

Spring Web Tier

Alef Arendsen
Erwin Vervaet

Practical session on Spring Web MVC and Spring Web Flow

- Alef Arendsen
 - ◆ Principal Consultant at Interface21
 - ◆ Spring Core Developer
 - ◆ Author, *Professional Development with Spring*
- Erwin Vervaet
 - ◆ Consultant at Ervacon
 - ◆ Extensive experience in J2SE and J2EE
 - ◆ Founder of the Spring Web Flow project

Agenda

- **Spring and Spring MVC**
- The Sample Application
- Data Binding
- Spring Web Flow
- Redirect-After-Post
- File Upload

Spring MVC - Topics

Spring
from the source

- Positioning of Spring MVC in the application architecture
- Spring MVC general concepts
- Request-processing flow



Spring MVC Introduction



- Request-driven MVC framework
- Some similarities with Struts
- Tight integration with other parts of Spring
- Basis for other HTTP-based functionality in Spring
- Can be used in combination with other MVC frameworks





Web MVC layer
(Spring MVC, Spring Web Flow
JSF, Struts, Tapestry, ...)

Service objects

Data access objects

- Browser issue requests: GET, POST
- Requests need to be *handled*
 - ◆ Invoke the appropriate business process
 - ◆ Example: search for list of items
- Responses must be *rendered*
 - ◆ Show the result of the business process
 - ◆ Example: display the items to the user

- The result:
 - ◆ **Controller** – handling requests, interacting with the business layer and preparing the model and the view
 - ◆ **View** – rendering the response to the user based on the Model
 - ◆ **Model** – the contract between the View and the Controller

Request-driven MVC – Controller

(or: everything you need to know about Spring MVC – 1/5)

Spring
from the source

- Processes logical requests
 - ◆ Parses and validates input
 - ◆ Invokes business logic
 - ◆ Creates model
 - ◆ Selects a view
- Often also called a *request processor* or *handler*



Request-driven MVC – Model

(or: everything you need to know about Spring MVC - 2/5)



- Data returned by the controller and rendered by the view
 - ◆ (In Spring) implemented as a `java.util.Map`
 - ◆ Technology-agnostic (e.g. no references to `HttpServletRequest`)



Request-driven MVC – View

(or: everything you need to know about Spring MVC – 3/5)

Spring
from the source

- Identified by a *logical view name* (a String)
- Renders responses to users
 - ◆ Accesses data in the Model
 - ◆ Renders a response
 - ◆ (Typically) does not contain business logic
 - Just simple 'scripts' to access and display Model data
 - ◆ Clear contract
 - Controller populates Model with data
 - View renders data from the Model



Request-driven MVC – ViewResolver

(or: everything you need to know about Spring MVC – 4/5)



- Resolves *logical view names* to
 - ◆ Java Server Pages
 - ◆ Velocity templates
 - ◆ Freemarker templates
 - ◆ Views rendering PDF or Excel files
 - ◆ XSLT files transforming XML
 - ◆ ...



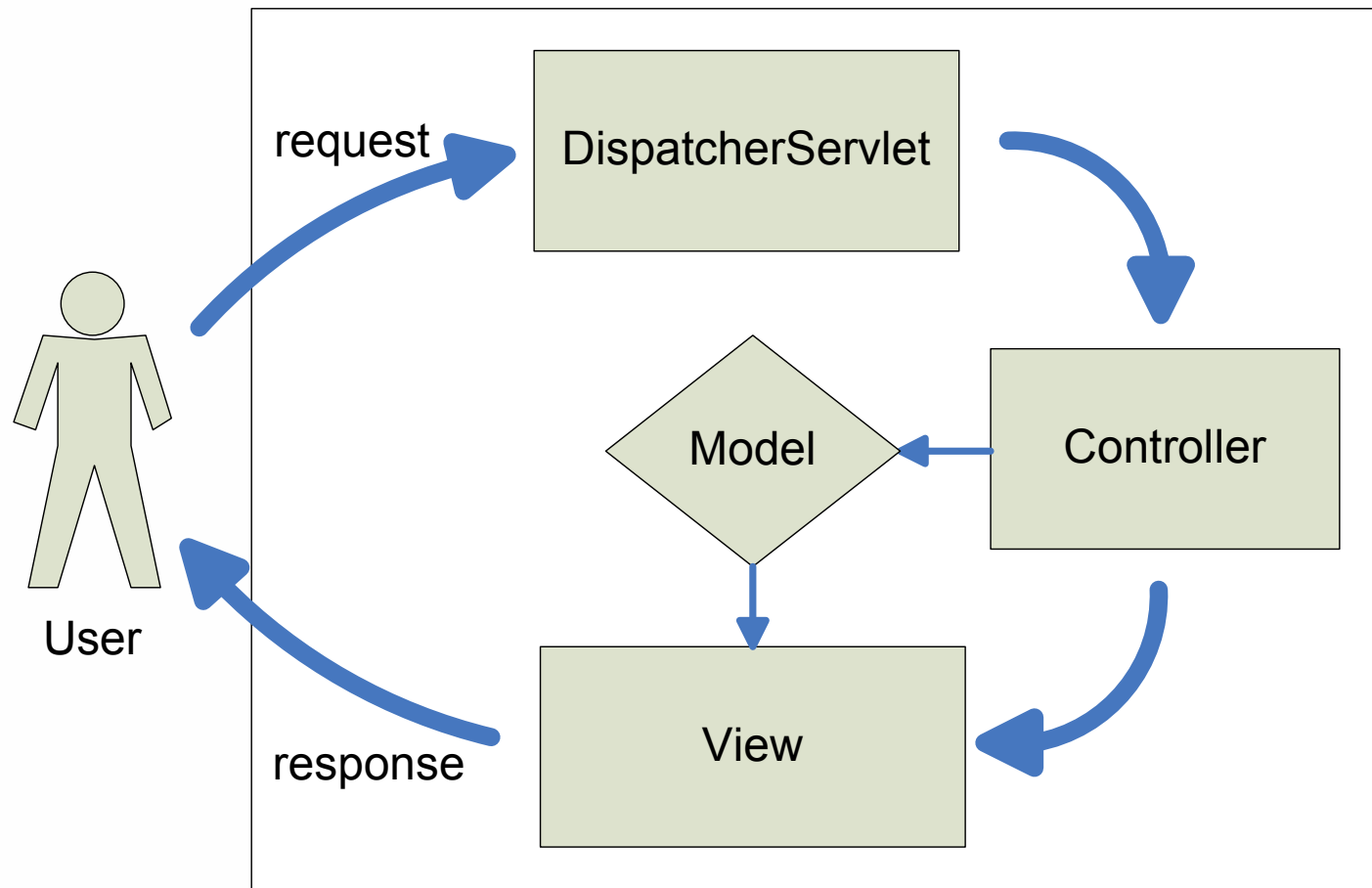
Request-driven MVC - DispatcherServlet (or: everything you need to know about Spring MVC – 5/5)



