Spring

Enterprise Java Made Easier

Ryan Cutter
November 2011

About Me

- First-semester CAETE Graduate student
- Used Enterprise JavaBeans, Servlets, etc from late 1990s until 2002
  - Not ideal for large application development...
- Flew in the U.S. Navy from 2003 until 2011
- Resumed Java development last year
  - Perhaps the greatest advancement in enterprise application development during my time away was Spring!
    - Maybe StackOverflow, too ;-)

Agenda

- What is Spring?
- Beans and Dependency Injection
  - Configuring with XML
  - Autowiring, Annotations, and Autodiscovery
- Aspect-Oriented Programming (AOP)
- Data Access / Object-Relational Mapping (ORM)
- Model-View-Controller framework (MVC)
- Building RESTful applications
- Other Spring components

- Along the way we'll code some simple apps to demonstrate bean wiring, AOP, MVC and REST

What is Spring?

- Enterprise JavaBeans (EJBs) made important strides in server-side enterprise services but also created discontent
  - Proved to be too unwieldy
  - Plain old Java object (POJO)-centric frameworks like Spring rapidly supplanted EJB as the true Java standard
- Spring was created by Rod Johnson in *Expert One-on-One: J2EE Design and Development* (2002) and released soon thereafter
- Open source framework dedicated to principals of:
  - Simplicity
  - Testability
  - Loose coupling
- *Spring simplifies Java development*

What is Spring?

- Lightweight development with POJOs
  - No more heavy/invasive demands from EJBs, etc
- Loose coupling through dependency injection (DI) and interface orientation
  - Objects given dependencies at creation time
- Declarative programming through aspects and common conventions
  - Aspect-oriented programming (AOP) captures functionality in reusable components
- Boilerplate reduction through aspects and templates

No, really: What is Spring?

- Spring Basics
  - Dependency Injection (DI)
  - Aspect-Oriented Programming (AOP)
- Core support for application development
  - Data persistence
  - Transaction management
  - Spring MVC (web framework)
  - Spring Security
- And so on (more functionality being rolled out all the time)
  - Spring Web Flow
  - Remote services
  - Messaging
  - RESTful resources

Beans - Containers

- Containers are the core of Spring Framework
  - Objects' lifecycles managed here cradle to grave
- Use DI to manage application's components
- Makes objects easier to understand, reuse, and test
  - ie, Wires the beans!
- Two kinds of containers:
  - Bean factories - Simple, low level
  - Application contexts - More commonly used

Beans

- Spring's most basic operation is wiring beans (slang for DI)
- Wiring sometimes accomplished using XML files
  - I know what you're probably saying but keep in mind *the developer decides how much to rely on XML*
    - Spring 3 offers an almost no XML implementation
  - XML file contains configuration management for all components which associates beans with each other
- Let's see this in action
  - Create animals in a Zoo and wire their corresponding beans
    - The Cheetah class on next page looks a little goofy but bear with me...it will be used to explain multiple concepts

Beans - The Zoo

```
package org.ryancutter.zoo;
public interface ZooAnimal {
    void talk() throws ZooAnimalException;
}
-----
package org.ryancutter.zoo;
public class Cheetah implements ZooAnimal {
    private int speedMPH = 10;

    public Cheetah() {}
    public Cheetah(int speedMPH) { this.speedMPH = speedMPH; }
    public void talk() throws ZooAnimalException {
        System.out.print("I am Cheetah, hear me roar");
    }

    private int numChildren;
    public int getNumChildren() { return numChildren; }
    public void setNumChildren(int numChildren) {
        this.numChildren = numChildren;
    }
}
```

Beans - The Zoo

[zoo.xml]

```
<?xml version="1.0" encoding="UTF-8"
<beans  xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
   <bean id="cary" class="org.ryancutter.zoo.Cheetah">
      <constructor-arg value="30" />
      <property name="numChildren" value="4" />
   </bean>
</beans>
```

- The <beans> element contains some standard Spring namespace schema
- cary the Cheetah is defined in the <bean>
  - Pass args into constructor with <constructor-arg>
  - Inject properties via getter/setters with <property>
    - Spring will convert type as needed (String -> int)

Beans - The Zoo

```
ApplicationContext ctx = new
    ClassPathXmlApplicationContext("org/ryancutter/zoo/zoo.xml");
ZooAnimal animal = (ZooAnimal) ctx.getBean("cary");
animal.talk();
```

