Web Applications Engineering: Web Design Patterns and Guideline

Service Oriented Computing Group, CSE, UNSW

Week 6

References used for the Lecture:

- http://java.sun.com/blueprints/patterns/index.html
- Core J2EE patterns, Deepak Alur, John Crupi and Dan Marlks, Prentice Hall
- Patterns of Enterprise Application Architecture, Martin Fowler, Addison-Wesley

J2EE Design Patterns

What is a design pattern?

- Patterns in software were popularised by the book *Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides* (aka Gang of Four)
- They observed and recognised recurring designs in numerous projects and documented this collection
- Since then many other design patterns are published covering patterns for various domains and purposes

J2EE patterns:

- J2EE patterns represent best practice design based on collective experience of working with J2EE projects
- Patterns aim to provide and document a solution to a known, recurring problem in building J2EE applications

`http://java.sun.com/blueprints/patterns/index.html`

J2EE Design Patterns

There are over 20 patterns (and growing).

We look at a few patterns that are highly relevant to COMP9321 ...

- **Model View Controller**: MVC is the J2EE BluePrints recommended architectural design pattern for interactive applications.
- **Front Controller (Command)**: For providing a central dispatch point to handle all incoming requests.
- **Service Locator**: Typically used in business layer for locating resources (such as database connection)
- **Data Access Object**: typical pattern for data access layer (linking the data storage layer with the application)
- **Business Delegate**: in business layer

Web Application Architecture

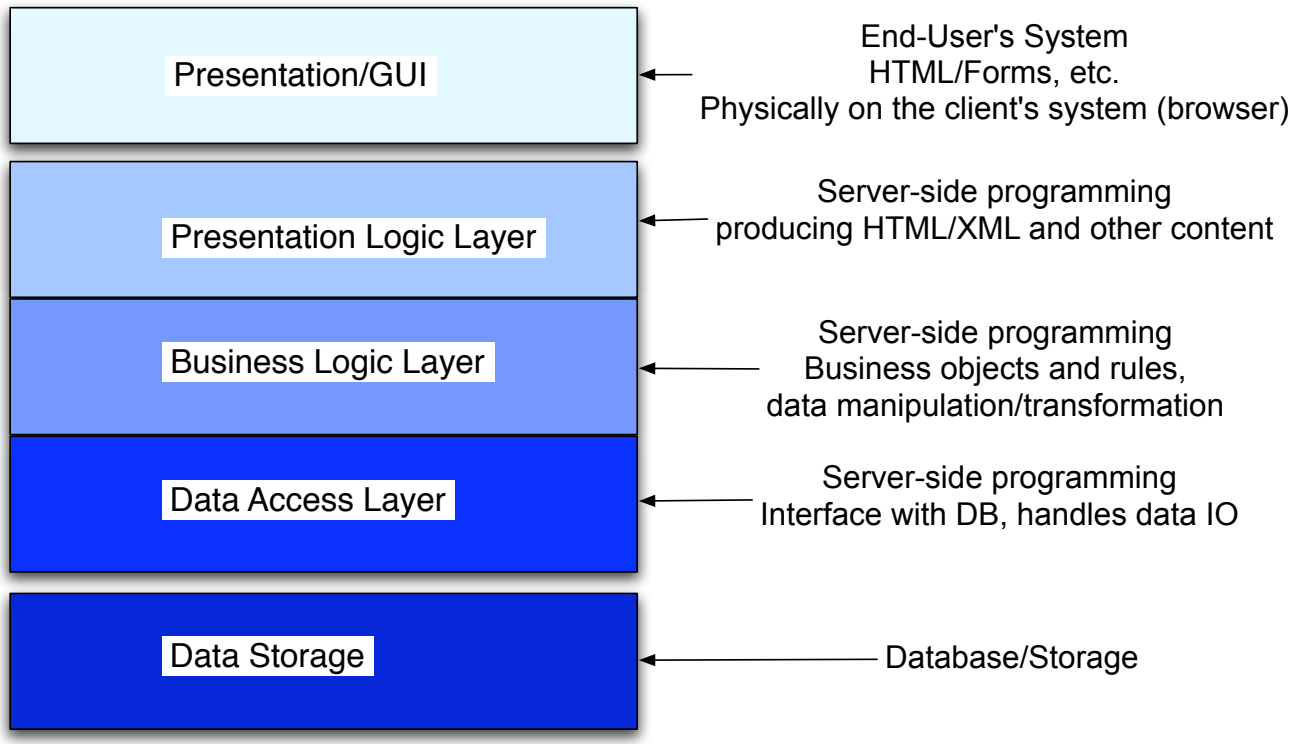
"...architecture is a subjective thing, a shared understanding of a system's design by the expert developers on a project. Commonly this shared understanding is in the form of the major components of the system and how they interact. It's also about decisions, in that it's the decisions that developers wish they could get right early on because they're perceived as hard to change".

- Martin Fowler, Patterns of Enterprise Application Architecture

Web Application Architecture

An application system consists of three "logical" layers.

- **The presentation layer**: what the user sees or interacts with. It consists of web pages, various visual objects, interactive objects, or reports. Typically, when people think of a Web application, they think of the presentation layer.
- This layer represents a small portion of the effort involved in building application systems.



Web Application Architecture

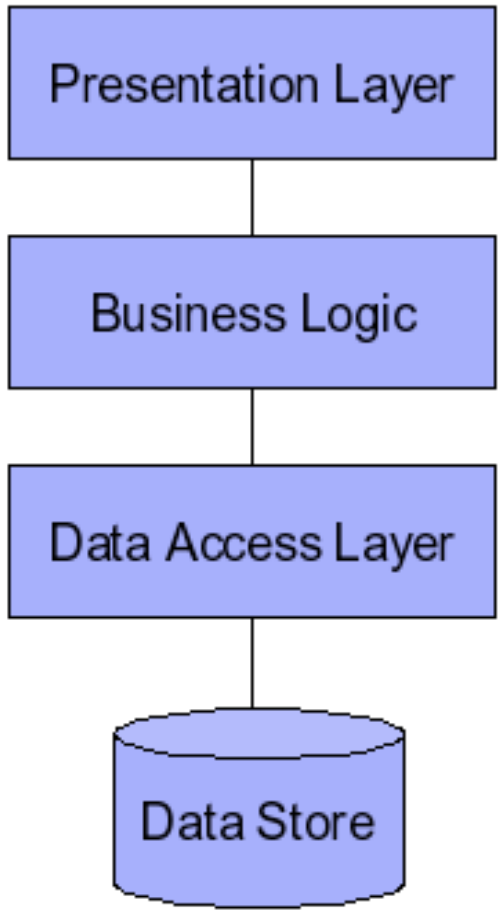
- **The business logic layer**: represents the business rules that are enforced via programming logic (eg., java classes) regarding how those rules are applied.
- **The data access layer:** consists of the definitions of database tables and columns and the computer logic that is needed to navigate the database. The data access layer enforces rules regarding the storage and access of information. For example, dates must be valid dates and numeric fields must never contain alphanumeric characters.

MVC pattern is designed to apply this logical separation of layers into system implementation.

Web Application Layers

Different solutions for each layer

- Presentation Layer
  - JSP, XSLT, CSS, HTML
- Business logic
  - Java classes
- Data Access Layer
  - Data Access Objects
- Data Store
  - RDBMS, OODBMS, XML Database



J2EE design guidelines: Web tier

J2EE Web Tier includes two technologies:

- Servlets
- JSP pages

Servlets:

- Java's solution to the old CGI technology
- They have access to the complete Java programming language
- HTML code were placed into the Servlets – difficult to maintain
- JSP is designed to solve this problem ...

eg., Mixing HTML with Java coding in Servlet

```
public class BadlyMixedServlet extends HttpServlet {
protected void processRequest(HttpServletRequest request,  ... {
   boolean databaseOpen = false;
   response.setContentType("text/html");
   java.io.PrintWriter out = response.getWriter();
   out.println out.println("<html><head>");
   out.println("<title>Servlet</title></head>");
   out.println("<body>");
   // go get a database connection ...
   if (databaseOpen()) {
      out.println("<h1>Well Done!</h1>");
   } else {
      out.println("<font color="red">Please make sure your
                                 database is running</font>");
      out.println("<br>");
   }
   out.println("</body></html>");
   out.close();
}
```

Servlet design guidelines: When to use Servlets

As a Controller

```
public class ControllerServlet extends HttpServlet {

        // dispatching the request to the handler ...
        if(VIEW_CAR_LIST_ACTION.equals(actionName))
            destinationPage = "/carList.jsp";
        else(ADD_CAR.equals(actionName))
            destinationPage = "/AddCart.jsp";
        else ... // more action checking here ...

        RequestDispatcher ds =
                    getServletContext().getRequestDispatcher(destinationPage);
        ds.forward(request, response);
}
```

Use dispatcher forward (do not commit output before forwarding!)

If your Servlet is emitting some HTML, you may want to re-think.

