Model-View Controller

The MVC Design Pattern for Web Apps

Java Servlet APIs for Implementing MVC

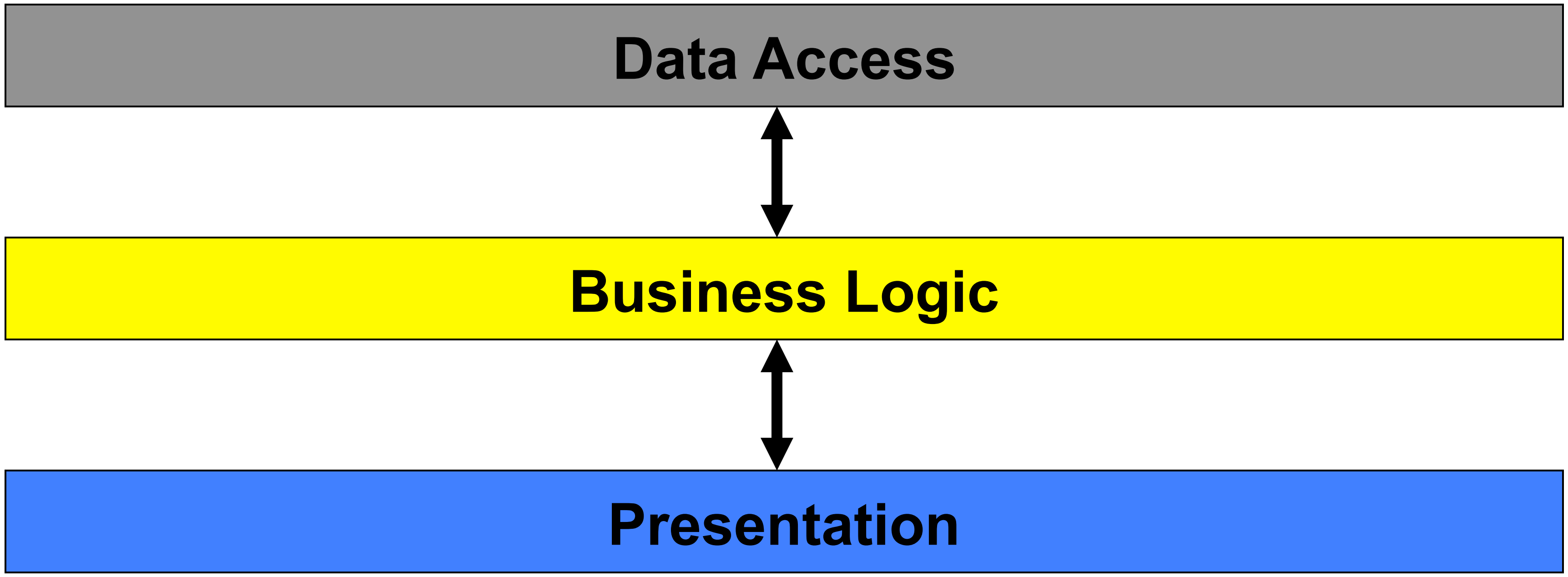
View Layer Solutions

Wrap-up

THE MVC DESIGN PATTERN FOR WEB APPLICATIONS

Separation of Concerns (SoC)

Web applications have 3 important aspects, or *concerns*



It is important to have *Separation of Concerns* (SoC):

- Allows custom expertise and frameworks to be applied to each concern
- Allows for greater run-time deployment flexibility & optimizations

Model-View-Controller (MVC)

THE design pattern of the web

- What is a *design pattern* again?
  - Driven by *forces,* or problems/issues/needs that require resolution
  - A *reusable solution* in context
  - The *context* here is a web application (our 6 steps)
  - The reusable solution is the framework we will use to handle those common 6 steps according to the MVC structure
    - Yes MVC is a Structural pattern
    - Web frameworks in any technology platform, server-side or client-side, organize their code around this structural pattern
      - E.g. Node/Express, SpringMVC, AngularJS – all based on MVC or a variant