- ClassPathXmlApplicationContext is just one of several ways to load context definitions
  - When building web apps, XmlWebApplicationContext will probably be used
- Referencing other beans is easy. Let's record a favorite toy by adding a private Toy attribute named "favoriteToy" with a getter and setter. ChewToy is of type Toy.

```
<bean id='chewtoy" class="org.ryancutter.zoo.ChewToy" />

<bean id="cassie" class="org.ryancutter.zoo.Cheetah">
        <constructor-arg value="20">
        <property name="numChildren" value="1" />
        <property name="favoriteToy" ref="chewtoy" />
</bean>
```

Beans - Wiring at runtime

- Complex applications certainly will need dynamic wiring
- One way to do with is with the Spring Expression Language (SpEL)
  - Wires values into bean property/constructor arguments using expressions evaluated at runtime
  - #{ } markers contain SpEL expressions
  - Let's create another Cheetah ("chester") that copies the number of children from our previous bean ("cary") using SpEL

```
<bean id="chester" class="org.ryancutter.zoo.Cheetah">
   <property name="numChildren" value="#{cary.getNumChildren()}" />
</bean>
```

Beans - Autowiring

- Using pure XML to configure doesn't necessarily scale
- Autowiring reduces `<property>` and `<constructor-arg>` elements
  - Four ways to automatically wire beans:
    - byName
      - Match properties with beans of same name
    - byType
      - Match properties with beans whose types are assignable
    - constructor
      - Match constructor with beans whose types are assignable to constructor arguments
    - autodetect
      - constructor first, then byType

Beans - Autowiring byName

```
<bean id='favoriteToy" class="org.ryancutter.zoo.ChewToy" />

<bean id="cole" class="org.ryancutter.zoo.Cheetah" autowire="byName">
        <constructor-arg value="20" />
        <property name="numChildren" value="1" />
</bean>
```

- byName autowiring establishes convention where property will automatically be wired with bean of same name
  - Consider all properties of "cole" the Cheetah (speedMPH, numChildren, favoriteToy) and look for beans declared with same name as properties
    - Cheetah.setFavoriteToy(Toy) will be called with ChewToy

Beans - Annotations

- Using annotations to automatically wire beans widely used
  - Similar to autowire XML attribute but more functional
  - Not enabled by default - requires some configuration
  - @Autowired is Spring-specific but JSR-330 (standards-based DI) supported as well with @Inject

```
@Autowired
public void setFavoriteToy(Toy toy) {
    this.favoriteToy = toy;
}
[ or ]
@Autowired
private Toy favoriteToy;
```

- Both do same thing, will initiate byType autowiring to find bean of type Toy.

Beans - Autodiscovery

- Like annotations, autodiscovery requires some configuration to use
- *Further reduces reliance on XML*
- `@Component` one of several special stereotype annoations
  - General-purpose indicating class is Spring component

```
package org.ryancutter.zoo
import org.springframework.stereotype.Component;

@Component
public class ChewToy2 implements Toy {}
```

- When zoo package is scanned by Spring, it will register chewtoy bean automatically
  - `@Component("name")` will declare bean name

AOP

- OOP is not suited for use in all circumstance
  - Consider how *cross-cutting concerns* (like security and logging) are integrated into large web applications
    - A cross-cutting concern is any functionality that affects multiple parts of an application
    - Separating these challenges from business logic is the heart of aspect-oriented programming (AOP)
- Aspects are an alternative to inheritance and delegation
- Define common functionality once and declaratively define how and where it is applied
  - Modules much cleaner and focus on primary concern

AOP

- Like OOP, AOP would take a long time to explain...
- Just realize we're injecting behavior into different points of a program's execution sequence
- Along with Spring, other big AOP frameworks include AspectJ and JBoss
  - Spring borrows liberally from AspectJ
- Next example shows things we'd like to do before and after a golf stroke

AOP - An Example

```
package org.ryancutter.golf;

public class GolfSwing {
    public void lineUpShot() { // before shot
        System.out.println("I am lining up my shot");
    }

    public void track() { // after shot
        System.out.println("I am tracking the ball to make sure I can find it");
    }

    public void curse() { // after bad shot
        System.out.println("#*$*&# @&@");
    }
}
```

AOP - An Example

```
<bean id="golfswing"
    class:"org.ryancutter.golf.GolfSwing" />

<aop:config>
 <aop:aspect ref="golfswing">
  <aop:before pointcut= "execution(* org.ryancutter.golf.Golfer.golf(..))"
      method="lineUpShot" />
  <aop:after-returning pointcut="execution(*org.ryancutter.golf.Golfer.golf(..))"
      method="admire" />
  <aop:after-throwing pointcut="execution(* org.ryancutter.golf.Golfer.golf(..))"
      method="curse" />
 </aop:aspect>
</aop:config>
```

