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**.

PLAY! FRAMEWORK

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 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
```

PATTERNS IN PLAY!

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;

3  @Entity
   public class User extends Model {
5    @Id
     public String name;
7    @Required
     public String pass;
9
     public User (String name, String pass) {
11       this.name  = name;
         this.pass  = pass;
13   }

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

17   public static List<User> all() {
       return find.all();
19   }
   }
```

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>
```

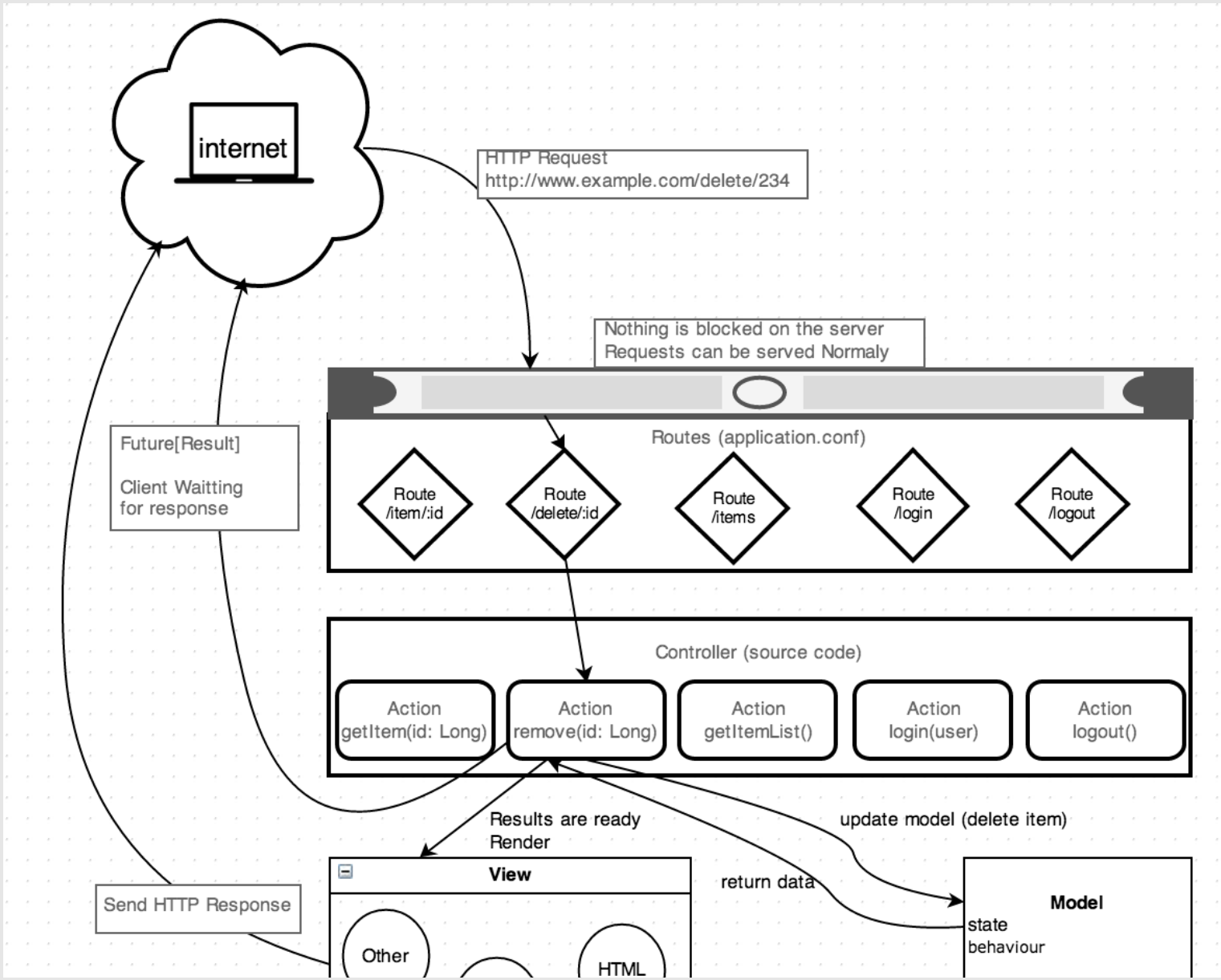
THE MVC APPLICATION MODEL: CONTROLLERS

- 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)