History: MVC did not start with the web

- Started in the 70s in the Smalltalk community
- Yes, the same community that gave us Go4 Design Patterns
- See the "Thing-Model-View_Editor" reading

How the Problem Started

We talk about Design Patterns having “forces”. The driving “force” behind MVC was that coders noticed that all web applications followed a common request/response processing pattern:

```
public class XXXServlet extends HttpServlet{
  public void doGet(HttpServletRequestrq, HttpServletResponsers)
    throws ServletException, IOException  // or doPost
    {
        A. <process request headers>
        B. <process request parameters>
        C. <perform processing>
        D. <Assemble the response payload>
        E. <set Content-type and other response headers>
        F. <write out results>
    }
…
}
```

Result: homegrown frameworks became all the rage, organized around the MVC Design Pattern

Best Practices: the MVC pattern

MVC:

- Controller servlet introduced
- JSP or Templates used for View only
- Java beans role stays the same, though they are manipulated by the Controller while being read by the View

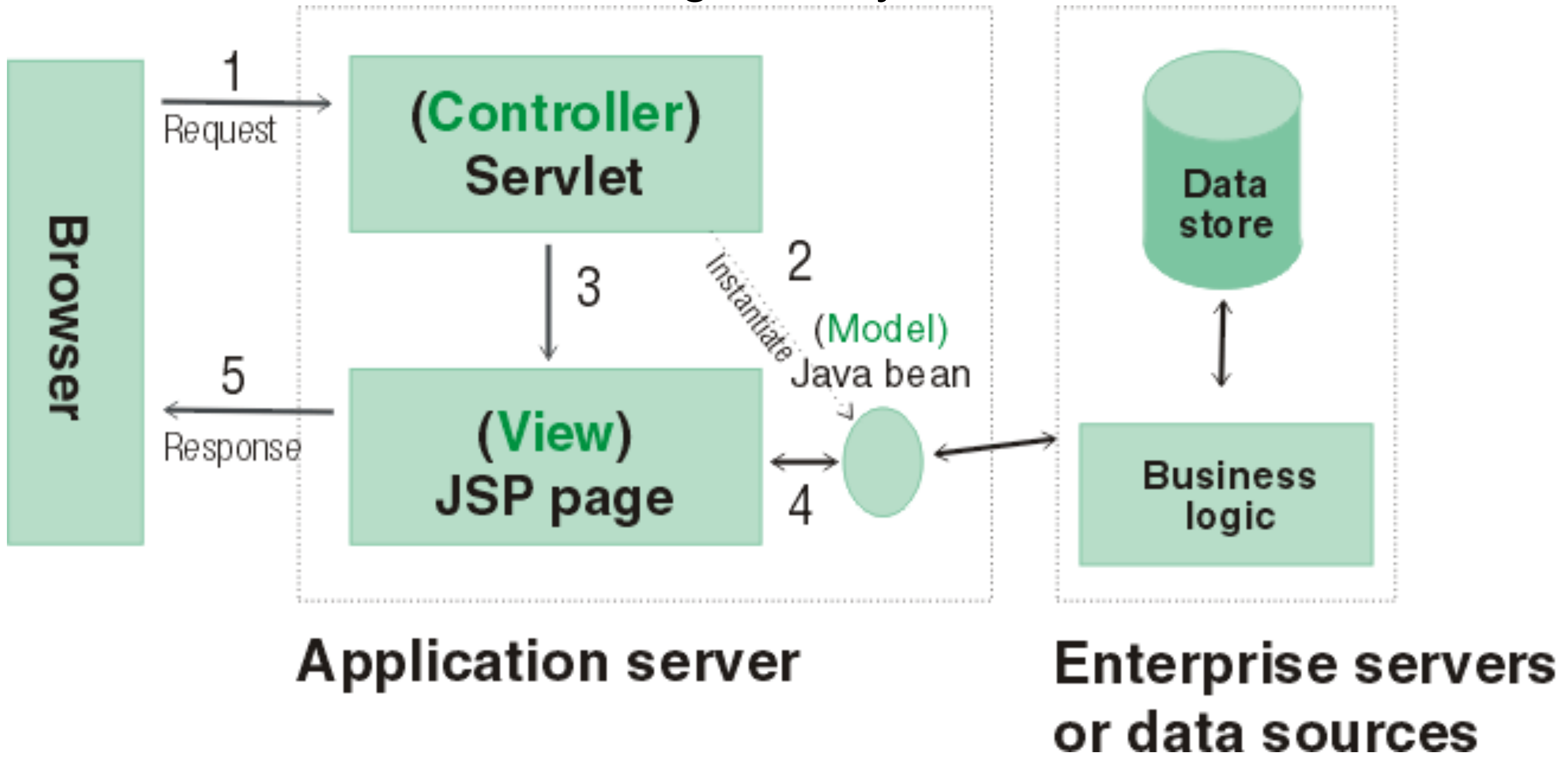


Image from Rational

Implementing MVC (“the Java way”)

Steps:

1. An HTTP request is made to *Controller* servlet
2. Controller (perhaps through *delegation*) instantiates the *Model* (one or more Javabeans)
   a) These are data-centric objects often pulled from a datasource
   b) The objects are a *projection* of the *world model*
3. The Controller servlet forwards the request to the appropriate *View* resource (JSP, template, servlet…)
4. The View (*read-only)* accesses the same Javabeans instantiated by the Controller (or its delegates)
5. The View produces the final rendered response to the client browser.

Implementing MVC

Implementing A-F of the Template pattern:

- Steps A and B have a wide variety of implementations in modern web frameworks. Two categories:
  1. **Action (page) oriented** – natural to what we are doing so far, these frameworks focus on a *request-as-action*, and delegate to the proper page to render as a result.
     - Example: Struts, Apache Click, JSPX, Spring MVC, Brutos …
