



Welcome To Spring-Web

Web Development With Spring

4-day Course



How You Will Benefit

- Learn to use Spring for rich web applications
- Gain hands-on experience
 - 50/50 presentation and labs
- Flash key to take with you
 - Lab environment
 - Presentations
- Access to SpringSource professionals



Course Logistics



- Hours
- Lunch and breaks
- Other questions?

A blurred background image of green grass at the bottom of the slide.

LOGISTICS

Covered in this section

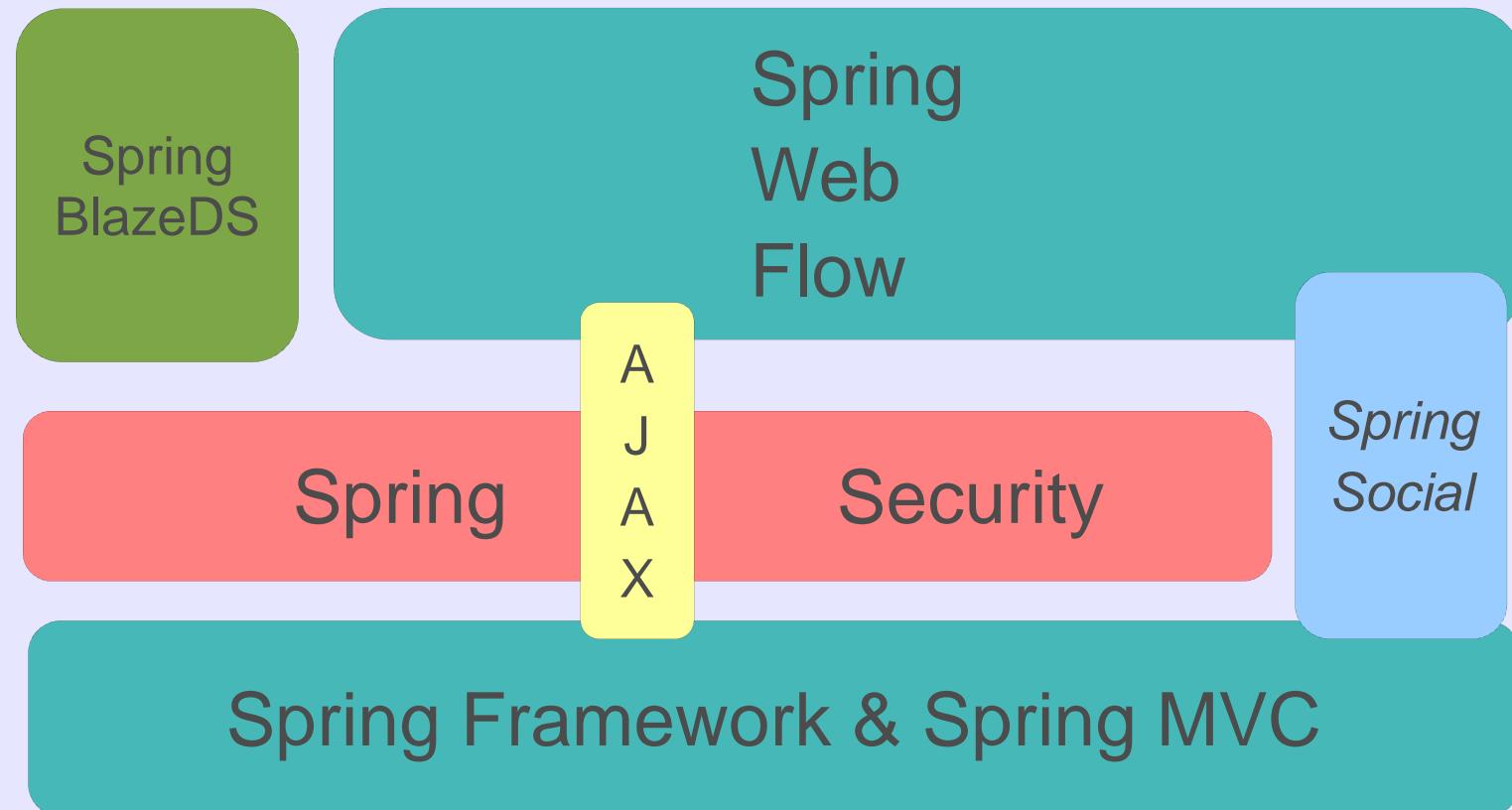


- Agenda
- SpringSource, a division of VMware



What's In Spring Web?

The Spring Web Stack



Day 1 – Spring MVC



- Quick start
 - Lab environment, tools, reference application
- Spring MVC essentials
 - @Controllers, convention over configuration
- Using layouts and views
 - Composite views, multiple rendering technologies



Day 2 – Spring MVC, REST and AJAX



- Processing forms pages
 - Data binding, validation, form tags
- REST
 - Introduction and Spring's RestTemplate
 - Server-side REST
- Enhancing web pages with rich behavior
 - Decorating HTML elements
 - Ajax events, REST and partial page rendering

2





Day 3 – Web Flow

- Getting Started with Spring Web Flow
 - Overview, basic configuration
- Authoring flow definitions
 - View states, events and transitions
- Adding flow behavior
 - Define variables, invoke actions, data binding

3



Day 4 – Security, Testing, Roo



- Web application security
 - Authenticate and authorize users
- Acceptance testing
 - End-to-end functional and performance tests
- Rapid development with Spring Roo

4



Covered in this section



- Agenda
- SpringSource, a division of VMware



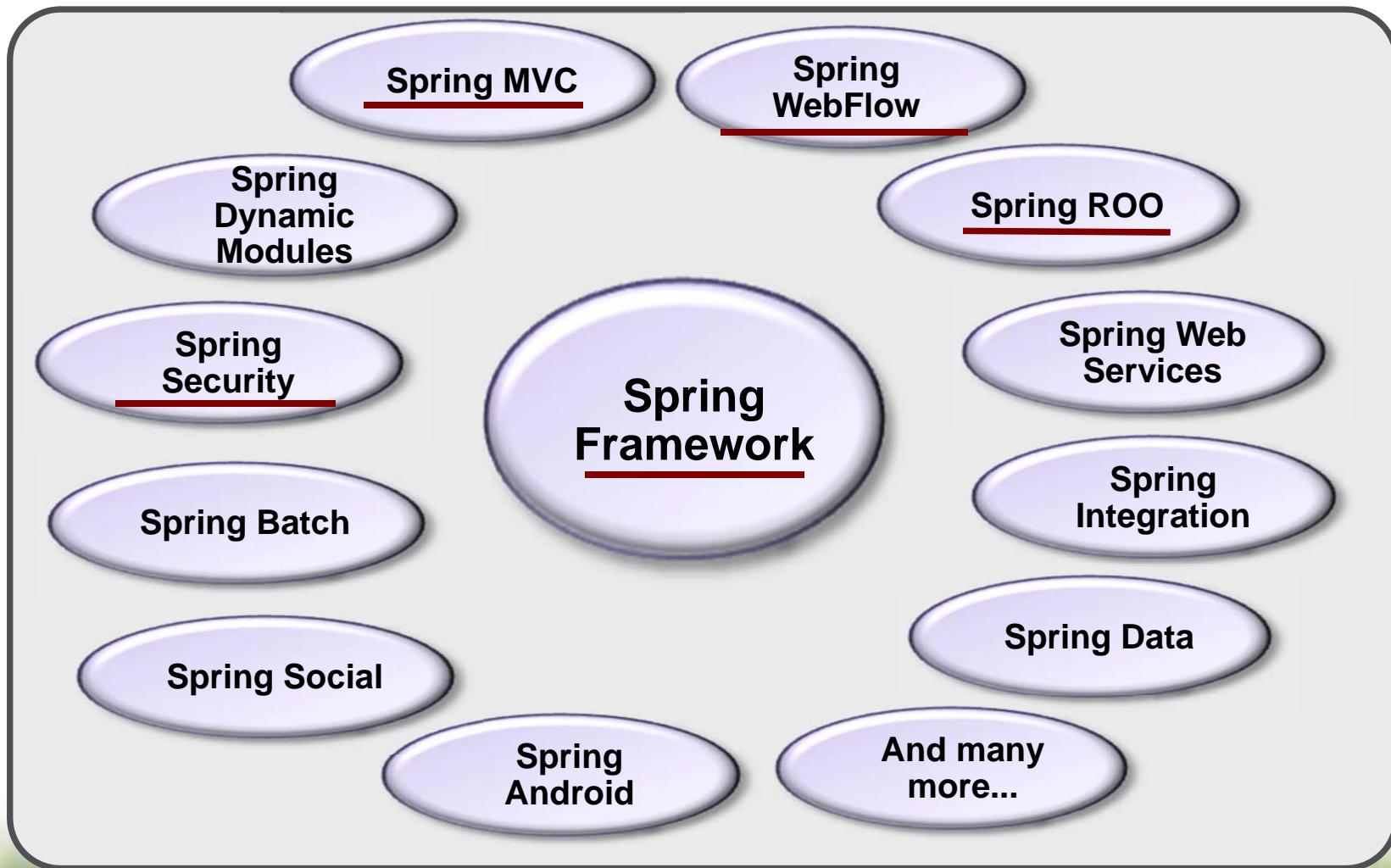
SpringSource, a division of VMware



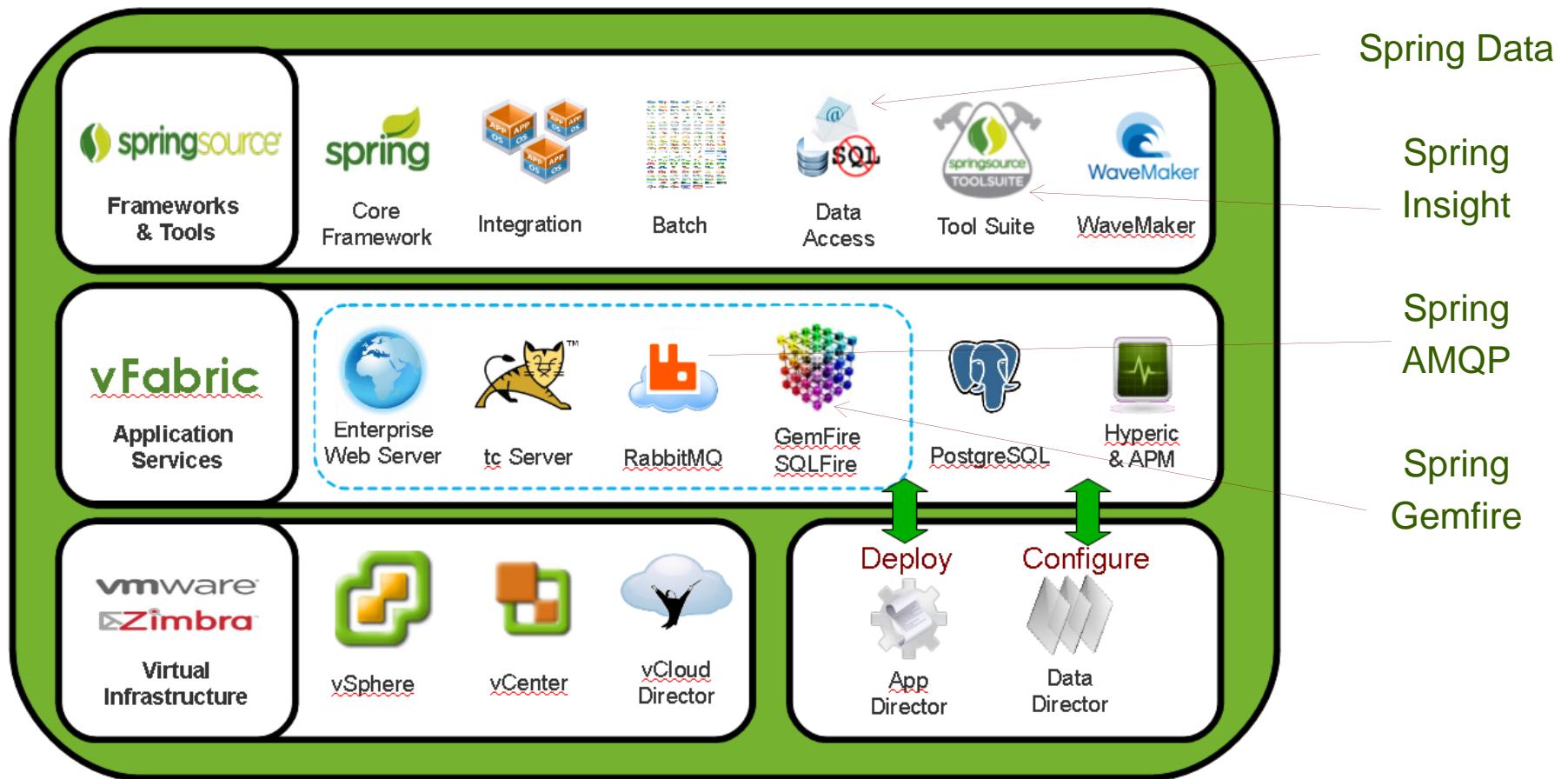
- SpringSource, the company behind Spring
 - acquired by VMware in 2009
- Spring projects key to VMware's *cloud* strategy
 - Virtualize your Java Apps
 - Save license costs
 - Deploy to private, public, hybrid clouds
 - Flexible deployment strategy
 - Handle peak loads cost-effectively
 - Charge per-use, ITSM



The Spring projects



VMWare and Spring



Open Source contributions



- Spring technologies
 - VMware develops 95% of the code of the Spring framework
 - Also leaders on the other Spring projects (Batch, Security, Web Flow...)
- Tomcat
 - 60% of code commits in the past 2 years
 - 80% of bug fixes
- Apache httpd
- Hyperic
- Groovy/Grails
- Rabbit MQ
- ...



Summary

- Agenda
- SpringSource, a division of VMware

Let's get on with the course..!





Spring Web Applications Development Environment



Overview

After completing this lesson, you should be able to:

- Understand the tools you will work with
- Understand the problems you will solve
- Understand URL Design



Road Map

- Tools you will work with
- Problems you will solve
- URL Design

IDE Tools



- SpringSource Tool Suite
 - Free, eclipse-based development environment
- Spring Core support
 - Bean diagram
 - Content assist for Java class names & properties
 - Easy navigation (diagram → XML → Java code)
 - AOP support
- Spring-related related technologies
 - Spring Web Flow, Spring ROO, etc.



Testing Tools

- Unit and Integration testing
 - JUnit and Spring Test Framework
- Performance testing
 - Apache JMeter
- Acceptance testing
 - Selenium



SpringSource tc Server



- Based on Tomcat
- Adds enterprise features
 - Management
 - Monitoring
- Developer edition contains Spring Insight
 - Lightweight introspection
 - Detailed trace per request



Debugging Tools

- Firefox add-ons
 - Firebug
 - Web Developer
- HttpMonitor built into Eclipse
- Java Debugger in Eclipse





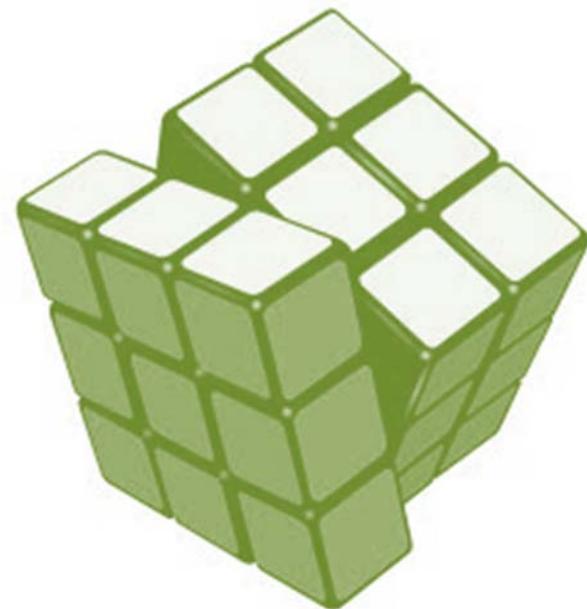
Road Map

- Tools you will work with
- Problems you will solve
- URL Design



Problems to solve

- Course taught in context of a real problem domain
 - Financial reward management
- Engaging requirements
 - Business and technical
- Teaches implementation best practice
 - Spring feature set utilized comprehensively

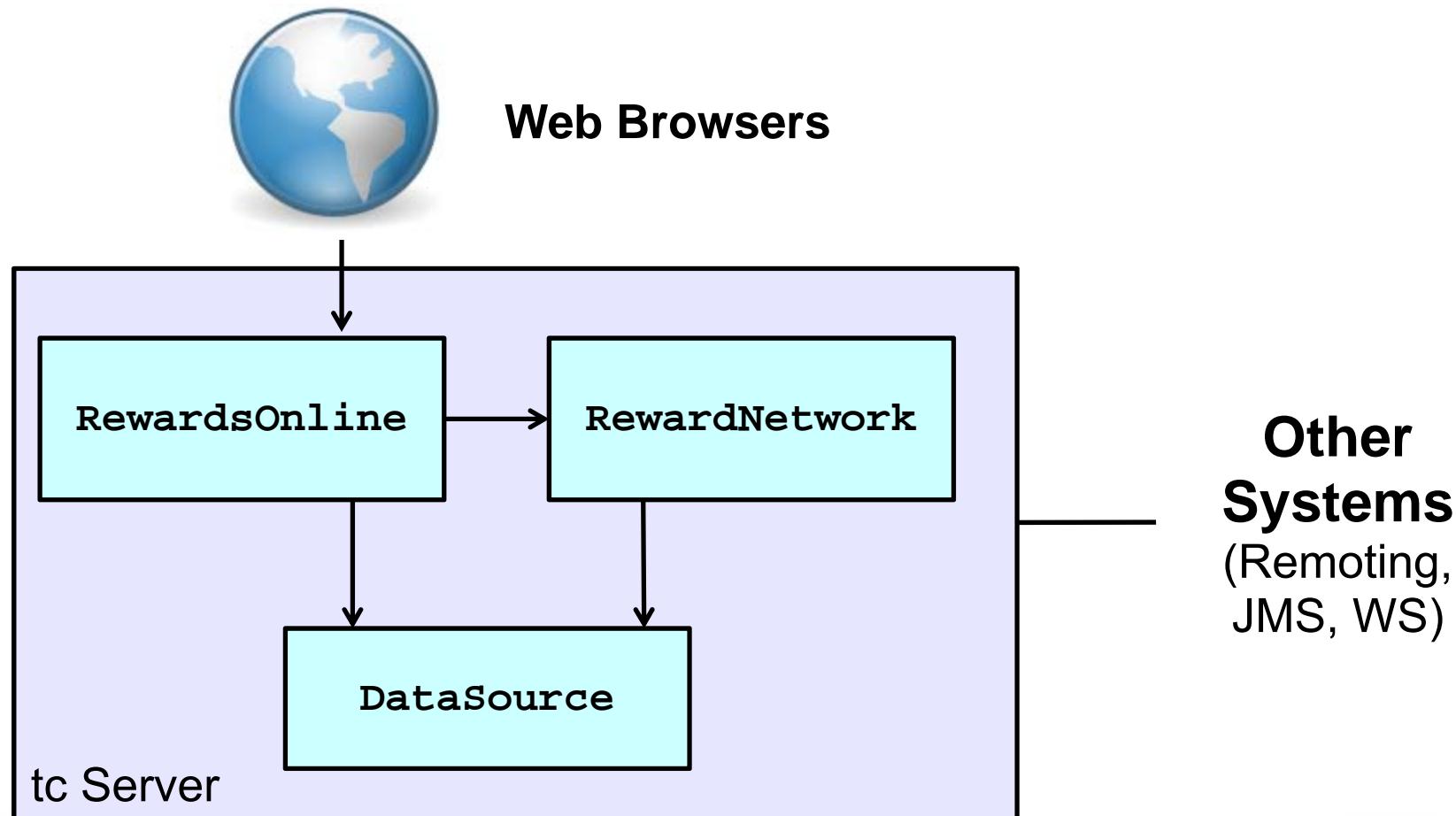


Business requirements

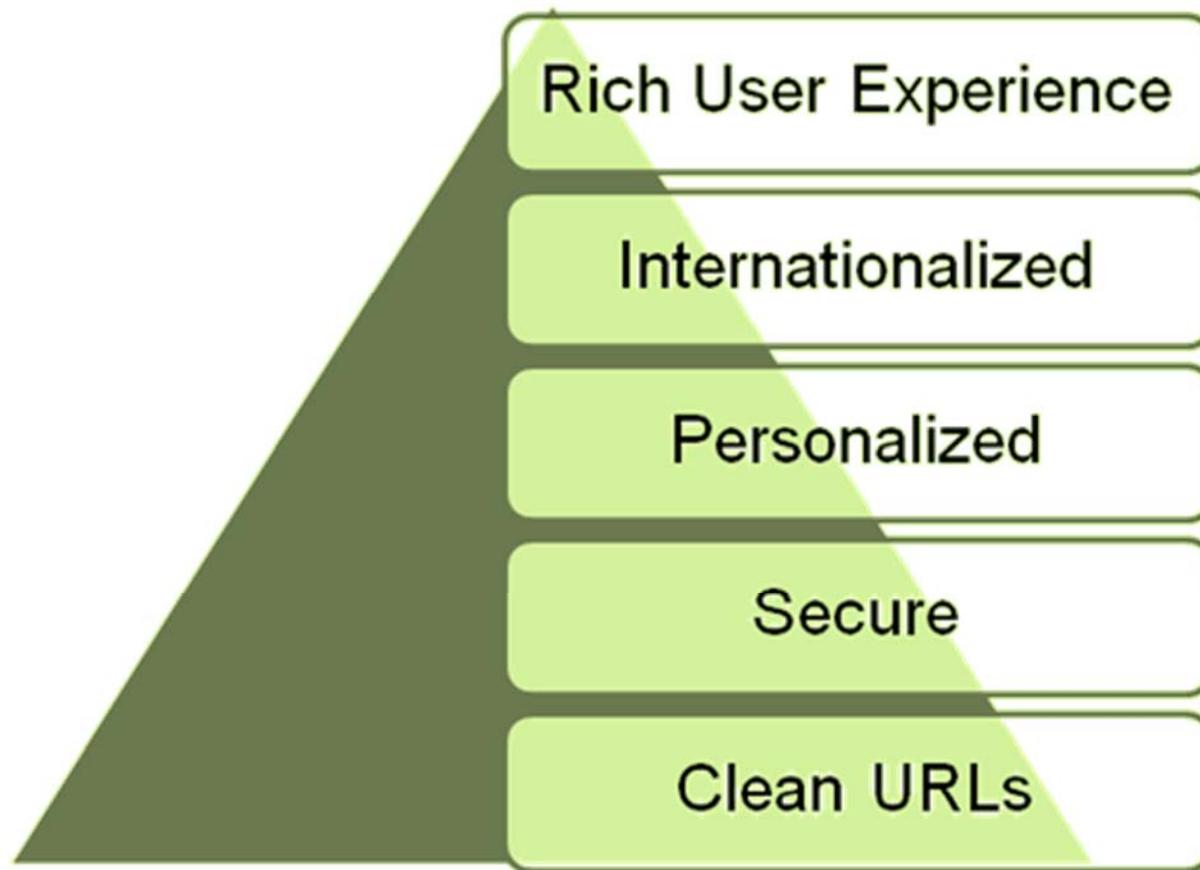
- Two applications, one database
- Headless Application: RewardNetwork
 - Record dining events and assign rewards points
- Used by multiple clients:
 - RewardsOnline
 - Remote terminals in restaurants
- Web Application: RewardsOnline
 - Find and update member accounts



Architecture Diagram



Web UI Requirements





UI patterns to apply

- Working with data entry forms
- Application transaction with no duplicate submit
- Data paging with Ajax and partial rendering
- CompositeView pattern





Road Map

- Tools you will work with
- Problems you will solve
- URL Design





Web Resources

- URLs should represent unique resources
 - /accounts# a collection of accounts
 - /accounts/123# a specific account
- Resources are a Web application's core value
 - They can be bookmarked, emailed, etc.



URL Templates

- An important technique for clean URL design
 - Embed information in URL segments
 - Resulting URI represents a unique resource
- Example templates
 - /accounts/{accountId}
 - /accounts/{accountId}/beneficiaries
 /{beneficiaryId}
- Example URL instances
 - /accounts/123
 - /accounts/123/beneficiaries/456

More on this later in the REST modules

URL Rewriting (1)

- Another useful technique for clean URL design
- URLs often contain a servlet-specific segment
 - /**admin**/accounts/3
- Two common approaches to overcome this
 - mod_rewrite
(Apache Web Server module)
 - UrlRewriteFilter
(Servlet filter)





URL Rewriting (2)

- Spring MVC provides a simpler alternative (mapping servlet requests to the root context)
 - Allows simple URLs like /accounts/3 without rewriting
 - Discussed in next section





Summary

After completing this lesson, you should have learnt:

- The tools you will work with
- The problems you will solve
- URL Design





LAB

Getting familiar with
the development environment



Spring Introduction

Spring Background for the
Spring Web and Integration courses



Overview

After completing this lesson, you should be able to:

- Describe Introduction to Spring Configuration
- Describe Bean Lifecycle
- Understand how to simplify configuration
- Describe Integration Testing with Spring



Lesson Roadmap

- Introduction to Spring Configuration
- Bean Lifecycle
- Simplifying Configuration
- Integration Testing with Spring

Goal of the Spring Framework



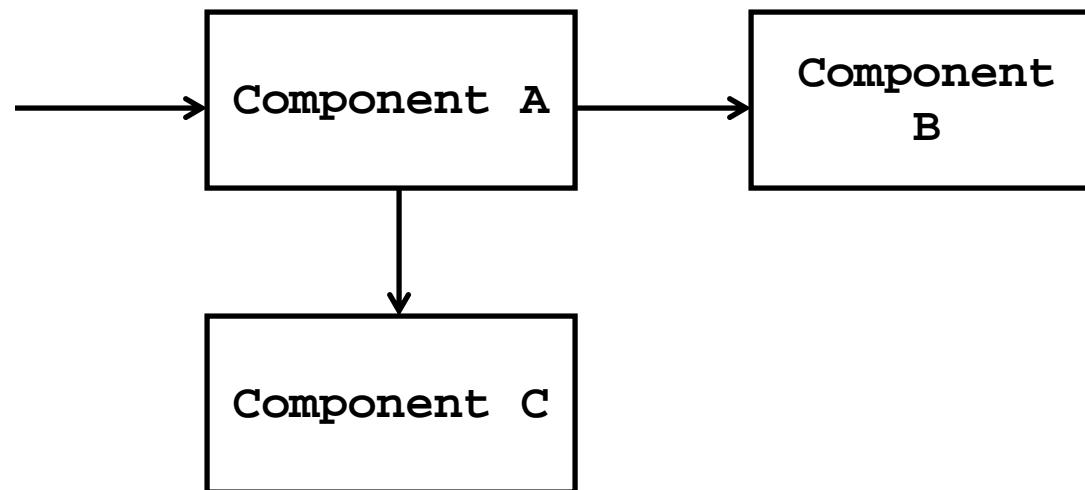
- Provide comprehensive infrastructural support for developing enterprise Java™ applications
 - Spring deals with the plumbing
 - So you can focus on solving the domain problem
- Key Principles
 - DRY – Don't Repeat Yourself
 - SoCs – Separation of Concerns



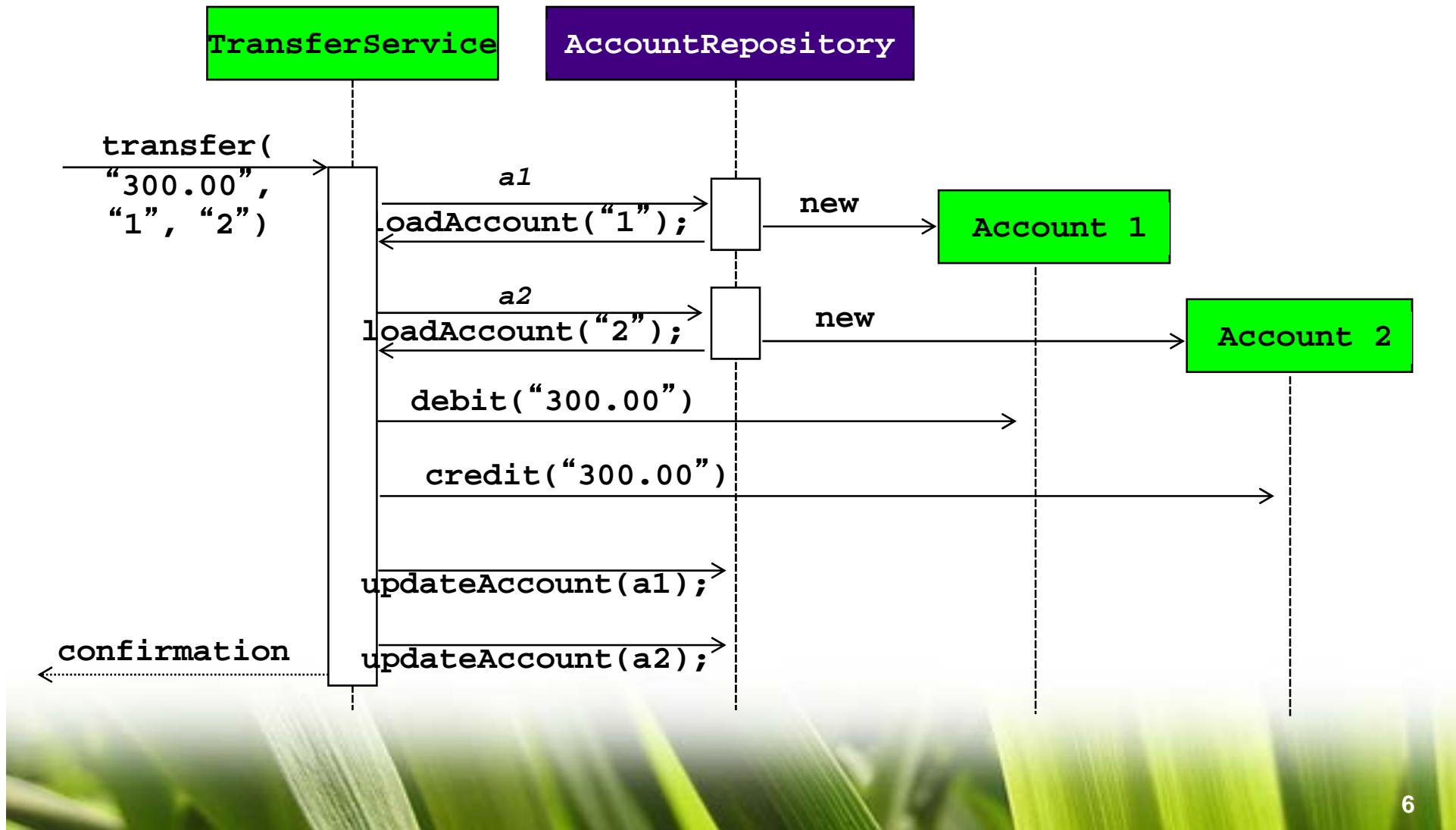
Application Configuration



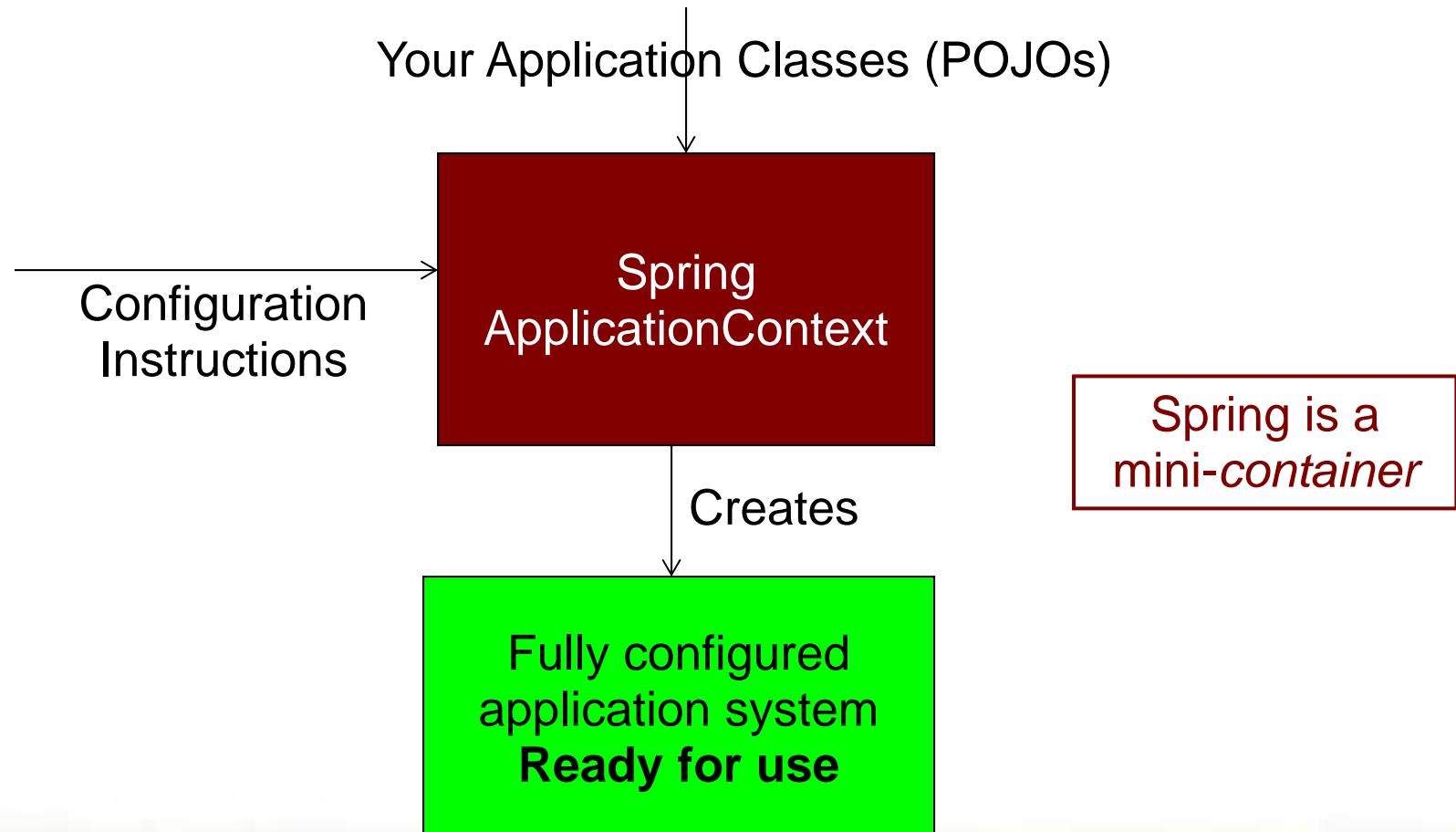
- A typical application system consists of several parts working together to carry out a use case



Example: A Money Transfer System

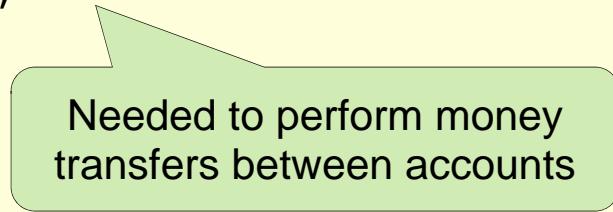


How Spring Works



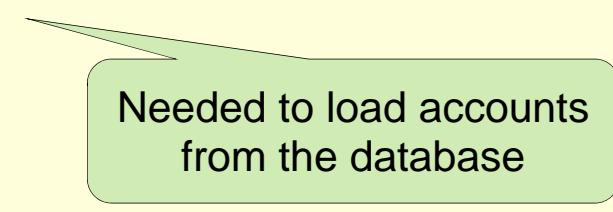
Your Application Classes

```
public class TransferServiceImpl implements  
TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```



Needed to perform money transfers between accounts

```
public class JdbcAccountRepository implements  
AccountRepository {  
    public setDataSource(DataSource ds) {  
        this.dataSource = ds;  
    }  
    ...  
}
```



Needed to load accounts from the database

Configuration Instructions



The Dependency Injection Pattern

```
<beans>
    <bean id="transferService" class="app.impl.TransferServiceImpl">
        <constructor-arg ref="accountRepository" />
    </bean>

    <bean id="accountRepository"
          class="app.impl.JdbcAccountRepository">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="dataSource"
          class="com.oracle.jdbc.pool.OracleDataSource">
        <property name="URL"
                  value="jdbc:oracle:thin:@localhost:1521:BANK" />
        <property name="user" value="moneytransfer-app" />
    </bean>
</beans>
```

application-config.xml

Creating and Using the Application



```
// Create the application from the configuration
ApplicationContext context = new
    ClassPathXmlApplicationContext("application-config.xml");

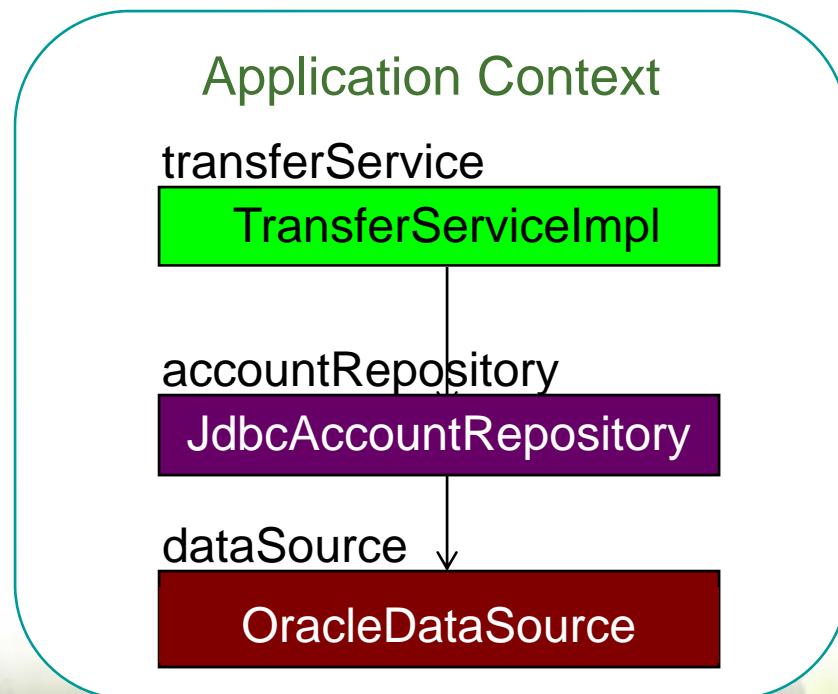
// Look up the application service interface
TransferService service =
    (TransferService) context.getBean("transferService");

// Use the application
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```

Inside the Spring Application Context



```
// Create the application from the configuration
ApplicationContext context =
    new ClassPathXmlApplicationContext
        ("application-config.xml");
```



Constructor vs Setter Injection?

- Constructor
 - Mandatory arg
 - Immutable arg
 - Concise in Java
 - Throw an exception
 - invalidates object
- Setter
 - Optional arg
 - Changeable arg
 - JavaBean pattern
 - Can be inherited
 - Named

Spring supports both!



Essentially the same rules as standard Java – no need to modify your class, just to use Spring. Be consistent on your project

Partitioning Configuration

```
<beans>
    <bean id="transferService" class="app.impl.TransferServiceImpl">
        <constructor-arg ref="accountRepository" />
    </bean>

    <bean id="accountRepository"
          class="app.impl.JdbcAccountRepository">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

application-config.xml

```
<beans>
    <!-- creates an in-memory database
         populated with test data for fast testing -->
    <bean id="dataSource"
          class=". . . EmbeddedDatabaseFactoryBean">
        <property name="databasePopulator" ref="populator" />
    </bean>

    <bean id="populator" . . .>
```

test-infrastructure-config.xml

Substitutable for multiple environments



Bootstrapping Multiple Environments

- In a test environment

```
// Create the application from the configuration
ApplicationContext context =
    new ClassPathXmlApplicationContext(
        "application-config.xml", "test-infrastructure-
        config.xml" );
```

- In a production Oracle environment

```
// Create the application from the configuration
ApplicationContext context =
    new ClassPathXmlApplicationContext(
        "application-config.xml", "oracle-
        infrastructure-config.xml" );
```

Working with prefixes

- Default location rules can be overridden

```
// Create the application from the configuration  
ApplicationContext context =  
    new ClassPathXmlApplicationContext(  
        "config/dao-config.xml", "config/service-config.xml",  
        "file:oracle-infrastructure-config.xml" );
```

Prefix

- Various prefixes
 - classpath:
 - file:
 - http:



These prefixes can be used anywhere Spring needs to deal with resources (not just in constructor args to application context)

Accessing a Bean in Spring 3

- Cast is not compulsory anymore

```
ApplicationContext context = new  
    ClassPathXmlApplicationContext(...);  
  
// Classic way: cast is needed  
ClientService service1 = (ClientService)  
    context.getBean("clientService");  
  
// Since Spring 3.0: no more cast, type is a method param  
ClientService service2 = context.getBean("clientService",  
    ClientService.class);  
  
// No need for bean id if type is unique  
ClientService service3 = context.getBean(ClientService.class);
```

New in Spring 3.0

Always cast to an interface – not implementation

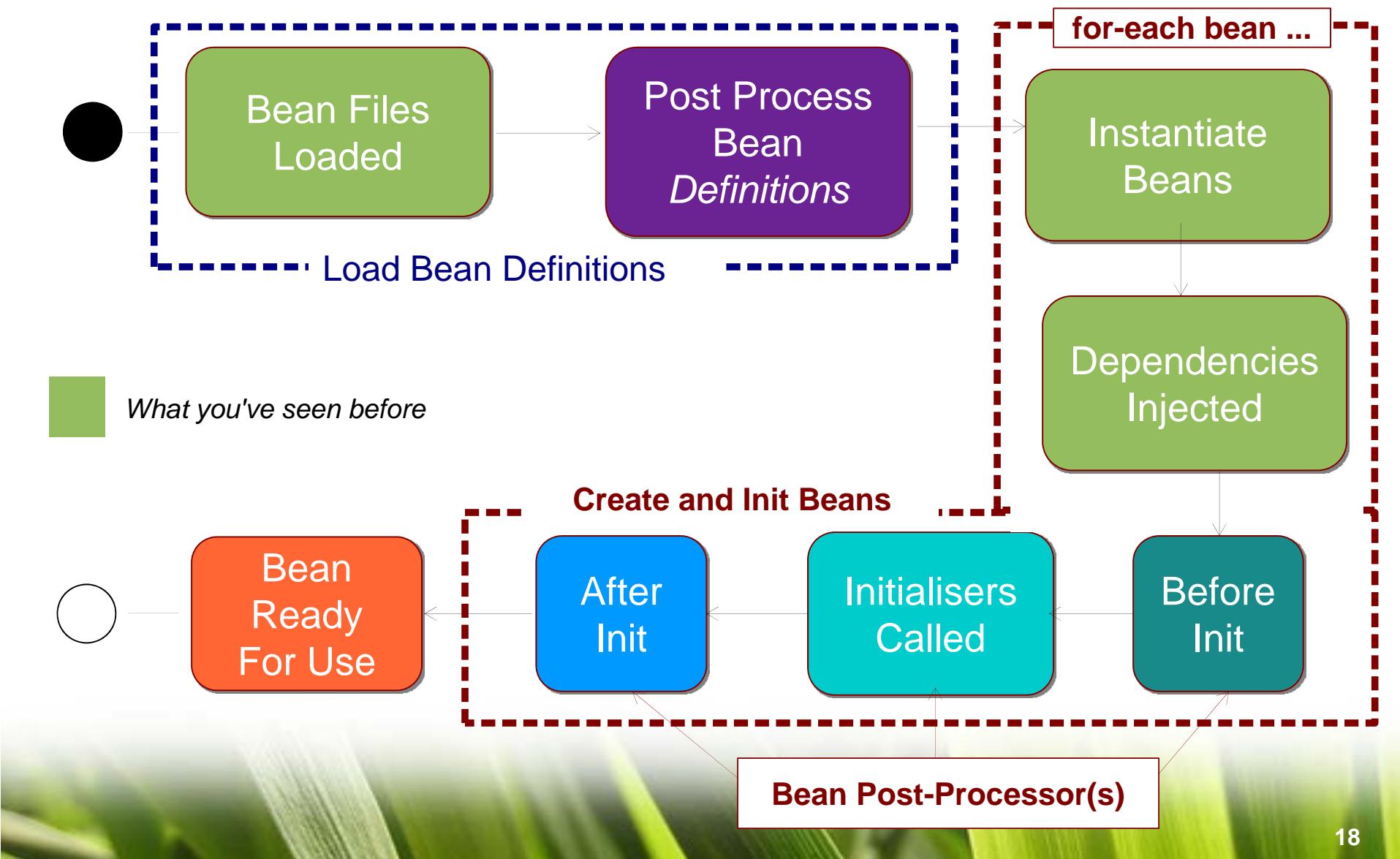


Lesson Roadmap

- Introduction to Spring Configuration
- Bean Lifecycle
- Simplifying Configuration
- Integration Testing with Spring



Bean Initialization Steps



Lifecycle Activities

- Bean Factory Post-Processor
 - Post-processes definitions
 - PropertyPlaceholderConfigurer
- Bean Post-Processor
 - Post processes the actual beans
 - Extra initialization
 - @PostConstruct, init-method
 - Add behaviour
 - Wrap bean in an AOP proxy
 - Security, @Transactional ...
 - Never cast a bean to its implementation



Using a PropertyPlaceholderConfigurer



Most commonly used BeanFactoryPostProcessor

```
<beans>
    <bean id="dataSource"
          class="com.oracle.jdbc.pool.OracleDataSource">
        <property name="URL" value="${datasource.url}" />
        <property name="user" value="${datasource.user}" />
    </bean>
```

Variables to replace

```
    <context:property-placeholder location="/WEB-
INF/configuration.properties" />
```

File where the variable
values reside

```
</beans>
```

Hides and simplifies the actual
PropertyPlaceholderConfigurer
bean definition

configuration.properties

```
datasource.url=jdbc:oracle:thin:@localhost:1521:BANK
datasource.user=moneytransfer-app
```

Easily editable

Importing a Custom XML Namespace

- “Programming instructions” for Spring

Assign namespace a prefix

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/bean"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation=
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-
context-3.0.xsd">

    <!-- You can now use <context:* /> tags -->

</beans>
```

Associate namespace with its XSD

Use *namespace* tab on XML editor in STS

Bean Post Processors I

- Initialize a Bean



- A bean can optionally register for one or more initialization callbacks
 - Useful for executing custom initialization behaviors
- There's more than one way to do it
 - JSR-250 @PostConstruct
 - <bean/> init-method attribute
 - Implement Spring's InitializingBean interface
 - See online documentation, not covered here



Registering for Initialization

```
public class TransferServiceImpl {  
    @PostConstruct  
    protected void init() {  
        // your custom initialization code  
    }  
}
```

```
<beans>  
    <bean id="transferService"  
          class="example.TransferServiceImpl" />  
    <context:annotation-config/>  
</beans>
```

Tells Spring to call this method *after* dependency injection

No restrictions on method name or visibility - must have *no args*

Enables several post-processors that recognize initialization annotations

Initializing a Bean You Do Not Control



- Use `init-method` to initialize a bean you do not control the implementation of

```
<bean id="broker"
      class="org.apache.activemq.broker.BrokerService"
      init-method="init">
    ...
</bean>
```

Class with no Spring dependency

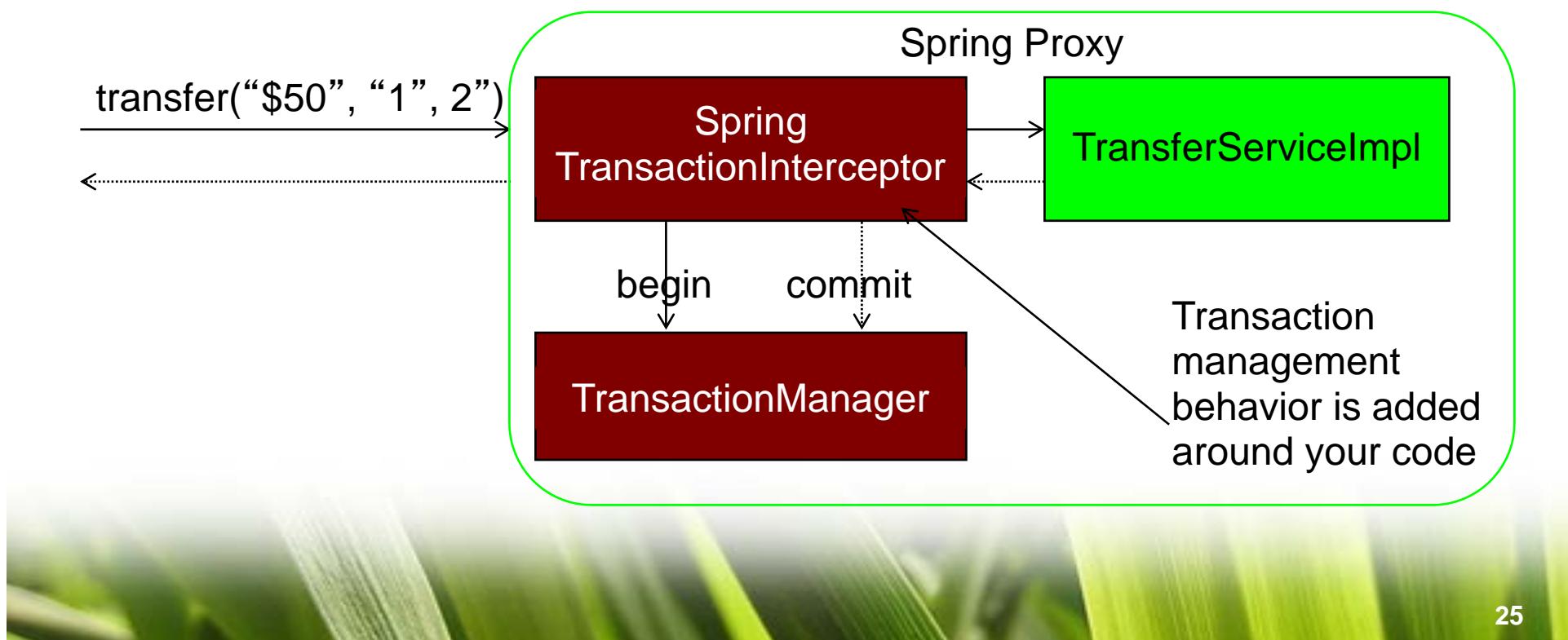
Method can be any name, any visibility, but must have *no* arguments

