USING JSP AND SERVLETS TO DEVELOP WEB APPLICATIONS

INTERACTIVE WEB APPLICATIONS AND SERVICES WITH JAVA AND XML

Lecture 4

Troels C. Damgaard

Department of Theoretical Computer Science (ITU)

20th of September / IWSJ Fall 2005
OUTLINE

1. COMBINING JSP AND SERVLETS
2. DO’S AND DONT’S
   - Do’s and Don’t’s: Servlets
   - Do’s and Don’t’s: JSP
3. MODEL VIEW CONTROLLER - AN ARCHITECTURAL PATTERN
4. MVC AND SERVLET/JSP APPLICATIONS
   - “Model 2”/MVC Web Application Design
   - Controller
   - View
   - Model
5. MVC APPLICATION FRAMEWORKS
6. MINI PROJECT 1 QUESTIONS
7. EXERCISES
8. LITERATURE / LINKS
COMBINING JSP AND SERVLETS

2 Do’s and dongt’s
  - Do’s and Dont’s : Servlets
  - Do’s and Dont’s : JSP

3 Model View Controller - an Architectural Pattern

4 MVC and Servlet/JSP Applications
  - “Model 2”/MVC Web Application Design
  - Controller
  - View
  - Model

5 MVC Application Frameworks

6 Mini Project 1 Questions

7 Exercises

8 Literature / links
Common approach:

- **Servlets** handle the **contents** (using lots of Java code)
- **JSP pages** handle the **presentation** (using lots of HTML)

– communicate using HttpSession attributes, forward requests using RequestDispatcher.
**Example Servlet receiving the original request**

```java
public class Register extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, ServletException {
        String email = req.getParameter("email");
        HttpSession session = req.getSession(true);
        session.setAttribute("email", email);
        RequestDispatcher dispatcher =
            getServletContext().getRequestDispatcher("/present.jsp");
        dispatcher.forward(req, resp);
    }
}
```
- this can quickly becomes a mess...

This lecture is intended to give ideas for

- how to structure applications larger than our Helloworld examples,
- and, more generally, what is good and what is bad practice when using JSP and servlets.
Outline

1. Combining JSP and Servlets
2. **Do’s and Don’t’s**
   - Do’s and Don’t’s: Servlets
   - Do’s and Don’t’s: JSP
3. Model View Controller - an Architectural Pattern
4. MVC and Servlet/JSP Applications
   - "Model 2"/MVC Web Application Design
   - Controller
   - View
   - Model
5. MVC Application Frameworks
6. Mini Project 1 Questions
7. Exercises
8. Literature / links

We’ll review selected parts on the following slides.
Do’s and Don’t’s: Servlets

Use servlets ...

- for implementing the logic behind a service (using JSP for presentation)
  Examples:
  - security, personalization, search functionality, application control, ...
- as controllers – as we shall see in the MVC (or “Model 2”) web application approach
- to generate binary content (JSP pages can’t), e.g:

Example of Servlet Producing Binary Content

```java
public class JpgWriterServlet extends HttpServlet {
    public void service(HttpServletRequest req, HttpServletResponse rsp) throws ... {
        rsp.setHeader("Content-type", "image/jpg");
        OutputStream os = rsp.getOutputStream();
        ...
Do not use servlets for generating primarily static content

- potential for many errors in producing content (proper escaping of special chars is a pain – e.g., like
  
  ```java
  output.println("<a href="\"mynameis?name=doc\">Doc Halligan</a>"
  )
  ```

- updating requires recompilation

Further, remember to respect the intention of `forward` and `include` –

- when `forward`’ing - response body must be empty!
Do’s and Don’t’s: JSP

Use JSP . . .

- for creating pages for presenting data – i.e. data changes, but not (overall) structure
- for generating XML; use JSP to
  - generate XML in std. formats (e.g. only attribute values change)
  - generating an XML page from several subdocuments – using JSP include and perhaps some processing for tying together
- (more generally) for constructing a page from different parts or templates
- but also; for generating plain (ascii) text or even structured plain text like emails.
Do not use JSP . . .

- for creating pages with varying structure
- for implementing business logic – beyond what is accessible (and implementable) via tags.
- for controlling request routines (i.e. do not use RequestDispatcher.forward)
- with extensive use of logic tags (looping, expression evaluation, variable setting, etc.).

Though available in the JSTL even light use violates presentation and logic separation. Always prefer implementing logic entirely in tags, servlets or helper classes.
In general, try to **avoid using scriptlets** in JSP pages – instead use **custom tags** and **helper classes**.

Apart from mixing logic and presentation, JSP scriptlets

- use implicit objects (defined by boilerplate code)
- are **hard to test** in isolation
- make JSP pages **harder to read**
- and make JSP pages **harder to debug**, because of the translation layer (as we saw at the last lecture)
  
  ... still the case, when debugging tags – but tags allow for some level of explicit contracts

At the least, try to only use scriptlets for **display logic** – *not business* logic.
COMBINING JSP AND SERVLETS

DO’S AND DON’T’S
  • Do’s and Don’t’s: Servlets
  • Do’s and Don’t’s: JSP

MODEL VIEW CONTROLLER - AN ARCHITECTURAL PATTERN

MVC AND SERVLET/JSP APPLICATIONS
  • “Model 2”/MVC Web Application Design
  • Controller
  • View
  • Model

MVC APPLICATION FRAMEWORKS

MINI PROJECT 1 QUESTIONS

EXERCISES

LITERATURE / LINKS
What is a Design Pattern

Terminology stems from Christopher Alexander, an Austrian architect who designed an simple-to-use pattern language for designing and building.

Term adopted by the computer science community, in particular, after Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994 by Gamma et. al. (Gang of Four)

Here the definition of a Design Pattern is repeated from Christopher Alexander’s:

**Definition of “Design Pattern”**

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice.” (Christopher Alexander *et. al.*, 1977)

A collection of good (object-oriented) design paradigms, rather than concrete solutions.
Model-View-Controller is generally not thought of as a design pattern, since it encapsulates 'more' than a typical design pattern.

Instead, sometimes thought of as a so-called architectural pattern.

Basically,

- Model: **Domain-specific representation** of the information on which the application operates.
- View: **Presentation layer** for model into a form suitable for interaction (typically a UI element).
- Controller: **Event handler** for typically user actions, and invokes changes on the model or view as appropriate.
Typical layout of MVC application:
Basic gains from using MVC:

- Data-source and views can be changed independent of each other.
- Decoupling of components, e.g.
  - provide presentation for different clients without changing model
  - change the way an application responds to user events, without changing view (maybe even runtime)
- As contracts and responsibilities are clearly specified debugging is easier.
- Meta issue: Helps developers with different skills work together, using clearly defined interfaces.
1. Combining JSP and Servlets
2. Do’s and don’t’s
   - Do’s and Don’t’s: Servlets
   - Do’s and Don’t’s: JSP
3. Model View Controller - an Architectural Pattern
4. **MVC AND Servlet/JSP APPLICATIONS**
   - “Model 2”/MVC Web Application Design
   - Controller
   - View
   - Model
5. MVC Application Frameworks
6. Mini Project 1 Questions
7. Exercises
8. Literature / links
Applying MVC to web applications: “Model 2”.

Basic ideas

- servlet(s) and regular Java classes handle communication and business logic
- JSP pages handle (most of) presentation
“Model 1” and “Model 2”?! 

- **Terminology** stems from original JSP specification (now outdated)
- Simply put:
  - “Model 1” : No controller servlet ⇒ decentralized
    - each JSP page or servlet processes parameters, requests, ...
    - next JSP page or servlet invoked is determined by hyperlinks or by request params
  - “Model 2” : Controller servlet ⇒ centralized
    - single point of access control, logging, filtering, ...
    - view selection invoked through controller (decouples components)
    - helps enforce data encapsulation of (data) model
Use “Model 1” for smaller, mostly static, applications with

- simple page and session flow,
- primarily fixed page navigation, and
- minor security constraints and logging requirements;

and which is expected to need only minor updates and changes

(Famous last words: “This application is future secure (DK: fremtidssikret)’

”)
Switch to “Model 2” when one or more of the above statements for your application are false.

- In particular, when you find yourself using scriptlets, custom tags, or JavaScript to forward requests
  (⇒ logic is embedded in page which is making decisions based on parameters or data)
In one slide:

- **Model**: Implemented by Java helper classes (perhaps Beans)
- **Controller**: (Single) servlet
- **View**: Collection of JSP pages
Recomendation is to use a single servlet as controller (in accordance with the Front Controller design pattern)

This servlet will

- handle all request processing, mapping them to operations on the application model
- handle all view selection, based on model and session state
- be the point of security, logging, . . .

Alternative strategy:
Filter Control Strategy: Use a filter to intercept and function as a controller (might use Filter Chaining to decompose functionality).
HOW TO: MAP REQUEST TO OPERATIONS

A client can

- indicate intended operation in a hidden form field using POST (invisible to client)
- or indicate the operation in a HTTP GET parameter; e.g.:
  http://myHost/myApp/servlets/myServlet?op=createUser
- We can also map from URLs with some suffix (e.g., *.do) to controller servlet, ⇒ controller can extract operation from request URL

Of these three, the last is the most versatile.
A controller can (as the example in the compendium) contain a large if-then-else statement:

**BAD CONTROLLER**