- Controls the entire request/response flow
 - ◆ Controller/View handling
 - ◆ Localization / Multipart (upload) handling
 - ◆ ...
- Implemented as a Front Controller (read more in PoEAA by Martin Fowler)
- Typically wire one Servlet per application
- Loads a Spring ApplicationContext
- Just a servlet, handling all HTTP-requests



Request-driven MVC – the flow



Some code - the Controller

```
public interface Controller {  
  
    public ModelAndView handleRequest(  
        HttpServletRequest req,  
        HttpServletResponse res)  
        throws Exception;  
  
}
```

Some code - the ModelAndView

```
public class ModelAndView {  
  
    public Map get/setModel();  
    public String get/setViewName();  
  
    ...  
  
}
```

Agenda

- Spring and Spring MVC
- **The Sample Application**
- Data Binding
- Spring Web Flow
- Redirect-After-Post
- File Upload

Sample: Pimp My Shirt

- Compose new shirts
 - ◆ Select long or short sleeve
 - ◆ Enter graphical or text print
 - ◆ Select color
- Rate previously created shirts
- Delete previously created shirts

Spring
from the source

DEMO

INTERFACE21





composeShirt flow
&
RateShirtsController

<<ShirtService>>

<<ShirtDao>> & <<RatingDao>>

Pimp my Shirt – layout

- **ShirtService** interface and implementation
 - ◆ Implementation of the use cases
- **RateShirtsController** Spring MVC Controller
 - ◆ Spring MVC controller enabling user to rate shirts
- **composeShirt flow** Spring Web Flow
 - ◆ Backed by Web Flow action (covered later on)

Setting up the infrastructure



- Configure a Spring DispatcherServlet
- Define web-specific ApplicationContext
 - ◆ Loaded automatically by DispatcherServlet
- Middle-tier ApplicationContext (loaded by ContextLoadListener) is automatically linked in



Setting up the DispatcherServlet

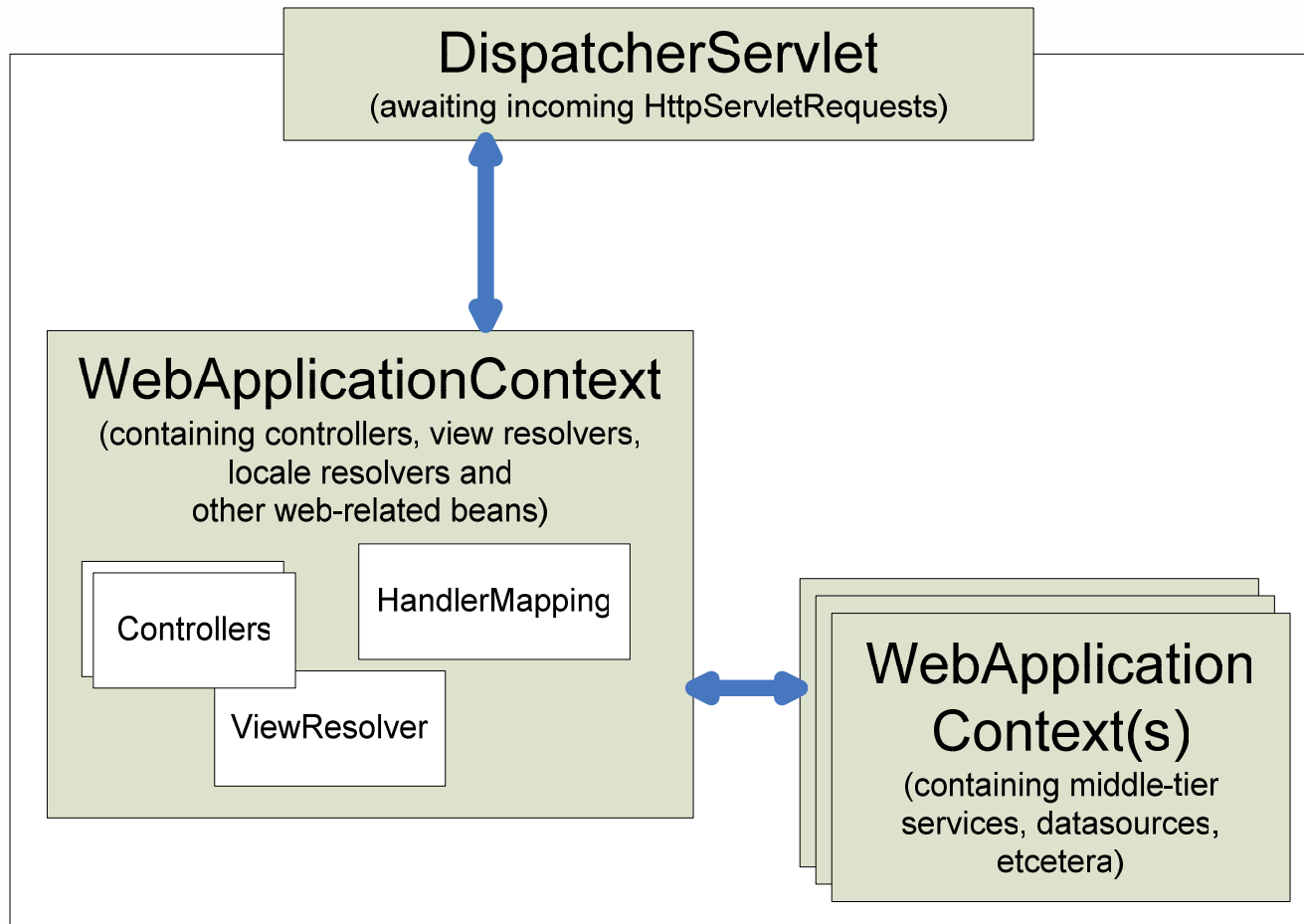
```
<servlet>  
  <servlet-name>pimpmyshirt</servlet-name>  
  <servlet-class>
```

```
org.springframework.web.servlet.DispatcherServlet  
  </servlet-class>  
  <load-on-startup>2</load-on-startup>  
</servlet>
```

