Separation of Concerns

Technion – Institute of Technology
236700
What are concerns?

• Big applications are consisted 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 that the "glue" between these components is minimal
• Code is modular and reusable
Test case – Web applications

A typical web app architecture

HTTP Request

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

HTTP Response

Web Server (Apache, Tomcat, etc.)

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

myApp

Prepare Response

Employee Logic

Get Employee data

Databases

Client

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

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

HTTP Response

Welcome back Mr’ Quixote

<html>
<body>
<h1>Welcome back Mr’ Quixote</h1>
...
</body>
</html>
Servlets

• We’ll write Java classes named **Servlets** extending the **HttpServlet class**

• Each class will be **mapped** to a URL and a method in it will be invoked when an HTTP request is made to that URL

```java
@WebServlet("/TimeServlet") // Tomcat 7.0+ only
public class TimeServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");

        response.getWriter().println(
            "<html><body>Current time " + new Date() + "</body></html>" );
    }
}
```
Servlets (cont.)

- Problem: Writing HTML in Java is hard

- Solution: Write Java in HTML

```html
<html>
  <body>
    Hello! The time is now <%= new java.util.Date() %>
  </body>
</html>
```
Servlets (cont.)

- Behind the scenes Tomcat will:
  - Process the JSP code and will create a **Servlet** Java code
  - Then compile and run it
  - You can see the generated .java servlet files in `<tomcat>/work/...` folder
  - You can learn more in the course: **236369 - Managing Data on the World-Wide Web**
Domain Specific Languages (DSL)

- JSP is an example for **External** DSL
  - ‘*External*’ means it is not written as part of the hosted language
  - In our case, Java is written 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

- We can also write our own **Internal** DSL
  - For most generic purposes, someone already wrote something better...
Typical Servlet Code

• What’s the problems with the web development approaches we’ve seen so far?

@WebServlet("/login")
public class HelloServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                        HttpServletResponse response) throws IOException {

    if (null == request.getParameter("name") ||
        null == request.getParameter("passwd")) {
      response.sendError(400, "Missing arguments");
      return;
    }

    String name = request.getParameter("name");
    String passwd = request.getParameter("passwd");

    if (valid(name, passwd)) {
      response.getWriter().println(
          "<html><body><h1>Hello " + name + "</h1></body></html>");
    } else {…}
  }
}
Client-Server coupling

- Servlet need to know the format of the answer (HTML/JSON/XML)
- Parameters are resolved within the Servlet
  - Type resolution: strings, integers, classes
  - Value: age, name, id
- As a result most of the Servlet code is handling "bureaucracy" which opens the door for bugs and consume a lot of development time
- Hard to decouple the UI (HTML/JSP) code from the data handling and logic.

In short, servlets do Everything
Model-View-Controller

- Ubiquitous *architectural* design pattern...
- A logical concerns separation to 3 parts:
  - Model: Data representation & manipulation
  - View: Graphical presentation / UI
  - Controller: Business Logic
- Goals:
  - Independent development of each concern
    - Sometimes by completely different professions
    - Reusability and modularity
  - Simpler, **shorter** code
  - Ignorance is bliss
- Most importantly, a **framework methodology/structure**
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**
- **State Change**
- **Change Notification**
- **View Selection**
- **User Gestures**
- **Method Invocations**
- **Events**
MVC with Spring

- A framework for Java based Web development
- [http://www.springsource.org/](http://www.springsource.org/)
- Eclipse integration
- We’ll only cover a fraction of this package’s features
  - You may find more videos/tutorials online and in the course website.
- It’s a “first-generation” MVC framework
- Next week we will see a newer framework, called Play
Spring programming model
The new Servlet

• The DispatcherServlet will redirect incoming requests to mapped @Controller methods
  • @RequestMapping annotation used to define mapping rules
  • Method parameters used to obtain request input
  • Method return values used to generate responses
• Simplest possible @Controller

```java
@Controller
public class HelloController {
    @RequestMapping("/")
    public @ResponseBody String hello() {
        return "Hello World";
    }
}
```
Mapping Requests

- By path
  - `@RequestMapping("path")`

- By HTTP method
  - `@RequestMapping(value="path", method=RequestMethod.GET)`
  - POST, PUT, DELETE, OPTIONS and TRACE are also supported

- By presence of query parameter
  - `@RequestMapping(..., params="foo")`
  - Negation also supported: `params={"foo", "!bar"}`

- By presence of request header
  - `@RequestMapping(..., header="content-type=text/*")`
  - Negation also supported
Request mapping at class level