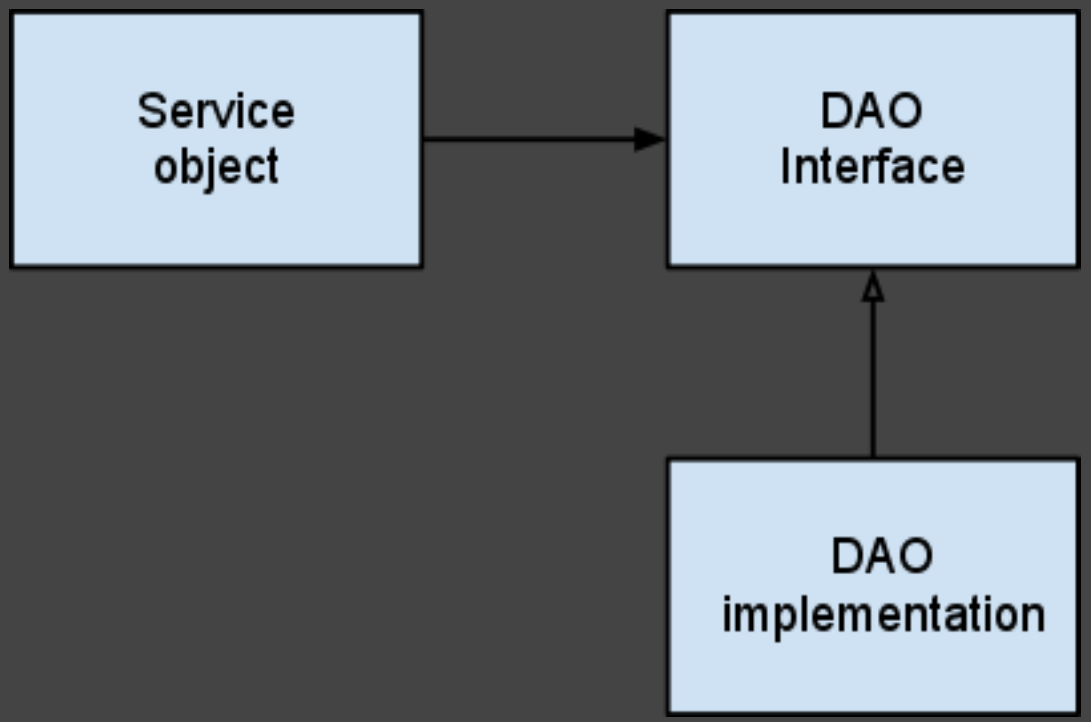
- pointcut defines join points (where/when *advice* is executed)
  - Written in AspectJ's pointcut expression syntax
- *Advice* defines the action to be taken (the method)
- An <aspect> is the combination of pointcut and advice

AOP - Further uses

- This is a trivial example
  - A more advanced app would want parameters passed into advice
  - As with DI, XML reliance can be reduced by annotation
- Ultimately, Spring AOP is weak compared to AspectJ's capabilities
  - Suitable for some applications but AOP-intensive applications might want to use AspectJ
  - AspectJ aspects are largely independent of Spring so both can be used simultaneously
  - Common to inject dependencies into AspectJ aspects
    - Let's save that one for another lecture...

Data Access

- Data access object (DAO) provides means to read/write data to the database
- Service objects access DOA through interface
  - In fact, Spring constantly encourages interface reliance



Data Access

- Spring separates fixed and variable portions of data access process into classes:
  - Templates - Fixed
    - Connect to data, start transaction, commit, close
    - Configured as bean or use support classes
  - Callbacks - Custom data access
    - Execute transaction, return data
    - This is where we really code the logic
- Spring's JDBC template framework eliminates all that custom coding once required
  - No more connection handling, statement creation, exception handling code
  - Free to focus on what you wanted to do in the first place

Data Access - ORM

- Simplified JDBC is nice but building complex applications probably requires object-relational mapping (ORM) services
- Spring supports Hibernate, iBATIS, JDO, and JPA
  - Hibernate is a very popular complement to Spring
- As with JDBC, Spring offers templates to manage Hibernate sessions, catching framework-specific exceptions, etc
  - However, `HibernateTemplate` no longer considered optimal with introduction of contextual sessions
  - Simply wire a Hibernate session directly into your DAO
  - Spring's Hibernate session factory beans provide access to Hibernate's `SessionFactory`
    - As usual, the coder can decide level of XML reliance
    - Not sufficient time for a code example here

MVC

- Spring's web framework is based on the Model-View-Controller (MVC) pattern
  - This is another topic which is the subject of many books but in general....
  - Request goes to Spring's `DispatcherServlet` where it is sent to a Spring MVC controller (based on handler mapper) to be processed
  - Controller packages up model data and identifies the appropriate view before `DispatcherServlet` renders the result (often a JSP but could be other things)

