WA2607 Comprehensive Spring Boot/Cloud

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Table of Contents

Chapter 1 - Introduction to the Spring Framework

Objectives
Key objectives of this chapter:

- ■ What is the Spring Framework?
- ■ Spring Modules
- ■ Role of Spring Container

1.1 What is the Spring Framework?

- ■ Spring is an open-source Java platform that provides infrastructure support for Java applications

- ■ Spring applications are more

  - ◇ Testable

  - ◇ Flexible

  - ◇ Easier to maintain

  - ◇ Can be used with existing frameworks such as JSF, Struts

- ■ Spring provides many things that are common to many Java applications so you can concentrate more on what is different about your application

What is the Spring Framework?

The Spring Framework was created by Rod Johnson. It has been an open source project since February 2003.

See http://projects.spring.io/spring-framework/ for more information.

1.2 Spring Philosophies

- ■ Java EE should be easy to use

  - ◇ Spring started as an alternative to EJB technology when EJB programming was more difficult and had odd restrictions that had to be followed

  - ◇ The Java EE programming model has been simplified in the last

several versions because of the popularity of frameworks like Spring

- Object-oriented design is more important than the underlying technology

- It is better to code to interfaces than classes

  ◇ This creates a "loosely coupled" application where what is important is not the implementation of a component but the functionality it provides to other components

- Applications should be easy to test

  ◇ A loosely coupled application can be more easily tested in isolation

- JavaBeans are a good way of assembling applications

Spring Philosophies

The creators of Spring believe that good design is more important than the underlying technology, whether it be EJB or otherwise. EJB and other frameworks force developers to implement applications in a certain (restrictive) way. The goal of Spring is to remove these restrictions and allow developers to implement their objects without thinking about the underlying infrastructure.

Additionally, it should be easy to test your code. Binding your code to a particular technology can make this difficult. For example, EJB forces you to redeploy to the EJB container in order to run tests.

Spring facilitates easier coding to interfaces. This makes your application loosely coupled, easier to maintain and easier to test. Objects no longer have to look up their dependencies as the Spring container will inject their dependencies into them.

JavaBeans is a simple specification. There isn't a lot that a developer needs to know to create a simple JavaBean. This allows them to focus on business logic rather than being concerned with infrastructure. JavaBeans combined with IoC is a powerful model.

1.3 Why Spring?

- Components in enterprise applications require services such as transactions, security and distributed computing

- As much as possible, these services should be kept separate from application business logic

- EJB was created to address this need and make enterprise application development easier

- While EJB solved many of these issues, it introduced new complications

- ◇ Many of these issues were solved with EJB 3.0 but existed when Spring was first developed and gained popularity

- ■ EJBs also required the use of a Java EE "application server" environment

  - ◇ With Spring you can leverage the benefits in "standalone" Java applications like Applets or Java Swing GUI programs

  - ◇ When Spring was first created many of the J2EE Application Servers had a reputation as "heavyweight" also

- ■ Spring also provides many utility functions not found in the Java EE specifications

1.4 Spring Modules
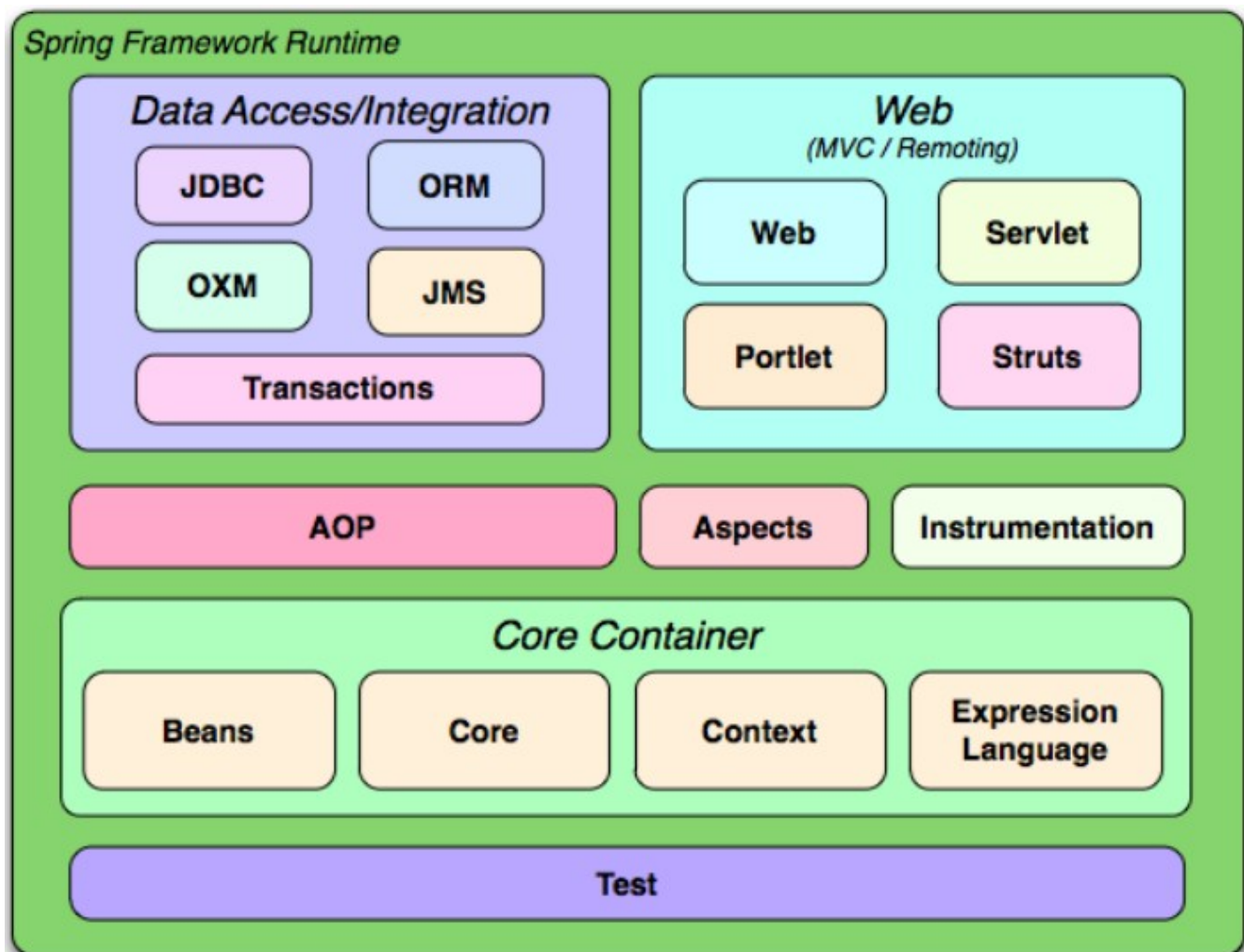
Spring Modules

The above diagram comes from the current Spring 3.0 documentation.

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the above diagram.

An application can utilize one of the modules without having to use other modules. For example, an application could use the Spring AOP module but not use Spring MVC.

Modules can build on one another as illustrated on the slide. All modules are built on the core container and utilities module.

- **Core Container** – Contains the core functionality of Spring. This includes the Core (IoC and Dependency Injection features), Beans (BeanFactory), Context (ApplicationContext), and Expression Language modules (querying and manipulating an object graph at runtime).

- **Data Access/Integration** - The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

    ◦ JDBC – Provides facilities that allow developers to write code that does not contain complex JDBC setup/teardown/exception handling logic. Also contains an enhanced exception hierarchy with more descriptive database error messages for many database vendors.

    ◦ ORM – This module facilitates easy integration with some popular ORM frameworks including JPA, Hibernate, iBATIS and JDO.

    ◦ OXM – This module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

    ◦ JMS – This module contains features for producing and consuming messages.

    ◦ Transaction – This module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs*.

- **Web** – This *Web* layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

    ◦ Web – This module featuring initialization of the IoC container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring's remoting support.

    ◦ Web-Servlet – This module contains Spring's model-view-controller (*MVC*) implementation for web applications.

    ◦ *Web-Struts* – This module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.

- *Web-Portlet* – This module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

- **AOP and Instrumentation**

  - AOP – Provides support for aspect-oriented programming and metadata programming using source code level annotations.

  - Aspects – Provides separate integration with AspectJ.

  - Instrumentation – This module provides class instrumentation support and classloader implementations to be used in certain application servers.

1.5 Requirements and Supported Environments

- One of the major changes is that Spring 3.0 requires a Java SE 5 or Java SE 6 environment

  - Spring 3.0 will not work in a Java JDK 1.4 environment or earlier

- Much of the Spring codebase has been revised to take advantage of Java 5 features like Generics and annotations

  - You will find methods in the Spring API like the following from the BeanFactory interface that use generics for strongly typed return values

```
T getBean(Class<T> requiredType)

T getBean(String name, Class<T> requiredType)

Map<String, T> getBeansOfType(Class<T> type)
```

1.6 Using Spring with Servers

- Spring 3.0 is compatible with Java EE servers as long as they also use Java SE 5 or above

  - WebLogic Server 9.x, 10.x, 11g, 12

  - WebSphere 6.1, 7.0, and 8.0

  - JBoss 4.2, 5.0, 5.1, 6.0, and 7.0

  - Glassfish v1, v2, and v3

- For those that may want to use a non-Java EE server environment Spring

3.0 is also compatible with some of the servers provided or supported by SpringSource

- ◇ SpringSource tc Server 6.0.19A and later

- ◇ SpringSource dm Server 2.0

- ◇ Tomcat 5.0, 5.5, 6.0, and 7.0

1.7 Role of Spring Container

- Typically in a Spring application, if you look just at the code of the application itself, there may seem to be something missing

  - ◇ All of your application components are very generic and loosely coupled but the question arises "How do they link up with each other?"

- The missing piece is the role of the Spring container, which takes the code of your application components, along with **configuration meta-data** and configures an environment where component instances are initialized and linked together according to this configuration

- The configuration of a Spring application can be provided in multiple ways

  - ◇ XML files – This was the original way to provide configuration

  - ◇ Annotations in source code – Now that Spring 3 requires Java SE 5 this is more popular

  - ◇ Java @Configuration classes – This used to be a separate project to allow this but is now part of the regular Spring framework

1.8 Spring Example

- This simple example will show one of the core features of Spring, Dependency Injection

- The core of this example is built around an interface for a component that will calculate taxes

```
public interface TaxCalculator {
     public double calculateTax(double amount);      }
```
- Of course there will also need to be a implementation of the interface to

actually do anything

```
public class SalesTaxCalculator implements TaxCalculator {
        private double taxRate = 0.05;      // default rate
        public double calculateTax(double amount) {
                return amount * taxRate;
        }       }
```

- There is also another Java component that uses the TaxCalculator component

  ◇ Notice that this class does not refer to the SalesTaxCalculator class

```
public class Register {
        private TaxCalculator taxCalc;
        public double computeTotal(double beforeTax) {
                return beforeTax +
taxCalc.calculateTax(beforeTax);
        }       }
```

1.9 Spring Example

- Looking at the code from the previous slide there are some questions:

  ◇ How do we avoid a NullPointerException in the Register class when using the 'taxCalc' field?

  ◇ How is the SalesTaxCalculator class used to calculate the tax when the Register class doesn't refer to it?

- The missing piece is the configuration that can be provided to the Spring container to link the two components

  ◇ The example below is simplified somewhat but you can see the Spring configuration indicates the 'salesTaxCalc' bean should be used as the 'taxCalc' property of the 'register' bean

```
<beans ... >
        <bean id="salesTaxCalc" class="SalesTaxCalculator" /
>
        <bean id="registerBean" class="Register">
                <property name="taxCalc" ref="salesTaxCalc" /
>
        </bean>
```

```
</beans>
```

Spring Example

The Spring XML syntax given would also assume there is a 'setTaxCalc' method in the 'Register' class although this is not the only way to link the two components.

1.10 Spring Example

- You could then initialize a Spring environment using the above configuration and obtain the "registerBean" from Spring with the link to the other component already initialized

```
ApplicationContext context = new
      ClassPathXmlApplicationContext("springConfig.xml");
// get the bean
Register register =
      context.getBean("registerBean", Register.class);
// use the bean
double total = register.computeTotal(18.45);
```

- Besides providing XML configuration you could also:

  ◇ Use Spring annotations like @Component to declare the Java classes as a Spring component and @Autowired to initialize the 'taxCalc' field

  ◇ Use Spring Java Configuration with the @Configuration annotation on a class to indicate it executes Java code to initialize Spring components and the @Bean annotation on methods that execute Java statements and return configured Spring components

1.11 Avoiding Dependency on Spring

- One of the goals of Spring is to be "non-intrusive" so it should be possible to use the code that you write with another framework that provides similar behavior and move away from Spring if desired

- Although this might not be a purely painless prospect there are a few things you can do to help ensure this:

  ◇ Make as many Java classes and components contain only "standard" Java code

- The classes from the example accomplished this
  ◇ If you reference Spring from your code try to limit it to annotations only
    - Even annotations would mean a Spring 'import' statement in your code but annotations are just added at the class/method/field level to enable certain Spring features
    - Other frameworks might provide similar annotations that would be a replacement
  ◇ Avoid extending Spring classes or implementing Spring interfaces with your code
    - This would create code that is more directly linked to Spring
    - If you need to do this "hide" these classes from other parts of the application to limit the amount of code that is Spring-aware

1.12 Additional Spring Projects/Frameworks

- Spring also provides many additional projects and frameworks for things like web applications, web services, etc
  ◇ Spring MVC – A web application framework part of the core "web" module
  ◇ Spring Web Flow – A separate project with a model for "wizard-like" web application interactions
  ◇ Spring Security – A security framework for web or non-web Java applications
  ◇ Spring Web Services – A Spring implementation of web services
  ◇ Spring Integration – A Spring implementation of Enterprise Integration patterns
  ◇ Many others - http://spring.io/projects
- If you want to avoid dependence on Spring look to see if you can implement the same functionality with Java standards without depending on Spring

1.13 Summary

- Spring is a modular Java application framework

- One of the prime functions is the ability to instantiate and configure Java components that can then be linked into other parts of the application

Chapter 2 - Spring Annotation Configuration

Objectives

Key objectives of this chapter:

- Annotation-based container configuration

- Wiring Beans using @Autowired

2.1 Spring Containers

- There are several different Spring containers

- The Spring container gets instructions by reading configuration metadata

- The Configuration metadata can be provided to container by:

  ◇ Annotation-based container configuration

  ◇ Java-based Container configuration

  ◇ XML Configuration

- Which approach to use? It depends on the developer strategy

- This chapter will focus on annotation

  ◇ Annotations are a feature of Java SE 5

  ◇ Since Spring 3.x requires Java SE 5 or higher EVERY Spring 3.x application can use annotations

- Wiring is the process of creating associations between collaborating application objects

Spring Containers

Spring containers are responsible for implementing IoC, instantiation, wiring, configuration and bean lifecycle management.

This chapter will focus on Annotation-based container configuration method to implement IOC.

2.2 Annotation-based Spring Bean Definition

- Spring has a number of annotations that can be used to mark a Java class

as a Spring bean

- These component "stereotype" annotations are @Component, @Service, @Repository, and @Controller

  - ◊ @Component is the generic annotation and the other three are specializations of it that let you identify the different roles Spring components might play in an application

- Besides the name of the annotation there is very little that distinguishes the different annotations

  - ◊ @Repository adds automatic data access Exception translation through Spring data access modules

  - ◊ @Controller is often used in Spring MVC web applications but is not limited to that use

- By default the name of the Spring component defined this way is the name of the class starting with lowercase letter

  - ◊ The annotations can take a String value as the name of the component

```
@Service("orderService")
public class OrderServiceImpl implements OrderService {...
```

Annotation-based Spring Bean Definition

Without the above value in the annotation the name of the Spring component would have been 'orderServiceImpl'.

2.3 Scanning for Annotation Components

- The annotation alone is not enough to define the Spring component

  - ◊ Spring must be instructed for the package to "scan" for annotated components

    - ■ This helps with start-up performance since you can be selective which classes are scanned for annotations

- The easiest way to configure this is to use the <context:component-scan> element in a Spring XML configuration file

  - ◊ This requires that the Spring 'context' XML schema is configured for the

Spring configuration file

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

- Even though there will still be some XML to configure the scanning the XML will be much less since most Spring beans may be defined by annotations

2.4 Defining Component Scope Using Annotations

- One of the important settings in a Spring component definition is the scope

- Defining the scope with annotations is done with the Spring @Scope annotation

   ◊ This takes as a String value the same options that can be used in XML configuration (singleton, prototype, request, session)

   ◊ This is really only required when a scope besides the default of singleton is required

```
@Service("orderService")
@Scope("prototype")
public class OrderServiceImpl implements OrderService {...
```

- If the component definition is overridden with a matching XML definition the @Scope annotation will be ignored since the 'scope' attribute of the XML <bean> definition will be applied

   ◊ This could turn a bean back to singleton scope even if the XML attribute is not present

2.5 JSR-330 @Named Annotation

- JSR-330, or Dependency Injection for Java, defines a number of standard annotations used by dependency injection frameworks

   ◊ Spring recognizes these annotations

- The JSR-330 annotation for declaring a named component is @Named

   ◊ This annotation also takes a String value for the name of the component instead of the default based on the class name

```
@Named("JSR330orderService")
```

```
public class OrderServiceImpl implements OrderService {...
```
- JSR-330 is not yet included in Java SE so you would also need to make sure the JAR with the annotation definitions is on the classpath

2.6 JSR-330 @Scope

- Although JSR-330 does have a @Scope annotation we can't use it the same way as the Spring version

- The JSR-330 @Scope annotation is designed for use in defining custom annotations that would declare a scope for a component

  ◇ The only standard scope defined in JSR-330 is @Singleton which obviously doesn't have much use since this is the default for Spring without any other setting

- If you want to avoid placing the Spring @Scope annotation in the code of the Java component is to define a custom annotation that contains the appropriate scope

  ◇ custom annotation:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Scope("prototype")
public @interface PrototypeScope {
```
  ◇ Used in Java component:

```
@Named("JSR330orderService")
@PrototypeScope
public class OrderServiceImpl implements OrderService {...
```

JSR-330 @Scope

Spring does not have annotations like the example above that would indicate the scope just with the presence of the annotation. The way Spring does it is to have a single @Scope annotation that can take different String values for the actual scope. Even if Spring did have an annotation like that above though using it would still place a Spring annotation in the code of your Java component.

2.7 Annotation-based Dependency Injection

- There are 4 annotations for dependency injection. They are @Autowired,

@Configurable, @Qualifier, and @Resource. There are available since Spring 2.5 except @Configurable which is from 2.0.

- The @Autowired annotation autowires by type, but it can also be used along with the @Qualifier annotation to provide more specific autowiring behavior if there will be multiple beans of the same type in the IoC container.

- The @Configurable annotation is used to mark a class eligible for Spring dependency injection, but it's typically used with classes instantiated outside of the IoC container. This annotation is used along with the context:spring-configured element and will be covered in the AOP chapter since it's AOP related.

- The @Resource annotation autowires by name and is from JSR-250, which is the specification for Commons Annotations.

- Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

2.8 Wiring Bean using @Inject

- Spring 3.0 add support for JSR-330(Dependency Injection for Java) annotation contained in the javax.inject package such as `@Inject, @Qualifier, @Named, and @Provider` if the JSR330 jar is present on the classpath. Use of these annotations also requires that certain `BeanPostProcessors` be registered within the Spring container.

- JSR 330's @Inject annotation can be used in place of Spring's @Autowired.

- @Inject does not have a required property unlike Spring's @Autowired annotation which has a required property to indicate if the value being injected is optional.

- This behavior is enabled automatically if you have the JSR 330 JAR on the classpath.

- The AutowiredAnnotationBeanPostProcessor must be registered with either the BeanFactory or ApplicationContext to have annotation-based autowiring on beans work.

- This bean post processor is implicitly registered by <context:component-scan/> and <context:annotation-config/>.

Wiring Beans using @Inject

In this chapter we will focus annotation based wiring only for dependency injection, XML based configuration we will discuss in another chapter.

When @Autowired is unable to find at least one match, a BeanCreationException will be thrown. By default it required that it autowires at least one value, but this can be changed by setting @Autowired(required=false). If more than one match is found when matching by type and it isn't an array or Collection class, a BeanCreationException will be thrown. If there is one bean that should be selected out of many of the same type, the bean element's primary attribute can be set to true. This will let autowiring by type pick a unique bean from a list of beans that are all the same type. There must be only one bean marked as the primary bean for a type or an exception will still be thrown.

BeanPostProcessors be registered within the Spring container as follows:

```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"

    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">

  <context:annotation-config/>

<context:component-scan base-package="com.simple.beans"/>

</beans>
```

2.9 @Autowired - Constructor

- Example:

```java
public class OrderManager {

    private OrderDao orderDao;

    @Autowired
```

```
        public OrderManager (OrderDao orderDao){
                this.orderDao = orderDao;
        }


}
```

@Autowired - Constructor

One or more parameters can be autowired for a constructor or method. Only one constructor can be marked as required, although more than one can be set as autowired. If there is more than one constructor marked as autowired, Spring will use the constructor that can have the most arguments matched based on the beans in the IoC container and the autowiring rules.

The two different parameters will be autowired by type. As previously mentioned, there must only be one bean defined in the container of each type being autowired or there will be an error.

2.10 @Autowired – Field

- Based on the type of the field, a OrderDao bean is set on the field. There must be only one matching bean.

- Example:

```
public class OrderManager {

        @Autowired
        private OrderDao orderDao = null;


}
```

2.11 @Autowired - method

- The autowired method has a OrderDao bean set on the method. There must be only one matching bean.

- Example:

```
public class OrderManager {

        private OrderDao orderDao;
```

```
        @Autowired
        public void setOrderDao(OrderDao dao) {
                this.orderDao = dao;
        }
}
```

2.12 @Autowired – Collection

- All beans matching the List's generic type will be set on the field and method.

- Example:

```
public class OrderManager {

        private List<String>list;

        @Autowired
        public void setFileList(List <String> list) {
                this.fileList = list;
        }

}
```

2.13 @Autowired – Maps

- All beans matching the Map's generic type value will be set on the field and method, and the key will be the bean's name.

- Example:

```
public class MovieService {

  private Map<String, MovieCatalog> movieCatalogs;

  @Autowired
  public void setMovieCatalogs(Map<String, MovieCatalog>
movieCatalogs) {
      this.movieCatalogs = movieCatalogs;
```

```
  }

  // ...
}
```

2.14 @Autowired & @Qualifier with Constructors, Fields, and Methods

- The @Qualifier annotation is used exclusively along with the @Autowired. The simplest usage of @Qualifier is to pass in a bean name.

- This changes the autowiring from by type to by name wherever @Qualifier specifies a name.

- Example 1 (@Autowired & @Qualifier Constructor):

```
@Autowired
public OrderManager(@Qualifier("hrDao")HrDao dao1,
        @Qualifier("customerService")CustomerService
service) {
...
}
```

@Autowired & @Qualifier with Constructors, Fields, and Methods

This is the equivalent of using the @Resource annotation when used on a field or method, but using @Autowired with @Qualifier is much more flexible since it can also be used with constructors and multiple parameter methods. If there isn't a match and @Autowired is marked as required, then a BeanCreationException will be thrown. When @Qualifier is used on constructors or methods with multiple parameters, any parameters without the annotation will still be autowired by type.

2.15 @Autowired & @Qualifier with Constructors, Fields, and Methods

- Example 2 (@Autowired & @Qualifier Field):

```
@Autowired
@Qualifier("customerDao")
protected CustomerDao cdao = null;

@Autowired
```

```
@Qualifier("hrDao")
protected HrDao hdao = null;
```
- - Example 3 (@Autowired & @Qualifier Method):

```
protected CustomerDao cDao = null;

@Autowired
public void
setCustomerDao(@Qualifier("customerDao")CustomerDao cDao) {
    this.cDao = cDao;
}
```

2.16 @Autowired & Custom Qualifiers

- - Customized qualifiers can be used with @Autowired

- - Example (**Simple Custom Qualifier**):

```
Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Dao {

    String value();

}
```

- - The @Qualifier indicates, that it is a custom qualifier annotated class
- - The @Target annotation indicates, that it is an annotation can be used on fields and parameters (constructor or method).

@Autowired & Custom Qualifiers

Custom Qualifiers with Autowiring give more detailed and meaningful annotation than just referencing a bean by type or name. The simple way to do this by creating an annotation.

For example, @Billing and @Shipping annotations are able be made just to autowire address beans to the correct address field. Obviously more meta data than this would really be needed since you need a unique match and would have many addresses in a real system, but it gives the general idea on how this might be used.

2.17 @Autowired & Simple Custom Qualifier Field

- @Dao annotation and its value will be matched and @Autowired by the field.

- Example:

```
public class OrderManager {
        @Autowired
        @Dao("customer")
        protected CustomerDao cDao = null;

        @Autowired
        @Dao("hr")
        protected HrDao hdao = null;
    …
}
```

@Autowired & Simple Custom Qualifier Field

HrDao and CustomerDao both use the qualifier element to identify the annotation type of the custom qualifier and also its value. The qualifier element's type attribute can be set with either the annotation's class name or the full package and class name. If only the class name is set and the name is unique, Spring will find it.

```
<bean id="customerDao"

    class="com.simple.dao.CustomerDao">

  <qualifier type="Dao" value="customer" />

</bean>


<bean id="hrDao"

    class="com.simple.dao.HrDao">

  <qualifier type="com.simple.dao.Dao"

            value="hr" />

</bean>
```

2.18 @Autowired & Simple Custom Qualifier Method

- This example show that custom qualifier annotation is used directly on the method parameter(s).

- Example:

```
public class OrderManager {
       protected CustomerDao cDao = null;

       @Autowired
       public void setCustomerDao
(@Dao("customer")CustomerDao  cDao) {
              this.cDao  = cDao ;
       }
       ...
}
```

2.19 @Autowired & CustomAutowireConfigurer

- If you don't want to use @Qualifier annotation for some reason or using earlier than Java 5 which do not support @Qualifier. CustomAutowireConfigurer class can be used to registered as XML bean definition.

- Example:

```
@Autowired
@CustomDatasource("customer")
protected CustomerDao customerDao = null;

@Autowired
@CustomDatasource("hr")
protected HrDao hDao = null;
```

2.20 @Autowired & CustomAutowireConfigurer

- Register CustomDao annotation class with CustomAutowireConfigurer, then it can be used as a custom qualifier during the autowiring process

```
<bean id="customAutowireConfigurer"
```

```
class="org.springframework.beans.factory.annotation.CustomA
utowireConfigurer">
    <property name="customQualifierTypes">
        <set>
            <value>com.simple.dao.CustomDao</value>
        </set>
    </property>
</bean>

<bean id="customerDao"
        class="com.simple.dao.CustomerDao">
    <qualifier type="CustomDao" value="customer" />
</bean>

<bean id="hDao"
        class="com.simple.dao.HrDao">
    <meta key="value" value="hr" />
</bean>
```

2.21 Dependency Injection Validation

- The @Required annotation applies to method that must have a value injected into it.

- This annotation is very useful for marking a critical resource that should be dependency injected example, Dao, Datasource.

- RequiredAnnotationBeanPostProcessor must be configured in the IoC container.

- The <context:annotation-config />element automatically registers this bean post processor.

Dependency Injection Validation

Example:

public class OrderManager {

        protected CustomerDao cDao = null;

@Required

public void setCustomerDao (CustomerDao cDao) {

    this.cDao = cDao ;

}

...

}

The container throws an exception if the affected bean property has not been populated; this allows for eager and explicit failure, avoiding `NullPointerExceptions` or the like later on.

2.22 @Resource

- @Resource annotation is also supported by Spring using JSR-250

- @Resource takes a name attribute, and by default Spring interprets that value as the bean name to be injected. (inject by name)

- If no name is specified explicitly, the default name is derived from the field name or setter method(bean property name)

- Example 1:

```
public class OrderManager {
      protected CustomerDao cDao = null;


      @Resource(name="myCustomerDao")
      public void setCustomerDao (CustomerDao  cDao) {
            this.cDao  = cDao ;
      }
      ...
}
```

2.23 @Resource

- Example 2:

```
public class OrderManager {
      protected CustomerDao cDao = null;
```

```
        @Resource
        public void setCustomerDao (CustomerDao   cDao) {
                this.cDao   = cDao ;
        }
        ...
}
```

- Above example takes the bean name as **cDao**

2.24 @PostConstruct and @PreDestroy

- This annotation is supported since Spring 2.5, to provide initialization callbacks and destruction callbacks

- CommonAnnotationBeanPostProcessor is is used to recognize those callbacks

- Example:

```
public class CachingOrderLister {

    @PostConstruct
    public void populateOrderCache() {
        // populates the order cache upon initialization...
    }

    @PreDestroy
    public void clearOrderCache() {
        // clears the order cache upon destruction...
    }
}
```

2.25 Summary

- There are many different annotations that can configure Spring beans

- Many of these annotations have similar purpose and the choice comes down to the function that is required and if standard annotations are preferred for a project

Chapter 3 - Spring Framework Configuration

Objectives
Key objectives of this chapter

Key objectives of this chapter

- Cover ways to configure Spring components in Spring Boot:
  - ◇ Java @Configuration classes
  - ◇ Spring Expression Language
  - ◇ External Resource Bundles
  - ◇ Spring Property Editors and Converters

3.1 Java @Configuration Classes

- Often using source code annotations and XML configuration are not sufficient to configure Spring applications

  - ◇ This is especially true when annotations can't be added to the classes being used and configuring only with XML is difficult

  - ◇ With Spring Boot, we don't typically use XML configuration ('spring-beans.xml')

- We can write Java code to initialize components

  - ◇ This can also be true if there is an existing code infrastructure for the "Factory" pattern

- To link this Java initialization code into the rest of Spring configuration the Spring **@Configuration** and **@Bean** annotation can be added to Java classes that provide this initialization logic

  - ◇ This initialization code can be used where needed and hooked into the rest of the Spring application which can be configured with the other mechanisms

- Spring components will also look for classes that implement a particular interface, and call initializer methods on those classes

  - ◇ e.g. WebApplicationInitializer for Spring-MVC apps

3.2 Defining @Configuration Classes

- Any Java class with the **@Configuration** annotation indicates to the Spring container that the class can be used as a source of bean definitions

- Methods of a **@Configuration** class are annotated with the **@Bean** annotation to indicate that the methods returns a Spring bean

  ◇ The return type of the method is critical as this will be the type of bean that Spring will use to compare against injection into other components

- One way to understand how the @Bean method works is to compare how the equivalent XML configuration would be done

```
@Configuration
public class AppConfig {
        @Bean
        public MyService myService() {
                return new MyServiceImpl();
        }
}
```

  ◇ Would be similar to this XML configuration:

```
<beans>
        <bean id="myService"
class="com.acme.services.MyServiceImpl"/>
</beans>
```

Defining @Configuration Classes

The simple example above does not really show much of the benefit of using @Configuration classes. The benefit is much greater when other Java code performs initialization of the object before the object is returned from the @Bean method.

3.3 Defining @Configuration Classes

- In order to use @Configuration classes you must have the 'cglib' code generation library as part of the project

  ◇ This library dynamically adds the feature of turning @Configuration classes into Spring bean definitions at startup time

- Since 'cglib' will sub-class your @Configuration classes this places a few

minor restrictions on the class definition used for @Configuration classes:

 ◇ @Configuration classes may not be *final*

 ◇ @Configuration classes must have a no-argument constructor

3.4 Loading @Configuration Classes

- A @Configuration class does not contribute to the Spring configuration unless it is visible to Spring

- In a plain Spring application, you can define your @Configuration class as a bean in the xml config

  ◇ Could also configure component scan in the xml config

  ◇ Put @ComponentScan on a @Configuration class would do the same thing

- Spring Boot is typically configured to scan for components in the classpath

  ◇ @SpringBootApplication on the main class is equivalent to @Configuration, @EnableAutoConfiguration and @ComponentScan

Loading @Configuration Classes

There is also a 'scan' method that can be used from Java code to replicate the behavior of the <context:component-scan> element from XML configuration.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.scan("com.acme.config");
```

3.5 Loading @Configuration Classes

- [Plain Spring] You can also add support to web.xml for the WebApplicationContext version of the AnnotationConfig class

```
<context-param> (or MVC Dispatcher Servlet <init-param>)
      <param-name>contextClass</param-name>
      <param-value>
org.springframework.web.
context.support.AnnotationConfigWebApplicationContext
      </param-value>
</context-param>
```

```
<context-param> (or MVC Dispatcher Servlet <init-param>)
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.config.AppConfig</param-value>
</context-param>
<!-- Usual ContextLoaderListener or
Spring MVC Dispatcher Servlet -->
```

3.6 Modularizing @Configuration Classes

- Although it would be possible to provide all bean definitions in a single @Configuration class, multiple classes are often used to provide the modularity of configuration that resembles the rest of the application

- You could use the @Import annotation to import one configuration class into another

```
@Configuration
public class ConfigA {
        // @Bean methods
}
@Configuration
@Import(ConfigA.class)
public class ConfigB { ... }
```
  - ◊ One weakness of this approach though is one @Configuration class has to refer directly to the other @Configuration class

- You can also use @ImportResource to import XML from Java @Configuration classes

```
@Configuration
@ImportResource("classpath:/com/acme/other-config.xml")
public class AppConfig { .. }
```

3.7 Modularizing @Configuration Classes

- Any class with a @Configuration annotation also has a @Component annotation added by Spring

  - ◊ This means you could even inject the configuration classes themselves into each other

  - ◊ These injected configuration classes could even implement interfaces

```
@Configuration
public class ServiceConfig {
  @Autowired private RepositoryConfig repositoryConfig;
  public @Bean TransferService transferService() {
    return new TransferServiceImpl(
      repositoryConfig.accountRepository());
  }
}
@Configuration
public interface RepositoryConfig {
  @Bean AccountRepository accountRepository();
}
@Configuration
public class DefaultRepositoryConfig implements
RepositoryConfig {
  @Bean public AccountRepository accountRepository() { .. }
}
```

Modularizing @Configuration Classes

One benefit of the approach above is that you can directly call the methods of the injected configuration class to return a bean of the desired type. Since you are writing Java code anyway this may be more intuitive than having Spring use the @Autowired annotation to inject a component as a field that is then used elsewhere in component initialization, as shown below.

```
@Configuration
public class ServiceConfig {
  private @Autowired AccountRepository accountRepository;

  public @Bean TransferService transferService() {
      return new TransferServiceImpl(accountRepository);
  }
}

@Configuration
public class RepositoryConfig {
  public @Bean AccountRepository accountRepository() { .. }
}
```

3.8 Qualifying @Bean Methods

- By default the name of a @Bean method is the name of the bean defined

```
@Bean public MyService myService() { // 'myService' bean
```
- You can also supply one or more bean names with the @Bean annotation

to override the default bean name

```
@Bean(name={"myService","myOtherBeanName"})
```
- You can change the scope of the bean definition by also adding a @Scope annotation along with the @Bean annotation to the method

  ◇ If no @Scope is present the Spring "singleton" default scope is used

```
@Bean
@Scope("prototype")
public MyService myService() { .. }
```
- You can also use @Primary to indicate this would be the primary bean to inject for dependency injection

```
@Bean  @Primary
public MyService myService() { .. }
```
- Finally, you can use any custom qualifier annotation or the Spring @Qualifier annotation on a @Bean method to indicate which qualifiers would match the bean definition

```
@Bean  @StrategyType(strategy=GenerationStrategy.RANDOM)
public IntGenerator randomGenerator() { .. }
```

Qualifying @Bean Methods

If a bean has more than one bean name they are typically referred to as bean "aliases".

3.9 Trouble with Prototype Scope

- If any Spring bean defined at "prototype" scope is used in a @Configuration class care must be taken to ensure proper behavior

- If you use @Autowired on a field of a @Configuration class that field will only be initialized once and using it more than once will use the same object

  ◇ This is not what is desired with prototype scope

```
@Configuration
public class AppConfig {
      @Autowired
      private SomePrototypeBean prototypeBean;
```

```
@Bean public OtherBean1 otherBean1 () {
        // this uses the injected instance
        return new OtherBean1(prototypeBean);
}

@Bean public OtherBean2 otherBean2 () {
        // this uses the SAME instance
        return new OtherBean2(prototypeBean);
}
```

3.10 Trouble with Prototype Scope

- If using a prototype bean that is declared in a @Configuration class the best way is to inject the @Configuration class itself and call the @Bean method directly

  ◇ This continues to work also if the bean is a singleton as Spring will return the proper bean from the @Bean method depending on the behavior indicated by the scope

```
@Configuration
public class AppConfig {
  @Autowired private OtherConfig otherConfig;

  @Bean public OtherBean1 otherBean1 () {
      return new OtherBean1(
          otherConfig.producePrototypeBean());
```

- If the prototype bean you need to use has been defined in XML it is a little more tricky as you need to use the @Scope("prototype") annotation on the @Configuration class itself so that the @Configuration class is created every time and has the @Autowired injection performed again

```
@Configuration
@Scope("prototype")
public class AppConfig {
        @Autowired
        private SomePrototypeXMLBean prototypeXMLBean;
```

3.11 Configuration with Spring Expression Language

- One of the big changes in Spring 3.0 was the introduction of the 'Spring Expression Language' (SpEL)

- One of the more useful aspects of the Spring EL is using it in XML configuration files to make them more dynamic

  ◇ You can set the value of one Spring component based on the property value of another

```
<bean id=".." class=".." >
      <property name="serverVendor"
      value="#{environmentBean.serverType}" />
</bean>
```

  ◇ You can also use the predefined variable 'systemProperties' to pull in values of environment variables using Spring EL

```
<bean id=".." class="..">
      <property name="defaultLocale"
      value="#{ systemProperties['user.region'] }"/>
</bean>
```

  ◇ You can also use the @Value annotation within Java code for a field

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

Configuration with Spring Expression Language

This allowed Spring to define and use it's own expression language instead of relying on an external implementation.

The Spring documentation focuses a lot on how to write Java code to use the Spring expression language to evaluate expressions. This will not be commonly done in Java code by end users

In the example above the 'environmentBean' could be a bean that retrieves various configuration parameters from some mechanism and then exposes them through bean properties. You could have several beans configured with the properties of this bean by using the Spring EL.

Using the @Value annotation makes the initialization of a component property more externally configurable with a dynamic value even though it is placed within the Java code.

The @Value annotation could be used on a component field or a 'setter' method.

3.12 Resolving Text Messages

- Sometimes configuration of an application might depend on the language or locale of the environment the application is run in

  ◇ Java has a great Internationalization framework for message bundles but how to use this within a Spring application?

- The Spring ApplicationContext can resolve messages based on locale

- To use this you could first declare a 'ResourceBundleMessageSource' bean on a configuration bean

```
@Configuration
public class SpringContext {
        @Bean
        @Qualifier("myMessages")
        public MessageSource getMessageSource() {
                ResourceBundleMessageSource rbms=new
                        ResourceBundleMessageSource();
                rbms.setBasename("Messages");
                return rbms;
        }
}
```

3.13 Resolving Text Messages

- You could then inject a 'MessageSource' into a @Configuration class and use the 'getMessage' method

```
@Configuration
public class ProductCodeConfiguration {
        @Autowired
        @Qualifier("myMessages")
        private MessageSource messageSource;

        @Bean
        public ProductCodeGenerator prodCodeGenerator() {
                String prefix = messageSource.
                getMessage("defaultPrefix", null, null);
                // use this to configure a bean
```

```
.. }
```

3.14 Spring Property Conversion

- Spring has two main ways to convert property values from XML configuration into Java types

  ◇ JavaBean PropertyEditors

  ◇ Spring Converter implementations

- For configuration both mechanisms are primarily used to convert Strings read from XML configuration files into other Java types for Spring bean properties

- For the following examples we will show converting a String into a custom Java type like 'PersonalSSN'

```
convert "XXX-XX-XXXX" into:
public class PersonalSSN {
      private int area;
      private int group;
      private int serial;
... }
```

Spring Property Conversion

It has been observed that both property conversion mechanisms in Spring can have a problem with behavior when you would expect the conversion to happen more than once. One example of this is if a Spring bean is defined at prototype scope and a property of the bean is converted from a String. It appears that the Spring property conversion is not triggered for any but the creation of the first prototype bean. It has been observed that if the same String is encountered that Spring has already converted it will simply use the same instance from the first conversion instead of creating a new instance. This would cause problems with prototype beans since the properties of two DIFFERENT instances of the prototype bean could be sharing an object set as the property of the bean and therefore sharing state BETWEEN THE BEANS.