```java
...  
if (command.equals("/search.do")) {
    model = new Model();
    ...
}
else if (command.equals("/select.do")) {
    model = (Model) session.getAttribute("model");
    ...
}
else if (command.equals("/display.do")) {
    model = (Model) session.getAttribute("model");
    ...
}
...  
```

This is officially considered bad practice™.

Results in hard-to-read and hard-to-maintain code.
Instead, use **Command** design pattern:

- design abstract class `Command` — declares an interface for executing operations
- each concrete `Command` subclass encapsulates the necessary code for a single operation on the model
- controller invokes by operations by delegating to appropriate commands
- mapping requests (e.g. URLs) to command can be externalized entirely from the controller (e.g. in an XML-file)
How to: Execute Operations on Model (cont.)

Example Command’s

```java
public abstract class Command {
    protected Model model;
    public Command(Model model) { this.model = model; }
    public abstract String getName();
    public abstract Object perform(HttpServletRequest req);
}

public class CreateUserCommand extends Command {
    public CreateUserCommand(Model model) {
        super(model);
    }
    public String getName() { return "createUser"; }
    public Object perform(HttpServletRequest req) {
        return model.createUser(req.getAttribute("userid"),
                                 req.getAttribute("pw"));
    }
}
```
**Nice Controller**

```java
public class ControllerServlet extends HttpServlet {
    private HashMap actions;
    public void init() throws ServletException {
        // use actions to map command names to concrete Command’s
    }
    public void doPost(HttpServletRequest req,
                        HttpServletResponse resp)
                    throws IOException, ServletException {
        String op = getOperation(req.getRequestURL());
        Command action = (Command)actions.get(op);
        Object result = null;
        if (action!=null) {
            result = action.perform(req);
        } else{
            // signal error - no such command
        } // use result to determine next view
    }
}
```

This controller is essentially independent of the concrete web application – since it is entirely parameterized by the commands in `actions`, and the `getOperation` method.
Controller is also responsible for choosing how the clients view is updated. Decision is based on

- current view
- result of command issued upon model by client (e.g., success or failure)
- possibly other server context state (page, request, session, or application) (e.g., time-of-day)

Basic solution: Forward request directly to JSP-page, servlet, ...:

```java
getServletContext().getRequestDispatcher("/nextview.jsp")
    .forward(request, response)
```
Advanced solution: Abstract screen flow selection into another generic component: A screen flow manager.
The **screen flow manager** - same idea as for generic controller - configure by external file (XML)

Might consider collecting configuration of **controller** and **screen flow manager** into a single XML file:

**AN XML FILE FOR CONFIGURING GENERIC CONTROLLER AND SCREEN FLOW MANAGER**

```xml
<url-mapping url="signoff.do" screen="signoff.screen">
    <command-class>
        myAppPackage.controller.actions.SignOffHTMLAction
    </command-class>
</url-mapping>
```
Screen flow might look more like this now:
View layer composed of mainly JSP pages and static HTML, but also servlets (for generating binary content), PDF documents, Flash, ... 

Idea: Use a main component – a Templating Service together with template JSP-files and a number of subviews. 
Implement Templating Service as a single generic servlet, which centralizes control of equal functionality and design in application. 

Standard solution: Use url-mapping in deployment descriptor to map all urls ....screen to Templating Service servlet.
Allows us to

- decompose and decouple subviews from main template (include by reference);
- and (again) configure a specific Templating Service by an external file (example on next slide).

This is essentially an instance of the Composite View pattern.
Example Template Service Configuration File

