Presentation patterns for web applications with Play! Framework

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

May 11, 2014

Table of Contents

INTRODUCTION

OUTLINE

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

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

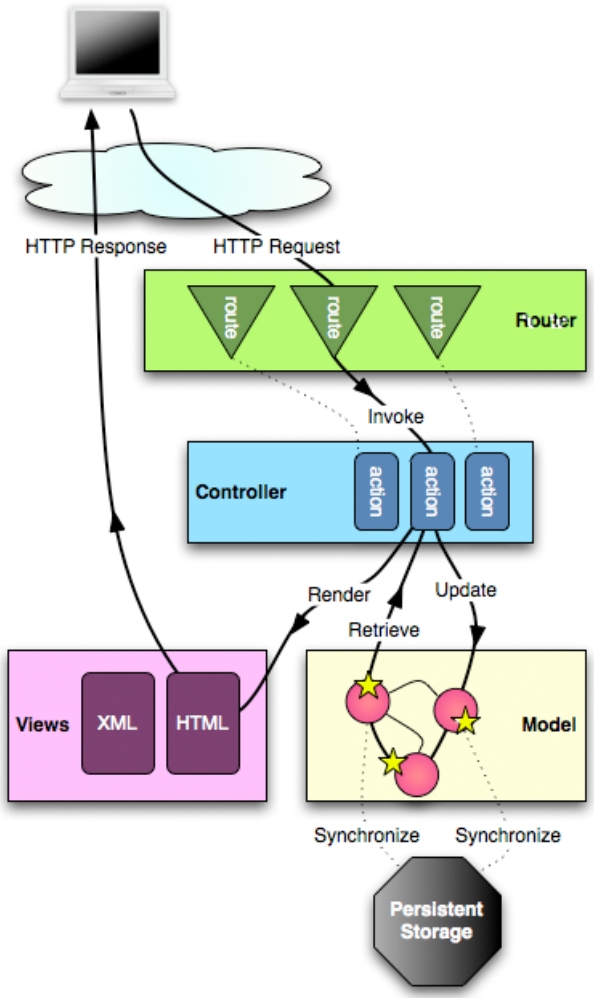
Patterns in Play!

- **Model-View-Controller**.
- Model.
  - Object-Relational Mapping.
- Controller.
  - Front Controller.
- View.
  - Template View.
  - Composite View.

The MVC application model

- Models in app/models
  - Java/Scala classes.
  - Data + Operations, mainly object-oriented.
  - Business logic and storage.
- Views in app/views
  - HTML/XML/JSON/Scala templates.
  - Directives as placeholders for data.
  - Render models to user interfaces.
- Controllers in app/controllers
  - Java/Scala classes.
  - Methods as actions, mainly procedural.
  - Receive requests, act (update models + render views) and response.