Servlet design guidelines: When to use Servlets

To generate binary content: e.g., downloading a jar file

```
public class CodeDownLoad extends HttpServlet {
   public void doGet(HttpServletRequest req, HttpServletResponse res) {

      res.setContentType("application/jar");
      ServletContext ctx = getServletContext();
      InputStream is = ctx.getResourceAsStream("/newPluginJar.jar");

      // variables snipped ...

      OutputStream os = res.getOutputStream();
      while ((read = is.read(bytes)) != -1) {
        os.write(bytes, 0, read);
   }
   os.flush();
   os.close();
}
```

e.g., Generating image:

`http://today.java.net/pub/a/today/2004/04/22/images.html`

J2EE design guidelines: Web tier

Java Server Pages (JSP) design guidelines:

- JSPs cannot create binary content
- Use custom/standard tags to avoid scriptlets (<% ... %>)
  - Scriptlet code is not reusable
  - Encourages copying and pasting
  - Mix programming logic with presentation
  - Make JSP pages difficult to read and maintain
  - Errors are difficult to interpret
- Use Include Directives and Actions appropriately
  - Difference between directives and actions ?!?!
  - e.g., Directive: <%@include file="header.jsp" %>
  - e.g., Action: <jsp:include page="/servlets/currentUserInfoServlet" />
  - e.g., Action: <jsp:userBean>, <jsp:setProperty>, <jsp:getProperty>

J2EE design guidelines: Web tier

Avoid forwarding requests from JSP - use a Servlet instead

eg., JSP acting as a Controller:

```
<% String creditCard  = request.getParameter("creditCardType");
   if (creditCardType.equals("Visa")) { %>
      <jsp:forward page="/getVisaDetails.jsp"/>
  <% } else if (creditCard.equals("American Express")) { %>
      <jsp:forward page="/getAmexDetails.jsp"/>
  <% } %>
```

Bad!! This JSP is 'processing' business logic/application flow rather than displaying content.

A Better Design ...

In **CreditCardServlet.java**:

```
public class CreditCardServlet extends HttpServlet {
   protected void processRequest(HttpServletRequest request, ...
      String creditCard = request.getParameter("creditCardType");
      if (creditCardType.equals("Visa"))
         nextPage="/getVisaDetails.jsp";
      else if (creditCardType.equals("American Express"))
         nextPage = "/getAmexDetails.jsp"
      ServletContext ctx = getServletContext();
      ctx.getRequestDispatcher(nextPage).forward(request,response);
}
```

In **getVisaDetails.jsp**

```
<%@page contentType="text/html"%>
<html><head><title>Processing Visa</title></head>
<body>
   Enter your Visa card details ...
   <form action="processCreditCardServlet" method="POST">
       <input type="text" name="VisaNumber">
       <input type="text" name="VisaAcctName">
       <input type="text" name="VisaSecurityCode">
       <input type="submit">
   </form>
</body></html>
```

General Guideline for Servlet/JSP/JavaBeans

1. The action from HTML forms points to a servlet
2. The servlet uses request.getParameter("...") to get the details of the action
3. The servlet invokes appropriate business logic which returns data (typically JavaBeans or collections of JavaBeans)
4. The servlet then uses request.setAttribute("data", data) to store the data in request scope
5. or use session object if data belongs in a session
6. The servlet uses RequestDispatcher to call a JSP page that is supposed to handle the display of the data
7. The called JSP has access to the JavaBeans in request scope (or session)

This is a basic scenario of the MVC (Model-View-Control) pattern in J2EE

Structuring Web applications

- **A Model 1 architecture** consists of a Web browser directly accessing Web-tier JSP pages. The JSP pages access Web-tier JavaBeans that represent the application model, and the next view to display (JSP page, servlet, HTML page, and so on) is determined either by hyperlinks selected in the source document or by request parameters.
- **A Model 2 architecture** introduces a controller servlet between the browser and the JSP pages or servlet content being delivered. The controller centralizes the logic for dispatching requests to the next view based on the request URL, input parameters, and application state. The controller also handles view selection, which decouples JSP pages and servlets from one another.

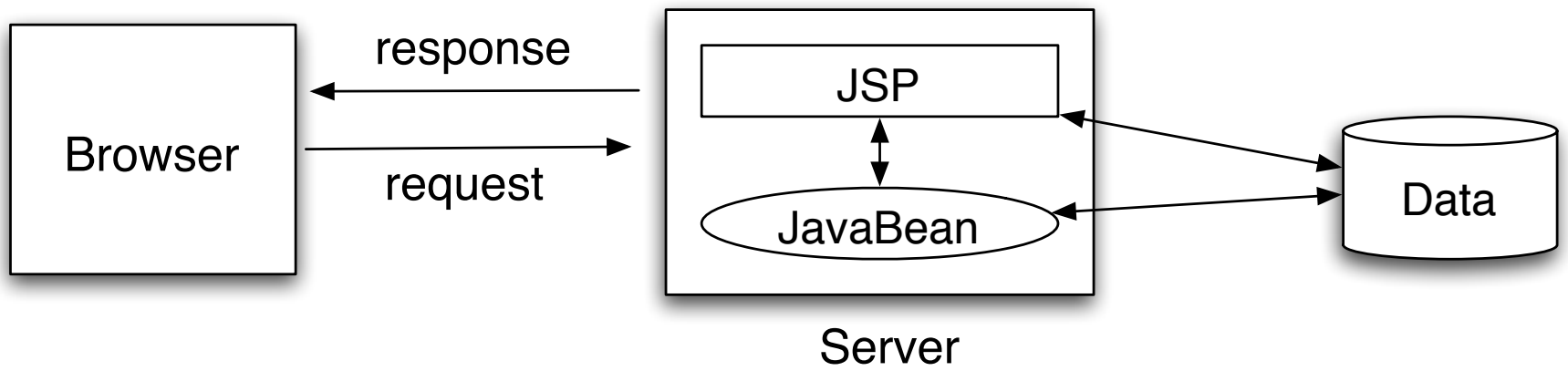
Model 1 and Model 2 simply refer to the absence or presence (respectively) of a controller servlet that dispatches requests from the client tier and selects views.

Web Application Structure Model 1 Example

Model 1 Architecture:

No distinct separation of presentation/business layers:

- The application consists of a series of JSP pages
- The user is expected to proceed from the first page to the next
- There may be JavaBeans performing business operations
- But each JSP page contains logic for processing its own output and maintaining application flow



Model 1 Architecture Example

Scenario: Processing login (three JSP pages, one JavaBean)

Login.jsp

```
<HTML><BODY>
<%
   if (request.getParameter("error")!=null) {
%>
Login failed. Please try again
<BR><HR>
<%
}
%>
<FORM METHOD="POST" ACTION="ProcessLogin.jsp">
  User Name: <INPUT TYPE=TEXT NAME="un">
  Password: <INPUT TYPE=PASSWORD NAME="pw">
  <INPUT TYPE=SUBMIT VALUE="Login">
</FORM>
</BODY></HTML>
```

Login.jsp calls ProcessLogin.jsp

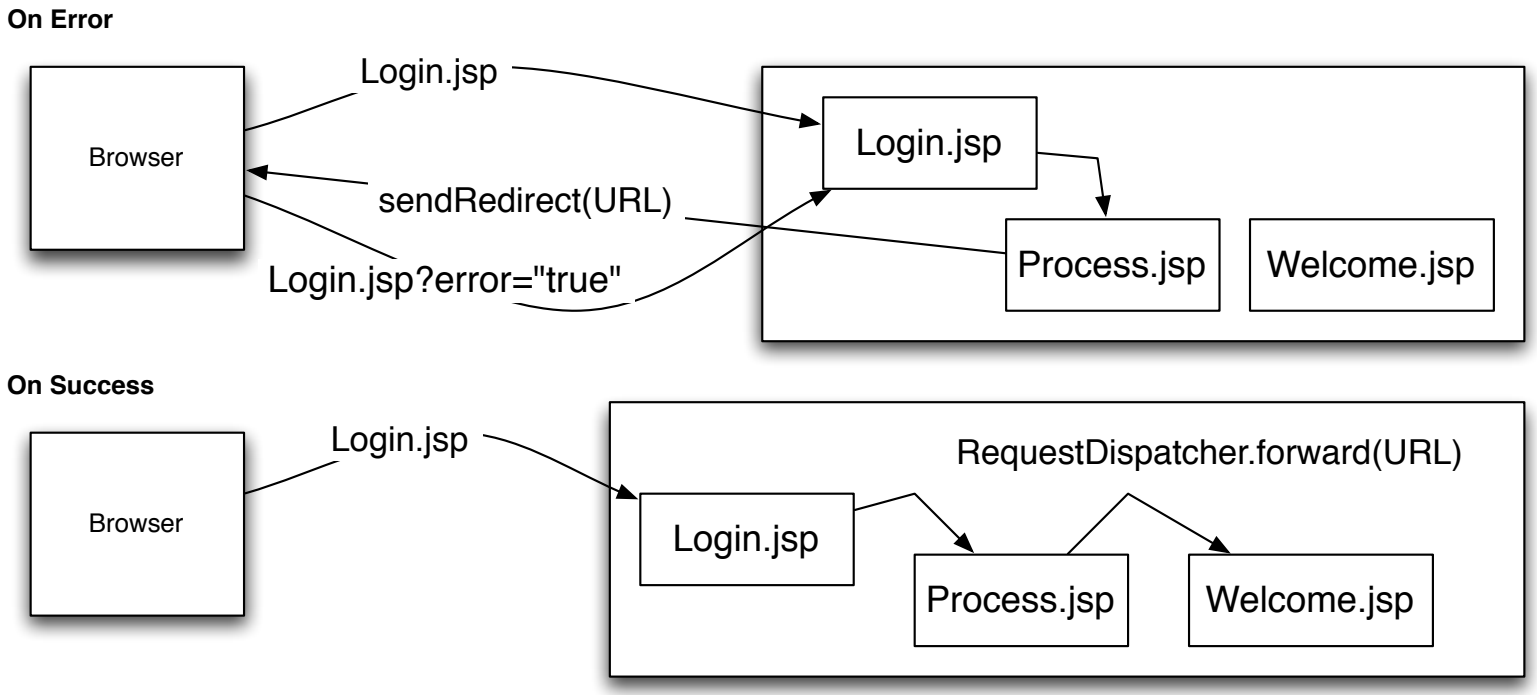
This example is from: Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions, by Budi Kurniawan, Chap 17.