Bean Post Processors II

- Proxy Power



- A BeanPostProcessor may wrap your beans in a *dynamic proxy*
 - add behavior to application *transparently*



Common Example: @Transactional Proxy



- Annotate class and/or methods

default settings

```
@Transactional(timeout=60)
public class RewardNetworkImpl implements RewardNetwork {

    public RewardConfirmation rewardAccountFor(Dining d) {
        // atomic unit-of-work
    }

    @Transactional(timeout=45)
    public RewardConfirmation
    updateConfirmation(RewardConfirmantion rc) {
        // atomic unit-of-work
    }
}
```

overriding attributes at
the method level

Class wrapped in a proxy

```
<tx:annotation-driven/>
<bean id="transactionManager" ... />
```

Activate Bean Post-processor



Lesson Roadmap

- Introduction to Spring Configuration
- Bean Lifecycle
- Simplifying Configuration
- Integration Testing with Spring



Before Inner Bean

```
<beans>

    <bean id="restaurantRepository"
          class="rewards.internal.restaurant.JdbcRestaurantRepository">
        <property name="dataSource" ref="dataSource" />
        <property name="benefitAvailabilityPolicyFactory"
                  ref="factory" />
    </bean>

    <bean id="factory"
          class="rewards.internal.restaurant.availability.
              DefaultBenefitAvailabilityPolicyFactory">
        <constructor-arg ref="rewardHistoryService" />
    </bean>
    ...
</beans>
```

Can be referenced by other beans
(even if it should not be)

With an Inner Bean

```
<beans>

    <bean id="restaurantRepository"
        class="rewards.internal.restaurant.JdbcRestaurantRepository">
        <property name="dataSource" ref="dataSource" />
        <property name="benefitAvailabilityPolicyFactory">
            <bean class="rewards.internal.restaurant.availability
                .DefaultBenefitAvailabilityPolicyFactory">
                <constructor-arg ref="rewardHistoryService" />
            </bean>
        </property>
    </bean>
    ...
</beans>
```

Inner bean has no id (it is anonymous)
Cannot be referenced outside this scope

Importing Configuration Files

- Use the `import` tag
 - encapsulates including another config file

```
<beans>
    <import resource="accounts/account-config.xml" />
    <import resource="restaurant/restaurant-config.xml" />
    <import resource="rewards/rewards-config.xml" />
</beans>
```

application-config.xml

```
<beans>
    <import resource="application-config.xml"/>
    <bean id="dataSource"
class="example.TestDataSourceFactory" />
</beans>
```

system-test-config.xml

```
new ClassPathXmlApplicationContext("system-test-config.xml");
```

Client only
imports one file

Injection of Properties

```
<bean id="notificationService"
      class="example.MyNotificationService">
    <property name="emailMappings">
      <util:properties location="/WEB-
      INF/mail.properties" />
    </property>
</bean>
```

Loads a
java.util.Properties from
the location

```
<bean id="notificationService"
      class="example.MyNotificationService">
    <property name="emailMappings">
      <value>
        server.host=mailhost.net
        server.port=1234
      </value>
    </property>
</bean>
```

Creates
java.util.Properties
from value text



SpEL XML Example

```
<bean class="mycompany.RewardsTestDatabase">
    <property name="databaseName"
              value="#{systemProperties.databaseName}" />
    <property name="username"
              value="#{dbProps.username}" />
    <property name="keyGenerator"
              value="#{strategyBean.databaseKeyGenerator}" />
</bean>

<util:properties id="dbProps"
                 location="classpath:db-config.properties"/>

<bean id="strategyBean" class="mycompany.DefaultStrategies">
    <property name="databaseKeyGenerator" ref="myKeyGenerator"/>
    ...
</bean>
```

Annotation DI

- XML-based dependency injection

```
<bean id="transferService"
      class="com.springsource.TransferServiceImpl">
    <constructor-arg ref="accountRepository" />
</bean>
```

External configuration

- Annotation-based dependency injection

```
@Component
public class TransferServiceImpl
    implements TransferService {
    @Autowired
    public TransferServiceImpl(AccountRepository repo) {
        this.accountRepository = repo;
    }
}
```

Annotations embedded with POJOs

```
<context:component-scan base-package="com.springsource"/>
```

Minimal XML Config

Usage of @Autowired

- Constructor-injection

```
@Autowired  
public TransferServiceImpl(AccountRepository repo) {  
    this.accountRepository = repo;  
}
```

Can also contain multiple params

- Method-injection

```
@Autowired  
public void setAccountRepository(AccountRepository r){  
    this.accountRepository = r;  
}
```

Method name does not have to start with “set”

- Field-injection – be careful

```
@Autowired  
private AccountRepository accountRepository;
```

XML vs Annotations syntax



- Same options are available

```
@Component( "transferService" )
@Scope( "prototype" )
@Lazy( false )
public class TransferServiceImpl
implements TransferService {

    @PostConstruct
    public void init() {    //...
}

    @PreDestroy
    public void destroy() {    //...
}
}
```

annotations

```
<bean id="transferService"
scope="prototype"
lazy-init="false"
class="TransferServiceImpl"

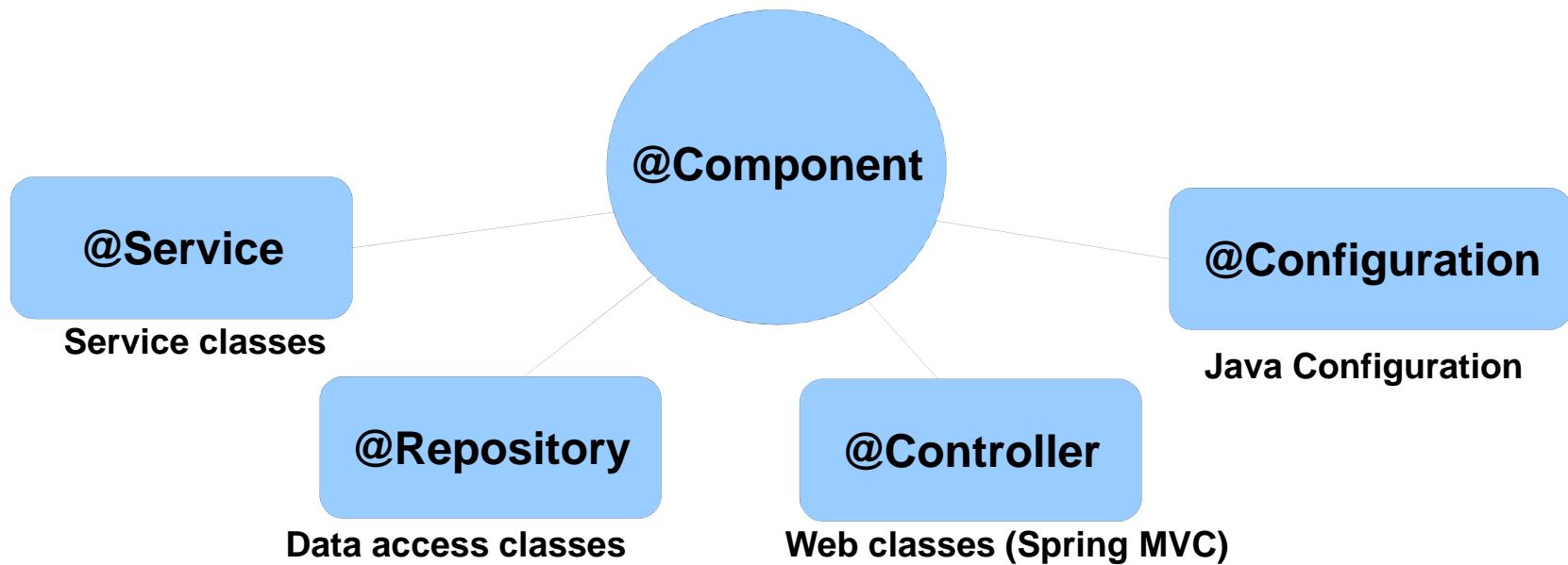
init-method="init"

destroy-method="destroy" />
```

xml config

Stereotype annotations

- Spring framework stereotype annotations



Other Spring projects provide their own stereotype annotations
(Spring WS, Spring Integration...)

@Value and systemProperties



- Use @Value for SpEL expressions in class
 - On a field or method
 - Parameter of autowired method/constructor

```
@Value("#{ systemProperties['user.region'] }")
private String defaultLocale;

@Value("#{ systemProperties['user.region'] }")
public void setDefaultLocale(String defaultLocale) { ... }

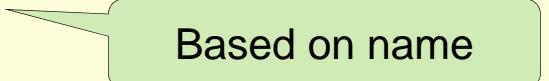
@Autowired
public void configure(AccountRepository accountRepository,
    @Value("#{ systemProperties['user.region'] }") String
        defaultLocale) {
    ...
}
```



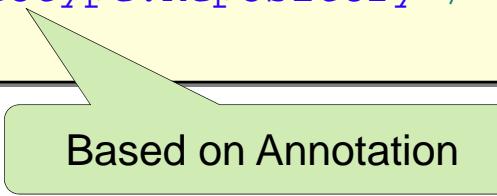
SpEL: Spring Expression Language (since Spring 3.0)

Controlling the Scanner

- Pick a sensible base-package
 - or list of packages
- Use filters to include or exclude beans
 - Based on type or class-level annotation

```
<context:component-scan base-package="com.acme.app">
    <context:include-filter type="regex"
        expression=".*/Stub.*Repository"/>
        
        Based on name

    <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```



Based on Annotation

Using Annotations Without Component Scanning

- Use `<context:annotation-config/>`
 - Processes all DI annotations
 - Does *not* perform component-scanning
 - Beans must be declared explicitly

```
<beans ...>
    <context:annotation-config/>

    <bean id="transferService" class="transfer.TransferService"/>
    <bean id="accountRepository" class="transfer.JdbcAccountRepository"/>
</beans>
```



`<property />` or `<constructor-arg/>` are not needed any more because classes use `@Autowired`



Lesson Roadmap

- Introduction to Spring Configuration
- Bean Lifecycle
- Simplifying Configuration
- Integration Testing with Spring



Unit Testing vs. Integration Testing



- Unit Testing
 - Tests one unit of functionality
 - Keeps dependencies minimal
 - Isolated from the environment - *including Spring*
- Integration Testing
 - Tests multiple units working together
 - Integrates infrastructure
 - Spring testing facilitates this
 - Test *without* running up your *JEE container*



Using Spring's test support



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:system-test-config.xml")
public final class TransferTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void performTransferTest() {
        transferService.transfer(...);
        ...
    }
}
```

Run with Spring support

Point to system test configuration file

Define a dependency

Test the system as normal

See *spring-test.jar*



Summary

After completing this lesson, you should have learnt to:

- Configure Spring
- Describe Bean Lifecycle
- How to simplify configuration
- Perform Integration Testing with Spring





Getting Started With Spring Web MVC



Overview

After completing this lesson, you should be able to:

- Describe Spring MVC
- Describe DispatcherServlet
- Explain Controller Programming Model
- Describe Spring MVC Views
- Describe Spring MVC Simplifying Configuration





Road Map

- Spring MVC Overview
- The DispatcherServlet
- Controller Programming Model Overview
- Spring MVC Views
- Simplifying Configuration



Spring Web MVC



- Popular request-driven framework
 - approach familiar to Struts users
- The foundation for all Spring Web modules
- Distributed with the Spring Framework
 - `spring-web.jar`
 - `spring-webmvc.jar`



Spring Web MVC Strengths



- Easy to understand and to get started with
- Uses Spring for its own configuration
- Flexible and extensible
- Supports various view technologies





Road Map

- Spring MVC Overview
- **The DispatcherServlet**
- Controller Programming Model Overview
- Spring MVC Views
- Simplifying Configuration



The DispatcherServlet



- Heart of Spring MVC
- Implements the Front Controller Pattern
 - analogous to Struts ActionServlet / JSF FacesServlet
- Handles every incoming request
- Coordinates request-handling activities

Martin Fowler's POEAA
Core J2EE Patterns

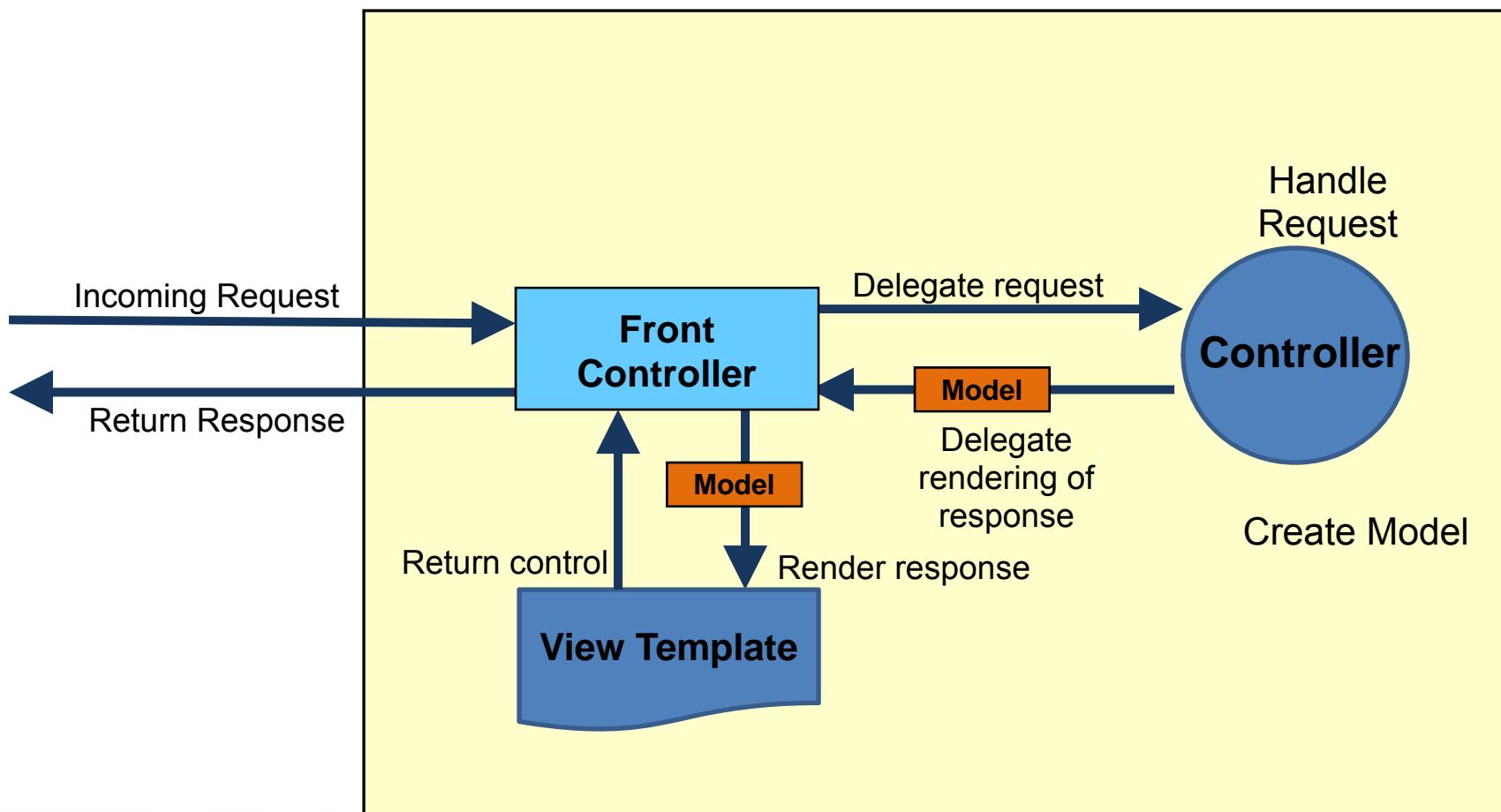


Request Processing

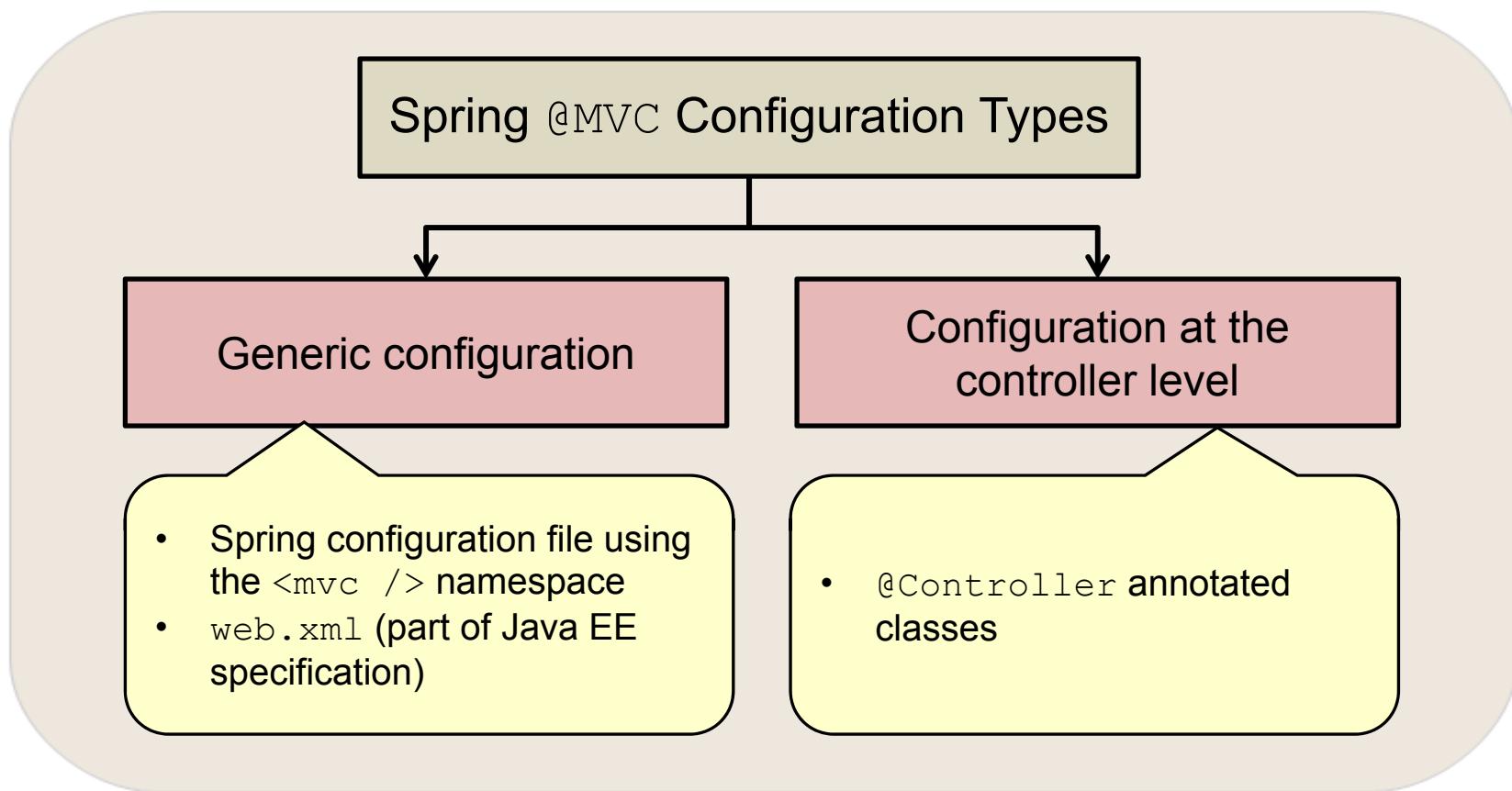
- The DispatcherServlet doesn't do all the work
 - delegates to other components
- Spring MVC infrastructure components
 - handler mappings
 - handler adapters
 - view resolvers
- User-provided web components
 - controllers
 - handler interceptors



Request Processing Lifecycle



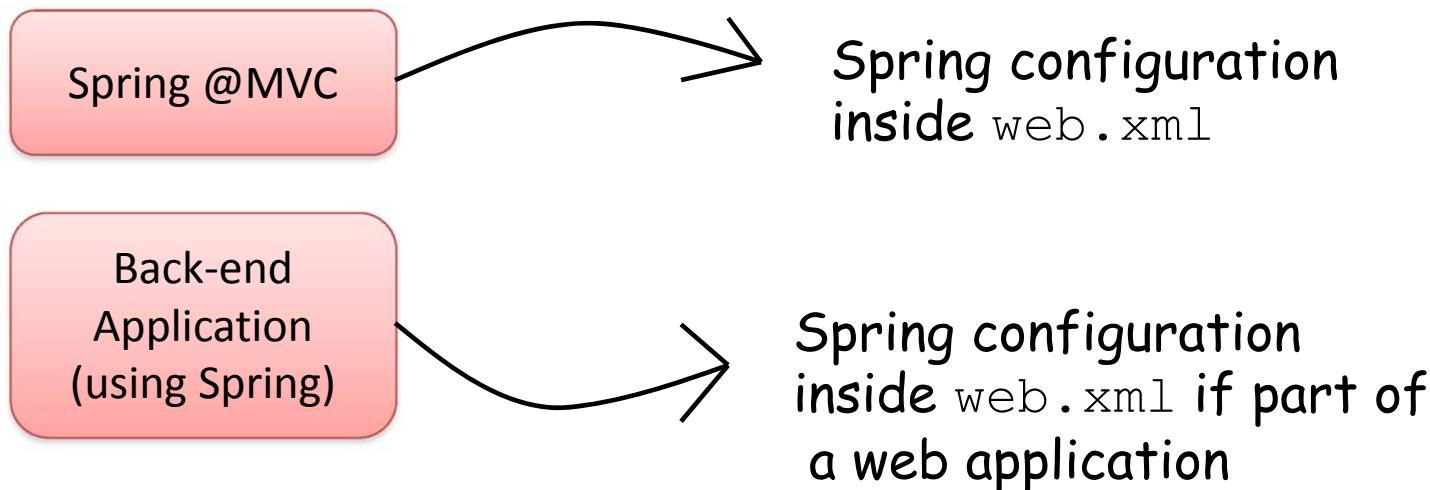
Spring @MVC Configuration



This diagram only shows about Spring @MVC configuration. Integration with other Spring configuration files will be shown in the next slide

Generic configuration

- How does Spring @MVC interact with Spring?



- Two ways
 - All-in-one
 - Separate configurations



First option: all-in-one

```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/web-config.xml
            /WEB-INF/spring/application-config.xml
        </param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/main/*</url-pattern>
</servlet-mapping>
```

Spring @MVC

Back-end

web.xml

All-In-One

- All Spring configuration is loaded by the Spring @MVC DispatcherServlet
 - It works well if Spring @MVC DispatcherServlet is the only entry point to Spring configuration

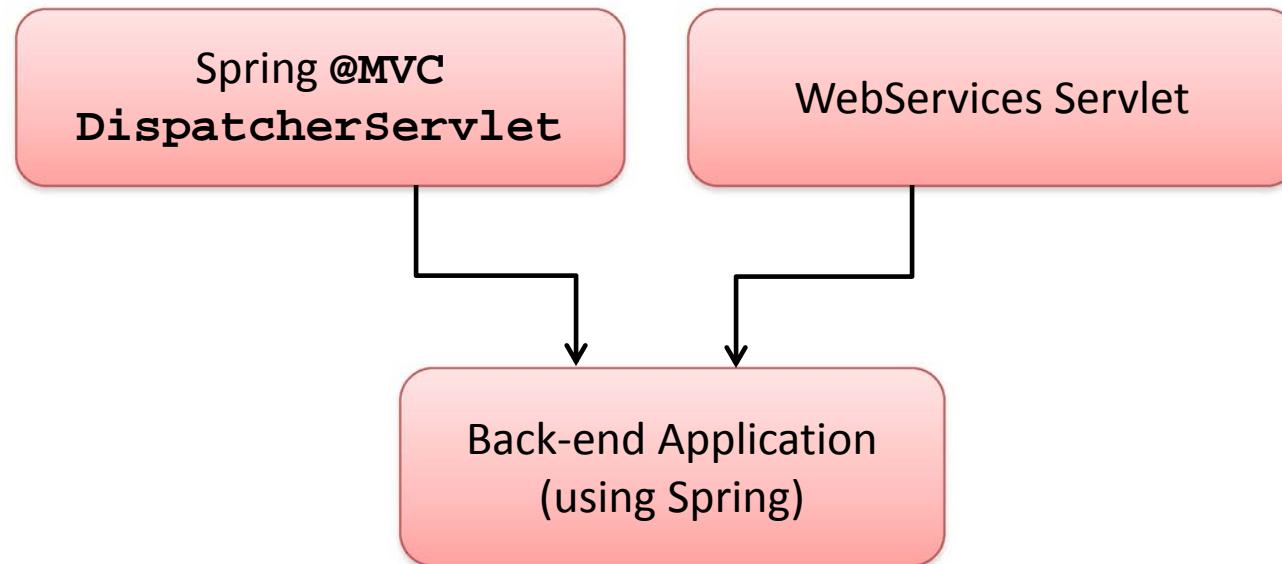
DispatcherServlet configuration

- Controllers
- Transactions
- Data access
- ...



All-In-One Limitation

- How to proceed when Spring @MVC is not the only entry point?



- In that case, the Spring back-end configuration should be declared in a separate listener

Separate configurations

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/app-config.xml</param-value>
</context-param>

<listener> <listener-class>
    org.springframework.web.context.ContextLoaderListener
</listener-class> </listener>

<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/web-config.xml</param-value>
    </init-param>
</servlet>

<servlet-mapping>...</servlet-mapping>
```



web.xml

Configuration options

```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/*-beans.xml
            classpath:com/springsource/application-config.xml
        </param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/main/*</url-pattern>
</servlet-mapping>
```

Wildcards are accepted

Can use the classpath prefix



Road Map

- Spring MVC Overview
- **The DispatcherServlet**
- Controller Programming Model Overview
- Spring MVC Views
- Simplifying Configuration



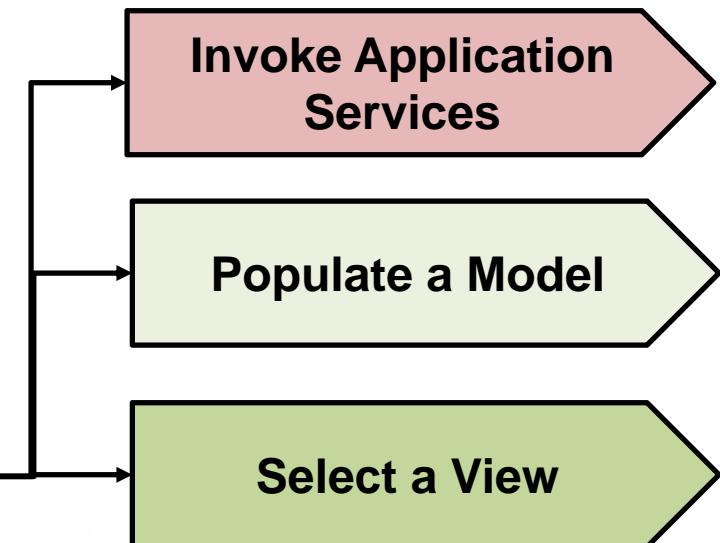
-
- Annotation based programming model
 - Since Spring 2.5
 - Supersedes the existing Controller base-class hierarchy
 - Deprecated in Spring 3.0

A close-up photograph of green grass blades, serving as a background for the slide.

@MVC

Steps To Creating A Controller

- 1 **Declare controller class as a POJO**
- 2 **Define one or more request-handling methods**
- 3 **Map URLs to methods using annotations**
- 4 **Implement method body**



Controller Example

```
@Controller  
public class AccountsController {  
  
    private AccountManager accountManager;  
  
    @Autowired  
    public AccountsController(AccountManager accountManager) {  
        this.accountManager = accountManager;  
    }  
  
    @RequestMapping("/accounts")  
    public String list(Model model) {  
        model.addAttribute("accountList",  
                          accountManager.findAllAccounts());  
        return "/WEB-INF/accounts/list.jsp";  
    }  
}
```

Declare class as a Spring bean and as a Spring MVC Controller

Inject Spring-managed service

Map URL to method

Mapping Requests (1)

- By URL only
 - /url/path

```
@RequestMapping("/springframework/cert")
public String list(Model model) { ... }
```

- By URL and request method
 - GET /url/path

```
@RequestMapping(value="/springframework/cert",
method=RequestMethod.GET)
public String list(Model model) { ... }
```

Mapping Requests (2)

- By the presence of a request parameter
 - /url/path?paramName=anything

```
@RequestMapping(value="/springframework/cert",
                 params={ "paramName" })
public String list(Model model) { ... }
```

- By parameter with a specific value
 - /url/path?paramName=paramValue

```
@RequestMapping(value="/springframework/cert", params=
                 { "stuName=Clarence" })
public String list(Model model) { ... }
```

RequestMapping at class and method levels

- `@RequestMapping` can be applied to class and method level
 - Class mapping is matched, then request

```
@Controller
@RequestMapping("/accounts")
public class AccountsController {
    @RequestMapping(method=RequestMethod.GET)
    public String list(Model model) {
        ...
    }
    @RequestMapping(value="/method2"
                    method=RequestMethod.GET)
    public String method2(Model model) {
        ...
    }
}
```

Any method without `@RequestMapping` is ignored



Select A View

- A Controller selects the view to render the model
- By default the view name is expected to be a JSP path

```
@RequestMapping(method=RequestMethod.GET)
public String list(Model model) {
    ...
    return "/WEB-INF/accounts/list.jsp";
}
```

- The returned String is known as the "view name"



Choose Method Arguments



- Request-handling methods have flexible signatures
- List of expected arguments (supply in any order)
 - Model
 - HttpServletRequest, HttpServletResponse
 - HttpSession
 - Locale
 - Principal
 - more options explained later



Access Request Parameters (1)

- When using URI Templates, access the parameters using the `@PathVariable` annotation
 - Parameter is extracted from the request
 - Type conversion is applied (primitive or wrapper)

```
@RequestMapping(value="/{id}")
public void show(@PathVariable ("id") long id,
                  Model model) {
    ...
}
```

`http://restbank.net/teller/accounts/123456789`

Access Request Parameters (2)

- @PathVariable annotations can be limited via regular expressions
 - Useful for resolving ambiguity in URLs

```
@RequestMapping(value="/{id:[\\d]*}")
public void show(@PathVariable("id") String id){
    ...
}
```

Will only match numeric IDs

- Annotation can be optional
 - Defaults to Java argument of same name
 - BUT: *Must* compile with debug symbols enabled

```
@RequestMapping(value="/{id}")
public void show(String id) { ... }
```

Access Request Parameters (3)



- Access URL parameters via `@RequestParam`
 - Parameter is extracted from the request
 - Type conversion is applied (primitive or wrapper)
 - Causes an exception if missing or wrong type

```
@RequestMapping(value="/account/show")
public void show(@RequestParam ("id") long id,
                  Model model) {
    ...
}
```

`http://mvcbank.org/teller/accounts/show?id=123456789`

Access Request Parameters (4)



- Parameter can be optional
 - Must be an object (set to null if not in URL)

```
@RequestMapping(value="/account/show")
public void show(@RequestParam
                  (value="id", required=false) Long id){
    ...
}
```

- Parameter *name* can be optional
 - Defaults to Java argument name

```
@RequestMapping(value="/account/show")
public void show(@RequestParam long id) {
    ...
}
```

Defaults to Id

@PathVariable Vs. @RequestParam



- `@PathVariable` assumes the variable is provided in the URL path

```
http://restbank.net/teller/accounts/{id}  
http://restbank.net/teller/accounts/3
```

- `@RequestParam` assumes the variable is provided after the URL path

```
http://mvcbank.org/teller/accounts/show?id=3
```

Formatting Request Parameters

- Date and number formats can be defined
 - `@PathVariable` and/or `@RequestParam` parameters
 - Avoids custom Property Editors for dates or numbers

```
@RequestMapping(value="/appointments/{day}")
public void show(@PathVariable ("day")
                  @DateTimeFormat (iso=ISO.DATE) Date day,
                  @RequestParam ("hourlyRate")
                  @NumberFormat (style=Style.CURRENCY)
                  double rate {
    ...
}
```

// Matches: /appointments/2012-03-14?hourlyRate=\$24.50

Set to 24.50

Also: PERCENT and numeric patterns: #,###.##

The <spring:url/> Tag



- Drop-in replacement for JSTL <c:url> tag
- Allows creation of template URLs
 - URI variables are URL encoded

```
<spring:url value="/accounts/{number}">
    <spring:param name="number"
                  value="${account.number}" />
</spring:url>
```

Add Model Attributes

- A model is always created
- Access the model to add attributes to it

```
@RequestMapping("/accounts")
public String list(Model model) {
    model.addAttribute("accountList",
                      accountManager.findAllAccounts());
    ...
}
```

- All attributes available in the JSP for rendering

Test Controllers



```
public class AccountsControllerTests {
    private AccountsController controller;

    @Before
    public void setUp() throws Exception {
        controller = new AccountsController(
            new StubAccountManager());
    }

    @Test
    public void testList() throws Exception {
        Model model = new BindingAwareModelMap();
        controller.list(model);
        List<Account> accounts = (List<Account>)
            model.asMap().get("accountList");
        assertEquals(1, accounts.size());
    }
}
```

Instantiate the controller

Test its methods





Road Map

- Spring MVC Overview
- The DispatcherServlet
- Controller Programming Model Overview
- Spring MVC Views
- Simplifying Configuration



The View Abstraction



- A Strategy for rendering the model
- Variation by content type
 - HTML, Excel, PDF
- Variations by rendering technology
 - JSP, Freemarker, Velocity, Facelets



Available View Types



- Display Views
 - JSP, Tiles
 - FreeMarker, Velocity
- File-generating Views
 - POI, jExcelApi (Excel)
 - Itext (PDF)
 - JasperReports
 - XSLT transformation
- Data-delivery Views
 - JSON, Java-XML Marshalling,
 - Atom, RSS



Just a sample
Not a complete list!



The `ViewResolver` Abstraction

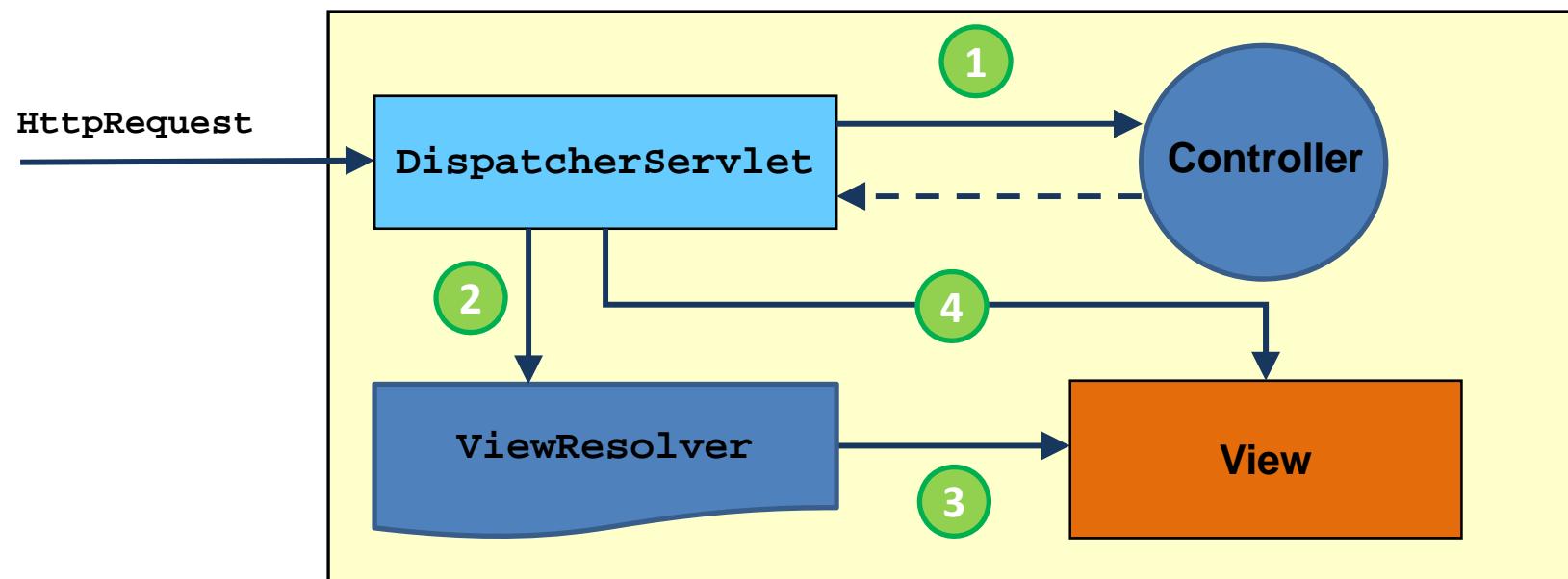


- A factory for creating View instances

```
public interface ViewResolver {  
    View resolveViewName(String viewName,  
                         Locale locale);  
}
```

- Decouples controller from view implementation
 - possible to switch from JSP to Tiles, etc.

View Resolution Sequence



1 Invoke "viewname"

2 ResolveView "viewname"

3 Create

4 render(model, request, response)

The Default viewResolver



- The default ViewResolver implementation is InternalResourceViewResolver
- It interprets view names as JSP paths
- Creates instances of JstlView



View Name Prefix and Suffix



- The InternalResourceViewResolver is commonly configured with default prefix/suffix

```
<bean class="org.springframework.web.servlet.view  
       .InternalResourceViewResolver">  
    <property name="prefix" value="/WEB-INF/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

- Controllers can then return "logical view name"

```
@RequestMapping("/accounts/list")  
public String list(Model model) {  
    ...  
    return "accounts/list";  
}
```

Maps to:
/WEB-INF/accounts/list.jsp



LAB

Getting Started With Spring MVC



Road Map

- Spring MVC Overview
- The DispatcherServlet
- Controller Programming Model Overview
- Spring MVC Views
- Simplifying Configuration
 - Simplifying Controller Setup
 - Applying Conventions
 - Mapping by Convention

The <mvc> Namespace

- Spring MVC uses the custom <mvc> namespace to simplify XML configuration
 - New in Spring 3.0
- Namespace tags simplify common configuration tasks

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans
                           /spring-beans-3.0.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc
                           /spring-mvc-3.0.xsd">
    ...

```

Mapping the DispatcherServlet



- Mapping the DispatcherServlet adds a servlet element to the URL
 - /**teller**/accounts/3
- As of Spring 3.0.4, the DispatcherServlet can be mapped to /
- An additional element must be defined to pass requests to Application Server
 - Used for static assets (images, javascript files, etc.)
 - Only required when mapping to /



Default DispatcherServlet Example

```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    ...
</servlet>

<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

web.xml

```
<mvc:annotation-driven/>
<mvc:default-servlet-handler/>
```

mvc-config.xml

Note: More on "annotation-driven" in the next module

Views Without a Controller

- Some views don't need a controller
 - We just want to render a view
- Use <mvc:view-controller> to declare such views
 - Specify URL and view name in configuration

```
<mvc:annotation-driven/>
<mvc:view-controller path="/login" view-name="/login"/>
<mvc:view-controller path="/welcome" view-name="/welcome"/>
```

Logical view names



Convention Over Configuration (CoC)



- Spring MVC has a number of conventions features
- Using conventions reduces the amount of configuration required
- Important to know these conventions
- Weigh up CoC convenience Vs long term maintenance, ease of understanding



Conventions for Model Attribute Names



- A model attribute name may be left unspecified
 - A default name is selected
 - based on the concrete *type* of the attribute
 - arrays and collections: *type* plus "*List*" suffix

```
Account acc = accountManager.findAccount(accountNumber);  
model.addAttribute(acc);           // Added as "account"  
  
MonetaryAmount amount =  
    accountManager.annualInterest(accountNumber);  
model.addAttribute(amount);        // Added as "monetaryAmount"  
  
List<Account> accounts = accountManager.findAllAccounts();  
model.addAttribute(accounts);      // Added as "accountList"
```

Convention for View Names



- Request-handling methods may leave view name unspecified:
 - return `null` or `void`
- A default *logical* view name is selected
 - Via `RequestToViewNameTranslator`
 - Leading slash + extension removed from URL
 - View resolver is required

```
@RequestMapping("/accounts/list.html")
public void list(Model model) { ... }
```

accounts/list

```
<mvc:view-controller path="/welcome"/>
```

welcome

Shortcut For Adding a Single Model Attribute



- A single model attribute can simply be returned

```
@RequestMapping("/accounts/list")
public List<Account> list() {
    return accountManager.findAllAccounts();
}
```

Model Attribute name convention used

```
@RequestMapping("/accounts/list")
public @ModelAttribute("accounts")
        List<Account> list() {
    return accountManager.findAllAccounts();
}
```

Optionally annotate return type with model attribute name

Convention for URL Mappings

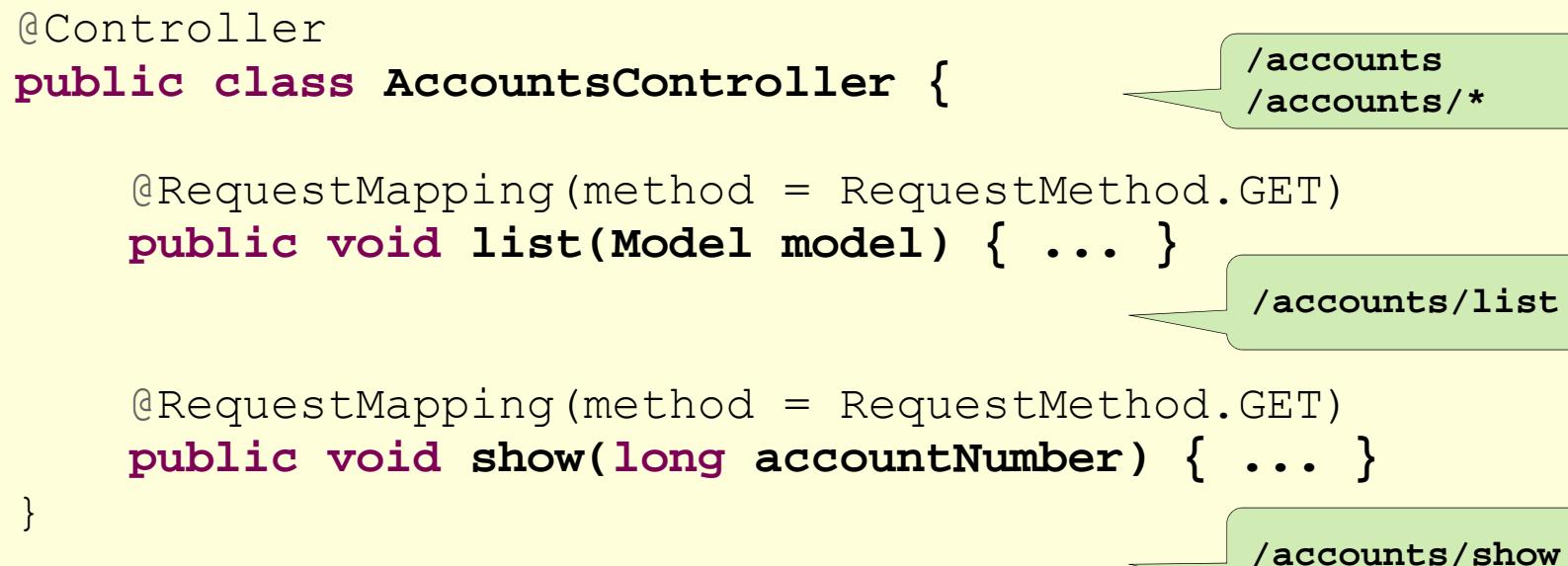
- The Controller Class Name Handler Mapping
 - Alternative to default Annotation based mapping
 - Requires configuration in bean-file
- Class name used to generate URL mappings
 - AccountsController → /accounts, /accounts/*
 - WelcomeController → /welcome, /welcome/*
- Method-level @RequestMapping further narrows down a method within a controller
 - 1st check HTTP request method (GET, POST, etc.)
 - 2nd check for presence of request parameter(s)
 - 3rd fall-back on the method name if necessary



Example Controller Mapped By Convention

- Mapped by using controller class name conventions with method-level annotations

```
@Controller  
public class AccountsController {  
  
    @RequestMapping(method = RequestMethod.GET)  
    public void list(Model model) { ... }  
  
    @RequestMapping(method = RequestMethod.GET)  
    public void show(long accountNumber) { ... }  
}
```



The code snippet shows a Java controller named AccountsController. It contains two methods: a GET request mapping to /accounts/list and another GET request mapping to /accounts/show. The code is annotated with @Controller, @RequestMapping, and Model. Callout boxes map the annotations to their corresponding URLs: /accounts for the class name and /accounts/* for the list method, and /accounts/show for the show method.

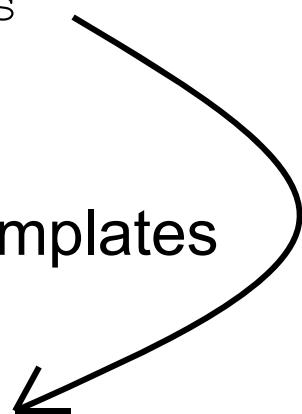
Handler Mapping Configuration

- Enable two ways of mapping requests
 - Annotations
 - Controller class name convention

```
<bean class="org.springframework.web...  
                  DefaultAnnotationHandlerMapping">  
    <property name="order" value="0">  
</bean>  
  
<bean class="org.springframework.web...  
                  ControllerClassNameHandlerMapping">  
    <property name="order" value="1">  
</bean>
```

A Note On URL Strategy

- Using Controller Class Name conventions results in a different URL strategy
 - GET /accounts/show?id=1 vs GET /accounts/1
 - GET /accounts/list vs GET /accounts
 - etc.
- Some support for conventions based on URI Templates is planned for Spring 3.1



RESTful style URLs





Summary

After completing this lesson, you should have learned to:

- Spring MVC
- DispatcherServlet
- Controller Programming Model
- Spring MVC Views
- Spring MVC Simplifying Configuration





Spring MVC Configuration Options

Making The Most of the Spring MVC Flexible Request
Processing Infrastructure



Overview

After completing this lesson, you should be able to:

- Describe Spring MVC Infrastructure Beans
- Describe URL Mappings
- Describe Handler Interceptors
- Describe Handler Adapters
- Describe Exception Resolvers
- Describe Message Source





Road Map

- Spring MVC Infrastructure Beans
- URL Mappings
- Handler Interceptors
- Handler Adapters
- Exception Resolvers
- Message Source



“Special” Spring MVC Beans



- DispatcherServlet looks for some beans by type
 - HandlerMapping
 - HandlerAdapter
 - ViewResolver
 - HandlerExceptionResolver
- Out-of-the box implementations of above interfaces are provided
- Common customizations via properties



Chaining Infrastructure Beans



- It's common to configure multiple infrastructure beans of the same type
- Chained infrastructure beans can be ordered
 - The first one to return a non-null value “wins”

```
<bean
    class="org.springframework...DefaultAnnotationHandlerMapping">
    <property name="order" value="0" />
</bean>

<bean class=
    "org.springframework...ControllerClassNameHandlerMapping">
    <property name="order" value="1" />
</bean>
```

Handler Mapping
Chain

Default Infrastructure Beans



- Defaults exist for all infrastructure beans
 - org/springframework/web/servlet/
DispatcherServlet.properties
- Example
 - HandlerAdapter
 - AnnotationMethodHandlerAdapter
 - HttpRequestHandlerAdapter
 - SimpleControllerHandlerAdapter

Note: Configuring infrastructure beans explicitly in effect
cancels all defaults for that bean type!