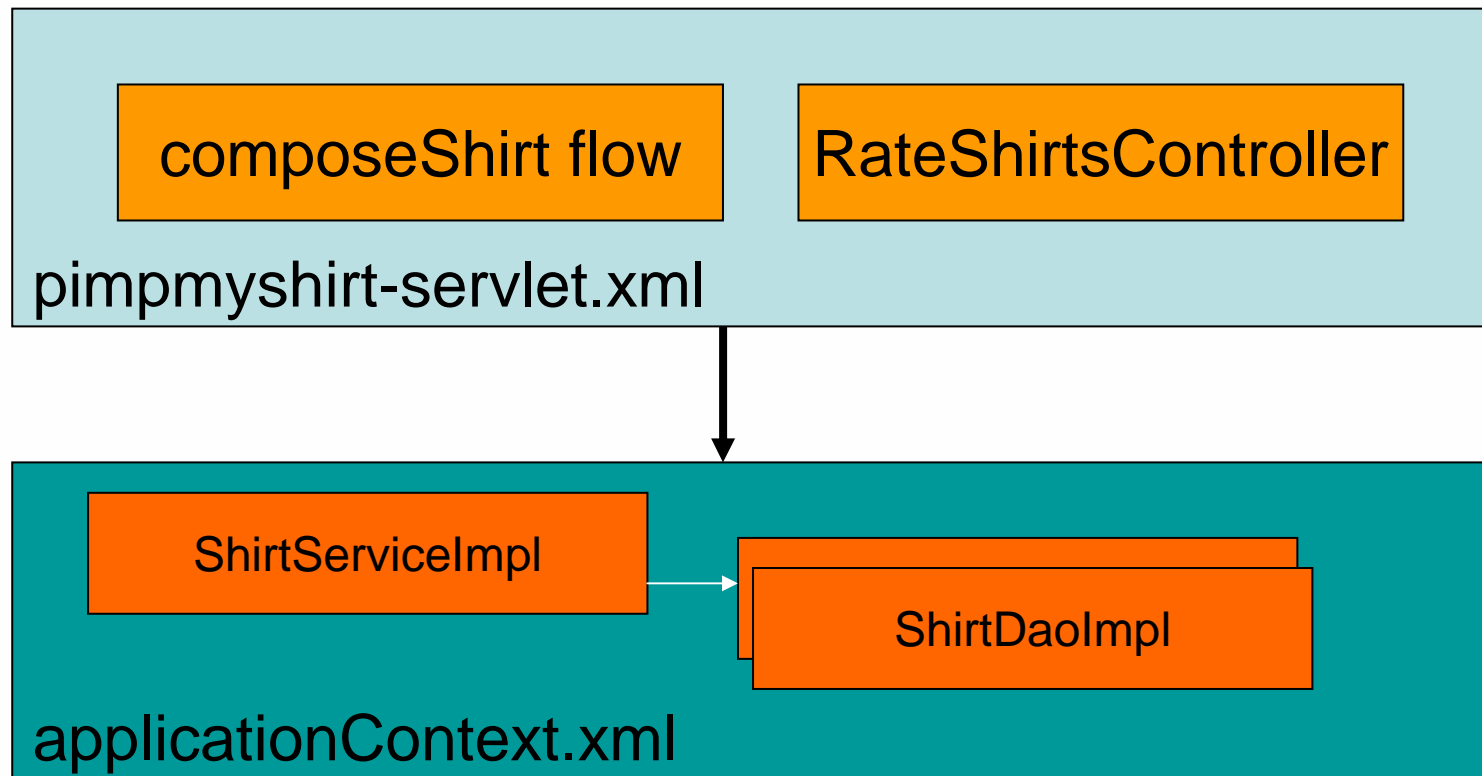
```
<servlet-mapping>  
  <servlet-name>pimpmyshirt</servlet-name>  
  <url-pattern>*.html</url-pattern>  
</servlet-mapping>
```

Defines name of
ApplicationContext

Layout of a typical Spring application



Pimp my Shirt – layout



Implementing a Controller

- Implement Controller interface...
- or implement AbstractController...
- or implement another controller
 - ◆ MultiActionController
 - ◆ SimpleFormController
 - ◆ ...



RateShirtsController

```
public RateShirtsController extends Abstractcontroller {  
  
    private ShirtService shirtService;  
  
    public void setShirtService(ShirtService service) {...}  
  
    public ModelAndView handleRequestInternal(  
        HttpServletRequest req, HttpServletResponse res) {  
        ModelAndView mav = new ModelAndView("frontpage");  
        // add data to MaV using shirtService  
        return mav;  
    }  
}
```

Wire the Controller

```
<bean name="/index.html"  
      class="...RateShirtsController">  
  <property name="shirtService"  
    ref="shirtService" />  
</bean>
```

- In reality uses MultiActionController
 - ◆ Multiple related actions grouped together in one class
 - ◆ Selection handler method done by MethodNameResolver (e.g. using an HTTP parameter value)
- Each action method must return ModelAndView

Spring
from the source

DEMO

INTERFACE21



Agenda

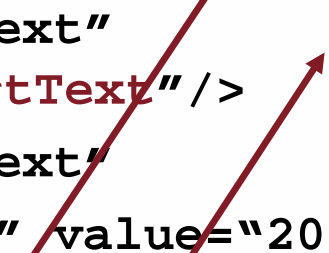
- Spring and Spring MVC
- The Sample Application
- **Data Binding**
- Spring Web Flow
- Redirect-After-Post
- File Upload

- Binding request parameters to objects
 - ◆ Domain objects
 - ◆ Data transfer objects
 - ◆ ...
- Primitives as well as complex objects
 - ◆ Uses `java.beans.PropertyEditors`
- Central class: `WebDataBinder`