Model 1 Architecture Example

ProcessLogin.jsp (calls either Login.jsp or Welcome.jsp)

```
<jsp:useBean id="loginBean" scope="page" class="model1.LoginBean" />
<%
if (loginBean.login(request.getParameter("un"), request.getParameter("pw")))
   request.getRequestDispatcher("Welcome.jsp").forward(request, response);
else
   response.sendRedirect("Login.jsp?error=yes");
%>
```

Login Process Scenario:



Model 1 Architecture Example

Welcome.jsp

```
<HTML>
<HEAD><TITLE>Welcome</TITLE></HEAD>
<BODY>
Welcome. You have successfully logged in.
</BODY></HTML>
```

LoginBean.java

```
package model1;

// note - this bean could use a database to check the login details
public class LoginBean {
   public boolean login(String userName, String password) {
   if (userName==null || password==null)  ||
                      !(userName.equals("koala") && password.equals("kitada")))
          return false;
    else
          return true;
   }
}
```

About Model 1 Architecture

- The model is easy to development.
- suitable for small projects (quick)
- It is hard to achieve division of labor between the page designer and the web developer because normally the web developer needs to be involved in the development of the page and the business objects.
- Model 1 architecture is hard to maintain and it is not flexible. This is especially true for large projects.

Java Sun Blueprint doc: The Model 1 architecture can provide a more lightweight design for small, static applications. Model 1 architecture is suitable for applications that have very simple page flow, have little need for centralized security control or logging, and change little over time. Model 1 applications can often be refactored to Model 2 when application requirements change.

Model 2 Architecture = MVC pattern

A Web application architecture pattern needs to:

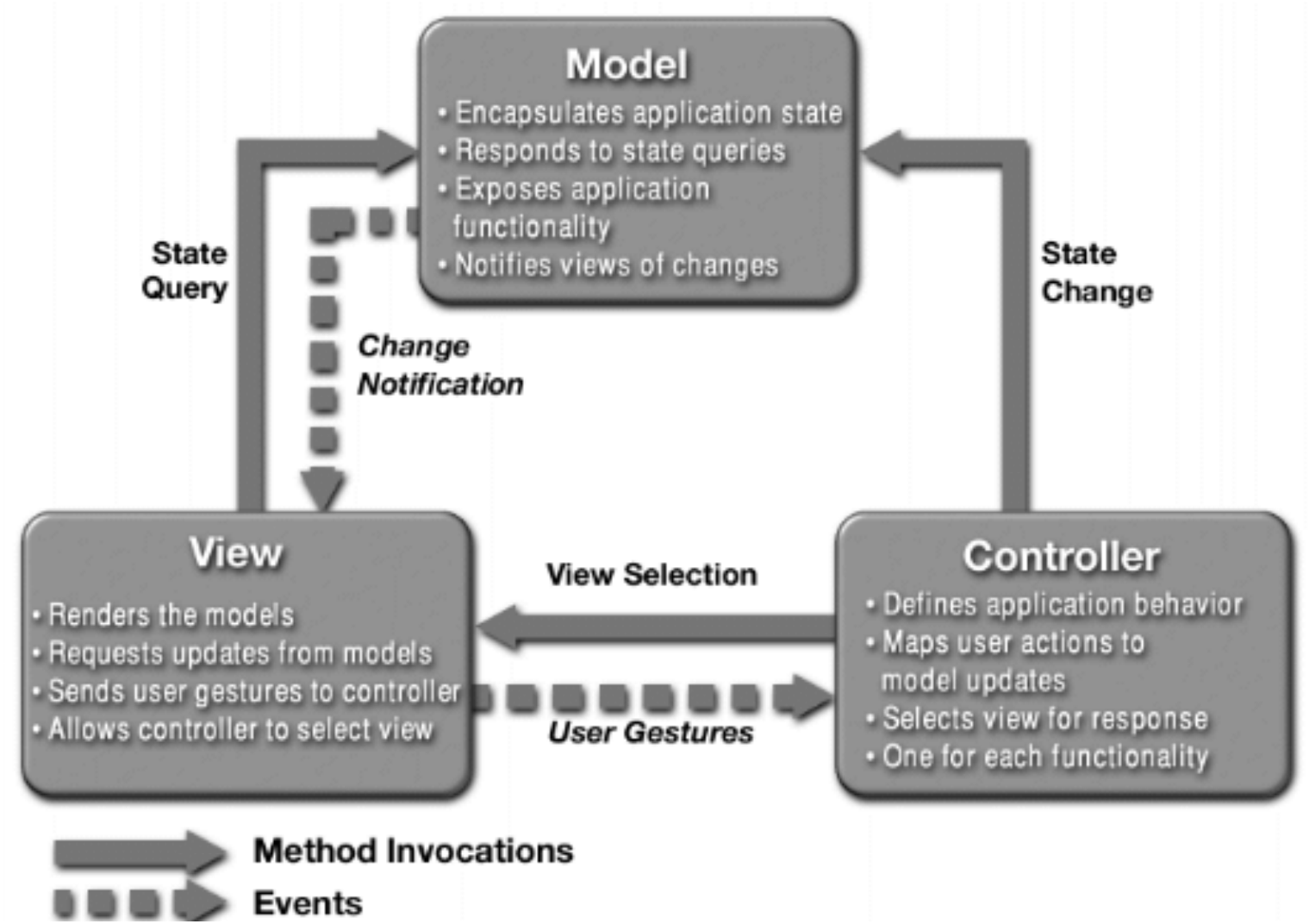
Problem:

- separate what the data that the clients sees with how it is presented
- support multiple client types (same data but presented differently).
- avoid placing business logic and application flow on client code

Solution:

- Separate data from view and use a controller to handle application flow between components
- View is used to present the data it is given
- Model represents data and some business logic
- Controller is used to handle client requests and invoke the appropriate business logic

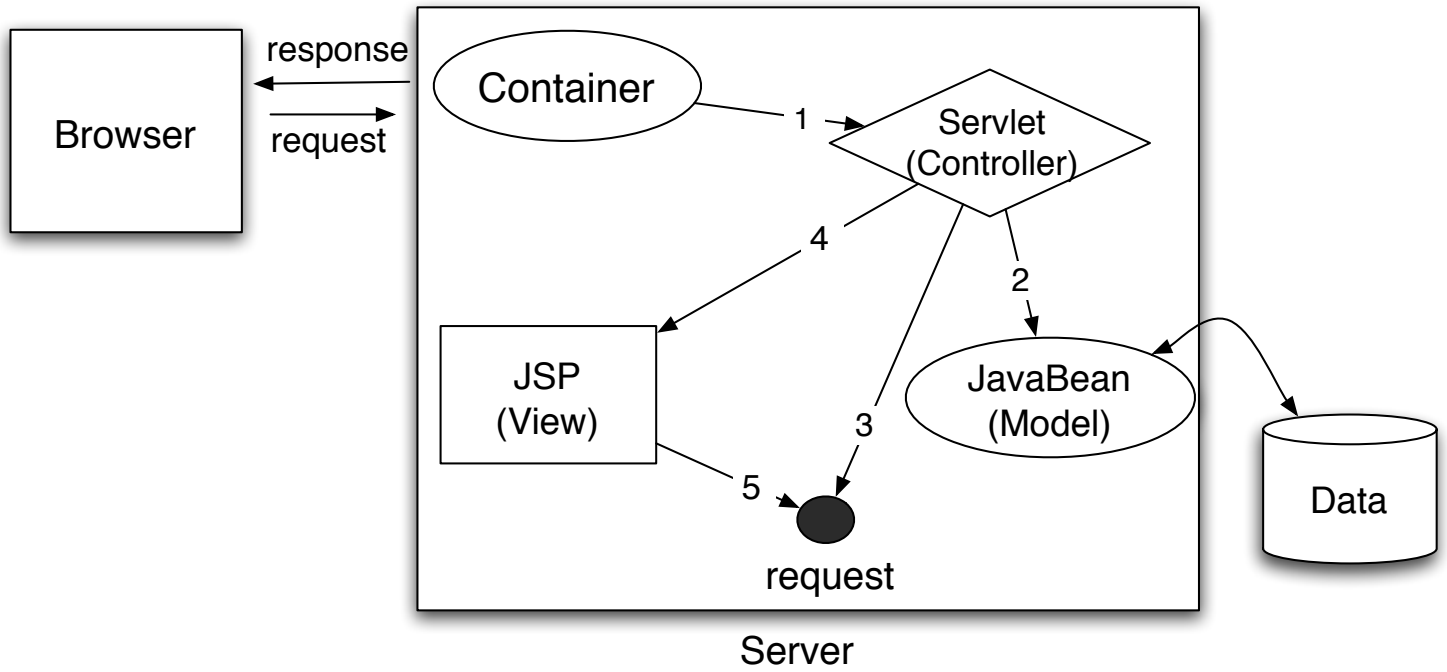
MVC Components and Roles



http://java.sun.com/blueprints/patterns/MVC-detailed.html

Model 2 Architecture = MVC pattern

- At the center is the controller servlet (dispatcher), which is the single entry point of the application.
- presentation parts (JSP pages) are isolated from each other.