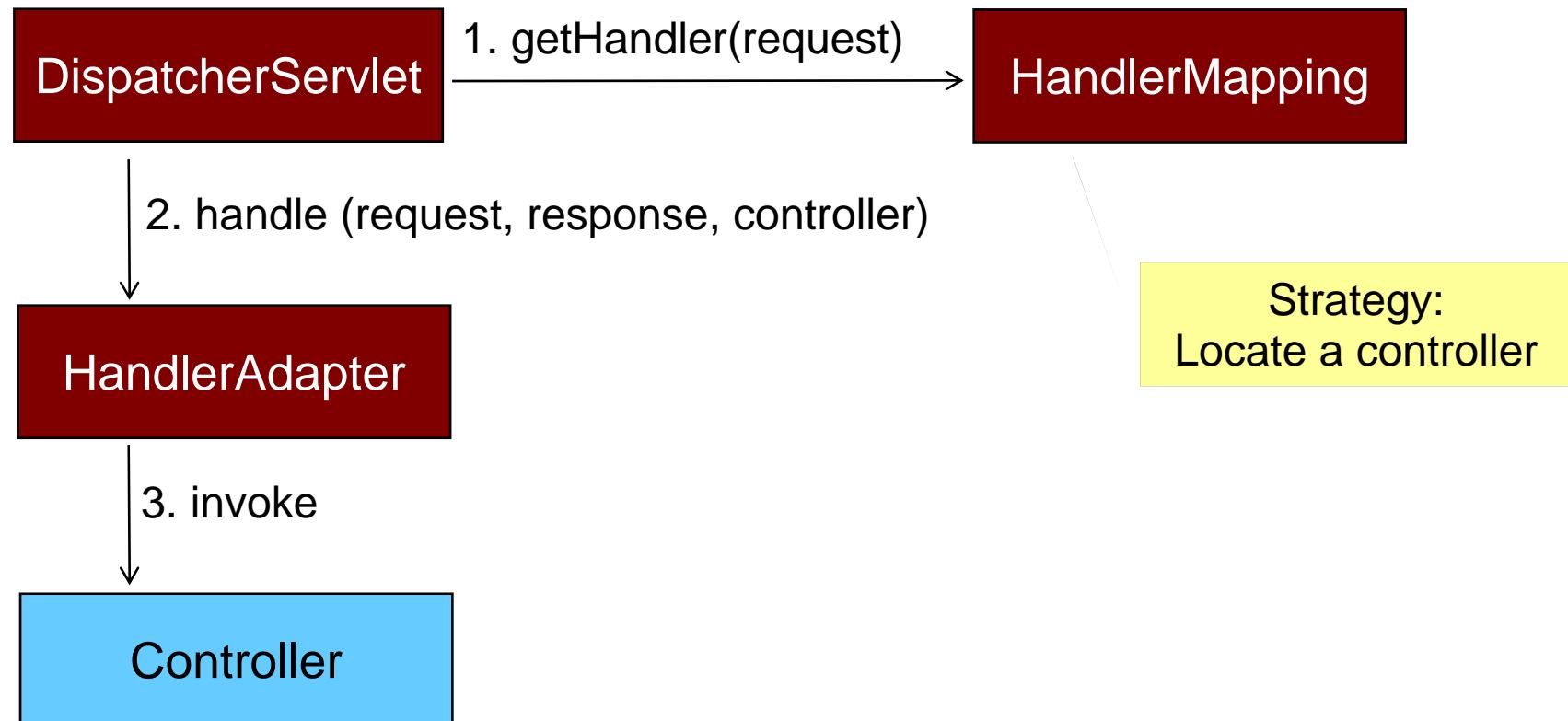


Road Map

- Spring MVC Infrastructure Beans
- URL Mappings
- Handler Interceptors
- Handler Adapters
- Exception Resolvers
- Message Source



Locating A Controller



DefaultAnnotationHandlerMapping

- An all annotations approach

```
@Controller  
@RequestMapping( "/accounts" )  
public class AccountsController {  
  
    @RequestMapping(method=RequestMethod.GET)  
    public String list(Model model) { ... }  
}
```

Class + method-relative
 @RequestMapping

.../accounts

```
@Controller  
public class AccountsController {  
  
    @RequestMapping( "/accounts" )  
    public String list(Model model) { ... }  
}
```

Absolute method-level
 @RequestMapping

.../accounts



RequestMappingHandlerMapping

- HandlerMapping strategy based on the new HandlerMethod abstraction
 - Allows RequestMappings for same URL to be in different controllers
 - Pluggable support for any argument/return type in handler methods
- Intended as a direct replacement of DefaultAnnotationHandlerMapping
 - Currently both supported as of Spring 3.1
 - Configured as THE HandlerMapping strategy for @RequestMapping with <mvc:annotation-driven>

New in Spring 3.1

ControllerClassName HandlerMapping

- Map controllers by class name convention
 - Add method-relative @RequestMapping

```
@Controller
public class AccountsController {
    ...
    @RequestMapping(method = RequestMethod.GET)
    public void list(Model model) {
        ...
    }
    ...
    @RequestMapping(method = RequestMethod.GET)
    public void show(@RequestParam ("id") long
accountNumber) {
        ...
    }
}
```

Controller-level mappings:
`/accounts, /accounts/*`

`.../accounts/list`

Method-relative
@RequestMapping

`.../accounts/show`

SimpleUrlHandlerMapping



- Declarative style URL-to-controller mappings

```
<bean class=".SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>
            /welcome=welcomeController
            /accounts/**=accountsController
        </value>
    </property>
</bean>
```

Controller-level
mappings

```
@Controller
public class AccountsController {
    @RequestMapping(method=RequestMethod.GET)
    public String list(Model model) { ... }
}
```

Method-relative
@RequestMapping

.../accounts/list

Spring MVC Log At Startup



```
DefaultAnnotationHandlerMapping D Looking for URL mappings in application
context: ...

DefaultAnnotationHandlerMapping D Rejected bean name 'flowMappings': ...
DefaultAnnotationHandlerMapping D Rejected bean name 'accountManager': ...
DefaultAnnotationHandlerMapping D Rejected bean name 'messageSource': ...

ControllerClassNameHandlerMapping D Looking for URL mappings in application
context: ...

ControllerClassNameHandlerMapping D Mapped URL path [/accounts] onto handler
[rewardsonline.accounts.AccountsController@db9199]
ControllerClassNameHandlerMapping D Mapped URL path [/accounts/*] onto handler
[rewardsonline.accounts.AccountsController@db9199]
```



Road Map

- Spring MVC Infrastructure Beans
- URL Mappings
- Handler Interceptors
- Handler Adapters
- Exception Resolvers
- Message Source



Handler Interceptors

- Extremely useful for applying functionality that is common to many controllers



Many useful cases

- Add common model attributes (menus, preferences)
- Set response headers
- Audit requests
- Measure performance (controller vs. rendering time)

Handler Interceptor Methods

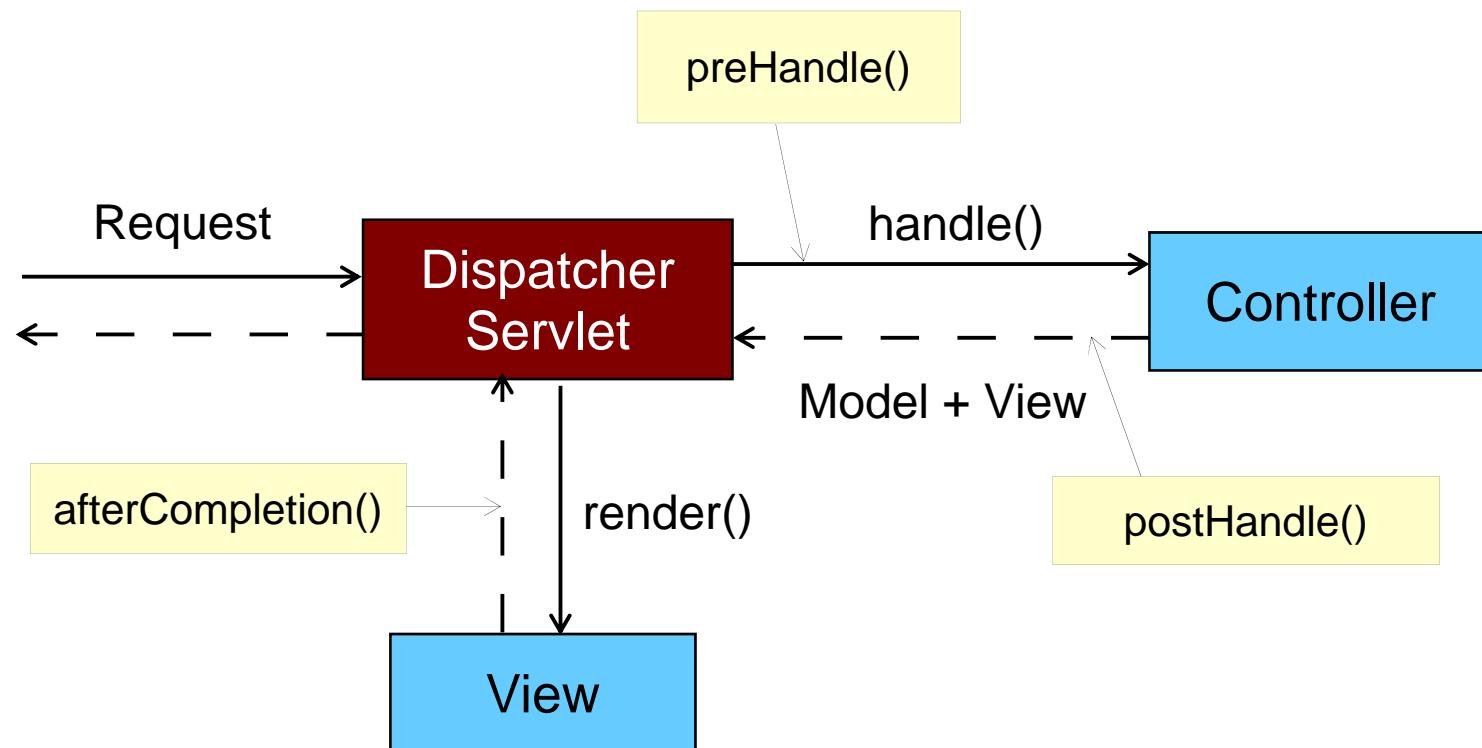


```
public interface HandlerInterceptor {  
  
    boolean preHandle(HttpServletRequest request, HttpServletResponse  
        response, Object handler) throws Exception;  
  
    void postHandle(HttpServletRequest request, HttpServletResponse  
        response, Object handler, ModelAndView modelAndView)  
        throws Exception;  
  
    void afterCompletion(HttpServletRequest request,  
        HttpServletResponse response, Object handler, Exception ex)  
        throws Exception;  
}
```

Before & After controller invocation

After view rendering

Handler Interceptor Methods



Configuring Interceptors

- Request interceptors are configured on the level of the HandlerMapping

```
<bean class="org.springframework.web...
                  DefaultAnnotationHandlerMapping">
    <property name="interceptors">
        <list>
            <bean class="rewardsonline.AuditInterceptor"/>
            <bean class="rewardsonline.PerformanceInterceptor"/>
        </list>
    </property>
</bean>
```



Simplifying interceptor configuration

- Interceptors can be specified using the mvc namespace and <mvc:interceptors> tag
- Interceptors are applied to all handler mappings

```
<mvc:interceptors>
    <bean class="rewardsonline.AuditInterceptor" />
    <bean class="rewardsonline.PerformanceInterceptor" />
</mvc:interceptors>
```

- To apply interceptors to a subset of handlers, use the “mapping” element

```
<mvc:interceptors>
    <mapping path="/secure/*" />
    <bean class="rewardsonline.SecurityInterceptor" />
</mvc:interceptors>
```

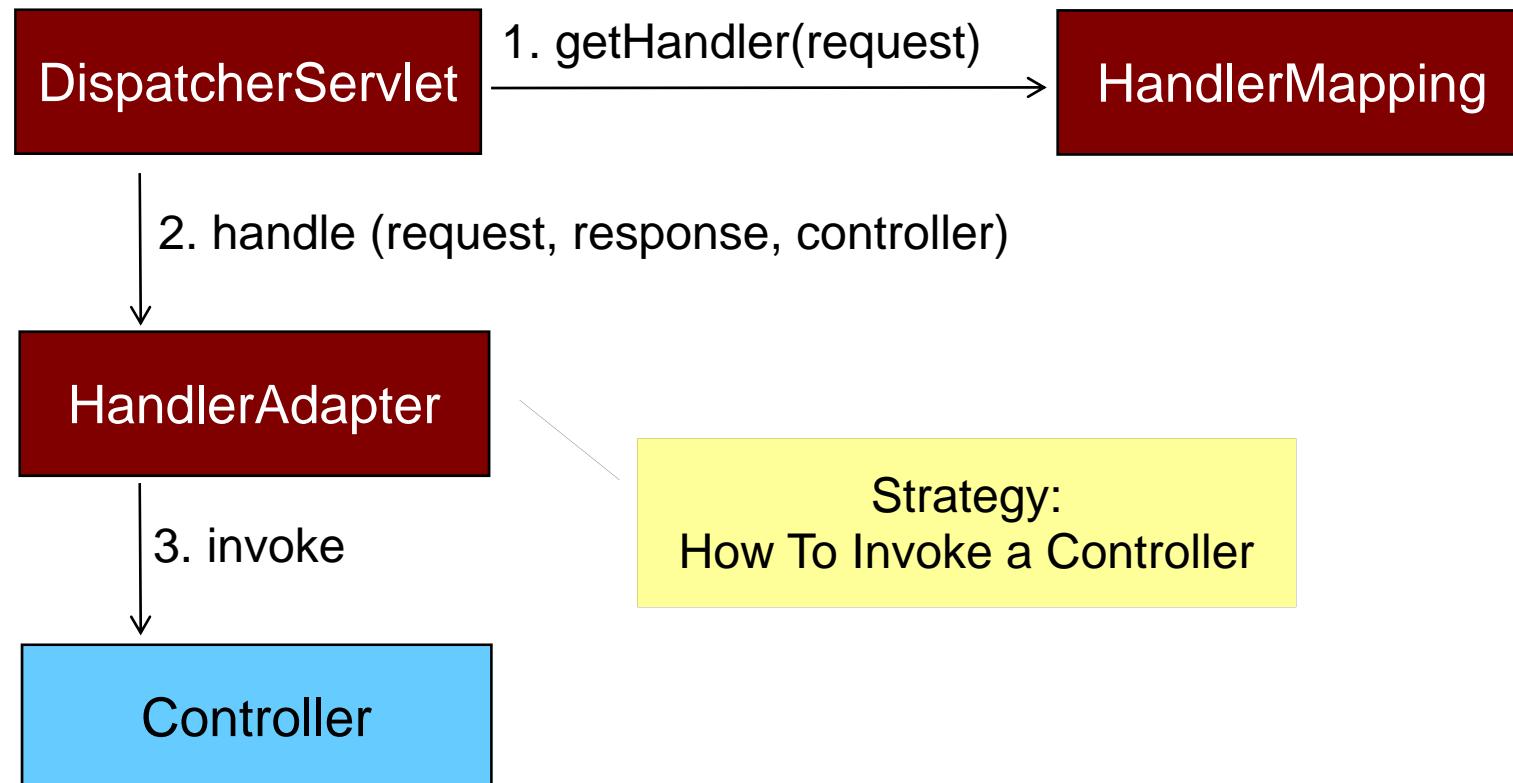


Road Map

- Spring MVC Infrastructure Beans
- URL Mappings
- Handler Interceptors
- Handler Adapters
- Exception Resolvers
- Message Source



Controller Invocation



Adapting Requests

- DispatcherServlet invokes controllers through a HandlerAdapter
- Many types of controllers
 - @Controller
 - Web Flow FlowExecutor
 - Adobe BlazeDS MessageBroker
- Allows new types of controllers to be added
 - Simply register a new adapter



AnnotationMethod HandlerAdapter



- Adapts calls to @RequestMapping methods
- Enables flexible method signatures
 - Introspects required input arguments
 - Interprets output values
- Configured by default as defined in DispatcherServlet.properties



RequestMethod HandlerAdapter



- Adapts calls to @RequestMapping methods
 - Part of the new HandlerMethod abstraction
- Enables even more flexible method signatures
 - Custom argument and return types supported
- Other enhancements
 - PathVariables automatically added to the Model
 - Supports parameterized URI template on redirect strings
 - Addition of consumes/produces argument to RequestMapping (better REST support)
 - Now the default with <mvc:annotation-driven>

New in Spring 3.1

Simplifying configuration with <mvc:annotation-driven>

- To simplify MVC configuration, use the <mvc:annotation-driven> tag
- Many beans automatically configured by this tag
 - RequestMappingHandlerMapping
 - RequestMappingHandlerAdapter
 - Validation, formatting, and type conversion (shown later)
- Infrastructure beans can still be chained
 - In *addition* to defaults

Different as of
Spring 3.1

Was DefaultAnnotationHandlerMapping
and AnnotationMethodHandlerAdapter





Simplifying configuration with @EnableWebMvc

- JavaConfig equivalent to <mvc:annotation-driven>

```
@EnableWebMvc  
  
@Configuration  
  
public class MyWebConfig extends WebMvcConfigurerAdapter {  
  
    @Override  
  
    public void configureResourceHandling  
        (ResourceConfigurer configurer) {  
  
        configurer.addPathMapping("/resources/**")  
            .addResourceLocation("classpath:/META-INF/")  
            .addResourceLocation("/resources/");  
  
    }  
    // Can override more WebMvcConfigurerAdapter methods ...  
}
```

New in Spring 3.1



Road Map

- Spring MVC Infrastructure Beans
- URL Mappings
- Handler Interceptors
- Handler Adapters
- Exception Resolvers
- Message Source



Controller Exceptions

- Controller execution can throw exceptions
- HandlerExceptionResolver
 - A Spring MVC strategy for controller exceptions
- Prepares a model and selects an error view
- Multiple handlers are supported
 - Chain using their `order` property
 - **Defaults:** `DefaultHandlerExceptionResolver` and `AbstractHandlerExceptionResolver`



SimpleMappingExceptionResolver



- A HandlerExceptionResolver implementation
- Maps exception class names to view names
- Adds “exception” model attribute
- Logs a message

Configuring Exception Resolution



```
<bean class="org.springframework.web...
                     SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <map>
            <entry key="DataAccessException"
                  value="databaseError" />
            <entry key="InvalidCreditCardException"
                  value="creditCardError" />
        </map>
    </property>
    <property name="defaultStatusCode" value="500"/>
    <property name="defaultErrorView" value="error"/>
</bean>
```

CustomExceptionResolver

```
public class CustomExceptionResolver
    extends SimpleMappingExceptionResolver {
protected String buildLogMessage(Exception e,
                                  HttpServletRequest req) {
    return "Custom log message...";
}

protected ModelAndView getModelAndView
    (String viewName, Exception e) {
    ModelAndView mav =
        super.getModelAndView(viewName, e);
    // Add model attributes for the error view ...
    return mav;
}
}
```



@ExceptionHandler

- Allows tighter control of exception handling within a single controller
- Annotated methods called automatically when controller methods throw exception.
 - Only applicable to current controller
- Method signatures are flexible
 - HttpServletRequest, HttpSession, etc can be specified
 - Conventions can be applied



Using @ExceptionHandler

```
@Controller  
public class AccountsController {  
  
    // Other controller methods skipped  
  
    @ExceptionHandler  
    public String handleException(DataAccessException ex) {  
        return "databaseError";  
    }  
}
```

Basic ExceptionHandler

```
@Controller  
public class AccountsController {  
    @ExceptionHandler  
    public void databaseError(DataAccessException ex) {  
    }  
}
```

ExceptionHandler with conventions



Road Map

- Spring MVC Infrastructure Beans
- URL Mappings
- Handler Interceptors
- Handler Adapters
- Exception Resolvers
- Message Source





Message Resolution

- Views often need access to properties
 - Externalized error codes
 - Internationalization
- Spring supports the standard Java resource bundle mechanism
- Spring MVC makes resource bundle properties accessible within views



Configuring a MessageSource

This bean id discovered by name (not by type) as there is only one MessageSource per application context

```
<bean id="messageSource"
      class="org.springframework.context... ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>/WEB-INF/messages/account</value>
      </list>
    </property>
</bean>
```

/WEB-INF/messages
account.properties
account_es.properties

Account.name=Account
Account.number=Number

...

Reloadable Message Source

- Will reload after cache-seconds
 - Never reload = -1 (default)
 - Always reload = 0 (never in production!)

```
<bean id="messageSource" class="org.springframework  
    .context.support.ReloadableResourceBundleMessageSource">  
    <property name="basenames" > ... </property>  
    <property name="cacheSeconds" value="${msgReloadPeriod}">  
</bean>  
<context:property-placeholder  
    location="/WEB-INF/config.properties" />
```

config.properties

msgReloadPeriod=60

*Don't use classpath: resource
– some class-loaders cache*

Resolving Messages

- In JSTL format tags

```
<%@ taglib prefix="fmt"  
        uri="http://java.sun.com/jsp/jstl/fmt" %>  
<fmt:message key="Account.name" />  
<fmt:message key="Account.number" />
```

- Also integrated with the Spring form tag library





Summary

After completing this lesson, you should have learnt:

- That much of Spring MVC's flexibility lies in its infrastructure beans
- To configure one or multiple infrastructure beans and order them by priority
- To explore the properties available on infrastructure beans
- To create your own implementations if needed





Managing Layouts in Spring MVC



Overview

After completing this lesson, you should be able to:

- Describe the Page Layout And Structure
- Create Re-Usable Templates With Tiles
- Describe Tiles Definitions
- Configure Tiles in Spring MVC



Road Map

- Page Layout And Structure
- Creating Re-Usable Templates With Tiles
- Tiles Definitions
- Configuring Tiles in Spring MVC



Common Page Elements

- Application pages have common elements
 - header/footer
 - navigation
 - main content
- Common elements are typically arranged the same way on every page





Include Statements

- Include statements affords a degree of re-usability
 - header.jsp, footer.jsp, etc.
- Although...
 - includes must be repeated everywhere
 - there is no common, re-usable page structure



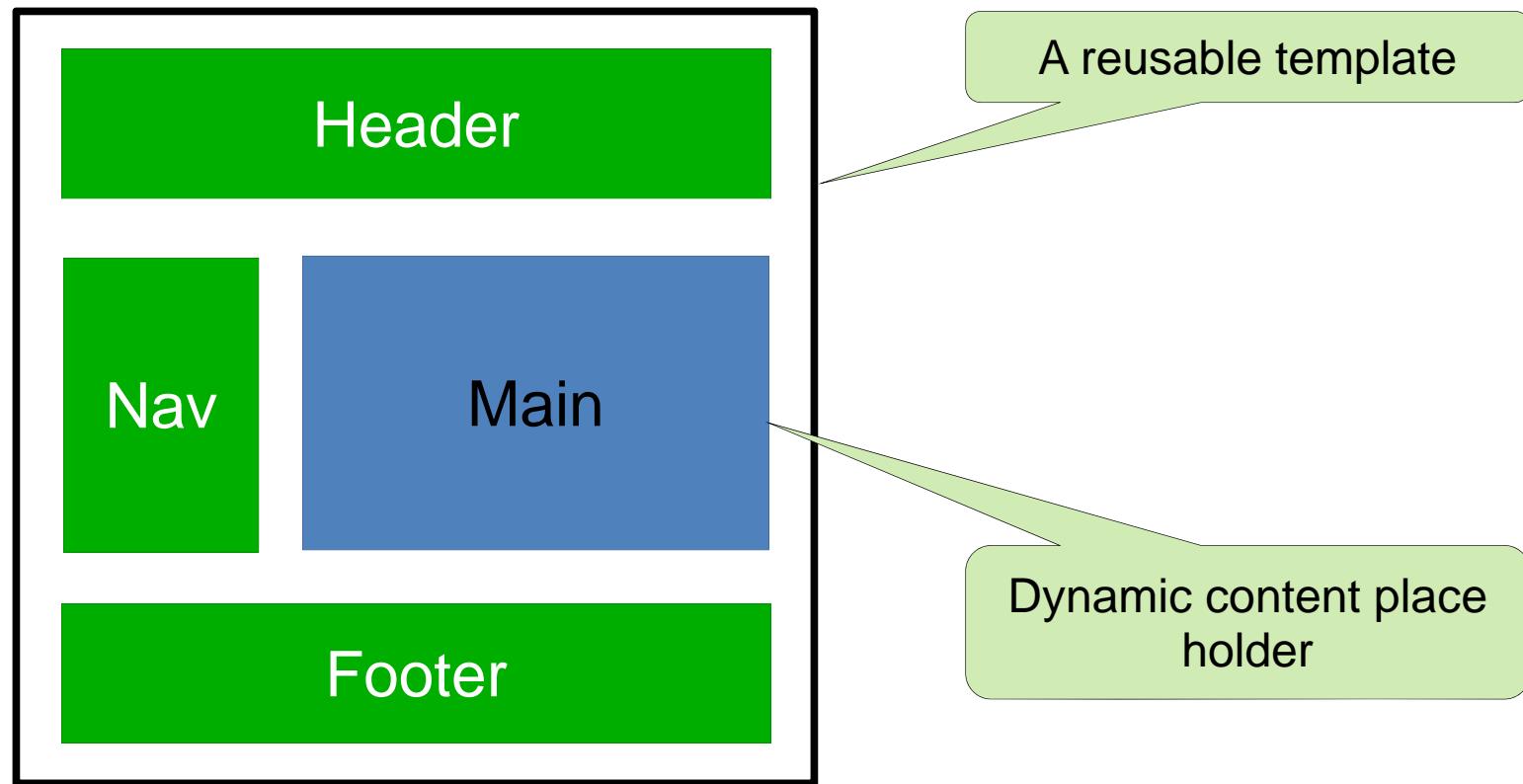
The Composite View Pattern



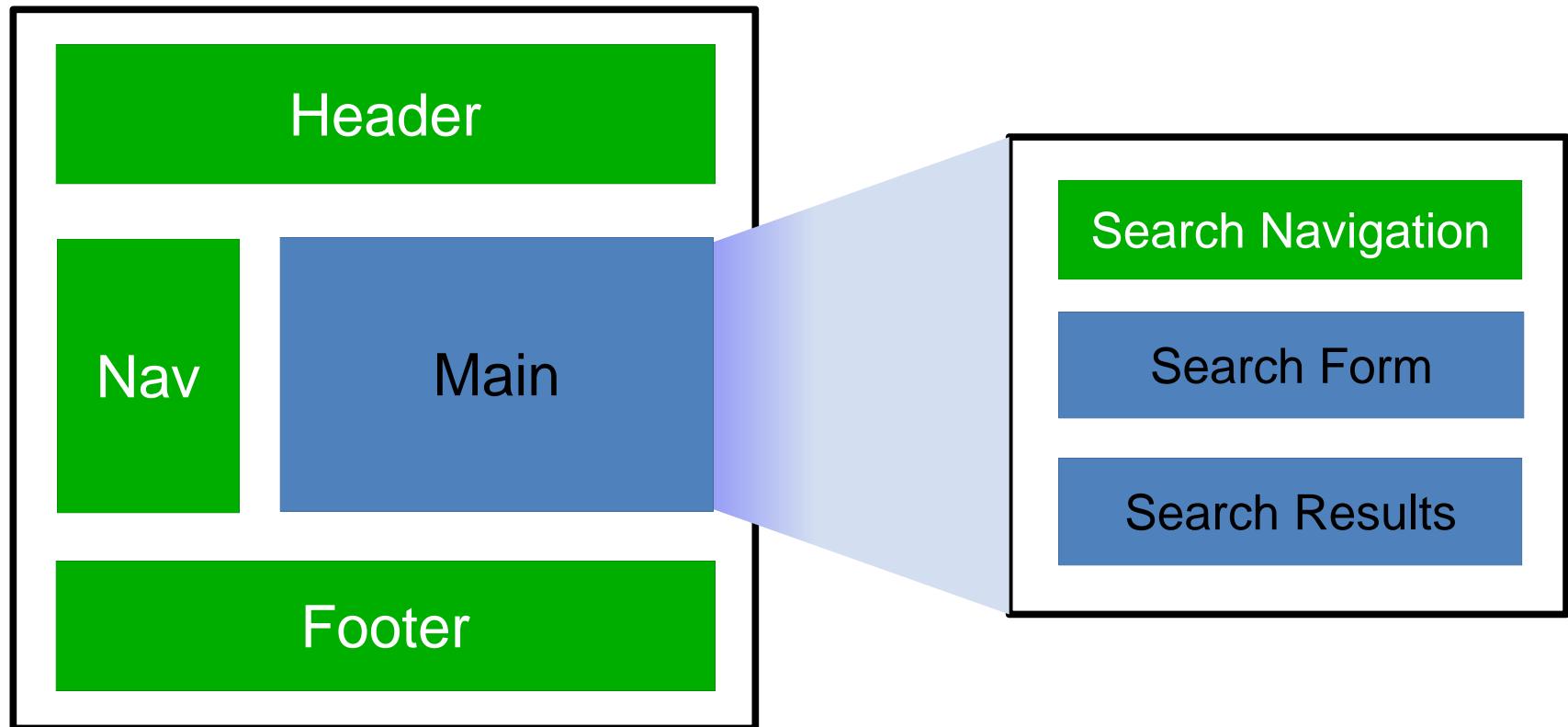
- Create a template that holds the common elements of a page
- Leave placeholders where dynamic content is needed
- Re-use the template and supply what it requires
 - sub-views
 - simple string values



Composite View Example



Nested View Composition





Road Map

- Page Layout And Structure
- Creating Re-Usable Templates With Tiles
- Tiles Definitions
- Configuring Tiles in Spring MVC





Apache Tiles

- A templating framework
- An independent Apache project
 - Formerly part of Struts
- Implements the Composite View pattern
- Allows creating re-usable templates and defining fragments (tiles) for composing views



A Tiles Template Example

```
<%@ taglib prefix="tiles"
           uri="http://tiles.apache.org/tags-tiles" %>
```

```
<head>
    <title>
        <tiles:insertAttribute name="title" />
    </title>
</head>
<body>
    <div id="header"> ... </div>
    <div id="body">
        <tiles:insertAttribute name="main" />
    </div>
    <div id="footer"> ... </div>
</body>
```

Placeholder for dynamic content

/WEB-INF/layouts/standard.jsp

Accessing A Tiles Template From a JSP Page



```
<%@ taglib prefix="tiles"  
        uri="http://tiles.apache.org/tags-tiles" %>
```

```
<tiles:insertTemplate template  
                      ="/WEB-INF/layouts/standard.jsp" >  
  
<tiles:putAttribute  
      name="title" value="Welcome To RewardsOnline" />  
  
<tiles:putAttribute name="main">  
  <h1>  
    Welcome to RewardsOnline  
  </h1>  
</tiles:putAttribute>  
</tiles:insertTemplate>
```

Specify the dynamic
content via attributes

/WEB-INF/welcome.jsp



Road Map

- Page Layout And Structure
- Creating Re-Usable Templates With Tiles
- **Tiles Definitions**
- Configuring Tiles in Spring MVC





Creating Tiles Definitions

- Tiles allows creating external XML configuration to define the structure of a page
- A Tiles configuration file contains one or more Tiles definitions
- Tiles definitions are reusable fragments consisting of a template and attributes



A Base Tiles Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software
Foundation//DTD Tiles Configuration 2.0//EN"
"http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
```

```
<tiles-definitions>
    <definition name="standardLayout"
        template="/WEB-INF/layouts/standard.jsp">
    </definition>
</tiles-definitions>
```

A reference to a template.

Attributes left unspecified.

Extending A Tiles Definition

```
<tiles-definitions>
    <definition name="standardLayout"
        template="/WEB-INF/layouts/standard.jsp" />
    <definition name="welcome" extends="standardLayout">
        <put-attribute name="title"
            value="Welcome To RewardsOnline" />
        <put-attribute name="main"
            value="/WEB-INF/welcome.jsp" />
    </definition>
</tiles-definitions>
```

/WEB-INF/tiles.xml

Define all attributes required
by the base Tiles definition

Tiles Attributes

- The placeholders for dynamic data
- An attribute value can be
 - a simple string value
 - a template (if it starts with '/')
 - another tiles definition

```
<tiles-definitions>
    <definition name="standardLayout" template="..." />
    <definition name="mainBody" template="..." />
    <definition name="welcome" extends="standardLayout">
        <put-attribute name="main" value="mainBody" />
    </definition>
</tiles-definitions>
```

/WEB-INF/tiles.xml

Tiles Attributes (2)

- Two approaches to insert attributes in JSP
 - `<tiles:insertAttribute name="navigationTab" />` (standard approach)
 - `<tiles:importAttribute name="navigationTab" />` (adds attribute to model)

```
<tiles:importAttribute name="navigationTab" />
<c:if test="${navigationTab eq 'home'}">
    ...
</c:if>
```

```
<definition name="accounts/list" extends="standardLayout">
    <put-attribute name="title" value="accounts.list.title" />
    <put-attribute name="main" value="/WEB-INF/accounts/list.jsp" />
    <put-attribute name="navigationTab" value="accounts" />
<definition>
```

/WEB-INF/tiles.xml



Road Map

- Page Layout And Structure
- Creating Re-Usable Templates With Tiles
- **Tiles Definitions**
- Configuring Tiles in Spring MVC



Using Tiles In Spring MVC



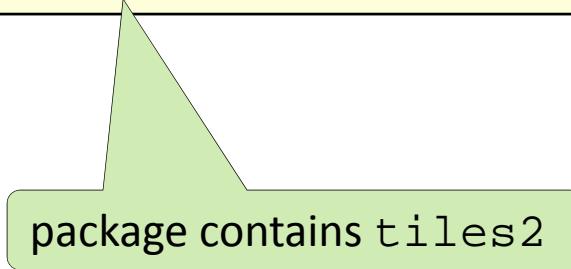
- Spring MVC provides support for configuring and using Tiles
- TilesConfigurer helps to bootstrap Tiles with a set of Tiles configuration files
- TilesView interprets logical view names as Tiles definition names



Bootstrapping Tiles (1)

- Adding the `TilesConfigurer` bean provided by Spring MVC configures the `TilesContainer`

```
<bean id="tilesConfigurer" class="org.springframework.web.  
servlet.view.tiles2.TilesConfigurer"/>
```



package contains tiles2

Bootstrapping Tiles (2)

- Provide one or more Tiles configuration files to the TilesConfigurer

```
<bean id="tilesConfigurer" class="org.springframework.web  
        .servlet.view.tiles2.TilesConfigurer">  
    <property name="definitions">  
        <list>  
            <value>/WEB-INF/tiles.xml</value>  
            <value>/WEB-INF/accounts/tiles.xml</value>  
        </list>  
    </property>  
</bean>
```

Tiles View Resolution

- Switch from InternalResourceViewResolver

```
<bean class="org.springframework.web.servlet.view  
          .InternalResourceViewResolver">  
    <property name="prefix" value="/WEB-INF/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

- To a TilesViewResolver

```
<bean class="org.springframework.web.servlet.view  
          .tiles2.TilesViewResolver" />
```

Summary

After completing this lesson, you should have learnt that:

- All web sites have a need for a reusable layout
- The composite view pattern provides a way to manage site layout and structure
- Apache Tiles is a framework for composing arbitrary views from a set of reusable fragments.





LAB

Using Tiles Layouts In Spring MVC



Using Views In Spring MVC



Overview

After completing this lesson, you should be able to:

- Describe Views and View Resolvers
- Describe how to setting up a `ViewResolver` Chain
- Describe Alternating Views
- Describe JSON Views



Road Map

- Views and View Resolvers
- Setting Up A View Resolver Chain
- Alternating Views
- JSON Views

URL-Based Views



- The logical view name matches to a file
 - JSP page
 - FreeMarker template
 - XSLT stylesheet
 - etc.
- All views are stored under /WEB-INF
 - hidden from direct browser access
 - rendering requires a model
- The URL may not be a file (e.g. Tiles definition)





URL-Based View Example

- JstlView
 - exposes the model attributes as request attributes
 - adds attributes for JSTL format/message tags
 - forwards to JSP page

```
JstlView view = new JstlView  
        ("WEB-INF/rewards/show.jsp");  
view.render(model, request, response);
```

UrlBasedViewResolver



- Interprets a view name as a URL
- Many subclasses
 - InternalResourceViewResolver (JSP/Servlets)
 - FreeMarkerViewResolver
 - XsltViewResolver
 - etc.
- Supports “redirect:” prefix



Content-Generating Views



- Extends from a base class
 - AbstractExcelView, AbstractPdfView, Etc
- Creates view content using an API
 - POI, iText
- The base class writes generated content to the response stream



Excel View Example

```
AccountsExcelView view = new AccountsExcelView();
view.render(model, request, response);
```

```
public class AccountsExcelView extends
                           AbstractExcelView {
    public void buildExcelDocument(Map model,
                                   HSSFWorkbook workbook,
                                   HttpServletRequest request,
                                   HttpServletResponse response) {

        Account account = (List<Account>)
                           model.get("accounts").get(0);
        HSSFSheet sheet = workbook.getSheetAt(0);
        ...
    }
}
```

BeanNameViewResolver



- Interprets a view name as a bean name

```
<bean class="org.springframework.web.servlet.view.  
BeanNameViewResolver"/>  
  
<bean name="accounts/list.xls"  
      class="rewardsonline.accounts.AccountsExcelView"/>
```

View bean name

```
@RequestMapping( "/accounts" )  
public String list(Model model) {  
    ...  
    return "accounts/list.xls";  
}
```

View name selected by
controller method

XmlViewResolver



- Picks up view beans from a given location file
 - Works just like BeanNameViewResolver
 - But matches only views defined in *its* bean file
 - Reduces bean-file clutter
 - Keep view beans separate

```
<bean class=".web.servlet.view.XmlViewResolver">
    <property name="location">
        <value>/WEB-INF/spring/views.xml</value>
    </property>
</bean>
```

Standard Spring Bean file
containing *only* view beans



Road Map

- Views and View Resolvers
- Setting Up A View Resolver Chain
- Alternating Views
- JSON Views



ViewResolver Chain



- The DispatcherServlet discovers ViewResolver beans by type
- Multiple ViewResolver beans are possible
 - Each is given a “chance” to match
 - You can specify the order
- The first resolver to return a View “wins”



ViewResolver Chain Example



```
<bean class=".web.servlet.view.BeanNameViewResolver">
    <property name="order" value="1">
</bean>

<bean
    class=".web.servlet.view.InternalResourceViewResolver">
    <property name="order" value="2">
    <property name="prefix" value="/WEB-INF/">
    <property name="suffix" value=".jsp">
</bean>
```

ViewResolver Chain Order



- All `viewResolver` beans implement `Ordered`
- Some `UrlBasedViewResolvers` can be anywhere in the chain
 - Depends on the view type served
 - Tiles, Velocity, Freemarker views check for files
 - return null if they don't exist
- Others must be last
 - JSTL/JSP, XSLT, JSON always forward rather than returning null
 - These resolvers *must* be last
 - Or sub-class to do a resource check





Road Map

- Views and View Resolvers
- Setting Up A View Resolver Chain
- Alternating Views
- JSON Views



Content Type Negotiation



- Previous techniques work well when each resource is associated with one view type
- Clients may request different content types for the same resource
 - Via file extension, request header, request parameter, etc.
- The process of determining which type to render is known as **Content Type Negotiation**





Content Type Negotiation Example

- Example requirement
 - Provide Excel link on Account Search Results page
- Sample requests (if using extensions)
 - HTML:
 - GET /accounts/list
 - GET /accounts/list.htm
 - Excel:
 - GET /accounts/list.xls
- If using the **format** parameter
 - GET /accounts/list?format=text/html
 - GET /accounts/list?format=application/vnd.ms-excel

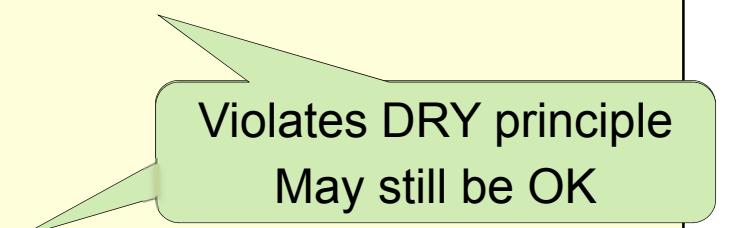


Option #1: Separate Methods

- Use a different controller method for each view type

```
@RequestMapping("/accounts.htm")
public String listHtml(HttpServletRequest rq, Model model) {
    model.addAttribute(accountManager.findAllAccounts());
    return "accounts/list";
}

@RequestMapping("/accounts.xls")
public String listXcel(HttpServletRequest rq, Model model) {
    model.addAttribute(accountManager.findAllAccounts());
    return "accounts/list.xls";
}
```



Violates DRY principle
May still be OK



Option #2: Controller Logic

- Controller detects URL extension (or request header) and selects view name

Extension ignored.
Equivalent to: /accounts.*

```
@RequestMapping("/accounts")
public String list(HttpServletRequest rq, Model model) {
    model.addAttribute(accountManager.findAllAccounts());
    // If Request URL ends with ".xls"
    return "accounts/list.xls";
    // Else
    return "accounts/list";
}
```

Works, but if-else logic will likely be replicated to other controllers!

Option #3: Special View Resolver



- ContentNegotiatingViewResolver (CNVR)
 - Introduced in Spring 3.0
- Does not do view resolution itself
 - Detects requested content type
 - Delegates to other view resolvers
 - Can use extensions, request parameters, or Accepts header to determine content type
- Each view class is associated with a content type
 - Many views already define content types

AbstractPdfView → application/pdf

AbstractExcelView → application/vnd.ms-excel

JstlView → text/html

ContentNegotiatingViewResolver Configuration



Property	Details
order	<ul style="list-style-type: none">• CNVR should be first in any resolver chain• Default is Ordered.HIGHEST_PRECEDENCE
defaultContentType	<ul style="list-style-type: none">• The type to render if a match is not found• No default
viewResolvers	<ul style="list-style-type: none">• The view resolvers to delegate to.• If undefined, all configured resolvers are detected
favorParameter	<ul style="list-style-type: none">• Looks for format parameter: http://..?format=pdf• Default is false
favorPathExtension	<ul style="list-style-type: none">• Determine content type from file-type in URL• Example: http://...somepage.pdf• Default is true



ContentNegotiatingViewResolver Example



```
<bean class="o.s.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="mediaTypes">
        <map>
            <entry key="html" value="text/html"/>
            <entry key="xls" value="application/vnd.ms-excel"/>
        </map>
    </property>
    <property name="viewResolvers">
        <list>
            <bean class="o.s.web.servlet.view.BeanNameViewResolver"/>
            <bean
                class="o.s.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix" value="/WEB-INF/"/>
                <property name="suffix" value=".jsp">
            </bean>
        </list>
    </property>
    <property name="defaultContentType" value="text/html"/>
</bean>
```

View classes declare the matching content type

Resolvers can be detected implicitly as well

ContentNegotiatingViewResolver Example Continued



```
@RequestMapping( "/accounts" )
public String list(Model model) {
    model.addAttribute
        (accountManager.findAllAccounts() );
    return "accounts/list";
}
```

```
<bean name="accounts/list"
      class="rewardsonline.accounts.AccountsExcelView" />
```

No need to specify suffix
(CNVR will do the match without it!)

ContentNegotiatingViewResolver and Accepts Header



- The ContentNegotiatingViewResolver can use the Accept header
 - This is most useful for REST web services and similar automation scenarios
 - Browsers generally send text/html for all requests but actual header varies by browser type
 - Do **not** use this for web applications
- Best to set ignoreAcceptHeader to true when configuring CNVR in web applications.





Simplifying CNVR Configuration

- The ContentNegotiatingViewResolver configuration can be significantly simplified.
- View resolvers can be configured separately from the CNVR
 - No need for `viewResolvers` property
- Extensions can be mapped to MIME types using the Java Activation Framework
 - No need for `mediaTypes` property
 - Presumes standard extensions (.pdf, .xls, etc.)
 - `activation.jar` must be on the classpath



ContentNegotiatingViewResolver Simplified Example



```
<bean  
    class="o.s.web.servlet.view.ContentNegotiatingViewResolver">  
        <property name="defaultContentType" value="text/html"/>  
</bean>  
  
<bean class="o.s.web.servlet.view.XmlViewResolver"  
      p:location="/WEB-INF/spring/pdf-views.xml" />  
  
<bean class="o.s.web.servlet.view.XmlViewResolver"  
      p:location="/WEB-INF/spring/excel-views.xml" />  
  
<bean class="o.s.web.servlet.view.InternalResourceViewResolver"  
      p:prefix="/WEB-INF/" p:suffix=".jsp" />
```

Types are determined automatically

Note: **p:x="y"** is short for

```
<property name="x" value="y" />
```

View Resolvers are defined externally



Road Map

- Views and View Resolvers
- Setting Up A View Resolver Chain
- Alternating Views
- JSON Views





Generating JSON

- Uses the `MappingJacksonJsonView`
 - Converts contents of Model to JSON
 - Ignores MVC internal classes such as form-binders
- Objects in model may need annotating
 - Tells Jackson knows how to convert them
 - Similar to JPA and JAXB annotations
- No `ViewResolver` supplied
 - Simple to write one



Need a JSON View Resolver

- Suggested implementation
 - Always returns a `MappingJacksonJsonView`
 - Only works alongside the CNVR

```
public class JsonViewResolver implements ViewResolver {  
    public JsonViewResolver() { }  
  
    @Override  
    public View resolveViewName(String viewName, Locale locale){  
        return new MappingJacksonJsonView();  
    }  
}
```



Generating JSON

- Request must contain Accept header
 - Accept: application/json

```
<bean  
    class="o.s.web.servlet.view.ContentNegotiatingViewResolver">  
    <property name="defaultContentType" value="text/html"/>  
</bean>  
  
<bean class="springweb.example.JsonViewResolver" />  
<bean class="o.s.web.servlet.view.InternalResourceViewResolver"  
      p:prefix="/WEB-INF/"  p:suffix=".jsp" />
```



REST and Spring MVC

- The `MappingJacksonJsonView` allows JSON to be returned
 - But there are no views for automatically mapping XML
 - You could implement one
 - Spring-OXM provides Object-XML marshallers
- Alternatively: `@ResponseBody` annotation to implement REST **without** views

[http://static.springsource.org
/spring/docs/3.1.x/reference/html/oxm.html](http://static.springsource.org/spring/docs/3.1.x/reference/html/oxm.html)

Summary

After completing this lesson, you should have learnt that:

- Spring MVC supports a diverse set of view types
- Chain of ViewResolvers can be used to support multiple view types in a single application
- The ContentNegotiatingViewResolver can be used to support multiple view types for the same resource





LAB

Using Spring MVC Views



Building Form Pages with Spring MVC



Overview

After completing this lesson, you should be able to:

- Describe what MVC Forms are
- Describe Form Rendering
- Describe Type Formatting
- Describe Data Binding
- Describe Spring MVC Form Validation
- Describe Spring MVC Form Object Management



Lesson Roadmap

- Overview
- Form Rendering
- Formatters
- Data Binding
- Validation
- Form Object Management



The Form Object

- Central to Spring MVC form processing
 - Form tags, data binding, validation, error processing

Reward Account for Dining

Dining Information

Credit Card :	<input type="text" value="1234123412341324"/>
Restaurant :	<input type="text" value="AppleBees"/>
Dining Amount :	<input type="text" value="\$100"/>
Dining Date :	<input type="text" value="5/22/2009"/>
One Click Reward :	<input type="checkbox"/>

Reward

```
public class DiningForm {  
    private String creditCardNumber;  
    private String merchantNumber;  
    private MonetaryAmount amount;  
    private SimpleDate date;  
}
```

Form Workflow

- Initial HTTP GET to present the form
 - Prepare the form object
- Subsequent HTTP POST to submit the form
 - Apply request values to the form object
 - Perform validation
 - Save changes
- POST-Redirect-GET on success
 - Redirect to the next page instead of rendering





Search Forms

- HTTP GET to present and submit the form
 - Submit has no “side effects”
- Redirect to detail page for direct hits
- Render the results in case of multiple hits



Exposing Domain Objects



- Domain objects can be used as form objects
- Potential security concern
 - Must set allowed fields for data binding purposes
- Puts a strain on OO design
 - Mandatory default constructor, getters & setters
- Web layer logic likely to creep in...



Web Layer Form Objects



- Form pages may differ from domain objects
 - Data aggregated from multiple domain objects
 - UI-specific needs
- Prefer separate web layer form objects
 - Model exactly what the screen needs
 - Encapsulate:
 - Web layer logic (reduces Controller bloat!)
 - Validation logic
 - Logic for copying to and from the domain object





Lesson Roadmap

- Overview
- Form Rendering
- Formatters
- Data Binding
- Validation
- Form Object Management



Displaying A Form

- Spring form tags bind the form object to the form

Reward Account for Dining

Dining Information

Credit Card :	<input type="text" value="1234123412341324"/>
Restaurant :	<input type="text" value="AppleBees"/>
Dining Amount :	<input type="text" value="\$100"/>
Dining Date :	<input type="text" value="5/22/2009"/>
One Click Reward :	<input type="checkbox"/>

Reward

```
public class DiningForm {  
    private String creditCardNumber;  
    private String merchantNumber;  
    private MonetaryAmount amount;  
    private SimpleDate date;  
}
```

Type conversion (object to string)





The Spring Form Tags

- Custom JSP tag library
- Makes it easier to build form pages
- Advantages
 - Populate HTML form fields with formatted values
 - Render field-specific error messages



Using Spring Form Tags

- Add the taglib directive

```
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>
```

- Use the Spring form tag

```
<form method="post" action="...">
```

Can use any attribute
from the model

```
<form:form method="post" action=""
            modelAttribute="diningForm">
```



Form Input Tag

- Plain HTML text input

```
<input type="text" name="amount" value="..." />
```

- Use the Spring form tag
 - Renders field populated with formatted value

```
<form:input path="amount" />
```

Property path (relative to
the Model-Attribute
specified on the form tag)

Form Select With Options

- Plain HTML select tag

```
<select name="pageSize">  
    <option value="5">5</option>  
    <option value="10" selected="selected">10</option>  
</select>
```

Static Selection

- Use the Spring form select tag

```
<form:select path="pageSize">  
    <form:option label="5" value="5"/>  
    <form:option label="10" value="10"/>  
<form:select>
```

Dynamic selection based on
the value of pageSize

Form Select Tag With Items

- Plain HTML select tag

```
<select name="merchantNumber">  
    ...  
<select/>
```

In JSP, loop over collection of Restaurants, Render options, Mark selected option

- Use the Spring form select tag

```
<form:select path="merchantNumber" items="${restaurants}"  
    itemLabel="name"  
    itemValue="number"/>
```

restaurants must have been added to model by Controller

```
public class Restaurant {  
    private String name;  
    private String number;  
}
```

Form Options Tag

- Render a list of options

One static option

```
<form:select path="merchantNumber">
    <form:option value="-->Please select</form:option>
    <form:options items="${restaurants}"
                  itemLabel="name" itemValue="number"/>
<form:select/>
```

Dynamic list of options

```
public class Restaurant {
    private String name;
    private String number;
}
```

More Form Custom Tags



- All tags have equivalent HTML form fields
 - checkbox, checkboxes
 - hidden
 - label
 - password
 - radiobutton, radiobuttons
 - textarea
- The <form:errors> tag
 - Displays error messages (to be discussed shortly)





Lesson Roadmap

- Overview
- Form Rendering
- **Formatters**
- Data Binding
- Validation
- Form Object Management



Formatting or Validation?