3.15 Spring Converter Interface

- The easiest way to implement conversion is with the new Spring 3 type conversion system

- To provide a custom Spring converter you implement the *Converter<S, T>*

interface

- ◇ This defines a way to convert from the 'Source' class 'S' to the 'Target' class 'T'

    - ■ The use of Java generics makes the 'Converter' interface very self-documenting

- ◇ This only requires the implementation of the 'convert' method

```
public class PersonalSSNConverter
      implements Converter<String, PersonalSSN> {
      public PersonalSSN convert(String number) {
            ...
      }
```

- ◇ A runtime 'IllegalArgumentException' should be thrown to indicate conversion errors

3.16 Using Custom Converters

- ■ To register a custom converter you would load it in the 'converters' property of the Spring 'ConversionServiceFactoryBean'

```
@Bean
public FactoryBean<ConversionService>
conversionService() {
      ConversionServiceFactoryBean factory=
            new ConversionServiceFactoryBean();
      Set<Converter<?,?>> converters=
            new HashSet<Converter<?,?>>();
      converters.add(new PersonalSSNConverter());

      factory.setConverters(converters);
      return factory;
}
```

- ■ Besides Spring now knowing how to use your custom converter when needed you can inject the conversion service and use it directly

```
public class MyService {
      @Autowired
      private ConversionService converter;
```

```
    public void someOtherMethod() {
            converter.convert(objectToConvert,
                    TargetType.class);
    ... }
```

3.17 Spring PropertyEditors

- Another way to implement conversion is to provide a PropertyEditor which extends the JavaBean PropertyEditorSupport class

- You would then override the 'setAsText' method to indicate how to convert from a String to the custom property type

  ◊ The key of this method is to call the 'setValue' method at the end from the base class passing in the object that was constructed by converting from the String

```
public class SSNEditor extends PropertyEditorSupport {
    public void setAsText(String toConvert) {
            int area = // parse area
            int group = // parse group
            int serial = // parse serial
            PersonalSSN ssn =
                    new PersonalSSN(area, group, serial);
            setValue(ssn);
    }
}
```

Spring PropertyEditors

Although the PropertyEditorSupport class also has a 'getAsText' method that you could override, the most common way a property editor is used is converting from the String in an XML configuration file to the custom property type of a bean class. This means the 'setAsText' method is used much more commonly.

3.18 Registering Custom PropertyEditors

- In order for a custom PropertyEditor to be used it must be registered to Spring

◇ The best way to do this is to have another class that implements the 'PropertyEditorRegistrar' interface and will register the PropertyEditor with Java code

```
public class SSNEditorRegistrar implements
PropertyEditorRegistrar {
        public void registerCustomEditors(
                PropertyEditorRegistry registry) {
                registry.registerCustomEditor(
                        PersonalSSN.class, new SSNEditor());
        }
}
```

◇ You would then configure the Spring 'CustomEditorConfigurer' with this PropertyEditorRegistrar

■ Note: This is a case where it actually is a little more direct to use xml. Import the file with @ImportResource

```
<bean class="org.springframework.
        beans.factory.config.CustomEditorConfigurer">
        <property name="propertyEditorRegistrars">
                <list><bean
class="example.SSNEditorRegistrar"/>
                </list>
        </property>
</bean>
```

Registering Custom PropertyEditors

You can register more than one PropertyEditor in a PropertyEditorRegistrar.

Although it would also be possible to directly register the custom property editor with the following property on this bean, using the Spring Property Editor Registrar method is better as it is more thread safe.

```
<bean class="org.springframework.
        beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
                <map>
                        <entry key="com.webage.bean.ComboIntType"
                        value="com.webage.editors.ComboIntPropertyEditor" />
                </map>
        </property>
</bean>
```

3.19 Summary

- There are many alternate mechanisms that can be used in Spring configuration

- Spring Boot in particular favors Java-based configuration

- The use of @Configuration classes can help when it is easier to initialize Spring components with direct Java code

- Spring has a few mechanisms for property conversion

Chapter 4 - Introduction to Spring Boot

Objectives
Key objectives of this chapter

- Overview of Spring Boot
- Using Spring Boot for building microservices
- Examples of using Spring Boot

4.1 What is Spring Boot?

- Spring Boot (*http://projects.spring.io/spring-boot/*) is a project within the Spring IO Platform (*https://spring.io/platform*)
- Developed in response to Spring Platform Request SPR-9888 "Improved support for 'containerless' web application architectures"
- Inspired by the DropWizard Java framework *(http://www.dropwizard.io/)*
- The main focus of Spring Boot is on facilitating a fast-path creation of stand-alone web applications packaged as executable JAR files with minimum configuration
- An excellent choice for creating microservices
- Released for general availability in April 2014
  - (*https://spring.io/blog/2014/04/01/spring-boot-1-0-ga-released*)

Notes:

A JAR (Java ARchive) is a file format and deployment unit used to package a set of Java class files and associated metadata and resources.

4.2 Spring Boot Main Features

- Spring Boot offers web developers the following features:
  - Ability to create WAR-less stand-alone web applications that you can run from command line
  - Embedded web container that is bootstrapped from the `public`

`static void main` method of your web application module:

- Tomcat servlet container is default; you have options to plug in Jetty or Undertow containers instead

◇ No, or minimum, configuration

- Spring Boot relies on Spring MVC annotations (more on annotations later ...) for configuration

◇ Build-in production-ready features for run-time metrics collection, health checks, and externalized configuration

Notes:

A WAR (Web application ARchive) file is a format for and unit of packaging and distribution of Java-based web applications that are deployed in the web container of a Java-enabled web server.

For steps how to switch from Tomcat to Jetty or Undertow, visit *http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html*

For an overview of the Spring framework, visit *http://docs.spring.io/spring/docs/current/spring-framework-reference/html/overview.html*

The following diagram, borrowed from the above Spring framework link, lays out the components of a full-fledged Spring web application:
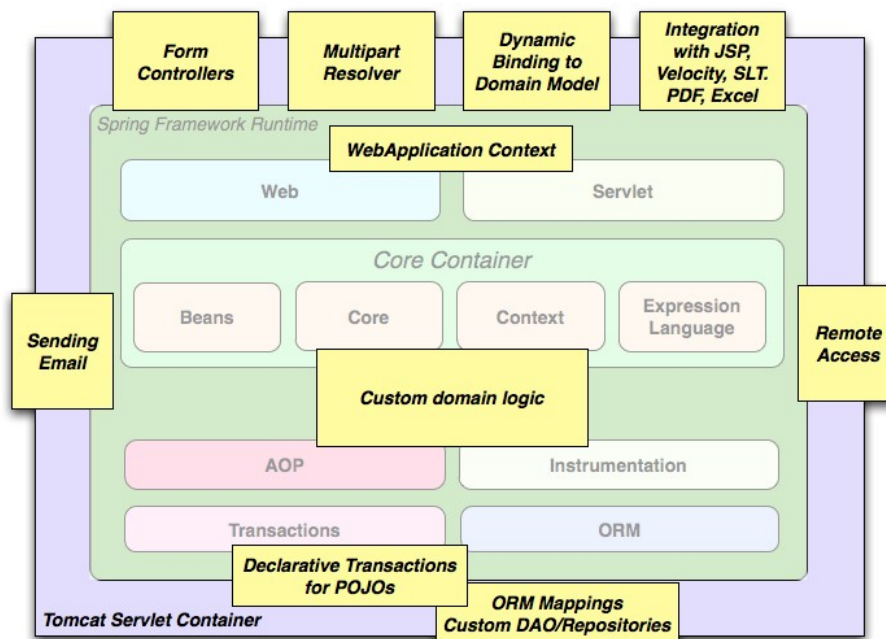
4.3 Spring Boot 2

- Spring Boot is at version 2

  - Version 2.2.0 will be the new version

- Spring Boot 2 requires Java 8 minimum

- Breaking changes in security configuration

  - No more default username and password

  - Everything is now secured by default, including static resources and endpoints

- Support for reactive programming

- Actuators endpoints are now independent of Spring MVC

- Custom endpoints can be created

- Predefined endpoints are no longer configurable

- Spring Boot DAO API are now in line with JPA APIs

  - .findOne is now called .findById

  - .findById returns an Optional<T> instead of T

4.4 Spring Boot on the PaaS

- PaaS (Platform as a Service) clouds offer robust, scalable, and cost-efficient run-time environments that can be used with minimum operational involvement

- There are two popular choices for deploying Spring Boot applications:

  - Cloud Foundry *(https://www.cloudfoundry.org/)*

  - Heroku (*https://www.heroku.com/*)

  - **Note**: Google App Engine PaaS has not advanced beyond the Servlet 2.5 API, so you won't be able to deploy a Spring Application there without some modifications

    - Spring Boot uses Servlet API version 3.1

- Spring Boot is being financed by Pivotal Software Inc. which also, in

partnership with VMware, sponsor Cloud Foundry

- ◇ While Cloud Foundry is OSS available for free, Pivotal offers a commercial version of it called Pivotal Cloud Foundry

4.5 Understanding Java Annotations

- Annotations in Java are syntactic metadata that is baked right into Java source code; you can annotate Java classes, methods, variables and parameters

- Basically, an annotation is a kind of label that is processed during a compilation stage by annotation processors when the code or configuration associated with the annotation is injected in the resulting Java class file, or some additional operations associated with the annotation are performed

- An annotation name is prefixed with an '@' character

- **Note:** Should you require to change annotation-based configuration in your application, you would need to do it at source level and then re-build your application from scratch

4.6 Spring MVC Annotations

- Spring Boot leverages Spring MVC (Model/View/Controller) annotations instead of XML-based configuration

- ◇ Additional REST annotations, like @*RestController* have been added with Spring 4.0

- The main Spring MVC annotations are:

- ◇ **@Controller / @RestController** – annotate a Java class as an HTTP end-point

- ◇ **@RequestMapping** – annotate a method to configure with URL path / HTTP verb the method responds to

- ◇ **@RequestParam** - named request parameter extracted from client HTTP request

4.7 Example of Spring MVC-based RESTful Web Service

- The following code snippet shows some of the more important artifacts of Spring MVC annotations (with REST-related Spring 4.0 extensions) that you can apply to your Java class

- **Note:** Additional steps to provision and configure a web container to run this module on are required

```java
// … required imports are omitted
@RestController
public class EchoController {
    @RequestMapping("/echoservice", method=GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
}
```

- The above annotated Java class, when deployed as a Spring MVC module, will echo back any *echo* message send with an HTTP GET request to this URL:

```
http(s)://<Deployment-specific>/echoservice?id=hey
```

4.8 Spring Booting Your RESTful Web Service

- Spring Boot allows you to build production-grade web application without the hassle of provisioning, setting up, and configuring the web container

- A Spring Boot application is a regular executable JAR file that includes the required infrastructure components and your compiled class that must have the `public static void main` method which is called by the Java VM on application submission

  ◇ The `public static void main` method references the `SpringApplication.run` method that loads and activates Spring annotation processors, provisions the default Tomcat servlet container and deploys the Spring Boot-annotated modules on it

4.9 Spring Boot Skeletal Application Example

- The following code is a complete Spring Boot application based on the Spring MVC REST controller from a couple of slides back

```
// … required imports are omitted
@SpringBootApplication
@RestController
public class EchoController {
    @RequestMapping("/echoservice", method=GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(EchoController.class, args);
    }
}
```

- You run Spring Boot applications from command line as a regular executable JAR file:

<div align="center">

java -jar Your_Spring_Boot_App.jar

</div>

- The default port the Spring Boot embedded web container starts listening to is 8080

  ◇ To change the default port, you need to pass a **server.port** System property or specify it in the Spring's **application.properties** file

Notes:

The **@SpringBootApplication** is functionally equivalent to three annotations applied sequentially:

@Configuration, @EnableAutoConfiguration, and **@ComponentScan**

4.10 Converting a Spring Boot Application to a WAR File

- In some scenarios, users want to have a Sprint Boot runnable JAR file converted into a WAR file

- For those situations, Spring Boot provides two plug-ins:

  ◇ **spring-boot-gradle-plugin** for the Gradle build system (*https://gradle.org/*)

  ◇ **spring-boot-maven-plugin** for the Maven build system (*https://maven.apache.org/*)

- For more details on Spring Boot JAR to WAR conversion, visit *https://spring.io/guides/gs/convert-jar-to-war/*

4.11 Externalized Configuration

- Lets you deploy the same Spring Boot artifact (jar file) in different environments - config is drawn from the environment

- Properties are pulled from (note - this is an incomplete list)

  ◊ OS Environment variable SPRING_APPLICATION_JSON

  ◊ Java System properties

  ◊ OS Environment variables

  ◊ Application property files - application.properties or application.yml

      ■ in a '/config' folder in current directory

      ■ in the current directory

      ■ in a classpath '/config' package

      ■ in the classpath root

      ■ In a folder pointed to by 'spring.config.location' on the command line.

4.12 Starters

- Spring Boot auto-configures based on what it finds in the classpath

- So, the set of modules is determined by what's in the 'pom.xml'

- The project provides a number of 'starter' artifacts that pull in the correct dependencies for a given technology

- Some examples:

  ◊ spring-boot-starter-web

  ◊ spring-boot-starter-jdbc

  ◊ spring-boot-starter-amqp

- You just need to include these artifacts in the 'pom.xml' to configure those technologies.

4.13 The 'pom.xml' File

- Assuming you're building with Apache Maven, the 'pom.xml' file describes all the artifacts and build tools that go into producing a delivered artifact.

- Generally, use the 'starter parent':

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2.RELEASE</version>
</parent>
```

- It defines all the dependency management and core dependencies for a Spring Boot application

4.14 Spring Boot Maven Plugin

- The Maven plugin can package the project as an executable jar file, that includes all the dependencies

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

4.15 HOWTO: Create a Spring Boot Application

- Create a new Maven project in the IDE of your choice
- In 'pom.xml',

- ◇ add the Spring Boot parent project (see above)

- ◇ Add dependencies to the 'starter' projects that describe the features you want (e.g. Spring MVC, JDBC, etc)

- ◇ Add the 'spring-boot-maven-plugin' to the build plugins

- ■ Add a default 'application.properties' or 'application.yml' file in 'src/main/resources'

  - ◇ List of properties is at:

    - ■ http://docs.spring.io/spring-boot/docs/1.4.3.RELEASE/reference/htmlsingle/#common-application-properties

  - ■ At least include 'spring.application.name'

4.16 HOWTO: Create a Spring Boot Application

- ■ Create a main class in src/main/java/<package>

- ■ e.g. in 'com.mycom.app',

```
@SpringBootApplication
public class MyApp {

        public static void main(String[] args) {
                SpringApplication.run(MyApp.class, args);
        }

}
```

- ■ Build with 'mvn install'

- ■ From the IDE, run the main class

- ■ From command line, run the jar file

4.17 Summary

- ■ Spring Boot eliminates many of the headaches related to provisioning, setting up and configuring a web server by offering a framework for running WAR-less web applications using embedded web containers

- Spring Boot favors annotation-based configuration over XML-based one

  ◇ Any change to such a configuration (e.g. a change to the path a REST-enabled method responds to), would require changes at source level and recompilation of the project

- Spring Boot leverages much of the work done in Spring MVC

Chapter 5 - Spring MVC

Objectives

Key objectives of this chapter:
 	
   ■ Overview of Spring MVC
 	
   ■ Mapping Web Requests
 	
   ■ Controller Handler Methods
 	
   ■ View Resolution
 	
   ■ Form Tag Library

5.1 Spring MVC

- Spring provides a MVC (Model/View/Controller) web application framework

- Addresses state management, workflow, validation, etc

- Spring MVC framework is modular, allowing various components to be changed easily

- The "Controllers" and "Handler Mappings" of Spring MVC underwent major changes in Spring 3 and later

    ◇ Now annotations are used heavily for these components

    ◇ The implementations of the Spring MVC 'Controller' interface that used to be the parent class of application controllers are now deprecated in favor of annotation controllers

- Spring 3 also introduced an 'mvc' namespace for configuration files to simplify Spring MVC configuration

- Spring 4 added some different view technologies

Spring MVC

Spring provides an MVC web application framework sometimes referred to as "Spring MVC" and sometimes as the "Spring web framework". It is similar to other leading web application frameworks such as Struts. However, Spring MVC is more modular and less intrusive than other web frameworks.

5.2 Spring Web Modules

Web
(MVC / Remoting)

- Web
- Servlet
- Portlet
- Struts
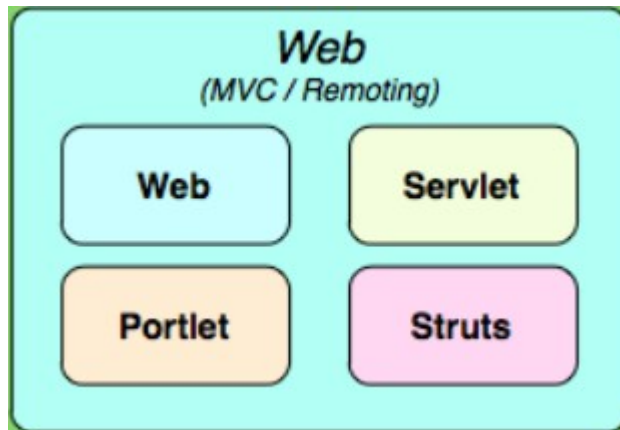
- ■ The Spring 'Web' module has core web support classes
  - ◊ It is required by other modules
- ■ The 'Servlet' module has Spring MVC and other Servlet support classes
- ■ There are also 'Portlet' and 'Struts' modules for integrating those types of web applications with Spring
- ■ Spring 2.5 had broken the Spring MVC classes into a separate distribution
  - ◊ It was the first step to the modular packaging of Spring 3

5.3 Spring MVC Components

1. Client request

Dispatcher Servlet

2. Lookup controller

3. Invoke controller and return ModelAndView

4. Resolve view

5. Pass model to view and execute

Handler Mapping

Controller

View Resolver

View
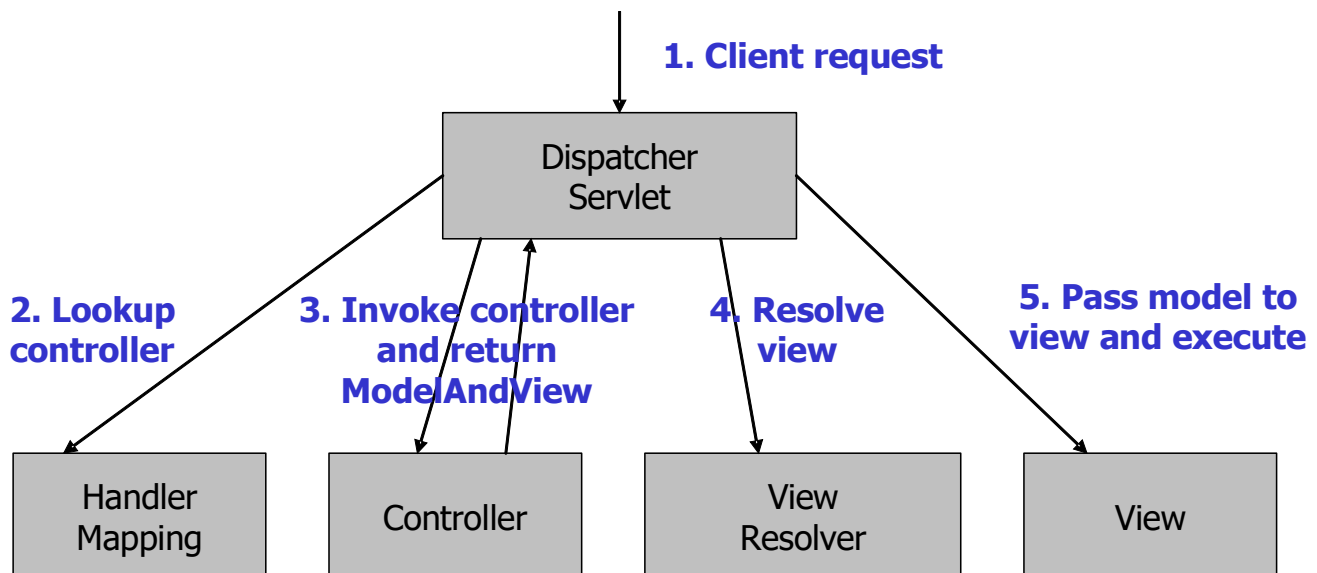
Spring MVC Components

The diagram on the slide depicts the high-level workflow for handling a web request. It illustrates the major Spring MVC components involved in handling a request.

It starts with a client request, typically from a web browser. (1) The DispatcherServlet is first to receive this request. (2) The dispatcher consults one or more HandlerMappings to determine which Controller should handle the request. (3) The dispatcher invokes the Controller to handle the request and the controller returns a ModelAndView object. A ModelAndView contains model information for the next view to display, as well as the logical name of the next view to display. (4) The dispatcher then uses a ViewResolver to map the logical view name to a View object. (5) Finally, the dispatcher passes the model of data to the View object. The view object uses the data to present its view back to the client.

5.4 DispatcherServlet

- Spring Boot automatically sets up a DispatcherServlet that:
  - recognizes the @Controller and @RequestMapping annotations
  - Automatically sets up a ContentNegotiatingViewResolver and a BeanNameViewResolver
  - includes HttpMessageConverters that can translate objects to JSON or XML
  - serves status content from the '/static', '/public', '/resources' or '/META-INF/resources' folders in the classpath
  - serves Webjars content from the /webjars/** folder in the classpath

5.5 Template Engines

- In addition to REST services, you can use Spring MVC to serve dynamic HTML content using one of several template engines:
  - FreeMarker
  - Groovy
  - Thymeleaf
  - Velocity (although this is deprecated in v1.4)

- ◇ Mustache
- Call out the 'starter' dependency for the template engine you want
    - ◇ e.g. spring-boot-starter-thymeleaf
- If you package your Spring Boot application as a war file, you can use JSP's but it's best to avoid them, so you can run Spring Boot applications in a stand-alone way.

5.6 Spring Boot MVC Example

- To demonstrate each part of a Spring MVC web application, we will develop an application that displays "Hello World"
- The steps are:
    - ◇ Call out the Spring Boot Web starter
    - ◇ Develop the controller
    - ◇ Configure Spring MVC controllers
    - ◇ Configure a template engine
    - ◇ Develop the view definition

Spring MVC Example

These steps assume that you have already configured your DispatcherServlet and context loader.

In the remainder of this section we will develop a simple web application to display "Hello World" or some other configurable message.

5.7 Spring Boot MVC Example

- Call out the Spring Boot Web Starter, by adding the following to the 'dependencies' in 'pom.xml'