  2. **Component (event) oriented** – these attempt to map the desktop GUI programming model to the Web. At first derailed as clunky and hidden “magic”, these are now natural due to rise of AJAX
     - Examples: JSF, Tapestry, Wicket, Seam, Vaadin, …

- Step C – need to transfer control of the request from the Controller (and its delegates) to the View for rendering

- Steps D, E & F – the View needs only to read the Model state in order to render a response. Some simple rendering logic is allowed.
  - Problem: too many SOC defects lead to unmaintainable code!

JAVA SERVLET APIS FOR IMPLEMENTING MVC

Step 3 Machinery: Dispatch Requests

Call the getRequestDispatcher method of ServletContext

- Supply URL relative to server or Web application root
- Example:

```
String url = "/presentations/ContentServlet";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(url);
```

What do you do with the Dispatcher?

- Call forward to completely transfer control to destination page (no communication with client in between, as there is with `response.sendRedirect`)
  - This is the normal approach with MVC
- Call include to insert output of the destination page into the output stream and then continue on
  - But this is NOT really MVC, just a convenience!
  - Really it is part of step D "Assemble the response payload"

Dispatch Requests to Resources

With the `forward` method of RequestDispatcher:

- Control is *permanently* transferred to new page
- Original page *cannot* generate any output

With the `include` method of RequestDispatcher:

- Control is *temporarily* transferred to new page
- Original page *can* generate output before/after included page
- Original servlet does not see the output of the included page

So who do you forward/include to?

- You typically *forward* to a *dynamic* resource
  - i.e. a servlet, or JSP, or template
- You typically *include* a *static* or *dynamic content* resource
  - Could be an HTML fragment, like your header or footer on each page
  - Could be a servlet or script that produces content
  - Your original servlet cannot consume the content and manipulate it
    - So, for example, that content-producing resource cannot be a template

Accessing Data in the View

How do you manage data when using *forward* (step C)?

