

# Presentation patterns for web applications with Play! Framework

Alberto García García  
< *agg180@alu.ua.es* >



May 20, 2014

# TABLE OF CONTENTS

INTRODUCTION

PLAY! FRAMEWORK

PATTERNS IN PLAY!

CONCLUSIONS

# INTRODUCTION

# OUTLINE

## INTRODUCTION

- Trends

- Challenges

- Addressing the challenges

# TRENDS

- ▶ Enterprises's needs lead the market.
- ▶ Offering services: SOA wins.
- ▶ The web changes the status quo.
- ▶ SOA is not web compliant.
- ▶ Exposing services through the web requires extra effort.
- ▶ The game changes: new possibilities and challenges.

# CHALLENGES

- ▶ Real time data has to be pushed.
- ▶ Huge amounts of data.
- ▶ Need for scalability and integration.
- ▶ Easy integration and accessibility.
- ▶ Interoperability.

# ADDRESSING THE CHALLENGES

- ▶ Embrace the internet.
  - ▶ HTTP Protocol
  - ▶ HTML5
  - ▶ XML/JSON
  - ▶ Javascript
  - ▶ CSS
- ▶ Paradigm shift: client-side.
- ▶ Simplicity.
- ▶ A framework to rule them all.
- ▶ **Patterns for enterprise applications.**





# OUTLINE

## PLAY! FRAMEWORK

What is Play! Framework?

RESTful Architecture

Project layout

# WHAT IS PLAY! FRAMEWORK?

- ▶ A web framework focused on:
  - ▶ Simplicity.
  - ▶ Productivity.
  - ▶ Scalability.
  - ▶ Designed for the modern web.
    - ▶ Concentrate on server-side.
    - ▶ Delegate AMAP to the client.
  - ▶ Embrace internet standards.
  - ▶ Java and Scala.
  - ▶ RESTful architecture web applications.
  - ▶ Model-View-Controller.

# RESTFUL ARCHITECTURE

- ▶ Implemented using HTTP and REST principles.
- ▶ Representational state transfer (REST) principles:
  - ▶ Uniform interface.
  - ▶ Stateless.
  - ▶ Caching.
  - ▶ Layers.
  - ▶ Code on demand.
- ▶ Goals:
  - ▶ Performance.
  - ▶ Scalability.
  - ▶ Portability.
  - ▶ Reliability.
  - ▶ SIMPLICITY.

# PROJECT LAYOUT

app	→ Application sources
├ assets	→ Compiled asset sources
│ └ stylesheets	→ Typically LESS CSS sources
│ └ javascripts	→ Typically CoffeeScript sources
├ controllers	→ Application controllers
├ models	→ Application business layer
├ views	→ Templates
build.sbt	→ Application build script
conf	→ Configurations files and other non-compiled resources
├ application.conf	→ Main configuration file
├ routes	→ Routes definition
public	→ Public assets
├ stylesheets	→ CSS files
├ javascripts	→ Javascript files
├ images	→ Image files
project	→ sbt configuration files
├ build.properties	→ Marker for sbt project
├ plugins.sbt	→ sbt plugins including the declaration for Play its dependencies
lib	→ Unmanaged libraries dependencies
logs	→ Standard logs folder
├ application.log	→ Default log file
target	→ Generated stuff
├ scala-2.10.0	
│ └ cache	
│ └ classes	→ Compiled class files
│ └ classes_managed	→ Managed class files (templates, ...)
│ └ resource_managed	→ Managed resources (less, ...)
│ └ src_managed	→ Generated sources (templates, ...)
test	→ source folder for unit or functional tests



# OUTLINE

## PATTERNS IN PLAY!

### Model-View-Controller

The MVC application model: Models

Request/Response path

### Model

Object Relational Mapping

### View

Template View

Composite View

### Controller

Front Controller

# THE MVC APPLICATION MODEL

- ▶ Models in app/models
  - ▶ Java/Scala classes.
  - ▶ Data + Operations, mainly object-oriented.
  - ▶ Business logic and storage.

## A MODEL EXAMPLE (MODELS/USER.JAVA)

```
1 package models;
2
3 @Entity
4 public class User extends Model {
5     @Id
6     public String name;
7     @Required
8     public String pass;
9
10    public User (String name, String pass) {
11        this.name = name;
12        this.pass = pass;
13    }
14
15    public static Finder<String, User> find = new Finder<String,
16        User>(String.class, User.class);
17
18    public static List<User> all() {
19        return find.all();
20    }
21 }
```



# THE MVC APPLICATION MODEL: VIEWS

- ▶ Views in app/views
  - ▶ HTML/XML/JSON/Scala templates.
  - ▶ Directives as placeholders for data.
  - ▶ Render models to user interfaces.

