Presentation patterns for web applications with Play! Framework

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

May 13, 2014

1

TABLE OF CONTENTS

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**.

INTRODUCTION
000

PLAY! FRAMEWORK
000

PATTERNS IN PLAY!
0000000000000000000

CONCLUSIONS

REFERENCES

PLAY! FRAMEWORK

OUTLINE

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.

11

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 resour
  └ 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
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
```

12

PATTERNS IN PLAY!

OUTLINE

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)

```java
package models;

@Entity
public class User extends Model {
  @Id
  public String name;
  @Required
  public String pass;

  public User (String name, String pass) {
      this.name  = name;
      this.pass  = pass;
  }

  public static Finder<String, User> find = new Finder<String,
    User>(String.class, User.class);

  public static List<User> all() {
    return find.all();
  }
}
```

16

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)

```
@(title: String, users: List[User])

<!DOCTYPE html>

<html>
    <head>
        <title>Play! Hello world</title>
    </head>
    <body>
      <header>
        <h1>@title</h1>
      </header>

      <section>
        <ul>
          @for(u <- users) {
            <li>@u.name</li>
          }
        </ul>
      </section>
    </body>
</html>
```

18

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.

19

A CONTROLLER EXAMPLE
(CONTROLLERS/APPLICATION.JAVA)

```java
package controllers;

import models.User;
import play.*;
import play.data.*;
import play.mvc.*;
import views.html.*;

public class Application extends Controller
{
    public static Result index()
    {
        return ok("Hello, world!", index.render(User.all());
    }
}
```

20

REQUEST/RESPONSE FLOW

internet

HTTP Request
http://www.example.com/delete/234

Nothing is blocked on the server
Requests can be served Normaly

Future[Result]

Client Waitting
for response

Routes (application.conf)

| Route /item/:id | Route /delete/:id | Route /items | Route /login | Route /logout |

Controller (source code)

| Action getItem(id: Long) | Action remove(id: Long) | Action getItemList() | Action login(user) | Action logout() |

Results are ready
Render

update model (delete item)

Send HTTP Response

View

Other   HTML

return data

Model
state
behaviour

21

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)

```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("Hello, world!", index.render(User.all());
       }
15 }
```

A MODEL EXAMPLE (MODELS/USER.JAVA)

```java
package models;

@Entity
public class User extends Model {
  @Id
  public String name;
  @Required
  public String pass;

  public User (String name, String pass) {
      this.name  = name;
      this.pass  = pass;
  }

  public static Finder<String, User> find = new Finder<String,
    User>(String.class, User.class);

  public static List<User> all() {
    return find.all();
  }
}
```

24

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

```
@(title: String, users: List[User])

<!DOCTYPE html>

<html>
    <head>
        <title>Play! Hello world</title>
    </head>
    <body>
      <header>
        <h1>@title</h1>
      </header>