```
<dependency>

<groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-web</artifactId>
                </dependency>
```

5.8 Spring Boot MVC Example

- Develop the controller

```
@Controller
public class HelloController  {
       @RequestMapping("/hello")
  public ModelAndView sayHello(
       @RequestParam("userMessage") String message) {
       return new ModelAndView(
         "helloView", "helloMessage", message);
  }
}
```

- The method above is invoked by the request:

```
localhost:8080/hello?userMessage=Howdy
```

- Configure Spring MVC to pick up controllers

  ◇ This is already done in Spring Boot, if you used the '@SpringBootApplication' annotation

Spring MVC Example

First we start by implementing a controller. In Spring MVC 3 this is simply a Java class which has the @Controller Spring stereotype annotation applied to it.

The controller can have arbitrary methods with the @RequestMapping annotation. This will indicate the request that will be handled by the method. This is used by the DispatcherServlet to figure out which controller method to invoke.

This method also has a String parameter that is used as the message. If we wish the user to supply this message the @RequestParameter annotation is used to tell Spring how to use 'request.getParameter' from the Servlet API to extract the request parameter and initialize the method parameter with the value.

The ModelAndView indicates that the next view should be the "helloView" view (logical name).

The ModelAndView includes a bean to be used as the model for the view. This model is the message String that is taken as a request parameter. It also indicates to use the bean name of "helloMessage". A JSP can use this name to refer to the model.

The configuration consists of a <context:component-scan> to detect the Java classes annotated with @Controller that indicates they are Spring components and the <mvc:annotation-driven> to look for @RequestMapping annotations. There is still a "View Resolver" as shown next but gone is the need to register each controller as individual <bean> elements and there is no "Handler mapping" configuration anymore compared to prior Spring MVC versions.

5.9 Spring Boot MVC Example

- Add a templating engine, by adding the following to 'pom.xml' in the dependencies section

```
        <dependency>

    <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-
thymeleaf</artifactId>
            </dependency>
```

Spring MVC Example

5.10 Spring Boot MVC Example

- Add a template under 'src/main/resources/templates'

- e.g. 'helloView.html'

```
<html xmlns:th="http://www.thymeleaf.org">

<body>
  <h1>Hello</h1>
  Hi there, system says: <span th:text="${helloMessage}">blah</span>
</body>
</html>
```

5.11 Spring MVC Mapping of Requests

- Prior to Spring 3 you would use various "Handler Mapping" beans in Spring configuration files to tell Spring how to determine how the incoming requests would map to controllers

- Now in Spring 3, 4 and Spring Boot you simply need to use @RequestMapping annotations on methods on the controller classes

  ◇ This is much simpler and easier to configure how individual methods will be invoked

5.12 Advanced @RequestMapping

- You can have the @RequestMapping annotation at the class level and the method level to combine how a common pattern is used for a set of requests

  ◇ This can even include @PathVariable declarations or the type of HTTP request

```
@Controller
@RequestMapping("/appointments")
public class AppointmentController {
      // gets all appointments
      @RequestMapping(method = RequestMethod.GET)
      public List<Appointment> getAll() { .. }

      // gets only appointments for a specific date
      @RequestMapping(value="/{day}",
            method = RequestMethod.GET)
      public List<Appointment> getForDay(
            @PathVariable Date day) { .. }
```

5.13 Composed Request Mappings

- Spring 4 added "composed" annotations that extend from @RequestMapping

  ◇ @GetMapping

  ◇ @PutMapping

  ◇ @PostMapping

  ◇ @DeleteMapping

■ They add in the appropriate 'method=...', so you don't need to type it.

5.14 Spring MVC Annotation Controllers

■ With Spring MVC 3 the "controller" classes do not need to extend or implement Spring-specific classes

  ◇ All that is needed is the @Controller annotation

■ This means that methods within these classes that are meant to handle requests have a number of options for the method signature

  ◇ What is used depends on what the method needs to do to handle the request

■ There are a number of options for "handler methods":

  ◇ Method parameters

  ◇ Method return types

Spring MVC Annotation Controllers

Rather than trying to detail every possible choice, which is done in the Spring documentation, the next few slides will focus on some of the most common options.

5.15 Controller Handler Method Parameters

■ Some of the most common parameters to controller handler methods are:

  ◇ Domain classes from the application

    ■ Spring can initialize this object matching properties of the object to request parameter names

```
@RequestMapping("/addCustomer")
public String addNewCustomer(Customer customerToAdd) { ..
```
  ◇ Parameters annotated with @PathVariable, @RequestParam, or @CookieValue

```
@RequestMapping("/display/{purchaseId}")
public String displayPurchase(@PathVariable int purchaseId)
```
    ◇ A Map, Spring 'Model', or Spring 'ModelMap' object which can have

objects added for the view that will be rendered

```
@RequestMapping("/edit/{purchaseId}")
public String editPurchase(Model model,
        @PathVariable int purchaseId) {
        Purchase toEdit =
                purchaseService.findPurchaseById(purchaseId);
        model.addAttribute("purchase", toEdit);
```

Controller Handler Method Parameters

It is also possible to have various Servlet API objects, like HttpServletRequest and HttpSession as method parameters. It is suggested not to use this approach though as that is more "low level" than you typically need to be with Spring MVC.

You can also have method parameters annotated with @RequestHeader and @RequestBody or of type HttpEntity<?> for low level access to the HTTP request.

5.16 Controller Handler Method Return Types

- Some of the most common return types of controller handler methods are:

    ◇ A Spring 'ModelAndView' object constructed in the method

```
public ModelAndView listUsers() {
        List<User> allUsers = ...;
        ModelAndView mav = new ModelAndView("userList");
        mav.addObject("users", allUsers);
        return mav;
```
    ◇ A String with the view name

```
public String listUsers(Model model) {
        List<User> allUsers = ...;
        model.addObject("users", allUsers);
        return "userList";
```
    ◇ A Map, Spring 'Model' object, or application domain class

```
public Map<String, Object> listUsers() {
        List<User> allUsers = ...;
        ModelMap map = new ModelMap()
        map.addAttribute("users", allUsers);
        return map;
```

Controller Handler Method Return Types

Notice that returning a ModelAndView object or using a Model object as a parameter and returning a String for the view name are very similar patterns. Both of these have control over the view name to be rendered and share data with that view.

The third option shown for returning just a Map, Model or domain class depends on some default mapping to a view of the request that had come in.

5.17 View Resolution

- Spring uses a ViewResolver to map the logical view name returned by a controller to a View bean

- The View Resolver is automatically configured when you place a dependency on the template engine 'starter'

- Views will correspond to the template engine

  - ◇ e.g. Thymeleaf will resolve a view called 'index' to a template file called 'index.html' in 'src/main/resources/templates'

- You can configure additional view resolvers in your @Configuration classes

- Spring Boot automatically configures a 'ContentNegotiatingViewResolver' and a 'BeanNameViewResolver', but these can be overridden by the template engine starter

View Resolution

A ViewResolver keeps the controller and view aspects of MVC nicely decoupled. Controllers specify a logical name of the view to dispatch to. A ViewResolver maps this logical name to a View object that will handle the presentation. If you want to use a different view for a particular controller, you can change the ViewResolver or the configuration of the ViewResolver to achieve this, without impacting the controller. For example, you may want to change a view from a JSP to a PDF.

5.18 Spring Boot Considerations

- Spring Boot uses an embedded web server

- Embedded web servers don't do well with JSP

- Spring Boot includes auto-configuration for

- ⋄ FreeMarker

- ⋄ Groovy

- ⋄ Thymeleaf

- ⋄ Velocity (although this is deprecated)

- ⋄ Mustache

- ■ Put templates in 'src/main/resources/templates'

5.19 Summary

- ■ Spring MVC provides a rich framework for web applications

- ■ Spring 3 modified the definition of "Controllers" and "Handler Mapping" quite a bit with the use of annotations like @Controller and @RequestMapping

- ■ The Spring MVC implementations of the Controller interface are deprecated and Spring MVC applications should use the new annotation-based approach

- ■ The view resolution and Spring form tags are still the same as defined in Spring 2.x

- ■ With Spring MVC many of the common use cases for applications are simple although you still have access to low-level API if needed

Chapter 6 - Overview of Spring Boot Database Integration

Objectives
Key objectives of this chapter:

- ■ DAO Support in Spring
- ■ Various data access technologies supported by Spring

6.1 DAO Support in Spring

- ■ A data access object (DAO) is a mechanism to read and write data from a database

- ■ DAOs use underlying data access technologies such as JDBC or an ORM framework like Hibernate

- ■ Spring DAO support is designed so it is easy to switch from one data access technology to another
    - ◇ E.g. JDBC, Hibernate, JDO, etc

DAO Support in Spring

The data access object design pattern is a proven pattern in J2EE application architecture. Its primary purpose is to provide a way of keeping upper software layers decoupled from lower level data access technologies and infrastructural details so upper layers can concern themselves with business logic and not have to worry about data persistence.

Spring recognizes the value of this pattern and provides a sophisticated framework based around it.

6.2 DAO Support in Spring
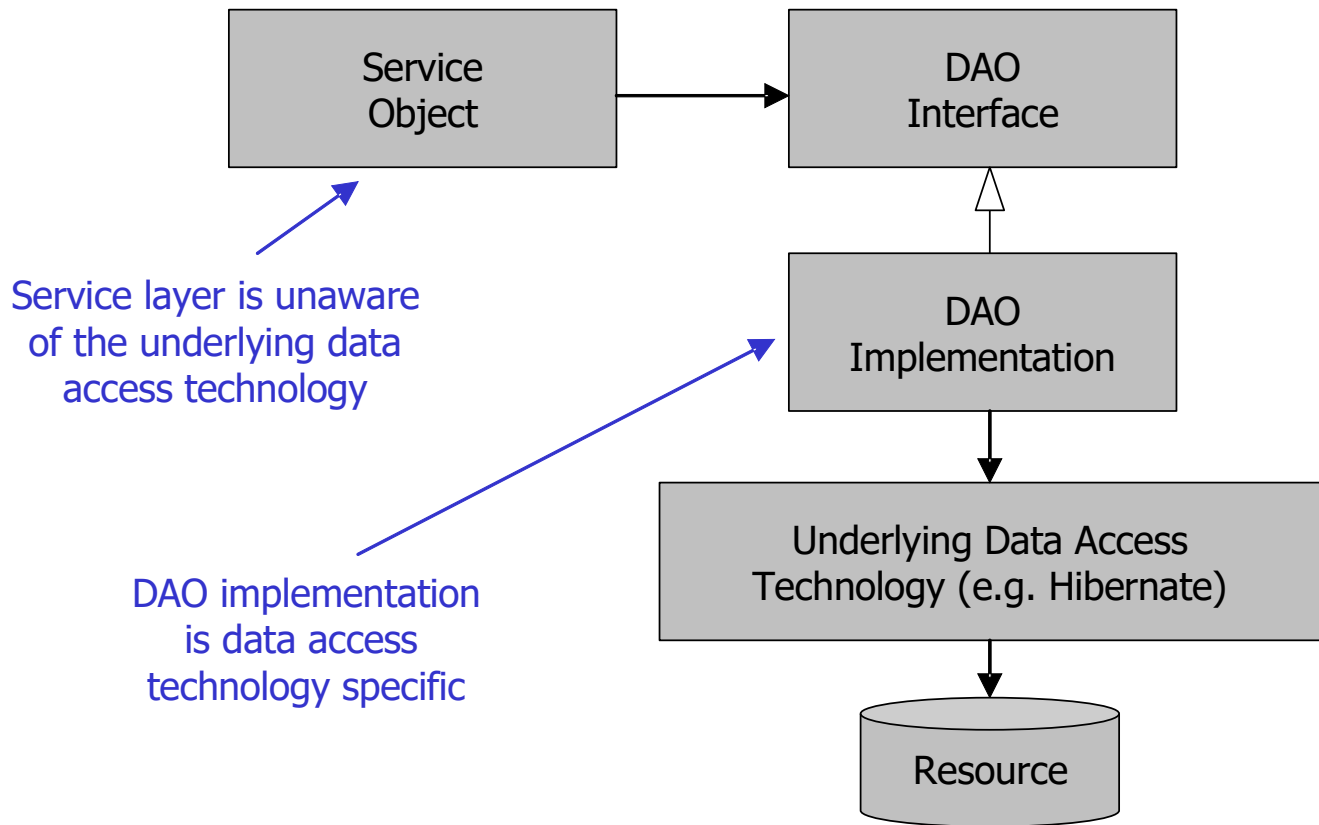
DAO Support in Spring

Spring promotes coding to interfaces and the area of DAOs is no exception. By coding service layers to DAO interfaces, you gain at least two advantages. First, the code is easier to test. Mock DAO implementations can more easily be swapped in during testing of service layers allowing you to fully test a service object without the need for the real DAO implementation and all of its dependencies.

Second, using an interface allows you the flexibility of swapping in a different DAO implementation at deployment time. For example, if you choose to change from JDBC to Hibernate as your data access technology, your service layers are oblivious to this fact and should not need to change.

6.3 Spring Data Access Modules
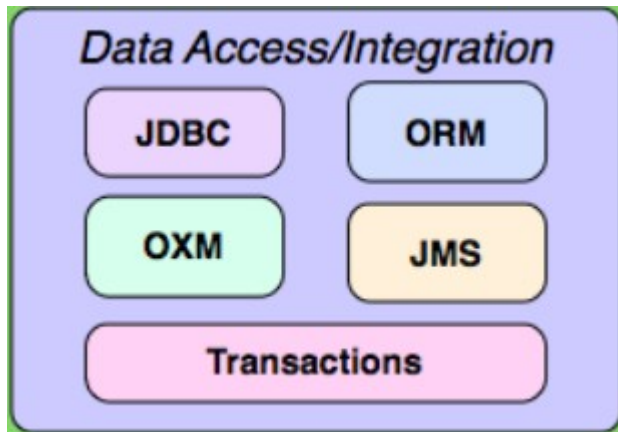
- Spring has 2 modules providing support for various data access options

  ◇ JDBC module

  ◇ ORM module

    - The ORM module depends on the JDBC module

- Both modules depend on the Spring Transaction module

  ◇ Data access is transactional in nature but Spring supports this with a separate module

6.4 Spring JDBC Module

- The Spring JDBC module provides a framework and supporting classes for simplifying JDBC code

  ◇ It is a good fit for applications that already use direct JDBC access and have not migrated to some Object Relational Mapping framework (ORM)

- The Spring JDBC module also has classes for DataSource access

  ◇ This includes DataSource implementations that could be used in testing or when running an application outside a Java EE server

  ◇ This also includes support for an embedded database which could be used in testing

  ◇ The DataSource support from this module is used even when an application is using the Spring ORM module which is why the ORM module depends on the JDBC module

- Much of the classes from this module used directly in applications are some form of JdbcTemplate class

6.5 Spring ORM Module

- The Spring ORM module has supporting classes for the following ORM frameworks:

  ◇ Hibernate

  ◇ Java Persistence API (JPA)

     ▪ This is the relatively new Java standard way to do persistence

  ◇ Java Data Objects (JDO)

  ◇ iBATIS SQL maps

- An application would typically use only one of these frameworks although the Spring ORM module contains support for all four

  ◇ For applications not already using one of these frameworks the JPA standard is generally the default choice since it is a Java standard

6.6 Spring ORM Module

- Code written using one of the ORM frameworks supported by Spring typically uses the persistence framework directly

  ◇ Spring provides supporting classes to integrate other Spring features like transactions with the ORM framework but these are often not seen in the code of the application itself

  ◇ Spring configuration files is the area that largely contains the integration with the ORM framework

  ◇ Spring has various XXXTemplate classes, like HibernateTemplate, but these generally should NOT be used as Spring 3 can properly integrate when the application uses the ORM API directly

6.7 DataAccessException

- The Spring DAO framework insulates higher layers from technology-specific exceptions

  ◇ E.g. SQLException and HibernateException

- Instead, it throws DataAccessException

  ◇ Root cause is still available using getCause()

- This "Exception translation" is one of the key benefits of Spring data access support

  ◇ This keeps higher layers decoupled from lower level technologies

- DataAccessException is a RuntimeException

  ◇ RuntimeExceptions are "unchecked" which means they do not need to be caught in order for the code to compile

  ◇ This is important because it means that application code is not required to catch these Exceptions since nothing can often be done about them anyway

    - Why add a bunch of code to catch Exceptions when you aren't doing anything except ignoring the Exception?

DataAccessException

By throwing DataAccessException from your DAO layer instead of underlying data access specific exceptions like SQLException, you keep upper software layers decoupled from the underlying data access technology. For example, you can change from JDBC to Hibernate without upper layers even knowing about it because they don't deal with SQLExceptions generated by JDBC.

Since DataAccessException is a RuntimeException, you are not forced to catch exceptions that you probably cannot recover from. For example, if the database goes down, there probably isn't much that the application can do about it. Even if there is, you can still catch the exception, you just aren't forced to.

6.8 DataAccessException

- Spring provides a hierarchy of DataAccessException subclasses representing different types of exceptions

  ◇ E.g. DataRetrievalFailureException – Data could not be retrieved

- Spring converts data access technology errors to subclasses in this hierarchy

  ◇ Spring understands some database specific error codes and ORM specific exceptions

■ Spring exceptions are more descriptive
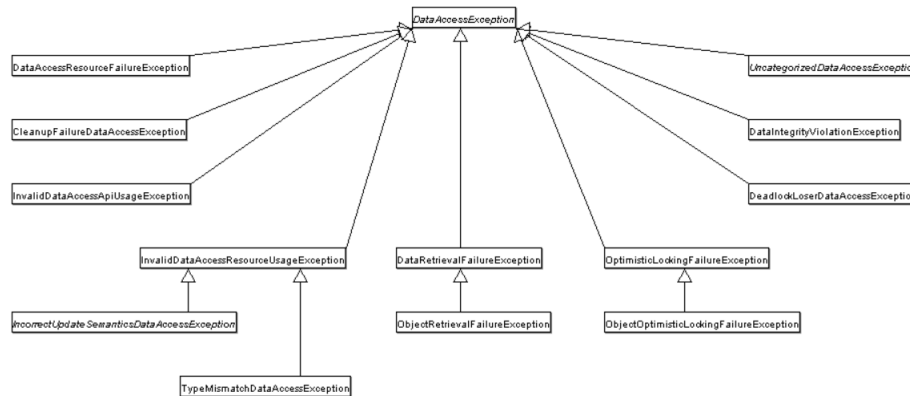
DataAccessException

The Spring DataAccessException hierarchy allows higher layers to code to technology independent exception classes, further supporting the ability to change data access technologies without impacting higher layers. Springs exceptions can be more descriptive than the underlying database error codes/exceptions because the creators of Spring have gone to great lengths to understand and handle database errors from different vendors.

Some examples exceptions include:

TypeMismatchDataAccessException – Type mismatch between Java type and data type

DataIntegrityViolationException – A write operation resulted in a database integrity violation of some kind (e.g. foreign key violation)

DeadlockLoserDataAccessException – The current process entered a database deadlock and was chosen to be the loser

6.9 @Repository Annotation

■ The DataAccessException translation described previously can easily be added to a DAO implementation by adding the @Repository annotation

■ This annotation is one of the Spring "stereotype" annotations for declaring Spring components with an annotation

■ This annotation is the only code that needs to be added to a class to enable the Exception translation functionality