- MVC separates content generation and content presentation.
- Model 2 is more flexible and easier to maintain, and to extend, because views do not reference each other directly.

Model 2 Architecture = MVC pattern

This servlet is the entry point to the scenario application.

LoginServlet.java

```
public class LoginServlet extends HttpServlet {
 public void doPost(HttpServletRequest request, HttpServletRespo... {
   String un = request.getParameter("un");
   String pw = request.getParameter("pw");
   if (un==null) {
     request.setAttribute("error", "no"); //fist time login
     RequestDispatcher rd = request.getRequestDispatcher("/Login.jsp");
     rd.forward(request, response);
  }
  else {
    if (pw!=null && un.equals("aibo") && pw.equals("kitada")) {
       RequestDispatcher rd = request.getRequestDispatcher("/Welcome.jsp");
       rd.forward(request, response);
    }
    else {  // login failed
       request.setAttribute("error", "yes");
       RequestDispatcher rd = request.getRequestDispatcher("/Login.jsp");
       rd.forward(request, response);
    }
}
```

Model 2: Model View Controller (MVC)

Login.jsp

```
<HTML><BODY>
<%  String err = (String)request.getAttribute("error");
        if (err.equals("yes"))  {  %>
           Login failed. Please try again
          <BR><HR>
<%
        }
%>
<FORM METHOD="POST">
  User Name: <INPUT TYPE=TEXT NAME="un">
  Password: <INPUT TYPE=PASSWORD NAME="pw">
  <INPUT TYPE=SUBMIT VALUE="Login">
</FORM></BODY></HTML>
```

- Login.jsp does not call anything (i.e., ProcessLogin.jsp is redundant now)
- all JSP isolated (co-ordinated through the controller)
- all JSP are only concerned with displaying info.

Case for a centralised controller

Even with MVC pattern, maintaining a Web application can be difficult

- Maintaining links between different pages (views)
- Duplication of functionality (e.g. validation,authentication) at different places
- Maintaining access to Data Sources
- Updating layouts

Solution: Have a central (single) controller

- Centralize functions such as authentication, validation, etc.
- Maintain central database of page templates
- View selection is in one place
- Updating links requires updating in just one place

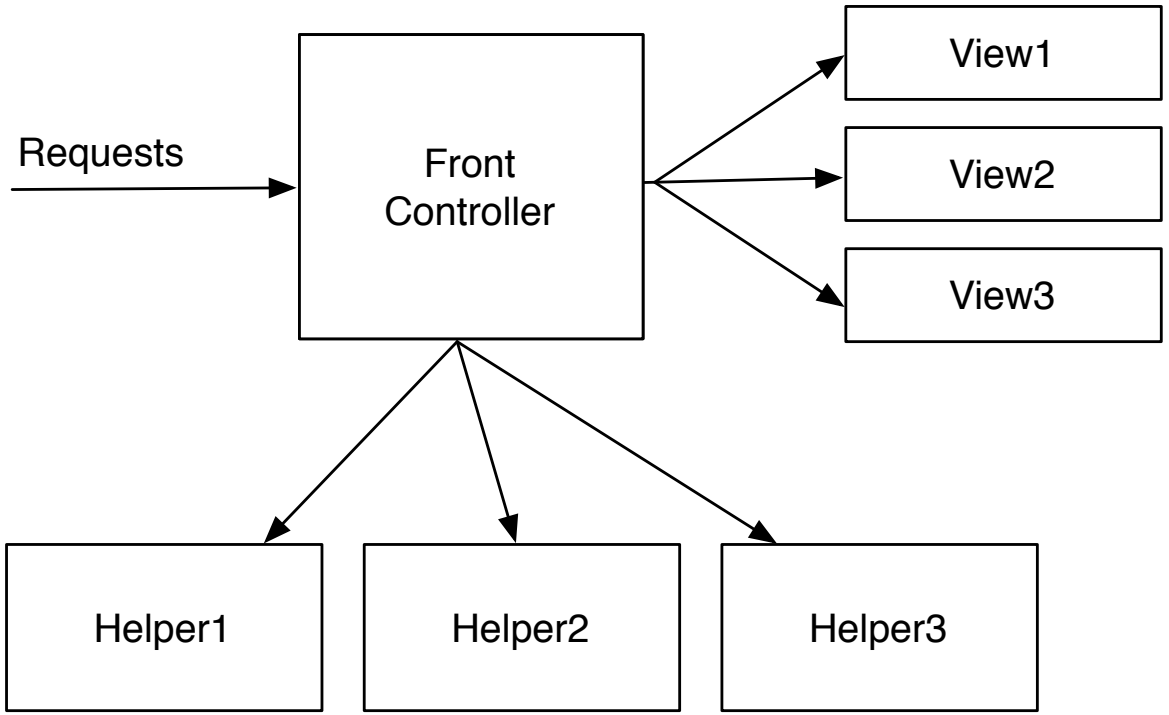
Case for multiple controllers

However, using a centralised controller has its disadvantages

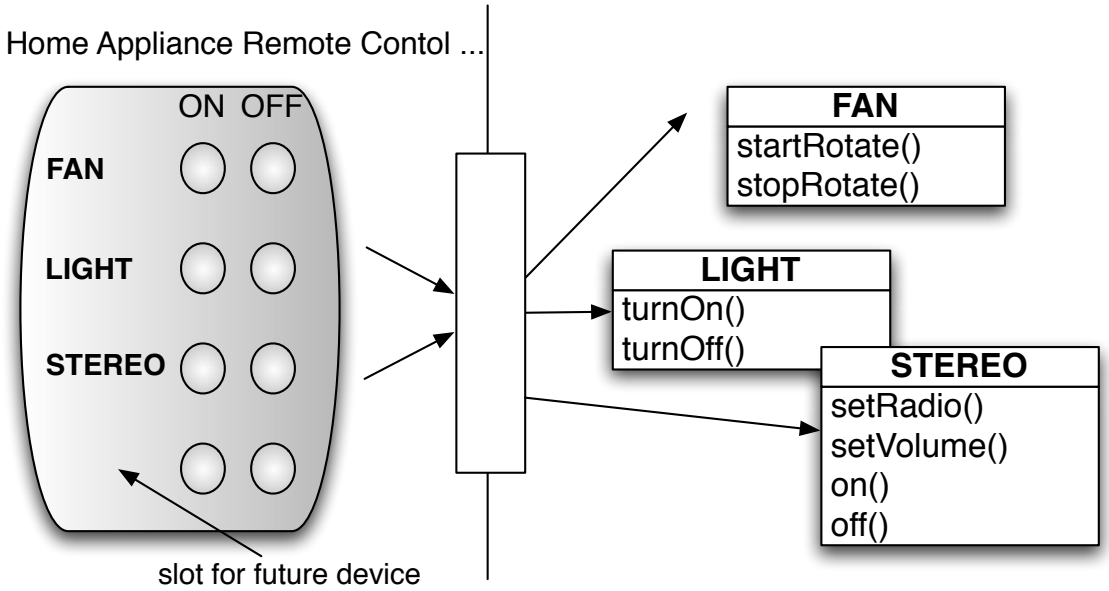
- Centralised controllers can become heavy, with too much logic in a single class file
- Multiple developers working on the Controller simultaneously
- Testing becomes difficult
- Certain sections of the web application may require different implementations of the same features (e.g. admin login vs normal user login)

Some frameworks solve this by having a controller per logic unit (or feature group). For example, there may be a controller devoted to the admin functionality and covers all the views for an admin user.

FrontController pattern



FrontController implementation: Command Design Pattern



- the remote should know how to interpret button presses and make requests
- but should not know how to operate FAN or STEREO or any future new devices ...
- want to avoid 'if slot1==FAN, then FAN.startRotate() else if 'slot1==LIGHT then light.on()', etc.

Command Design Pattern

Problem:

- Want to issue requests without knowing who or what will service it
- Want to service a request without knowing who/what requested it
- Avoid too many if statements $\rightarrow$ code bloat

Solution:

- Need to decouple the sender and receiver
  - A sender is an object that invokes an operation, and
  - a receiver is an object that receives the request (ie., running a command) to execute a certain operation.
  - With decoupling, the sender has no knowledge of the Receiver's interface.
- Create a generic Command Object
  - The key to the pattern. It declares an interface for executing operations

Command Design Pattern

www.javaworld.com/javaworld/javatips/jw-javatip68.html

Two "Receiver" classes (different behaviour)

```
class Fan {
     public void startRotate()
     public void stopRotate()
}
class Light {
     public void turnOn( )
     public void turnOff( )
}
```

Say, we want an "Invoker" class `Switch` that will work with both.

```
class Switch {
     void flipUp( )
     void flipDown( )
}
```

The idea is calling `Switch().flipUp()` will call either `Fan().startRotate()` or `Light().TurnOn()`. That is, `Switch` should be independent of the specific Receiver class' interface.