```
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()));
    }
}
```

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 }
```

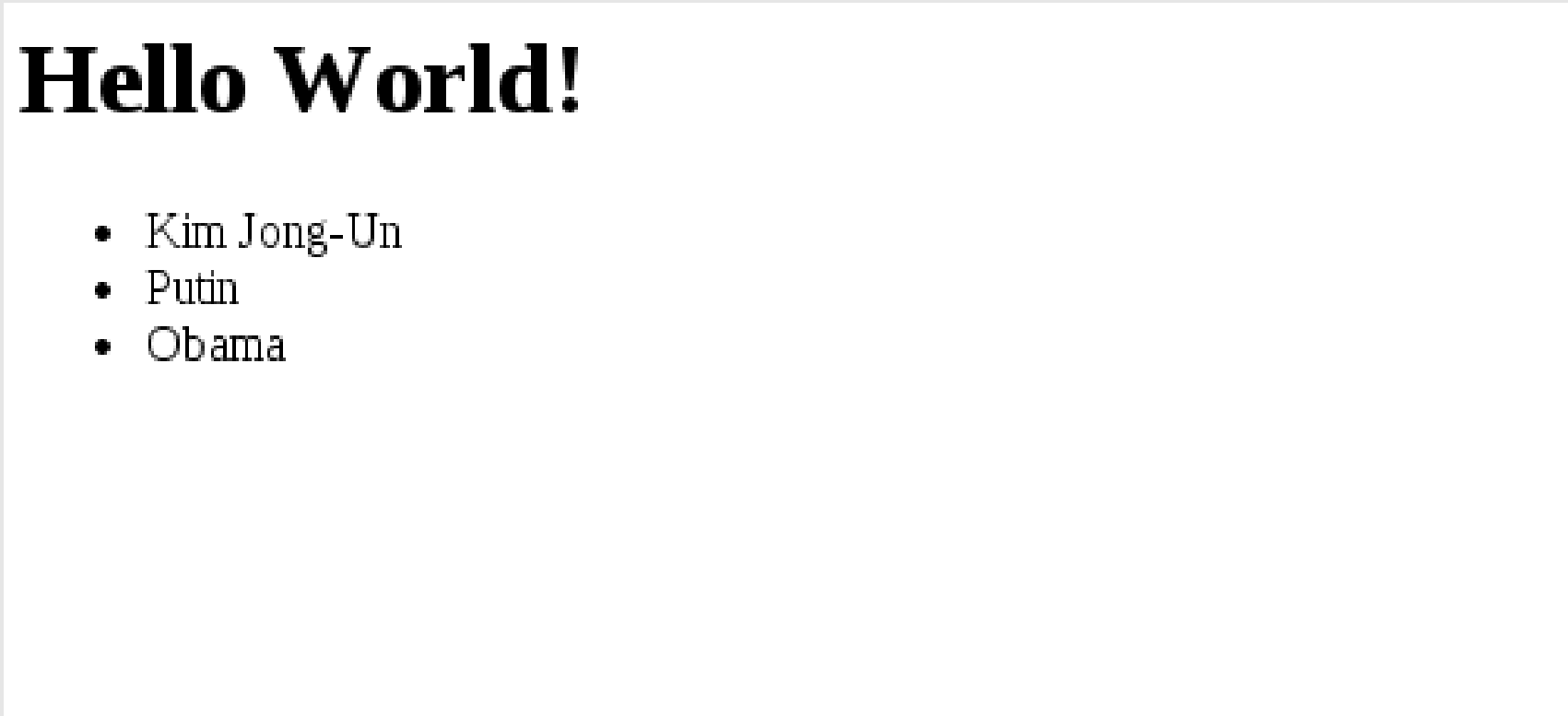
A MODEL EXAMPLE (MODELS/USER.JAVA)

```
1  package models;

3  @Entity
   public class User extends Model {
5    @Id
     public String name;
7    @Required
     public String pass;
9
     public User (String name, String pass) {
11       this.name  = name;
         this.pass  = pass;
13   }

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

17   public static List<User> all() {
       return find.all();
19   }
   }
```

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>
```

THE END RESULT



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

```
package models;

@Entity
public class Post extends Model {

  @Id
  public Long id;

  @Constraints.Required
    public String title;

    @Formats.DateTime(pattern="dd/MM/yyyy")
    public Date postedAt;

    public String content;