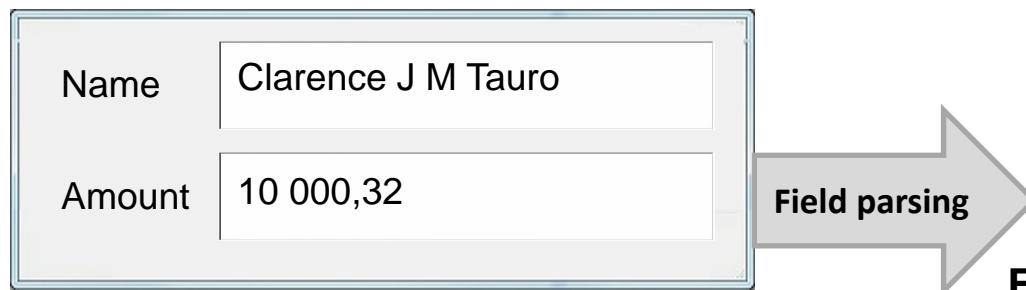
- Validation rules
 - “This String should contain between 7 and 10 characters”
 - “First name should not contain any digit”
 - “Contact date should be a date in the past”
- Formatting
 - `java.util.Date` → MM/dd/yyyy or dd/MM/yyyy
(depending on the locale)
 - `java.lang.Float` → 100 000,12 or 100,000.12
(depending on the locale)

The following slides discuss Formatters
(Validation rules will be discussed later)



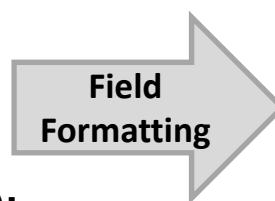
Formatters

- Formatters parse string data



```
Float amount = new Float(10000.32);
```

- Formatters can format bean data for display or editing



```
Float amount = new Float(5000.24);
```

Name	Clarence J M Tauro
Amount	5 000,24



3 ways to use Formatters

- Annotations: inside the bean class

```
public class Account {  
    @DateTimeFormat(iso=ISO.DATE)  
    private java.util.Date endDate;  
}
```

Used by all JSPs displaying info for this bean

- JSP tags (using the “fmt” tag library)

```
<fmt:formatDate value="${account.endDate}"  
pattern="MM/dd/yyyy" />
```

Each JSP can have its own formatting pattern

- Register custom Formatters

- Usually used for custom business beans (eg. SocialSecurityNumber, BankAccount...)

Formatting Annotations Example

```
public class Account {  
  
    private final String name;  
    private final String number;  
  
    @NumberFormat(style=Style.CURRENCY)  
    private final BigDecimal amount;  
  
    @NumberFormat(style=Style.NUMBER, pattern="#,###.###")  
    private final BigDecimal interestAmount;  
  
    @DateTimeFormat(iso=ISO.DATE)  
    private final Date endDate;  
}
```

Annotations can also be used on controller method arguments

Formatting inside a JSP

```
public class Account {  
    //...  
  
    private final BigDecimal interestAmount;  
    private final Date endDate;  
}
```

No more annotations
in that case

```
<fmt:formatNumber  
    value="${account.interestAmount}" type="percent" />  
  
<fmt:formatDate value="${account.endDate}"  
    pattern="MM/dd/yyyy" />
```

Annotations can also be used on controller method arguments

Default configuration

- The mvc namespace registers a set of formatters by default

```
<mvc:annotation-driven />
```

Registers formatters
for numbers and
Dates

- Register a conversion service to add your own

```
<mvc:annotation-driven  
conversion-service="conversionService" />
```

More details in the
next slides

Formatter for Joda Time is also registered by default
When Joda Time is in the classpath

Creating a custom formatter

- Format any Object to and from a String

```
public class SSNFormatter
    implements Formatter<SocialSecurityNumber> {

    public String print(SocialSecurityNumber ssn, Locale locale) {
        return ssn.getPrefix() + "-" + ssn.getSuffix(); }

    public SocialSecurityNumber parse(String text, Locale locale)
        throws ... {
        SocialSecurityNumber securityNumber
            = new SocialSecurityNumber();
        securityNumber.setPrefix(text.substring(0, 3));
        securityNumber.setSuffix(text.substring(4, 6));
        return securityNumber;
    }
}
```

Registering a custom formatter

- An explicit conversion service should be declared

```
<mvc:annotation-driven  
      conversion-service="conversionService" />  
  
<bean id="conversionService" class="org.springframework.  
      format.support.FormattingConversionServiceFactoryBean">  
  <property name="formatters">  
    <list>  
      <bean class="com.springsource.SSNFormatter" />  
    </list>  
  </property>  
</bean>
```

Bean XML

How about a short break?

- This is a long but important section
- Might be time for a quick break?

See you in a bit ...





Lesson Roadmap

- Overview
- Form Rendering
- Formatters
- Data Binding
- Validation
- Form Object Management



Submitting A Form

- Spring MVC binds the request to the form object

POST /rewards

creditCardNumber	1234123412341234
merchantNumber	1234567890
amount	\$100.00
date	2009-05-22

Reward Account for Dining

Dining Information

Credit Card :	<input type="text" value="1234123412341324"/>
Restaurant :	<input type="text" value="AppleBees"/>
Dining Amount :	<input type="text" value="\$100"/>
Dining Date :	<input type="text" value="5/22/2009"/>
One Click Reward :	<input type="checkbox"/>
Reward	

```
public class DiningForm {
    private String creditCardNumber;
    private String merchantNumber;
    private MonetaryAmount amount;
    private SimpleDate date;
}
```

Getters and setters
required, but omitted ...

Type conversion (string to object)



Data Binding Property Paths



- Request parameter names are EL expressions matching to arbitrary form object structure

```
POST /person/1
      name Leslie
      age 31
children["Amy"].name Amy
children["Amy"].age 2
      ...
addresses[0].street 1551 Ocean Ave
addresses[0].city Fairfield
      ...
```

```
public class Person {
    private String name;
    private int age;
    private Map<String, Person> children;
    private List<Address> addresses;
}
```

```
public class Address {
    private String street;
    private String city;
    ...
}
```

Data Binding On Submit

- Declare the form object as an input argument

```
@Controller  
public class RewardsController {  
  
    @RequestMapping(method=RequestMethod.POST)  
    public String update(DiningForm diningForm)  
    ...  
}
```

<form:form ... modelAttribute="diningForm">

Submitted form data copied in automatically

- Optionally annotate it with @ModelAttribute

```
@RequestMapping(method=RequestMethod.POST)  
public String update(@ModelAttribute("dining")  
                     DiningForm diningForm) { }  
  
modelAttribute="dining"
```

Control Data Binding Fields

- Allowed fields (white list)

Add this method
to your Controller

```
@InitBinder  
public void initBinder(WebDataBinder binder) {  
    binder.setAllowedFields("date", "amount");  
}
```

Recommended!

- Disallowed fields (black list)

```
@InitBinder  
public void initBinder(WebDataBinder binder) {  
    binder.setDisallowedFields("id", "*Id");  
}
```

Wild cards
may be used

Check Data Binding Errors

- Add a `BindingResult` input argument immediately after the form object

```
@RequestMapping(method=RequestMethod.PUT)
public String update(DiningForm diningForm,
                     BindingResult result) {
    if (result.hasErrors()) {
        return "rewards/edit";
    }
    // continue on
    // success ...
}
```

On error, return to form page

Display field-specific errors

```
<form:form
modelAttribute="diningForm">
    <form:input path="amount"/>
    <form:errors path="amount"/>
    ...
</form:form>
```

Customize “Type Mismatch” Data Binding Error Messages



- Add any of the following to your MessageSource

```
// Any “type mismatch” error  
typeMismatch=Incorrect value  
  
// Type mismatch errors for a named field  
typeMismatch.amount=Incorrect amount  
  
// Type mismatch errors for a named field on a named model  
attribute  
typeMismatch.diningForm.amount=Incorrect dining amount  
  
// Type mismatch errors for all fields of a specific type  
typeMismatch.common.money.MonetaryAmount=Incorrect  
monetary amount
```



Lesson Roadmap

- Overview
- Form Rendering
- Formatters
- Data Binding
- Validation
- Form Object Management





Validating Form Objects

- Spring 3.0 supports JSR 303 (Bean Validation) for validating form objects
 - Hibernate Validator is the reference implementation
 - Annotation-driven
 - Both API and implementation must be on the classpath
- Custom validation
 - Write own validator
 - Implement
`org.springframework.validation.Validator`



Validation Annotations

- @NotNull
 - The field cannot be null
- @Size(min=" ", max=" ")
 - A String field must have a length in the range (min,max)
- @Pattern(regexp=" ")
 - A String field is not null and matches the given pattern
- @NotEmpty
 - Not standard, but supported in Hibernate Validator



Validation Example

```
public class DiningForm {  
  
    @Pattern(regexp="\d{16}")  
    private String creditCardNumber;  
  
    @Size(min=2)  
    private String merchantNumber;  
  
    @Min(0)  
    private BigDecimal monetaryAmount;  
  
    @NotNull  
    private Date date;  
  
}
```

Invoking Validation

- `@Valid` inside controller method

```
@RequestMapping(method=RequestMethod.POST)
public String update(@Valid DiningForm diningForm,
                     BindingResult result) {
    if (result.hasErrors()) {
        return "rewards/edit";
    }
    // continue on success...
}
```

- Errors are registered in the `BindingResult`
 - Combined with binding errors

Validation in the JSP form

```
<form:form modelAttribute="diningForm">  
    <form:errors path="*" />  
  
    <form:input path="firstName" />  
    <form:errors path="firstName" />  
  
    <form:input path="lastName" />  
    <form:errors path="lastName" />  
    ...  
</form:form>
```

Display all errors

Display field-specific errors



Configuring Validation

- Use <mvc:annotation-driven>
- Registers LocalValidatorFactoryBean
 - Enables JSR-303 globally within Spring MVC
 - JSR-303 dependencies must be on the classpath!



Customize Validation Error Messages



- Error messages defined in MessageSource override default values

```
// Any "field not null" error  
NotNull=Required value  
  
// "field not null" errors for a named field  
NotNull.amount=Required amount  
  
// "field not null" errors for a named field on a named  
model attribute  
NotNull.diningForm.amount=Required dining amount  
  
// "field not null" errors for all fields of a specific  
type  
NotNull.common.money.MonetaryAmount=  
Required monetary amount
```

Custom Validations I

- Via custom JSR-303 constraint annotations

```
@Constraint(validatedBy = NumericValidator.class)
public @interface Numeric {
    String message() default "{numberExpected}";
    Class<?>[] groups() default {};
}

@Numeric
private String merchantNumber;

public class NumericValidator
    implements ConstraintValidator<Numeric, String> {

    public void initialize(Numeric n) { // nothing to do }

    // All numeric, no leading zero
    public boolean isValid
        (String str, ConstraintValidatorContext ctx) {
        return (str == null) || str.matches("[1-9][0-9]*");
    }
}
```

Custom Validations II

- Write a Spring MVC validator:

```
class DiningValidator extends Validator {  
    public void validate(Object target, Errors errors) {  
        if ((DiningForm)target)  
            .merchantNumber.matches("[1-9][0-9]*") )  
        errors.rejectValue("merchantNumber", "numberExpected");  
    }  
  
    public boolean supports(Class<?> clazz) {  
        return clazz instanceof Dining.class;  
    }  
}  
  
@InitBinder  
public void initBinder(WebDataBinder binder) {  
    binder.setValidator(new DiningValidator());  
}
```

Register per controller

Custom Validations III

- Explicitly call a form method from controller

```
public class DiningForm {

    public void validate(Errors errors) {
        // Custom validation checks
        if (merchantNumber.matches("[1-9][0-9]*"))
            errors.rejectValue
                ("merchantNumber", "numberExpected");
    }
}

@RequestMapping(method=RequestMethod.PUT)
public String update(DiningForm diningForm,
                     BindingResult result) {
    diningForm.validate(result);
    // process failure or success normally ...
}
```



Lesson Roadmap

- Overview
- Form Rendering
- Formatters
- Data Binding
- Validation
- Form Object Management



Form Object Management



- The form workflow requires access to the form object across two or more requests
 - Initial GET
 - Subsequent POST
- Three alternatives to manage the form object
 - Create it on every request
 - Retrieve it on every request
 - Store it the session between requests



Create It On Every Request



```
@Controller  
@RequestMapping("/rewards")  
public class RewardsController {  
  
    @RequestMapping(value="/new", method=RequestMethod.GET)  
    public String new(Model model) {  
        model.add(new DiningForm());  
    }  
  
    @RequestMapping(method=RequestMethod.POST)  
    public String save(DiningForm dining) {  
        ...  
    }  
}
```

Create a new object during the initial GET

Spring MVC creates a new object, binds request parameters to it, and then passes that in.

Retrieve It On Every Request



```
@Controller  
@RequestMapping("/accounts/{number}")  
public class AccountsController {  
  
    @ModelAttribute  
    public Account addToModel(@PathVariable String number) {  
        return accountManager.findAccount(number);  
    }  
  
    @RequestMapping(value="/edit" method=RequestMethod.GET)  
    public String edit() {  
        ...  
    }  
  
    @RequestMapping(method=RequestMethod.PUT)  
    public String update(Account account, BindingResult result)  
    {  
        ...  
    }  
}
```

**@ModelAttribute method
invoked before any
@RequestMapping method!**

Store It In The Session



```
@Controller  
@RequestMapping("/accounts/{number}")  
@SessionAttributes("account")  
public class AccountsController {  
  
    @RequestMapping(value="/edit" method=RequestMethod.GET)  
    public String edit(@PathVariable String number,  
                       Model model) {  
        model.add(accountManager.findAccount(number));  
    }  
}  
  
Attribute "account" put in HTTP Session
```

Model attributes named “account” to be stored in the HTTP session

```
@RequestMapping(method=RequestMethod.PUT)  
public String update(Account account,  
                     SessionStatus status) {  
    ...  
    accountManager.update(account);  
    status.setComplete();  
}
```

Extracted from HTTP Session

Signals “account” can be removed from the HTTP session

Selecting A Strategy

Create it every time

- Works for new objects
- Form contains all required data

Retrieve it every time

- Works for editing existing objects
- Scales well, simple

Store it in the session

- Works for new and existing objects
- Performs better but doesn't scale as well



Summary

After completing this lesson, you should have learnt that:

- The form object is central to form processing
- Spring MVC provides a form tag library for form rendering
- Data binding populates the form object on submit and provides type conversion





LAB

Building Forms with Spring MVC



Site Personalization With Spring MVC

Allowing users to customize an application's
look-and-feel and locale



Overview

After completing this lesson, you should be able to:

- Describe Internationalization Support
- Manage Locales
- Describe the Look-And-Feel Changes Using Themes



Road Map

- Internationalization Support
- Managing Locales
- Look-And-Feel Changes Using Themes

The MessageSource

- Spring's abstraction for resolving messages from a standard ResourceBundle
- Define a bean named messageSource in a Spring application context
- Get access to localized messages



Configuring a MessageSource

```
<bean id="messageSource"
      class="org.springframework.context...
                           ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>/WEB-INF/messages/account</value>
        </list>
    </property>
</bean>
```

/WEB-INF/messages

account_en.properties

account_es.properties

Account.name=Name

Account.number=Number

Account.name=Nombre

Account.number=Número

Resolving Messages In Java



```
@Controller  
public class AccountsController {  
  
    private MessageSource messageSource;  
  
    @Autowired  
    public void AccountsController(MessageSource messageSource) {  
        this.messageSource = messageSource;  
    }  
  
    @InitBinder  
    public String initBinder  
        (WebDataBinder binder, Locale locale) {  
        String datePattern =  
            messageSource.getMessage("date.pattern", null,  
            "MM-dd-yyyy", locale);  
        ...  
    }  
}
```

Inject the MessageSource

Access localized properties

Resolving Messages In Views



- Spring MVC builds on Spring's MessageSource
- JstlView enables message resolution via JSTL
- The Spring form tags resolve error codes
- Also supports switching locales



CharacterEncodingFilter



- Spring MVC provides a filter that can apply character encoding to requests
- The filter can work in two modes
 - Enforce the encoding
 - Add if the encoding is not already specified
- UTF-8 supports a wide range of languages
- ISO 8859-1 is Western languages mostly



CharacterEncodingFilter Example Configuration



```
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-class>
        org.springframework.web
            .filter.CharacterEncodingFilter
    </filter-class>

    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>!!
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
```

/WEB-INF/web.xml



Road Map

- Internationalization Support
- Managing Locales
- Look-And-Feel Changes Using Themes





The LocaleResolver

- An abstraction for determining the current locale
- The DispatcherServlet looks for a LocaleResolver
 - The bean id is expected to be “localeResolver”
- The default is to use the browser locale
- A LocaleResolver can also store a different locale as requested by the user



LocaleResolver Types

AcceptHeaderLocaleResolver
(enabled by default)

- Reads the locale from the request

CookieLocaleResolver

- Reads/writes the locale to a cookie

SessionLocaleResolver

- Reads/writes the locale to the HTTP Session

The LocaleChangeInterceptor



- A Spring MVC provided interceptor
- Detects requests for locale changes
 - Expects a “locale” request parameter by default

```
http://localhost:8080/rewardsonline?locale=en
```

- Uses the configured LocaleResolver to store the new locale choice

Configuring Locale Switching



- Configure the LocaleChangeInterceptor

```
<mvc:interceptors>
    <bean class=
        "org.springframework.web...LocaleChangeInterceptor" />
</mvc:interceptors>
```

- Specify the LocaleResolver
 - Cannot be the AcceptHeaderLocaleResolver

```
<bean id="localeResolver" class="org.springframework.web
    .servlet.i18n.CookieLocaleResolver"/>
```

Using The RequestContext

- Provides access to request-specific state
 - Current locale, theme, binding errors
- Expose a `requestContext` attribute

```
<bean class="org.springframework.web...  
          InternalResourceViewResolver">  
    <property name="requestContextAttribute"  
              value="requestContext" />  
</bean>
```

- Access request data in a JSP

```
<c:out test="${requestContext.locale.language}" />
```



Road Map

- Internationalization Support
- Managing Locales
- Look-And-Feel Changes Using Themes





What Is A Theme?

- Themes allow dynamic determination of look-and-feel related values
 - Paths to CSS files, images, etc.
- Each theme has a name and is backed by a dedicated `MessageSource` instance
- A user can change an application's look-and-feel at runtime by switching the theme





The ThemeResolver

- An abstraction for determining the current theme name
- The DispatcherServlet looks for a ThemeResolver
 - The bean id is expected to be “themeResolver”
- FixedThemeResolver is configured by default
 - Default theme name of “theme”



ThemeResolver Types

FixedThemeNameResolver
(enabled by default)

- Uses a single theme

CookieThemeResolver

- Reads/writes the theme name to a cookie

SessionThemeResolver

- Reads/writes the theme name to the HTTP Session

Getting Started With Themes



- Create theme.properties on the classpath

/WEB-INF/classes/theme.properties

```
main.css=/styles/main.css  
branding.image=/images/branding.gif
```

- Use the theme tag to resolve theme properties

```
<spring:theme var="mainCss" code="main.css" />  
<c:url var="mainCssUrl" value="${mainCss}" />  
<link type="text/css" type="stylesheet"  
      href="${mainCssUrl}" />
```



ThemeChangeInterceptor

- A Spring MVC provided interceptor
- Detects requests to change the theme name
 - Expects a “theme” request parameter

```
http://localhost:8080/rewardsonline?theme=breezy
```

- Uses the configured ThemeResolver to store the new theme name choice



Example With Two Themes

- Create theme property files on the classpath

/WEB-INF/classes/
theme1.properties
theme2.properties

```
main.css=/theme1/main.css  
branding.image=/theme1/branding.gif  
...  
...
```

- Switching themes

To theme1: <http://localhost:8080/rewardsonline?theme=theme1>
To theme2: <http://localhost:8080/rewardsonline?theme=theme2>



Accessing The Theme Name

- Use the RequestContext if you need to access the current theme in a view
- Expose a requestContext attribute

```
<bean class="org.springframework.web...  
          InternalResourceViewResolver">  
    <property name="requestContextAttribute"  
              value="requestContext" />  
</bean>
```

- Access request data in a JSP

```
<c:out test="${requestContext.theme.name}" />
```



Summary

After completing this lesson, you should have learnt that:

- Spring's architecture has built-in support for internationalization
- Spring MVC enables message resolution within views as well as locale management
- Themes are named resources bundles that allow changing an application's look-and-feel





LAB

Enable Site Personalization Through
Locale And Theme Switching



Introduction to REST

Understanding REST Web Services



Overview

After completing this lesson, you should be able to:

- Describe core REST concepts
- Understand RESTful architecture & design
- Describe the advantages of REST
- Understand RESTful clients with RestTemplate





Lesson Roadmap

- Core REST Concepts
- RESTful architecture & design
- Advantages of REST
- RESTful clients with RestTemplate





What is REST?

- Representational State Transfer
 - Term coined up by Roy Fielding (author of HTTP spec)
- Web apps not just usable by browser clients
 - Programmatic clients can also connect via HTTP
- REST is an architectural style that describes best practices to do this
 - HTTP as *application* protocol, not just transport
 - Emphasizes scalability



REST is NOT

- An API or framework
 - It is only an *architectural style*
- Equal to HTTP
 - REST principles can be followed using other protocols
- The opposite of SOAP
 - REST vs. SOAP is a false dichotomy



Identifiable Resources



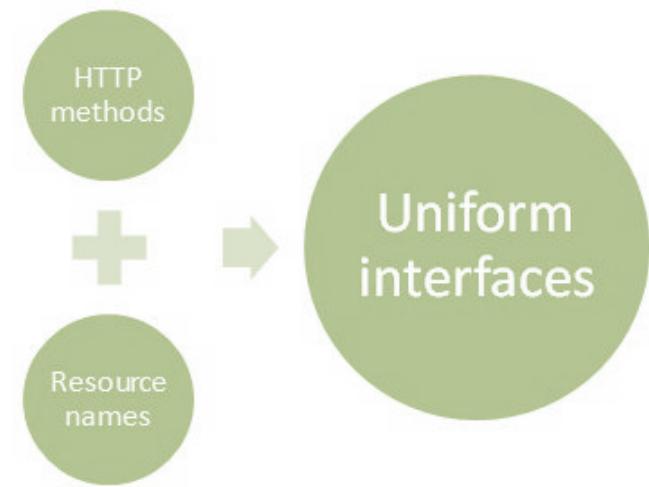
- Everything is a resource
 - Customer
 - Car
 - Shopping cart
- Resources are expressed by URIs
 - Meaning URLs: REST community prefers the term URI
- Each URI adds value to the client
 - Don't just expose your entire domain

```
http://travelx.com/bookings/hotel/ibis/city  
/sydney/room/1234
```



Uniform Interface

- Interact with resources using a *constrained* set of operations
 - Many nouns (resources)
 - Few verbs (operations)
- HTTP has this kind of limited interface
 - GET
 - POST
 - PUT
 - DELETE
 - HEAD and OPTIONS for meta-data



GET



- GET retrieves a Representation of a Resource
- GET is a *safe* operation
 - Has no side effects
- GET is *cacheable*
 - Servers may return ETag header when accessed
 - Clients send this header on subsequent retrieval
 - If the resource has not changed, 304 (Not Modified) is returned, with empty body
 - Similar solution exists for Last-Modified header



GET Examples

```
GET /transfers/121  
Host: www.mybank.com  
Accept: application/xml  
...
```

Accept header
defines representation

```
GET /transfers/121  
Host: www.mybank.com  
Accept: application/json  
...
```

HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type: application/xml

```
<transfer id="121"  
          amount="300.00">  
  <credit>S123</credit>  
  <debit>C456</debit>  
</transfer>
```

HTTP/1.1 200 OK
Date: ...
Content-Length: 83
Content-Type: application/json

```
{ id: 121, amount: 300.00,  
  credit: S123, debit: C456 }
```

Etags – Caching

```
GET /transfers/121  
Host: www.mybank.com  
...  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
ETag: "b4bdb3-5b0-  
43ad74ee73ec0"  
Content-Length: 1456  
Content-Type: text/html  
...
```

```
GET /transfers/121  
If-None-Match: "b4bdb3-  
5b0-43ad74ee73ec0"  
Host: www.mybank.com  
...
```

```
HTTP/1.1 304 Not Modified  
Date: ...  
ETag: "b4bdb3-5b0-  
43ad74ee73ec0"  
Content-Length: 0
```

POST



- POST creates a new Resource
 - Usually as child of existing Resource
 - URI of child in Location response header
- POST is powerful, don't overuse it!
 - Not *safe*, not *idempotent*
 - *Cannot* just resend

 **Idempotent**

Multiple invocations have same end-result

```
POST /transfers
Host: www.mybank.com
Content-Type: application/xml

<transfer>
...
</transfer>
```

```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://mybank.com/
transfers/123
...
```

PUT



- PUT updates a resource or creates it with a known destination URI
- Idempotent operation
 - Same request yields same result
- Not safe! (has side-effects)

Successful update – nothing to return

```
PUT /transfers/123
Host: www.mybank.com
Content-Type: application/xml

<transfer>
...
</transfer>
```

```
HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
...
...
```



PUT Outcomes



```
PUT /transfers/123
Host: www.mybank.com
Content-Type: ...

<transfer>
...
</transfer>
```

HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
...

Updated existing resource

HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://...
...

Created new resource

DELETE

- DELETE deletes a resource
- Idempotent
 - Post condition is always the same
- Not safe!

Successful
delete –
nothing to
return

```
DELETE /transfers/123
Host: www.mybank.com
...
...
```

HTTP/1.1 204 No Content

Date: ...

Content-Length: 0

...

HTTP Methods



	Safe	Idempotent	Cacheable
GET	✓	✓	✓
HEAD	✓	✓	✗
PUT	✗	✓	✗
POST	✗	✗	✗
DELETE	✗	✓	✗

Resource representations



- Resources are abstract
 - Only accessed through a particular *representation*
 - Multiple representations are possible
 - text/html, image/gif, application/pdf
- Request specifies the desired representation using the Accept HTTP header
 - Or sometimes interpreted through file extension in URI like .xml, .pdf, ...
- Response reports the actual representation using the Content-Type HTTP header
 - Best practice: use well-known media types



Stateless Conversation



- Server does not maintain state
 - e.g.: Don't use the HTTP Session!
- Client maintains state through links
- Very scalable
 - Every server instance can serve a request
 - Just put a load balancer in front
- Enforces loose coupling (no shared session knowledge)



Hypermedia

- Resources contain links
 - Client state transitions are made through these links
 - Links are provided by server
- Seamless evolution
 - Don't have to update clients when changing the server, just send new links
- Not always easy
 - Client needs some knowledge of server semantics
 - But REST has no WSDL
 - Should abstract fully: HATEOAS



HATEOAS

- Hypermedia As The Engine of Application State
 - Probably the world's worst acronym!
 - RESTful responses contain the links you need
 - Just like HTML pages do
 - Warning: no standard for this yet
 - Least understood part of Roy Fielding's dissertation

[http://roy.gbiv.com/untangled/2008/
rest-apis-must-be-hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven)

- For example: **GET /orders/123**
 - Response should include possible URLs from here
 - Such as to get order item details, add new item ..



HATEOAS Example

```
GET /orders/123
```

Host: www.myshop.com

Accept: application/xml

...

```
HTTP/1.1 200 OK
```

Date: ...

Content-Length: 1456

Content-Type: application/xml

```
<order id="123" total="300.00">
    <PO-number>S123</PO-number>
    <ship-date>2012-10-27</ship-date>
    <items xlink:href="items" rest:methods="get,post">
        <item desc="widget" xlink:href="items/0"/>
    </items>
</transfer>
```

Just a suggestion – there
is no standard for this



Lesson Roadmap

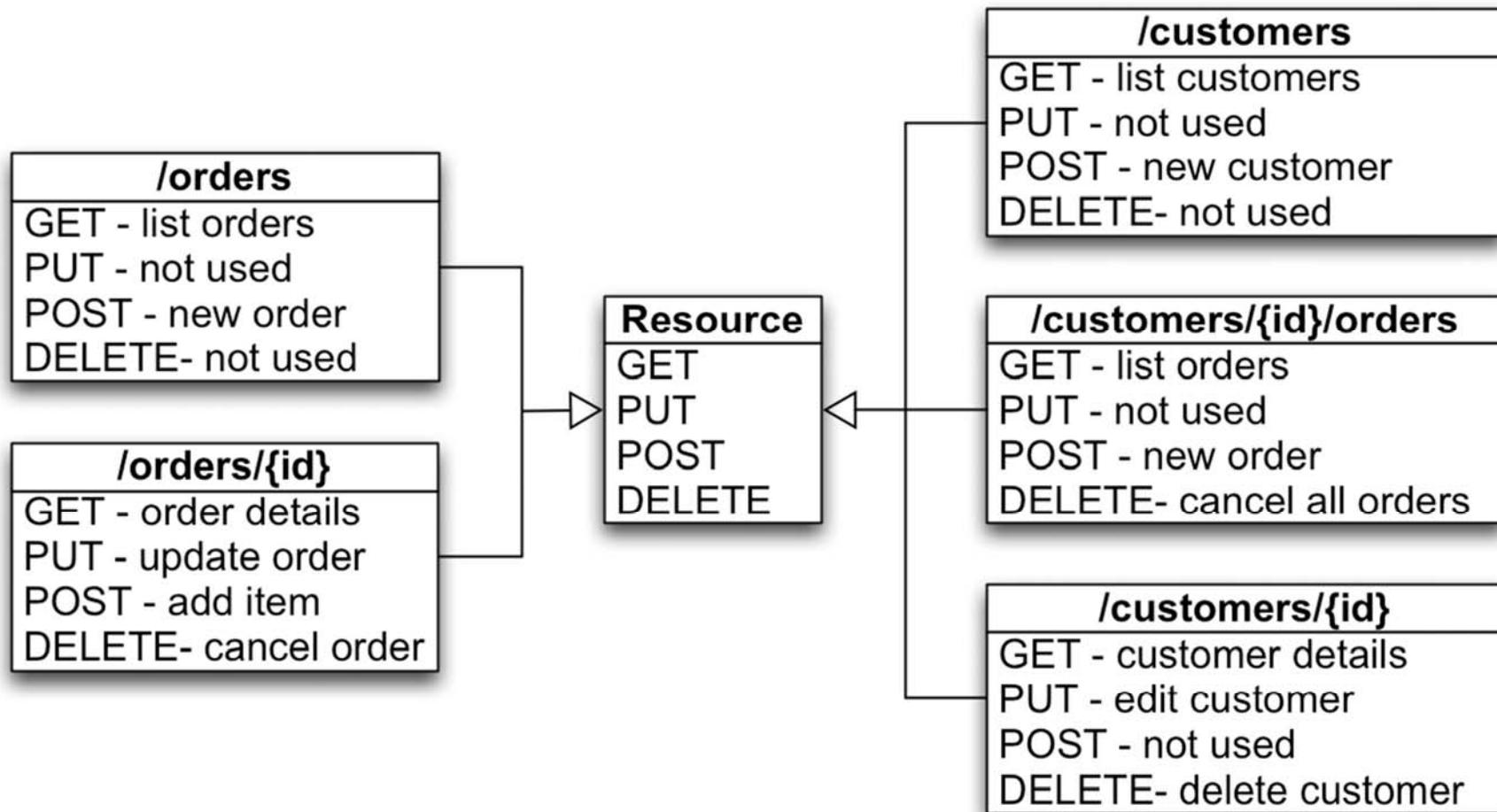
- Core REST Concepts
- RESTful architecture & design
- Advantages of REST
- RESTful clients with RestTemplate



RESTful Architecture

<p><<Service>></p> <p>OrderManagement</p> <p>+ getOrders() : List + submitOrder(Order) + getOrderDetails() : Order + getOrdersForCustomer() : List + updateOrder(Order) + addOrderItem(Item) + cancelOrder(Order)</p>	<p><<Service>></p> <p>CustomerManagement</p> <p>+ getCustomers() : List + addCustomer(Customer) + getCustomerDetails() : Customer + updateCustomer(Customer) + deleteCustomer(Customer)</p>
--	--

RESTful Architecture



RESTful Application Design



- Identify resources
 - Design URLs
- Select representations
 - Or create new ones
- Identify method semantics
- Select response codes





Lesson Roadmap

- Core REST Concepts
- RESTful architecture & design
- Advantages of REST
- RESTful clients with RestTemplate



Advantages of REST



- Widely supported
 - Languages
 - Scripts
 - Browsers
 - Only GET and POST through HTML
- Scalability
- Support for redirect, caching, different representations, resource identification, ...
- Support for XML, but also other formats
 - JSON and Atom are popular choices



Security

- REST uses HTTP Basic or Digest
- Sent on every request
 - REST is stateless, remember ☐
- Requires SSL (transport-level security)
- Use XML-DSIG or XML-Encryption for message-level security
 - Like WS-Security for SOAP messages





Lesson Roadmap

- Core REST Concepts
- RESTful architecture & design
- Advantages of REST
- RESTful clients with RestTemplate



RestTemplate Introduction

- Provides access to RESTful services
 - Encapsulates all HTTP Methods
 - Pass variables as Map or String...
 - Specify url as `java.net.URI` or String
- Configuration Options
 - Support for URI templates
 - Uses `HttpMessageConverters` (later slide)
 - Custom `execute()` with callbacks



RestTemplate API

HTTP Method	RestTemplate Method
DELETE	delete(String url, String urlVariables)
GET	getForObject(String url, Class<T> responseType, String urlVariables)
HEAD	headForHeaders(String url, String urlVariables)
OPTIONS	optionsForAllow(String url, String urlVariables)
POST	postForLocation(String url, Object data, String urlVariables) postForObject(String url, Object data, Class<T> responseType, String uriVariables)
PUT	put(String url, Object data, String urlVariables)

Message Convertors

- Perform object to payload conversion
 - RestTemplate only deals in objects
 - Several defaults available “out of the box”

Converter	Data Type Handled
StringHttpMessage Converter	A string from request/response. <i>Media type (in): text/*, (out): text/plain</i>
MarshallingHttp MessageConverter	XML data using a Spring marshaller/un-marshaller. <i>Media-type: application/xml</i>
MappingJacksonHttp MessageConverter	JSON data using Jackson's ObjectMapper. <i>Media-type: application/json</i>
AtomFeedHttp MessageConverter	ATOM feed using ROME's Feed API. <i>Media: application/atom+xml</i>
RssChannelHttp MessageConverter	RSS feed using ROME's feed API. <i>Media: application/rss+xml</i>

Defining a RestTemplate

- Just call constructor in your code

```
RestTemplate template = new RestTemplate();
```

- Has default `HttpMessageConverters`
 - Same as on the server (depending on classpath)
- Or use external configuration
 - To use Apache Commons HTTP Client, for example

```
<bean id="restTemplate" class="org.sfw.web.client.RestTemplate">
    <property name="requestFactory">
        <bean class=
            "org.sfw.http.client.CommonsClientHttpRequestFactory" />
    </property>
</bean>
```

RestTemplate Usage Examples

```
RestTemplate template = new RestTemplate();
String uri = "http://example.com/store/orders/{id}/items";

// GET all order items for an existing order with ID 1:
OrderItem[] items = template.getForObject(uri,
OrderItem[].class, "1");

// POST to create a new item
OrderItem item = // create item object
URI itemLocation = template.postForLocation(uri, item, "1");

// PUT to update the item
item.setAmount(2);
template.put(itemLocation, item);

// DELETE to remove that item again
template.delete(itemLocation);
```



From Spring 3.0.2, can use `HttpEntity` to define headers for the HTTP request – pass instead of item above

Simulating PUT/DELETE (1)

- One of the largest REST users are browsers via AJAX
 - But setting Accept header in a browser can be hard
- Spring has a neat solution
 - JSP view with Spring form tags

```
<form:form method="put" action="..." modelAttribute="...">
    ...
</form:form>
```

- Resulting HTML form

```
<form method="post" action="...">
    <input type="hidden" name="_method" value="put" />
    ...
</form>
```

Simulating PUT/DELETE (2)

- Special filter intercepts all “POST” requests and wraps HttpServletRequest

```
<filter>
    <filter-name>hiddenMethodFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.HiddenHttpMethodFilter
    </filter-class>
</filter>
```

- Subsequent processing in Spring MVC “sees” request with HTTP PUT or DELETE

Summary [1]

After completing this lesson, you should have learnt that:

- Web apps not just for browser clients
 - Programmatic clients can also connect via HTTP
 - REST is an architectural style that can be applied to HTTP-based applications
- REST exposes resources through URIs
 - Lots of nouns, few verbs (HTTP methods)
 - Multiple representations: HTML, XML, JSON, ...
- REST supports diverse clients
 - Every platform/language supports HTTP
 - Standard HTTP headers and status codes define results



Summary [2]

After completing this lesson, you should have learnt that:

- REST is a viable alternative to the WS-* stack
 - Not necessarily easier
 - Difficult in different ways
- REST builds on
 - Identifiable Resources
 - Uniform interface
 - Stateless conversation
 - No sessions, scalable
 - Build highly scalable systems
 - Resource representations
 - Hypermedia





LAB

Accessing a RESTful Application
with Spring's RestTemplate



Implementing REST with Spring MVC

Using Spring MVC to build RESTful web services



Overview

After completing this lesson, you should be able to:

- Describe Spring MVC support for RESTful applications
- Understand how to access Request/Response Data
- Use MessageConverters





Lesson Roadmap

- REST and Java
- Spring MVC support for RESTful applications
- Accessing Request/Response Data
- Using MessageConverters
- Putting it all together





REST Frameworks

- Multiple Java frameworks for REST exist
- For Spring users, two options are particularly interesting:
 - JAX-RS
 - Spring-MVC 3.0 with updated REST-support
- Both are valid choices depending on requirements and developer experience



JAX-RS

Java API for RESTful Web Services (JSR-311)

- Part of Java EE 6
- Standard for writing RESTful applications

Focuses mostly on application-to-application communication

- Less focus on browsers as RESTful clients

Very complete

Jersey is the reference implementation

- Ships with Spring Integration out of the box
- RESTEasy and Restlet support Spring as well

JAX-RS Example

```
import javax.ws.rs.*;  
  
@Path( "/reward/{number}" )  
@Scope( "request" )  
public class RewardService {  
    @GET  
    @Produces( "text/xml" )  
    public Reward getReward(@PathParam( "number" )  
                           String number) {  
        return findReward(number);  
    }  
}
```

Service ceases to be a singleton

- JAX-RS pollutes your service class
 - Only allows pre-defined representations per class
 - Bad separation of concerns



Spring MVC 3.0

- Spring MVC in Spring 3.0 includes REST support
 - URI templates
 - Message converters
 - Declaring response status codes
 - Content negotiation
 - RestTemplate for clients
 - And more
- Easier for existing MVC users than JAX-RS
 - As JAX-RS requires different programming model
- Also supports browsers as REST clients
 - Through HTTP Method conversion





Spring MVC 3.0 Example

- Rendering the model is delegated to a Converter
 - Allows *multiple* representations with a *single* controller

```
@Controller  
@RequestMapping( "/reward/{number}" )  
public class RewardController {  
    @Autowired  
    RewardService rewardService;  
  
    @RequestMapping(method=GET)  
    public @ResponseBody Reward  
        getReward(@PathVariable("number") String number){  
        return rewardService.findReward(number);  
    }  
}
```



Lesson Roadmap

- REST and Java
- Spring MVC support for RESTful applications
- Accessing Request/Response Data
- Using MessageConverters
- Putting it all together





Spring-MVC and REST

- Some features have been covered already
 - Map requests based on HTTP method
 - URI templates to parse 'RESTful' URLs
- Will now extend Spring MVC to support REST
 - More Annotations
 - Message Converters
- Support for RESTful web applications targeting browser-based clients is also offered





HTTP Status Code Support

- Web apps just use a handful of status codes
 - Success: 200 OK
 - Redirect: 302/303 for Redirects
 - Client Error: 404 Not Found
 - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many *additional* codes to communicate with their clients
- Add `@ResponseStatus` to controller method
 - don't have to set status on `HttpServletResponse` manually



Common Response Codes

200

- After a successful GET where content is returned

201

- When new resource was created on POST or PUT
- Location header should contain URI of new resource

204

- When the response is empty
- e.g. after successful update with PUT or DELETE

404

- When requested resource was not found

405

- When HTTP method is not supported by resource

409

- When a conflict occurs while making changes
- e.g. when POSTing unique data that already exists

415

- When request body type not supported



@ResponseStatus

```
@RequestMapping(value="/orders", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrder(HttpServletRequest request) {
    Order order = getOrder(request); // Get order info from
                                    // request
    orderService.amendOrder(order);
}
```

- When using `@ResponseStatus`, **void** methods no longer imply a default view name!
 - There will be no View at all
 - Example generates a response with an empty body

@ResponseStatus & Exceptions



- Can also annotate exception classes with this
 - Given status code used when exception is thrown from controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
public class OrderNotFoundException extends RuntimeException {
    ...
}
```

```
@RequestMapping(value="/orders/{id}", method=GET)
public String showOrder(@PathVariable("id")
                      long id, Model model) {
    Order order = orderRepository.findOrderById(id);
    if (order == null) throw new OrderNotFoundException(id);
    model.addAttribute(order);

    return "orderDetail";
}
```

@ExceptionHandler

- For existing exceptions you cannot annotate,
 - Use `@ExceptionHandler` method in controller
 - Method signature similar to request handling method
 - Also supports `@ResponseStatus`

```
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler({DataIntegrityViolationException.class})
public void conflict() {
    // could add the exception,
    // response, etc. as method params
}
```



Lesson Roadmap

- REST and Java
- Spring MVC support for RESTful applications
- Accessing Request/Response Data
- Using MessageConverters
- Putting it all together



Accessing Servlet Environment

- Can explicitly inject `HttpServletRequest/Response`
 - But makes Controller methods hard to test
 - Consider Spring's `MockHttp...` classes
- Spring can inject some request data automatically
 - `@RequestParam`, `@PathVariable`
 - Principle
- To perform REST we need more
 - Current URL
 - Request headers
 - Response headers



Accessing Request Properties

- Access URL without using HttpServletRequest?
 - Use `@Value` and SpEL instead
 - Can access *any* request properties this way
 - A POST typically needs this

```
@RequestMapping(value= "/orders", method=RequestMethod.POST)
@ResponseBody(HttpStatus.CREATED)      // 201
public void createOrder (@Value( "#{request.requestURL}")
                           StringBuffer originalUrl, ... )
{ ... }
```

Accessing Request Data

- Spring can inject via other annotations
 - `@RequestHeader`
 - `@CookieValue` (not usual for REST)

```
@RequestMapping(value="/orders/{id}",
method=RequestMethod.GET) @ResponseStatus(HttpStatus.OK)// 200
public void getOrder (
    @RequestHeader("user-agent") String browserType,
    @CookieValue("jsessionid") String sessionId,
    @Value("#{request.requestURI}") String url)
{ ... }
```

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>

Setting Response Data

- Avoid explicit use of `HttpServletResponse`
 - Use `HttpEntity` and `HttpHeaders` instead
 - `@ResponseStatus` still needed (to disable views)

```
@RequestMapping(value="/orders/{id}" , method=RequestMethod.GET)
@ResponseStatus(HttpStatus.OK)      // 200
public HttpEntity<String> getOrder
    (@RequestParam( "id" ) long orderId) {
    String order = orderService.find(orderId);

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set( "Content-Type" , "text/xml" );

    return new
        HttpEntity<String>(order.toXml() , responseHeaders);
}
```

Response
body type

Response
body

POST Response - Determining Location Header



- URL of created child resource usually a sub-path
 - POST to **http://www.myshop.com/store/orders** gives **http://www.myshop.com/store/orders/123**
 - Can use Spring's UriTemplate for encoding as needed
 - Handles escaping characters in URL

```
private String getLocationForChildResource
                (StringBuffer url, Object childId) {
    // Define child URL using template parameter(s)
    String childUrl = url.append("/{childId}").toString();
    // Create a template, resolve parameter(s),
    // return URL as string
    return new
        UriTemplate(childUrl).expand(childId).toASCIIString();
}
```



Lesson Roadmap

- REST and Java
- Spring MVC support for RESTful applications
- Accessing Request/Response Data
- Using MessageConverters
- Putting it all together



HttpMessageConverter

- Converts between HTTP request/response and object
- Various implementations registered by default when using `<mvc:annotation-driven/>`
 - XML (using JAXP Source or JAXB2 mapped object*)
 - Feed data*, i.e. Atom/RSS
 - Form-based data
 - JSON*
 - Byte[], String, BufferedImage
- Define HandlerAdapter explicitly to register other HttpMessageConverters

* Requires 3rd party libraries on classpath



You **must** use `<mvc:annotation-driven/>` or register custom converters to have any HttpMessageConverters defined at all!

@RequestBody

- Use converters for request data by annotating method parameter with `@RequestBody`

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrder(@RequestBody Order updatedOrder,
                        @PathVariable("id") long id) {
    // process updated order data and return empty response
    orderManager.updateOrder(id, updatedOrder);
}
```

- Right converter chosen automatically
 - Based on content type of request
 - Order could be mapped from XML with JAXB2 or from JSON with Jackson, for example

@ResponseBody

- Use converters for response data by annotating method with @ResponseBody

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseBody(HttpStatus.OK) // 200
public @ResponseBody Order getOrder
    (@PathVariable("id") long id) {
    // Order class is annotated with JAXB2's @XmlRootElement
    Order order= orderRepository.findOrderById(id);
    // results in XML response containing marshalled order:
    return order;
}
```

- Converter handles rendering to response
 - No ViewResolver and View involved any more!

Content Negotiation - Accept

```

@RequestMapping(value="/orders/{id}" ,
                 method=RequestMethod.GET)
@ResponseBody @ResponseStatus(HttpStatus.OK) // 200
public Order getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}

```

GET /store/orders/123
 Host: www.myshop.com
Accept: application/xml, ...
 ...

HTTP/1.1 200 OK
 Date: ...
 Content-Length: 1456
Content-Type: application/xml
<order id="123">
...
</order>

GET /store/orders/123
 Host: www.myshop.com
Accept: application/json, ...
 ...

HTTP/1.1 200 OK
 Date: ...
 Content-Length: 756
Content-Type: application/json
{
 "order": {"id": 123, "items": [...], ... }
}

Views and Annotations - 1

- REST methods cannot return HTML, PDF, ...
 - No message converter
 - Views better for presentation rich representations
- Need two methods on controller *for same URL*
 - One uses a converter, the other a view
 - Identify using **produces** attribute
- Use **consumes** to differentiate RESTful POST/PUT from a form submission

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,  
    produces = {"application/json"})  
  
@RequestMapping(value="/orders/{id}", method=RequestMethod.POST,  
    consumes = {"application/json"})
```

Views and Annotations - 2

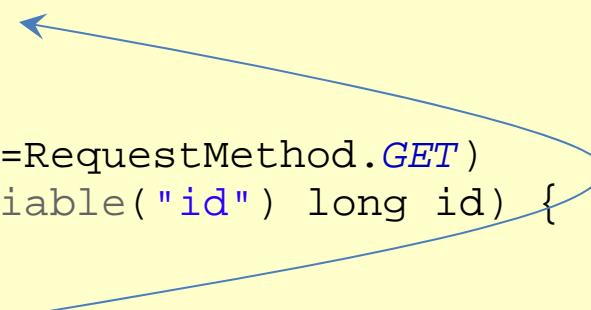
- Recommendation
 - Mark RESTful method with produces
 - To avoid returning XML to normal browser request
 - Call RESTful method from View method
 - Implement all logic once in RESTful method

```

@RequestMapping(value="/orders/{id}" , method=RequestMethod.GET ,
    produces = {"application/json"})
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}

@RequestMapping(value="/orders/{id}" , method=RequestMethod.GET)
public String getOrder(Model model, @PathVariable("id") long id) {
    model.addAttribute(getOrder(id));
    return "orderDetails";
}

```





Lesson Roadmap

- REST and Java
- Spring MVC support for RESTful applications
- Accessing Request/Response Data
- Using MessageConverters
- Putting it all together



Retrieving a Representation: GET



```
GET /store/orders/123
Host: www.myshop.com
Accept: application/xml,
...
...
```

```
HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type:
application/xml

<order id="123">
...
</order>
```

```
@RequestMapping(value="/orders/{id}" , method=RequestMethod.GET)
@ResponseStatus(HttpStatus.OK) // 200: this is the default
public @ResponseBody Order
    getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}
```

Creating a new Resource: POST

```
POST /store/orders/123/items  
Host: www.myshop.com  
Content-Type: application/xml  
  
<item>  
...  
</item>
```

```
HTTP/1.1 201 Created  
Date: ...  
Content-Length: 0  
Location:  
http://www.myshop.com/store/orders/123/items/abc
```

```
@RequestMapping(value="/orders/{id}/items", method=RequestMethod.POST)  
@ResponseStatus(HttpStatus.CREATED) // 201  
public ResponseEntity<?> createItem(@PathVariable("id") long id,  
        @RequestBody Item newItem,  
        @Value("#{request.requestURL}") StringBuffer originalUrl) {  
    orderRepository.findOrderById(id).addItem(newItem); // adds id to Item  
  
    HttpHeaders responseHeaders = new HttpHeaders();  
    responseHeaders.setLocation  
        (getLocationForChildResource(originalUrl, newItem.getId()));  
    return new ResponseEntity<String>(responseHeaders);  
}
```

Updating existing Resource: PUT



```
PUT /store/orders/123/items/abc  
Host: www.myshop.com  
Content-Type: application/xml  
  
<item>  
...  
</item>
```

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

```
@RequestMapping(value="/orders/{orderId}/items/{itemId}" ,  
    method=RequestMethod.PUT)  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
public void updateItem(@PathVariable("orderId") long orderId,  
    @PathVariable("itemId") String itemId  
    @RequestBody Item item) {  
    orderRepository.findOrderById(orderId)  
        .updateItem(itemId, item);  
}
```

Deleting a Resource: DELETE

```
DELETE /store/orders/123/items/abc  
Host: www.myshop.com  
...
```

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

```
@RequestMapping(value="/orders/{orderId}/items/{itemId}",  
    method=RequestMethod.DELETE)  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
public void deleteItem(@PathVariable("orderId") long orderId,  
    @PathVariable("itemId") String itemId) {  
    orderRepository.findOrderById(orderId).deleteItem(itemId);  
}
```

Summary

After completing this lesson, you should have learnt that:

- Java provides JAX-RS as standard specification
 - Breaks MVC
- Spring-MVC adds REST support using a familiar programming model
 - Extended by @Request/@ResponseBody, @ResponseStatus, @PathVariable
 - Message convertors replace views
 - Automatic content-negotiation
 - Use produces to support both





LAB

Building a RESTful server
with Spring MVC



Building Rich Web Applications with Ajax



Overview

After completing this lesson, you should be able to:

- Describe what Ajax is
- Describe XMLHttpRequest
- Understand the basics of jQuery
- Simplify HTML code using Custom Tags



Lesson Roadmap

- Introducing Ajax
- Introduction to jQuery
- Custom Tags



What Is Ajax?