```
@Repository
public class PurchaseDAOJDBCImpl implements PurchaseDAO {
```

6.10 Using DataSources

- Spring Boot will auto-configure a datasource based on the 'application.properties' file

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

- All you need to do then is annotate a DataSource property with '@Autowired'

- For extra convenience, Spring Boot also sets up a JdbcTemplate and NamedParameterJdbcTemplate instance

  ◇ Again, just declare a reference to the JdbcTemplate type and annotate it '@AutoWired'

Using DataSources

A DataSource provides Connection objects to the underlying database and often provide a connection pool for resource management purposes.

The jndi:lookup element is new to Spring 2.x. It requires the use of the new XML schema-based configuration syntax. In the example on the slide, a DataSource object is looked up using JNDI and assigned to the id "dataSource".

DriverManagerDataSource is a simple data source that uses a DriverManager. It is useful for testing. It has properties for driverClassName, url, username, and password.

6.11 DAO Templates

- Data access logic often consists mostly of infrastructural code

  ◇ Resource management, exception handling, connection management, etc

- Spring uses the template method pattern to provide the infrastructural logic for you

- The developer focuses on creating the application-specific logic using a "DAO callback"

DAO Templates

The template method design pattern is illustrated in the classic book "Design Patterns – Elements of Reusable Object-Oriented Software" by Erich Gamma, et al (the "Gang of Four").

6.12 DAO Templates and Callbacks

DAO Template

Prepare resources
Start transaction
Invoke callback

Commit/rollback
Close resources
Handle exceptions

DAO Callback

You write the callback

Execute app logic
Return data
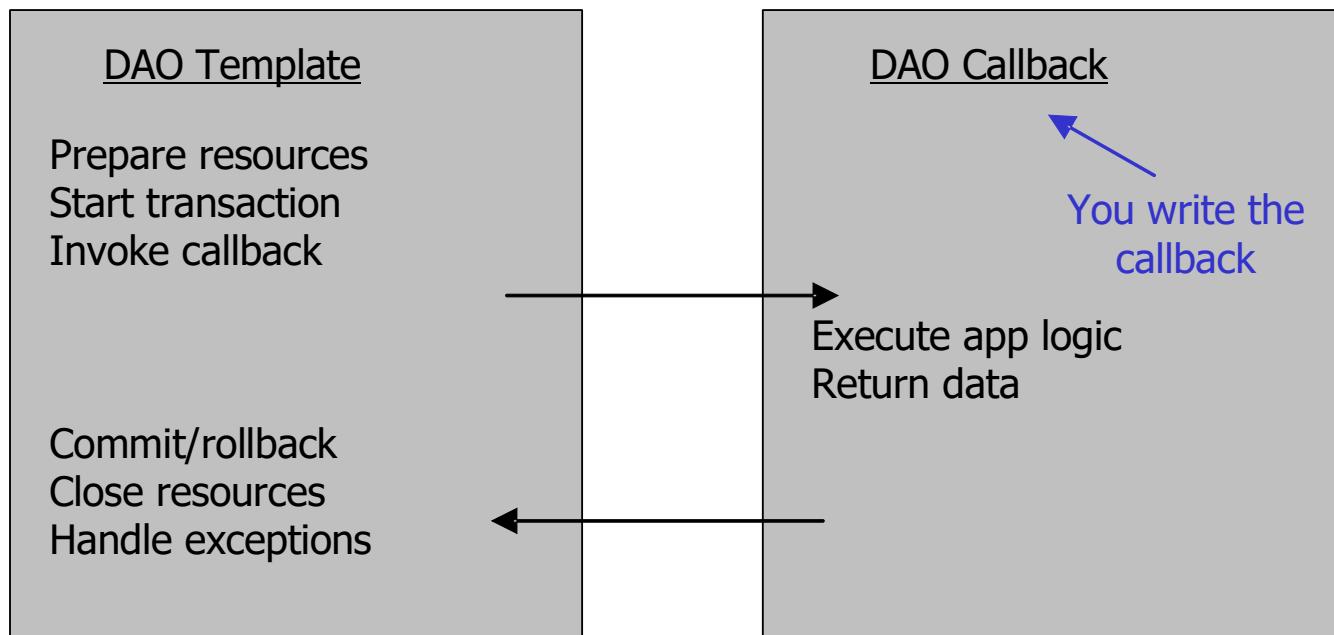
DAO Templates and Callbacks

The diagram on the slide illustrates templates and callbacks. The DAO template is responsible for all of the data access "plumbing" logic such as resource management and exception handling. Spring provides this for you.

The DAO callback is where the application developer writes his/her application specific data access logic.

6.13 ORM Tool Support in Spring

- ORM (object/relational mapping) frameworks provide functionality over JDBC such as:

- ◇ Lazy loading, caching, cascading updates, etc
- ■ Spring provides integration support for several ORM frameworks including:
    - ◇ Hibernate, JDO, iBATIS SQL Maps, Apache OJB
- ■ Spring also provides services on top of these frameworks including:
    - ◇ Transaction management
    - ◇ Exception handling
    - ◇ Template classes
    - ◇ Resource management

ORM Tool Support in Spring

Spring does not provide an ORM framework since several excellent frameworks already exist. However, it does provide sophisticated integration with many of these frameworks as well as adding additional services on top of them.

6.14 Summary

- ■ Spring provides modules to simplify data access code
    - ◇ These modules are some of the most used features of Spring
- ■ No matter what technology is used Spring provides a common architecture so it is easy to switch data access technologies with minimum impact on the rest of an application
- ■ The Spring @Repository annotation should be used on DAO components to enable Spring's data Exception translation

Objectives
Key objectives of this chapter: ■ Using Spring with JPA ■ Using Spring with Hibernate

7.1 Spring JPA

■ Besides basic JDBC support, Spring also provides support for ORM, or Object Relational Mapping

   ◇ ORM is the ability of mapping values of Java objects in memory to tables and columns in a database and having the environment automatically synchronize the two

■ Spring Boot provides an JPA module that provides this support

```
<dependency>
    <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
```

■ This brings in

   ◇ Java Persistence (JPA)

   ◇ Hibernate

   ◇ Spring Data JPA

■ Although Spring does support the "template" approach like with JDBC, you are often using the ORM framework natively and using Spring just to bootstrap it into the environment

7.2 Benefits of Using Spring with ORM

■ Simply using an ORM framework can simplify an application considerably

   ◇ You can simply indicate what is persisted and let the framework calculate the actual SQL code to accomplish that

- Using Spring's ORM support in addition to an ORM framework provides the following benefits:

    ◇ Spring can provide the configuration for the ORM framework as Spring configuration which can make it easy to swap out different configurations for testing, production, etc

    ◇ Spring can translate the data access exceptions of the ORM framework into a standardized set of Spring exceptions

    ◇ Spring can provide access to the ORM managed resources avoiding many common issues when used without Spring

    ◇ The ORM support of Spring also integrates with the transactional support so that transactional behavior of a Spring application is consistent and the ORM framework is integrated into this correctly

7.3 Spring @Repository

- One of the Spring configuration stereotype annotations is specifically designed for Spring components in an application that work with data access

    ◇ This is the @Repository annotation

- By using this annotation (and the appropriate annotation scanning XML configuration) you can register the Java class as a Spring component and enable the Spring data access exception translation

    ◇ Java class:

```
@Repository
public class ProductDaoImpl implements ProductDao {
```

    ◇ Spring Boot normally includes @ComponentScan

    ◇ Spring configuration:

```
<context:component-scan ... />
<context:annotation-config />
```

Spring @Repository

In the example above there are two <context:...> elements in the XML configuration. Both of these are required to get the detection of the @Repository annotation and the registration of the Spring data

access exception translation. The <context:component-scan ..> element is what would scan the Java class to pick up the @Repository annotation while the <context:annotation-config /> registers the PersistenceExceptionTranslationPostProcessor Spring class as a bean to enable exception translation.

If you want to register the exception translation Spring bean manually in XML instead of using <context:annotation-config /> you could use the following although there is not much point to since you are using some kind of XML configuration either way:

<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

7.4 Using JPA with Spring

- The use of JPA is based on the JPA 'EntityManager'

  ◇ This is the interface used to persist or query data managed by JPA

  ◇ Java classes that are mapped by JPA to database tables are 'Entities'

```
@Entity
public class Product { ...
```

- The EntityManager is often injected into a Spring component so that code in the rest of the Spring component can use JPA directly

```
public class PurchaseDAOJPAImpl implements PurchaseDAO {
     @PersistenceContext
     private EntityManager em;
```

  ◇ The @PersistenceContext annotation is a standard JPA annotation that is used for injection provided the Spring <context:annotation-config /> element is in your Spring configuration

- Spring 3 can be used with JPA 2.0 as long as the JPA provider being used also supports JPA 2.0

  ◇ Hibernate 3.5+ would support JPA 2.0

Using JPA with Spring

Technically it is possible to register the appropriate Spring bean post processor directly instead of using <context:annotation-config /> to enable Spring injection of @PersistenceContext but again it is not really beneficial to do this manually.

<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

7.5 Configure Spring Boot JPA EntityManagerFactory

- Spring Boot automatically sets up an EntityManagerFactory, so you can just inject the EntityManager with @AutoWired or @PersistenceContext

- In Spring Boot, we don't need a 'persistence.xml' file

  ◇ All Entity classes are picked up by the classpath scanner.

7.6 Application JPA Code

- Although Spring is providing the JPA configuration the actual code that works with persisted data can use only the JPA API

```
public class PurchaseDAOJPAImpl implements PurchaseDAO {
      @PersistenceContext
      private EntityManager em;

      public void savePurchase(Purchase purchase) {
            if (getPurchase(purchase.getId()) == null) {
                  em.persist(purchase);
            } else {
                  em.merge(purchase);
            }
      }

      public List<Purchase> getAllPurchases() {
            Query q = em.createQuery("select p from
Purchase p");
            List<Purchase> results = (List<Purchase>)
                  q.getResultList();

            return results;
      }
```

Application JPA Code

The above code could also have Spring annotations at the class or method level such as @Repository or @Transactional but this could also be configured in Spring configuration files if the goal is to not have ANY Spring classes mentioned in the application source code.

You can have this Spring configuration in addition to the above JPA code:

<beans>

        <context:annotation-config/>

        <bean id="myPurchaseDao" class="com.mycom.PurchaseDAOJPAImpl"/>

        <!-- Various <aop:config> and <tx:advice> transaction configuration -->

</beans>

instead of the following Spring annotations in the class:

@Repository

@Transactional

public class PurchaseDAOJPAImpl implements PurchaseDAO {

7.7 "Classic" Spring ORM Usage

- The preferred way to use ORM frameworks with Spring is described in the lecture up to this point

- Some older Spring applications though may still use the older ways of doing ORM which used more of the XXXTemplate and XXXCallback approach to using ORM

  - ◇ This style is described in the rest of the chapter

  - ◇ This style was also more similar to the JDBCTemplate approach of Spring JDBC so it seemed more "familiar" at the time

- The problem with this style is that it has Spring API classes used in the application code creating a dependence on Spring (besides for just configuration)

  - ◇ Since it is now possible to write fully functional ORM code using only the API of the ORM framework (JPA, Hibernate, etc) and use Spring only for the configuration that approach is preferred

7.8 Spring JpaTemplate

- Use Springs JpaTemplate to access JPA

```
<bean id="jpaTemplate"
```

```
      class="org…JpaTemplate">
    <property name="entityManagerFactory">
      <ref bean="entityManagerFactory"/>
    </property>
  </bean>
  …
  <bean id="orderDao"
      class="com…OrderDaoJpaImpl">
    <property name="jpaTemplate">
      <ref bean="jpaTemplate"/>
    </property>
  </bean>
```

Spring JpaTemplate

Spring provides a DAO template for JPA called JpaTemplate. All it requires is an EntityManagerFactory. Once a JpaTemplate bean is configured in your Spring configuration file, you can wire it into DAO beans as needed.

The methods of JpaTemplate are thread-safe so you only need to create one, singleton instance of JpaTemplate for all DAOs.

JpaTemplate is in the org.springframework.orm.jpa package.

7.9 Spring JpaCallback

- Use Springs JpaCallback to provide application specific logic

```
public List getOrders(final Integer prod) {
   return (List)jpaTemplate.execute(
     new JpaCallback() {
       public Object doInJpa(EntityManager mgr)
           throws PersistenceException {
         Query q = mgr.createQuery("from Order…");
         return q.execute(prod);
       }
     });
}
```

- Exception handling, resource management, etc is handled by JpaTemplate

Spring JpaCallback

The JpaCallback interface defines one method called doInJpa. The developer places their application specific logic in this method and returns an object that will ultimately be returned by the JpaTemplate.execute method. The developer does not need to handle exceptions or deal with resource management.

PersistenceException and EntityManager are in the javax.persistence package (i.e. they are part of the JPA).

7.10   JpaTemplate Convenience Features

- JpaTemplate has several convenience methods for common requests
  - ◇ E.g. find method (using JPA query language)

```
List list = jpaTemplate.find("from Order");
```

- Spring also provides the JpaDaoSupport convenience class
  - ◇ This base class provides get and setEntityManagerFactory and getJpaTemplate methods

JpaTemplate Convenience Features

Similar to HibernateTemplate, JpaTemplate also provides several convenience methods for common operations.

Spring also provides the JpaDaoSupport class. You can code your DAO classes to extend JpaDaoSupport so you don't have to code getters and setters for common properties such as the EntityManagerFactory and JpaTemplate. JpaDaoSupport is in the org.springframework.orm.jpa.support package.

7.11   Spring Boot Considerations

- Spring Boot will automatically configure an embedded database if there's one available on the classspath
  - ◇ H2
  - ◇ HSQL
  - ◇ Apache Derby

- At startup time, it runs the scripts 'schema.sql' and 'data.sql' from the resources root.

- Override by configuring 'spring.datasource.*' in 'application.properties'

7.12 Spring Data JPA Repositories

- Spring Data JPA repositories are interfaces that you define to access data

- Spring Data reads the name of the methods on your interface and automatically creates queries to implement them

```
public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

- e.g. 'findByNameAndCountryAllIgnoringCase' provides enough information (along with the type definitions) to synthesize a query

7.13 Summary

- Spring provides exceptional support for various ORM frameworks

- The preferred approach in modern Spring applications is:

  ◇ Use the JPA API to create standards-based persistence code

  ◇ Use a JPA provider library (like Hibernate) to provide the JPA support

  ◇ Use Spring to configure JPA and link to other Spring configuration or the environment in a Java EE 5+ server

- Direct use of the Hibernate API in application code is still possible

Chapter 8 - Introduction to MongoDB

Objectives

This chapter will cover the following topics:

- Basic information about MongoDB

- MongoDB Query Language

- MongoDB vs Apache CouchDB

8.1 MongoDB

- MongoDB is an open-source document-oriented NoSQL database developed and supported by *MongoDB Inc*.

- Main features:

  ◇ Dynamic schemas

    - Changes to database schema can be done real-time without affecting existing data or application code

  ◇ JSON-based dynamic data model

    - JSON-based documents are serialized into BSON (Binary JSON) format and persisted directly on the data store

  ◇ Auto-sharding

    - Together with data replication, it ensures data redundancy and high data availability

    - Each shard is treated by MongoDB as an independent physical database partition

Notes

MongoDB gets its name from the word "hu**mongo**us".

Written in C++ for run-time efficiency; binaries are available for Windows, Linux, OS X, and Solaris.

MongoDB stores JSON-based documents in format called BSON (Binary JSON, pronounced Bee-Son). The BSON data format was developed by MongoDB engineers.

Sharding (data set partitioning) is a technique for achieving horizontal scalability where data storage location is determined by calculating some data-derived index, e.g. a hash value of the key or selected

attribute; each partition is called a "shard".

The term "sharding" was coined by Google engineers, and made mainstream after publication of Google's BigTable architecture.

Designs of many NoSQL databases provide for dynamic scaling out via auto-sharding.

Sharding also improves system overall throughput.

Sharding is also related to a "**shared nothing architecture**"—once a data set was sharded, each shard can live on its own in a totally separate physical database server.

8.2 MongoDB Features

- ⬦ Rich queries

  - ▪ Searches can be performed by field, range or regular expression

- ⬦ Full-index support for faster querying

  - ▪ any document field can be indexed

- ▪ MongoDB uses master/slave replication with the master performing both reads and writes while a slave only performing reads from data copied over from the master. The slaves can select a new master should the current one goes down

8.3 MongoDB on the Web

- ▪ Main site:

  ```
  https://www.mongodb.com
  ```

- ▪ Community Server Download Page

  ```
  https://www.mongodb.com/download-center?
  jmp=homepage#community
  ```

- ▪ Logo

8.4 Positioning of MongoDB
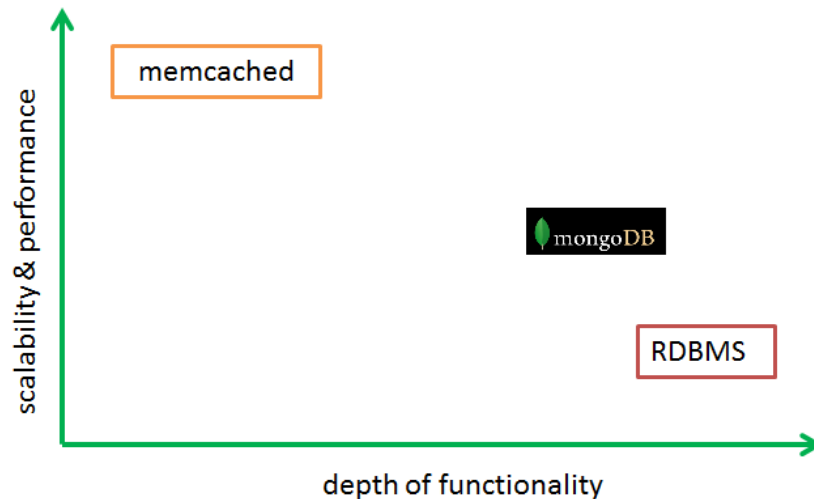
Source: Dwight Merriman, Founder and CEO of MongoDB Inc.

