTTP requests.

**Template filename pattern:**

<template-name>.scala.<content-type>

Out of the box types are .html, .xml, and .txt, but Play! 2 is extensible enough to enable us to add new ones.

Play! 2 will detect these files based on the pattern and will compile them into functions that will be available for the controllers or other templates.

As a template will be compiled into a function, it can accept parameters just as any other function in Scala.

* The filename is the function name parameters.
  + So, logically, this is the first line of all templates.
* These parameters are declared in exactly the same way as for Scala functions.
  + @(title : String, notificationItems: List[models.database.commons.Notification])
* We need to use the magic character (@), because it's Scala code in a template.

The Play! 2 compiler will make available to us a function with the following signature:

def function(level:Int, items:Seq[String]):Html

Play! 2 has worked out that the file is an HTML template, from the second extension (.html) of the template name—that is, the Html result type of the function.

The content of a template is inserted into the file directly after the parameters' declaration.

Scala is used to create some relevant content based on the server-side data using parameters.

**Getting data from Model and showing into View.**

**package** models;

**public** **class** User {

**public** String name;

}

**public** **class** UserController **extends** Controller{

**static** **Form<User>** ***userForm*** = form(User.**class**);

**public** **static** Result test(){

Map<String, String> toBind = **new** HashMap<String, String>();

toBind.put(“name”, “Manish”);

User user = ***userForm***.bind(toBind).get();

**return** *ok*(data.render(user));

}

}

The test action is doing the following tasks:

* It creates a container of data (toBind) representing the structure: a simple dictionary.
* It adds one piece of data that is keyed by name and a dummy value.
  + The **key** is well chosen in order to match the **user's name** field name.
* It uses the bind method on the form using the data container.
  + This will feed the data to the underlying binder.
* It calls get on the resulting (filled) form to retrieve the structure expected.
* It calls our template with the result to show what has been bound.

Extracting Data from View(client/browser) and sending it to server:



Option:

An Option is a Scala structure that defines an absent or present data in a type-safe way.

A given value will be of the type **some**, whereas an absent one will be of the type **None**.

Option[User] enables us to re-use the same template for two different cases:

* To show the view when the optional user is present
* To present the form otherwise

Route:

GET /data/post controllers.UserController.post()

**public** **class** UserController **extends** Controller{

**static** Form<User> *userForm* = form(User.**class**);

**public** **static** Result post(){

Map<String, String> toBind = **new** HashMap<String, String>();

toBind.put(arg0, arg1);

User user = *userForm*.bindFromRequest().get();

**return** *ok*(data.render(Scala.Option(user));

}

}

**bindFromRequest**

We have used a new method on the form, bindFromRequest, which is meant to retrieve the data needed by the form somewhere in the request.

bindFromRequest method is able to look up data of several types and in several places.

Depending on the content type, the HTTP method, and the encoding, this binding will look for data in:

* **The query string**: This is our case, because we have sent a GET request.
* **The body**: For POST/PUT methods. There are several options here because the content could be a map of URL parameters, a multipart, or JSON encoded.

In fact, the binding method will look everywhere to gather all the data and then it will retain the ones declared in the form definition to find which it's expecting.

Scala

This is Play!'s specific class, which contains a lot of methods to interact from Java to Scala.

Here we used it to create Scala. Option by giving it the options user (results in Some(user)) or null (correctly cast to have None of the underlying type User).

Data-Validation:

Java has a Java Specification Request (JSR) defined for validation, JSR 303, wherein how constraints can be added to Java models—using annotations—is specified.

So, Play! 2 takes advantages of this JSR and enables us to use it to validate data.

It also defines custom validators that are missing in the specification.

The validation information is available on the browser side as well.

This is possible because of HTML helpers

We find everything we need in the package play.api.validation,

Especially the static methods available in the Constraints class that defines the most common annotations.

But, sometimes we might need ones from the JSR; for instance, in the User model we imported Valid from the javax.validation package.

**Content**

Common content types include JSON, XML, HTML, images or even videos to be stored and streamed.

Play! 2 provides a clean way of dealing with such content types with the help of body parsers.

A body parser is a component that is responsible for **parsing** the body of an HTTP request as a stream to be converted into a predefined structure.

An **action** in Play! 2 represents the piece of software that is able to handle an HTTP request; therefore, they are the right place to use a body parser.

In the Java API, an action is allowed to be annotated with the Of annotation available in the BodyParser class.

* This annotation declares the expected type of request routedto it, and
* It requires a parameter that is the class of the parser that will be instantiated to parse the incoming request's body.

There are a plenty of predefined body parsers that can be used to handle our requests, and they are all defined in the BodyParser class as static inner classes.

We never had to add such annotations to our actions!!! Play! 2, being a web framework; takes advantage of HTTP; will look into the header for content-type to find the right parser to apply.

So annotations are mandatory. Let’s see their use in the bindFromRequest method for example.

content type was set in the header by the HTML forms themselves.

bindFromRequest method's job was to look for data according to the provided content type.

bindFromRequest method will:

* Gather data as URL-form encoded data, if any
* Gather data from parts (if the content type is multipart-data)
* Gather data as JSON-encoded, if any
* Gather data from the query string (that's why GET requests were working as well)
* Fill in the form's data with all of them (and validate)

Then what is the worth of annotations;

* They allow new types of parsers, but they can also enforce certain actions' requests to match a given content type.
* They allow us to extend or narrow the length of the body that can be handled.
  + By default, 100 K are accepted, and this can be either configured (parsers.text.maxLength=42K) or passed as an argument to the annotation.

**Rendering contents**

Play! 2 server is able to render different resources in different ways rather than simply providing HTML pages.

actions builder are the static methods available in the play.mvc.Results.java class such as **ok**, **redirect, badRequest,** and **unauthorized.**

These methods (ok, redirect, badRequest, and unauthorized) have been overloaded several times in order to accept several representations. The following are some examples:

* Content: This takes content that is of the base type of classic string representations such as Html, Xml, and Txt. This is also the result-type of a rendered template.
* String: This will be rendered as is, as a plain text content (an overloaded version of the method accepts the encoding as a second argument).
* JsonNode: This is trivial. If we create an instance of such a class, we'll have our resource serialized as application/json.
* InputStream: This is a convenient way to dump a stream into a response body (accepts chunks for an HTTP-chunked encoded connection).
* File: This helps us avoid typing new FileInputStream(...) in InputStream. This accepts the file and will deal with the stream for us.

Handling multipart content types

The HTTP protocol is ready to accept, from a client, a lot of data and/or large chunks of data, at once.

Much requests will have a body that can hold several data pieces formatted differently and attributed with different names.

A way to achieve this is to use a specific encoding type: multipart/form-data.

In a very few lines, we declared a new action that expects requests to be multipart encoded containing at least two parts, where the first is a map of data (no matter how this map is encoded) to fill in imageForm (essentially a caption). The second will be the image part.

After binding the request with the form and verifying that no errors have occurred, we can move to the body content in order to recover the binary data that was sent along with its metadata: the file content, its content type, its length, and so on.

That was quite an intuitive thing to do – asking the body to be parsed as a

multipart/multidata and and get it as an Http.MultipartFormData object, which has a getFile method that returns an Http.MultipartFormData.FilePart value.

To understand why we didn't specify a body parser, recall that Play! 2 is able, most of the time, to discover which method fits best by itself.

The Http.MultipartFormData.FilePart type is not only allowing us to recover the content as a file, but also its key inthe multipart body, its filename header, and (especially) its content type.

Having all of these things in hand, we are now able to check the content-type validity against the image's enum, and to store the image by getting the file path of the provided file.