Command Design Pattern

We start with designing the Command interface.

```
public interface Command {
        public abstract void execute ( );
}
```

Then, we implement the Commands. Each concrete Command class specifies a "receiver-action" pair by storing the Receiver as an instance variable.

```
class LightOnCommand implements Command {
        private Light myLight;  // here, the receiver class
        public LightOnCommand (Light L) { myLight  =  L; }
        public void execute( ) { myLight.turnOn( ); }  // here, its action
}

class FanOnCommand implements Command {
        private Fan myFan;  // here, the receiver class
        public FanOnCommand ( Fan F) { myFan  =  F; }
        public void execute( ) {  myFan.startRotate( ); } // here, its action
}
// Snip LightOffCommand and FanOffCommand classes
```

Command Design Pattern

The Invoker: Now the `Switch` looks like this.

```
class Switch {

   private Command UpCommand,
   private Command DownCommand;

   public Switch(Command Up, Command Down) {
              // concrete Command gets registered when invoker is created
      UpCommand = Up;
      DownCommand = Down;
   }
   void flipUp( ) {
            // invoker calls concrete Command,
            // which executes the Command on the receiver
      UpCommand.execute ( ) ;
   }

   void flipDown( ) {
      DownCommand.execute ( );
   }
}
```

Command Design Pattern

A Client Program:

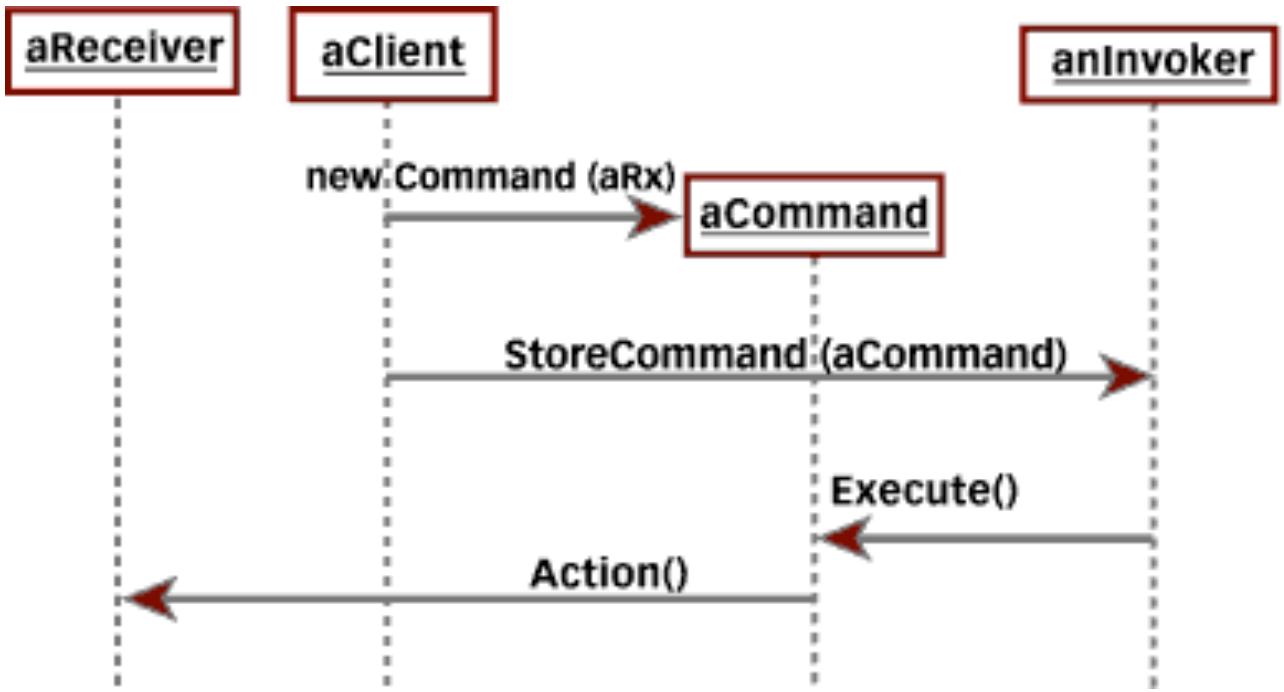
```
Light  testLight = new Light( );
LightOnCommand testLOC = new LightOnCommand(testLight);
LightOffCommand testLFC = new LightOffCommand(testLight);
Switch lightSwitch = new Switch(testLOC, testLFC);
Fan testFan = new Fan( );
FanOnCommand testFOC = new FanOnCommand(testFan);
FanOffCommand testFFC = new FanOffCommand(testFan);
Switch fanSwitch = new Switch(testFOC, testFFC);
// using the switches
fanSwitch.flipUp( );
fanSwitch.flipDown( );
lightSwitch.flipUp( );
lightSwitch.flipDown( );
```

the Command pattern completely decouples the object that invokes the operation (Switch) from the ones having the knowledge to perform it (Light and Fan).

This gives us a lot of flexibility: the object issuing a request must know only how to issue it; it doesn't need to know how the request will be carried out.

Command Design Pattern

- The key to this pattern is a Command interface, which declares an interface for executing operations.
- Each concrete Command class specifies a "receiver-action" pair by storing the Receiver as an instance variable.
- The Receiver has the knowledge required to carry out the request



Identifying the Command Pattern in the phonebook lab

- **The Receiver**: whoever knows how to carry out the command (eg., ContactDelegate)

```
class ContactDelegate {
   public UserBean login()
   public List getRecords()
   public void addRecord()
   public void deleteRecord()
}
```

- **Command Interface**: the command interface has one method `execute()`
- **Concrete commands**: AddCommand, DeleteCommand, ListCommand and LoginCommand.

```
class AddCommand {
   private static ContactDelegate cd; // here, the receiver
   public String execute() // here, the action: calls cd.addRecord()
}
```

- **The Invoker** is the ControllerServlet class.

Implementing the Command Pattern

The invoker in the phonebook lab

```
public class ControllerServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
      commands = new HashMap();
      commands.put("add", new AddCommand());
    }

    protected void processRequest(HttpServletRequest request, ...) {
      Command cmd = resolveCommand(request);
      String next = cmd.execute(request, response);
      RequestDispatcher ds = getServletContext().getRequestDispatcher(next);
      ds.forward(request, response);
    }

    private Command resolveCommand(HttpServletRequest request) {
      Command cmd = (Command) commands.get(request.getParameter("operation"));
      return cmd;
    }
```

Service Locator Pattern

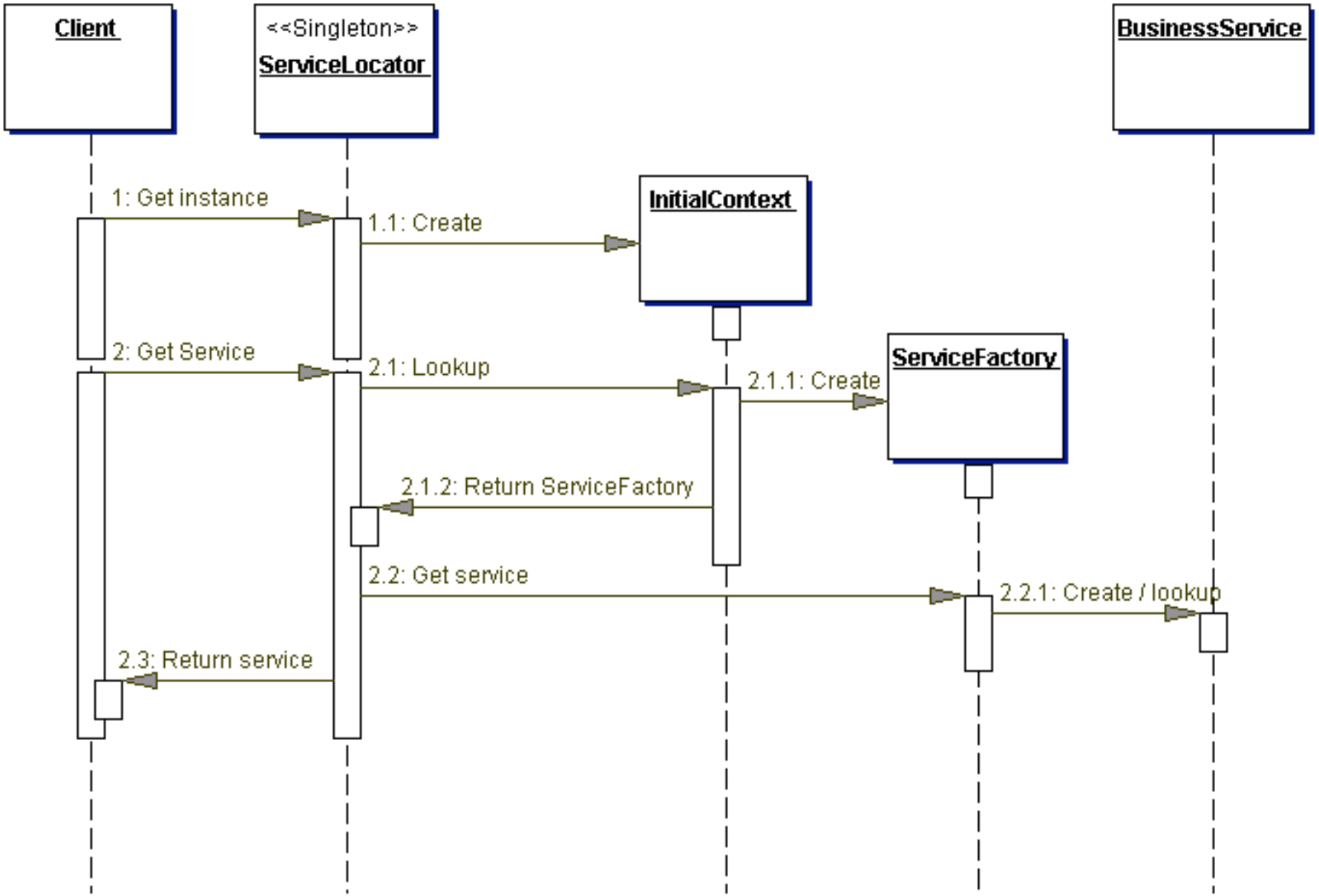
Problem:

- When J2EE clients interact with the server side components (EJB) or DataSources, clients must locate the service component (referred to as a lookup operation in JNDI)
- locating a JNDI-managed service object is common to all clients that need to access that service object.
- It is easy to see that many types of clients repeatedly use the JNDI service, and the JNDI code appears multiple times across these clients. This results in an unnecessary duplication of code in the clients that need to look up services.