Notes:

While many key/value data stores (e.g. memcached) are extremely fast, they are also limited in functionality. Those stores are basically implemented as in-memory caches that do not support data search and data retrieval is only possible by keys.

8.5 MongoDB Applications

- MongoDB can be administered via a command-line shell or dedicated GUI
  - ◇ Some GUI Choices include:
    - Robo 3T (Formerly Robomongo) - Free
    - Compass - commercial product from same company as MongoDB

- MongoDB supports a number of clients via appropriate client libraries called "drivers"
- Drivers are available for many programming languages:

- ◇ C, C++
- ◇ Java, C#, .NET
- ◇ Node.js
- ◇ Perl, PHP, Python, Ruby, Scala,...

8.6 MongoDB Data Model

- A MongoDB system holds a set of databases
- A database holds a set of collections (similar to tables in relational databases)
- A collection holds a set of documents (a document instance is conceptually similar to a record in relational databases)
- A document consists of a set of fields
- Each document is uniquely identified by a primary key field

8.7 MongoDB Limitations

- These common database features are not supported:
  - ◇ Transactions
  - ◇ Joins between collections
- Inserts and updates are only atomic at document level
- More information about these limitations is available here:

  ```
  https://docs.mongodb.com/manual/core/write-
  operations-atomicity/
  ```

8.8 MongoDB Use Cases

- Real-time data analytics
- Content Management Systems
  - ◇ Documents retrieved from MongoDB after conversion from BSON into

the JSON format can be directly rendered in the Web layer via the client-side Java Script

- High volume and throughput of data-driven web sites

- The server-side infrastructure for

  ◇ operations that require real-time inserts, updates, and queries

  ◇ geospatial indexes

- Agile development and fluid business requirements

  ◇ MongoDB's dynamic data model lends itself to iterative/agile development methodologies where the "ALTER TABLE" relational database style of operations is avoided

  ◇ Superior Big Data experiences is delivered at any scale

- More information on use-cases can be found here:

```
https://docs.mongodb.com/ecosystem/use-cases/
```

8.9 MongoDB Query Language (QL)

- MongoDB has its own QL which is intuitive and easy to use

- The use of MongoDB QL in various MongoDB drivers (clients) has slight variations related to language semantics (e.g. the Java client will return the Java-specific *iterable* interface to navigate the returned list of documents)

- You retrieve documents from MongoDB using either of the following methods on the target collection:

  ◇ *find*

  ◇ *findOne*

- MongoDB supports the full CRUD set of operations

8.10 The CRUD Operations

- The Collection interface supports an extended set of CRUD (create, read,

update, and delete) operations:

- ◇ Create
- ◇ Read
- ◇ Update
- ◇ Delete
- ◇ Count
- ◇ Replace
- ◇ Aggregate
- ◇ Distinct
- ◇ Bulk, One or Many
- ◇ Find and Modify

8.11 The find Method

- The *find ()* method returns documents that match the find's *query* parameter. The method has the following syntax:

  ```
  <collection>.find( <query>, <projection> )
  ```
  - ◇ The find() method is analogous to the SELECT statement in SQL

  - ◇ **Note**: To produce a well-formed JSON document for output, use the *pretty()* method on *find()*

    ```
    <collection>.find().pretty()
    ```
  - ◇ The <query> parameter corresponds to the SQL's WHERE clause; it contains attribute(s) of the underlying JSON document used in search predicates

  - ◇ The <projection> parameter corresponds to the SELECT's list of fields to return

8.12 The findOne Method

- Most MongoDB drivers (clients) provide native API to this method

- If used without parameters, *find*() will return all documents in the collection

(or more precisely, an iterable cursor to all documents)

- The following find command will return all documents from an order_history document collection with *year* attribute's value of 2000:

```
order_history.find( { year: 2000 } )
```

- The *findOne()* has syntax similar to the *find()* method and returns the first document matching the *query* parameter.

  ◇ Internally, the *findOne()* method delegates the call to the *find()* method with a limit of 1

8.13 A MongoDB Query Language (QL) Example

- The MongoDB query language provides excellent support for *ad hoc* queries

- The script below runs from the MongoDB interactive JavaScript-based shell and fetches all employees in job level 7 who earns wages in excess of 90,000 ordered by salary in ascending order

```
> db.employee.find(
      {job_level : 7, { salary : { $gt : 90000}}}
      ).sort({salary:1});
```

- The command is functionally equivalent to the following SQL statement:

```
select * from employee where job_level=7 and
      salary > 90000 order by salary asc
```

8.14 Inserts

- The following command will perform an insert of a simple document in the *user* collection (we also supply our own value for the *_id* primary key):

```
db.user.insert(
   {
      _id: "some_new_user",
       created: new Date()
```

```
            }
        )
```

8.15 MongoDB vs Apache CouchDB

■ MongoDB and CouchDB are both noSQL databases

■ They are both:

◇ Document centric

◇ Support dynamic schemas

◇ Scale easily across multiple instances

■ MongoDB

◇ Favours Consistency

◇ Supports dynamic queries on constantly changing data

◇ Supports Master-Slave replication

■ CouchDB

◇ Favours Availability

◇ Works best with occasionally changing data and pre-defined queries

◇ Offers Master-Master replication

8.16 Summary

■ In this chapter we reviewed:

◇ Main features of MongoDB

◇ Introduced the MongoDB query language

◇ Discussed how to choose between MongoDB and CouchDB

Chapter 9 - Working with Data in MongoDB

Objectives
This chapter will cover the following topics:

- ■ CRUD operations in MongoDB
- ■ Cursors
- ■ Aggregation pipe-lines

9.1 Reading Data in MongoDB

- ■ Read operations are based on queries that allow users select and retrieve documents from a database collection

- ■ MongoDB queries are similar to SQL's SELECT statements in that you can specify document retrieval criteria

  - ◇ The generic queries retrieve the matching documents in their entirety (with all the fields in them) and are similar to the SELECT * … type of SQL statement

- ■ To limit the amount of data returned to the user, MongoDB queries can optionally include a projection that specifies the fields from the matching documents to return

9.2 The Query Interface

- ■ The query interface is centered on the *db.collection.find()* method invoked on the target collection from which you plan to retrieve the matching documents

- ■ The *find()* method accepts both the query criteria and projections (the list of fields to be returned) and returns a cursor pointing to the result set of matching documents

  - ◇ A variant to the *find ()* method is *findOne()* which returns the first matching document

9.3 Query Syntax is Driver-Specific

- The *find()* method invocation syntax depends on your driver (client application)

- MongoDB documentation uses JavaScript syntax used in the MongoDB Admin shell, e.g.

```
cursor  = db.things.find
         ({field1: {$ne: 0}, field1: {$gt: 1000} });
```

  ◇ The *$ne* and *$gt* operators are equivalent to '!=' and '>' (more on the supported operators a bit later ...)

- The same query in the Java driver will look as follows:

```
query = new BasicDBObject("field1", new BasicDBObject("$ne", 0))
    .append("field2", new BasicDBObject("$gt", 1000));

cursor = coll.find(query);
```

9.4 Projections

- By default, the *find()* method returns all fields in all matching documents

- A projection in a query allows users to exclude some of the fields

- A projection object is the second (optional) argument to the *find()* method

- A projection can either specify a list of fields to return or a list of fields to exclude in the matching documents

- To include a field, it must be have a value of 1 in projection; a field will be excluded if it has value 0

```
db.clients.find( dob: {$gt: 1980}, {"income": 0} )
```

- The above statement will return all fields in the clients collection except for *income* (the {"income": 0} document is a projection)

- **Note:** The *_id* field is returned by default, so you will need to explicitly exclude it with this projection: *{"_id" : 0}*

9.5 Query and Projection Operators

- MongoDB uses the following operators in queries and projections:

- ◇ **$eq** - Matches values that are equal to a specified value

- ◇ **$gt** - Matches values that are greater than a specified value

- ◇ **$gte** - Matches values that are greater than or equal to a specified value

- ◇ **$lt** - Matches values that are less than a specified value

- ◇ **$lte** - Matches values that are less than or equal to a specified value

- ◇ **$ne** - Matches all values that are not equal to a specified value

- ◇ **$in** - Matches any of the values specified in an array

- ■ For complete list of operators, visit http://docs.mongodb.org/manual/reference/operator/query/

9.6 MongoDB Query to SQL Select Comparison

SQL SELECT	MongoDB's find()
SELECT * FROM users	db.users.find()
SELECT * FROM users WHERE user_id = "qwerty12345"	db.users.find({ user_id: "qwerty12345" })

SELECT user_id, email FROM users	db.users.find(
	   { }, # no query parameters
	   { user_id: 1, email: 1 } # this is a projection
	)

- ■ For information, see https://docs.mongodb.org/manual/reference/sql-comparison/

9.7 Cursors

- ■ The *find()* method returns a cursor (a result set) pointing to the matching documents in the target collection

- ■ To sequentially access the matching documents, you need to iterate over the cursor

- ■ Cursor iteration is supported by the *hasNext()* and *next()* cursor methods

```
var myCursor = db.mycoll.find();

while (myCursor.hasNext()) {
   print(tojson(myCursor.next()));
}
```

- **Note:** A cursor does not put an exclusive lock on the underlying collection -- in other words, a cursor is not isolated

  ◇ To prevent changes to the underlying cursor, you need to use *snapshot* cursor mode by using the *snapshot()* method on the cursor

9.8 Cursor Expiration

- By default, MongoDB will automatically close the cursor after 10 minutes of inactivity

- To prevent cursor expiration, use the *DBQuery.Option.noTimeout* parameter to the cursor's *addOption()* method:

```
var myCursor = db.myColl.find().addOption(DBQuery.Option.noTimeout);
```

9.9 Writing Data in MongoDB

- Write operations in MongoDB work on a single collection and are atomic on the level of a single document

- MongoDB supports the following write operations:

- **insert** - add a new document to a collection

  ◇ The first insert into a collection creates a collection with the name specified as the target of the insert operation

- **update** - modify the existing data; can take a criteria as a parameter

- **remove** - delete data from the target collection; can take a criteria as a parameter

- **Note**: The insert, update, or remove operations can not affect more than one document atomically

9.10 An Insert Operation Example

- An *insert* operation will insert a new document in the target collection (*myColl* in the example below):

```
db.myColl.insert ( {p1: "value", p2 : 1234})
```

- The *_id* field, if not specified (as in the above example), will be created and inserted by MongoDB automatically

9.11 The Update Operation

- The method has the following signature:

    - `db.collection.update(query, update, options)`
- The *update()* method takes the following parameters:

    ◇ A query for locating the document(s)

    - By default, the *update()* method updates a single document; you need to set the *multi* parameter in the options part to *true* to update all documents that match the query criteria

    ◇ An update object

    - the update object is set as a value to the *$set* property

    ◇ Options

9.12 An Update Operation Example

```
db.employee.update (
   {name: "John Doe"}, # the query to locate the record for update
      {$set : {job_category : 8}}, # the update object
      {multi : true}) # options
```

- The above update operation will set the job_category of all the John Doe's in the company to 8

9.13 A Remove Operation Example

```
db.employee.remove( {job_category : 8} )
```

- The above *remove* operation will delete all employees with the job category 8 from the *employee* collection

9.14 Limiting Return Data

- The cursor object offers the *limit()* method for limiting the number of documents returned by a query

- The *limit()* method is similar to the LIMIT operator used in most RDBMSes

- It takes a number of documents to be returned

```
db.collectrion.find().limit(3)
```

9.15 Data Sorting

- The *sort()* method of a cursor allows users specifies the order in which the matching documents are to be returned

- The *sort()* method takes a document that defines the desired sorting order in the `{field : sort_order, …}` format

- The sort order of a field is specified as follows

  ◇ a value of 1 specifies an ascending sort order

  ◇ a value of -1 specifies a descending sort order

```
db.clients.find().sort({ client_id: 1, debt: -1 })
```

9.16 Aggregating Data

- MongoDB provides an extensive set of aggregation operations that are performed on a collection

- As of version 2.2, MongoDB supports the building of multi-stage data processing pipelines consisting of operations (referred to as stages) for data

- Operations that can be used in processing pipelines include:

  - data filtering, finding unique values, data grouping, matching, sorting, transformation, etc.

- The aggregation operations in a pipe-line are grouped together as elements of an array passed as a parameter to the *aggregate()* method of the target collection:

```
db.collection.aggregate ([ {stage1}, {stage2}, ...])
```

9.17 Aggregation Stages

- Aggregation operations (stages) are identified as key / value pairs where keys start with a *'$'*, e.g.

  - **$match** - filter by a condition (similar to the WHERE SQL clause)

  - **$skip** - skip the first n documents

  - **$group** - group input documents by a specified identifier expression

  - **$sort** - reorder documents by a specified sort key

- For a complete list of aggregation stages, visit http://docs.mongodb.org/manual/meta/aggregation-quick-reference/

9.18 Accumulators

- Accumulators are aggregation functions that are available only for the *$group* stage

- Accumulation is done on a specified group key

- Useful accumulation operations:

  - **$avg** - return average for each group (non-numeric values are ignored)

  - **$sum** - return a sum for each group (non-numeric values are ignored)

  - **$min / $max** - return minimum and maximum value for each group

- For more information, visit http://docs.mongodb.org/manual/meta/aggregation-quick-reference/

9.19 An Example of an Aggregation Pipe-line

```
db.sales.aggregate( [
    $group: {
      _id: "$month",
      salesPerMonth: { $sum: "$sale" }
}} ] )
```

is functionally equivalent to the following SQL statement:

```
SLECT month, SUM(sale) AS salesPerMonth FROM sales GROUP BY month
```

- For more examples, visit http://docs.mongodb.org/manual/reference/sql-aggregation-comparison/ and http://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set/

9.20 Map-Reduce

- Leveraging the success of the MapReduce (one word) distributed data processing model (for example, the one used by Hadoop), MongoDB also offers a similar capability, called Map-Reduce

- For Map-Reduce specific operations, MongoDB provides the *mapReduce()* method on a target collection

- **Note**: According to MongoDB documentation,

  ◇ *"For most aggregation operations, the Aggregation Pipeline provides better performance and more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline."*

- The coverage of this topic is beyond the scope of this module

- For more information, visit http://docs.mongodb.org/manual/core/map-reduce/

9.21 Summary

- MongoDB supports the CRUD operations in the form of *insert(), find(), update(),* and *remove()* methods

- You can also use limiting and sorting type of operations found in SQL

- MongoDB offers a rich set of data aggregation operations that can be organized into aggregation pipe-lines; support for the Map-Reduce programming model is also provided

- MongoDB documentation recommends aggregation pipelines over Map-Reduce

Chapter 10 - Spring Data with MongoDB

10.1 Why MongoDB?

- MongoDB fits into Spring's philosophy of *auto-configuration*
- High performance and massively scalable
- Auto shardable
- Document oriented (JSON-like)
- Schema-less
- Distributed
- Free and open-source (GNU license)

10.2 MongoDB in Spring Boot

- Add MongoDB support to *pom.xml*
- Create an interface that extends *MongoRepository*
  - ◇ Add custom find method
- Configure properties in *application.properties*
- Use your Repository

Note

Spring supports both *MongoRepository* and *MongoTemplate* to access MongoDB. In this module we are focusing on MongoRepository.

10.3 Pom.xml

- Add the following dependency to pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

10.4 Application Properties

```
#application.properties
spring.data.mongodb.host=
spring.data.mongodb.port=
spring.data.mongodb.username=
spring.data.mongodb.password=
spring.data.mongodb.database=
spring.data.mongodb.repositories.enabled=
spring.data.mongodb.uri=
spring.data.mongodb.authentication-database=
spring.data.mongodb.field-naming-strategy=
spring.data.mongodb.grid-fs-database=
```

Note

Spring connects automatically to MongoDB in its default configuration. If you have customized MongoDB you may need to explicitly define some of these properties.

10.5 MongoRepository

- Spring interface for MongoDB access
- Defines CRUD methods for MongoDB access (some examples)
  - ◇ Create: **insert**
  - ◇ Read: **find, findAll, findOne**
  - ◇ Update: **save, saveAll**
  - ◇ Delete: **delete, deleteAll**
- Extend MongoRepository to add custom methods

10.6 Custom Query Methods

- Extend the **MongoRepository** interface

  ◇ Date type: SushiBar

  ◇ ID type: String

- Add query method declarations following naming convention

- Spring generates query method implementations

```
// query method examples
public interface SushiRepository
                    extends MongoRepository<SushiBar, String> {
  List<SushiBar> findByCity(String city);
  List<SushiBar> findByRatingGreaterThan(int rating);
  List<SushiBar> findByCityAndRating(String city, int rating);
  List<SushiBar> findByLocationNear(Point point);
}
```

Note

You only have to define the interface, Spring generates the implementation.

10.7 Supported Query Keywords

Keyword	Sample
(No Keyword)	findByAge(int age)
GreaterThan	findByAgeGreaterThan(int age)
LessThan	findByAgeLessThan(int age)
Between	findByAgeBetween(int min, int max)
IsNotNull, NotNull	findByFirstnameNotNull()
IsNull,Null	findByFirstnameNull()
Like	findFirstnameLike(String name)
Not	findByFirstNameNot(String name)
Near	findByLocationNear(Point point)

Within	findByLocationWithin(Circle circle)
Within	findByLocationWithin(Box box)

Note

The *Near* and *Within* keywords support MongoDB geospatial queries.

10.8 Complex Queries

- Query keywords can be combined

```
List<Person> findByNameAndTitle(String name, String title);

List<SushiBar> findByCityAndRatingGreaterThan(
                                  String city, int rating);
```

10.9 Create JavaBean for Data Type

- Use the **@Document** annotation to specify collection name

```
@Document(collection="restaurants")
public class SushiRestarant {

  @Id private String id;
  private String name;
  private in rating;

  public SushiRestaurant() {}

  public String getId(){return id;}

  public String getName(){return name;}
  public void setName(String name){this.name=name;}

  public int getRating(){return rating;}
  public void setRating(int rating){this.rating=rating;}

}
```

10.10 Using the Repository

```
@RestController
@SpringBootApplication
@RequestMapping("/sushi")
public class SushiServiceApplication {

  @Autowired
  private SushiRepository sushiRepository;

  // CREATE
  @RequestMapping(
              value = "/add",method = RequestMethod.POST)
  public ResponseEntity addCar(
          @RequestBody SushiRestaurant sushiRestaurant)
                              throws URISyntaxException{
    sushiRepository.save( sushiRestaurant );
    ...
```

10.11 Summary

In this module we examined

- MongoDB

- MongoDB in Spring Boot

- MongoRepository

- CRUD operations

Objectives
Key objectives of this chapter

- A Basic introduction to REST-style services

- Using Spring MVC to implement REST services

- Combining the JAX-RS standard with Spring

11.1 Many Flavors of Services

- Web Services come in all shapes and sizes

  ◇ XML-based services (SOAP, XML-RPC, RSS / ATOM Feeds)

  ◇ HTTP services (REST, JSON, standard GET / POST)

  ◇ Other services (FTP, SMTP)

- While SOAP is the most common style of service, increasingly organizations are utilizing REST for certain scenarios

  ◇ REpresentational State Transfer (REST), first introduced by Roy Fielding (co-founder of the Apache Software Foundation and co-author of HTTP and URI RFCs)

  ◇ REST emphasizes the importance of resources, expressed as URIs

  ◇ Used extensively by Amazon, Google, Yahoo, Flickr, and others

11.2 Understanding REST

- REST applies the traditional, well-known architecture of the Web to Web Services

  ◇ Everything is a resource

  ◇ Each URI is treated as a distinct resource and is addressable and accessible using an application or Web browser

  ◇ URIs can be bookmarked and even cached

- Leverages HTTP for working with resources

- ◊ GET – Retrieve a representation of a resource. Does not modify the server state. A GET should have no side effects on the server side.