- A term coined by Jesse James Garrett
- Ajax is a set of technologies that allow web applications to provide
 - Rich Interaction
 - Just-in-time Information
 - Dynamic Information without required page refresh
- Ajax → **Asynchronous JavaScript and XML**

Ajax: A New Approach To Applications
<http://adaptivepath.com/ideas/essays/archives/000385.php>



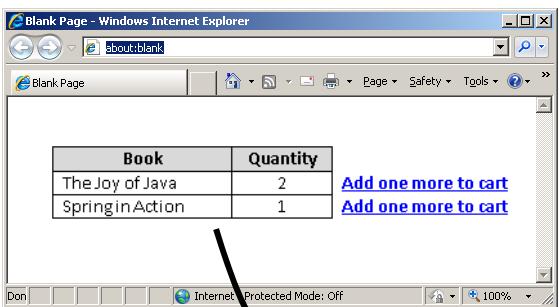
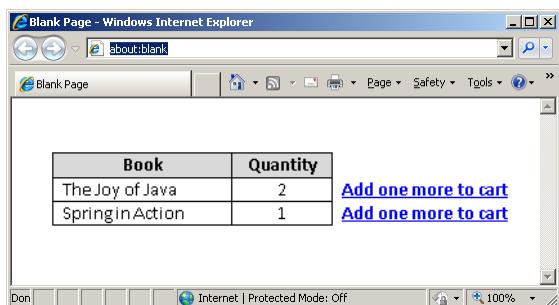
Ajax and Spring MVC



- Spring MVC allows you to choose an AJAX/JavaScript library
 - No JavaScript library is integrated by default
 - It is the developer's responsibility to choose a JavaScript library
 - usually an easy task
 - jQuery is a common choice



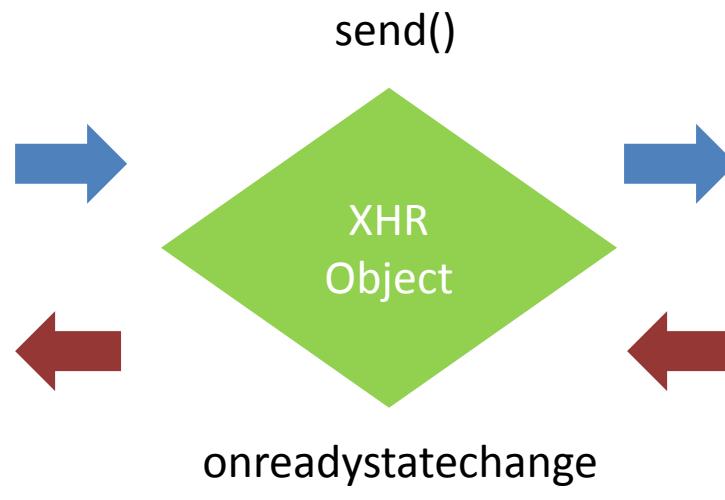
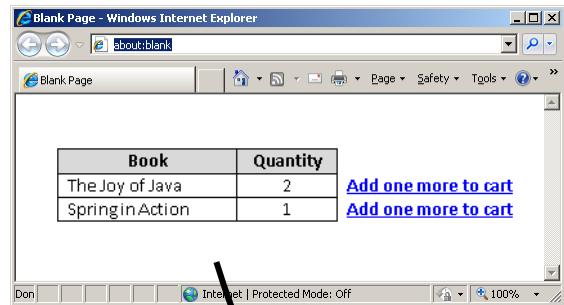
Classic Web



Server

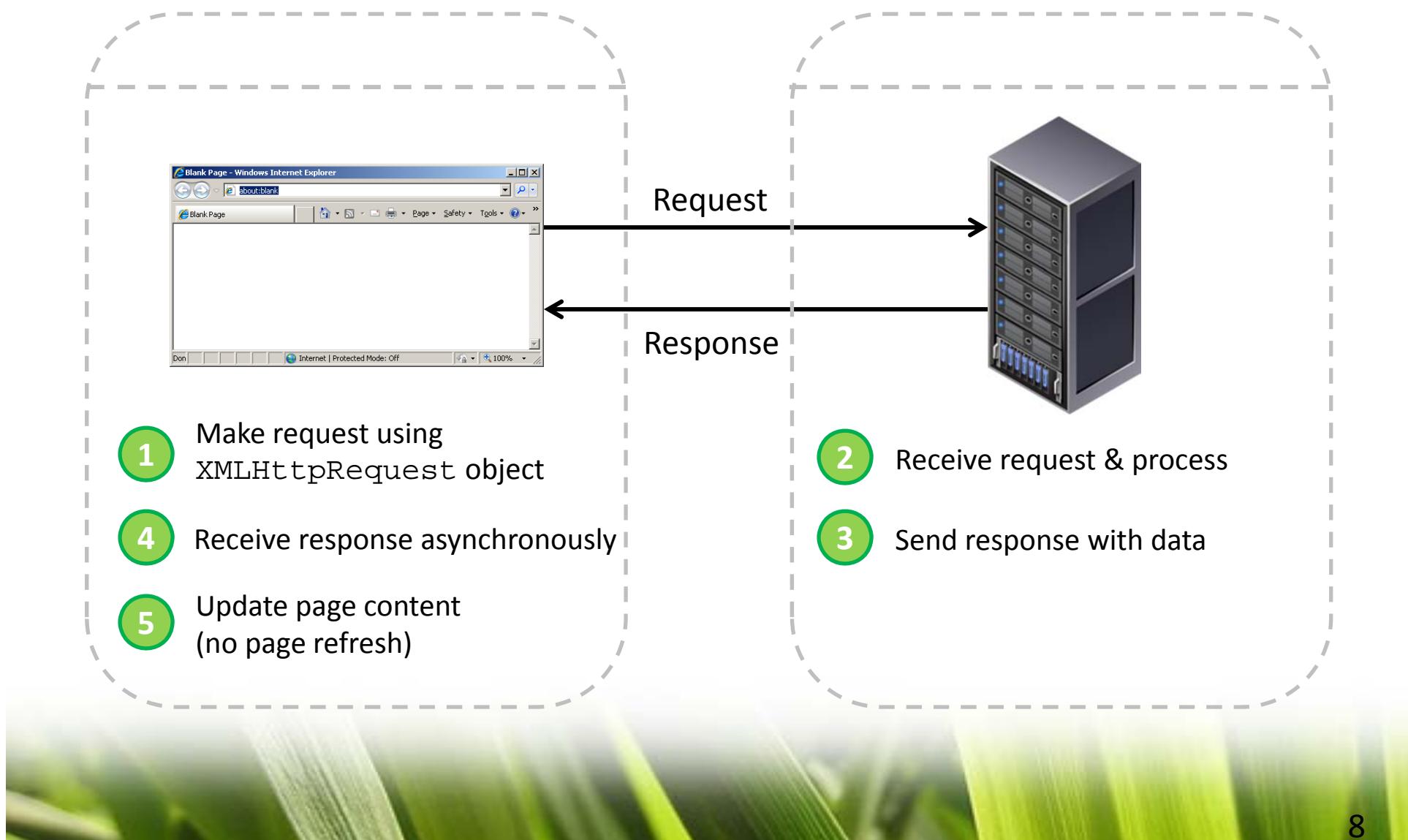
The whole page gets refreshed everytime
you get a response

Web with Ajax



Updates a portion of the same page
(partial rendering)

A Closer Look at AJAX



Creating XMLHttpRequest Object



- All modern browsers support the XMLHttpRequest object (IE5 and IE6 use an ActiveXObject).
- The XMLHttpRequest object is used to exchange data with a server behind the scenes.
 - Update parts of a web page, without reloading the whole page

```
var xhr = null  
xhr = new XMLHttpRequest();
```

XMLHttpRequest Methods

Method	Description
<code>open("method", "url", [, asynchFlag [, "username" [, "password"]]])</code>	Sets up the request object for sending a request
<code>send(content)</code>	Sends the request to the server. Can be null
<code>abort()</code>	Stops the request
<code>getAllResponseHeaders()</code>	Returns all response headers for the HTTP request as key/value pairs
<code>getReponseHeader("header")</code>	Returns the string value of the specified header
<code>setRequestHeader("header", "value")</code>	Sets the specified header to the supplied value

XMLHttpRequest Properties



Property	Description
onreadystatechange	The event handler that fires at every state change
readystate	The state of the request: 0 → uninitialized 1 → loading 2 → loaded 3 → interactive 4 → complete
responseText	The response from the server as a text string
responseXML	The response from the server as an XML document
status	The HTTP status code from the server 200 → Ok; 201 → Created; 400 → bad request; 403 → Forbidden; 500 → internal sever error
statusText	The text version of the HTTP status code

Making a Request

- The JavaScript function `getResponse` will be invoked when the `readystate` property changes on the `XmlHttpRequest` object

```
xhr.onreadystatechange = handleAjaxResponse;  
xhr.open("GET", "response.xml", true);
```

`xhr.open("GET", "response.xml", true);`

the type of request:
GET or POST

the location of the file
on the server

Asynchronous
True or False?

Parsing the Response

- To get the response from a server, use the `responseText` or `responseXML` property of the `XMLHttpRequest` object
- The `responseText` property returns the response as a string, and you can use it accordingly

```
document.getElementById( "div" ).innerHTML = xhr.responseText;
```

- If the response from the server is XML, and you want to parse it as an XML object, use the `responseXML` property

```
xmlDoc = xhr.responseXML;  
var data = xmlDoc.getElementsByTagName( "someElement" );
```

Parsing the Response

- The `onreadystatechange` event is triggered every time the `readyState` changes.
- The `readyState` property holds the status of the XMLHttpRequest.
- When `readyState` is 4 and `status` is 200, the response is ready

```
xhr.onreadystatechange = handleAjaxResponse() {
    if (xhr.readyState==4 && xhr.status==200) {
        document.getElementById("myDiv").innerHTML =
            xmlhttp.responseText;
    }
}
```

Ajax Request Example

```
<script type="text/javascript">
function loadData() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = handleAjaxResponse();
    if (this.readyState == 4) {
        if (this.status >= 200 && this.status < 300) {
            document.getElementById('myResultArea')
                .innerHTML = this.responseText;
        }
    }
    xhr.open('GET', 'myUrl');
    xhr.send();
}
</script>
```

Creating the request

Initiate request

Client invoking Ajax on the click of a button

```
<div id="myResultArea"> </div>
<button type="button" onclick="loadData()">Update</button>
```



Lesson Roadmap

- Introducing Ajax
- Introduction to jQuery
- Custom Tags



Why Use a JavaScript Frameworks



Frameworks can simplify the process of formulating an Ajax request

- more expressive
- less ceremony

Focus on key aspects of the request

- what server resource will be called (url)
- what parameters will be passed
- what callback method to respond to



jQuery Overview

- jQuery is a library of JavaScript Functions
- The jQuery community is very dynamic
 - Many plugins available



jQuery Overview

The jQuery library contains the following features:

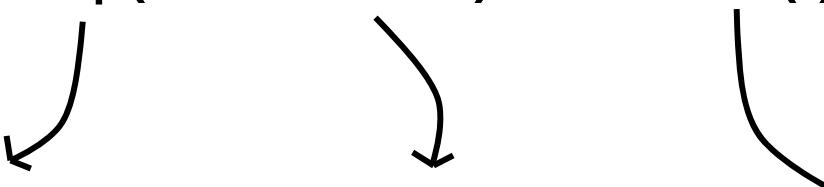
- Small footprint and fast
- Strong cross-browser support
- HTML element selections
- HTML element manipulation
- CSS manipulation
- HTML event functions
- JavaScript effects and animations



jQuery Syntax

- The jQuery syntax is tailor made for selecting HTML elements and perform some action on the element(s)

`$(selector).action()`



A diagram illustrating the components of the jQuery syntax. The code `$selector.action()` is shown at the top. Three arrows point downwards from each part to their respective descriptions: the dollar sign from the first character, the selector from the second character, and the action() from the third character.

A dollar sign to define jQuery

A (selector) to "query" HTML elements

A jQuery action() to be performed on the element(s)



Selecting Elements with jQuery



- Selecting elements by type

```
jQuery("p") // Selects all <p> elements  
$("p")      // Shorter, more commonly used method  
$("p a")    // Selects all anchor tags inside a  
             // paragraph tag
```

HTML element

- Selecting by tag id

```
$("#eName") // Selects element having id=eName
```

- Selecting by tag class

```
$(".myHeader") // Selects element(s) having  
                // class=myHeader
```

Getting & Setting Values

- Getting a value (ex from a text field)

```
var myFieldVal = $("#searchString").val();
```

- Setting a value

```
$( "#eName" ).val("Clarence");
var myVar = "Clarence";
$( "#myId" ).val(myVar); // Set using a variable
```

jQuery: Manipulating HTML



- jQuery provides several methods for manipulating HTML and text of selected elements

```
// Getting HTML content of selected element
var htmlContent = $("#myId").html();

// Replacing HTML content of selected element
var htmlContent = "<ul><li>one</li><li>two</li><li>
                     three</li></ul>"
$("#myId").html(htmlContent);

// Clearing contents of selected element
$("#myId").empty();

// Appending HTML to the selected element
$("#myId").append("<tr><td>Some data</td></tr>");
```

jQuery: Visual Tricks



- jQuery provides some basic helper functions for hiding and showing HTML elements & children

```
// Hide selected element(s) and children
$("#myId").hide(); //Optional parameters slow|fast|normal

// Show selected element(s) and children (affects visibility)
$("#myId").show(); // Optional parameters slow|fast|normal

// Fading out (affects opacity)
$("#myId").fadeOut(); // Optional parameters
                      // slow|fast|normal

// Fading in
$("#myId").fadeIn(); // Optional parameters slow|fast|normal
```

Attaching Behavior to Elements using JavaScript events



- Only one function per event

```
<button id="submitButton" value="Click Me" />

$(function() {
    // Attach 'onClick' DOM level 0 event handler to
    // the submit button
    $('#submitButton')[0].onclick =doSomethingOnClick;
} );

function doSomethingOnClick(event) {
    // Do something here
}
```



plain JavaScript

Attaching Behavior to Elements using jQuery events



- Adding behavior using jQuery event model
- Wraps JavaScript event
 - Directly exposes common event properties
 - Provides access to Event object itself
- Can attach multiple functions per event

```
$(function() {  
    // Attach 'onClick' event handler to the submit button  
    $('#submitButton').bind('click', doSomethingOnClick);  
});  
  
function doSomethingOnClick(event) {  
    // Do something here (like make an Ajax call)  
}
```

jQuery syntax



JavaScript Vs. jQuery

The JavaScript Way

Get me all of the elements that have the tag name of “p”

```
document.getElementsByTagName  
("p") [0].innerHTML = "Text";
```

Get me the zeroth element

Set the HTML inside this element

..to this

jQuery Way

Get the paragraph element

```
$( "p" ).html( "Text" );
```

jQuery uses a “selector engine,” which means you can get elements with selectors just like CSS does.

Change the HTML of that element to this

jQuery GET Ajax Call



- `$.get(url, parameters, callback, type)`
 - `url`: The URL of the server side resource to request via GET
 - `parameters`: Any data that should be passed via GET (string, Object, Array)
 - `callback`: A function to contact as a callback. This can be defined inline or can reference another function by name
 - `type`: How to interpret the response body (html, text, xml, json, script, etc). By default, uses the content-type of the response body.



jQuery GET Example



```
$(function() {
    // 'onClick' event handler to the submit button
    $('#submitButton').bind('click', doAjaxCall);
}) ;

function doAjaxCall(event) {
    $.get('/ajaxUrl', {searchString : 'foo',
                      maximumResults : 5},
          function(response) {
              // Handle response data (may have to
              // determine return type)
          }
    );
}
```

Optional Parameter



jQuery GET JSON Ajax Call



- `$.getJSON(url, parameters, callback)`
 - `url`: The URL of the server side resource to request via GET
 - `parameters`: Any data that should be passed via GET (string, Object, Array)
 - `callback`: A function to contact as a callback. This can be defined inline or can reference another function by name
- The response is interpreted as a JSON object





Spring MVC, Ajax and jQuery

- The primary benefit of AJAX is that we can have one JSP page to handle both the request and result on the same page
- Controller to handle the page request



Spring MVC and jQuery

- The `@ResponseBody` annotation instructs Spring MVC to serialize
- Spring MVC automatically serializes to JSON because the client accepts that content type

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public @ResponseBody Integer add
    (@RequestParam(value="inputNumber1", required=true)
     Integer inputNumber1,
     @RequestParam(value="inputNumber2", required=true)
     Integer inputNumber2, Model model) {

    // Delegate to service to do the actual adding
    Integer sum = springService.add(inputNumber1, inputNumber2);

    // @ResponseBody will automatically convert the returned
    // value into JSON format
    return sum;
}
```

Spring MVC and jQuery [The Client]



- The add() function is a wrapper to jQuery's post() function
- It takes any number of arguments and a callback function to handle the result.

```
<input type="submit" value="Add" onclick="add()" />

function add() {
    jq(function() {
        jq.post("/spring-mvc-jquery/krams/main/ajax/add",
            { inputNumber1: jq("#inputNumber1").val(),
              inputNumber2: jq("#inputNumber2").val() },
            function(data){
                jq("#sum").replaceWith(' <span id="sum">' + data + '</span> ');
            });
    });
}
```



Lesson Roadmap

- Introducing Ajax
- Introduction to jQuery
- Custom Tags



Need for Custom Tags

- Spring @MVC does not come with taglibs for HTML generation
 - Only comes with taglibs for ‘Form’ and ‘URL Rewriting’
 - This can lead to verbose JSPs

```
<%@ taglib prefix="c"  
        uri="http://java.sun.com/jsp/jstl/core" %>  
  
<c:url value="/${url}" var="url" /> "injects the context path"  
<a href="${url}">Click here</a>
```

Verbose, because we need
two lines for a simple link

Need for Custom Tags

```
<%@ taglib prefix="c"  
        uri="http://java.sun.com/jsp/jstl/core" %>  
  
<c:url value="/${url}" var="url"/>  

```

"injects the context path"

Verbose, because we need two lines for a simple link

```
<a href="<%="request.getContextPath( )%>/home">  
    Click here  
</a>
```

Difficult to maintain such code

Not Good! It is using a Java scriptlet, which should be avoided when possible

What Is a Custom Tag?

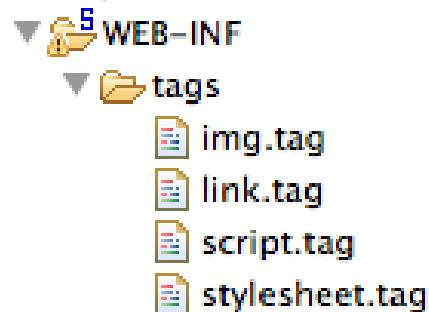
- A custom tag is a user-defined JSP language element
- In addition to the JSP built-in tags like `<jsp:useBean>` and `<jsp:setProperty>`, you can create your own tags
- Custom tags are a simple way to reduce the size of your JSPs significantly
- Example:

```
<mt:addressBook>  
<mt:doSomething>
```



Creating a tag file: Example

- A dedicated file needs to be created for each tag:



```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ attribute name="url" required="true" rtexprvalue="true" %>
<c:url value="/home" var="url"/>

```

img.tag

```
<%@ taglib prefix="mytags" tagdir="/WEB-INF/tags" %>
...
<mytags:img url="images/logo.png" />
```

Using the tag inside a JSP

When tag files are not enough?



- Tag files have been introduced with J2EE 1.4
- Earlier there was a more complicated solution
 - Called TLD (Tag Library Descriptors)
 - more verbose
 - requires XML configuration and a dedicated Java class for each Tag



Summary

After completing this lesson, you should have learnt:

- What Ajax is?
- The basics of XMLHttpRequest
- The basics of JQuery
- How to simplify HTML code using Custom Tags





LAB



Getting Started with Spring Web Flow

Introduction to implementing controlled
navigations with Spring



Overview

After completing this lesson, you should be able to:

- Describe what Web Flow is
- Describe the problems that Web Flow solves
- Describe Web Flow Architecture
- Understand how Web Flow processes a request
- Using Web Flow with Spring MVC



Road Map

- What Web Flow is
- Problems Web Flow solves
- Web Flow Architecture
- How Web Flow Works
- Using Web Flow with Spring MVC

Spring Web Flow



- Framework for implementing stateful “flows”
- Plugs into Spring MVC as a Controller
 - Complements the stateless @Controller model
- Supports a variety of view technologies
 - Integrates JSP and JSF natively





What is a Flow?

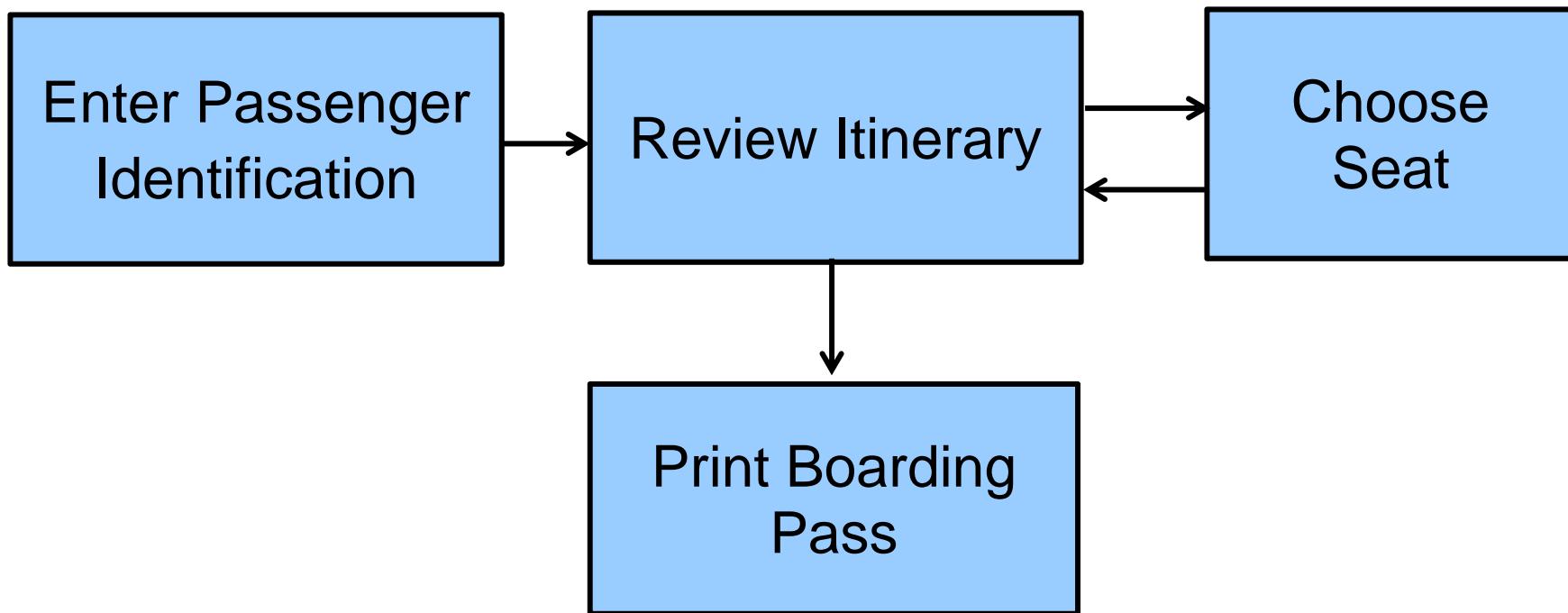
- A flow models a task the user wishes to accomplish
 - BookFlight, ApplyForLoan, GetInsuranceQuote
- Guides a user through several steps
- Often reused in different contexts
 - e.g. many places a Login flow can be invoked



Good Example of a Flow



- Flight Check In



How are Flows Defined?



- As a single artifact
 - Often an XML file
 - Packaged with supporting resources such as views
- Implemented in a high-level “flow definition language”
- With a natural mapping to a graphical flow diagram
 - Visual tooling provided in the SpringSource Tool Suite



Example Flow

```
<flow>
    <view-state id="enterPassengerIdentification">
        <transition on="submit" to="reviewItinerary" />
    </view-state>

    <view-state id="reviewItinerary">
        <transition on="print" to="printBoardingPass" />
    </view-state>

    <view-state id="printBoardingPass" />
</flow>
```



Road Map

- What Web Flow is
- Problems Web Flow solves
- Web Flow Architecture
- How Web Flow Works
- Using Web Flow with Spring MVC



Problems Web Flow Solves

Short-circuiting navigation rules

Duplicate submissions

Stale session state

Browser back and refresh buttons

State collision between windows





Road Map

- What Web Flow is
- Problems Web Flow solves
- **Web Flow Architecture**
- How Web Flow Works
- Using Web Flow with Spring MVC



Web Flow Architecture



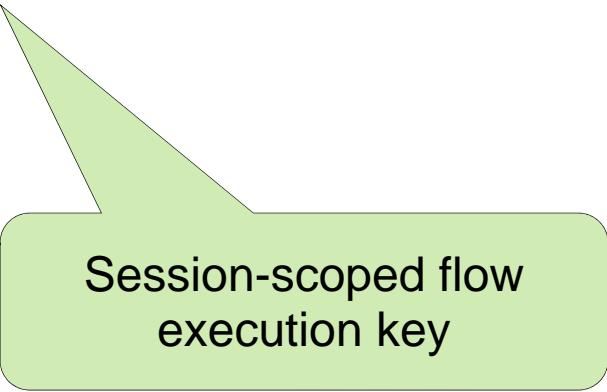
- Flow definitions are registered with the system
- Registered flows are exposed as web resources
- Clients start flows by accessing these resources
 - for example accessing:
<http://localhost/flights/checkin>
 - starts the “flights/checkin” flow



Starting Flow Execution



- Starting a flow begins a new *flow execution*
 - Stores the context for a task currently “in progress” but not yet completed
- Execution context is user *session-scoped*
 - <http://localhost/flights/checkin?execution=e1s1>



Session-scoped flow execution key

Signaling Flow Events

- A user event resumes flow processing
 - Events triggered by activating UI buttons, links
 - `/flights/checkin?execution=e1s1&_eventId=update`
- Provided user data is bound to flow context
- Flow handles event, decides what to do next
 - Display another view to the user?
 - Finish the task?





Ending a Flow Execution

- When a flow execution ends, it cannot be resumed
 - Not possible to complete the same transaction multiple times
- Flow state cleaned up automatically





Road Map

- What Web Flow is
- Problems Web Flow solves
- Web Flow Architecture
- How Web Flow Works
- Using Web Flow with Spring MVC



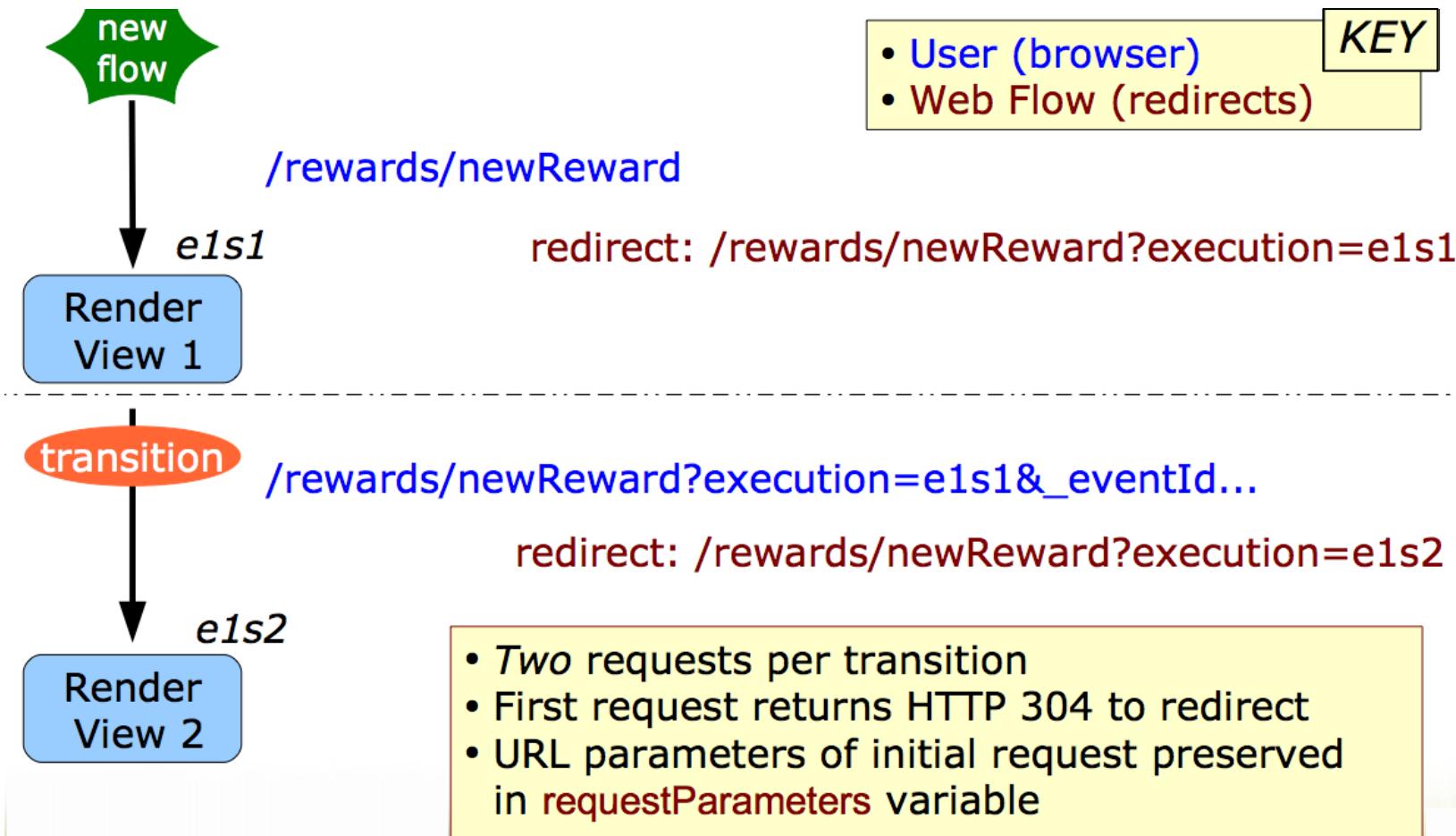
How Web Flow Works



- When Web Flow receives a transition request
 - It is submitted to the same URL as current state
 - The flow “wakes up”
 - The next state is determined
 - A redirect is issued to the new URL for the new state
- So every Web Flow interaction uses *two requests*
- Consequence:
 - Data associated with the initial request is *not available* when the next state (next view) is generated



How Web Flow Works





Road Map

- What Web Flow is
- Problems Web Flow solves
- Web Flow Architecture
- How Web Flow Works
- Using Web Flow with Spring MVC





Configuring Web Flow

- Web Flow provides a Spring XML namespace for configuration
- Use it to setup Web Flow inside Spring MVC
- Recommended to define setup across two configuration files
 - `webflow-config.xml` for core Web Flow system
 - `mvc-config.xml` for Spring MVC adapters



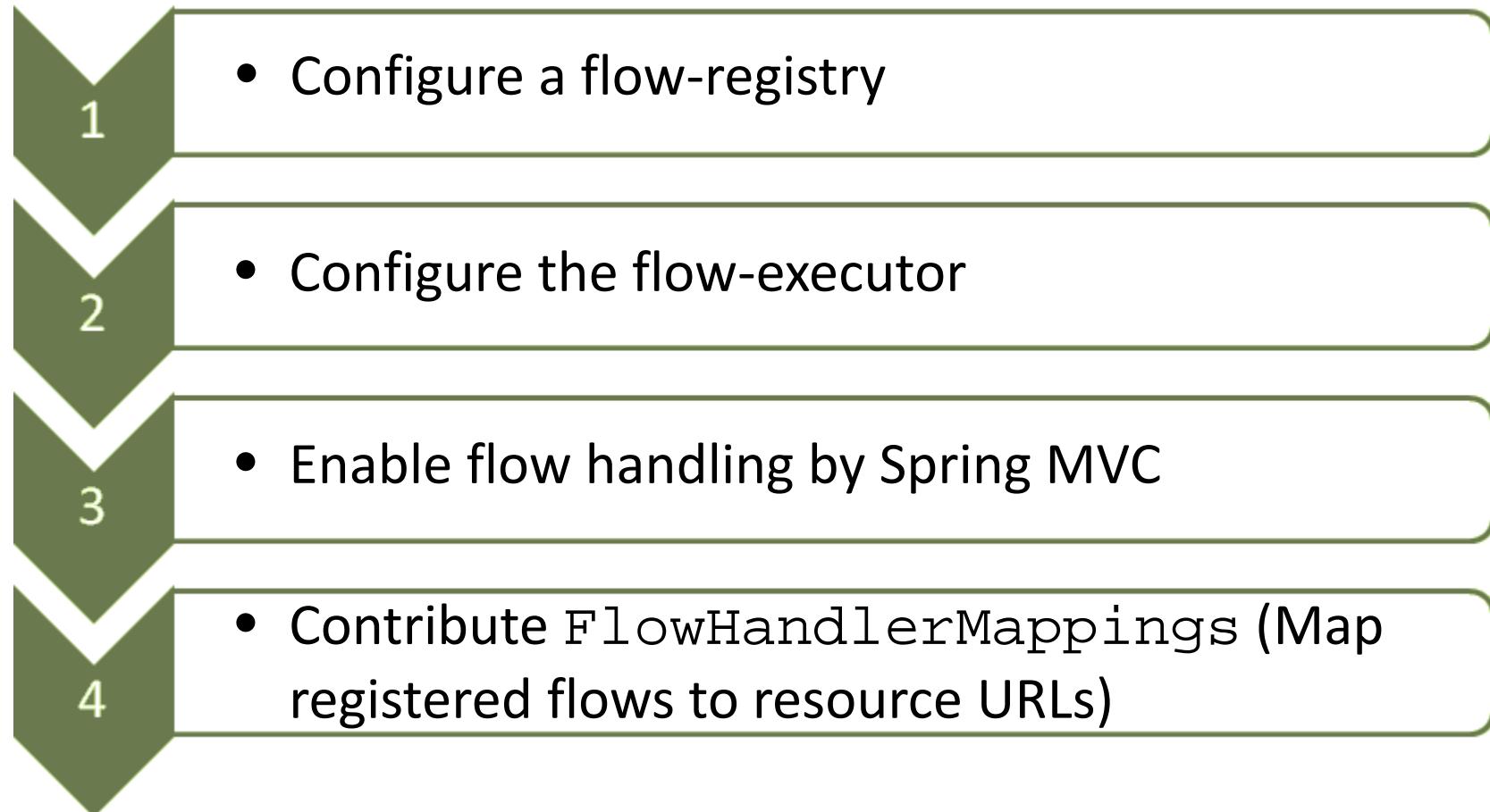
Web Flow Config Schema



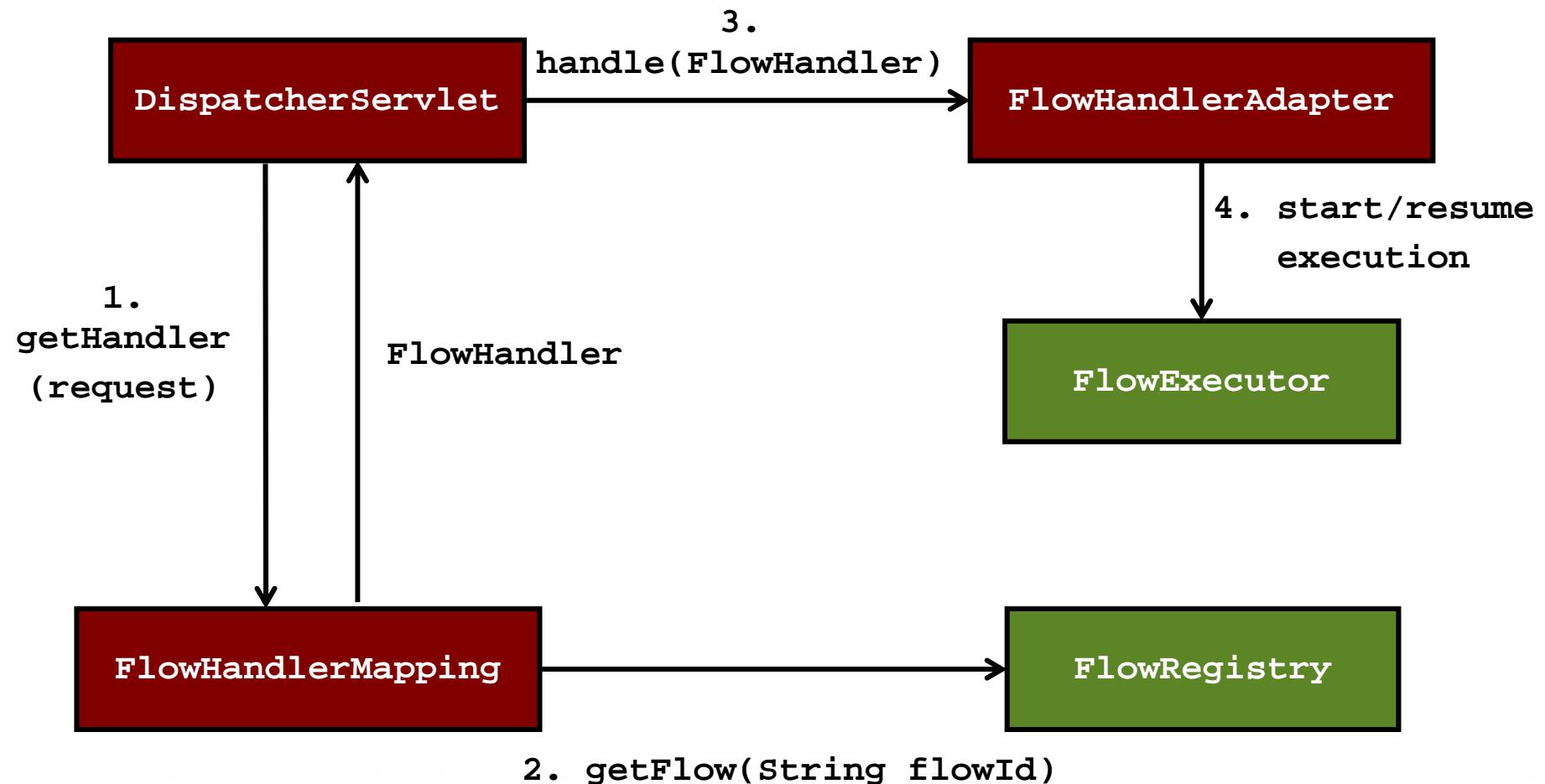
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:webflow="http://www.springframework.org/schema/webflow-
       config"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/webflow-config
           http://www.springframework.org/schema/webflow-config/spring-
           webflow-config-2.0.xsd">

    <!-- Web Flow system configuration elements go here -->
</beans>
```

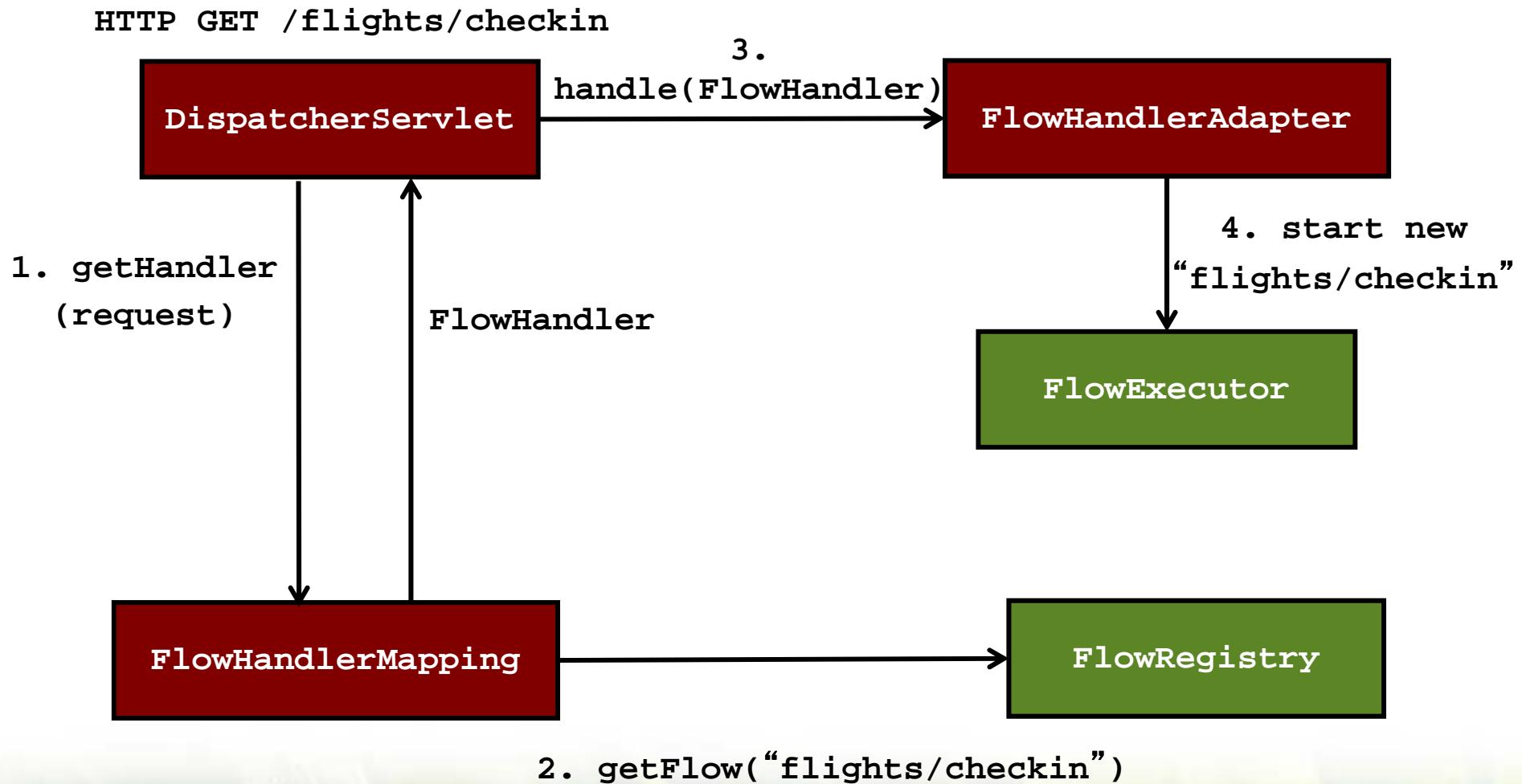
Steps to use with Spring MVC



Spring MVC and Web Flow



Flight Checkin Example





The Flow Registry

- Stores your flow definitions by unique id
- Flows typically stored under /WEB-INF
- Options for registering flows
 - One at a time
 - Using a wildcard pattern
- Can be customized with “flow builder services”



Registering Flow Definitions

- One a time

```
<webflow:flow-registry id="flowRegistry">
    <webflow:flow-location id="flights/checkin"
        path="/WEB-INF/flights/checkin/checkin.xml" />
</webflow:flow-registry>
```

- By using a wildcard pattern
 - The derived flow id is relative to the base path

```
<webflow:flow-registry id="flowRegistry"
    base-path="/WEB-INF/" />
<webflow:flow-location-pattern value="**/*-flow.xml" />
</webflow:flow-registry>
```

Custom Flow Builder Services



- Used to customize the flow registry
- Enable development mode
 - Flow definition changes are detected at runtime
- View factory creator
 - Allows customizing the view technology
- Conversion service
 - Register custom type converters for data binding



Flow Builder Services Example



```
<webflow:flow-registry id="flowRegistry" base-path="/WEB-INF/"  
                      flow-builder-services="flowBuilderServices">  
    <webflow:flow-location-pattern value="**/*-flow.xml" />  
</webflow:flow-registry>  
  
<webflow:flow-builder-services id="flowBuilderServices"  
    view-factory-creator="viewFactoryCreator"  
    conversion-service="conversionService"  
    development="true" />  
  
<bean id="viewFactoryCreator" class="...MvcViewFactoryCreator" />  
    <property name="viewResolvers" ref="tilesViewResolver" />  
</bean>  
  
<bean id="conversionService"  
      class="o.s.binding.convert.service.DefaultConversionService">  
    <constructor-arg ref="myConversionService" />  
</bean>
```

Enable development mode

Register converters

Reuse Spring MVC
view beans



The Flow Executor

- Central entry-point into Web Flow system
 - Manages starting and resuming flows
- Default configuration very simple
- Explicitly configure to observe the execution life-cycle of flows



Flow Executor Example

- Default flow configuration

```
<webflow:flow-executor id="flowExecutor"  
                      flow-registry="flowRegistry" />
```

- With custom flow execution listeners

```
<webflow:flow-executor id="flowExecutor"  
                      flow-registry="flowRegistry">  
    <webflow:flow-execution-listeners>  
        <webflow:listener  
              ref="secureFlowExecutionListener" />  
        <webflow:flow-execution-listeners>  
    </webflow:flow-executor>
```

Remaining steps

- Enable flow handlers within Spring MVC

```
<mvc:annotation-driven/>
<bean class="org.springframework.webflow.mvc
    .servlet.FlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor"/>
</bean>
```

This is still needed for
regular Spring MVC controllers

- Contribute FlowHandlerMappings from registry

```
<bean class="org.springframework.webflow.mvc
    .servlet.FlowHandlerMapping">
    <property name="flowRegistry" ref="flowRegistry"/>
    <property name="order" value="-1" />
</bean>
```

<mvc:annotation-driven> defines
a handlerMapping with id 0

Summary

After completing this lesson, you should have learnt:

- What Web Flow is
- The problems that Web Flow solves
- Web Flow Architecture
- To use Web Flow with Spring MVC





LAB

Getting started with Spring Web Flow



Web Flow Language Essentials

The basics of authoring flows using the flow definition language



Overview

After completing this lesson, you should be able to:

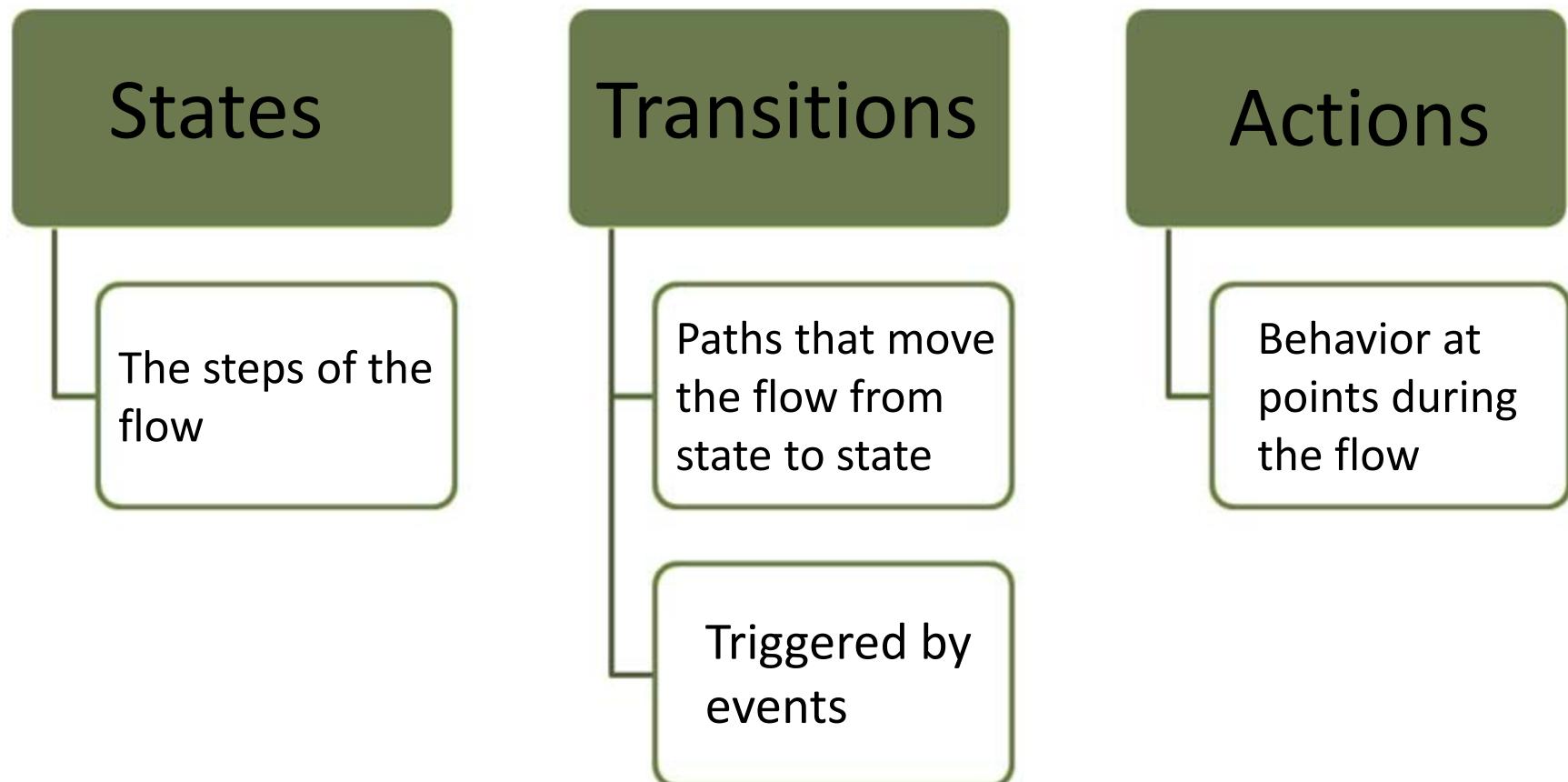
- Describe the typical flow makeup
- Describe Flow packaging guidelines
- Author flows using the XML language
- Test flow execution



Road Map

- Typical flow makeup
- Authoring flows using the XML language
- Testing flow execution

What does a Flow consist of?