Solution:

- Use a Service Locator object to abstract all JNDI usage to hide the complexities of initial context creation and lookup operations
- Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control

Service Locator Pattern



Service Locator Pattern

To build a service locator pattern, we need:

- **Service Locator**: The Service Locator abstracts the API lookup services, vendor dependencies, lookup complexities, and business object creation, and provides a simple interface to clients.
- **InitialContext**: The InitialContext object is the start point in the lookup and creation process.
- **ServiceFactory**: The ServiceFactory object represents an object that provides life cycle management for the BusinessService objects. eg., The ServiceFactory object for enterprise beans is an EJBHome object.
- **BusinessService**: is a role that is fulfilled by the service the client is seeking to access. The BusinessService object is created or looked up or removed by the ServiceFactory. The BusinessService object in the context of an EJB application is an enterprise bean. The BusinessService object in the context of JDBC is a DataSource

Identifying Service Locator Pattern in the phonebook lab

com.enterprise.common.AbstractJndiLocator: the Service Locator. Defines connection details to JNDI, creation of initialContext and lookup() operation

```
public abstract class AbstractJndiLocator {

   // connection details to JNDI
   private String initialContextFactory = "org.jnp.interfaces.NamingContextFactory"
   private String urlPkgPrefix= "org.jboss.naming:org.jnp.interfaces";
   private String url = "jnp://localhost:1099";

   // get initialContext
   public AbstractJndiLocator()  {
        ctx = new InitialContext(initialContextFactory, urlPgkprefix, url);

   // Lookup operation
   public Object lookup(String name) throws NamingException {
       Object o = ctx.lookup(name);
       return o;
}
```

Identifying Service Locator Pattern in the phonebook lab

The service locator is abstract. DBConnectionFactory inherits from it.

com.enterprise.common.DBConnectionFactory: It does resource specific lookup. This lookup is for retuning a JDBC DataSource

```
public class DBConnectionFactory extends AbstractJndiLocator {

private DataSource ds;

public Connection createConnection() {
        return getDataSource().getConnection();
   }

public DataSource getDataSource()  {
        //JNDI name for the data source
        ds = (DataSource) lookup("java:/DefaultDS");
        return ds;
    }
}
```

(you could add, for example, EJBHomeFactory.java that extends AbstractJndiLocator but gives you EJB home objects this time ...)

Identifying Service Locator Pattern in the phonebook lab

The client code (DAO implementation class) calls createConnection() on DBConnectionFactory to get a data source.

```
public class ContactDAOImpl implements ContactDAO {

   private DBConnectionFactory services;

   public ContactDAOImpl() {
       services = new DBConnectionFactory();
   }

   public void insert(ContactBean bean) {
        // get connection by calling createConnection() on DBConnectionFactory
        // Then do your normal JDBC actions
    }
}
```

Dependency Injection

In any non-trivial Web application, there may exist many dependencies such as:

- Database Connection information
- Location of resources
- Connection information for external services
- Multiple implementations for the same interface
- Multiple modes of instantiation of service objects

Often, these are explicit and expressed through abstract classes and inheritance.
However, in many cases, configuration dependencies are implicit and scattered throughout the code. This makes it very hard to test, maintain and upgrade the web application.

SAX Books Parser Example

As an example, let's take the SAX Books Parser Example provided in Week 4.

The `SAXServlet` obtains a list of `BookBeans` from the `BookParser` and forwards it on to a JSP file. Let's say that we want to test the `BookParser` alone, outside of the Servlet Container.

This is currently not possible since `SAXServlet.init()` locates the XML file and provides an `InputStream` object to the parser.

What if we want to replace the XML file with a SQL database ?....

What is "dependency injection" ?

Dependency injection is a software design pattern that provides the ability to pass by reference (or "inject") service objects into a client (a class or a delegate) at deployment time. This is a top-down approach, in contrast to a bottom-up one wherein the clients discover or create service objects on their own. The benefits of dependency injection are:

- Loose coupling of different parts of a web application - avoid monolithic architecture.
- Provide modularity wherein one service or a component can be replaced with another implementation at deployment time.
- Provide strong typing for different services and reduce the amount of string-based lookups by replacing them with declarative Java annotations.
- Provide a central location for modifying configuration data.
- Enable unit testing of different components and easy mocking of different services.

More patterns

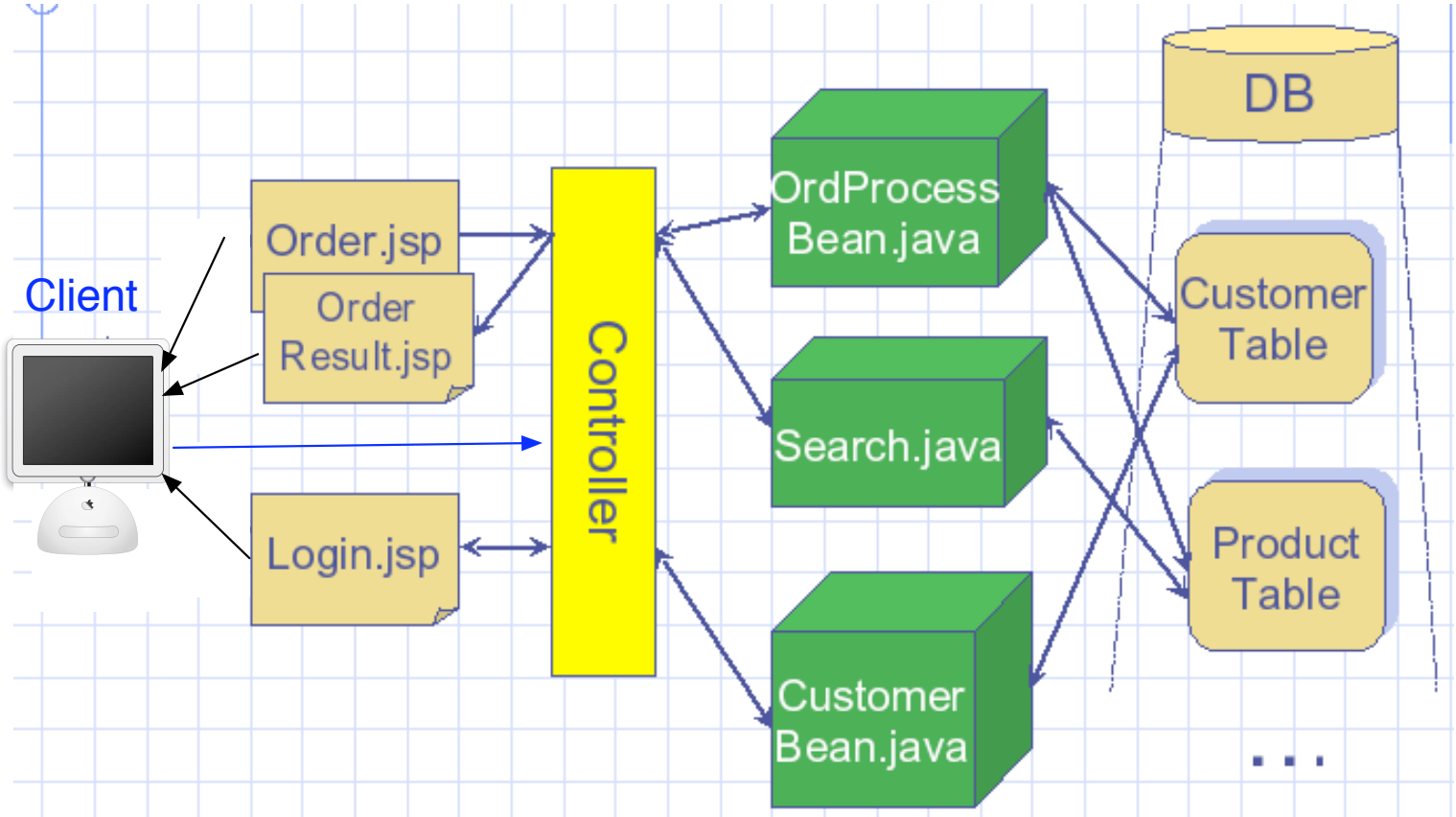
Core J2EE Patterns Catalog

`http://java.sun.com/blueprints/corej2eepatterns/index.html`

The above web site describes every pattern (problem/solution/sample codes) and their relationships.

Assignment 2 Architecture

- presentation, business, data access layer
- at least include the MVC pattern and DAO, plus others



A few more things to consider

Controlling Client Access (Guarding a View)

Scenario: You may want to restrict or control client access, e.g.,

- Only logged in users should access page X and Y
- page X should be accessed only after page B and C
- page X and Y should only be accessed by users with role Z

Basically, you want to prohibit users from directly accessing certain pages via a browser invocation.

A few strategies available ...