    @ManyToOne
    public User author;

    @OneToMany(mappedBy="post", cascade=CascadeType.ALL)
    public List<Comment> comments;
}
```

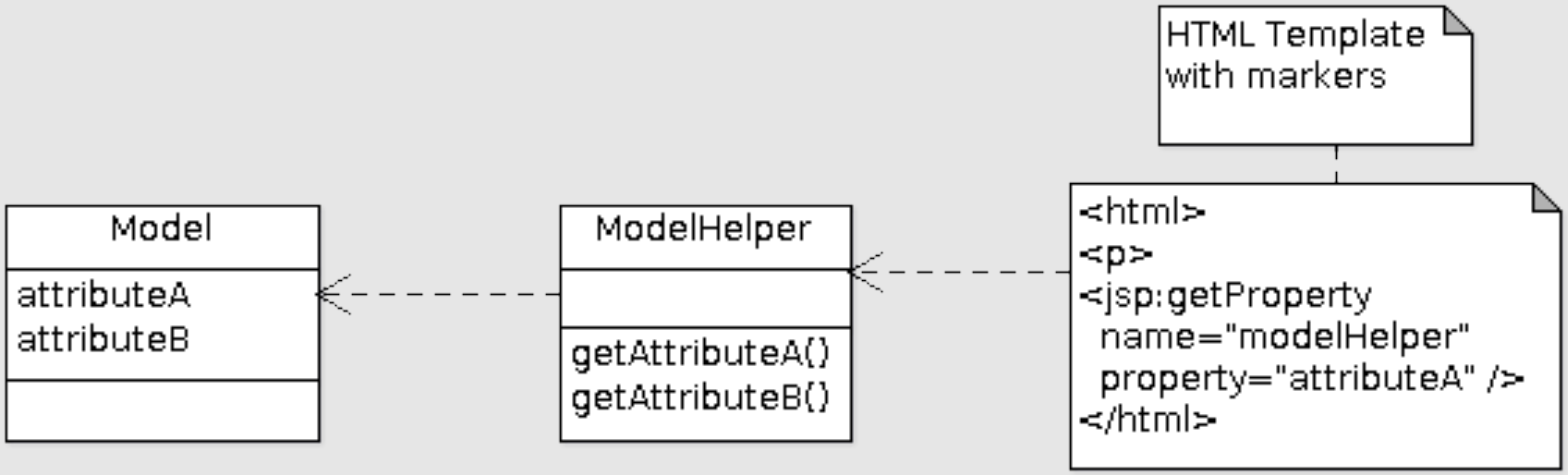
ORM: USAGE

```
User user = new User("test@test.com", "Test", "test");
user.save();

User user = User.find.where().eq("email", "test@test.com").
    findUnique();