Data binding - example

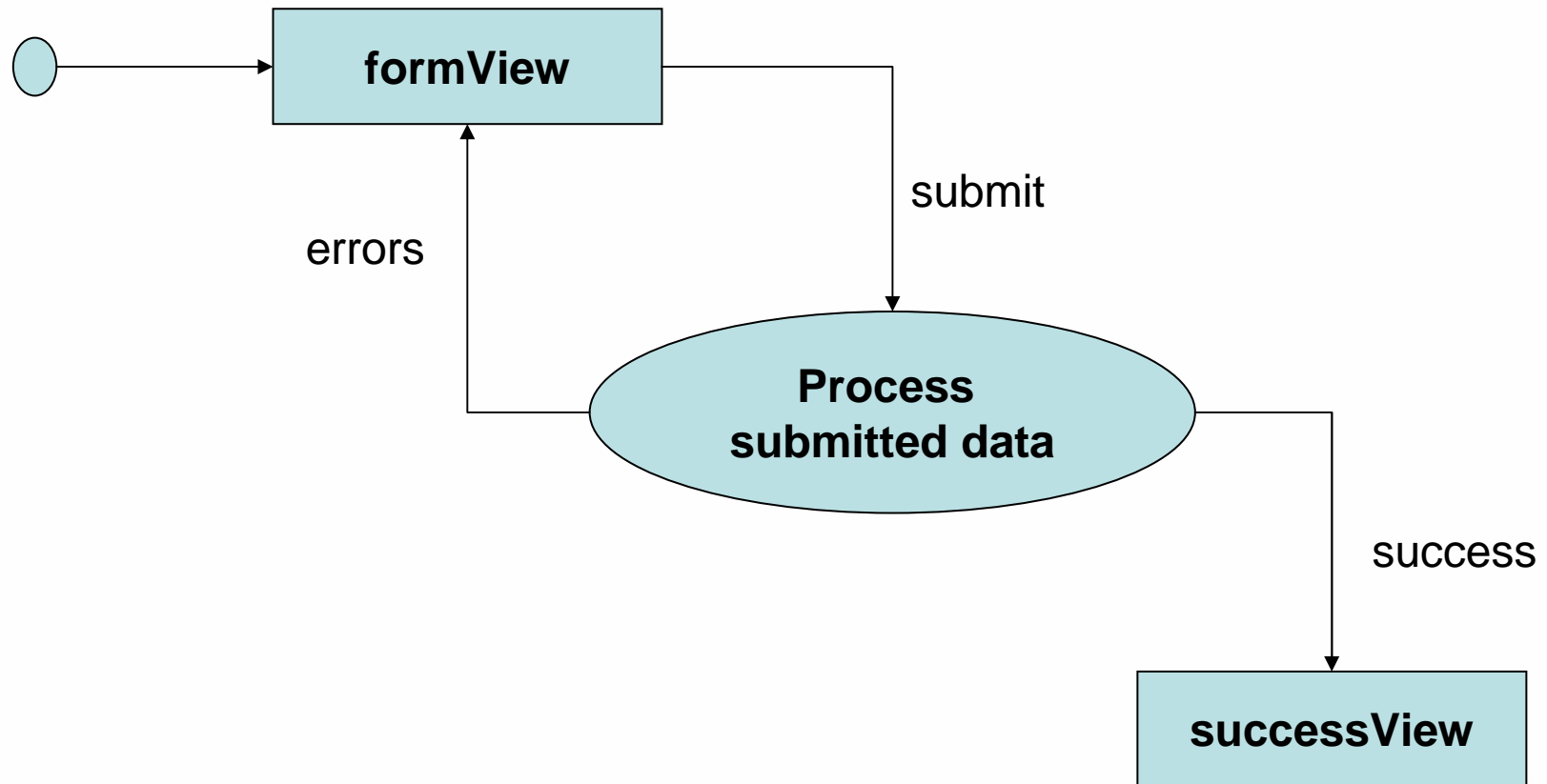
```
public class ShirtFbo {  
    private String shirtText;  
    private Date date;  
    public void set/getShirtText() { ... }  
    public void set/getDate() { ... }  
}
```

```
<input type="text"  
    name="shirtText"/>  
<input type="text"  
    name="date" value="2005-12-13 14:30"/>
```



- Controller to model simple form workflow
 - ◆ GET request → display the form
 - ◆ (User fills in HTML form and submits)
 - ◆ POST request → trigger submit logic
 - Bind HTTP parameters to form backing object
 - Perform validation of form backing object
 - Something wrong? → redisplay HTML form
 - All okay? → perform some action (eg save object)

Simple Form Controller Flow



Binding and validation process

1. **formBackingObject()**
 - Create object to which HTTP parameters will be bound
2. **bindAndValidate()**
 - **initBinder()**
 - Prepare `WebDataBinder` to convert complex types using `PropertyEditors`
 - Bind using `WebDataBinder`
 - Validation using `validators`
3. **processSubmitAction()** either one of
 - `doSubmitAction()`
 - `onSubmit()`
 - ...

- Convert Strings to complex objects
- Convert complex objects to Strings
- Standard JavaBeans infrastructure
 - ◆ Shirt ID as text `<input name="shirtId"/>` converted to Shirt objects
- Out of the box implementations in Java as well as Spring (Date, Class, Number, et cetera)

PropertyEditors - example

```
// don't use this at home!!
public class StupidClassEditor
    extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(Class.forName(text));
    }

    public void getAsText(String text) {
        return getValue().toString();
    }
}

public void initBinder(HttpServletRequest request,
    ServletRequestDataBinder dataBinder) {
    dataBinder.registerCustomEditor(Class.class, new StupidClassEditor());
}
```

<spring:bind/>

- Spring tag library to use property editors in web pages
- Simplifies setting up HTML forms
- Enables looking at status of given property
 - ◆ String-representation of property value
 - ◆ Error messages for the property (conversion or validation errors)
- Does not produce HTML!

<spring:bind/> - example

```
<spring:bind path="shirtFbo.shirtText">
```

```
  <input type="text"  
    name="{status.expression}"  
    value="{status.value}"/>
```

```
  <c:if test="{status.error}">  
    <span class="error">  
      {status.errorMessage}  
    </span>  
  </c:if>
```

```
</spring:bind>
```

Corresponds
to shirtText property
in ShirtFbo object

Agenda

- Spring and Spring MVC
- The Sample Application
- Data Binding
- **Spring Web Flow**
- Redirect-After-Post
- File Upload

- In a nutshell
 - ◆ An advanced controller for Web MVC targeting use cases demanding *controlled navigation*
 - ◆ Integrates with several frameworks
 - Spring Web MVC & Spring Portlet MVC
 - Struts
 - JSF
 - ...