# A VIEW EXAMPLE (VIEWS/INDEX.SCALA.HTML)

```
2  @(title : String , users : List[User])
4  <!DOCTYPE html>
6  <html>
8      <head>
10         <title>Play! Hello world</title>
12     </head>
14     <body>
16         <header>
18             <h1>@title</h1>
20         </header>
22         <section>
24             <ul>
26                 @for(u <- users) {
28                     <li>@u.name</li>
30                 }
32             </ul>
34         </section>
36     </body>
38 </html>
```

# THE MVC APPLICATION MODEL: CONTROLLERS

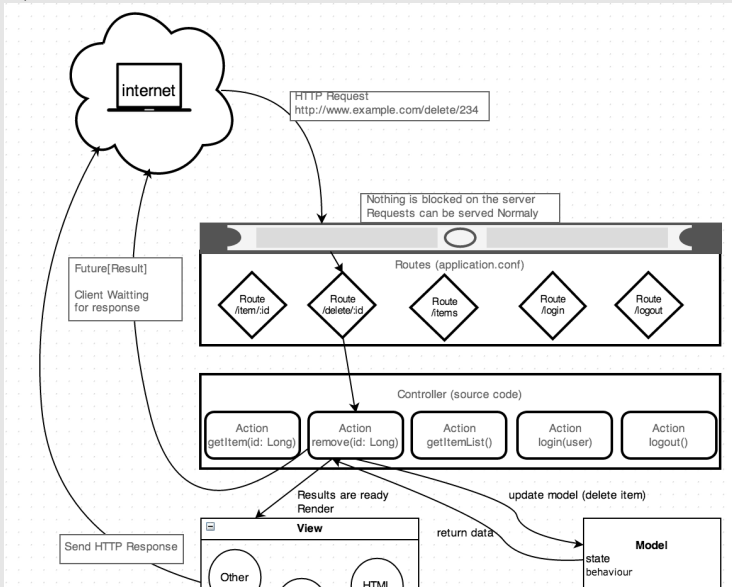
- ▶ Controllers in app/controllers
  - ▶ Java/Scala classes.
  - ▶ Methods as actions, mainly procedural.
  - ▶ Receive requests, act (update models + render views) and response.

# A CONTROLLER EXAMPLE

## (CONTROLLERS/APPLICATION.JAVA)

```
1 package controllers;  
2  
3 import models.User;  
4 import play.*;  
5 import play.data.*;  
6 import play.mvc.*;  
7 import views.html.*;  
8  
9 public class Application extends Controller  
10 {  
11     public static Result index()  
12     {  
13         return ok("Hello , world!", index.render(User.all()));  
14     }  
15 }
```

# REQUEST/RESPONSE FLOW



# THE HTTP REQUEST AND THE ROUTER (EXAMPLE)

- ▶ Suppose that we receive the HTTP Request: **GET /**
- ▶ The server processes it, looks for the proper action to response the GET / request in **conf/routes**.
- ▶ The called action is: **Application.index()**

```
1 # Routes
  # All application routes (Higher priority routes first)
3 # ----
  # Home page
5 GET      /          controllers.Application.index()
7 # Login
  GET      /login     controllers.Application.login()
9 POST     /login     controllers.Application.authenticate()
11 #Logout
  GET      /logout    controllers.Application.logout()
```

# A CONTROLLER EXAMPLE

## (CONTROLLERS/APPLICATION.JAVA)

```
1 package controllers;  
  
3 import models.User;  
import play.*;  
5 import play.data.*;  
import play.mvc.*;  
7 import views.html.*;  
  
9 public class Application extends Controller  
{  
11     public static Result index()  
    {  
13         return ok(index.render("Hello , world!", User.all()));  
    }  
15 }
```

## A MODEL EXAMPLE (MODELS/USER.JAVA)

```
1 package models;
3 @Entity
4 public class User extends Model {
5     @Id
6     public String name;
7     @Required
8     public String pass;
9
10    public User (String name, String pass) {
11        this.name = name;
12        this.pass = pass;
13    }
14
15    public static Finder<String, User> find = new Finder<String,
16        User>(String.class, User.class);
17
18    public static List<User> all() {
19        return find.all();
20    }
21 }
```



## A VIEW EXAMPLE (VIEWS/INDEX.SCALA.HTML)

```
1  @(title : String , users: List[User])
2
3  <!DOCTYPE html>
4
5  <html>
6      <head>
7          <title>Play! Hello world</title>
8      </head>
9      <body>
10         <header>
11             <h1>@title</h1>
12         </header>
13
14         <section>
15             <ul>
16                 @for(u <- users) {
17                     <li>@u.name</li>
18                 }
19             </ul>
20         </section>
21     </body>
22 </html>
```