- Embedding a "*guard*" directly within a view
- Using a controller as a delegation point for this type of access control
- Using standard security constraints (e.g., J2EE security)

Guarding a View

Embedding a "guard" within View:

- All-or-Nothing Guard: When you want to prevent a particular user from accessing a particular view in its entirety; e.g., if the client must still be logged into the site to view page X

```
// Inside page X
<% taglib uri="/tld/ViewGuard.tld" prefix:guard" %>

<guard:check_login/>
<HTML>
...
</HTML>
```

- Portions of View Not Displayed Based on User Role:

```
// display suppliers here ... then:
<c:if test="${UserRole eq 'manager'}">
    <a href="updateDetails.jsp"/>Update Supplier</a>
</c:if>
```

- The code for access check is distributed (maintenance?)

Guarding a View

Using the controller:

```
public class ControllerServlet extends HttpServlet {

        // dispatching the request to the handler ...
        if(VIEW_CAR_LIST_ACTION.equals(actionName))
            destinationPage = "/carList.jsp";
        else(ADD_CAR.equals(actionName) and checkUserRole().equals="manager")
            destinationPage = "/AddCart.jsp";
        else ... // more action and role checking here ...

        RequestDispatcher ds =
                    getServletContext().getRequestDispatcher(destinationPage);
        ds.forward(request, response);
}
```

(suitable for All-or-Nothing access)

Guarding a View

Using the container's security constraints: J2EE supports 'declarative security/role-based security' model. You can configure the container to allow/prevent access to certain resources

```
<security-constraint>
     <web-resource-collection>
         <url-pattern>/restricted/*</url-pattern>
          <url-pattern>/manager/addCars.jsp</url-pattern>
          <http-method>GET</http-method>
          <http-method>POST</http-method>
     </web-resource-collection>
     <auth-constraint>
         <role-name>manager</role-name>
     </auth-constraint>
 </security-constraint>
 <login-config>
     <auth-method>BASIC</auth-method>
     <realm-name>Delicious Baking Company</realm-name>
 </login-config>
```

JBoss doc: 8.1.5. Web Content Security Constraints

`http://docs.jboss.com/jbossas/jboss4guide/r5/html/ch8.chapter.html`

Container Security Constraints

e.g., Authentication support in JBoss ...

```
CREATE TABLE Users(username VARCHAR(64) PRIMARY KEY, passwd VARCHAR(64))
CREATE TABLE UserRoles(username VARCHAR(64), userRoles VARCHAR(32))

INSERT INTO Users VALUES ('koala','j2ee')
INSERT INTO UserRoles VALUES ('koala','manager')
```

Then in a configuration:

```
<authentication>
   <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule">
         <module-option name="dsJndiName">java:/DefaultDS</module-option>
         <module-option name="principalsQuery">
             select passwd from Users where username=?
         </module-option>
         <module-option name="rolesQuery">
             select userRoles,'Roles' from UserRoles where username=?
         </module-option>
   </login-module>
</authentication>
```

The security framework is based on the *Java Authentication and Authorization Service* (JAAS API). To work with these configurations, you need to use JAAS API.

Guarding a View

A simple way to use the container to hide certain resources from client's direct access is to use `WEB-INF` directory.

- things under the `WEB-INF` directory cannot be accessed via browsers (container prohibits it)

  `http://localhost:8080/myapp/WEB-INF/restricted/welcome.jsp`

  vs.

  `http://localhost:8080/myapp/welcome.jsp`

- You can move certain resources into a subdirectory of `WEB-INF`
- Access to such resources can only be gained via *internal requests* (e.g., a controller servlet)
  e.g., in a servlet:

```
// if access should be given:
destinationPage = "/WEB-INF/restricted/welcome.jsp"
dispatch.forward (...)
```

- Additionally, the controller can delegate the access checks to a helper class who will determine whether the resource should be served

A few more things to consider

Duplicate Form Submissions

Users working with a browser can:

- use the Back button and inadvertently resubmit the same form they had previously submitted.
- click the stop button before receiving a confirmation page and subsequently resubmit the form
- In most cases, we want to trap and disallow these duplicate submissions

A few more things to consider

Duplicate Form Submissions

JavaScript (i.e., client side solution):

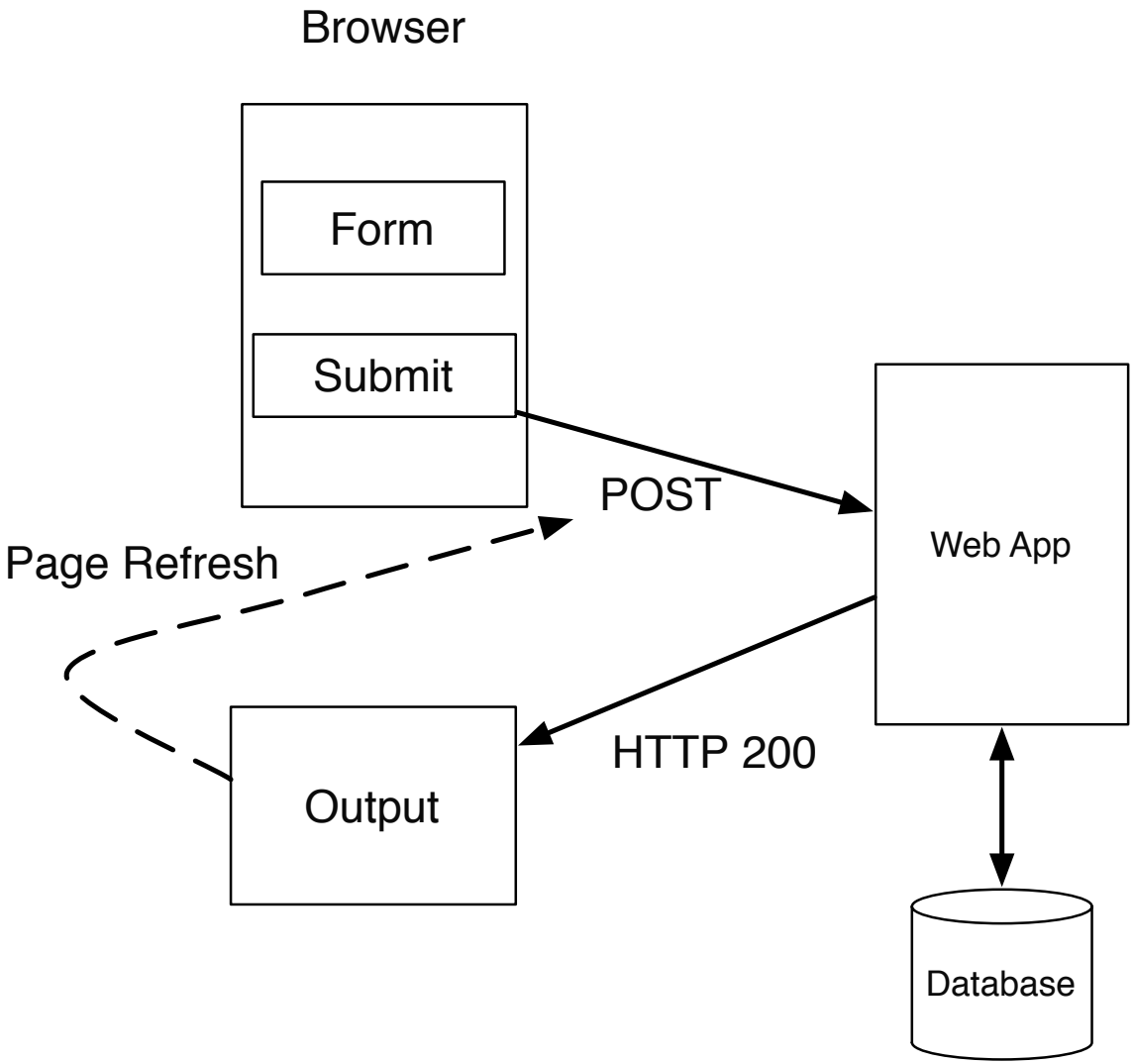
```
<SCRIPT LANGUAGE="JavaScript" TYPE="text/JavaScript">
function buttonControl(submitted)
{
   if(submitted=="1")
      theForm.Submit.disabled=true
}
</SCRIPT>

<FORM action="" method="post" name="theForm">
   How did you find our site?
   <input type="text" maxlength="30" size="20"><br/>
   <input type="submit" name="Submit" onClick="buttonControl(1)">
</FORM>
```

Reasonable ... but cannot rely on it, why?