- ◊ DELETE – Remove a representation of a resource

- ◊ POST – Create or update a representation of a resource

- ◊ PUT – Update a representation of a resource

Understanding REST

The notion of a resource "representation" is very important within REST. Technically you should never have a direct pipeline to a resource, but rather access to a representation of a resource. For example, a resource which represents a circle may accept and return a representation which specifies a center point and radius, formatted in SVG, but may also accept and return a representation which specifies any three distinct points along the curve as a comma-separated list. This would be two distinct representations of the same resource.

11.3 RESTful Services

- ■ A RESTful Web service services as the interface to one or more resource collections.

- ■ There are three essential elements to any RESTful service

  - ◊ Resource Address – expressed as a URI

  - ◊ Representation Format – a known MIME type such as TXT or XML, common data formats include JSON, RSS / ATOM, and plain text

  - ◊ Resource Operations – a list of supported HTTP methods (GET, POST, PUT, DELETE)

11.4 REST Resource Examples

- ■ GET /checkflightstatus/AA1215 HTTP/1.1

  - ◊ Check the flight status for American Airlines flight #1215

- ■ POST /checkflightstatus HTTP/1.1

  - ◊ Upload a new flight status by sending an XML document that conforms to a previously defined XML Schema

⋄ Response is a "201 Created" and a new URI

```
201 Created
Content-Location: /checkflightstatus/AA1215
```

- PUT /checkflightstatus/AA1215 HTTP/1.1

  ⋄ Update an existing resource representation

- DELETE /checkflightstatus/AA1215 HTTP/1.1

  ⋄ Delete the resource representation

11.5 REST vs SOAP

- REST

  ⋄ Ideal for use in Web-centric environments, especially as a part of Web Oriented Architecture (WOA) and Web 2.0

  ⋄ Takes advantage of existing HTTP tools, techniques, & skills

  ⋄ Little standardization and general lack of support regarding enterprise-grade demands (security, transactions, etc.)

- SOAP

  ⋄ Supports robust and standardized security, policy management, addressing, transactions, etc.

  ⋄ Tools and industry best practices for SOA and WS assume SOAP as the message protocol

11.6 REST Services With Spring MVC

- The Spring MVC web framework provides support for implementing REST services

  ⋄ The core of this support is provided by the @RequestMapping annotation within Spring MVC @Controller classes

- Unlike other Spring MVC web applications you do not use "View Handlers" as the return type from a Spring MVC handler method is often used as the body of the response directly

11.7 Spring MVC @RequestMapping with REST

- The primary support for REST services in Spring MVC is the @RequestMapping annotation in a @Controller class

- The annotation can be used at the class level and method level

  ◇ If used both places the settings are combined

- The primary attribute used is the 'value' which becomes the path that the handler method is mapped to

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {
```

- It is also very common in REST services to use the 'method' attribute to map a particular method to a specific HTTP method

```
@RequestMapping(value="/appointments",
      method = RequestMethod.GET)
public AppointmentList getAllAppointments() {
```

- It is also very common to use the @PathVariable annotation to get data from the request URL directly

```
@RequestMapping(value="/owners/{ownerId}",
      method=RequestMethod.GET)
public PetList findAllPets(@PathVariable int ownerId) {
```

11.8 Working With the Request Body and Response Body

- With REST services it is common to have the entire request body processed as incoming data or to return data as the body of the response

- With Spring MVC you can use the @RequestBody and @ResponseBody annotations on method parameters or the method return type

  ◇ The @ResponseBody annotation is easiest to have on the method since a method can only have one return type

  ◇ The classes used as method parameters and return types should be used with JAXB to allow conversion between Java and XML

```
@RequestMapping(method=RequestMethod.POST)
@ResponseBody
```

```
public QuoteResponse getQuote(@RequestBody QuoteRequest
request) {
```

11.9 @RestController Annotation

- Spring 4 added the @RestController annotation

- It combines @Controller, @ResponseBody, @RequestBody

- So now you just need one annotation on a REST controller class.

11.10 Implementing JAX-RS Services and Spring

- JAX-RS is the official Java specification for defining REST services in Java

  ◇ Similar to Spring MVC this is done mainly with JAX-RS annotations although these annotations are different

- Spring MVC is NOT a JAX-RS implementation

- It is possible though to use standard JAX-RS code and implementation to define the REST service itself and simply inject Spring components into the JAX-RS service class

  ◇ This would provide for more "standard" code for the REST service but allow the other features of Spring to be used as well

- The best way to do this would be to use the SpringBeanAutowiringSupport class from within a @PostConstruct method of the JAX-RS class which contains @Autowired Spring components

```
public class QuoteService {
@Autowired
private QuoteGenerator generator;
@PostConstruct
public void initSpringComponents() {
        SpringBeanAutowiringSupport.
```

```
                    processInjectionBasedOnCurrentContext(this);
}
```

Implementing JAX-RS Services and Spring

Since Spring is not a JAX-RS implementation it will not recognize the JAX-RS annotations.

11.11 JAX-RS Annotations

- JAX-RS uses different annotations from Spring MVC

- @Path annotation for linking classes or methods to the request path that will map to them

```
@Path("/quotes")
public class QuoteService {
```

- Separate annotations for mapping to the various HTTP methods

```
@GET   @POST  @PUT   @DELETE
```

- You can use a @PathParam annotation with a method parameter to extract data from the URL

```
@Path("/thisResource/{resourceId}")
public String getResourceById(@PathParam("resourceId")
String id) { … }
```

11.12 Java Clients Using RestTemplate

- The most common types of clients for REST services are JavaScript and AJAX clients

- It is perhaps common also to need to have Java code communicate with REST services as a client

- Although REST service requests and responses are fairly simple using basic Java APIs like the java.net package would be difficult and low-level

- JAX-RS 1.x does not provide a client API although JAX-RS 2.0 will

- Spring provides the RestTemplate class which has a number of convenience methods for sending requests to REST services

  ◇ These REST services do not need to be implemented using Spring MVC or JAX-RS

- To use the RestTemplate class in a Java client you would still need to include the Spring MVC module in the application

11.13 RestTemplate Methods

- There are many different methods and these are generally provided based on the HTTP method that will be sent

  ◇ The parameters and return type differ depending on if the request body will contain data from an object and if the response body will contain data coming back

  ◇ All take a String for URL and a variable argument Object... parameter for parameters in the URL

- DELETE methods are probably the simplest

```
void delete(String url, Object... urlVariables)
```
- GET takes no request body but returns an object for the response body

```
<T> getForObject(String url, Class<T> responseType,
Object... urlVariables)
```
- PUT takes a request body but returns nothing

```
void put(String url, Object request, Object...
urlVariables)
```
- POST takes a request body and can return a response body

```
<T> postForObject(String url, Object request, Class<T>
responseType, Object... uriVariables)
```

11.14 Summary

- Spring MVC provides basic support for implementing REST services

- Spring could also be used behind standard JAX-RS REST services

---

Objectives
Key objectives of this chapter

- Overview of Spring Security

- Configuring Spring Security

- Defining security restrictions for an application

- Customizing Spring Security form login, logout, and HTTPS redirection

- Using various authentication sources for user information

12.1 Securing Web Applications with Spring Security 3.0

- Spring Security (formerly known as Acegi) is a framework extending the traditional JEE Java Authentication and Authorization Service (JAAS)

- It can work by itself on top of any Servlet-based technology

  ◇ It does however continue to use Spring core to configure itself

- It can integrate with many back-end technologies

  ◇ Support for OpenID, CAS, LDAP, Database

- It uses a servlet-filter to control access to all Web requests

- It can also integrate with AOP to filter method access

  ◇ This gives you method-level security without having to actually use EJB

12.2 Spring Security 3.0

- Because it is based on a servlet-filter, it can also work with SOAP based Web Services, RESTful Services, any kind of Web Remoting, and Portlets

- It can even be integrated with non-Spring web frameworks such as Struts, Seam, and ColdFusion

- Single Sign On (SSO) can be integrated through CAS, the Central Authentication Service from JA-SIG

  ◇ This gives us access to authenticate against X.509 Certificates, OpenID (supported by Google, Facebook, Yahoo, and many others), and LDAP

  ◇ WS-Security and WS-Trust are built on top of these

- It can integrate into WebFlow

- There's support for it in SpringSource Tool Suite

Notes

Security is a many-headed problem. In this course we're mostly concerned with request-level authentication and authorization, but there's much more to deal with. Cross-Site Replay attacks, SQL injection, Man-In-The-Middle, Denial-Of-Service and Phishing attacks are just some of the concerns a comprehensive security offering needs to address.

The Spring Security site is:

http://static.springsource.org/spring-security/site/

12.3 Authentication and Authorization

- Authentication answers the question "Who are you?"

  ◇ Includes a User Registry of known user credentials

  ◇ Includes an Authentication Mechanism for comparing the user credentials with the User Registry

  ◇ Spring Security can be configured to authenticate users using various means or to accept the authentication that has been done by an external mechanism

- Authorization answers the question "What can you do?"

  ◇ Once a valid user has been identified, a decision can be made about allowing the user to perform the requested function

- ◇ Spring Security can handle the authorization decision

  - ■ Sometimes this may be very fine-grained. For example, allowing a user to delete their own data but not the data of other users

12.4 Programmatic v Declarative Security

- Programmatic security allows us to make fine grained security decisions but requires writing the security code within our application

  - ◇ The security rules being applied may be obscured by the code being used to enforce them

- Whenever possible, we would prefer to declare the rules for access and have a framework like Spring Security enforce those rules

  - ◇ This allows us to focus on the security rules themselves and not writing the code to implement them

- With Spring Security we have a DSL for security that enables us to declare the kinds of rules we would have had to code before

  - ◇ It also enables us to use EL in our declarations which gives us a lot of flexibility

  - ◇ This can include contextual information like time of access, number of items in a shopping cart, number of previous orders, etc.

12.5 Getting Spring Security from Maven

- Spring 3.0 split many different packages into different modules available from Maven so you can use just what you need

- The following will almost always be used

  - ◇ Core – Core classes

- ◇ Config – XML namespace configuration

- ◇ Web – filters and web-security infrastructure

- ■ The following will be used if the appropriate features are required

- ◇ JSP Taglibs

- ◇ LDAP – LDAP authentication and provisioning

- ◇ ACL – Specialized domain object ACL implementation

- ◇ CAS – Support for JA-SIG.org Central Authentication Support

- ◇ OpenID – 'OpenID for Java' web authentication support

Getting Spring Security from Maven

The exact syntax of how you add the above Spring Security modules using Maven will differ depending on if you get them from:

Maven Central – http://search.maven.org/

SpringSource Enterprise Bundle Repository (EBR) – http://ebr.springsource.com/repository/

The following is an example of getting them from the SpringSource EBR:

```
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>org.springframework.security.core</artifactId>
        <version>3.0.4.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>org.springframework.security.web</artifactId>
        <version>3.0.4.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>org.springframework.security.taglibs</artifactId>
        <version>3.0.4.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>org.springframework.security.config</artifactId>
        <version>3.0.4.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>org.springframework.security.ldap</artifactId>
        <version>3.0.4.RELEASE</version>
</dependency>
```

12.6 Spring Security Configuration

- If Spring Security is on the classpath, then web applications will be setup with "basic" authentication on all HTTP endpoints.

- There is a default AuthenticationManager that has a single user called 'user' with a random password.

  ◇ The password is printed out during application startup

  ◇ Override the password with 'security.user.password' in 'application.properties'.

- To override security settings, define a bean of type 'WebSecurityConfigurerAdapter' and plug it into the configuration

12.7 Spring Security Configuration Example

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class ApplicationSecurity
        extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http)
                throws Exception {

            http.authorizeRequests()
                .antMatchers("/css/**").permitAll().anyRequest()
                .fullyAuthenticated().and().formLogin()
                .loginPage("/login")
                .failureUrl("/login?error")
                .permitAll().and().logout().permitAll();
        }

        @Override
        public void configure(AuthenticationManagerBuilder auth)
            throws Exception {
            auth.inMemoryAuthentication()
               .withUser("user").password("user").roles("USER");
        }

}
```

12.8 Authentication Manager

- The AuthenticationManager class provides user information

  ◇ You can use multiple <authentication-provider> elements and they will be checked in the declared order to authenticate the user

- In the example above, the WebSecurityConfigurerAdapter's 'configure()' method gets called with an AuthenticationManagerBuilder.

- We can use this to configure the AuthenticationManager using a "fluent API"

  ◇ auth.jdbcAuthentication().dataSource(ds).withDefaultSchema()

  ◇ See the Spring Security JavaDocs for more details.

12.9 Using Database User Authentication

- You can obtain user details from tables in a database with the jdbcAuthentication() method

  ◇ This will need a reference to a Spring Data Source bean configuration

  ```
  auth.jdbcAuthentication().dataSource(ds).withDefaultSchema()
  ```

- If you do not want to use the database schema expected by Spring Security you can customize the queries used and map the information in your own database to what Spring Security expects

```
auth.jdbcAuthentication().dataSource(securityDatabase)
        .usersByUsernameQuery("SELECT username, password,
        'true' as enabled FROM member WHERE username=?")
        .authoritiesByUsernameQuery("SELECT
        member.username, member_role.role as authority
        FROM member, member_role WHERE member.username=?
        AND member.id=member_role.member_id");
```

Using Database User Authentication

The configuration of the 'securityDatabase' Data Source above is not shown but it is just like Spring database configuration.

The queries that Spring Security uses by default are:

```
SELECT username, password, enabled FROM users WHERE username = ?
SELECT username, authority FROM authorities WHERE username = ?
```

The default statements above assume a database schema similar to:

```
CREATE TABLE USERS (
        USERNAME VARCHAR(20) NOT NULL,
        PASSWORD VARCHAR(20) NOT NULL,
        ENABLED SMALLINT,
        PRIMARY KEY (USERNAME)
);
CREATE TABLE AUTHORITIES (
        USERNAME VARCHAR(20) NOT NULL,
        AUTHORITY VARCHAR(20) NOT NULL,
        FOREIGN KEY (USERNAME) REFERENCES USERS
);
```

Notice in the custom queries defined in the slide the 'enabled' part of the query is mapped as 'true' since it is assumed the table referenced does not have this column but Spring Security expects it. If the table does have some column similar to 'enabled' it should map to a boolean type (like a '1' for enabled and '0' for disabled).

The custom queries above would work with a database schema of:

```
CREATE TABLE MEMBER (
        ID BIGINT NOT NULL,
        USERNAME VARCHAR(20) NOT NULL,
        PASSWORD VARCHAR(20) NOT NULL,
        PRIMARY KEY (ID)
);
CREATE TABLE MEMBER_ROLE (
        MEMBER_ID BIGINT NOT NULL,
        ROLE VARCHAR(20) NOT NULL,
        FOREIGN KEY (MEMBER_ID) REFERENCES MEMBER
);
```

12.10 LDAP Authentication

- It is common to have an LDAP server that stores user data for an entire organization

- The first step in using this with Spring Security is to configure how Spring Security will connect to the LDAP server with the ldapAuthentication builder

```
auth.ldapAuthentication()
      .contextSource()
      .url("ldap://localhost").port(389)
      .managerDn("cn=Directory Admin")
      .managerPassword("ldap");
```

- You can also use a "embedded" LDAP server in a test environment by not providing the 'url' attribute and instead providing ldif files to load

```
auth.ldapAuthentication()
      .contextSource()
      .url("ldap://localhost").port(389)
      .managerDn("cn=Directory Admin")
      .managerPassword("ldap");
```

LDAP Authentication

The 'manager-dn' and 'manager-password' attributes of <ldap-server> are used for how to authenticate against the LDAP server to query user details.

If using the embedded LDAP server the default for the 'root' will be "dc=springframework,dc=org" if you do not supply a value.

In order to configure Spring Security there are a number of attributes related to LDAP that have various defaults that may affect how your LDAP configuration behaves. This slide is meant to simply introduce the feature. One step you should take when attempting to use Spring Security with LDAP is to avoid configuring everything at once. Start with an embedded list of users to test the other configuration settings and then switch to using LDAP. Also try using the embedded LDAP server with an ldif file exported from your LDAP server with a few sample users.

12.11 Summary

- Spring Security has many features that simplify securing web applications

- Making use of many of these features only requires configuration in a Spring configuration file

- Spring Security can work with many different sources of user and permission information

Chapter 13 - Spring JMS

<table>
<tr><td>Objectives</td></tr>
</table>

Key objectives of this chapter

- Spring JMS

- JmsTemplate

- Async Message Receiver, Message-Driven POJOs(MDPs)

- Transaction management

- Producer Configuration

- Consumer Configuration

- Message-Driven POJO's Async receiver Configuration

13.1 Spring JMS

- Spring provides a Template based solution (JmsTemplate class) to simplify the JMS API code development

- JmsTemplate class can be used for sending messages and synchronously receive messages

- For Asynchronous message receiving, just like JEE MDBS, Spring uses Message-Driven POJOs (MDPs)

- Spring 3.x supports JMS 1.1 API, JMS 1.0.2 is deprecated

- JMS 1.1 provides domain-independent API, that means messages can be sent and received to/from different model Queue or Topic using the same Session object

- For different JMS API, Spring provide two sets of template classes, JmsTemplate102 and JmsTemplate for JMS1.0.2 and JMS1.1 respectively

- This template converts JMSException into org.springframework.jms.JmsException

Spring JMS

org.springframework.jms.support.converter provides a MessageConverter abstraction to convert

between Java objects and JMS messages.

org.springframework.jms.support.destination provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI.

org.springframework.jms.connection provides an implementation of the ConnectionFactory suitable for use in standalone applications. It also contains an implementation of Spring's PlatformTransactionManager for JMS (the cunningly named JmsTransactionManager). This allows for seamless integration of JMS as a transactional resource into Spring's transaction management mechanisms.

13.2 JmsTemplate

- The JmsTemplate class helps to obtain and release the JMS resources

- To send messages, simply call send method which take two parameters, destination and MessageCreator

- MessageCreator object is usually implemented as an anonymous class

- MessageCreator interface has only one method createMessage() to be implemented.

