Model Viewer
Controller

Technion – Institute of Technology
236700
What are components?

• Big applications consist of many components/modules
• A single feature usually involves several steps, that includes several components that interact with each other
• It is easier to write and maintain a program where each module has a defined set of responsibilities and the explicit "glue" between these components is minimal
• Code is modular and reusable
• Can be written by engineers of different skill sets and expertise
Test case – Web applications

• Today, applications are more often accessed through the browser than through the operating system

• This has several advantages:
  • Code is portable
    • Across operating systems and devices
  • Users don’t need to install anything
  • No need to update the software on the client side

• But some disadvantages:
  • Users expect the application to be truly responsive
    • Sometimes this means more work than supporting a single operating system: you need to support all major browsers, all handhelds, tablets, etc.
  • Usually not as performant or pretty as a native application
Web Application – Architecture

When the user points their browser to our site:

Get:
http://<server-address>/myApp/employee_details?username=don&passwd=12345

HTTP Request

Web Server
(Apache, Tomcat, etc.)

HTTP Response

myApp

Prepare Response

Get Employee data

Employee Logic

Get : employee_details
Param#1 = username, value = don
Param#2 = passwd, value = 12345

Client

<html>
<body>
<h1>Login page</h1>
...
</body>
</html>

Get :
employee_details

Databases

<html>
<body>
<h1>Welcome back Mr' Quixote</h1>
...
</body>
</html>
Model-View-Controller

- Ubiquitous *architectural* design pattern
- A logical concern separation to 3 parts:
  - Model: Data representation & manipulation
  - View: Graphical presentation / UI
  - Controller: Business Logic and choosing the view
- Goals:
  - Independent development of each concern
    - Reusability and modularity
  - Simpler, shorter code
  - Ignorance is bliss
- Most importantly, a framework methodology/structure
  - The framework will handle all connections between components
Model-View-Controller

- **Model**
  - Encapsulates application state
  - Responds to state queries
  - Exposes application functionality
  - Notifies views of changes

- **View**
  - Renders the models
  - Requests updates from models
  - Sends user gestures to controller
  - Allows controller to select view

- **Controller**
  - Defines application behavior
  - Maps user actions to model updates
  - Selects view for response
  - One for each functionality

- **State Query**
  - Change Notification

- **State Change**

- **View Selection**
  - User Gestures

- **Method Invocations**
  - Events
MVC with Play

• A framework for Java based Web development

• [https://www.playframework.com/](https://www.playframework.com/)

• We’ll only cover a fraction of this package’s features
  • You may find more videos/tutorials online
Play programming model
RESTful API

REST stands for Representational State Transfer

- Supports CRUD(L) = Create, Read, Update, Delete, List operations

- In the context of HTTP, the conventions are to use:
  - Create with POST
  - Read and List with GET
  - Update with PUT
  - Delete with DELETE
  - (You don’t have to support all of these)

- REST is about using a predefined language to access resources

- Comparing with SOAP that uses XML, REST APIs tend to be simpler, and more uniform
RESTful API (Cont.)

• Every **resource** is given a path in our application
• For example, if we have a students model
  • /students/ returns all students
  • /students/10293 returns the student with ID 10293
  • /students/10293/grades/236700/MoedA will return the students grade in Moed A in the course 236700
  • sending an HTTP POST request to /students/ will create a new student with the data in the payload

• Non-RESTful API:
  • Creating new data:
    create_student?id=10293&name=Gal
  • Get grades for course:
    grades?id=10293&grades=236700&moed=MoedA
The Router

- Defines mappings between URLs and Controllers
- Routes are defined in a `conf/routes` file
- Format is:
  
  METHOD PATH Controller.Method

  - Static routing (exact match):
    
    GET /students/ controllers.Students.list()

  - Dynamic routing (REST style)
    
    GET /students/:id controllers.Students.get(id: Long)
    
    Play will not only extract the parameters from the path, it will also convert them to the correct type!