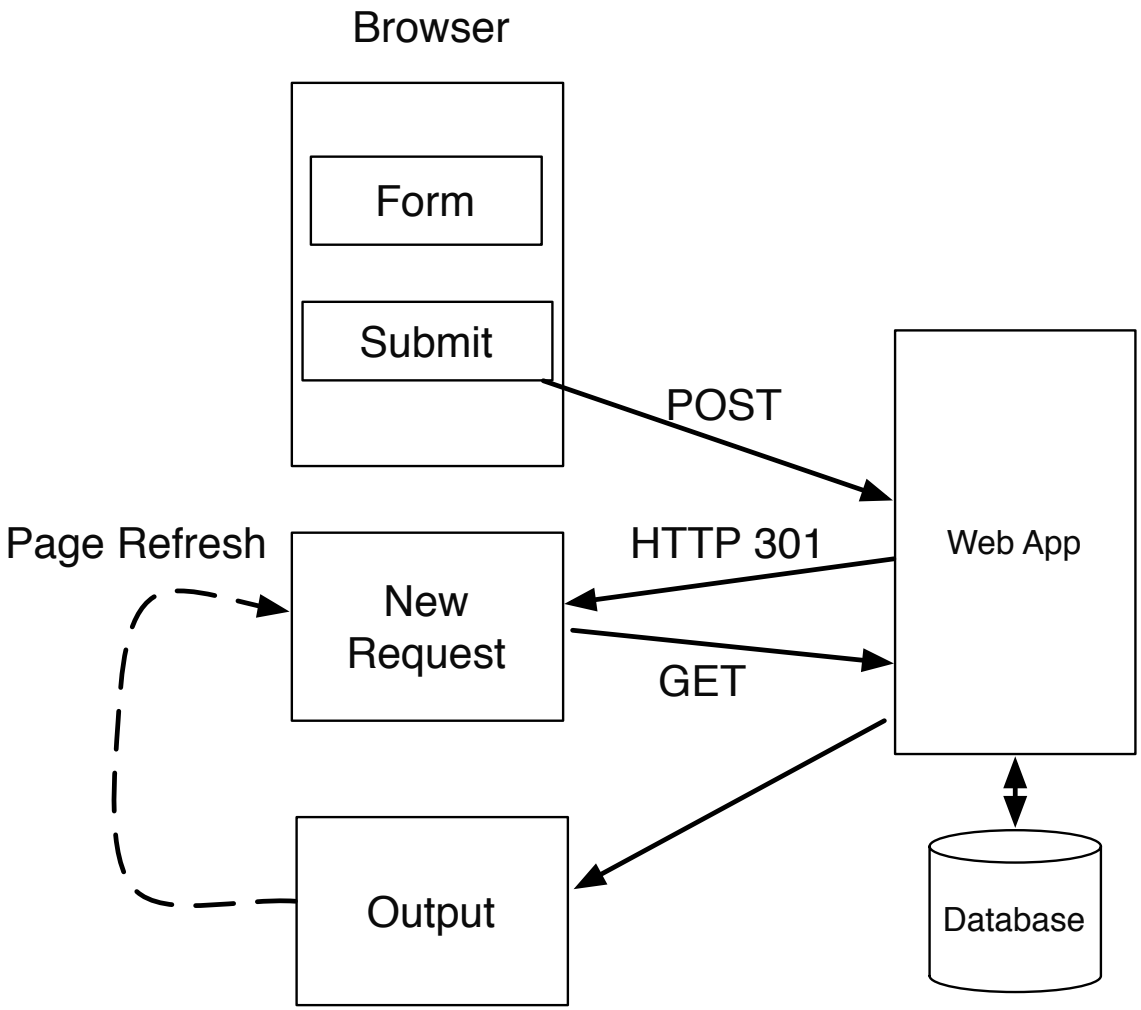
A few more things to consider

Duplicate Form Submissions - POST REDIRECT GET



A few more things to consider

Duplicate Form Submissions - POST REDIRECT GET



A few more things to consider

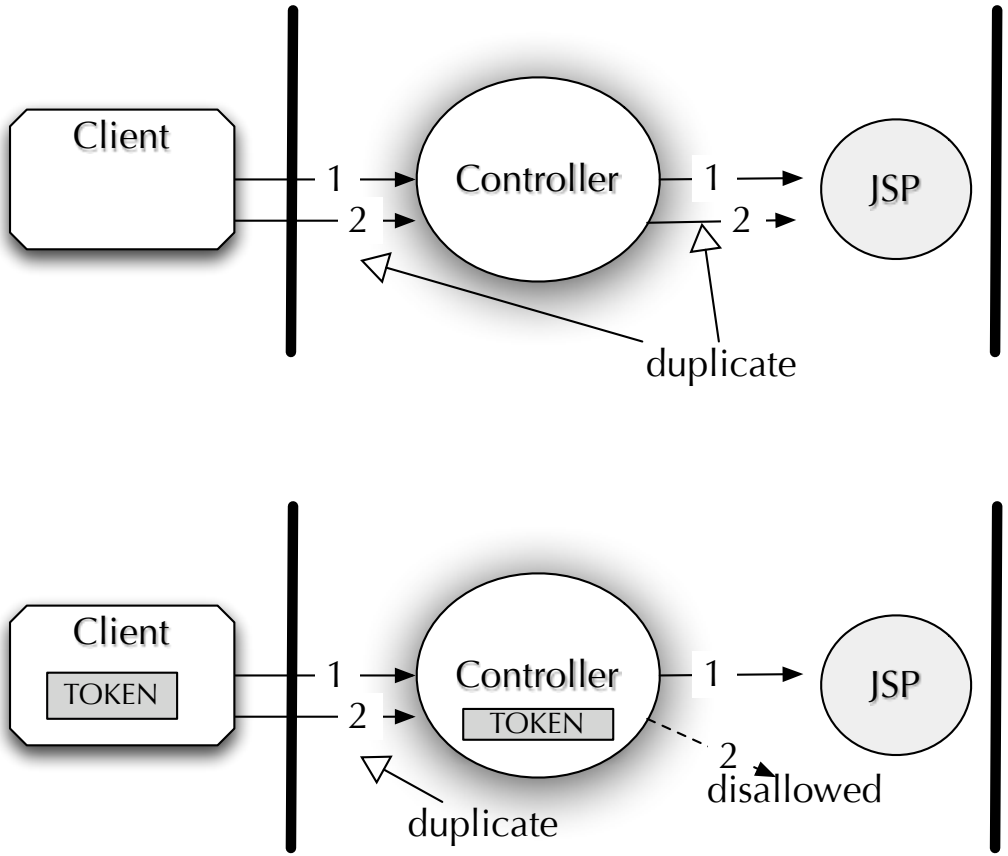
Duplicate Form Submissions

Synchronizer Token:

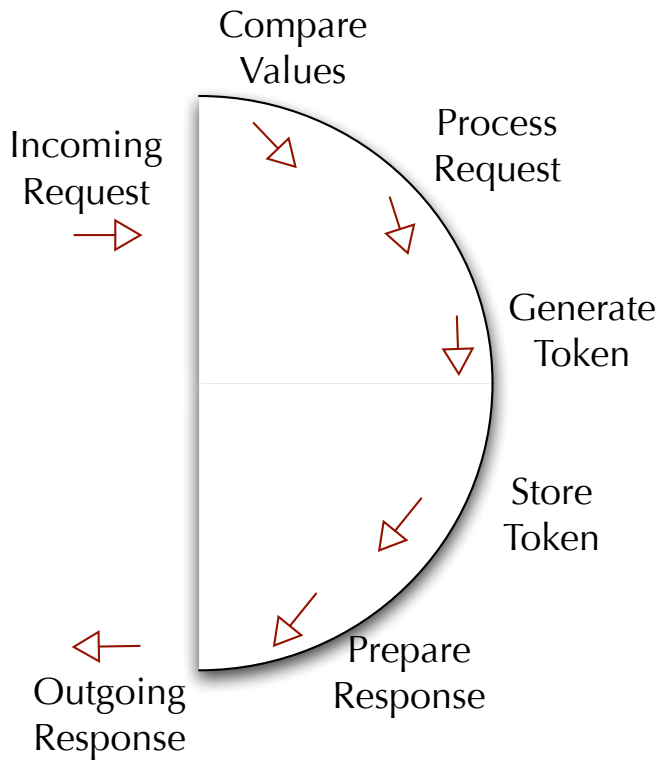
- a synchronizer token is set in a user's session
- the token is included with each form returned to the client (hidden field)
- When that form is submitted, the value of the token from the client is compared with that of in the user session.
- If it does not match, an error is returned to the user
- If it matches, the form submission is allowed
- At this point, the token value in the session is reset (i.e., new value) for the next use

A few more things to consider

Synchronizer Token:



<Synchronizer Token Lifecycle>



A few more things to consider

Synchronizer Token:

You could also use this strategy to control direct browser access to certain pages. For example, assume a user bookmarks page A of an application where page A should only be accessed from page B and C.

You synchronise the access flow using tokens.

When the user selects page A via the bookmark, the page is accessed out of order and the synchronizer token will be in an unsynchronized state or not exist at all.

A few more things to consider

Background Tasks:

Scenario: Background tasks can be required in many situations, e.g.

- You want to run a task periodically for collecting data or for maintenance. E.g. Generate a report every Saturday night
- You want to execute a function that could take a long time but do not want to make the user wait - performance.

Basic requirement - fire off a task from a servlet, let the servlet continue and the task should come back to you with results.

However, there are issues:

- How do you manage the task's progress ?
- How do you receive results ?
- Where do you fire off the tasks ?

Up until the recent past, this would've required dealing with threads...

A few more things to consider

Background Tasks:

A new concurrency library (`java.util.concurrent`) was introduced in Java 1.5. This provided some new (and much-needed) concurrent programming abstractions such as:

- `Callable<V>` - A task that returns a result
- `ExecutorService` - An object that can accept `Callable` and `Runnable` tasks and manage their execution
- `Future<V>` - Represents the results of a Callable that are available when the task completes execution
- `ScheduledExecutorService` - An `ExecutorService` that can run tasks after a delay or periodically

A few more things to consider

Background Tasks:

For example: A class to report number of users in a database every minute.

```
public class UpdateUsers implements Runnable{
        /* Methods for accessing database to
        get number of users */
}
```

```
ScheduledExecutorService s =
        Executors.newSingleThreadScheduledExecutor();
s.scheduleAtFixedRate(new UpdateUsers(), 0, 1, TimeUnit.MINUTE
```

Where do you put the scheduler ? In the ServletContextListener, if it is applicable throughout the web application
Take a look at java.util.concurrent for more examples..