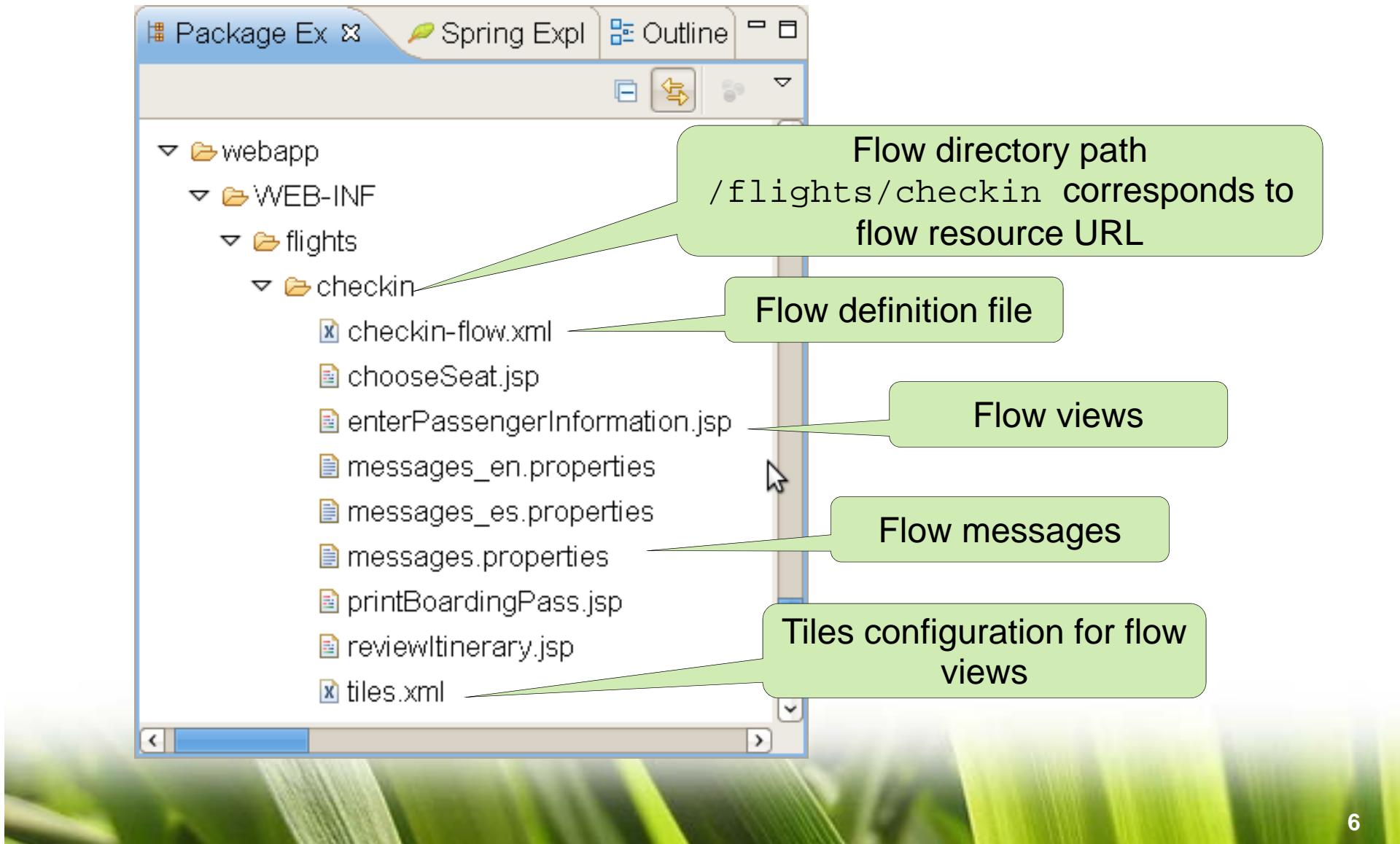
How are Flows authored?



- A flow is defined in a single XML file
 - Called a “flow definition”
 - Encapsulates the entire task algorithm
- Dependent resources co-located with the flow definition
 - Views, messages
 - All stored in a self-contained flow directory



Flow Definition Example



Guide to Authoring Flows



- Implement “flow” logic first
 - Define your states and transitions
- Review your flow with business analysts
 - Often using mock views
- Add “behavior” last
 - Define actions after client is happy with flow





Road Map

- Typical flow makeup
- Authoring flows using the XML language
- Testing flow execution



XML Flow Definition

- Begin with the <flow> element

```
<flow
    xmlns="http://www.springframework.org/schema/webflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/webflow
        http://www.springframework.org/
            schema/webflow/spring-webflow-2.0.xsd">

    <!-- Define states here -->
</flow>
```

- Next add states
 - First state in the file becomes “start state”

View States

- Define a <view-state> to render a view
 - Allows the user to participate in the flow

```
<view-state id="enterPassengerInformation" />
```

- The view-state id resolves to a view template
 - Default resolves to a .jsp in flow working directory
 - e.g. enterPassengerInformation.jsp
- View resolution is pluggable
 - Configurable when setting up system

Transitions

- Define a <transition> to handle a user event
 - May also perform a navigation to another state
 - If no navigation is performed, the view will be refreshed

```
<view-state id="enterPassengerInformation">
    <transition on="findItinerary"
        to="reviewItinerary" />
</view-state>
<view-state id="reviewItinerary" />
```

A black downward-pointing arrow is positioned between the two view-state definitions, indicating the flow of the transition from the first state to the second.

Global Transitions

- Define a <global-transition> for transitions shared by any state in the flow.

```
<flow ...>
    <global-transition on="cancel" to="cancelled" />
    ...
</flow>
```

- Use these sparingly
 - Overuse can make your flow hard to understand



End States

- Define one or more <end-state> to end the flow
 - A logical flow outcome

```
<end-state id="finish" />
```

- By default the flow is restarted
- Flow can optionally send a “final response”
 - Typically a redirect to another resource

Understanding The End State



- **Common question:** How can I display a confirmation page after the end-state is reached?
 - The flow no longer has access flow data!
- **Solution:** Redirect to a stateless confirmation page passing along an id that can be used to display confirmation data



Sending End State Redirects

- Use the <end-state> view attribute to specify the URL to redirect to

```
<end-state id="finish"  
view="externalRedirect:contextRelative  
:/flights/checkin/confirm?id=1" />
```

Add “contextRelative”,
“serverRelative”,
or “servletRelative”

Choose
“externalRedirect:”
or “flowRedirect:”

This *would*
normally be a
variable



Flow Authoring Example

- Flight Check-in Flow
 - Define view and end states
 - Define transitions between states
 - Create mock views
 - Add behavior (next module)



Define View and End States

- Focus on steps and logical outcomes

```
<flow ...>
    <view-state id="enterPassengerInformation" />
    <view-state id="reviewItinerary" />
    <view-state id="chooseSeat" />
    <view-state id="printBoardingPass" />
    <end-state id="finish" />
</flow>
```

Define Transitions

- Define paths users can follow through the flow

```
<flow ...>
    <view-state id="enterPassengerInformation">
        <transition
            on="findItinerary" to="reviewItinerary" />
    </view-state>

    <view-state id="reviewItinerary">
        <transition on="print" to="printBoardingPass" />
        <transition on="viewSeat" to="chooseSeat" />
    </view-state>

    <view-state id="chooseSeat"/>
    <view-state id="printBoardingPass"/>
    <end-state id="finish"/>
</flow>
```



Create Mock Views

- Mock views are basic views that allow users to interact with the flow process
- Contain static data
 - Add dynamic data and behavior later
- Useful for prototyping
- Support usability testing
 - Get front-end right before developing the back-end



Mock JSP

- To signal an event mock views use a naming convention for form buttons: `_eventId_xxx`

```
<form>
    Record Locator: <input type="text"
                           name="recordLocator" />
    ...
    <button name="_eventId_findItinerary" type="submit">
        Find Itinerary
    </button>
</form>
```

enterPassengerInformation.jsp

- A view can also add `_eventId` to HTML links
 - `/flights/checkin?execution=e1s1&_eventId=update`

Mock Facelet (JSF)

- JSF views specify the event in the action attribute of command buttons and links

```
<h:form id="enterPassengerInformation">
    Record Locator: <input type="text"
        name="recordLocator" />
    ...
    <h:commandButton id="findItineraryButton"
        action="findItinerary"
        value="Find Itinerary" />
</h:form>
```

enterPassengerInformation.xml



Road Map

- Typical flow makeup
- Authoring flows using the XML language
- Testing flow execution





Testing Flows

- Every flow should have a unit test
 - Verifies flow logic works as expected
- Web Flow provides JUnit test support
 - Extend from a base test class
- Define a new TestCase class per Flow
 - e.g. FlightCheckinFlowExecutionTests



Example Flow Unit Test – 1

- Provide *file path* to flow definition
- Kick-off flow and ensure it enters start-state

```
public class FlightCheckinFlowExecutionTests
    extends AbstractXmlFlowExecutionTests {

    public FlowDefinitionResource getResource
        (FlowDefinitionResourceFactory factory) {
        return factory.createFileResource
            ("src/main/webapp/WEB-INF/flights/checkin/checkin.xml");
    }

    public void testStartFlightCheckinFlow() {
        MockExternalContext context = new MockExternalContext();
        startFlow(context);
        assertEquals("enterPassengerInformation");
    }
    ... // continued next slide
}
```

Example Flow Unit Test - 2

- Remaining tests
 - Jump to a state
 - Setup flow context
 - ensure any data expected from previous states is there
 - `resumeFlow()`

```
public void testSubmitPassengerInformation() {  
    setCurrentState("enterPassengerInformation");  
    MockExternalContext externalContext =  
        new MockExternalContext();  
  
    externalContext.setEventId("findItinerary");  
    resumeFlow(externalContext);  
    assertEquals("reviewItinerary");  
}
```

Summary

After completing this lesson, you should have learnt:

- That Flows consist of states, transitions, actions
- That Flow definitions are authored in XML
 - Packaged in a self-contained flow directory
- To define “flow” elements before behavior
 - view-state, transition, end-state essential elements
- To Unit test flow logic outside of container





LAB

Authoring a web flow



Web Flow Actions

Adding dynamic behavior to a flow to carry out
business tasks



Overview

After completing this lesson, you should be able to:

- Describe Flow definition best practices
- Common Flow behaviors



Road Map

- Creating scoped objects
- Executing actions
- Data binding
- Validation

Creating Scoped Objects



- Each flow provides a *context* where objects are stored
 - State is managed for you
 - Several “*data scopes*” provided by this context
 - Add objects to these scopes during flow execution
 - Objects cleaned up when they go out of scope
- Explicit variable declaration allocates a scoped object
 - *Flow* and *View* scope using `<var>`
- Can also assign variables dynamically using actions
 - Assignment in *any* scope possible
 - See `<evaluate>` discussed later



Web Flow Scopes

Flow

- Lasts for the life of the flow

View

- Associated with each view-state

Request

- Lasts a single request

Flash

- Cleaned up after the next view is rendered

Conversation

- Global execution scope (shared by all subflows)

Flow Variables - 1

- Use `<var>` tag at flow level
- Variable created automatically
 - Must have a default or `@Autowired` constructor
 - May have `@Autowired` property setters too

```
<flow>
<var name="timeNow" class="java.util.Date">
<var name="paxInfo" class="flights.checkin.PassengerInfo" />
```

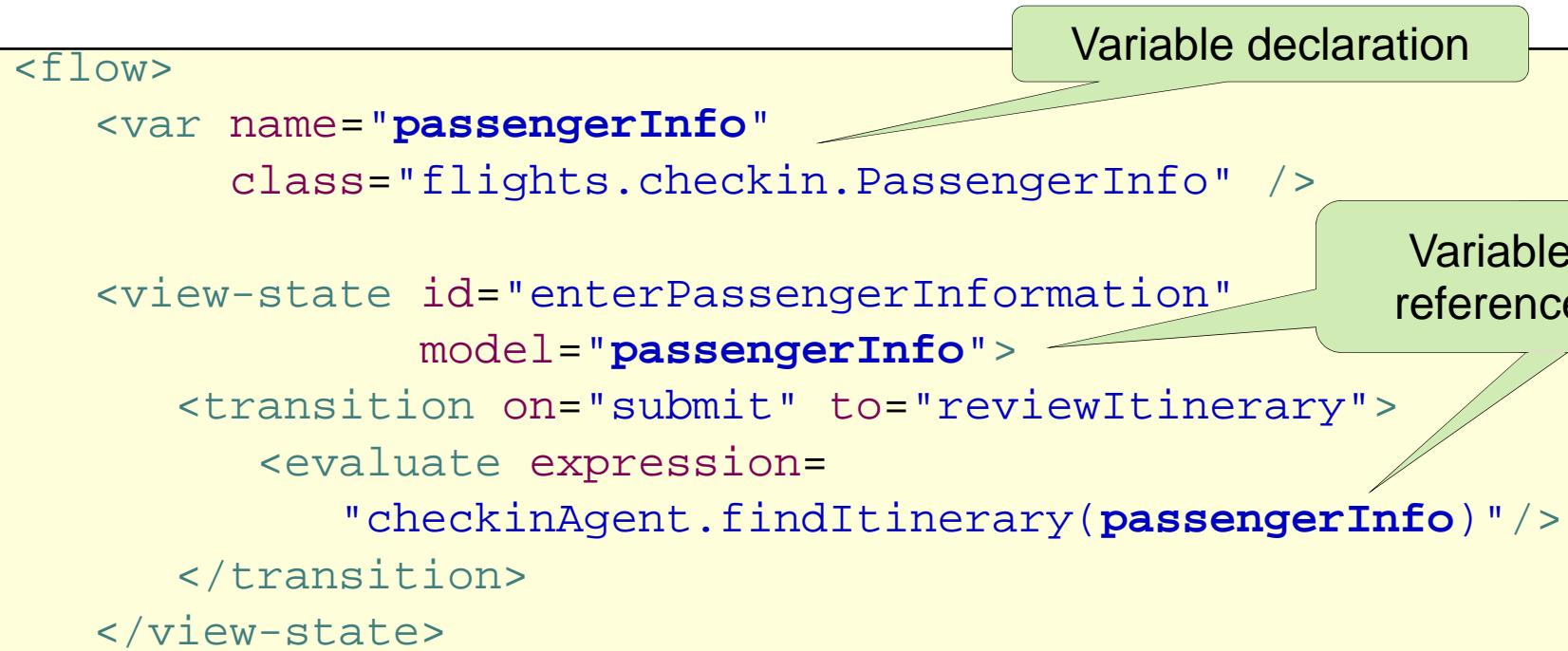
```
public class PassengerInfo implements Serializable {
    @Autowired
    public PassengerInfo(BookingService bookingSvc) { ... }
}
```

Flow Variable - 2

- Often used as the data model for views
- Often passed to services invoked by the flow

```
<flow>
    <var name="passengerInfo"
        class="flights.checkin.PassengerInfo" />

    <view-state id="enterPassengerInformation"
        model="passengerInfo">
        <transition on="submit" to="reviewItinerary">
            <evaluate expression=
                "checkinAgent.findItinerary(passengerInfo)" />
        </transition>
    </view-state>
```



The diagram illustrates the usage of a flow variable named `passengerInfo`. It shows two instances of this variable: one in a `<var>` declaration and another in a `<model>` attribute of a `<view-state>`. A callout bubble labeled "Variable declaration" points to the first instance, and another labeled "Variable reference" points to the second instance.



Conversation Variables

- Like flow scope variables
 - But available to any sub-flows too
 - Similar to global variables: *with all the dangers*
- Be careful
 - Introduces a dependency between flow and sub-flow
 - Sub-flows accept input/output parameters: a better option

```
<evaluate result="conversationFlow.itinerary"  
expression="checkinAgent.findItinerary(passengerInfo)"/>
```



View Variables - 1

- Declare using the view-state <var> tag
 - Convenient way to allocate a view-scoped object
- Created when entering the view-state
 - May also be @Autowired with dependencies
- Referenced by expressions
 - Often used as the data model for a single view
 - Often updated over a series of Ajax requests



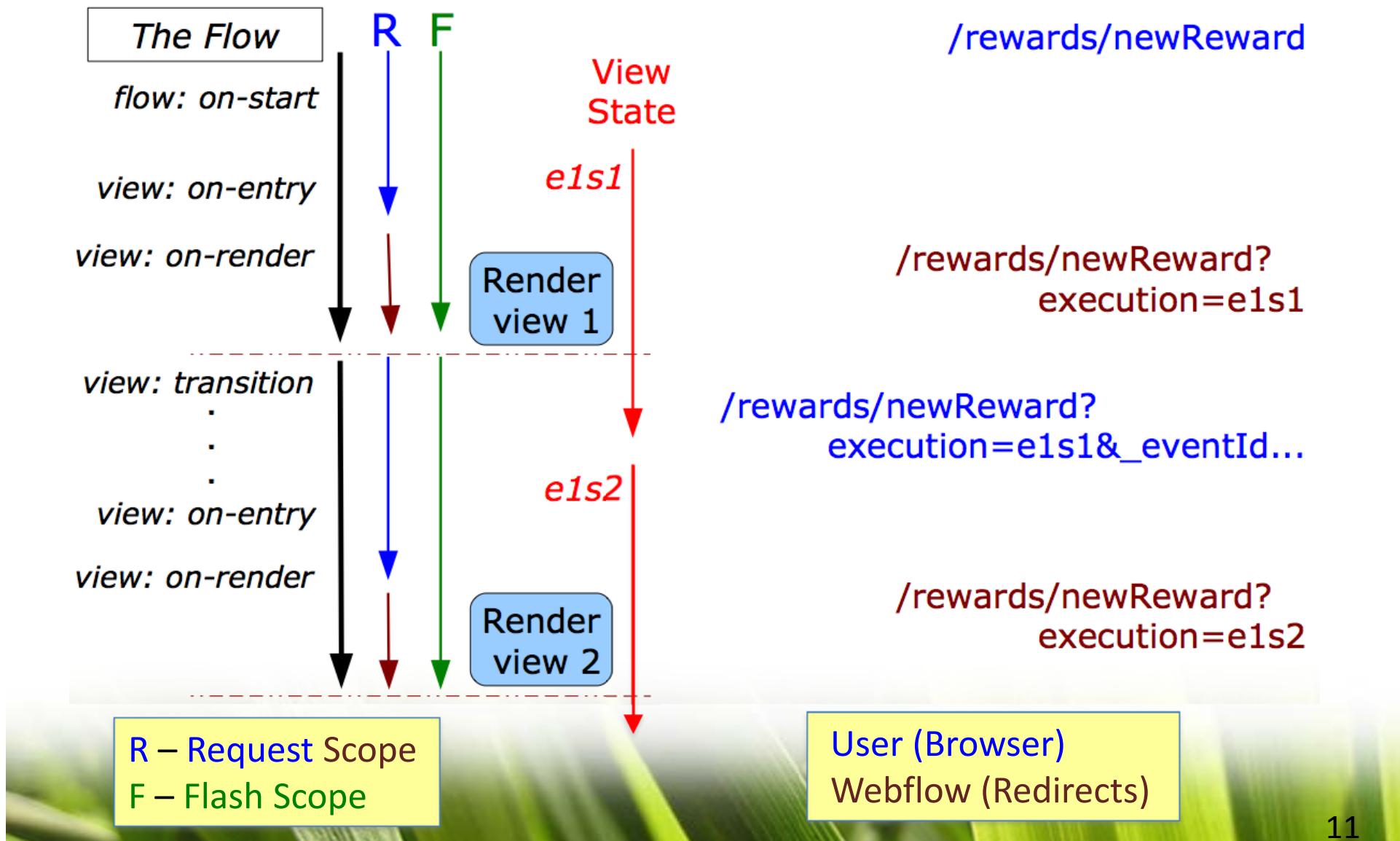
View Variables - 2

```
<view-state id="enterPassengerInformation"
            model="passengerInfo">
    <var name="passengerInfo"
         class="flights.checkin.PassengerInfo" />
    <transition on="submit" to="reviewItinerary">
        <evaluate expression=
                    "checkinAgent.findItinerary(passengerInfo)" />
    </transition>
</view-state>
```

Variable declaration

Variable Reference

Request vs. Flash Scope



Flash Scope

- Flash Scope
 - Lasts for the entire flow
 - But is cleared each time a view renders
- Effective scope is
 - Transition from current view *to* end of render of next view
- Used to pass data from one state to the next
 - Especially request/view scoped data
 - Very useful scope



Request Scope

- Two requests per interaction
 - *First:* transition from current state *to* on-entry of next state
 - *Second:* From on-render of next state *to* end of render of same state
- Typical Usage
 - Data that must be refetched every time a state is redisplayed
 - If data can be cached, make it view scoped
 - Data needed to initialize the next state, but not displayed by it
- Generally not that useful
 - Flash scope is easier



Road Map

- Creating scoped objects
- Executing actions
- Data binding
- Validation



Actions

- Actions execute behavior at specific points within the flow
- Several points where actions can be executed
 - on startup, before rendering, on transition, etc
- Flow actions often act on Java objects
 - Invoke Spring-managed services
 - Assign variables



Writing an expression



- evaluate
 - Evaluate an expression
 - Can *prevent* a transition
- set
 - Set the value of a variable
 - *Never prevent* a transition, *regardless* of variable value



Evaluate Action

- Use `<evaluate>` to evaluate an expression

```
<evaluate result="flowScope.itinerary"  
expression="checkinAgent.findItinerary(passengerInfo)" />
```

- Evaluate any object in flow context
 - Resolve properties on scoped beans
 - Invoke methods
- Can also assign result of evaluation
 - e.g. save result in flow scope
 - Optionally perform a result type conversion



Expression Language (EL)



- Actions are expressed in EL
 - Concise syntax for executing behavior
 - Access object properties, invoke methods, etc.
- Supported EL implementations
 - Spring EL (recommended default)
 - Unified EL Standard
 - OGNL





Spring EL Syntax

- <variable>[.property]
 - Variable *resolved* against current flow context
 - Property can be JavaBeans property or method
- Variable resolution algorithm
- Search reserved variables first
 - Search web flow scopes second
 - request/flash/view/flow/conversation
 - Search flow application context third
 - Resolve Spring beans

Spring EL Reference: <http://tinyurl.com/spel-reference>

Generic SpEL Examples



- `flowScope.foo`
 - Resolve 'foo' in flow scope
- `foo`
 - Search all scopes looking for 'foo'
- `foo.bar()`
 - Invoke method 'bar' on foo
- `foo.bar(baz)`
 - Invoke method bar passing in variable 'baz'



Reserved SpEL Variables

- conversationScope
- flowScope
- viewScope
- requestScope
- flashScope
- flowExecutionUrl
- flowRequestContext
- requestParameters
- currentEvent
- currentUser
- externalContext
- messageContext
- resourceBundle

Note: no **sessionScope**. Use **externalContext.sessionMap**



Reserved EL Variables in JSP

- Only the following are valid in a JSP page
 - Any variables in scope
 - requestScope, viewScope , flashScope
 - flowExecutionUrl = url of current page
 - flowRequestContext – the RequestContext
- This is in addition to usual EL variables
 - sessionScope, pageScope, params, etc

For anything else, use **flowRequestContext**
– **flowRequestContext.currentState.id**



Evaluate Action Examples

- Invoke a method on bean

```
<evaluate expression=
    "checkinAgent.update(passengerInfo)" />
```

- Assign result to view or flow scope variable

```
<evaluate expression=
    "checkinAgent.findItinerary(passengerInfo)"
    result="flowScope.itinerary" />
```

- Convert result to another type

```
<evaluate expression="bookingAgent.find(searchCriteria)"
    result="flowScope.hotelsFound" result-type="long" />
```

string, short, integer, int, byte, long, float, double,
bigInteger, bigDecimal, boolean, a classname

Set Action

- Assign a *new* variable in a scope
 - *Must* specify a scope, no default

```
<set name="flowScope.accountNumber"  
      value="requestParameters.accountNumber" />
```

- Update the value of an existing variable
 - Or one of its properties
 - Scope optional: scope resolution algorithm used

```
<set name="searchCriteria.sortBy"  
      value="requestParameters.sortBy" />
```

Places to Invoke Actions From



- When a flow starts/ends

```
<flow>
  <on-start>
    <evaluate expression="..." />
  </on-start>
</flow>
```

```
<flow>
  <on-end>
    <evaluate expression="..." />
  </on-end>
</flow>
```

- When a state is entered/exited

```
<view-state>
  <on-entry>
    <evaluate
      expression="..." />
  </on-entry>
</view-state>
```

```
<view-state>
  <on-exit>
    <evaluate
      expression="..." />
  </on-exit>
</view-state>
```

Actions During Transitions

- Prior to a transition

```
<transition on="modify" to="searchForm">  
    <evaluate expression="...">  
</transition>
```

Warning: only
one evaluate per
transition.

- Transition can be prevented by `<evaluate/>` action
 - If *expression* returns false or throws an exception
 - Only one evaluate is allowed
 - Recall: `<set/>` never prevents a transition
 - Use as many sets as you want



Actions Before Rendering



- Executed before a view is rendered

```
<view-state id="enterPassengerInformation">
    <on-render>
        <evaluate expression="...">
    </on-render>
</view-state>
```

- Executed again on every browser refresh
 - contrast to on-entry

Actions and Scope

- Data for current view
 - *flash1, flash2, req2, all* – visible on `enterInfo` page
 - *req1* lost – *no point doing this*
 - *flash1* lost if view refreshed

```
<view-state id="enterInfo">
  <on-entry>
    <set name="flashScope.flash1" value="'flash1'" >
    <set name="requestScope.req1" value="'req1'" >
  <on-entry>
  <on-render>
    <set name="requestScope.all" value="infoService.findAll()" >
    <set name="flashScope.flash2" value="'flash2'" >
    <set name="requestScope.req2" value="'req2'" >
  </on-render>
</view-state>
```

Or use Flash Scope

Request Parameters

- All the parameters from URL query-string
 - For the current request only
 - Lost during Web Flow redirection
 - These parameters are *not* in the `requestScope`
- Assign to a new scope to keep them
 - Cannot be set/modified manually

```
<view-state id="reviewBooking">
    <transition on="cancel" to="cancelled">
        <evaluate expression=
            "checkinAgent.cancel(requestParameters.bookingId)">
        </transition>
    </view-state>
```

Best Practices for Actions



- A flow definition is “Web layer glue”
 - navigation, validation, form setup
 - invocation of middle-tier services
- Keep the number of actions to a minimum
 - avoid “programming” in the flow
- Delegate to Java for more extensive work





Road Map

- Creating scoped objects
- Executing actions
- Data binding
- Validation



View Model Object

- Data binding and validation
 - a common need for views
- A view can be associated with a model object

```
<view-state id="enterPassengerInformation"  
           model="passengerInfo" />
```

- The model object can be in different scopes
 - flow scope, view scope, ...

Building the View

- Fully compatible with the Spring form tags

```
<form:form modelAttribute="passengerInfo">
    <form:input path="recordLocator" />

    <input type="submit" value="Submit" />
    <input type="hidden" name="_eventId" value="submit" />
</form:form>
```

Submitting the Form

- Automatic data binding + validation
- Can be suppressed

```
<view-state id="enterPassengerInformation"
            model="passengerInfo">
    <transition on="submit" to="reviewItinerary" />
    <transition on="cancel" to="end"
                bind="false" validate="false"/>
</view-state>
```

- Flow remains in same view on failure

Restricting Automatic Binding

- Not all fields should be bound to
 - fields which are read but not written by the end-user
 - application security can be compromised
- 'binder' white-lists elements eligible for binding

```
<view-state id="enterDiningInformation"  
model="diningForm">  
    <binder>  
        <binding property="amount" required="true" />  
        <binding property="creditCardNumber"  
               required="true" />  
        ...  
    </binder>  
</view-state>
```



Indicates a required field



Customizing Type Conversion

- Spring 3 Formatters are used for Web Flow type conversion
 - Same approach as in Spring MVC
- Custom formatters defined for Spring MVC are *not* automatically defined for Web Flow as well
 - Must be explicitly associated using flow builder services
 - Default formatters are present in both “out of the box”





Registering Custom Converters

```
<mvc:annotation-driven conversionService="converter" />

<bean id="converter" class=".DefaultConversionService" />
```

mvc-config.xml

```
<webflow:flow-registry id="flowRegistry"
    flow-builder-services="flowBuilderServices">
    <webflow:flow-location
        path="/WEB-INF/flights/checkin/checkin.xml" />
</webflow:flow-registry>

<webflow:flow-builder-services id="flowBuilderServices"
    conversion-service="converter" />
```

Customizing Data Binding Error Messages



- Web Flow uses a different strategy for generating typeMismatch and required error codes
- The error code is appended to the end and not at the beginning

```
typeMismatch  
amount.typeMismatch  
diningForm.amount.typeMismatch  
common.money.MonetaryAmount.typeMismatch
```

```
required  
amount.required  
diningForm.amount.required  
common.money.MonetaryAmount.required
```



Customizing Data Binding Error Messages (2)

- To use same error code strategy for MVC and Webflow, add a DefaultMessageCodesResolver

```
<bean id="viewFactoryCreator"
      class="...MvcViewFactoryCreator">
    <property name="viewResolvers" ref="tilesViewResolver" />
    <property name="messageCodesResolver ref="mcr" />
</bean>

<bean id="mcr" class="...DefaultMessageCodesResolver" />
```

webflow-config.xml



Road Map

- Creating scoped objects
- Executing actions
- Data binding
- Validation



Validation Options

- JSR 303 validation support
 - annotate the form object
 - automatically used for validation
 - *but:* you must enable this
- Custom validator
 - well-known method or class
 - automatically invoked
- Can use either, both or neither



JSR 303 Validation



- Enable same validator as MVC
 - via Web Flow builder services

Requires Web Flow 2.3 or above

```
<webflow:flow-builder-services  
    id="flowBuilderServices" validator="validationService"/>  
  
<bean id="validationService"  
    class="...validation.beanvalidation.LocalValidatorFactoryBean">  
</bean>
```

```
public class AccountSearchCriteria {  
    @NotEmpty  
    private String searchString = "";  
    @Min(0) @Max(100)  
    private int maximumResults = 10;  
}
```

Validation Via Model Object



- Method “discovered” by convention

- `validate${view-state id}(ValidationContext context)`

```
<view-state id="searchForm"
            ...
</view-state>
```

```
public class AccountSearchCriteria {
    public void validateSearchForm
        (ValidationContext context) {
        ...
    }
}
```

Note: Different methods for different forms (views).
Cannot do this with JSR 303 annotations.

Validation with Validator

- Validator class “discovered” by convention
 - Bean name: **`#{model object name}validator`**
 - If you didn't write/can't modify the form model object

```
<view-state id="searchForm"
            model="accountSearchCriteria" >
    ...

```

```
@Component( "accountSearchCriteriaValidator" )
public class AccountSearchCriteriaValidator {

    public void validateSearchForm(
        AccountSearchCriteria target,
        ValidationContext context) {
    }
}
```

Method automatically passed the
model object

Manual Validation using evaluate



- If you don't like using our conventions
 - Validate using an `<evaluate>` action
 - If validator returns `false`, transition is prevented

```
<view-state id="enterBooking" model="booking">
    <transition on="submit" to="confirmBooking">
        <evaluate expression=
            "bookingValidator.check(booking, messageContext)">
        </transition>
    </view-state>
```

```
public class BookingValidator {
    public boolean check(Booking booking,
                        MessageContext context) {
        ...
    }
}
```

Registering Errors

- Using an Errors object
 - commonly used in Spring MVC

```
errors.rejectValue("creditCardNumber",
    "creditCard.invalidNumber",
    "A credit card number must have 16 digits.");
```

- Using a ValidationContext

```
context.getMessageContext().
    addMessage(new MessageBuilder().error()
        .source("creditCardNumber")
        .code("creditCard.invalidNumber")
        .defaultText
        ("A credit card number must have 16 digits.").build());
```

- Both can be used in validate method signatures



Summary

After completing this lesson, you should have learnt:

- Flow definition best practices
- Common Flow behaviors





LAB

Adding behavior to flows



Web Flow Actions

Implementing Dynamic Control Flow Using Action States



Overview

After completing this lesson, you should be able to:

- Describe Branching
- Describe Action State
- Describe Different Types of Actions
- Describe Decision State
- Understand Exception Handling



Road Map

- Branching
- Action State
- Different Types of Actions
- Decision State
- Exception Handling





Branching

- Flow navigation logic is often dynamic
 - e.g. If <x> then go <y>, else go <z>
- view-states only go so far for these cases
 - You can prevent the user from leaving a view because an action fails
 - But you cannot easily change the target state to transition to based on some condition
- action-state and decision-state types exist for implementing dynamic navigation rules





Road Map

- Branching
- Action State
- Different Types of Actions
- Decision State
- Exception Handling



Action State

- Allows you to transition to a state that executes one or more actions
 - <evaluation/>, <set/>
- The action's result is *raised* as an event
 - <evaluation/> only
 - <set/> always returns true
- This event triggers a transition to another state
 - Different events can be raised to trigger different transitions dynamically



Action State (2)

- Several result types possible
 - Always converted to a String
- Options are:
 - Booleans (always evaluate to “yes” or “no”)
 - Will not match “true” or “false”
 - Strings
 - Enumerated types (converted to a String)
 - org.springframework.webflow.execution.Event
 - transition uses event's id String
 - Any other value evaluates to “success”



Action State Examples

- Transition directly on the return value:

```
<action-state id="bookingFound">
    <evaluate
        expression="checkinService.bookingExists(paxInfo)"/>
    <transition on="no" to="enterPassengerIdentification"/>
    <transition on="yes" to="reviewItinerary"/>
</action-state>
```

NOTE: boolean
return types evaluate
to "yes" or "no"

- Use a set and test if necessary:

```
<action-state id="bookingFound">
    <set name="flowScope.booking"
        value="checkinService.findBooking(paxInfo)"/>
    <evaluate expression="booking != null"/>
    <transition on="no" to="enterPassengerIdentification"/>
    <transition on="yes" to="reviewItinerary"/>
</action-state>
```

Transition Action Examples

- A transition occurs if its `<evaluate>` succeeds
 - Result of evaluation can be
 - Boolean: true
 - String: “yes”, “true”, “success”
 - Enumerated values converted to Strings
 - Any other object treated as “success”

**Strings are
case-sensitive**

```
<view-state id="enterPassengerIdentification" model="paxInfo">  
  ...  
  <transition on="submit" to="reviewItinerary">  
    <evaluate expression="checkinService.bookingExists(paxInfo)" />  
  </transition>  
</view-state>
```

Method must return: `true`,
“yes”, “success” or an object

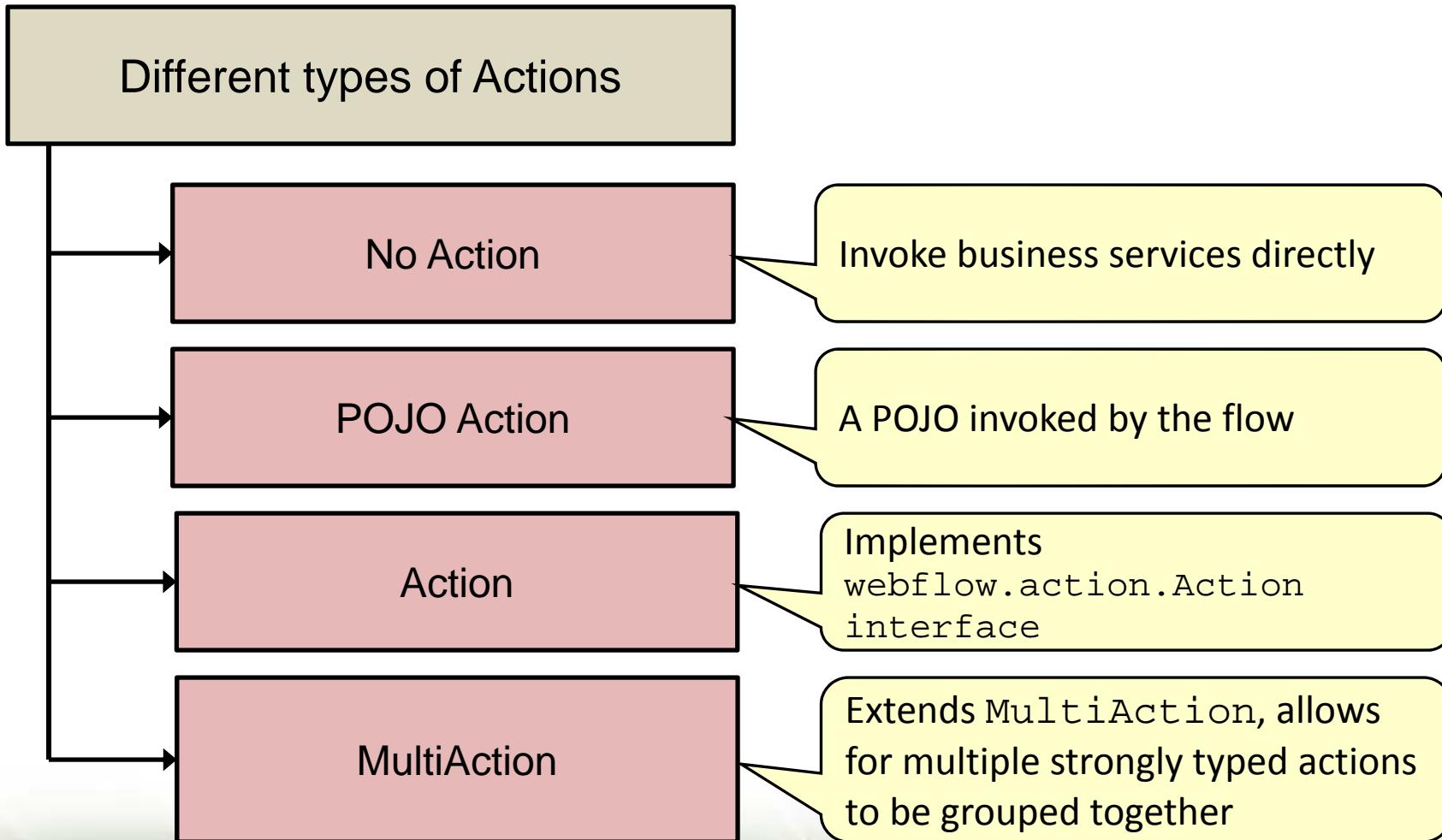


Road Map

- Branching
- Action State
- Different Types of Actions
- Decision State
- Exception Handling



Different Types of Actions



No Action

- The approach you've seen so far
 - Transition on result of method call on a business-level service
 - Methods can be injected with request context and/or flow variables



No Action Example

```
<action-state id="performBooking">
    <evaluate expression="bookingService.make(booking)"/>
    <transition on="success" to="bookingComplete"/>
</action-state>
```

Transition **always**
occurs

```
@Service("bookingService")
public class BookingServiceImpl implements BookingService {
    public BookingConfirmation make(Booking booking) {
        BookingConfirmation confirmation;
        ...
        return confirmation;
    }
}
```

Remember, most return types
evaluate to "success"

POJO Action

- A variation on what you've seen so far
 - Transition on result of method call on a display-layer POJO
 - Typically a web-flow object or a spring-bean
- Methods
 - Can have any sensible signature
 - Any variables in scope can be passed
 - Can also be injected with request context
 - Hence context is managed explicitly by you
 - Must return a String to transition on



POJO Action Example

```
<action-state id="performBooking">
    <evaluate expression=
        "bookingAction.book(booking, flowRequestContext)"/>
    <transition on="success" to="bookingComplete"/>
</action-state>
```

```
@Component
public class BookingAction {
    public String book(Booking booking, RequestContext
context) {
        BookingConfirmation confirmation
            = bookingService.make(booking);
        ...
        return "success";
    }
}
```

Context passed
explicitly by you

Plain Action

- Create a simple class for each action
- Two ways to create
 - Implement Action interface
 - Extend AbstractAction class

```
package org.springframework.webflow.execution;  
  
public interface Action {  
    public Event execute(RequestContext context)  
        throws Exception;  
}
```

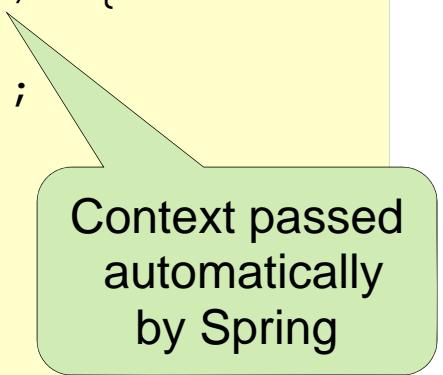
Context passed
automatically by Spring



Action Example

```
<action-state id="performBooking">
    <evaluate expression="bookingAction"/>
    <transition on="success" to="bookingComplete"/>
</action-state>
```

```
@Component
public class BookingAction implements Action {
    public Event execute(RequestContext context) {
        Booking booking = (Booking)
            context.getFlowScope().get("booking");
        BookingConfirmation confirmation
            = bookingService.make(booking);
        ...
        return new Event(this, "success");
    }
}
```



Context passed automatically by Spring

MultiAction

- Subclass MultiAction class
 - Allows many action methods on one class
 - Methods must have the same signature

Context passed automatically by Spring

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

- Helper methods provided for returning Events
 - success(), error(), yes(), no()
 - result(), result(String),
error(Exception)

MultiAction Example

```
<action-state id="performBooking">
    <evaluate expression="bookingAction.makeBooking" />
    <transition on="success" to="bookingComplete" />
</action-state>
```

Context passed
automatically by Spring

```
@Component
public class BookingAction extends MultiAction {
    public Event makeBooking(RequestContext context) {
        Booking booking = (Booking)
            context.getFlowScope().get("booking");
        BookingConfirmation confirmation
            = bookingService.make(booking);
        ...
        return success();
    }
}
```

When to use which action?

No Action

- Looks good on paper, allows a flow to invoke your business logic directly
- However, tighter coupling between your flow and your business logic can bloat your flow definition

POJO Action

- Commonly used to glue your flow logic with your business layer in a decoupled way
- Allows for flexibility to encapsulate the handling of business-layer exceptions, for example
- Typically results in slightly more complex flow definitions

When to use which action? (2)

Plain Action

- Rarely used directly

Multi Action

- Commonly used to glue your flow logic with your business layer in a decoupled way
- Allows for flexibility to encapsulate the handling of business-layer exceptions, for example
- Typically result in slightly more complex Action code

Suggestion

- Use either POJO Action or MultiAction
- Weigh complexity of flow definition vs. complexity of Action

RequestContext

- Passed to any of the action types
 - Provides access to all data currently in scope
 - `getFlowScope()` - can read and write flow data
 - Info about the flow itself
 - `getActiveFlow()`, `getCurrentState()`,
`getFlowExecuteURL()`
 - Access to the message-context
 - `getMessageContext()`
 - Info about the calling context and how the flow was invoked
 - `getExternalContext()`





Road Map

- Branching
- Action State
- Different Types of Actions
- Decision State
- Exception Handling



Decision State

- Similar to an action state
 - For explicit state transition control
- Provides a convenient if/then syntax for evaluating a boolean expression
 - Boolean return value determines next state to go to
 - For more than one condition, use an action state

```
<decision-state id="isValidBooking">
    <if test="checkinService.bookingExists(passengerInfo)"
        then="reviewItinerary"
        else="enterPassengerIdentification"/>
        <!-- Try again -->
</decision-state>
```



Road Map

- Branching
- Action State
- Different Types of Actions
- Decision State
- Exception Handling



Exception Handling (1)

- Actions often invoke services that may throw checked Business Exceptions
 - How should the flow handle these exceptions?
- *Option One*
 - Let the exception propagate
 - Have the XML flow definition handle it in a transition

```
<transition on-exception="accounts.NoSuchAccountException"  
          to="endCancel" />
```

(2) Global Transitions

- Option Two
 - Declare exception handling globally
 - Place at *end* of flow
 - Exception can be thrown from any state in the flow

```
<flow>
  ...
  <global-transitions>
    <transition
      on-exception="accounts.NoSuchAccountException"
      to="endCancel"  />
  </global-transitions>
</flow>
```

(3) Handle in Action Class

- Using an `ExceptionHandler` is not as clean
 - Can lead to XML bloat
- Option Three
 - Catch exception in an Action and return an error event

```
public class BookingAction extends MultiAction {  
    public Event makeBooking(RequestContext context) {  
        try {  
            Booking booking = ...  
            return success();  
        } catch (SomeException e) {  
            return error(e);  
        }  
    }  
}
```

A callout bubble points from the text "Available as \$exception in event-context" to the variable "e" in the catch block of the code.