# THE END RESULT

## Hello World!

- Kim Jong-Un
- Putin
- Obama

# OBJECT RELATIONAL MAPPING

- ▶ Need for persistence (objects outlive the application).
- ▶ Persistence by means of a database.
  - ▶ **Relational** (RDBMS)
  - ▶ Object (ODBMS)
- ▶ Gap between domain model and the relational database.
- ▶ The Object-Relational impedance mismatch.
  - ▶ Granularity
  - ▶ Inheritance
  - ▶ Identity
  - ▶ Associations
  - ▶ Data navigation
- ▶ Logical representation to atomized one to store in a DB.

# OBJECT RELATIONAL MAPPING TOOLS

- ▶ Object Relational Mapping tools are a possible solution.
- ▶ Provide simple ways to determine the mapping.
  - ▶ XML configuration files.
  - ▶ Annotations in the classes.
- ▶ Provide data query and retrieval facilities.
- ▶ All that glitters is not gold...
  - ▶ Pros: Simplicity, dramatically decrease the amount of code.
  - ▶ Cons: Higher abstraction drawbacks...
    - ▶ Performance issues.
    - ▶ Poor database design.
- ▶ Play! uses Ebean as its ORM of choice.

# ORM: ANNOTATED JAVA MODEL

```
1 package models;
2
3 @Entity
4 public class Post extends Model {
5
6     @Id
7     public Long id;
8
9     @Constraints.Required
10    public String title;
11
12    @Formats.DateTime(pattern="dd/MM/yyyy")
13    public Date postedAt;
14
15    public String content;
16
17    @ManyToOne
18    public User author;
19
20    @OneToMany(mappedBy="post", cascade=CascadeType.ALL)
21    public List<Comment> comments;
22 }
```

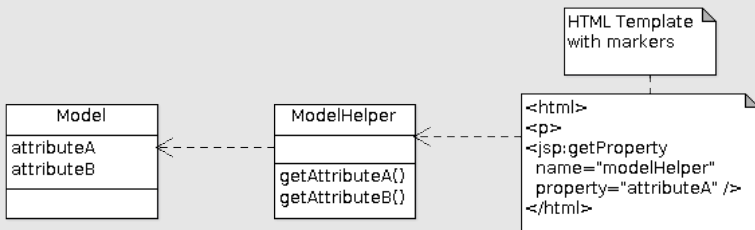
# ORM: USAGE

```
1 User user = new User("test@test.com", "Test", "test");  
2 user.save();  
  
4 User user = User.find.where().eq("email", "test@test.com").  
    findUnique();  
  
6 User.find.ref("test@test.com").delete();
```

ORM Hate by Martin Fowler [Fow12]

# TEMPLATE VIEW

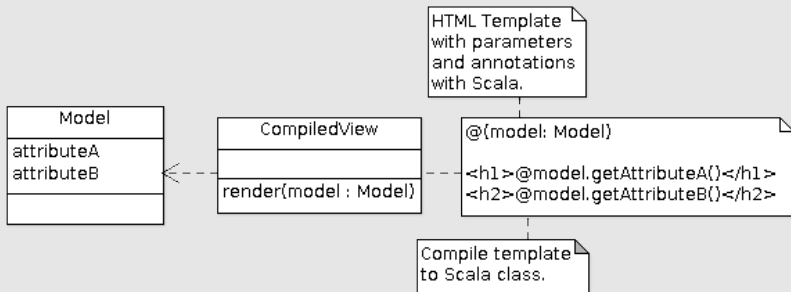
- *"Renders information into HTML by embedding markers in an HTML page"[Fow02]*



- Pros: Lot of power and flexibility in presentation.
- Cons: Messy code, difficult to maintain, need helpers.

# TEMPLATE VIEW

- ▶ *The template with annotations is compiled to a Scala.class with a render() method with the template parameters.*



- ▶ The controller calls the render method of the view.
- ▶ The view communicates with the model (parameter).