MVC - An example

- Let's implement a "Hello World" example in which a static page is served up
  - Assume HelloService is a class that simply returns a "Hello World" String when asked getGreeting()
- If we declare a DispatcherServlet in web.xml called "hello", this might be hello-servlet.xml:

```
<beans [...lots of schema omitted...]>
   <mvc:resources mapping="/resources/**" location="/resources/" />
   <context:component-scan base package="org.ryancutter.hello.mvc" />
</beans>
```

- mvc:resources states content will be served from /resources
- context:component helps get our class automatically discovered and registered as a bean

MVC - An example

```
package org.ryancutter.hello.mvc;
[...necessary imports...]

@Controller
public class HelloController {
    private HelloService helloService;
    @Inject
    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }

    @RequestMapping(("/"))
    public String showHelloPage(Map<String, Object> model) {
        model.put("greeting", helloService.getGreeting());
        return "hello";
    }
}
```

- @Controller means this is a controller class
- @Inject injects HelloService when controller is instantiated
- @RequestMapping identifies showHelloPage as a request-handling method for all requests to /

MVC - An example

- DispatcherServlet must consult a view resolver to serve output to user
- View resolver maps view name to JSP (although Velocity or other view technologies can be employed)
- Many different kinds of Spring view resolvers but this example will use InternalResourceViewResolver
- Add to hello-servlet.xml:

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

- hello.jsp should be in /WEB-INF/views/ with appropriate code to serve "Hello World" in greeting element

MVC

- Obviously this is a ludicrously simple example but presented for time constraints
- Spring MVC offers powerful tools to build complex web layers out of near-POJOs
  - Controllers and view resolvers enjoy loose coupling
    - Controllers delegate to other beans using dependency injection
    - Handler mappings choose controller, view resolvers choose how results displayed, and the two operate independently

REST

- Spring 3 moved aggressively to support Representational State Transfer (REST)
- REST is a simpler alternative to SOAP whose popularity has increased substantially of late
- REST is focused on transferring state of resources
  - http://www.ryancutter.org/HelloWorld/greetings/19
    - Use HelloWorld servlet to access greeting's 19th resource
      - Perhaps greeting #1 is "Hello World" and #19 is "Hola Mundo"
- Spring REST correlates tightly with Spring MVC so previous example needs only to be slightly modified to be RESTful
  - Here, HelloWorld holds greetings in different languages

REST - An example

```
package org.ryancutter.hello.mvc;
[...necessary imports...]

@Controller
@RequestMapping("/greetings")
public class HelloController2 {
    private HelloService helloService;
    @Inject
    public HelloController2(HelloService helloService) {
        this.helloService = helloService;
    }

    @RequestMapping(value="/{id}", method=RequestMethod.GET)
    public String getGreeting(@PathVariable("id") long id, Model model) {
        model.addAttribute(helloService.getGreetingById(id));
        return "greetings/view";
    }
}
```

- Recall @RequestMapping indicates which requests to handle
  - getGreeting() method will respond to GETs and grab {id} from URL placeholder

REST

- Spring MVC controllers can field requests to manipulate RESTful resources
- While we only looked at a simple GET, Spring can obviously create controllers to handle POST, PUT, and DELETEs too
- Spring can represent data in a format preferred by the client
  - `ContentNegotiatingViewResolver` can select best view in view-based responses
  - Annotations in controller handler methods can assist with converting returned values into responses
- `RestTemplate` provides template-based methodology to consuming RESTful resources

Other Spring components

- Spring is a vast, sprawling framework with many components that couldn't be discussed today
- Other key elements of Spring:
  - Transactions: Spring enables developer to declaratively apply transactional policies in objects using AOP
  - Web Flow: Spring Web Flow framework builds conversational, flow-based web applications
  - Security: Spring has a full featured security layer
  - Remote Services: Support for remoting technologies is baked in
  - Messaging: Spring can be used with JMS to asynchronously message between applications

Spring - In closing

- At its heart, Spring strives to be a loosely coupled framework dedicated to making Java easier to use
- In early Spring releases, focus on heavy, XML-centric configuration structures turned some potential adopters off
  - Recent Spring versions continue to remove the focus on XML and allow greater flexibility
- Developers wishing to employ DI and AOP in a complex web application would be hard pressed to find a more full, rich set of capabilities
  - Spring's ability to be used in components allows for side-by-side implementation with other technologies

Questions?

Check out http://www.springsource.org or Spring in Action (3rd Ed) by Craig Walls for more information