User.find.ref("test@test.com").delete();
```

ORM Hate by Martin Fowler [Fow12]

TEMPLATE VIEW

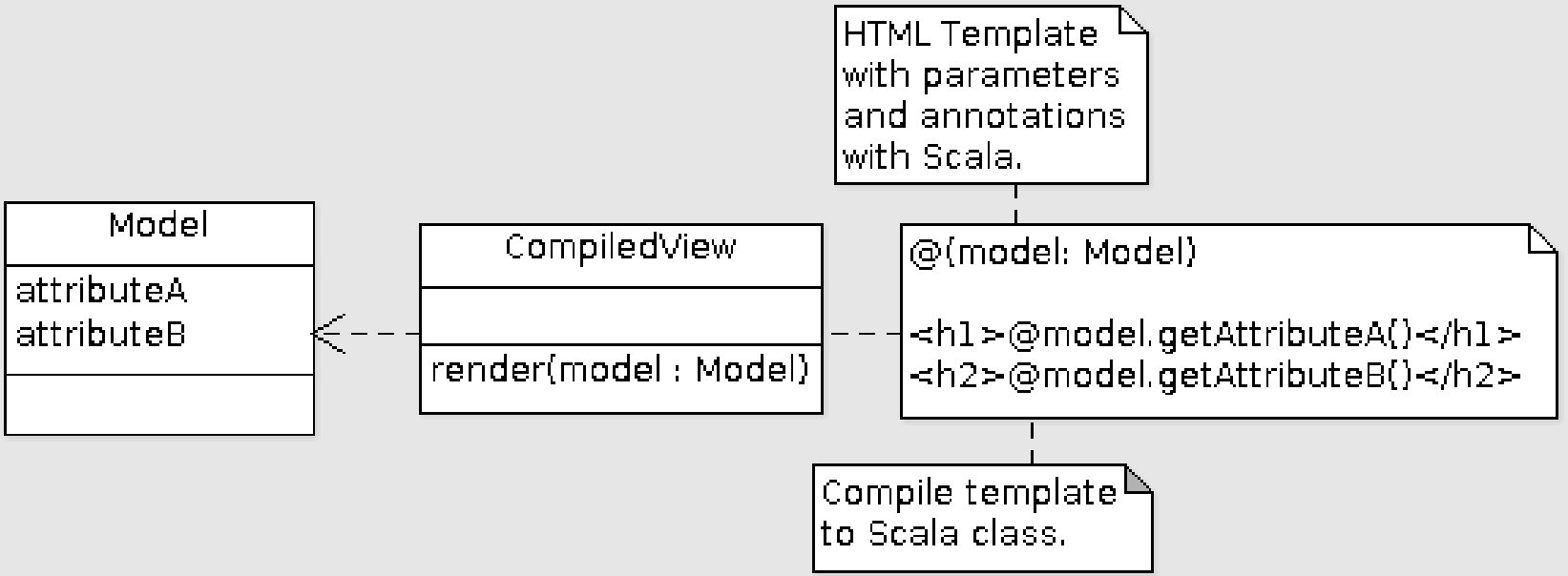
- *”Renders information into HTML by embedding markers in an HTML page”[Fow02]*



- 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.*



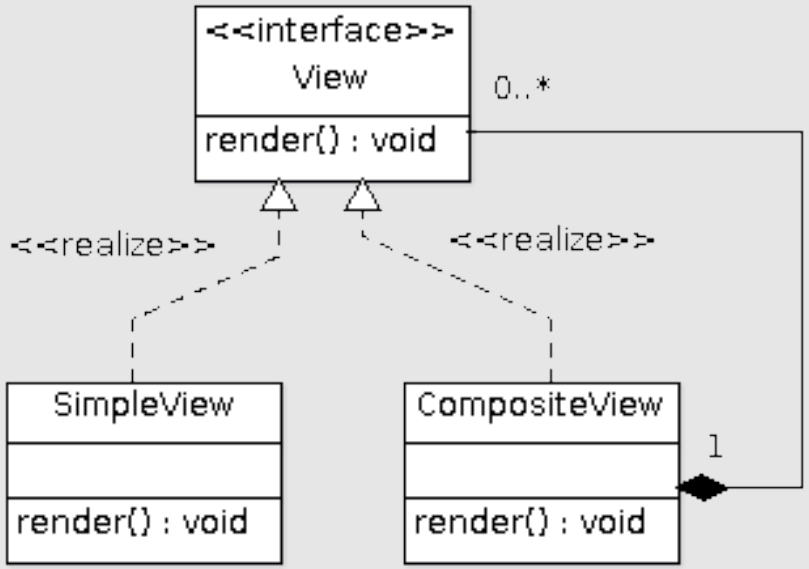
- The controller calls the render method of the view.
- The view communicates with the model (parameter).

TEMPLATE VIEW EXAMPLE

```
@(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>
```

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>
}
```