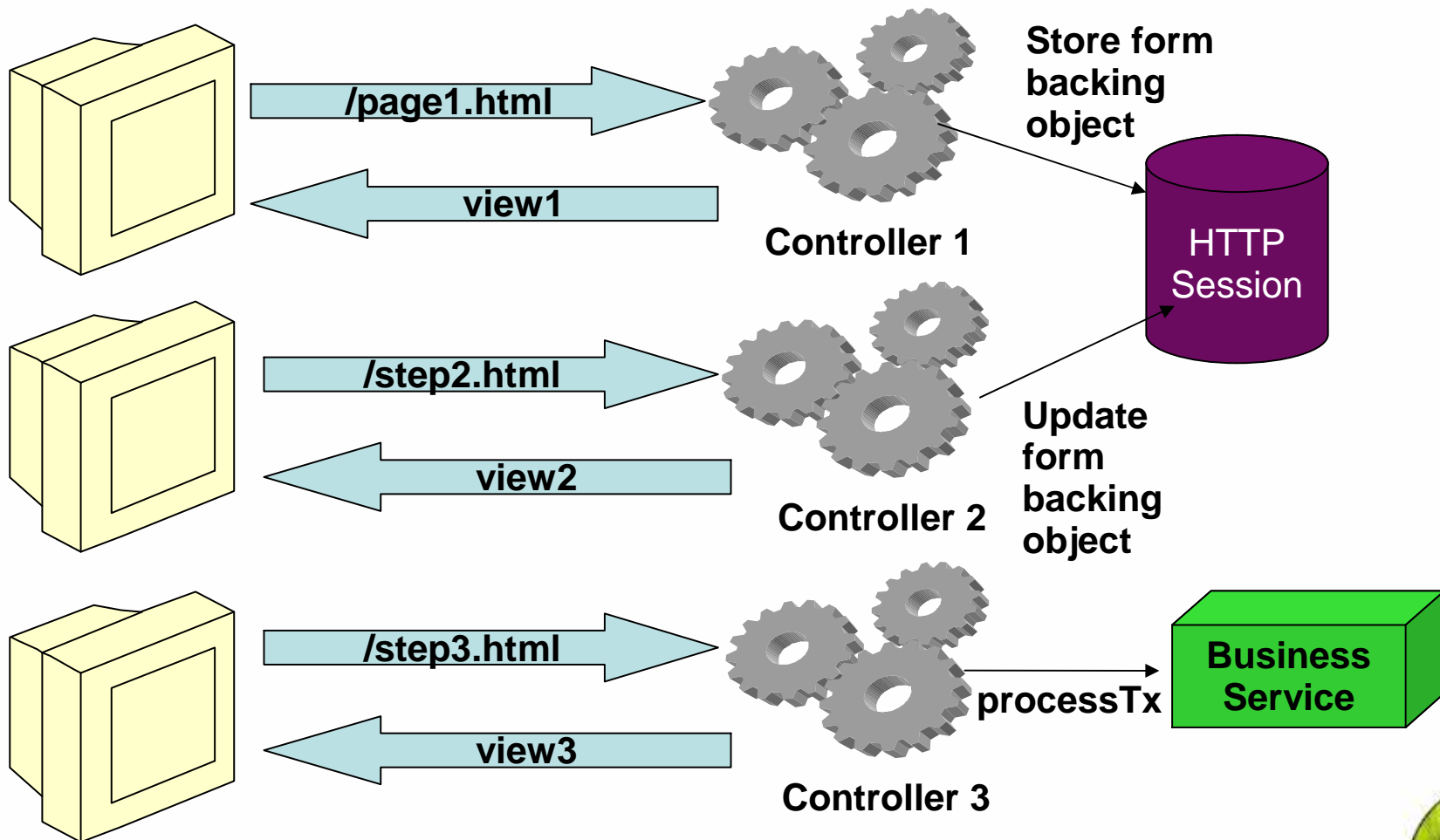
Typical Example

- A wizard to compose a shirt
 - ◆ Spans several steps
 - ◆ Some steps require user input
 - ◆ Some steps are decision points
 - ◆ Some steps perform *logic* (calculations, ...)
 - ◆ Navigation is controlled

Classic Implementation

- What would this look like with Spring MVC?
 - ◆ Use `AbstractWizardFormController` ...
 - ◆ Manage session scoped form backing object to hold wizard data
 - ◆ Controller (and View) for each step
 - ◆ Expose each Controller at a request URL
 - ◆ Views submit to particular to particular URL:
a controller handling one step in the flow