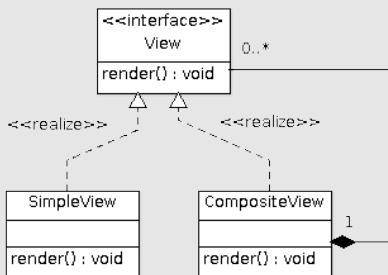


# TEMPLATE VIEW EXAMPLE

```
1  @(title : String , users : List[User])
2
3  <!DOCTYPE html>
4
5  <html>
6      <head>
7          <title>Play! Hello world</title>
8      </head>
9      <body>
10         <header>
11             <h1>@title</h1>
12         </header>
13
14         <section>
15             <ul>
16                 @for(u <- users) {
17                     <li>@u.name</li>
18                 }
19             </ul>
20         </section>
21     </body>
22 </html>
```

# COMPOSITE VIEW

- ▶ *A view is built from other views that combine into a composite whole, managing the content and the layout independently.*



- ▶ Pros: Modularity, reuse.
- ▶ Cons: Performance, maintainability.

# COMPOSITE VIEW

- ▶ *A sample simple view: [simpleview.scala.html](#)*

```
1  @(someModel: Model)
2
3  @compositeView(someModel) {
4    <header>
5      <hgroup>
6        <h1>Model data</h1>
7        <h2>@someModel.doSomething()</h2>
8      </hgroup>
9    </header>
10 }
```

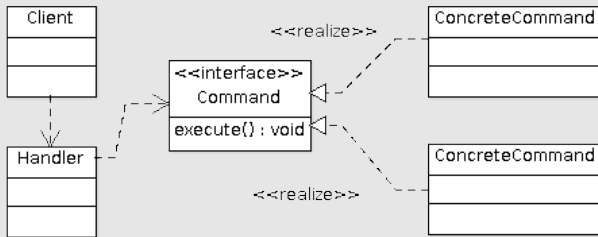
# COMPOSITE VIEW

- ▶ *A composite view: `compositeView.scala.html`*

```
@(someModel: Model)(simpleView: Html)
2 <html>
   <head>
4     <title>Composite View Example</title>
   </head>
6   <body>
      @simpleView
8
      <section id="main">
10        @someModel.showSomething()
      </section>
12   </body>
</html>
```

# FRONT CONTROLLER PATTERN

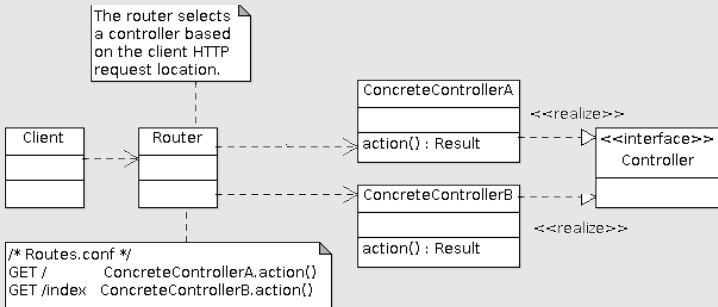
- *"Consolidates all request handling by channeling requests through a single handler object" [Fow02]*



- Pros: Centralized control, Thread safety, Configurability.
- Cons: Possible performance issues, Maintenance costs.

# FRONT CONTROLLER IN PLAY!

- ▶ The router (handler) selects a controller (command) and a particular action (execute) depending on the HTTP request.



- ▶ `Routes.conf` file determines the location-action relationship.
- ▶ Actions return a result that holds the HTTP Response.

# THE ROUTER

- The conf/routes file is compiled to a Router class.

```
1 # Routes
  # All application routes (Higher priority routes first)
3 # ~~~~
  # Home page
5 GET      /          controllers.Application.index()

7 # Login
  GET      /login     controllers.Application.login()
9 POST     /login     controllers.Application.authenticate()

11 #Logout
  GET      /logout    controllers.Application.logout()
```












- ▶ Web programming is evolving (again).
  - ▶ Scalability.
  - ▶ Client-side.
  - ▶ Frameworks to deal with complexity.
- ▶ Patterns are not just theoretical models.
  - ▶ Applied in modern applications.
  - ▶ Guides, not fixed templates.
  - ▶ With great power, comes great responsibility.
- ▶ Play! applies patterns for modern applications.
  - ▶ Easy learning curve.
  - ▶ Deep enough for complex applications.
  - ▶ Supported by Typesafe (Scala creators).
  - ▶ Not perfect.
    - ▶ Difficult upgrades.
    - ▶ Documentation is not so great.
    - ▶ Not the most popular.

# REFERENCES

-  Martin Fowler, *Patterns of enterprise application architecture*, Addison-Wesley Professional, 2002.
-  Martin Fowler, *Orm hate*,  
<http://martinfowler.com/bliki/OrmHate.html>, May 2012.
-  Nicolas Leroux and Sietse de Kaper, *Play for java*, Manning Publications, 2014.
-  Erik Bakker Peter Hilton and Francisco Canedo, *Play for scala*, Manning Publications, 2014.
-  Alexander Reelsen, *Play framework cookbook*, Packtpub Publications, 2014.