4 ways:

1. Pass (and augment) the HTTP Request
2. Stuff data into the Session
3. Stuff data into the application Context
4. Manage your own Model data through the use of Factories, Resource Pools, Caches, etc.

The 3rd one we won't cover as it is really is not an appropriate vehicle for MVC as the scope is to the application not per each user.

The last one is just a best programming practice and requires no special servlet API help (though web frameworks may help you out here)

The first 2 are the most prevalent; we will discuss next

Storing Data for Later Use: Request Scope

Purpose

- Storing data that a Controller servlet looked up and that the View page will use only in this request.

Servlet syntax to store data

```
SomeClass value = new SomeClass(…);
request.setAttribute(key, value);
// Use RequestDispatcher to forward to View
// View can then retrieve the data using
// request.getAttribute(key)
```

Example: JSP syntax to retrieve data

```
<jsp:useBean id="key" class="SomeClass" scope="request" />
```

Why is this helpful? When do you use it?

- This is most useful for parameterized data you want to be available only to the View, but it really isn't persistent Model data
- This is a widely used scope!

Storing Data for Later Use: Session Scope

Purpose

- Storing data that a Controller servlet looked up and that a View page will use in this request and in later requests *from the same client*.

Servlet syntax to store data

```
SomeClass value = new SomeClass(…);
HttpSession session = request.getSession(true);
session.setAttribute(key, value);
// Use RequestDispatcher to forward to View
// View can then retrieve the data using
// session.getAttribute(key)
```

Example: JSP syntax to retrieve data

```
<jsp:useBean id="key" class="SomeClass" scope="session" />
```

Why is this helpful? When do you use it?

- This is most useful for conversational state just as you would use it before. But now it is immediately available to the View as well.
- This is a the most over- (and incorrectly) used scope!

Variation for Session Tracking

Use `response.sendRedirect` instead of `RequestDispatcher.forward`

Distinctions: with sendRedirect:

- User sees View URL (user sees only Controller URL with `RequestDispatcher.forward`)
- Two round trips to client (only one with forward)

Advantage of sendRedirect

- User can visit View page separately
  - User can bookmark View page

Disadvantage of sendRedirect

- Since user can visit View page without going through Controller first, Model data might not be available
  - So, View page needs code to detect this situation

VIEW LAYER SOLUTIONS

The View: Part One

The Problem: the Controller forwards to a web resource to produce a rendered response.

- What exactly, is the best type of web resource to forward to?

Solution 1: Why not just use a servlet?

- getRequestDispatcher.forward() forwards to a servlet
- The servlet just worries about generating the response
- Separates our concerns – see MVC_Calc and MVC_HG examples

- So what's wrong? It does SoC???
  - UI developers do not have tools to gen UIs as servlets
  - All those out.printlns – blech!
  - Can you keep your concerns separated? Nothing from preventing the View servlet from doing something non-View

The View: Part Two (Templates)

Solution 2: Templates – Mix ‘n Match static & dynamic content

- Web UI devs would create static parts of a page
- Backend devs create dynamic parts
- UI devs owned the structure, or layout of a page
  - “Holes” left for the dynamic devs to fill in
  - Result: “box”-ey layouts that you still see today

- Pros
  - Still SoC
  - Let each dev role stick to what it was good at
  - The UI now started with the markup, not the other way around
- Cons
  - Not a lot of layout flexibility; not “responsive”
  - As Javascript became more popular on both sides, ended up with scope clashes on the page

Template engines

Popular Java ones:

1. Apache Velocity (velocity.apache.org)

   Example: http://edwin.baculsoft.com/2011/06/beginning-apache-velocity-creating-a-simple-web-application/

2. Freemarker (freemarker.apache.org)

   Example: http://viralpatel.net/blogs/freemarker-servlet-tutorial-example/

3. More recent entries include thymeleaf (www.thymeleaf.org) and pebble (http://mitchellbosecke.com/pebble/home)

The basic idea:

- Each template framework has its own servlet you forward to
- Data is marshalled to the View in some way, through their API
- Templates bind to objects, call properties to extract content
- Usually a pseudo-**scripting** language is embedded for display logic

Note there are a number of these for various languages

- Server-side scripting languages (PHP, Python, NodeJS) use them a lot – for example Pug or EJS in NodeJS if you took 421
- http://en.wikipedia.org/wiki/Comparison_of_web_template_engines

The View: Part Three (JSPs)

JSPs can do anything servlets can do, yet make it easier to:

- Read, Write, and Maintain HTML
- Use HTML editing tools to create markup elements and pages.
- Separate the (Java) code that creates the content from the (HTML) code that presents it
  - The UI experts on your team do the HTML instead of the Java coders

<hackalert>

- JSPs an ugly reactive hack
- What is the root of the problem?
  - Very easy to violate SoC because you can write any Java in there
  - In fact, JSPs are cross-compiled to servlets by the servlet container!

</hackalert>

The View: Part 4 (Other Techniques)

XML/XSLT:

- Define/marshal your information in presentation-agnostic XML
- Transform it using XSLT to presentation-centric markup
  - Yes that markup is usually HTML, but could be browser-specific, device-specific (WML anyone?), or even support B2B
- Some consider it a templating technique as well (that is what the "T" stands for) – but it is not the same as the server-side ones

Browser-plugins / embedded viewers

- Yes I do mean Flash and related, including Silverlight, JavaFX, and (yikes) even applets
- Initially these would embed full UI state in the downloaded object
- Evolved (with AJAX) to accept presentation information and dynamically render it in the "player"
- Rapidly falling away in favor of HTML5 (*stay tuned…*)

The View (Part 5): Porting a UI Component Model

Porting the Desktop GUI model server-side:

- A component-oriented *widget* model on the server-side
- *JSF* really revolutionized this at the time
  - IDE plug-ins started becoming desktop-like UI design tools
  - But it was very hard to debug, the rich libraries didn't catch on, and eventually Javascript/AJAX rendered this approach obsolete
- Examples: JSF, Vaadin, GWT

Porting the Desktop GUI model client-side:

- A component-oriented *widget* model on the client-side
- *HTML5* components are JS/CSS-enabled to provide UI-stateful behaviors
- GUI models are really now client-side, handling the Presentation concern through the modern HTML5 stack
- The server-side's "presentation" is really a marshalling of presentation information in JSON (maybe XML)

MVC WRAP-UP

Pros and Cons

Cons: (or: *Is anyone else in here tired?*)

1. It seems to be a lot of work to separate concerns
2. Sometimes you don’t draw the right boundaries
   - And misusing a design pattern is worse than not using it at all
3. Welcome to the rise of endless configuration
   - XML files to wire things together
   - Property files to endlessly customize
   - So many annotations you feel like you learned a new coding language
   - You just lost control of your web.xml
4. Want a new feature? Learn a new toolchain!
5. Want a job? Hope you picked the right framework to learn!
6. Sometimes you just end up throwing the code away anyway

Pros: (*yes, there still are some*)

1. Those toolchains can make you very productive
2. MVC does give you a nice SOC, when properly applied
3. Sometimes you don’t end up just throwing away the code

Wrap-up: Revisiting slides 2-4

Recall the MVC steps in slides 6-7:

1. An HTTP request is made to *Controller* servlet
2. Controller (perhaps through *delegation*) instantiates the *Model* (one or more Javabeans)
3. The Controller servlet forwards the request to the appropriate *View* resource (JSP, template, servlet…)
4. The View (*read-only)* accesses the same Javabeans instantiated by the Controller (or its delegates)
5. The View produces the final rendered response to the client browser.

And the Template pattern steps in slide 5:

A. <process request headers>

B. <process request parameters>

C. <perform processing>

D. <assemble the response payload>

E. <set the response headers>

F. <write out results>

Note how this is not a 1:1 mapping; Rather they interleave. But in general, A&B are done by the Controller, D,E,&F by the View, and C by the Model