Result



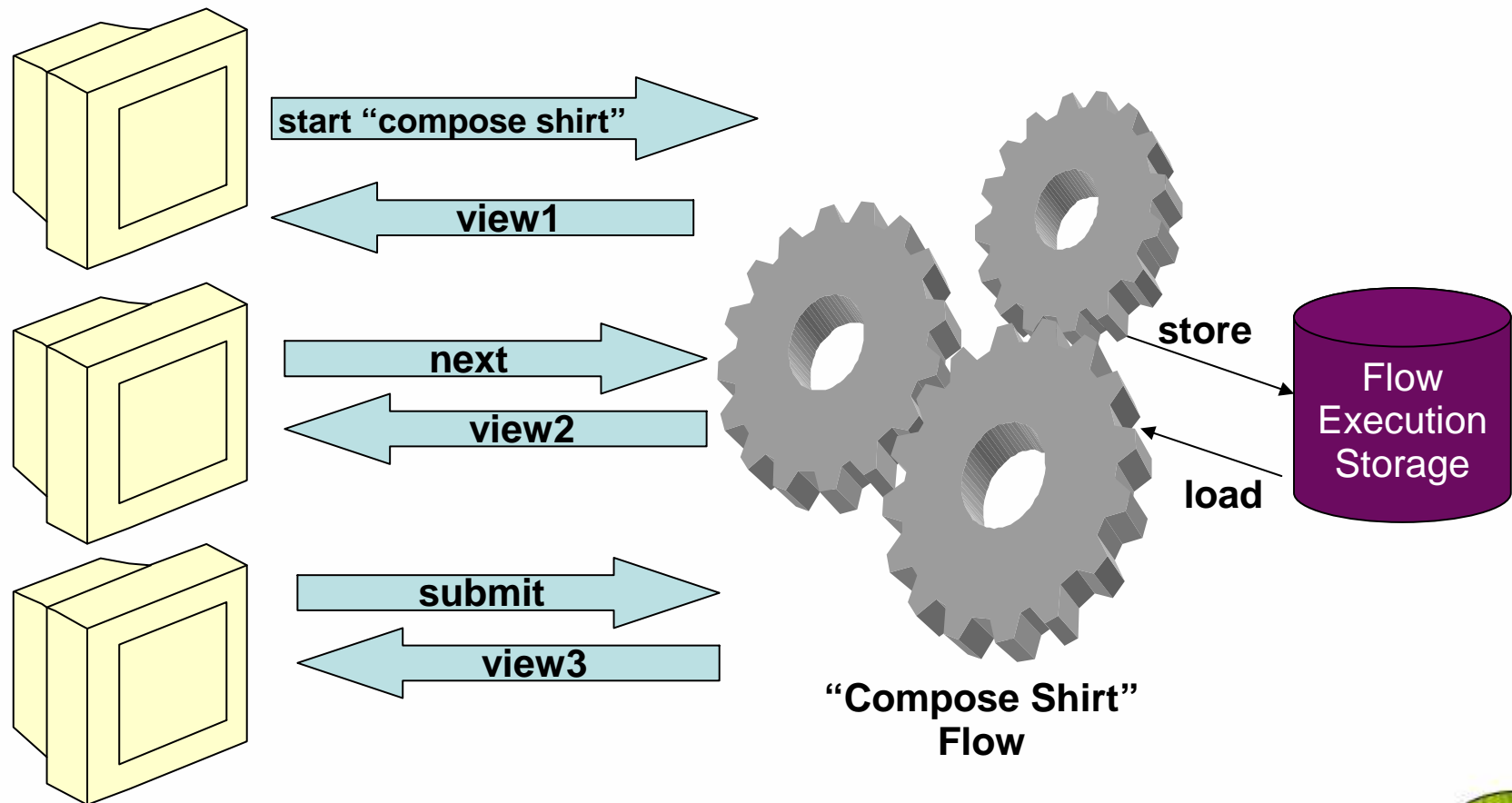
Problems with this approach

- Request centric
 - ◆ No concept of a conversation or *flow*
 - ◆ The flow is chopped up in individual controllers
- Brittle dependency on request URLs
- Manual state management
- *Loosely* controller navigation
- ...

Consequences

- Custom code to handle problems
- Hard to maintain
- Hard to reuse
- A key abstraction is missing: the flow!
 - ◆ A web conversation: longer than a single request but shorter than a user's session

Refactored with Spring Web Flow



Spring Web Flow – Concepts (1)

- One flow controls entire conversation
- When user input is required, the flow pauses and a view is rendered
- Clients *signal* events to tell the flow something happened
- The flow responds to the events and decides what to do next

- A flow is a simple finite state machine
 - ◆ A set of states with transitions between them
- SWF is not a general purpose work flow engine
 - ◆ Lacks things like split/join
 - ◆ Focus on web conversations allow many simplifications

- Flow states execute behavior when entered
 - ◆ **View state** renders view & solicits user input
 - ◆ **Action state** executes a command
 - ◆ **Subflow state** spawns child flows
 - ◆ **Decision state** selects paths through flow
 - ◆ **End state** terminates a flow
- Events drive state transitions

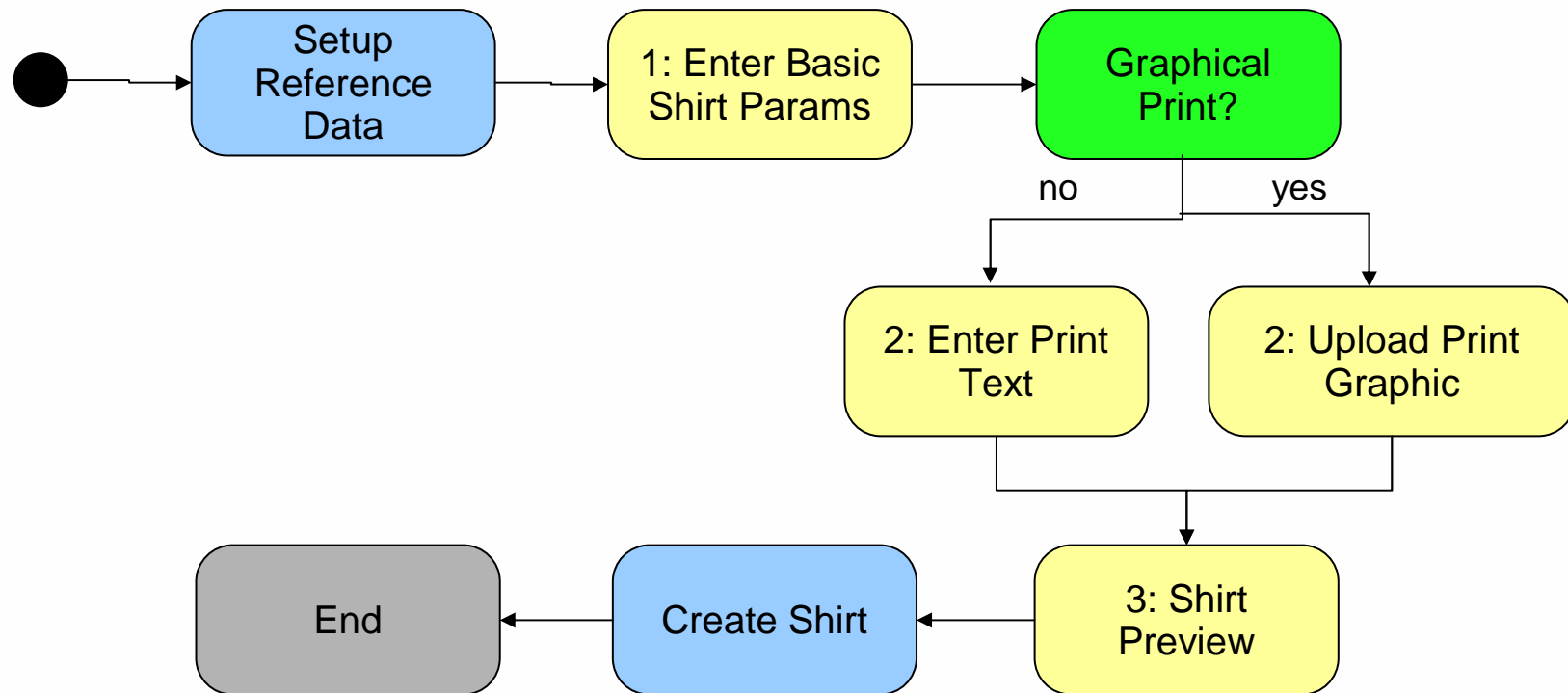
Spring Web Flow – Concepts (4)

- A flow is a reusable application module
 - ◆ Flow *logic* fully encapsulated
 - ◆ Flows can easily use other flows as subflows
 - ◆ A flow defines an input-output contract

Defining a flow

- Start with a simple state transition diagram
- Capture that in a flow definition
 - ◆ Typically XML
 - ◆ Other definition formats possible (e.g. Java)

Compose shirt flow



● Action State

● View State

● Decision State

● End State



- An Action State

```
<action-state id="referenceData">  
  <action bean="composeShirtAction"/>  
  <transition on="success" to="step1"/>  
</action-state>
```

- A Decision State

```
<decision-state id="printSwitch">  
  <if test="\${flowScope.shirtFbo.graphical}"  
    then="step2Graphical"  
    else="step2Text"/>  
</decision-state>
```

- A View (Form) State

```
<view-state id="step1" view="composeShirtStep1">
  <entry>
    <action bean="composeShirtAction" method="setupForm"/>
  </entry>
  <transition on="ok" to="printSwitch">
    <action bean="composeShirtAction" method="bindAndValidate">
      <property name="validatorMethod" value="validateStep1"/>
    </action>
  </transition>
  <transition on="cancel" to="end"/>
</view-state>
```

- An End State

```
<end-state id="end" view="redirect:/index.html"/>
```

Where is `composeShirtAction`?