Available as `$exception` in event-context

// Optionally pass exception

Summary

After completing this lesson, you should have learnt:

- What Branching is
- About Action State
- The Different Types of Actions
- The Decision State
- How Exceptions are Handled





LAB

Implementing Dynamic Control Flow Using Action States



Web Application Security with Spring

Addressing Common Web Application Security
Requirements





Overview

After completing this lesson, you should be able to:

- Describe the High-Level Security Overview
- Describe Motivations of Spring Security
- Understand Spring Security in a Web Environment
- Configure Web Authentication
- Use Spring Security's Tag Libraries
- Describe Method security



Road Map

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security

Security Concepts

Principal

- User, device or system that performs an action

Authentication

- Establishing that a principal's credentials are valid

Authorization

- Deciding if a principal is allowed to perform an action

Secured item

- Resource that is being secured

Authentication

- Many authentication mechanisms
 - basic, digest, form, X.509
- Many options to store credentials
 - database, LDAP, in-memory (development)



Authorization



- Authorization depends on authentication
 - Before deciding if a user can perform an action, user identity must be established
- The decision process is often based on roles
 - ADMIN can cancel orders
 - MEMBER can place orders
 - GUEST can browse the catalog





Road Map

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security





Motivations: Portability

- Servlet-Spec security is not portable
 - Requires container specific adapters and role mappings
- Spring Security is portable across containers
 - Secured archive (e.g. WAR) can be deployed as-is
 - Also runs in standalone environments





Motivations: Flexibility

- Supports all common authentication mechanisms
 - Basic, Form, X.509, Cookies, Single-Sign-On, etc.
- Provides configurable storage options for user details (credentials and authorities)
 - RDBMS, LDAP, Properties file, custom DAOs, etc.
- Uses Spring for configuration



Motivations: Extensibility



- Security requirements often require customization
- With Spring Security, all of the following are extensible
 - How a principal is defined
 - Where authentication information is stored
 - How authorization decisions are made
 - Where security constraints are stored



Motivations: Separation of Concerns



- Business logic is decoupled from security concerns
 - Leverages Servlet Filters and Spring AOP for an interceptor-based approach
- Authentication and Authorization are decoupled
 - Changes to the authentication process have no impact on authorization



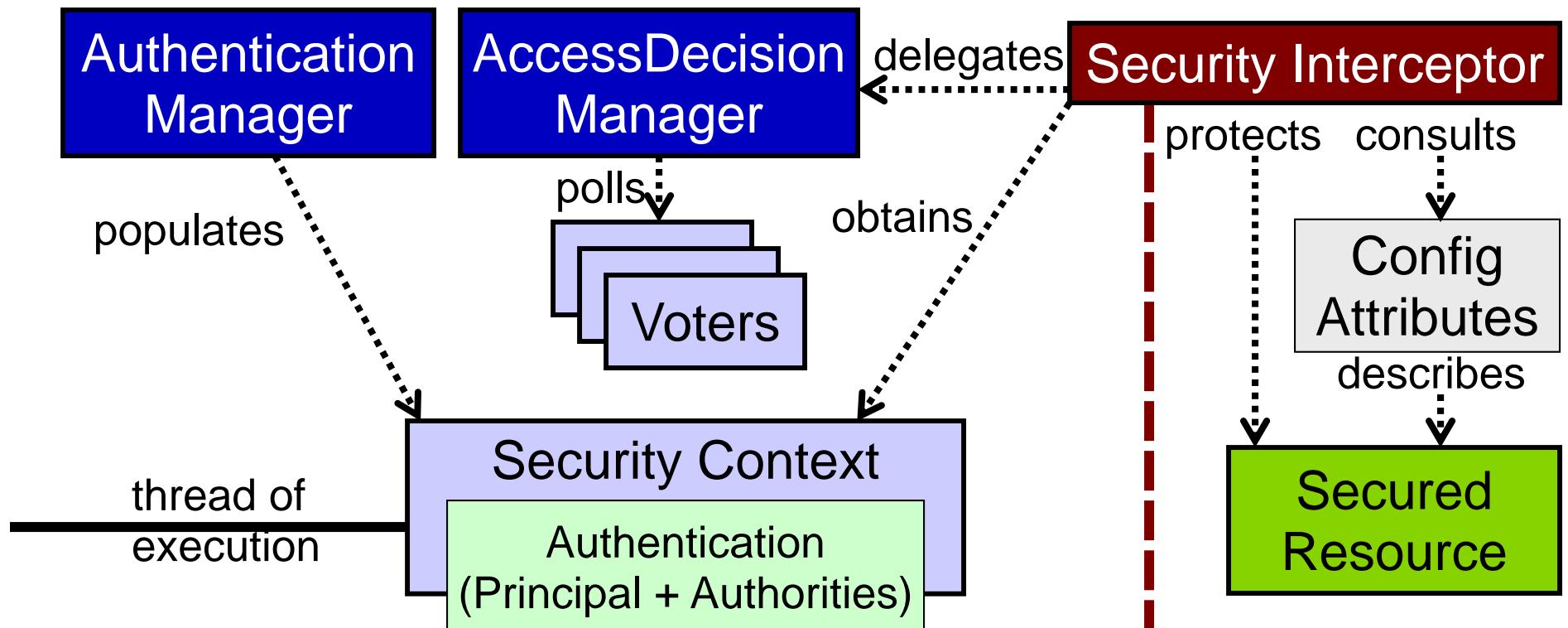
Motivations: Consistency



- The goal of authentication is always the same regardless of the mechanism
 - Establish a security context with the authenticated principal's information
- The process of authorization is always the same regardless of resource type
 - Consult the attributes of the secured resource
 - Obtain principal information from security context
 - Grant or deny access



Spring Security: the Big Picture





Road Map

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security



Configuration in the Application Context



- Spring configuration
- Using Spring Security's "Security" namespace

Match all URLs starting with
/accounts/ (ANT-style path)

```
<beans ...>
  <security:http>
    <security:intercept-url pattern="/accounts/**"
        access="IS_AUTHENTICATED_FULLY" />
    <security:form-login login-page="/login.htm" />
    <security:logout />
  </security:http>
</beans>
```

Spring Configuration File

Configuration in web.xml

- Define the single proxy filter
 - `springSecurityFilterChain` is a mandatory name
 - refers to an existing Spring bean with the same name

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

intercept-url

- intercept-urls are evaluated in the order listed
 - the first match will be used
 - specific matches should be put on top

```
<beans ...>
  <security:http>
    <security:intercept-url
      pattern="/accounts/edit*" access="ROLE_ADMIN" />
    <security:intercept-url
      pattern="/accounts/account*" access="ROLE_USER" />
    <security:intercept-url pattern="/accounts/**"
      access="IS_AUTHENTICATED_FULLY" />
  </security:http>
</beans>
```



Syntax available from Spring Security 2.0

Security EL expressions

- `hasRole('role')`
 - Checks whether the principal has the given role
- `hasAnyRole('role1', 'role2', ...)`
 - Checks whether the principal has any of the given roles
- `isAnonymous()`
 - Allows access for unauthenticated principals
- `isAuthenticated()`
 - Allows access for authenticated or remembered principals



Available from Spring Security 3.0
Previous syntax still works in Spring Security 3.0

intercept-url and Expression Language

- Expression Language provides more flexibility
 - Many built-in expressions available

Expression Language needs to be enabled explicitly

```
<beans ...>
    <security:http use-expressions="true">
        <security:intercept-url pattern="/accounts/edit*"
            access="hasRole('ROLE_ADMIN')"/>
        <security:intercept-url pattern="/accounts/account*"
            access="hasRole('ROLE_USER')"/>
        <security:intercept-url pattern="/accounts/**"
            access="isAuthenticated() and
            hasIpAddress('192.168.1.0/24')"/>
    </security:http>
</beans>
```

Spring configuration file

Working with roles

- Checking if the user has one single role

```
<security:intercept-url pattern="/accounts/update*"  
access="hasRole('ROLE_ADMIN')"/>
```

- “or” clause

```
<security:intercept-url pattern="/accounts/update*"  
access="hasAnyRole('ROLE_ADMIN', 'ROLE_MANAGER')"/>
```

- “and” clause

```
<security:intercept-url  
pattern="/accounts/update*" access  
="hasRole('ROLE_ADMIN') and hasRole('ROLE_MANAGER')"/>
```

- Previous and new syntax can't be mixed

```
<security:intercept-url pattern=  
"/accounts/update*" access="hasRole('ROLE_MANAGER')"/>  
<security:intercept-url pattern="/accounts/update*"  
access="ROLE_ADMIN"/>
```

Not correct!!

0

login and logout

```
<beans ...>
    <security:http use-expressions="true">
        <security:form-login login-page="/login.htm"
            authentication-failure-url="/login.htm?login_error=1"/>

        <security:intercept-url pattern="/accounts/update*"
            access="hasRole('ROLE_MANAGER')"/>

        <security:logout
            logout-success-url="/comeAgain.html"/>
    </security:http>
</beans>
```

Login Page

Redirected there in case of
incorrect login/password

Spring configuration file

Will be redirected to
this page after logout



Road Map

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security



An Example Login Page

URL that indicates an authentication request

The expected keys for generation of an authentication request token

```
<form action="      method="POST">  
    <input type="text" name="j_username"/>  
    <br/>  
    <input type="password" name="j_password"/>  
    <br/>  
    <input type="submit" name="submit" value="LOGIN"/>  
</form>
```

login-example.jsp



Above example shows default values (*j_spring_security_check*, *j_username*, *j_password*). All of them can be redefined using Spring configuration

Configure Authentication



- DAO Authentication provider is default
- Plug-in specific UserDetailsService implementation to provide credentials and authorities
 - Built-in: JDBC, in-memory
 - Custom

```
<security:authentication-manager>
    <security:authentication-provider>
        ...
    </security:authentication-provider>
<security:authentication-manager>
```

The In-Memory user service (1/2)

- Useful for development and testing
 - Without encoding

```
<security:authentication-manager>
    <security:authentication-provider>
        <security:user-service
            properties= "/WEB-INF/users.properties" />
        </security:authentication-provider>
    <security:authentication-manager>
```

- With encoding

**admin=secret,ROLE_ADMIN
testuser1=pass,ROLE_MEMBER**

```
<security:authentication-manager>
    <security:authentication-provider>
        <security:password-encoder hash="md5" />
        <security:user-service
            properties= "/WEB-INF/users.properties" />
    </security:authentication-provider>
<security:authentication-manager>
```

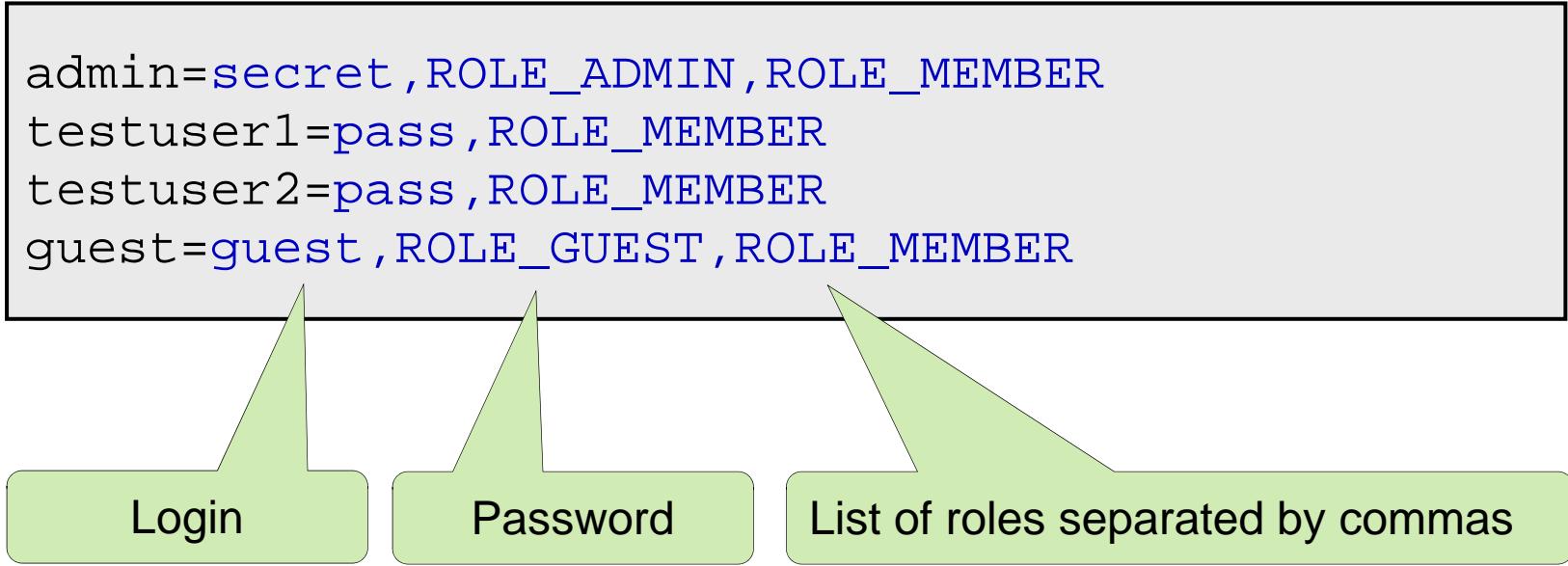
Spring configuration file

The In-Memory UserDetailsService (2/2)



- The properties file

```
admin=secret,ROLE_ADMIN,ROLE_MEMBER
testuser1=pass,ROLE_MEMBER
testuser2=pass,ROLE_MEMBER
guest=guest,ROLE_GUEST,ROLE_MEMBER
```



Login

Password

List of roles separated by commas

Password salting

- Secure passwords by adding a well-known string
 - makes brute force attacks against password store more complex
- System wide salt source
 - Add static application-wide string
- Reflection based salt source
 - Use constant property of entity (e.g. id)

```
<security:password-encoder hash="md5">
    <security:salt-source user-property="id" />
</security:password-encoder>
```

Spring configuration file

The JDBC user service (1/2)



Queries RDBMS for users and their authorities

- Provides default queries
 - SELECT username, password, enabled FROM users WHERE username = ?
 - SELECT username, authority FROM authorities WHERE username = ?

The JDBC user service (2/2)

- Configuration:

```
<beans ....>
    <security:http> ... <security:http>

    <security:authentication-manager>
        <security:authentication-provider>
            <security:jdbc-user-service
                data-source-ref="myDatasource" />
            </security:authentication-provider>
        <security:authentication-manager>
    </beans>
```

Spring configuration file

possibility to customize queries
using attributes such as
authorities-by-username-query

Other Authentication Options

- Implement a custom `UserDetailsService`
 - Delegate to an existing User repository or DAO
- LDAP
- X.509 Certificates
- JAAS Login Module
- Single-Sign-On
 - SiteMinder
 - Kerberos
 - JA-SIG Central Authentication Service

Authorization is not affected by changes to Authentication!



Road Map

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security



Tag library declaration

- The Spring Security tag library can be declared as follows

```
<%@ taglib prefix="security"  
uri="http://www.springframework.org/security/tags" %>
```

Available since Spring Security 2.0

Spring Security's Tag Library



- Display properties of the Authentication object

```
You are logged in as:
```

```
<security:authentication property="principal.username"/>
```

jsp

- Hide sections of output based on role

```
<security:authorize access="hasRole('ROLE_MANAGER')">
    TOP-SECRET INFORMATION
    Click <a href="/admin/deleteAll">HERE</a>
          to delete all records.
</security:authorize>
```

jsp

Authorization in JSP based on intercept-url



- Role declaration can be centralized in Spring config files

```
<security:authorize url="/admin/deleteAll">  
    TOP-SECRET INFORMATION  
    Click <a href="/admin/deleteAll">HERE</a>  
</security:authorize>
```

jsp

URL to protect

```
<security:intercept-url pattern="/admin/*"  
access="hasAnyRole('ROLE_MANAGER', 'ROLE_ADMIN')"/>
```

Spring configuration file

Pattern that matches the URL to be protected

Matching Roles



Road Map

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security





Method security

- Spring Security uses AOP for security at the method level
 - xml configuration with the Spring Security namespace
 - annotations based on Spring annotations or JSR-250 annotations





Method security with XML config

- Allows to apply security to many beans with only a simple declaration

```
<security:global-method-security>
    <security:protect-pointcut
        expression="execution(* com.springframework..
                    *Service.*(..))" access="ROLE_USER" />
</security:global-method-security>
```

Spring configuration file

Method security with Spring annotations



- Spring Security annotations should be enabled

```
<security:global-method-security  
    secured-annotations="enabled" />
```

- on the Java level:

```
import  
org.springframework.security.annotation.Secured;  
  
public class ItemManager {  
    @Secured( "ROLE_MEMBER" )  
    public Item findItem(long itemNumber) {  
        ...  
    }  
}
```

Method security with JSR 250 annotations



- JSR-250 annotations should be enabled

```
<security:global-method-security  
        jsr250-annotations="enabled" />
```

- on the Java level:

```
import javax.annotation.security.RolesAllowed;  
  
public class ItemManager {  
    @RolesAllowed("ROLE_MEMBER")  
    public Item findItem(long itemNumber) {  
        ...  
    }  
}
```



Summary

After completing this lesson, you should have learnt:

- The High-Level Security Overview
- The Motivations of Spring Security
- Spring Security in a Web Environment
- To Configure Web Authentication
- To Use Spring Security's Tag Libraries
- Method security





LAB

Applying Security to a Web Application



Web Flow Advanced Features





Overview

After completing this lesson, you should be able to:

- Describe Flow Input and Output
- Describe Subflows
- Understand Flow Inheritance
- Understand Execution Listeners
- Secure Web Flows
- Tune the Flow Execution



Road Map

- Flow Input and Output
- Subflows
- Flow Inheritance
- Execution Listeners
- Securing Web Flow Definitions
- Tuning

Flow Contract

- A flow is like any other method with an input-output contract
- Named parameters are passed in and out of flows, essentially values in a map

```
<flow>
    <input name="accountNumber" required="true" />
    ...
    <end-state id="confirm">
        <output name="confirmationNumber"
               value="confirmation.id" />
    </end-state>
</flow>
```



Road Map

- Flow Input and Output
- Subflows
- Flow Inheritance
- Execution Listeners
- Securing Web Flow Definitions
- Tuning



Subflows

- A flow invoked from another flow
 - Self contained execution
 - Parent flow is suspended while subflow is executing
- Has own scope
 - Flow scope isolated to particular flow or subflow
 - Conversation scope shared between parent flow and all subflows



Subflow State

- Subflow launched from special state in parent flow
- Transitions keyed from subflow's end-state id

```
<subflow-state id="beneficiaries"
                subflow="editBeneficiaries">
    <transition on="endCommit" to="confirm"/>
    <transition on="endCancel" to="edit"/>
</subflow-state>
```

Subflow Input/Output

- Pass named parameters between calling flow and subflow

```
<subflow-state id="beneficiaries" subflow="editBeneficiaries">
    <input name="accountNumber" value="account.number"/>
    <output name="confirmationNumber"
           value="flowScope.confirmationNumber"/>
    ...
</subflow-state>
```

calling flow

```
<flow ...>
    <input name="accountNumber" required="true"/>
    ...
    <end-state id="endCommit">
        <output name="confirmationNumber" value="confirmation.id"/>
    </end-state>
    <end-state id="endCancel"/>
</flow>
```

subflow "editBeneficiaries"

Mocking Subflows

```
getFlowDefinitionRegistry().registerFlowDefinition
    (createMockEditBeneficiariesSubflow());
...
public Flow createMockEditBeneficiariesSubflow() {
    Flow mockSubflow = new Flow("editBeneficiaries");
    mockSubflow.setInputMapper(new Mapper() {
        public MappingResults map(Object source, Object target) {
            // test input mapping
            assertEquals(1L, ((AttributeMap) source)
                .get("accountNumber"));
            return null;
        }
    });
    // immediately return so the caller can respond
    new EndState(mockSubflow, "endCancel");
    return mockSubflow;
}
```



Road Map

- Flow Input and Output
- Subflows
- Flow Inheritance
- Execution Listeners
- Securing Web Flow Definitions
- Tuning





Flow Inheritance

- Share common states, actions across flows
- Support for multiple inheritance
 - More like composition than inheritance
- Merge (not override) differences between parent and child flow
- Child flow and parent flow are both registered in the same FlowRegistry



Flow Level Inheritance

- Merges all elements of parent flow into child
 - parent's actions are executed first

```
<flow ... parent="parentFlowName">
```

- Comma separated flow names for multiple inheritance
 - order has no effect on resulting flow
- Abstract flows cannot be directly instantiated, only extended

```
<flow ... abstract="true">
```

State Level Inheritance

- Only specified state's elements are merged

```
<view-state ... parent="parentFlowName#stateId">
```

- Must specify flow name and state id
 - flowId[#stateId]
- default stateId is current state's id
 - Only single inheritance supported
 - Must be of the same state type (i.e. view-state to view-state, end-state to end-state)

Inheritance Example

```
<flow abstract="true">
    <end-state id="endCancel" />
    <global-transitions>
        <transition on-exception
            ="accounts.NoSuchAccountException" to="endCancel" />
    </global-transitions>
</flow>
```

common.xml

```
<flow parent="common">
    ...
    <view-state id="confirm">
        <transition on="save" to="endComplete" />
    </view-state>
    <end-state id="endComplete" />
</flow>
```

child.xml

Inheritance Example (2)

```
<flow>
    ...
    <view-state id="confirm">
        <transition on="save" to="endComplete" />
    </view-state>

    <end-state id="endComplete" />

    <end-state id="endCancel" />

    <global-transitions>
        <transition on-exception
            ="accounts.NoSuchAccountException" to="endCancel" />
    </global-transitions>
</flow>
```

child.xml after inheritance applied



Road Map

- Flow Input and Output
- Subflows
- Flow Inheritance
- Execution Listeners
- Securing Web Flow Definitions
- Tuning



Execution Listeners

- Observe flow execution life-cycle events
 - must implement `FlowExecutionListener`
 - registered with the flow execution engine

```
<webflow:flow-executor id="flowExecutor"  
                      flow-registry="flowRegistry">  
    <webflow:flow-execution-listeners>  
        <webflow:listener ref="myFlowListener" />  
    </webflow:flow-execution-listeners>  
</webflow:flow-executor>
```



FlowExecutionListener

- Methods to respond to web-flow events:
 - created, loaded, paused, resumed, saved, removed
 - eventSignaled
 - requestProcessed, requestSubmitted
 - sessionStarting, sessionStarted, sessionEnded
 - stateEntering, stateEntered
- Several predefined
 - Security, Persistence, Faces (JSF)
 - Can write your own (for example: event tracing)





Road Map

- Flow Input and Output
- Subflows
- Flow Inheritance
- Execution Listeners
- Securing Web Flow Definitions
- Tuning



Flow-Based Security

- Web Flow also integrates with Spring Security
- These “resources” can be secured
 - flows
 - states
 - transitions





How To Secure A Flow

- Standard Spring Security configuration
 - secure the web-flow URLs
 - orthogonal (independent) to Web Flow
- A `SecurityFlowExecutionListener` declaration
 - configured and added to the `flowExecutor`
- Add `<secured>` elements to the flow definition



Flow Security Attributes

Secure a Flow

```
<flow ...>
  <secured attributes="ROLE_USER" />
</flow>
```

Secure a State

```
<flow ...>
  <view-state id="edit">
    <secured attributes="ROLE_USER" />
  </view-state>
</flow>
```

Secure a
Transition

```
<flow ...>
  <view-state id="edit">
    <transition on="next" to="beneficiaries">
      <secured attributes="ROLE_USER" />
    </transition>
  </view-state>
</flow>
```

Configuring a Security Flow Execution Listener

```
<bean id="securityFlowExecutionListener" class=
    "...webflow.security.SecurityFlowExecutionListener"/>

<webflow:flow-executor id="flowExecutor">
    <webflow:flow-execution-listeners>
        <webflow:listener ref="securityFlowExecutionListener"/>
    </webflow:flow-execution-listeners>
</webflow:flow-executor>
```

- Flow `<secured>` element will be ignored without a `SecurityFlowExecutionListener` in place

Non-Role Based Authorities

- Application may choose to authorize on something other than roles
- Can override default AccessDecisionManager
 - use “`accessDecisionManager`” property on listener

```
<bean id="securityFlowExecutionListener" class=
"....webflow.security.SecurityFlowExecutionListener">
    <property name="accessDecisionManager"
              ref="myDecisionMgr" />
</bean>
```



Exception Handling

- The `SecurityFlowExecutionListener` throws `AccessDeniedException`
- Bubbles up and is caught by a Spring Security servlet filter
- Avoid catching/suppressing that exception
 - If using `SimpleMappingExceptionResolver` , override the `doResolveException` method to *rethrow* the exception



Ignoring Exceptions in the Resolver



Ignoring an exception in the exception resolver:

```
public class SmartExceptionResolver  
    extends SimpleMappingExceptionResolver {  
  
    protected ModelAndView doResolveException  
        (HttpServletRequest req, HttpServletResponse res,  
         Object handler, Exception ex) {  
        if (ex instanceof AccessDeniedException)  
            throw (AccessDeniedException) ex;  
  
        return super.doResolveException(req, res, handler, ex);  
    }  
}
```



Road Map

- Flow Input and Output
- Subflows
- Flow Inheritance
- Execution Listeners
- Securing Web Flow Definitions
- Tuning



Flow Execution Tuning

- Limit the number of flow executions and snapshots for a user session
 - reduce memory overhead from “hog” users
 - less likely to encounter an `OutOfMemory` exception

```
<webflow:flow-execution-repository max-executions="5"  
max-execution-snapshots="30" />
```

Maximum 5 concurrent flow executions per user.
No more than 30 snapshots per execution.

Summary

After completing this lesson, you should have learnt to:

- Describe Flow Input and Output
- Describe Subflows
- Understand Flow Inheritance
- Understand Execution Listeners
- Secure Web Flows
- Tune the Flow Execution





Getting Started With Spring Roo

Overview

After completing this lesson, you should be able to:

- Describe what Spring Roo is?
- Download and Install Spring Roo
- Describe Roo Shell
- Create projects using Spring Roo
- Describe how Roo Works
- Describe how to setup Spring MVC using Spring Roo
- Use Spring Roo commands





Lesson Roadmap

- Introduction
- Install
- Setting up a new project
- Working with entities
- Database access
- The Web layer



What Is Spring Roo?

- Easy-to-use, extensible, RAD tool for Java® developers
- Development-time only (no runtime, no lock-in)
- Command-line tool for smart code generation



1.1.5.RELEASE [rev d3a68c3]

```
Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.  
roo>
```



End Users Description

- Roo is a little genie who sits in the background and handles the things I don't want to worry about





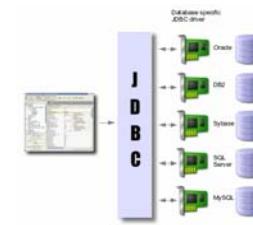
What Is Spring Roo?

- Removes the tedium of creating web apps
 - Productivity without compromise
 - No need to learn a new syntax - it's all Java
- Features
 - Best practice and standards oriented
 - Multiple view technologies
 - GWT, JSP, and JSF (shortly)
 - Database reverse engineering
 - from schema to application
 - Views automatically updated if domain model changes



Finger in every pie

- Java® Platform
 - Java 6+
 - Java Bean Validation
 - JDBC
 - Java Message Service
 - Java Transaction API
 - Java Server Pages
 - Java Persistence API





Finger in every pie

- JPA Implementations
 - Hibernate
 - Apache OpenJPA
 - EclipseLink
 - Google App Engine





Finger in every pie

- Java® Servlet Apps
 - Jetty
 - Apache Tomcat
 - Apache Tiles
 - Spring MVC
 - Spring Web Flow

jetty://





Finger in every pie

- Popular Open Source
 - Apache Maven
 - Google Web Toolkit
 - Adobe Flex
 - Dojo Toolkit
 - Log4J
 - AspectJ
 - Selenium
 - JUnit

maven



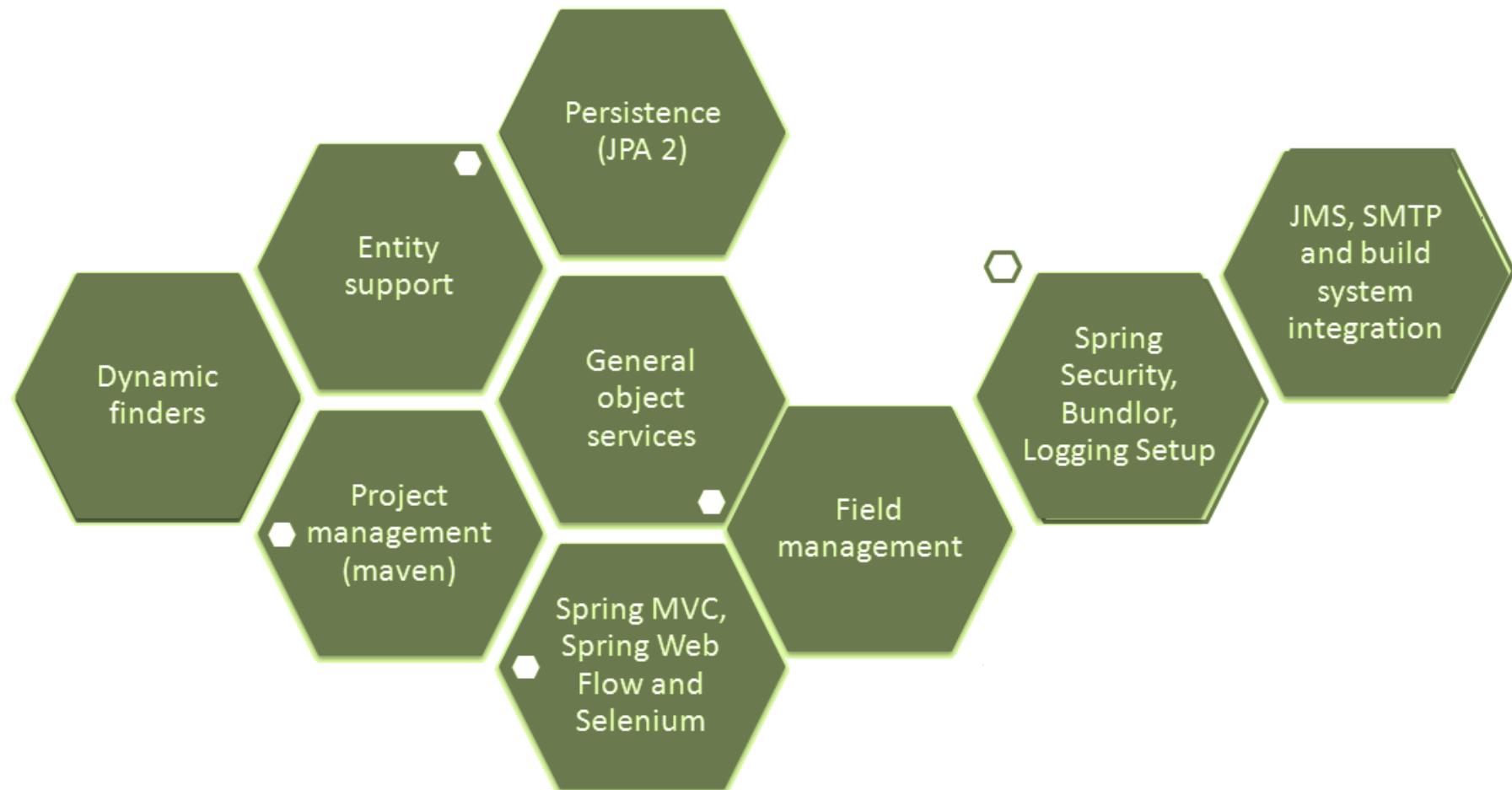
dōjō



aspectj *crosscutting objects for better modularity*



Out Of The Box





Lesson Roadmap

- Introduction
- Install
- Setting up a new project
- Working with entities
- Database access
- The Web layer





Requirements

- Java 6
 - Spring 3 already requires Java 5 or above
- Apache Maven
 - Most commonly used build system for Java projects
 - Version 2.0.9 or above required





Download and Easy Install

- Option 1: manual install
 - Install Maven
 - Download Roo from www.springframework.org/roo
 - Unzip and ensure `roo` script is in the path
- Option 2: use Roo inside STS
 - STS ships with the latest Roo version
 - STS also ships with Maven



Running Roo

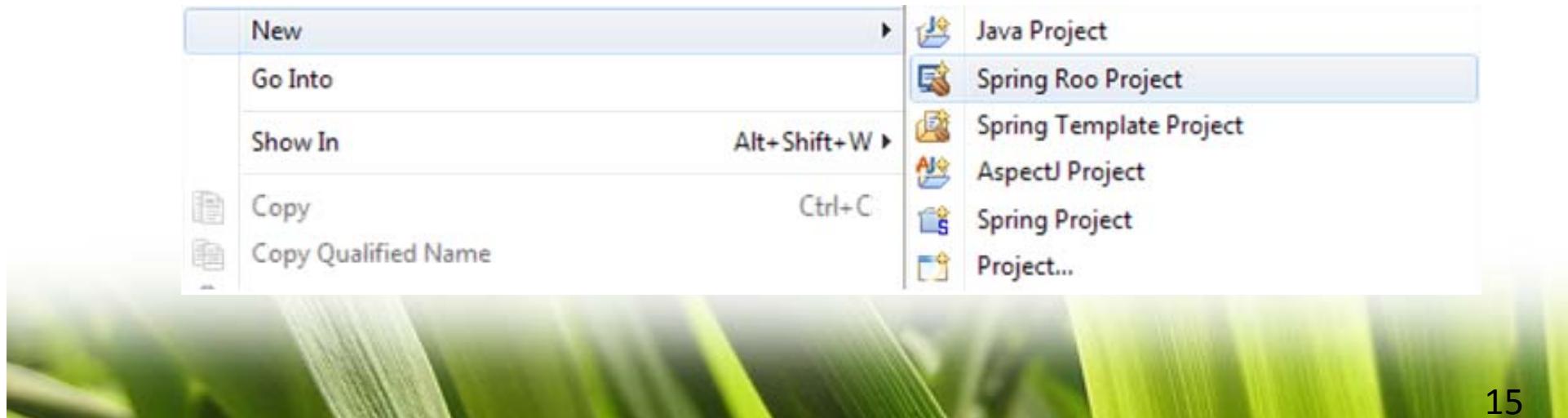
- Launch `roo.bat` or `roo.sh` from the command-line



1.1.5.RELEASE [rev d3a68c3]

```
Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.  
roo>
```

- Alternatively, create new Roo project in STS



Roo Shell

- Roo shell is a command line interface
 - Run in a shell/command window
 - Alternatively run the Roo shell from within STS
- Roo shall provides various commands to
 - Create a project
 - Enable persistence
 - Create domain objects
- Use tab for smart auto-completion (`Ctrl+Space` in STS)
- Roo shell also supports round-tripping
 - Change a Roo-generated Java file manually
 - Roo “wakes up” and resyncs everything



Hint

- Provides step-by-step hints and context-sensitive guidance

```
roo> hint
At this stage of the project, you have a few options:

 * List all hint topics via 'hint topics'
 * Create more fields with 'hint fields'
 * Create more entities with 'hint entities'
 * Create a web controller with 'hint controllers'
 * Create dynamic finders with 'hint finders'
 * Setup your logging levels via 'hint logging'
 * Run tests via Maven (type 'exit' and then 'mvn test')
 * Build a deployment artifact (type 'perform package')
 * Learn about Eclipse integration by typing 'hint eclipse'
 * Discover all Roo commands by typing 'help'
```

```
roo>
```



Lesson Roadmap

- Introduction
- Install
- Setting up a new project
- Working with entities
- Database access
- The Web layer



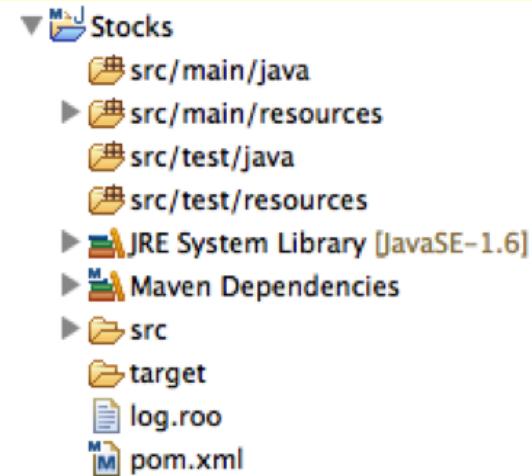
Project Creation

- Create a new project in the current directory

```
roo> project --topLevelPackage com.stocks  
Created E:\workspaces\workspace\Stocks\pom.xml  
Created SRC_MAIN_JAVA  
Created SRC_MAIN_RESOURCES  
Created SRC_TEST_JAVA  
Created SRC_TEST_RESOURCES  
Created SRC_MAIN_WEBAPP  
Created SRC_MAIN_RESOURCES\META-INF\spring  
Created SRC_MAIN_RESOURCES\META-INF  
\spring\applicationContext.xml  
Created SRC_MAIN_RESOURCES\META-INF  
\spring\log4j.properties
```

Maven project descriptor

Uses Log4J



Roo's Persistence Support

- JPA 2 for ORM
 - 4 choices of ORM providers
 - Hibernate, EclipseLink, OpenJPA, and DataNucleus
- 14+ database choices
 - Oracle, SQL Server, DB2, MySQL, Postgres ...
 - Derby, HSQLDB (for dev) ...



Persistence Setup (1/3)

- Setting up database configuration in one line

```
roo> jpa setup --database DERBY --provider HIBERNATE
```

- What exactly happens there:

```
roo> jpa setup --database DERBY --provider HIBERNATE

Created SRC_MAIN_RESOURCES/META-INF/spring/database.properties
...
Updated SRC_MAIN_RESOURCES/META-INF
                           /spring/applicationContext.xml
Created SRC_MAIN_RESOURCES/META-INF/persistence.xml
```

Persistence Setup (2/3)

- Generated JPA configuration file

```
<persistence>
  <persistence-unit name="persistenceUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.DerbyDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create"/>
      ...
    </properties>
  </persistence-unit>
</persistence>
```

`src/main/resources/META-INF/persistence.xml`

- Generated database configuration file

```
database.password=
database.url=jdbc\:derby\:Stocks;create\=true
database.username=
database.driverClassName=org.apache.derby.jdbc.EmbeddedDriver
```

 `src/main/resources/META-INF/spring/database.properties`

- Default values
- used by Spring configuration
- can be changed in the command line "database properties set..."

Persistence Setup (3/3)

- Generated Spring configuration file

```

<beans>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="url" value="${database.url}" />
        <property name="username" value="${database.username}" />
        <property name="password" value="${database.password}" />
    </bean>
    <bean class="org.springframework.orm.jpa.JpaTransactionManager"
          id="transactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
          id="entityManagerFactory">
        <property name="persistenceUnitName" value="persistenceUnit" />
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>

```

Database connection pool

Transactions

JPA

src/main/resources/META-INF/spring/applicationContext.xml



Lesson Roadmap

- Introduction
- Install
- Setting up a new project
- Working with entities
- Database access
- The Web layer



Creating an Entity

- Run the entity command

```
roo> entity --class ~domain.Account
```

'~' refers to default package path

- But what happens exactly there?

```
roo> entity --class ~domain.Account
```

Created SRC_MAIN_JAVA/com/stocks/domain
Created SRC_MAIN_JAVA/com/stocks/domain/Account.java
Created SRC_MAIN_JAVA/com/stocks/domain/Account_Roo_Configurable.aj
Created SRC_MAIN_JAVA/com/stocks/domain/Account_Roo_Entity.aj
Created SRC_MAIN_JAVA/com/stocks/domain/Account_Roo_ToString.aj

Entity class

AspectJ files

Entity Example

- Generated Entity class skeleton

```
import javax.persistence.Entity;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;
import org.springframework.roo.addon.entity.RooEntity;

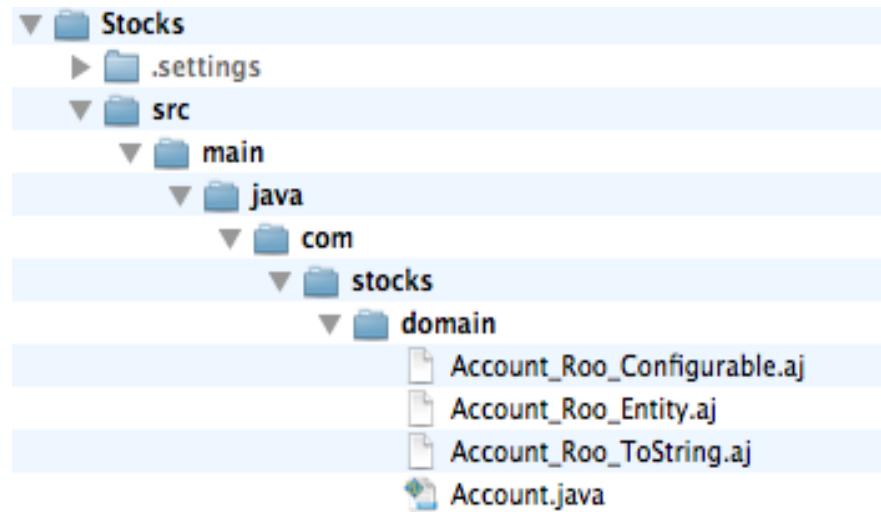
@Entity
@RooJavaBean
@RooToString
@RooEntity
public class Account { }
```

JPA annotation

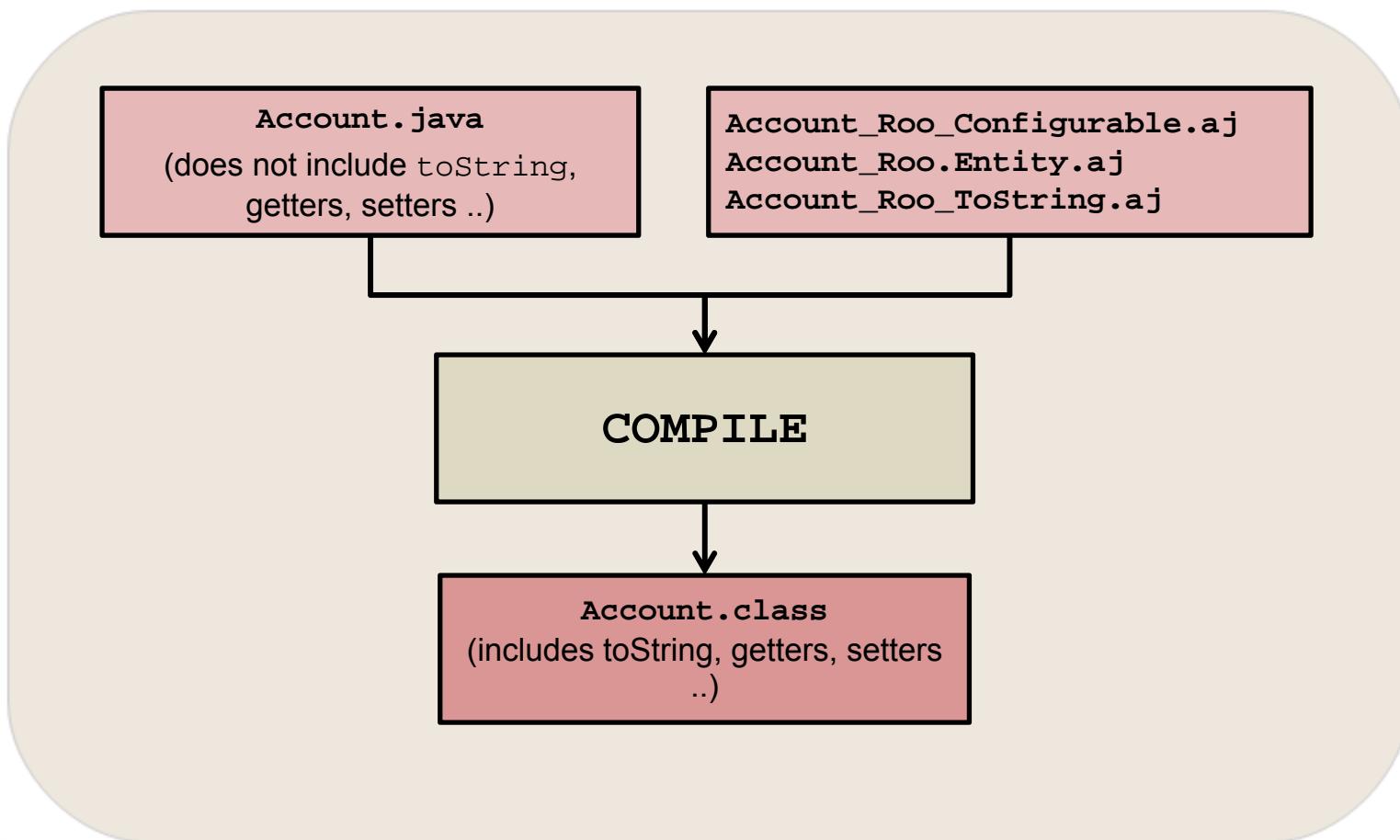
Roo-specific annotations

AspectJ files

- Add behavior to entities
 - `toString()` method
 - getters and setters
 - Basic JPA features
 - ...
- Updated automatically when changes are made to Entity classes
- Hidden by default in Eclipse



Compile Time Weaving



Working with attributes

- Attributes can be added in the Entity class

```
@Entity  
@...  
public class Account {  
  
    private String name;  
    private Long accountNumber;  
}
```

- Generated methods updated in the background
 - Getters, setters, `toString`...



Auto-complete

- Auto-complete works in STS!

```
@Test  
public void createAccount() {  
    Account account = new Account();  
    account.g  
}  
    ● getAccountNumber() : Long - Account  
    ● getClass() : Class<?> - Object  
    ● getId() : Long - Account  
    ● getName() : String - Account  
    ● getVersion() : Integer - Account
```



The attributes “id” and “version” are added implicitly. They are mostly used for database access.