- Non-REST API

  GET /students controllers.Students.get(id: Long)
  
  Play will parse the arguments from the HTTP parameter list, e.g., /students?id=1203
Controller – Static (old)

Two kinds of controller:

• Static:

```java
public static Result show(String page) {
    String content = Page.getContentOf(page);
    response().setContentType("text/html");
    return ok(content);
}
```

• All methods have to be static

• Meaning controllers are **singletons**
  • And we already know why that's bad...
Controller – Dependency Injection (new in 2.5)

Using dependency injection:

```java
public class MyComponent {
    private final WSClient ws;

    @Inject
    public MyComponent(WSClient ws) {
        this.ws = ws;
    }

    public Result show(String page) {
        String content = Page.getContentOf(page);
        response().setContentType("text/html");
        return ok(content);
    }
}
```

- A new instance will be created for every request
- Can use @Singleton on a class to ensure as single instance
- Bindings are handled using Modules, as usual
  - Module are configured in an application.conf file
Controller – CompletionStage

• Play is built from the ground up with asynchronisty in mind
• This means that our controllers should not block...
• By default, controllers wrap our code in futures
• If we want to explicitly return a CompletableFuture (why?), we can still do so

```java
public CompletionStage<Result> index() {
    return methodThatReturnsAFuture()
        .thenApply(this::toJson)
        .thenApply(this::ok);
}
```
Controllers – KISS

• Controllers are one of the hardest parts to test in the application
  • This is this because their input is complex (HTTP) and their output is even more so (HTML/XML wrapped in HTTP)
• Adhere to the **KISS** principle: **Keep It Short and Simple**
• Controllers should parse the input, and return a response/view
• If you need logic that is more complicated than that, it should be placed elsewhere in the code
  • If it’s front-end logic, it should be in the View (see next)
  • If it’s database or model logic, it should sit in the model
  • If it’s some other kind of backend logic, it should be in its own class and package and injected to the controller
Views – HTML in java woes

What if we want to return a **dynamic** html page (the dynamic parts are **bolded**)?

```html
<h1> Welcome John! </h1>
<ul>
  <li> Order 1: ... </li>
  <li> Order 2: ... </li>
  ...
</ul>
```

- Dynamically constructing the string:

```java
public Result rawHtml() {
    StringBuilder sb = new StringBuilder();
    sb.append("<h1> Welcome " + getCustomerName() + "! </h1>"),
    sb.append("<ul>");
    for (Order order : getOrders())
        sb.append("<li> " + order.getTitle() + " </li>"),
    sb.append("</ul>");
    return ok(sb.toString()).as("text/html");
}
```

- Result HTML structure is undecipherable, and HTML is **all about structure**
  - We're writing very imperative code: we want to write an HTML with dynamic contents, not concatenate strings

- Writing HTML in Java is hard
  - So let's write Java (well, Scala) in HTML!
Views – The Template Engine

Example page:

```
@(customer: Customer, orders: List[Order])
<h1>Welcome @customer.name!</h1>
<ul>
@for(order <- orders) {
  <li>@order.getTitle()</li>
}
</ul>
```

- External DSL code is based on Scala
- Returning a View from the Controller:

```
public Result show(String page) {
  return ok(views.customers.render(getCustomer(), getOrders()));
}
```

- Play compiles the HTML, so it will fail compilation if we have errors in our templates
Domain Specific Languages (DSL)

- The template engine is an example for **External** DSL
  - ‘External’ means it is not written as part of the hosted language
  - In our case, Scala is written inside HTML
  - Last week, we wrote SQL and HQL inside Java
- DSL’s **purpose** is making our lives **easier** by:
  - Allowing us to write less code
  - Code is simpler and easier to read
  - Force us into good coding practices
Model – Defining a model

- The `find` field is an API entry point to the ORM
- For example: `Task.find.byId(34L);`
Model – transactional controllers

Ensures each invocation is its own transaction:

```java
@.Transactional
public Result done(long id) {
    Task task = Task.find.byId(34L);
    task.done = true;
    task.save();
    return ok();
}
```

- Transactions succeed or fail together
- Database is always in a valid and legal state
Models – other frameworks

Some frameworks, e.g., **Rails** (Ruby), **Django** (Python), automatically create a database schema, CRUDL REST API, and even GUI editing tools just from the model definition.

![Django administration](image-url)
Summary

Pros:
• The frameworks handles all the **communications** between the different components
  • As per usual when using frameworks, we only have to write the code that is **unique to our application**
  • We focus on our **business rules** rather than pesky infrastructure work
• MVC frameworks are bigger than most of the frameworks we've seen so far
  • The framework **is** the **application**, we're only filling in the blanks (similar to JUnit, but in production)

Cons:
• All of our components are tightly coupled
  • If we decide to replace Play with another framework, we would have to rewrite all of our views, and a lot of our backend code
    • "All or Nothing" approach
  • But we **can** write plain old JavaScript if we want to
    • We don't have to use all the fancy bells and whistles if we don't want to
Other MVC Frameworks

- C#
  - Razor / ASP.NET
- Python
  - Django
- Ruby
  - Ruby on Rails
- Groovy
  - Grails
- Java
  - Spring Framework
- JavaScript/HTML
  - Angular JS