```xml
<screen-definitions>
  <template>/template.jsp</template>
  <screen>
    <screen-name>main</screen-name>
    <parameter key="title">Welcome to the BluePrints Petstore</parameter>
    <parameter key="banner" value="/banner.jsp"/>
    <parameter key="sidebar" value="/sidebar.jsp"/>
  </screen>
</screen-definitions>
```

Example template JSP-file on next page
TEMPLATE JSP FILE

<%@ taglib uri="/WEB-INF/tlds/taglib.tld" prefix="template" %>
<html>
<head>
    <title><template:insert parameter="title" /></title>
</head>

<body bgcolor="#FFFFFF">
<table width="100%" border="0" cellpadding="5" cellspacing="0">
    <tr>
        <td colspan="2">
            <template:insert parameter="banner" />
        </td>
    </tr>
    <tr>
        <td width="20%" valign="top">
            <template:insert parameter="sidebar" />
        </td>
        <!-- ... -->
    </tr>
</table>
</body>

<!-- ... -->
Implement a Web Application model as a Java application to encapsulate data application state.

Interface

- with **Controller** using mutators or commands (as sketched earlier)
- with **View** using accessors (perhaps using Java Bean conventions and JSP EL for convenience)

Further, present a **simple API** for the **View** and the Controller implementors to interface with; might use

- a **Facade** – a single class presenting a higher-level interface for low-level Model components
- a **Command** structure

Read more: “Design Patterns” – Gamma et. al., or
http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch5.html#1091720
Handling Multiple Client Types

Developing an application using MVC abstractions greatly eases the display and access to a resource (the model) across several differing interfaces.

For larger web applications might need to serve different client types. E.g.

- Different browsers
- Web service interfaces
- Mobile phones (WML)
- ...

Use Controller abstraction - implement a Controller (and View) for each client-type (⇒ share model).

Might collect common functionality (e.g. security) – possible using Filters.
What do we need to do to support a new client?

**Answer:** Code a new view, and implement some logic in your controller to map the signals from the new client to existing Commands.

Outline

Combining JSP and Servlets

Do's and don't's
- Do's and Don't's: Servlets
- Do's and Don't's: JSP

Model View Controller - an Architectural Pattern

MVC and Servlet/JSP Applications
- "Model 2"/MVC Web Application Design
- Controller
- View
- Model

MVC Application Frameworks

Mini Project 1 Questions

Exercises

Literature / links
Basic idea

Basic idea of MVC application frameworks

The good old:

“Now, if many of us are going to develop our applications according to a common paradigm, why not abstract common parts into a common framework. Heck, in fact, let’s make this framework highly configurable and parameterized so we can just instantiate it to any web application by configuration, reference and extension; rather than by reprogramming (read: copy-paste programming).“

– and it’s a “…BluePrints recommended best practice to […] choose an existing, proven Web-tier application framework…”.

There are, of course, all the standard benefits and problems with customizing a possibly heavy-weight framework instead of building you own custom solution…
Apache Struts is a free, open-source MVC framework developed at the Apache Software Foundation. Includes configurable

- Front Controller, action classes and mappings (Commands), utility classes for XML, Web forms with validation, . . .

– list is expanding as we speak.

Further, provides custom tag library to support all levels of the MVC application.

Primarily uses XML for configuration.
JavaServer Faces – a standard for Java Web Application frameworks.

JavaServer Faces is another project under the political development of a Java Community Process (JCP) (JSR-127). Basically, provides most of what Struts provide; but also standardizes and provides configurable

- standard GUI components – handling state, input validation, page navigation, support for internationalization, . . .
- and an event-model for server-side events (ala Swing event-model)
- Further, provides a JSP tag library for expressing a JavaServer Faces interface within a JSP page.

Basic idea: With a standardized architecture, tools will be able to provide direct support for the framework components.
More on differences between Struts and JavaServer Faces:
http://java.sun.com/j2ee/javaserverfaces/faq.html#differences

J2EE BluePrints Web Application Framework (WAF) – the framework developed for the BluePrints book and freely available from the book website:
http://java.sun.com/blueprints/guidelines/
OUTLINE

1. Combining JSP and Servlets
2. Do’s and don’t’s
   - Do’s and Don’t’s : Servlets
   - Do’s and Don’t’s : JSP
3. Model View Controller - an Architectural Pattern
4. MVC and Servlet/JSP Applications
   - “Model 2”/MVC Web Application Design
   - Controller
   - View
   - Model
5. MVC Application Frameworks
6. MINI PROJECT 1 QUESTIONS
7. Exercises
8. Literature / links
MINI PROJECT 1 QUESTIONS

First – if you haven’t checked the webpage recently, please do.

Second – if you find all this too easy, please consider the suggested extensions:

- Extend the client so that it displays a web page before the user rates it.
- Extend your client to support more advanced queries (for example, show all pages rated by john doe@itu.dk, with a rating above 3, and within the last 24 hours. You will have to implement this combining the queries described above (you cannot send other queries to the server).

And now: Any questions?
1. Combining JSP and Servlets
2. Do’s and dont’s
   - Do’s and Don’t’s : Servlets
   - Do’s and Don’t’s : JSP
3. Model View Controller - an Architectural Pattern
4. MVC and Servlet/JSP Applications
   - “Model 2”/MVC Web Application Design
   - Controller
   - View
   - Model
5. MVC Application Frameworks
6. Mini Project 1 Questions
7. EXERCISES
8. Litterature / links
Exercises: 10.1, 10.4, and (instead of 10.5 and 10.10):

- Rewrite the example `select.jsp` on page 121 of the compendium to use plain Java code instead of the JSTL.
- Write a “Model 2” version of the Quickpoll example.
1. Combining JSP and Servlets
2. Do’s and dont’s
   - Do’s and Don’t’s : Servlets
   - Do’s and Don’t’s : JSP
3. Model View Controller - an Architectural Pattern
4. MVC and Servlet/JSP Applications
   - “Model 2”/MVC Web Application Design
   - Controller
   - View
   - Model
5. MVC Application Frameworks
6. Mini Project 1 Questions
7. Exercises
8. Literature / Links
LITERATURE AND LINKS