REQUEST/RESPONSE FLOW

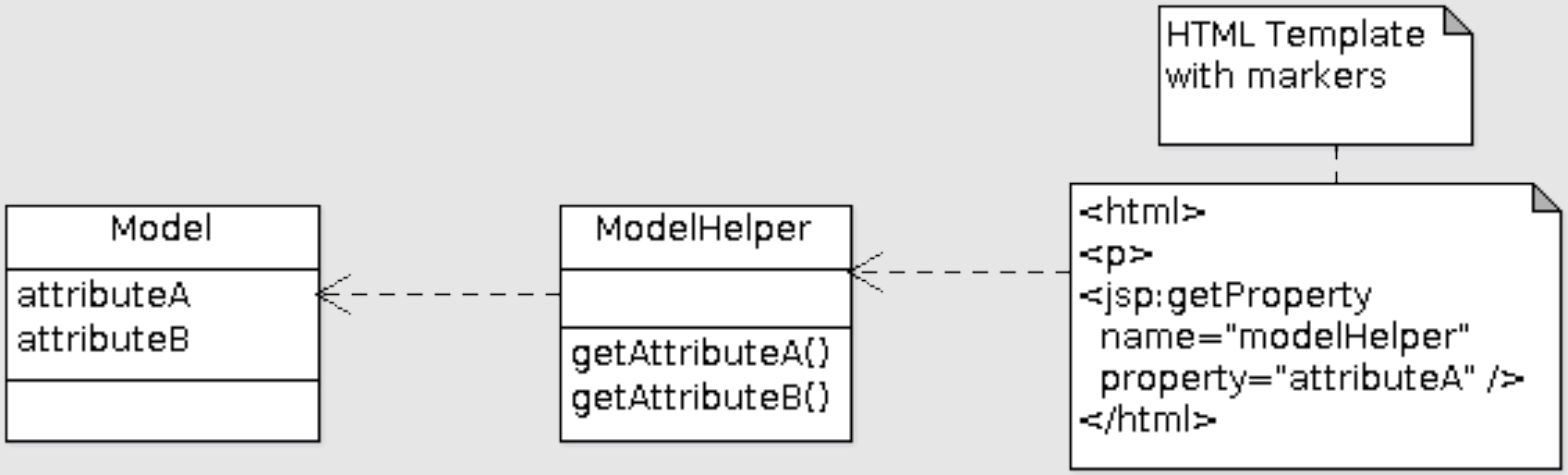


Object Relational Mapping

- a

TEMPLATE VIEW

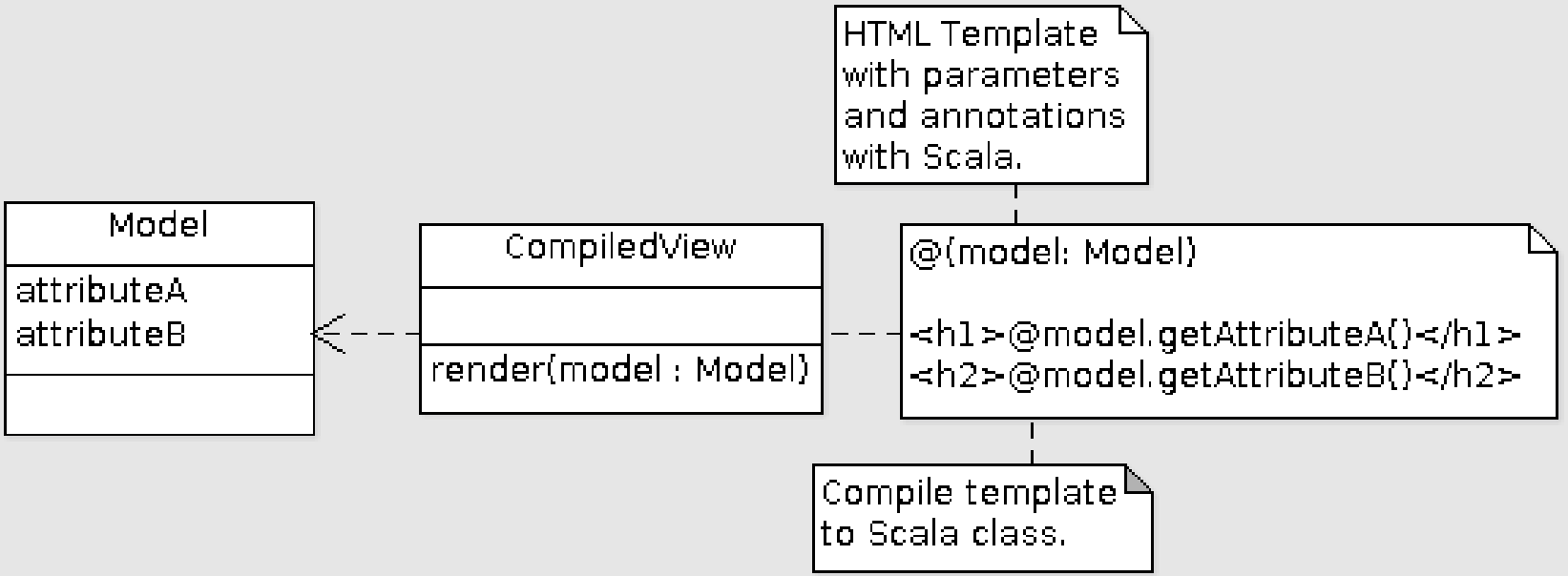
- *"Renders information into HTML by embedding markers in an HTML page"[Fow02]*



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

TEMPLATE VIEW

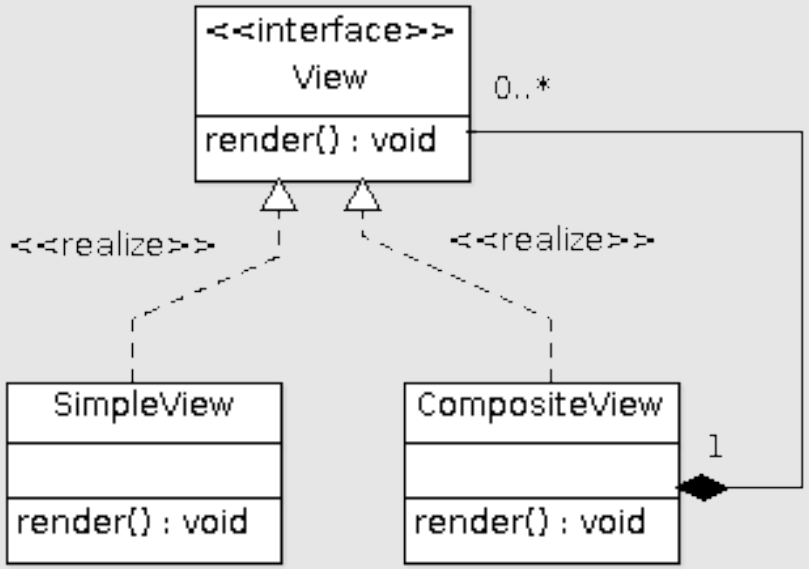
- *The template with annotations is compiled to a Scala.class with a render() method with the template parameters.*