      <section>
        <ul>
          @for(u <- users) {
            <li>@u.name</li>
          }
        </ul>
      </section>
    </body>
</html>
```

25

THE END RESULT

Hello World!

- Kim Jong-Un
- Putin
- Obama

OBJECT RELATIONAL MAPPING

- a

TEMPLATE VIEW

▶   *"Renders information into HTML by embedding markers in an HTML page"[Fow02]*
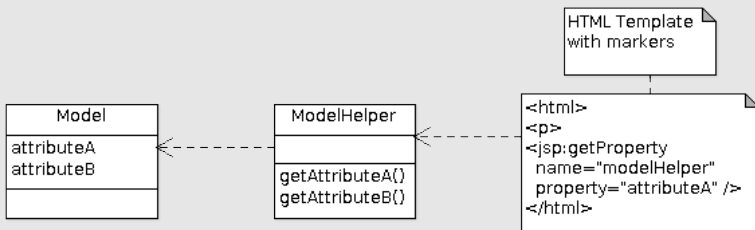
▶ Pros: Lot of power and flexibility in presentation.
▶ Cons: Messy code, difficult to maintain, need helpers.

28

TEMPLATE VIEW

► *The template with annotations is compiled to a Scala.class with a render() method with the template parameters.*
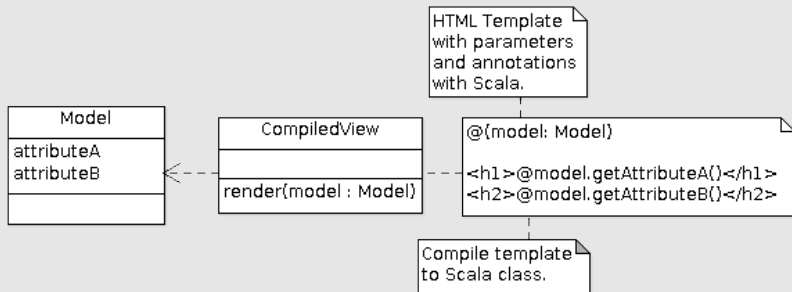
► The controller calls the render method of the view.
► The view communicates with the model (parameter).

COMPOSITE VIEW

► *A view is built from other views that combine into a composite whole, managing the content and the layout independently.*
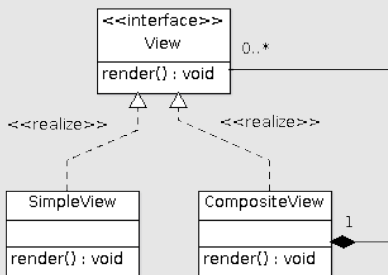
► Pros: Modularity, reuse.
► Cons: Performance, maintainability.

COMPOSITE VIEW

▶ *A sample simple view: simpleview.scala.html*

```
@(someModel : Model)

@compositeView (someModel) {
  <header>
    <hgroup>
      <h1>Model data</h1>
      <h2>@someModel. doSomething ()</h2>
    </hgroup>
  </header>
}
```

31

COMPOSITE VIEW

►   *A composite view: compositeView.scala.html*

```
@(someModel: Model)(simpleView: Html)
<html>
    <head>
        <title>Composite View Example</title>
    </head>
    <body>
        @simpleView

        <section id="main">
         @someModel.showSomething()
        </section>
    </body>
</html>
```

32

FRONT CONTROLLER PATTERN

▶ *"Consolidates all request handling by channeling requests through a single handler object"* [Fow02]
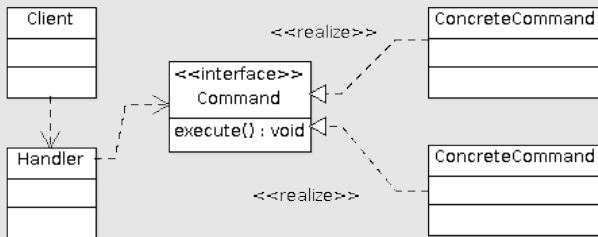
▶ Pros: Centralized control, Thread safety, Configurability.
▶ Cons: Possible performance issues, Maintenance costs.

FRONT CONTROLLER IN PLAY!

▸ The router (handler) selects a controller (command) and a
  particular action (execute) depeding on the HTTP request.
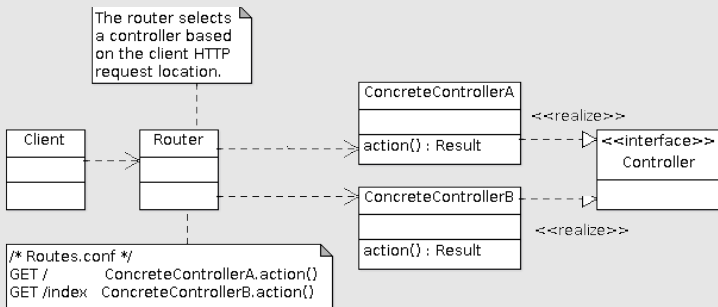
▸ Routes.conf file determines the location-action relationship.
▸ Actions return a result that holds the HTTP Response.

34

CONCLUSIONS

OUTLINE

CONCLUSIONS

- a

REFERENCES

📄 Martin Fowler, *Patterns of enterprise application architecture*, Addison-Wesley Professional, 2002.

📄 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.