- Defined as a bean in the application context
- Implementation of `Action` interface

```
<bean id="composeShirtAction"  
  class="org.pimpmyshirt.web.ComposeShirtAction">  
  <property name="shirtService" ref="shirtService"/>  
</bean>
```

- Support for *multi-actions* is provided

```
public interface Action {  
    public Event execute(RequestContext context)  
        throws Exception;  
}
```

```
public Event methodName(RequestContext context)  
    throws Exception;
```

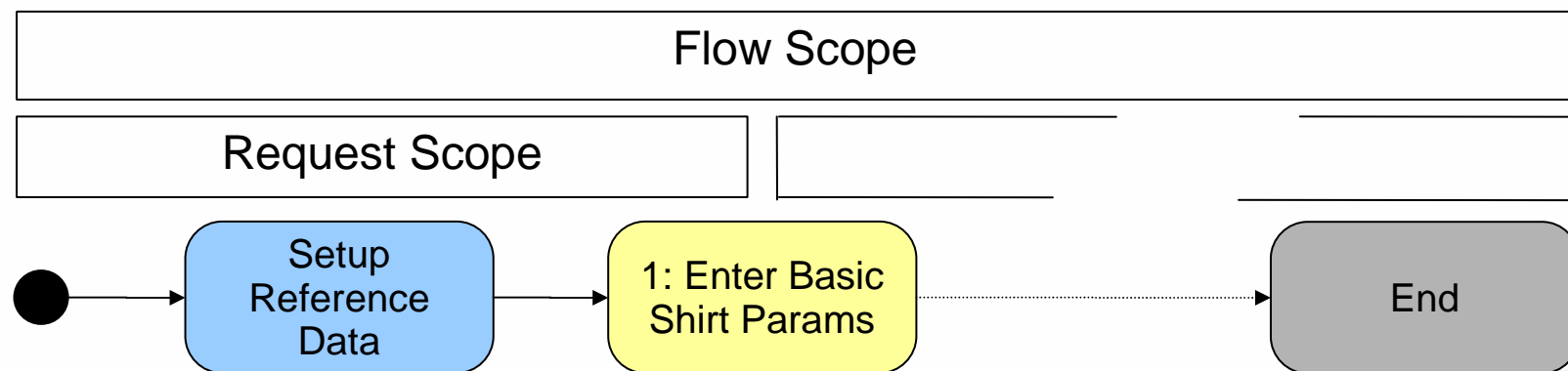


The FlowExecution?

- The data associated with the execution of a flow for a particular client
 - ◆ Flow ~ Class
 - ◆ FlowExecution ~ Object
- Has a unique identifier
 - ◆ Client will have to maintain this id
- Client code (Action) uses RequestContext to interact with the flow execution

Flow Scope & Request Scope?

- **Flow Scope** (`RequestContext.getFlowScope()`)
 - ◆ Storage that spans the entire flow execution
- **Request Scope** (`RequestContext.getRequestScope()`)
 - ◆ Storage that spans a single request



- Add a FlowController to your application

```
<bean id="flowController"  
  name="/flow.html"  
  class="org.springframework.webflow.mvc.FlowController"  
>
```

- Single Flowcontroller for entire application
- Define the flow as a bean

```
<bean id="composeShirt"  
  class="org.springframework.webflow.config.XmlFlowFactoryBean">  
  <property name="location"  
    value="/WEB-INF/composeShirtFlow.xml"/>  
</bean>
```

■ Launching a new flow

- ◆ `<a href="<c:url
value="flow.html?_flowId=composeShirt"/>">Add Shirt`
- ◆ Send `_flowId` but not `_flowExecutionId`

■ Participating in a flow

- ◆ `<a href="<c:url value="flow.html?_eventId=cancel
&_flowExecutionId=${flowExecutionId}"/>">Cancel`
- ◆ Send `_eventId` and `_flowExecutionId` but not `_flowId`

- Also possible with HTML forms

```
<form action="<c:url value="flow.html"/>" method="post">
  <input type="hidden" name="_eventId" value="ok"/>
  <input type="hidden" name="_flowExecutionId"
    value="{flowExecutionId}"/>
  <input type="hidden" name="_currentStateId"
    value="step1"/>
  ...
  <input type="submit" value="Next"/>
</form>
```

- Notice optional *_currentStateId*
 - ◆ Also available in model:
\${currentStateId}

Spring Web Flow - Advantages



- A clear and encapsulated flow definition capturing all navigation rules
- State management is automatic (**FlowExecutionStorage**)
- Flows are reusable application modules
- HTTP independent
- Observable flow life cycle (**FlowExecutionListener**)



- Don't overuse!
 - ◆ Targeted at controlled navigations!
 - ◆ Side by side with Spring MVC controllers
- Be careful with *browser back button*
 - ◆ Consider *continuations*
- Guard against double submit (with back button)
 - ◆ Consider *TransactionSynchronizer*

Advanced data binding

- `WebDataBinder` not usable in all situations
- Manual data binding still possible
 - ◆ Using `HttpServletRequest`
 - ◆ Using convenient `RequestUtils` and `WebUtils`
- If things became nasty,
just use manual data binding