- 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>
}
```

COMPOSITE VIEW

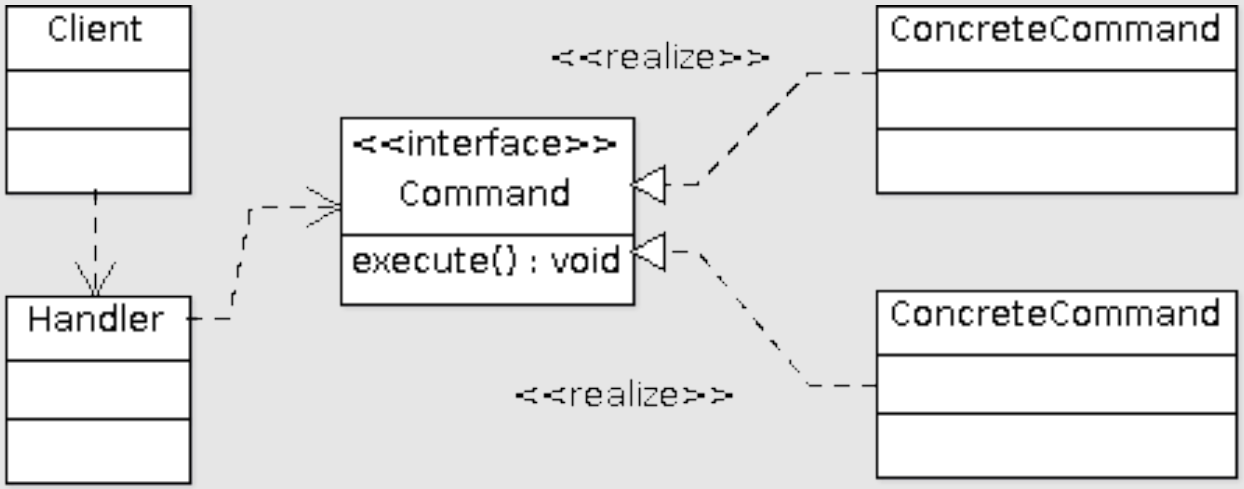
- *A composite view: compositeView.scala.html*

```
@(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

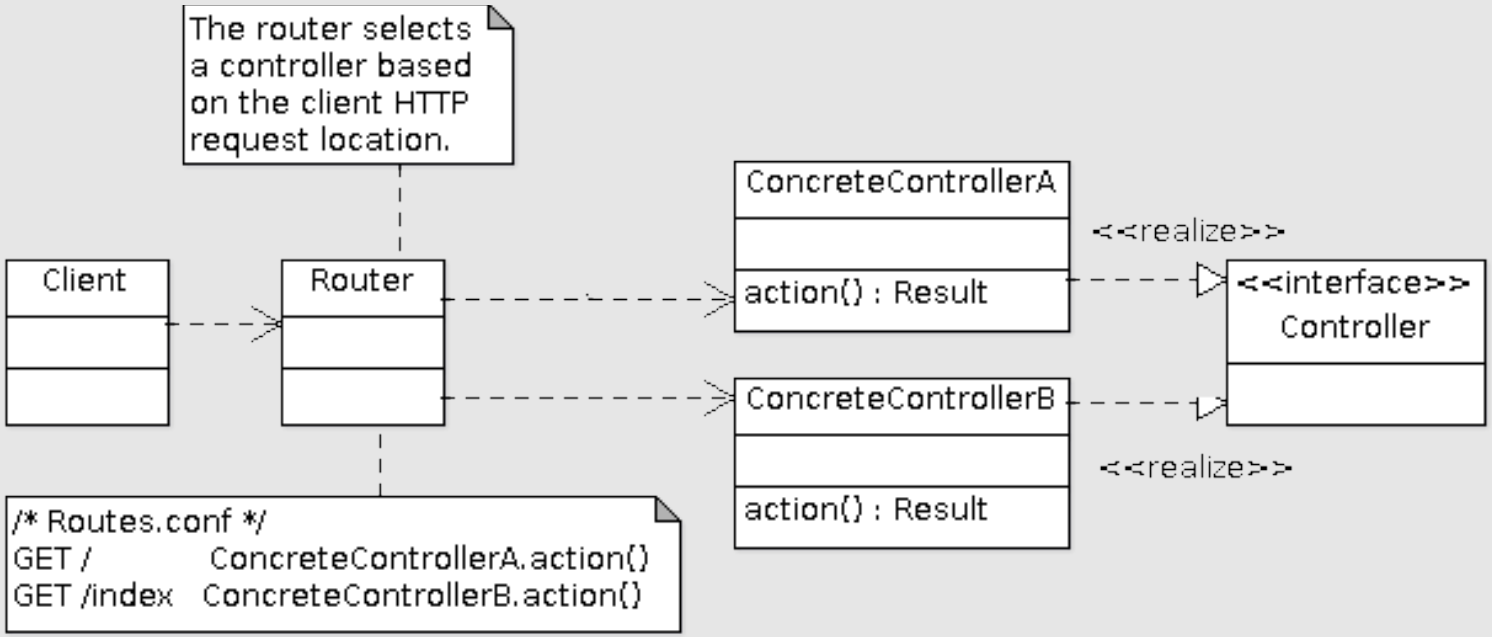
- *”Consolidates all request handling by channeling requests through a single handler object” [Fow02]*



- 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.

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.