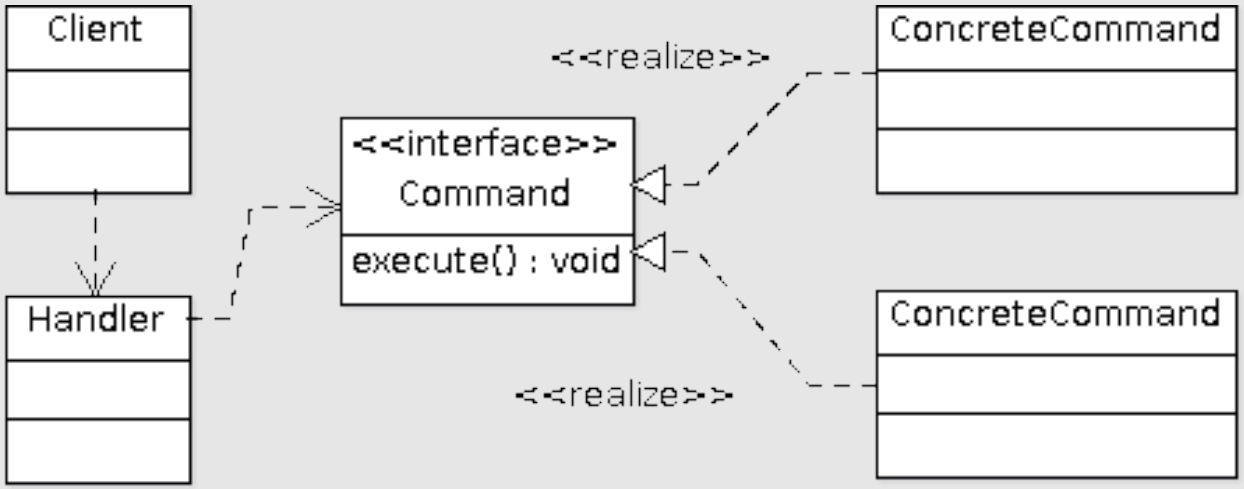
COMPOSITE VIEW

- *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>
```

FRONT CONTROLLER PATTERN

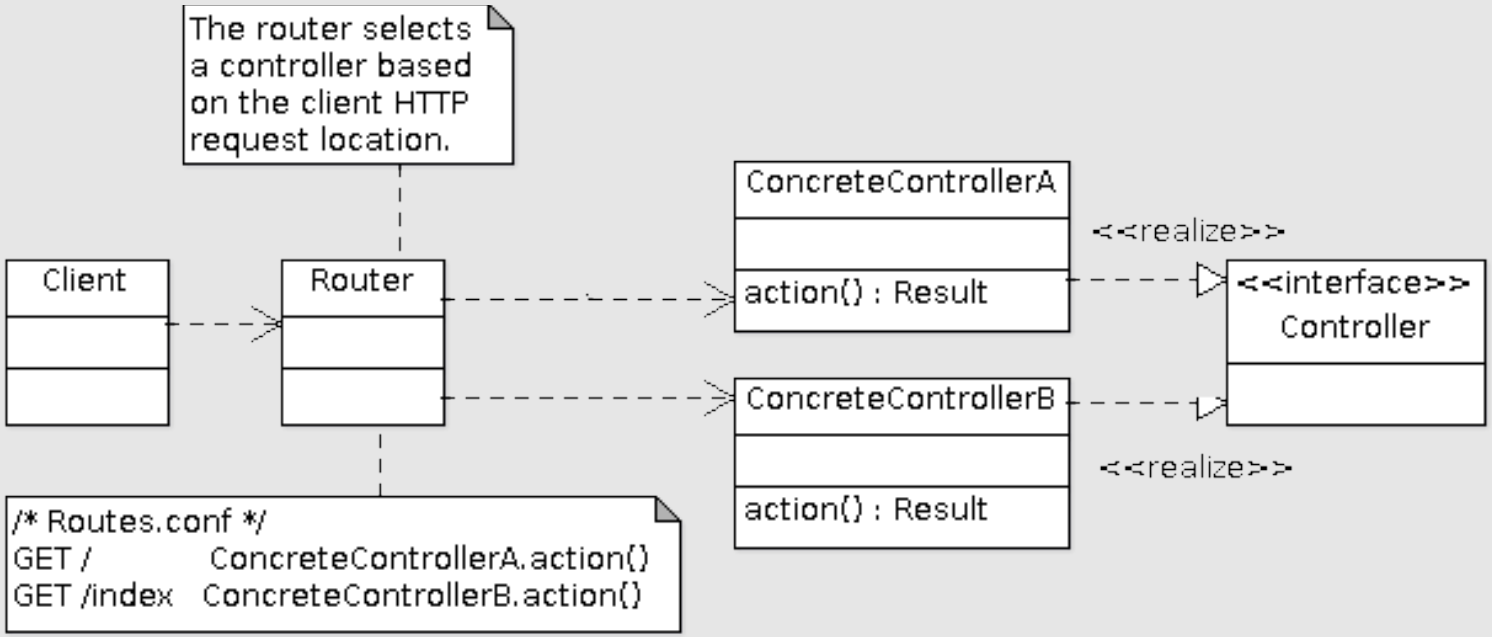
- *”Consolidates all request handling by channeling requests through a single handler object” [Fow02]*



- 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.

THE ROUTER

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

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

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

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

CONCLUSIONS

Outline

Conclusions

- 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.