- @RequestMapping can be used at the class level
- Concise way to map all requests within a path to a @Controller

```java
@Controller
@RequestMapping("/accounts/*")
public class AccountsController {

  @RequestMapping("active")
  public @ResponseBody List<Account> active() {...}

  @RequestMapping("inactive")
  public @ResponseBody List<Account> inactive() {...}

}
```
Obtaining request data

- Previously we needed to investigate the HttpServletRequest object for parameters
- Say we want to add a new Employee to our online DB

```java
@override
protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
    String name = req.getParameter("name");
    if (name == null) {
        //handle missing param
    }
    String ageStr = req.getParameter("age");
    if (ageStr == null) {
        //...
    }
    int age;
    try {
        age = Integer.parseInt(ageStr);
    } catch (NumberFormatException e) {
        //handle invalid numeric value
    }
    Employee employee = new Employee(name, age);
    //Add the employee to the DB
}
```
Obtaining request data, take II

• Using @RequestParam

```java
@Controller
@RequestMapping("/employee/*")
public class EmployeeController {

  @RequestMapping("add")
  public String add(@RequestParam String name, @RequestParam int age) {
    Employee employee = new Employee(name, age);
    //Add employee
    ...
  }
}
```
Obtaining request data, take III

- But why not just use **Employee** as the method parameter?

```java
@Controller
@RequestMapping("/employee/*")
public class EmployeeController {

  @RequestMapping("add")
  public String add(Employee employee) {
    //Add employee
    ...  
  }
}
```

- Spring can convert request parameters to all of Java’s simple types, as well as a few basic types (String, Arrays, Date, etc.) and to **Java beans**

- You can get request data from URL/headers/cookies/request body
Java Beans

- A Java bean is a Java class with getters & setters for some of its private members

```java
@RooJavaBean
public class Employee {
    private String name;
    private int age;
    public String getName() { return name; }
    public void setName(String _name) { this.name = _name; }
    public int getAge() { return age; }
    public void setAge(int _age) { this.age = _age; }
}
```

- Spring will find *name* and *age* in the request parameters and will match them to the Employee bean by their name.

- More complex data conversions are possible via custom *Resolvers*. These will be *injected* when needed.

- Note that beans are *mutable* by definition
Generating responses

- Can return a POJO annotated with `@ResponseBody`

```java
@Controller
@RequestMapping("/accounts/*")
public class AccountsController {

    @RequestMapping("active")
    public @ResponseBody List<Account> active() {...}

    @RequestMapping("inactive")
    public @ResponseBody List<Account> inactive() {...}
}
```

- Will be returned as the body of the response
- Automatic conversions to client’s expected format (JSON, XML, ...)
- Return a new `ResponseEntity<T> object`
  - More powerful; allows setting custom headers and status code
Returning a View

- In order for the controller to avoid messing with the view code, it is custom to simply pass on control to a designated view.
- The view can be any static or dynamic created content.

```java
@Controller
@RequestMapping("/employee/*")
public class EmployeeController {

  @RequestMapping("add")
  public String add(Employee employee) {
    //Add employee
    ...
    if (success) return "successAdd";
    return "failedAdd";
  }
}
```

Will redirect to successAdd.jsp
Model processing

- The *Model* is a simple data container.
- The @Controller should prepare the *Model* for the specific action and pass it on to the View to be rendered accordingly.

```java
@Controller
@RequestMapping("/employee/*")
public class EmployeeController {

  @RequestMapping("add")
  public String add(Model model, Employee employee) {
    //Add employee to DB
    ...
    model.addAttribute(employee);
    if (success) return "successAdd";
    return "failedAdd";
  }
}
```
Model processing

• SuccessAdd.jsp code:

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Employee added</title>
  </head>
  <body>
    <h3>Employee ${employee.getName()} successfully added</h3>
  </body>
</html>
```

• Try doing this using standard methods and you’ll need: cookies/sessions/static variables...
Views vs. @ResponseBody

• Which one do I use?
• Use views to generate documents for display in a web browser: HTML, PDF, etc.
• Use @ResponseBody to exchange data with web service clients (e.g. JS AJAX): JSON, XML, etc.
Not exclusive to Java

- C#
  - Razor / ASP.NET
- Python
  - Django
- Ruby
  - Ruby on Rails
- Scala / Java
  - Play Framework (we will see next week)
Only scratched the surface...

- Haven’t covered today:
  - Type conversion
  - Validation
  - FTP services
  - Exception handling
  - Load balancing
- Reference Manual
  - http://www.springsource.org/documentation
- Show case video
  - http://www.infoq.com/presentations/Mastering-Spring-MVC-3