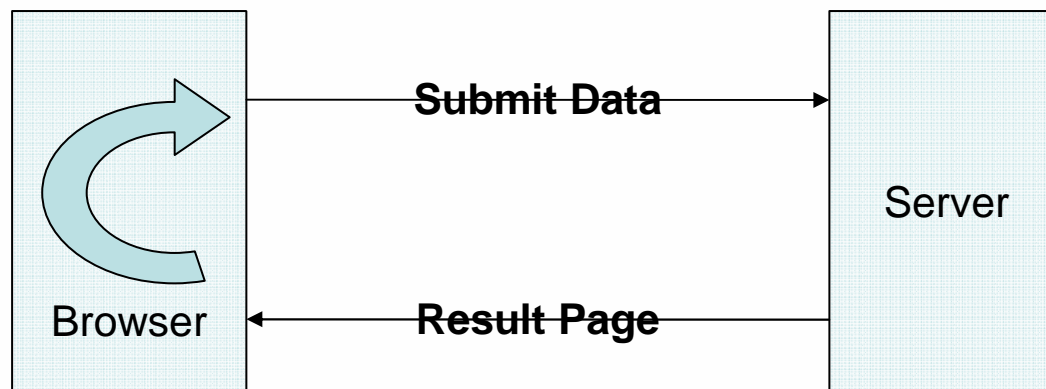
Agenda

- Spring and Spring MVC
- The Sample Application
- Data Binding
- Spring Web Flow
- **Redirect-After-Post**
- File Upload

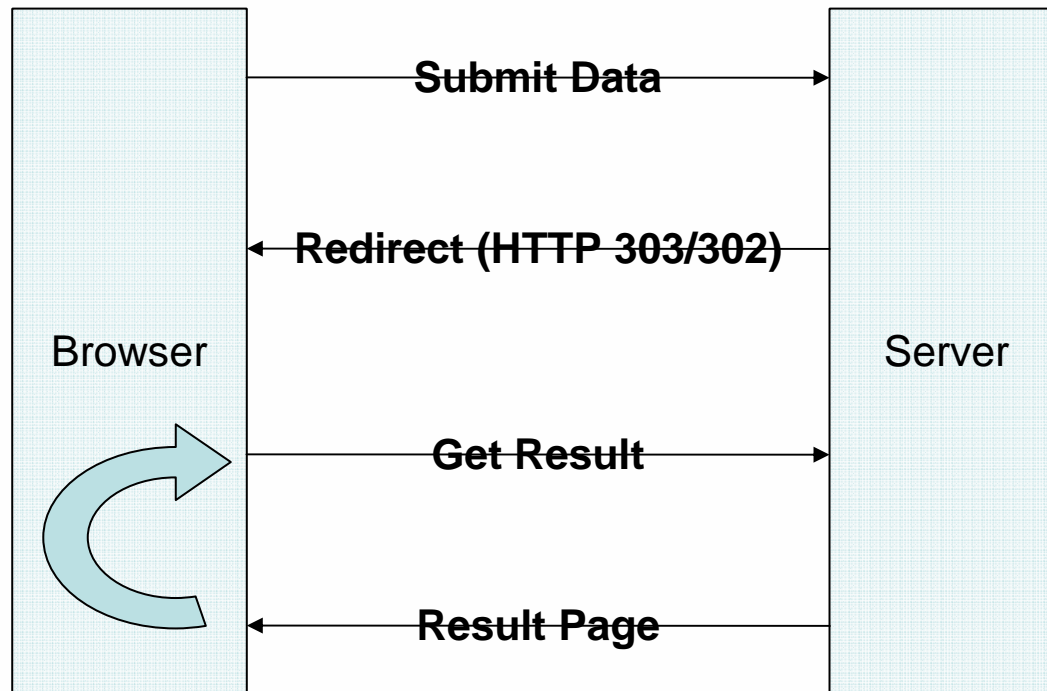
- Most important HTTP request methods
 - ◆ POST: submit user input to server
 - ◆ GET: retrieve data from server (idempotent)
- Really *Redirect-After-Submit*
 - ◆ Sometimes GET has side effects; e.g. because it's used to submit data
 - ◆ Best practice: only use HTTP POST for submits

Double submit problem

- What if the server returns data (a page) in response to a submit (POST) request?



Solution: Redirect-After-Submit



Be careful with caching

- Don't cache submit result pages

```
<bean id="flowController" name="/flow.html"  
  class="org.springframework.webflow.mvc.FlowController">  
  <property name="cacheSeconds" value="0"/>  
</bean>
```

- Use the RedirectView

```
RedirectView rv = new RedirectView("/index.html");  
rv.setContextRelative(true);  
return new ModelAndView(rv);
```

- Use the *redirect:* prefix

- ◆ Detected by UrlBasedViewResolver

```
return new ModelAndView("redirect:/index.html");  
  
<end-state id="end" view="redirect:/index.html"/>
```

Agenda

- Spring and Spring MVC
- The Sample Application
- Data Binding
- Spring Web Flow
- Redirect-After-Post
- **File Upload**

Handle File Uploads

- Spring Web MVC fully supports file upload
 - ◆ Leverages existing frameworks
 - Commons fileupload
 - O'Reilly COS (*buy-a-book* license)
- SWF can use Spring Web MVC file upload support
 - ◆ Just like with data binding

- DispatcherServlet needs a MultipartResolver
 - ◆ Wraps multi-part requests in a MultipartHttpServletRequest containing MultipartFile objects
- Deploy in web context

```
<bean id="multipartResolver"  
      class="org...multipart.common.CommonsMultipartResolver"/>
```

- Use a *special* HTML form

```
<form action="<c:url value="flow.html"/>"
  method="post" enctype="multipart/form-data">
  ...
  <spring:bind path="shirtFbo.shirt.print">
    <input type="file" name="{status.expression}">
    <c:if test="{status.error}">
      ...
    </c:if>
  </spring:bind>
  ...
</form>
```

- Directly in controller

```
if (request instanceof MultipartHttpServletRequest) {  
    MultipartFile print = (MultipartFile)  
        ((MultipartHttpServletRequest)request)  
        .getFile("shirt.print");  
    ...  
}
```

- Using ByteArrayMultipartFileEditor

```
binder.registerCustomEditor(byte[].class,  
    new ByteArrayMultipartFileEditor());
```

- Using a custom PropertyEditor
 - ◆ Bind multiple properties of the uploaded file to a domain layer object

```
binder.registerCustomEditor(Print.class,
    new PropertyEditorSupport() {
        public void setValue(Object value) {
            if (value instanceof MultipartFile) {
                MultipartFile multipartFile = (MultipartFile) value;
                ImagePrint print = new ImagePrint();
                try {
                    print.setImage(multipartFile.getBytes());
                    print.setContentType(multipartFile.getContentType());
                    print.setFilename(multipartFile.getOriginalFilename());
                }
                catch (IOException ex) { ... }
                super.setValue(print);
            }
            ...
        }
    });
```