Overriding default behavior

- Auto-generation only applies to methods that have not been explicitly declared

```
@Entity  
@...  
public class Account {  
  
    private String name;  
}
```

getName() and
setName(....)
declared in AspectJ file

```
@Entity  
@...  
public class Account {  
  
    private String name;  
    private void setName  
        (String name)  
    { //...  
    }
```

- setName(....) not declared in AspectJ file anymore
- visibility can be private



Lesson Roadmap

- Introduction
- Install
- Setting up a new project
- Working with entities
- Database access
- The Web layer



Generic configuration

- As discussed before, an entityManagerFactory has been declared in Spring configuration file

```
<beans>
    <!-- -->

    <bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
          id="entityManagerFactory">
        <property name="persistenceUnitName" value="persistenceUnit" />
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

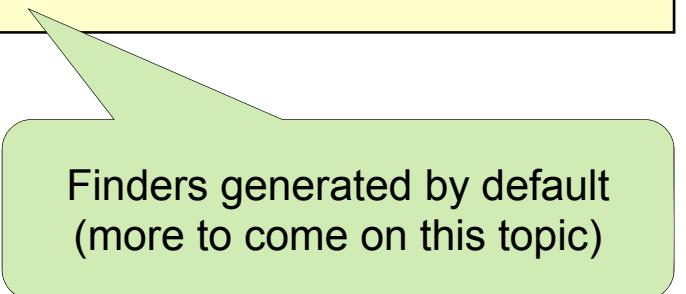
src/main/resources/META-INF/spring/applicationContext.xml

- entityManagerFactory is the entry point to access a database with JPA

Implicit methods

- Entities expose more methods than you would think
 - persist(), findAll...()

```
Account account = new Account();  
// call setter methods  
account.persist();  
  
List<Account> accountList = Account.findAllAccounts();  
Account customerAccount = Account.findAccount(2L);
```



Finders generated by default
(more to come on this topic)

Adding Finders: finder list

- A set of finders can be generated on demand
 - Use “finder list” to know the list of them

```
roo> finder list --class com.stocks.domain.Account
findAccountsByAccountNumberEquals(Long accountNumber)
findAccountsByAccountNumberGreaterThan(Long accountNumber)
findAccountsByAccountNumberGreaterThanOrEqual(Long accountNumber)
findAccountsByAccountNumberIsNotNull()
findAccountsByAccountNumberIsNull()
findAccountsByAccountNumberLessThan(Long accountNumber)
findAccountsByAccountNumberLessThanOrEqual(Long accountNumber)
findAccountsByAccountNumberNotEqual(Long accountNumber)
findAccountsByNameEquals(String name)
...
```

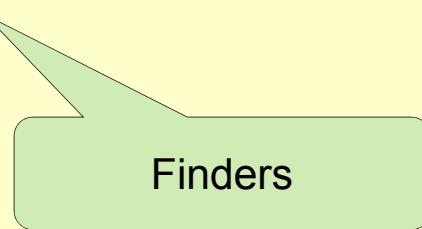
Adding Finders: finder add

- Use “finder add” command to add a finder

```
~.domain.Account roo> finder add  
    --finderName findAccountsByAccountNumberGreaterThan  
Updated SRC_MAIN_JAVA/com/stocks/domain/Account.java  
Created SRC_MAIN_JAVA/com/stocks/domain/Account_Roo_Finder.aj
```

- Finders are declared on top

```
@Entity  
@RooEntity(finders={ "findAccountsByAccountNumberGreaterThan"  
})  
@...  
public class Account {  
    private String name;  
    //...  
}
```



Finders

Adding Finders: Generated Code



AspectJ syntax
(close to Java syntax)

```
public static TypedQuery<Account> Account
    .findAccountsByAccountNumberGreaterThan(Long accountNumber) {
    //...
    EntityManager em = Account.entityManager();
    TypedQuery<Account> q =
        em.createQuery("SELECT o FROM Account AS o WHERE
                      o.accountNumber > :accountNumber", Account.class);
    q.setParameter("accountNumber", accountNumber);
    return q;
}
```

Account_Roo_Finder.aj

- Finder generation works well for simple queries
- For more advanced needs, you should write the finder yourself



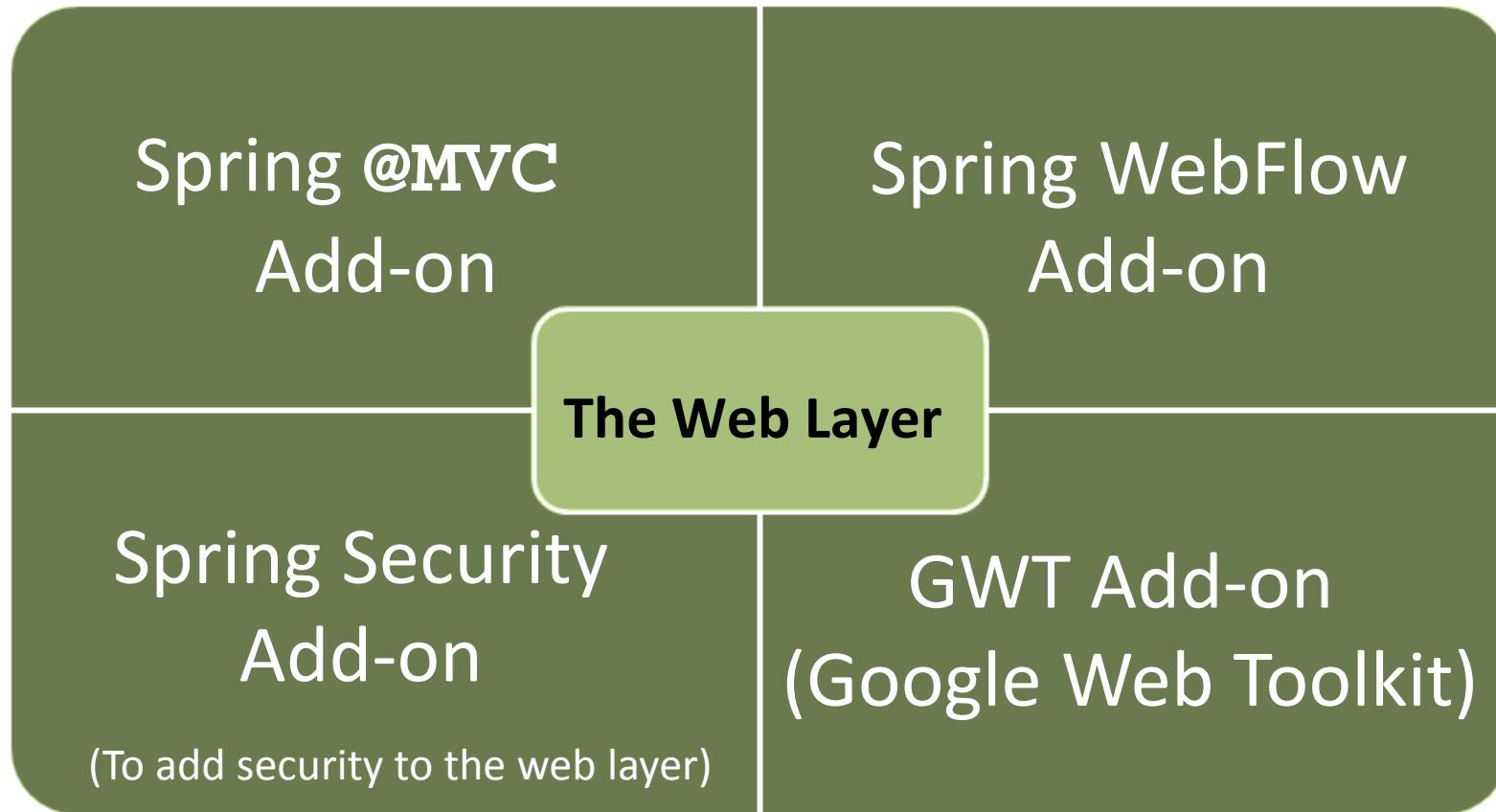
Lesson Roadmap

- Introduction
- Install
- Setting up a new project
- Working with entities
- Database access
- The Web layer





The Web Layer



Spring @MVC setup

- Use “web mvc setup” for generic configuration
 - Generates a default UI layout
 - Can be easily removed or customized

```
~.domain.Account roo> web mvc setup
Created SRC_MAIN_WEBAPP/WEB-INF/spring/webmvc-config.xml
Created SRC_MAIN_WEBAPP/WEB-INF/web.xml
...
Created SRC_MAIN_WEBAPP/images/create.png
Created SRC_MAIN_WEBAPP/images/list.png
Created SRC_MAIN_WEBAPP/images/resultset_previous.png
...
Created SRC_MAIN_WEBAPP/styles/alt.css
Created SRC_MAIN_WEBAPP/styles/standard.css
...
Created SRC_MAIN_WEBAPP/WEB-INF/views/header.jspx
Created SRC_MAIN_WEBAPP/WEB-INF/views/footer.jspx
```

Generic Spring and Java EE configuration

Default images and stylesheet

Default views



Working with the Web Layer

Controllers and Views
managed by Spring
Roo

Fast but less flexible

`web mvc scaffold`

Controllers and Views
handled manually

Close to “raw” Spring
@MVC

`web mvc controller`

"web mvc scaffold" controller classes



- Spring Roo generates CRUD methods for controllers
- Based on REST
- Controllers and views are updated when there is a change in the domain

```
web mvc scaffold  
  --class com.stocks.controller.AccountController  
  --backingType com.stocks.domain.Account
```

Scaffolding: controller classes

- Generated method sample

```

@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public String AccountController.show(@PathVariable("id")
                                      Long id, Model uiModel) {
    uiModel.addAttribute("account", Account.findAccount(id));
    uiModel.addAttribute("itemId", id);
    return "accounts/show";
}

@RequestMapping(method = RequestMethod.POST)
public String AccountController.create(@Valid Account account,
                                         BindingResult bindingResult, Model uiModel,
                                         HttpServletRequest httpServletRequest) {
    //...
    account.persist();
    return "redirect:/accounts/" + ...;
}

```

REST style URL

Static call to Entity

JSR 303 Validation

call to Entity

`com.stocks.controller.AccountController_Roo_Controller.aj`



In the above sample, the Controller calls the Entity directly. As an alternative, Roo can generate a service layer from Roo 1.2

Scaffolding: controller classes

- All methods are generated by default
 - They can be overridden

Indicates that Spring Roo generates CRUD methods

```
@RooWebScaffold(path = "accounts",
                  formBackingObject = Account.class)
@RequestMapping("/accounts")
@Controller
public class AccountController {
}
```

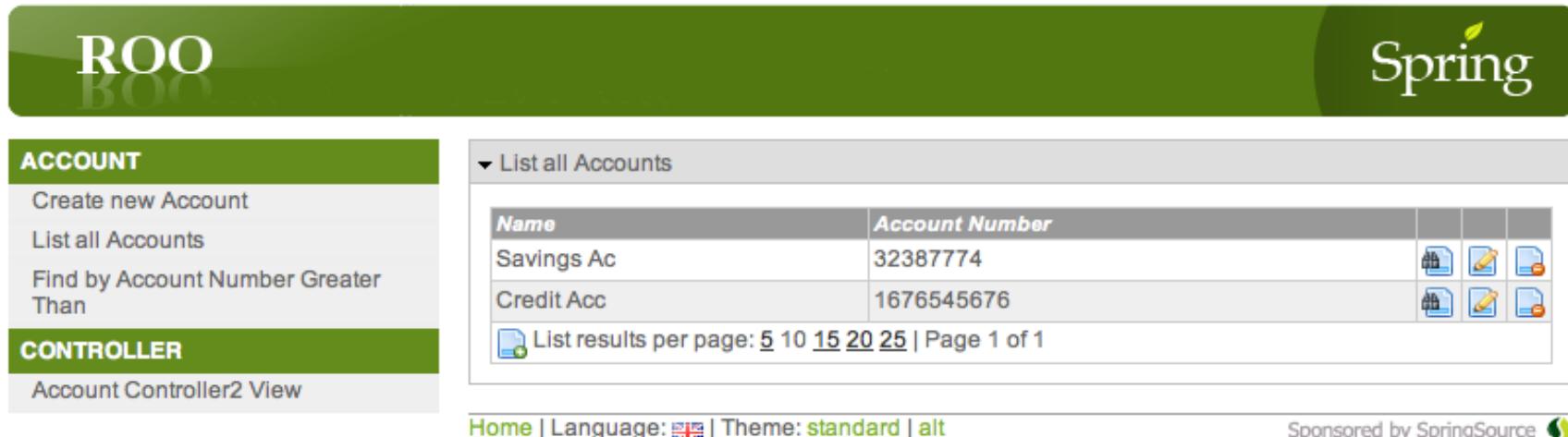
Entity class

Controller class

com.stocks.controller.AccountController

Scaffolding: the view layer

- Uses JSP by default
 - .jspx extension (Java scriptlets not allowed)
 - Based on Dojo and Apache Tiles



The screenshot shows the Roo application interface. At the top, there's a dark green header bar with the 'ROO' logo on the left and the 'Spring' logo on the right. Below the header is a sidebar with two sections: 'ACCOUNT' and 'CONTROLLER'. The 'ACCOUNT' section contains links for 'Create new Account', 'List all Accounts', and 'Find by Account Number Greater Than'. The 'CONTROLLER' section contains the link 'Account Controller2 View'. The main content area has a title 'List all Accounts'. It displays a table with two rows of account data:

Name	Account Number	Actions
Savings Ac	32387774	 
Credit Acc	1676545676	 

Below the table, there's a link to change the number of results per page: 'List results per page: 5 10 15 20 25 | Page 1 of 1'. At the bottom of the page, there are links for 'Home', 'Language: EN', 'Theme: standard | alt', and a 'Sponsored by SpringSource' link with the Spring logo.



Apache Tiles and Dojo are used by default. If you do not wish to use them, you can create your custom Roo Add-on

Scaffolding: the view layer

- Round-trip engineering is supported

```
<table:table data="${accounts}" id="l_com_stocks_domain_Account"
    path="/accounts" z="vPb3xhvmHXcfnDwsQHW3hEnArCQ=>

    <table:column id="c_com_stocks_domain_Account_name"
        property="name" z="4eCOfD6vXh4t/ASrdeEr/Gz7X1E=>

    <table:column id="c_com_stocks_domain_Account_accountNumber"
        property="accountNumber" z="MABRxEmNDpTkPYh5UbNukQyEWJs=>
</table:table>
```

JSP sample

Remove the attribute “z” if you wish to customize a section of the JSP

```
@Entity ...
public class Account {
    private String name;
    private Long accountNumber;
}
```

Entity class

- Order of attributes in the Entity reflects order in the JSP
- When an attribute is added to an Entity, it is added to the corresponding JSP

"web mvc controller"



```
web mvc controller --class  
com.stocks.controller.AccountController
```

- No scaffolding anymore
- Skeleton of a controller is created
 - Controller methods should be created manually
- Views should be created manually

No scaffolding
annotation anymore

```
@RequestMapping( "/accounts" )  
@Controller  
public class AccountController {  
    @RequestMapping  
        public void get(ModelMap modelMap, HttpServletRequest  
Request, HttpServletResponse response) { }  
}
```

A couple of skeleton methods are
created to help you getting started

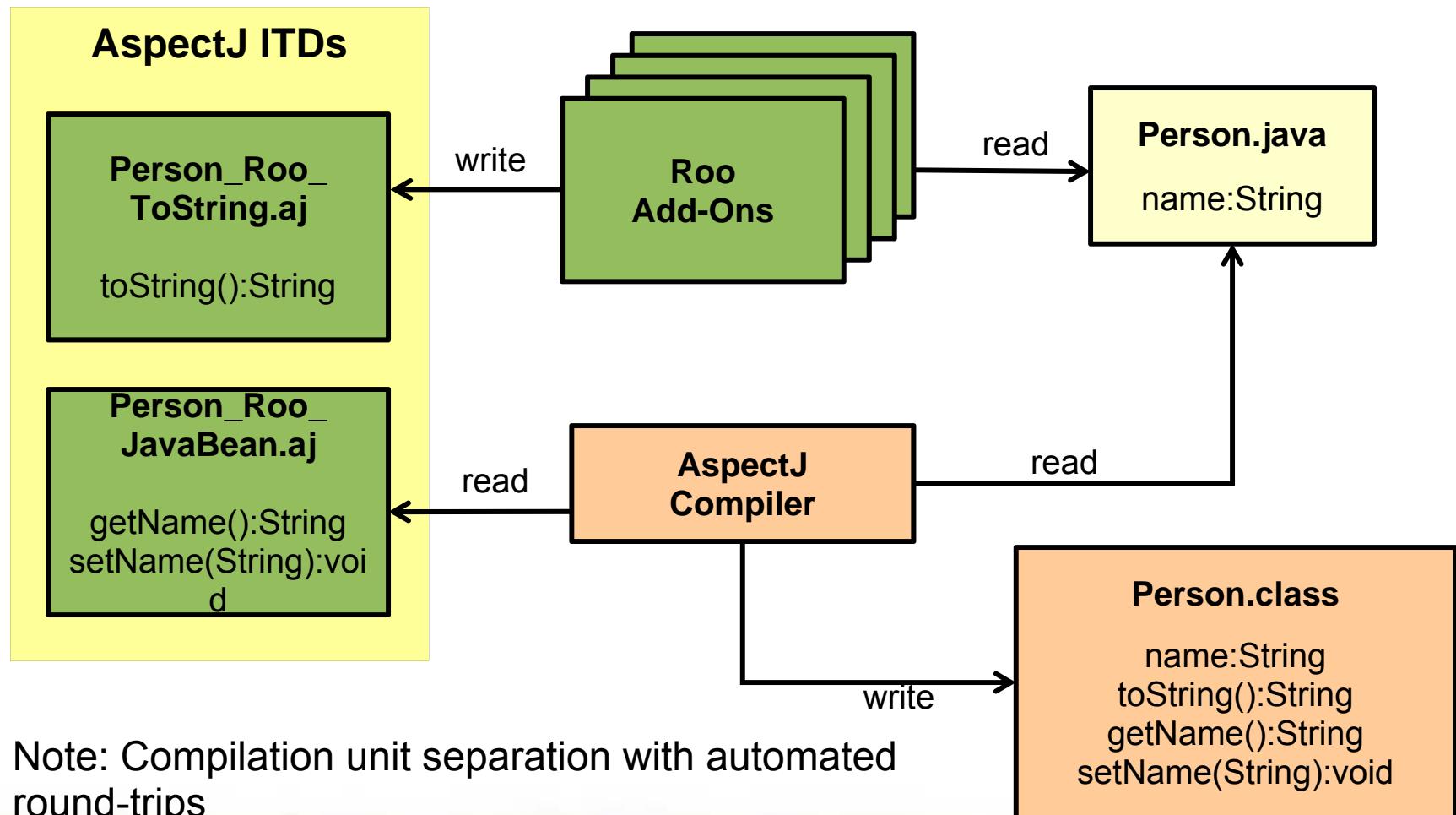
com.stocks.controller.AccountController

Web Layer Strategy

- Should you use scaffolding for generating the Web layer?
 - Only suitable for typical CRUD applications
- For other applications, it is a common choice to handle the Web layer manually



Active Generation



Note: Compilation unit separation with automated round-trips



Build System Integration

- Assorted build options
 - perform clean runs `mvn clean`
 - perform eclipse runs `mvn eclipse:eclipse`
 - perform tests runs `mvn test`
 - perform package runs `mvn package`
 - perform command custom `mvn command`
- You must have Maven set up and in path to use



Miscellaneous

- logging setup manages log4j.properties
- script executes a script
 - Specify clinic.roo, wedding.roo and vote.roo for inbuilt sample applications
 - Specify a fully-qualified file system path for your files
- Roo has tab completion for files as well!



Summary

After completing this lesson, you should have learnt:

- What Spring Roo is?
- Download and Install Spring Roo
- How to work with Roo Shell
- Create projects using Spring Roo
- How Roo Works
- How to setup Spring MVC using Spring Roo
- Use Spring Roo commands



Appendix – Roo Commands

- Brief Overview of Roo Commands:
 - project, jpa setup
 - entity, field, finder
 - test, selenium
 - controller, webflow
 - email, jms
 - perform, logging
 - security
 - script
- For complete list of commands, visit

<http://static.springsource.org/spring-roo/reference/html/command-index.html>



Resources

- For more information ...
 - Home: <http://www.springsource.org/roo>
 - Forum: <http://forum.springsource.org>
 - Issues: <http://jira.springframework.org/browse/ROO>
 - Twitter: @SpringRoo
- Public Git and Jira available
 - <git://git.springsource.org/roo/>
 - <https://jira.springsource.org/browse/ROO>





Debugging and Testing Web Applications



Overview

After completing this lesson, you should be able to:

- Debug Applications in a Browser
- Test Web Applications
- Describe Selenium
- Describe Apache JMeter



Road Map

- Debugging Applications in a Browser
- Testing Web Applications
- Selenium
- Apache JMeter

Firefox Profiles



- Many Web development tools run on Firefox
- It's also your everyday browser
- Use different profiles to separate settings

```
> firefox -p
```

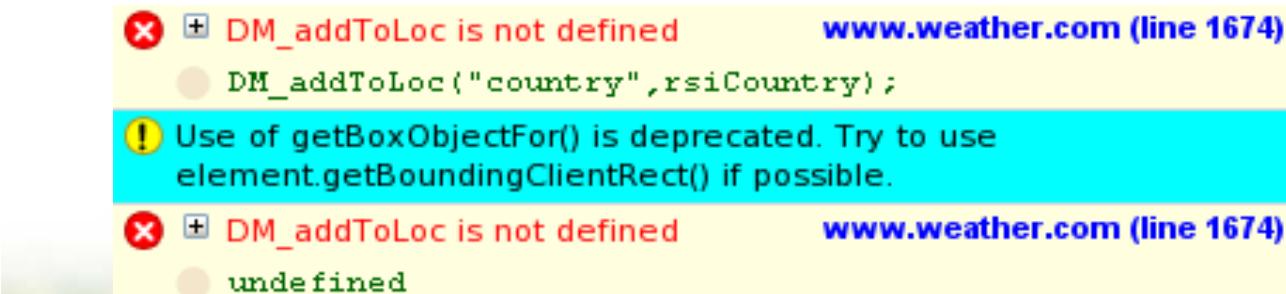


Command Line



Firebug

- Logging for web pages
 - console.debug('debug message')
- JavaScript debugging
 - step through line by line
- JavaScript error reporting
 - click-through to error
- Ajax Spy
 - monitor XHR request details



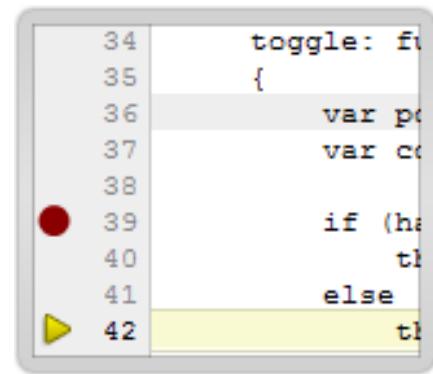
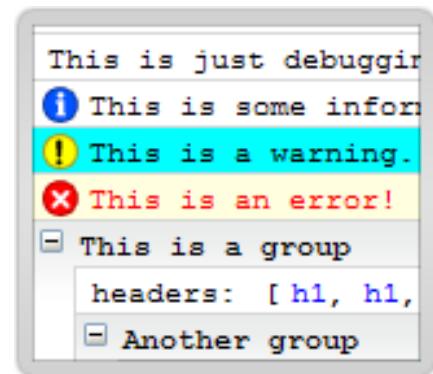
DM_addToLoc is not defined www.weather.com (line 1674)

DM_addToLoc ("country", rsiCountry);

Use of getBoxObjectFor() is deprecated. Try to use element.getBoundingClientRect() if possible.

DM_addToLoc is not defined www.weather.com (line 1674)

undefined



YSlow Firebug Add-on



The screenshot shows the YSlow Firebug Add-on interface. At the top, there's a toolbar with tabs like Inspect, Performance (which is selected), Stats, Components, Tools, Help, and a search bar. Below the toolbar, there's a menu bar with Console, HTML, CSS, Script, DOM, Net, YSlow (selected), and Options. The main area displays the "Performance Grade: A (96)". Below the grade, there's a list of 13 performance guidelines, each with a letter grade (B, A, A, A, B, A, A, n/a, A, A, A, A, A) and a numbered description. One guideline is highlighted in red: "1 external stylesheets were found outside the document HEAD." with a link to "http://1.yimg.com/d/i/ww/sp/onload_1.4.2.css". At the bottom, there's a "Done" button and some status icons.

- B** 1. Make fewer HTTP requests ▾
- A** 2. Use a CDN
- A** 3. Add an Expires header
- A** 4. Gzip components
- B** 5. Put CSS at the top ▾
 - 1 external stylesheets were found outside the document HEAD.
↗ http://1.yimg.com/d/i/ww/sp/onload_1.4.2.css
- A** 6. Put JS at the bottom
- A** 7. Avoid CSS expressions
- n/a** 8. Make JS and CSS external ▾
- A** 9. Reduce DNS lookups
- A** 10. Minify JS
- A** 11. Avoid redirects
- A** 12. Remove duplicate scripts
- A** 13. Configure ETags

Firefox add-on with 13 suggested performance guidelines from Yahoo

Warning: YSlow can interfere with Firebug's Ajax request spying. If you run into this issue try disabling YSlow.

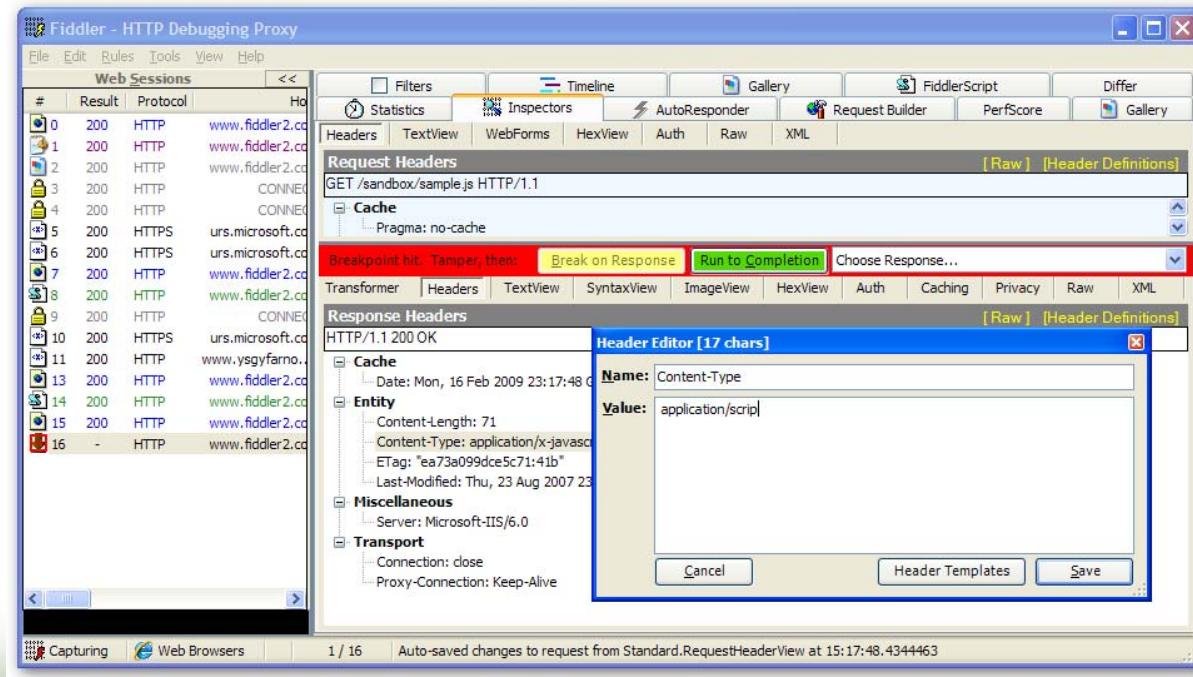
Web Developer Firefox Add-on



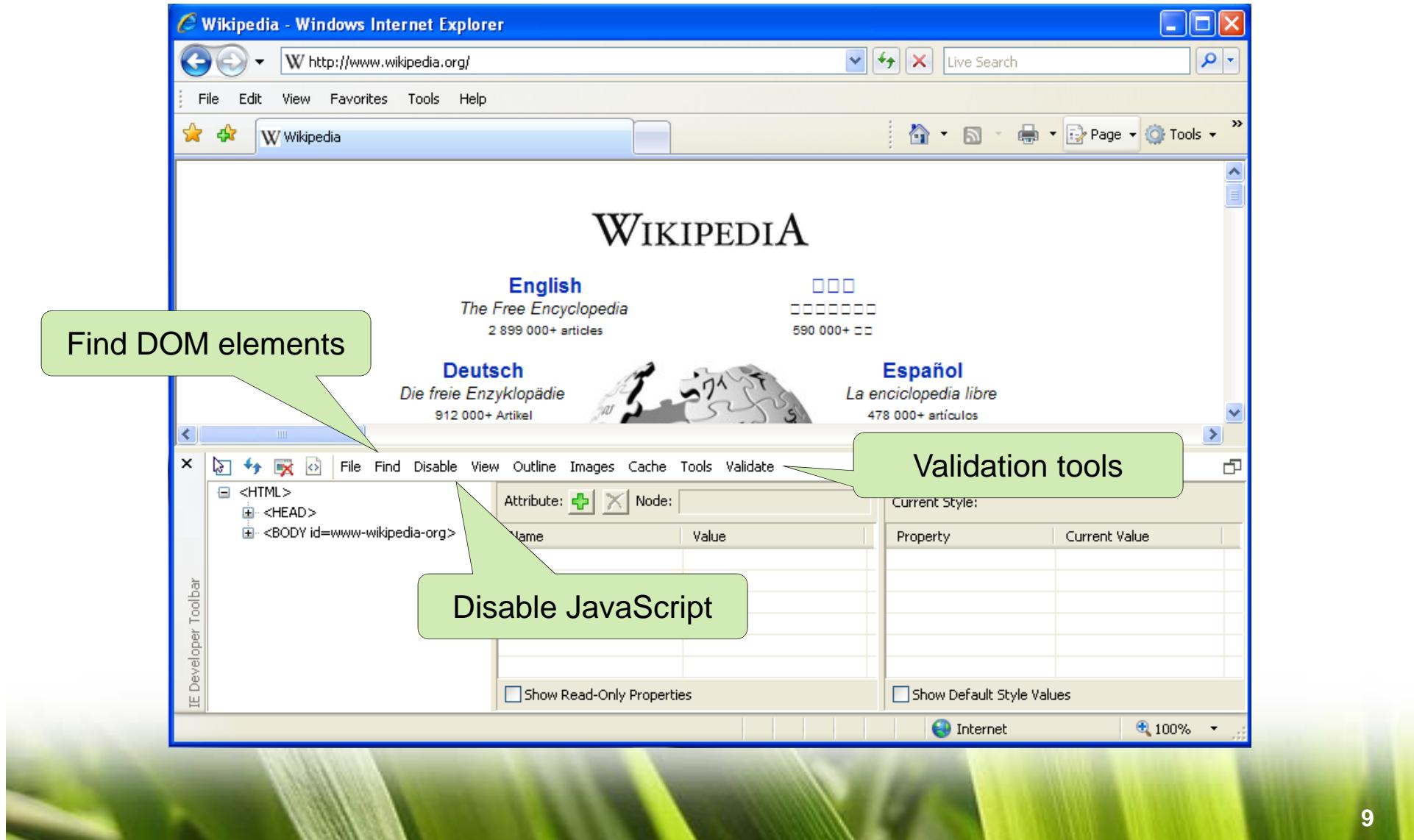
- Toolbar with many useful bookmarklets
 - Disable JavaScript, Cache, CSS
 - Display CSS by media type
 - View/delete domain cookies, session cookies
 - Populate form fields
 - Display Alt attributes on images
 - View JavaScript (convenient for searches)
 - Linearize page
 - Small screen rendering
 - Outline table cells
 - Validation tools (HTML, CSS, accessibility)
 - ...



- Windows-based tool that can be used with IE
 - Inspect HTTP requests and responses
 - View statistics
 - Pause HTTP traffic and make edits

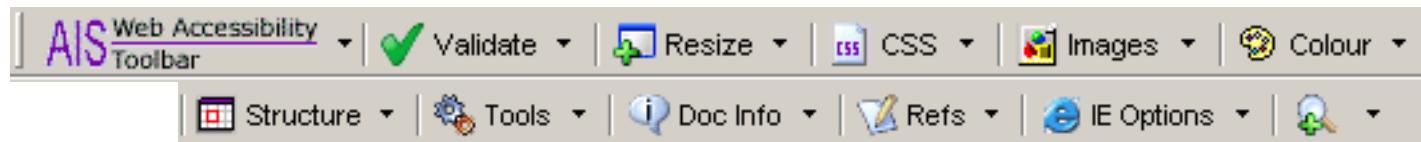


IE Developer Toolbar



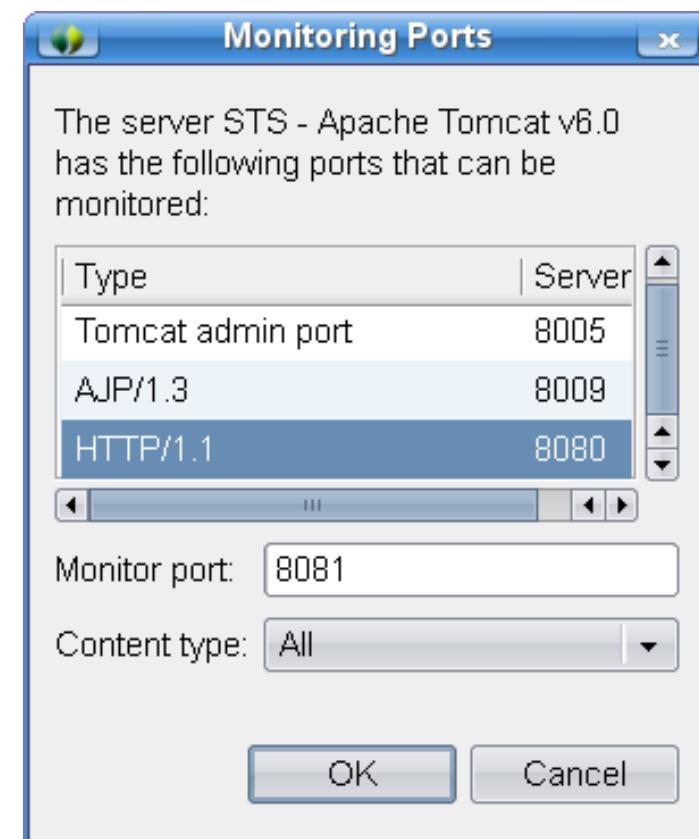
Web Accessibility Toolbar

- A very useful IE plugin to examine web pages for accessibility
 - Page structure
 - Use of Alt attributes
 - Linearizing
 - View as Lynx (text browser)
 - Page colors
 - Validation tools



HTTP Monitor (Eclipse)

- Enabled from the Servers view
 - right-click server, properties, monitoring
- Record every http request
- TCP/IP Monitor view
 - request/response headers
 - Content
- Complements Firebug
 - retains history of requests





Road Map

- Debugging Applications in a Browser
- Testing Web Applications
- Selenium
- Apache JMeter



Types of Tests

Unit tests

- one method at a time
- isolate dependencies (mocking or stubbing)

Integration tests

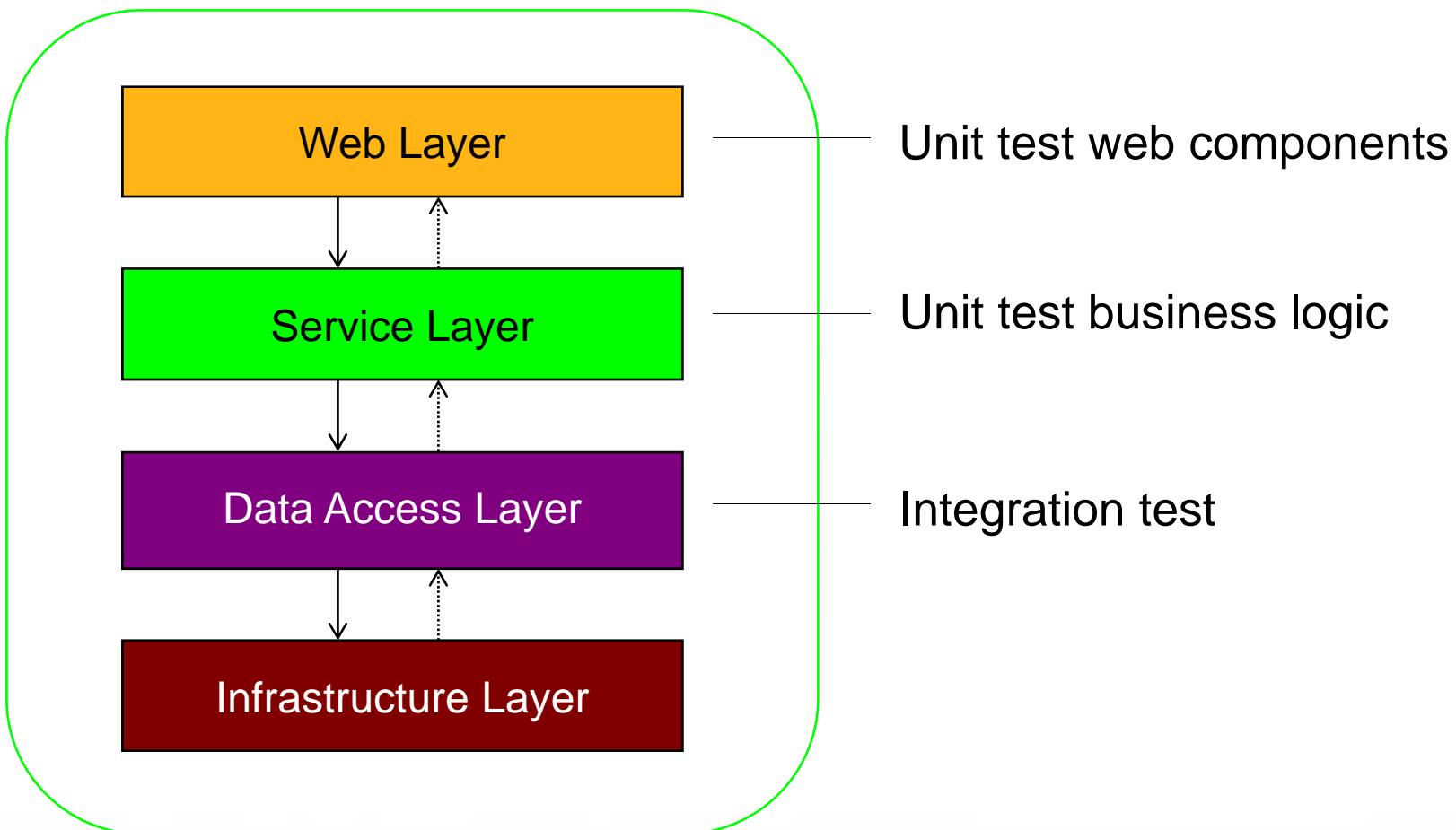
- a component interaction scenario
- allow dependencies (JDBC driver)

End-to-end tests

- exercise the product user interface
- fully integrated system



Testing Each Layer



Testing the Web Layer



- Unit test web components
 - controllers, interceptors, custom views
- Use stubs or mocks to isolate services
 - run fast and often
- What to test?
 - the logic adapting HTML requests to the service layer
 - input parameters, validation, etc.



End-to-End Testing



- Exercise the application running as a fully integrated system
- Unit/integration tests verify “no surprises” from underlying code
- There is more that requires verification
 - will application work when used from a browser?



What Still Needs Verification?



- Server-side configuration
 - servlet container
 - frameworks (Spring MVC, Tiles, JSF)
- Client-side behaviour
 - browser technologies
- Performance



Tools For Testing

- End-to-end testing
 - JWebUnit/HTMLUnit with embedded Jetty server
 - Selenium
- Client layer testing
 - JUnit, Crosscheck
- Performance testing
 - JMeter, Grinder





Road Map

- Debugging Applications in a Browser
- Testing Web Applications
- Selenium
- Apache JMeter



Selenium



- A tool for end-to-end functional testing
- Implemented in browser technologies
 - JavaScript, DHTML, IFrames
- Simple and effective
- Tested with various browsers on Windows, Mac OS, and Linux



How Selenium Works

- Uses JavaScript and IFrames to embed itself inside the browser
- Monitors user interaction with the browser
 - click a link
 - select an option
 - typed a value
 - a server request
- Test cases recorded in simple HTML tables



Test Cases

```
<table cellpadding="1" cellspacing="1" border="1">
    <thead><tr><td rowspan="1"
colspan="3">SearchAccounts</td></tr></thead>
    <tbody>
        <tr>
            <td>open</td>
            <td>
                /webtest-solution/admin/accountsearch/newSearch
            </td>
            <td></td>
        </tr>
        <tr>
            <td>clickAndWait</td>
            <td>link=Accounts</td>
            <td></td>
        </tr>
        <tr>
            <td>type</td>
            <td>searchString</td>
            <td>a</td>
        </tr>
    </tbody>
</table>
```

Test Suite

```
<body>
  <table cellpadding="1" cellspacing="1" border="1">
    <tbody>
      <tr><td><b>Test Suite</b></td></tr>
      <tr><td>
          <a href=". /SearchAccounts.html">Search Accounts</a>
        </td></tr>
      <tr><td>
          <a href=". /EditAccount.html">Edit Account</a>
        </td></tr>
      <tr><td>
          <a href=". /NewReward.html">New Reward</a>
        </td></tr>
    </tbody>
  </table>
</body>
```

Test Case

Selenium Concepts



- Actions
 - change the state of the application (click, type)
- Assertions
 - check if page title is X
- Element locators
 - Which HTML element a command/assertion refers to?
- Patterns
 - can be used for specifying expected values



Selenium Core Tools

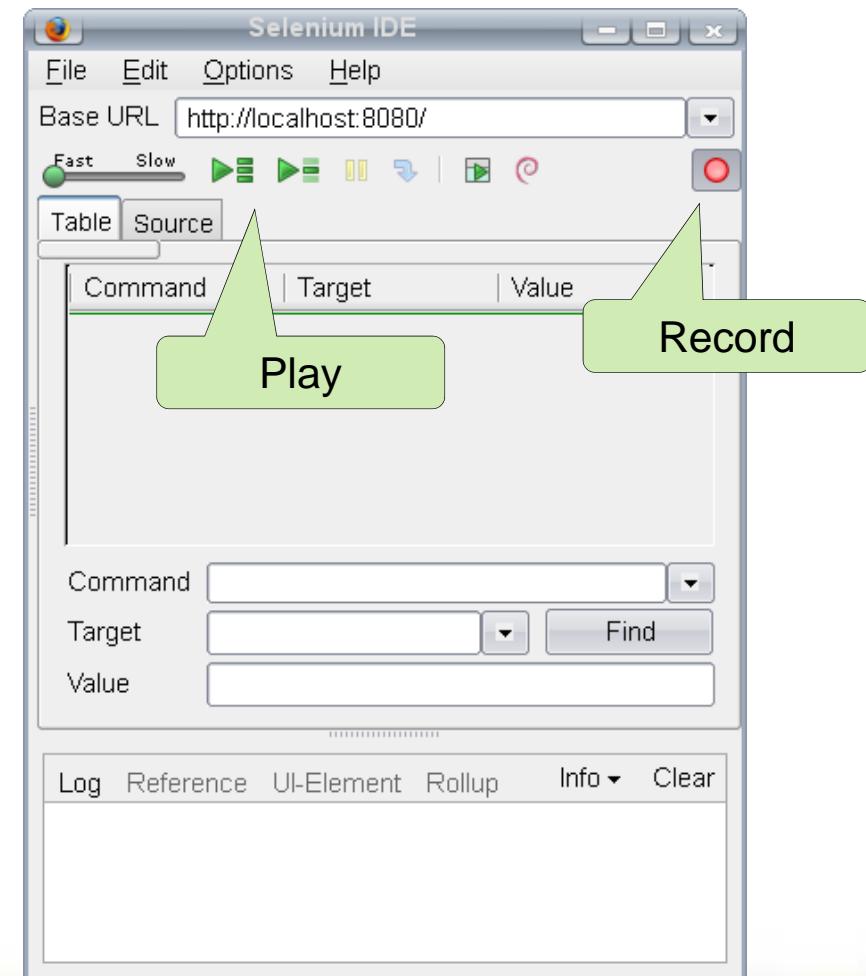


- There are several Selenium tools
 - Selenium IDE
 - Selenium Core
 - Selenium Remote Control
- Different deployment strategies



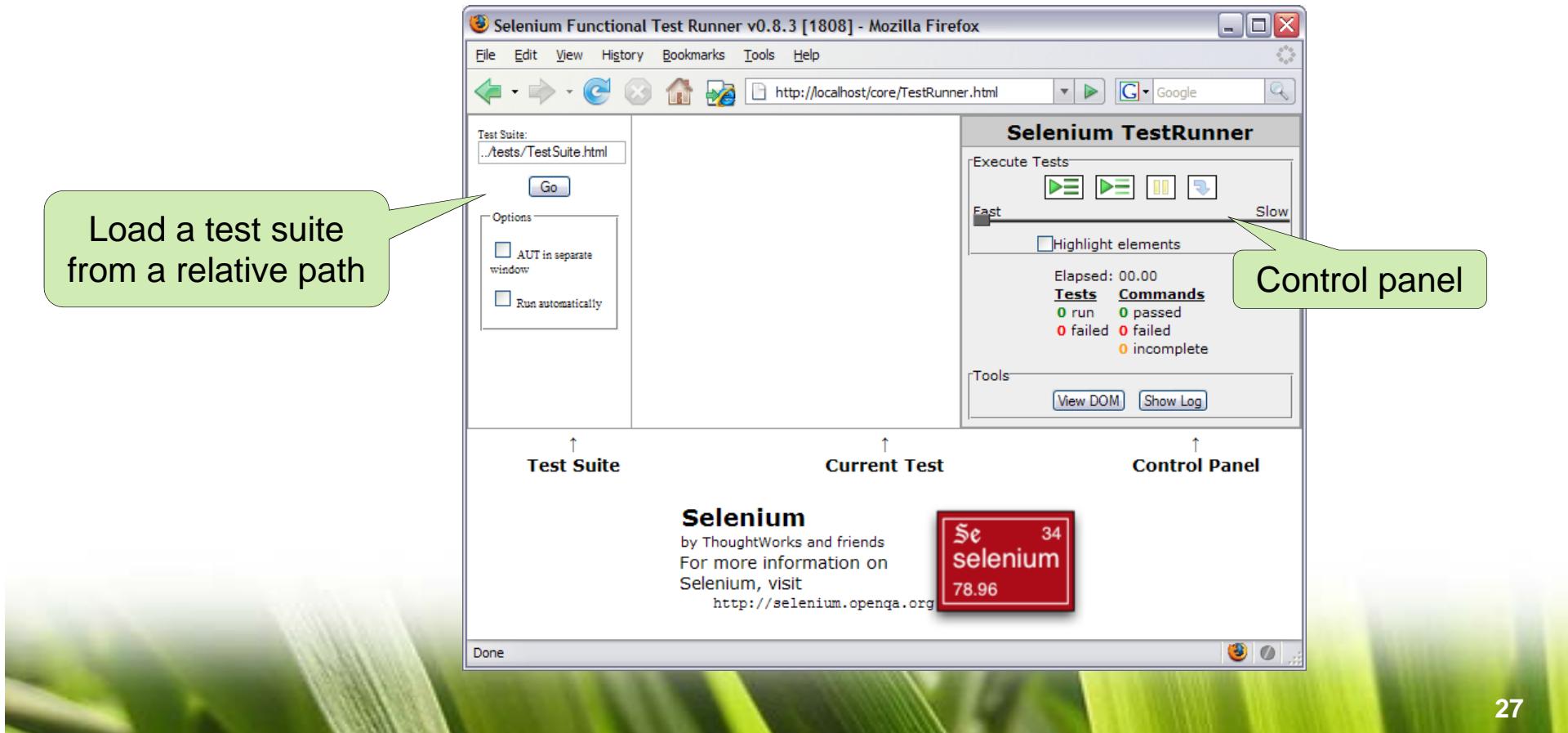
Selenium IDE

- A Firefox extension
 - includes Selenium Core
- Features
 - record and playback tests
 - code completion for commands
 - debug tests
 - help with field selection
- Useful for recording tests
 - not great for automation

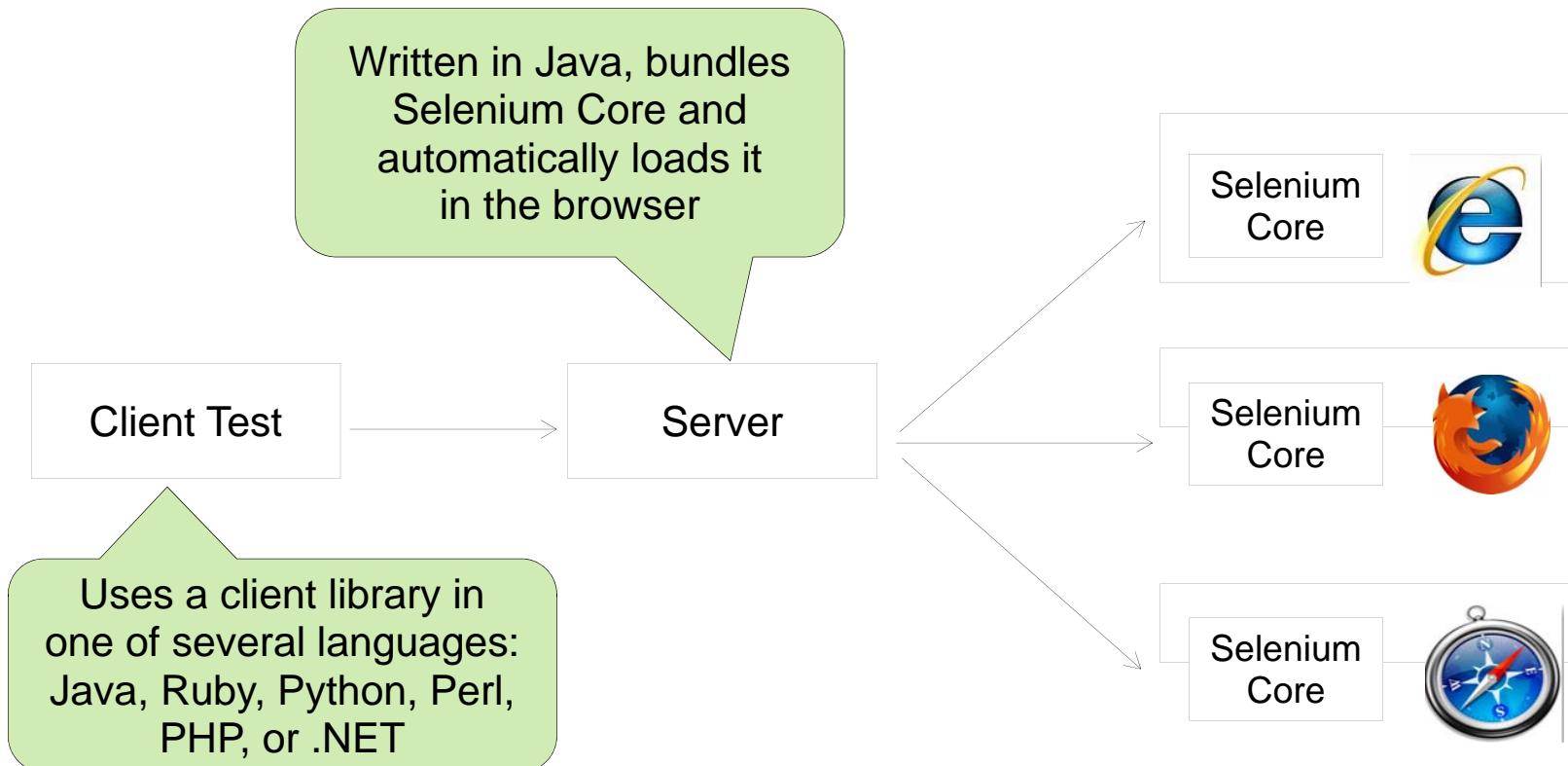


Selenium Core

- Install Selenium Core on the server side
- Load the Test Runner page



Selenium Remote Control



Note: Selenium RC can be plugged into continuous integration, can test different browsers and platforms, and save results.



Road Map

- Debugging Applications in a Browser
- Testing Web Applications
- Selenium
- Apache JMeter



Apache JMeter Overview



- Provides a graphical user interface
 - 100% Java desktop application
- Allows recording a test plan by acting as an HTTP proxy between browser and server
- Provides great flexibility in simulating loads



What Can You Use It For?



- Test performance of a diverse set of server types
 - HTTP, SOAP, JMS, POP3, ...
- Functionally test applications
 - create assertions to verify server response
- Flexible and very customizable
 - many pre-built components





How It Works

- Takes the place of a browser
 - but it is NOT a browser!
- Sends requests to the server
- Can simulate simultaneous user requests



Apache JMeter Test Plan



The screenshot shows the Apache JMeter interface with various components highlighted:

- Test Plan**: The main menu bar at the top.
- Thread Groups**: A yellow box highlighting the "Thread Groups" node in the left sidebar.
- Controllers**: A yellow box highlighting the "Random Controller" under the "Thread Group" node in the left sidebar.
- Assertions**: A yellow box highlighting the "Response Assertion" under the "HTTP Request" node in the left sidebar.
- Samplers**: A yellow box highlighting the "HTTP Request", "SOAP/XML-RPC Request", and "JDBC Request" nodes in the left sidebar.
- Listeners**: A yellow box highlighting the "Aggregate Report" and "View Results in Table" nodes in the left sidebar.
- Load Parameters**: A yellow box highlighting the "Number of Threads (users)", "Ramp-Up Period (in seconds)", and "Loop Count" fields in the right-hand configuration panel.

The right-hand panel displays the configuration for a "Thread Group":

- Name:** Thread Group
- Comments:** Action to be taken after a Sampler error
 - Continue
 - Stop Thread
 - Stop Test
- Thread Properties**
 - Number of Threads (users):** 50
 - Ramp-Up Period (in seconds):** 1
 - Loop Count:** Forever
 - Scheduler

Thread Groups

- Starting points of a test plan
- Control number of threads for a test
 - Thread count
 - Ramp-up period
 - Loop Count



Samplers

- Samplers perform the actual work
 - HTTP Request
 - FTP Request
 - JDBC Request
 - SOAP Request
 - LDAP Request
 - Java Request
 - more...



Logic Controllers



- Organize and control samplers
 - Random Controller
 - Simple Controller
 - Interleave Controller
 - more...
- It's possible to nest controllers with samplers as leaf elements in the test plan tree



Other JMeter Components



- **Listeners** read, save, and view test results
 - Simple Data Writer, Aggregate Report, Assertion Results, more...
- **Assertions** verify results from a sampler
 - Response, Duration, Size, and other assertions
- **Post Processors** invoked after a sampler
 - Regular Expression, XPath Expression, and other post processors



Recording Test Plans



- Set browser HTTP proxy preferences
- Create HTTP Proxy element in JMeter and start the proxy
- Use the browser to exercise the application
- Watch and modify requests as they get added to test plan



Dynamic Parameter Values



- Test plans record parameter values to be sent to the server
- Some parameter values need to be dynamic
 - e.g. flow execution key
- Use Regular Expression Post-Processor
 - extract parameter from the last response
 - save as a variable
 - use variable in subsequent sampler requests





Summary

After completing this lesson, you should have learnt that:

- Firefox provides many tools for developers
 - JavaScript debugging, logging
- Web application testing is essential part of a developer's responsibilities
- Selenium is a simple but capable tool for functional testing
- Apache JMeter is a versatile tool for performance testing





LAB

Functional and Performance
Web Application Testing