- JmsTemplate objects are thread-safe and keeps the reference to ConnectionFactory

- Define template on the configuration file

```
<bean id="jmsTemplate"
      class="org. springframework. jms.core. JmsTemplate">
      <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

13.3 Connection and Destination

- Spring provided SingleConnectionFactory which is a implementation of JMS ConnectionFactory

- Calls to createConnection() on ConnectionFactory return a Connection object

- JmsTemplate can be configured with a default destination via the property defaultDestination

  ◇ The default destination can be used on JmsTemplate with send and receive operations that do not refer to a specific destination

```
<bean id="jmsTemplate"
    class="org.(sf).jms.core.JmsTemplate">
    …
    <property name="connectionFactory"
        ref="connectionFactory" />
    <property name="defaultDestination"
        ref="orderDestination" />
</bean>
```

Connection to Messaging Server

The CachingConnectionFactory extends the functionality of SingleConnectionFactory and adds the caching of Sessions, MessageProducers, and MessageConsumers.

13.4 JmsTemplate Configuration

- Declare JmsTemplate, ConnectionFactory, and Destinations

  ◇ The ConnectionFactory and Destination may depend on the JMS implementation of the environment

  ◇ Below is shown using ActiveMQ as a JMS implementation in Tomcat

    ■ In a Java EE Environment you wouldlikely use <jee:jndi-lookup> to link Spring beans to JNDI lookups

```
<bean id="connectionFactory" class=
"org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
        value="tcp://localhost:61616" />
</bean>
<bean id="orderDestination"
     class="org.apache.activemq.command.ActiveMQQueue">
     <constructor-arg value="order.queue" />
</bean>
<bean id="jmsTemplate" class="org.sf.jms.core.JmsTemplate">
    <property name="connectionFactory"
        ref="connectionFactory" />
    <property name="defaultDestination"
        ref="orderDestination" />
</bean>
```

JmsTemplate Configuration

The declaration of ConnectionFactory and Destinations to link to JNDI lookups would be something like:

```
<jee:jndi-lookup id="connectionFactory"
        jndi-name="connectionFactory"/>

<jee:jndi-lookup id="orderDestination"
        jndi-name="jms/orderQueue"/>
```

13.5 Transaction Management

- JmsTransactionManager provides support for managing transactions on a single ConnectionFactory

- Add the followings to bean configuration file (see the example below)

   ◊ <tx:annotation-driven/>

   ◊ JmsTransactionManager

- JmsTemplate automatically detects transaction

- JmsTemplate can be used with JtaTransactionManager (XA) for distributed transactions

13.6 Example Transaction Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
    ...>
  <tx:annotation-driven/>
    <bean id="transactionManager" class=
"org.springframework.jms.connection.JmsTransactionManager">
        <property name="connectionFactory">
            <ref bean="connectionFactory" />
        </property>
    </bean>
</beans>
```

13.7 Producer Example

- The following example illustrates how to call the send method on the

JmsTemplate

```
@Component
public class OrderProducer {
   @Autowired private JmsTemplate jmsTemplate;

   public void addOrder(final Order order) {
      jmsTemplate.send(new MessageCreator() {
        public Message createMessage(Session session)
           throws JMSException {
          MapMessage m = session.createMapMessage();
          m.setLong("id", order.getId());
          m.setString("custName", order.getCustName());
          …
          return m;
        }
      });
   }
}
```

Producer Example

Now we can use JmsTemplate to send messages. We do this by calling the send method of JmsTemplate.

The send method takes a MessageCreator that implements the callback portion of the template method pattern. You place your application specific logic in the createMessage method. In the above code notice the use of the anonymous inner class when creating the MessageCreator in the parameters of the 'send' method. You don't need to worry about exception handling, resource management, etc. This is similar to other template techniques used in Spring (e.g. JDBC).

The createMessage method can use the JMS Session argument to perform JMS specific logic. In the example on the slide, we create a MapMessage and build the message using the properties of the order. We then return this message as the one to be sent.

Note that createMessage throws JMSException but we don't have to handle the exception. We leave that to JmsTemplate (which will catch it and re-throw it as a Spring unchecked JmsException).

MessageCreator is in the org.springframework.jms.core package.

13.8 Consumer Example

- Call receive method on the JmsTemplate

```
@Component
public class OrderConsumer {
    @Autowired private JmsTemplate jmsTemplate;

    public Order receiveOrder() {
      MapMessage m = (MapMessage)jmsTemplate.receive();
      try {
        Order order = new Order();
        order.setId(m.getLong("id"));
        order.setCustName(m.getString("custName"));
        …
        return order;
      }
      catch (JMSException e) {
        throw JmsUtils.convertJmsAccessException(e);
      }
    }
}
```

Consumer Configuration

However, when extracting information from the received MapMessage object, you still have to handle the JMS API's JMSException. This is in stark contrast to the default behavior of the framework, where it automatically maps exceptions for you when invoking methods on the JmsTemplate. To make the type of the exception thrown by this method consistent, you have to make a call to JmsUtils.convertJmsAccessException() to convert the JMS API's JMSException into Spring's JmsException.

By default, this template will wait for a JMS message at the destination forever, and the calling thread is blocked in the meantime.

To avoid waiting for a message so long, you should specify a receive timeout for this template. If there's no message available at the destination in the duration, the JMS template's receive() method will return a null message.

If you're expecting a response to something or want to check for messages at an interval, handling the messages and then spinning down until the next interval. If you intend to receive messages and respond to them as a service, you're likely going to want to use the message-driven POJO functionality.

13.9 Converting Messages

- You can use a MessageConverter to encapsulate the logic to convert

messages to domain objects

```
public class OrderConverter implements MessageConverter {
  public Object fromMessage(Message m) {
    MapMessage mm = (MapMessage)m;
    Order order = new Order();
    try {
      order.setId(mm.getLong("id"));
      …
    }
    catch (JMSException e) {
      throw new
        MessageConversionException(e.getMessage());
    }
    return order;
  }
}
```

Converting Messages

The service methods in the examples on the previous slides contain considerable logic to convert domain objects to JMS messages and vice versa. To increase the reusability of this logic, Spring provides the MessageConverter interface.

In the example on the slide, we implement this interface as the OrderConverter class.   It defines the fromMessage and toMessage methods (toMessage is depicted on the next slide).

The fromMessage method takes a JMS Message and converts it to an Object. We have transplanted the logic from the MessageCreator createMessage method to fromMessage.

13.10 Converting Messages

■ And convert domain objects to messages

```
…
public Message toMessage(Object object,
    Session session) throws JMSException {
  Order order = (Order)object;
  MapMessage m = session.createMapMessage();
  m.setLong("id", order.getId());
  m.setString("custName", order.getCustName());

  …
  return m;
```

```
  }
}
```

Converting Messages

The toMessage method takes an Object and converts it to a JMS Message. We have transplanted the logic from the receiveOrder method to toMessage.

13.11 Converting Messages

- Configure JmsTemplate and MessageConverter

```
<bean id="jmsTemplate" class="org…JmsTemplate">
  …
  <property name="messageConverter">
    <ref bean="messageConverter"/>
  </property>
</bean>
<bean id="messageConverter" class="com…OrderConverter"/>
```
- The addOrder method is much simpler now

```
public void addOrder(Order order) {
  jmsTemplate.convertAndSend(order);
}
```
- The receiveOrder method is also simpler

```
public Order receiveOrder() {
  return (Order)jmsTemplate.receiveAndConvert();
}
```

Converting Messages

Configure the JmsTemplate to use the OrderConverter by wiring it into the "messageConverter" property.

Now the service methods are much cleaner. Use the convertAndSend method to convert an object to a JMS message using the currently registered MessageConverter of the JmsTemplate. Use the receiveAndConvert method to receive a message and convert it to an object.

13.12 Message Listener Containers

- Spring provides message-driven POJOs (MDPs) similar to Message-

Driven Bean in EJB container, to receive messages

- A message listener container can be used to receive messages from a JMS message queue and drive the MessageListener that is injected into it.

  ◇ A message listener container is the intermediary between a MDP and a messaging provider

  ◇ This container is responsible for all threading of message reception and dispatches into the listener for processing.

- Spring provides two standard JMS message listener containers

  ◇ SimpleMessageListenerContainer

    - Creates a fixed number of JMS sessions and consumers at startup, registers the listener using the standard JMS MessageConsumer.setMessageListener() method

    - JMS provider should perform listener callbacks

    - Does not support managed Transaction environment, like Java EE

  ◇ DefaultMessageListenerContainer

    - It supports managed Transaction environment, like Java EE

    - Received messages are registered with XA transaction

13.13 Message-Driven POJO's Async Receiver Example

- Create a message listener class using MessageListener interface

```
public class OrderListener implements MessageListener {
    public void onMessage(Message message) {
        MapMessage msg = (MapMessage) message;
        try {
            Orderinfo info = new OrderInfo();
            info. setOrderID(msg.getString("orderID"));
            info.setDate(msg.getString("orderDate"));
            info.setDeliveryDate(msg.
              getDouble("deliveryDate"));
            displayOrderInfo(info);
        } catch (JMSException e) {
            throw JmsUtils.convertJmsAccessException(e);
```

```
        }
    }
    private void displayOrderInfo(Orderinfo info) {
        System. out. println("OrderID" + info.getOrderID()
            " received and to be shipped on "  +
            info.getDeliveryDate);
    }
}
```

13.14 Message-Driven POJO's Async Receiver Configuration

```xml
<bean id="connectionFactory" class=
        "org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
        value="tcp://localhost:61616" />
</bean>
<bean id="orderListener"
        class="com.simple.OrderListener" />
<bean class=
"org.(sf).jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory"
        ref="connectionFactory" />
    <property name="destinationName" value="order.queue" />
    <property name="messageListener" ref="orderListener" />
</bean>
```

Message-Driven POJO's Async Receiver Configuration

There are two types of message listener containers:

- SimpleMessageListenerContainer, does not support transactions

- DefaultMessageListenerContainer, does support transactions

DefaultMessageListenerContainer requires a JMS 1.1 or higher provider since it uses the domain-independent API. Use DefaultMessageListenerContainer102 for JMS 1.0.2 providers.

13.15 Spring Boot Considerations

■ Spring Boot will automatically configure an 'AmqpTemplate' and 'AmqpAdmin' if you use the 'spring-boot-starter-amqp' starter

- Support for RabbitMQ is built-in

- Configure using application properties:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

- To setup a listener, simply annotate a method with '@RabbitListener'

```
@RabbitListener(queues = "someQueue")
public void processMessage(String content) {
    // ...
}
```

13.16 Summary

- JMS support in Spring, how to use Spring to build message-oriented architectures.

- You learned how to configure both producer and consumer messages using a message queue.

- You learned how to build message-driven POJOs.

Chapter 14 - Microservices

Objectives

Key objectives of this chapter:

- Make an attempt to define a *Microservice*

- Provide a side-by-side comparison between *Microservices* and *SOA*

- Provide a quick overview of some of the features of application frameworks for creating microservices

14.1 What is a "Microservice"?

- The term *Microservice* has recently been coined to describe a rapidly provisionable, independently deployable service that is accessible over common communication protocols

- A microservice usually encapsulates narrow and distinct functionality that helps realize bigger applications

- A microservice is accessed through its known API which acts as a published contract for the service consumers

- Microservices collaborate with each other in a decentralized fashion using built-in intelligence for request routing and embedded business logic

- Adoption of microservices is facilitated by the DevOps practices for continuous deployment and integration of software systems

- The common practice is to run services in their own processes

Notes:

Some IT practitioners are critical of the term *Microservice* claiming it to be too fuzzy and misleading. Other such terms, in their opinion, are *NoSQL* and *Big Data,* which are ambiguous at best.

14.2 One Helpful Analogy

- It may help to understand what microservices are by drawing an analogy with Unix command-line tools

- You can build quite sophisticated data processing chains by joining various Unix command-line tools through the piping ('|') mechanism

- For example, the following commands will count the number of lines containing the '#' character in all files with the extension .csv in the working folder:

```
cat *.csv | grep '#' | wc -l
```

- You can think of a microservice as a stand-alone Unix tool that does a small (and useful) job and which lends itself for participating in some form of collaboration with other microservices to produce a solution to a task in a specific problem domain

  ◇ Solutions are created from suites of loosely coupled and cohesive microservices

Notes:

A microservice is not necessarily represented by a single runtime process - it may consist of multiple processes that work in concert and are deployed together.

14.3 SOA - Microservices Relationship

- There is a blurred line between the notion of SOA and microservices

  ◇ In part, this is because there is still no commonly accepted definition of the term "SOA"

- Most IT practitioners tend to believe that microservices are just a <u>subset</u> of SOA, representing a more practical side of SOA

- Some claim that SOA deals with integrating applications across different problem domains whereas each microservice of an application belongs to a single data domain and offer services around this domain

- Some microservices may be infrastructure-related while SOA's primary focus lies on business services

- One job common to SOA and microservices appears to be the one of breaking up big monolithic applications and decomposing them into smaller chunks of functionality served in a distributed fashion

14.4 ESB - Microservices Relationship

■ In many cases, an SOA is implemented on top of some sort of Enterprise Service Bus (ESB)

  ◇ ESB is viewed by many in the SOA space as the backbone for SOA

■ Microservices see an ESB as an "Egregious Spaghetti Box" using which causes more trouble than offers benefits and which functionality (the "smarts") must be factored out and moved to the microservice end-points

14.5 Traditional Monolithic Designs and Their Role

■ Monolithic application design is prevalent in

  ◇ High-performance/critical real-time systems

  ◇ Systems built around intricate transaction semantics

  ◇ Systems where tight coupling and/or monolithic integration is required (due to security considerations, high cost of cross-process interactions, etc.)

■ Monolithic applications are typically packaged and deployed as a single archive (e.g. Java WAR or EAR file)

14.6 Disadvantages of Monoliths

■ Tight coupling of system components

■ Difficult to test

■ Resistance to change

■ Difficult to scale

■ Long term commitment to single technology (potentially, a vendor lock-in situation)

14.7 Moving from a Legacy Monolith

■ Breaking up the monolith design of your application and moving towards a microservices-based design is motivated by the following considerations:

- ◇ Scalability at a service basis

- ◇ Independent evolution of services

- ◇ Support for domain-driven design

- Refactoring a monolith into a microservices-based application is facilitated by the following architecture attributes of the monolith application:

  - ◇ Layered architecture

  - ◇ Separation of concerns

  - ◇ High cohesion and low coupling (via configurable interface / implementation object realization)

14.8 When Moving from a Legacy Monolith

- This transition offers an opportunity to do the following:

  - ◇ Critically examine existing code base; remove *dead code*

  - ◇ Consider whether new patterns might better solve problems

  - ◇ Develop a DevOps policy that accounts for iterative service development

  - ◇ Provide for increased network communications

    - Focus on security (encryption, authentication, authorization) and performance (move UI generation logic to the client)

  - ◇ Perform development methodology review

    - Agile methodologies (e.g. SCRUM, Kanban, XP) are suited to iterative development

  - ◇ Review testing approach (now it can be done on the per-service basis)

  - ◇ Decide on service-level scalability requirements

14.9 The Driving Forces Behind Microservices

- An increased demand for faster development and deployment cycles

  - ◇ Monolithic applications are cumbersome to evolve, they have larger

footprint, and they have to be updated as a whole

- ◇ Microservices introduce system partitioning that facilitates service evolution, faster development and simpler deployment models

- ■ The need for better testability of complex systems

- ■ Better resilience to system failures

- ■ System scalability

  - ◇ Applications designed and built with microservices can be scaled out and up at a finer level

Notes:

Netflix runs its business on Amazon Web Services (AWS) infrastructure and uses microservices architecture style for designing and building their applications. To test their (micro) services, they use a tool called Chaos Monkey that deliberately causes failures in the system to see how resilient the system is to such failures.

14.10 How Can Microservices Help You?

- ■ Development of microservices can be done by smaller teams leveraging agile development practices

- ■ It becomes easier to view and treat applications (composed of a number of interconnected microservices) as distinct products with encapsulated functionality

  - ◇ The product development model (as opposed to the project model) leads to the feeling of "ownership" of the application (your team carries the application production pager 24/7), which contributes to its overall better quality and yields other intangible benefits

- ■ You will get a better alignment between IT and business needs

  - ◇ The product model facilitated by microservices maps better to business needs

  - ◇ Product-oriented microservices are built around and aligned with business capabilities

    - ■ *Business Capability A → { Product X [, Product Y, …]}*

Notes:

A development team working on a microservice is often referred to as a "Two pizza box" team.

14.11 The Microservices Architecture

- The Microservices Architecture refers to a software architecture design style of creating (complex) composite applications made up of functionally orthogonal or complementary processes that are represented by microservices

    ◇ In line with the SOA philosophy, microservices are cohesive and loosely coupled

- Within a Microservices Architecture, microservices work in concert toward a common goal of delivering a business / utility value to end-users

- Microservices operational life-cycles (deployment, upgrade, decommissioning of older versions, scheduled maintenance, etc.) must be carried out in ways that minimize (ideally, eliminate) service disruption

    ◇ This requirement is often realized through the use of lightweight virtualization, or container-based environment

Notes:

"Microservices Architecture" as a new architectural style is regarded by many in the industry as an attempt by some IT practitioners to re-invent SOA and gain new consulting and other such "hype monetization" opportunities.

14.12 Utility Microservices at AWS

- AWS infrastructure is built around utility microservices

    ◇ You can hear claims that AWS is, rather, built using SOA principles

- Consider their Elastic Compute Cloud (EC2) service and the utility microservices around it, including:

    ◇ EC2 Instances (deployed on-demand on virtual machines)

    ◇ Elastic Block Store (virtual persistent disks for extra storage attached to EC2 instances)

- ◇ Amazon Machine Images (a catalog of more than 500 OS (+ software) images to be deployed on virtual machines)

- ◇ Public/private key pair generation infrastructure

- ◇ The Load Balancer service installed in front of a cluster of EC2 instances

- ◇ et al.

- The services are accessible from the Management Console (GUI), command-line client and query API (HTTP-based: SOAP/REST)
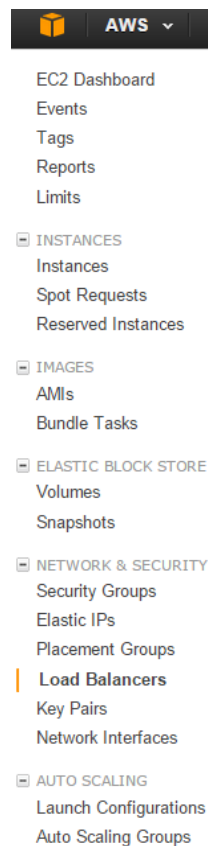
Notes:

Fig. 1 The EC2 Dashboard listing all the supported EC2 services.

14.13 Microservices Inter-connectivity

- Following the inter-connectivity approaches adopted in service-oriented

architectures (SOA), microservices use technology-agnostic protocols

- HTTP has become the transport of choice for microservice inter-connectivity

    ◇ The overall consensus is to use, where appropriate, the REST style of interactions with the emphasis on the statelessness of the communication channels

- Asynchronous communication is carried out in a number of ways:

    ◇ It can be done via lightweight messaging systems, e.g. RabbitMQ, using wire-level protocols, e.g. AMQP, for interoperability

        - Messaging systems used in microservices deal with message routing only, leaving message transformation and other processing tasks to the services themselves

    ◇ Using a number of commonly used techniques for non-blocking remote calls, including AJAX, WebSockets, and other mechanisms of reactive programming

Notes:

"A wire-level protocol is a description of the format of the data that is sent across the network as a stream of octets. Consequently any tool that can create and interpret messages that conform to this data format can interoperate with any other compliant tool irrespective of implementation language."
[source: Wikipedia.org]

14.14 The Data Exchange Interoperability Consideration

- Microservices may be written in different languages (Java, C#, Python, Go, Ruby, etc.)

- To ensure efficient data exchange between microservices inter-connected within complex distributed systems, the following technological approaches are used:

    ◇ Wire-level protocols (AMQP mentioned above) with data encoding / decoding implemented in the respective client

    ◇ Interoperable data interchange formats, such as:

        - Google's protocol buffers [*https://developers.google.com/protocol-*

buffers/]

- Apache Avro [https://avro.apache.org/]

- JSON (JavaScript Object Notation)

- XML

14.15 Managing Microservices

- After a specific number of microservices deployed and running in your software system, their efficient coordination and non-disruptive management may become an issue

- You may be required to establish control over services' lifecycle, their centralized configuration management, logical to physical resource mapping, transparent service versioning, etc.

- Generally, it is a complex, multifaceted activity which depends on your operational environment and the nature of your business

- In some cases, you may satisfy your operational needs by using Apache Zookeeper (*https://zookeeper.apache.org*) which offers services for highly reliable distributed coordination, centralized configuration management , and distributed synchronization

- Netflix Open Source Software (OSS) Center (*https://netflix.github.io/*) provides a complete set of Java-based infrastructure components that can be used to support microservices

- You may also want to consider moving to a whole new deployment and execution platform, e.g. Cloud Foundry (*https://www.cloudfoundry.org/*)

Notes:

Cloud Foundry is a PaaS (Platform as a Service) that you can install on-premise and run over VMware's vSphere virtualization infrastructure.

14.16 Implementing Microservices

- One of a more popular approach for implementing microservices appears to be by means of an application framework that facilitates a fast-track

creation of stand-alone and independently deployable applications that can be rapidly set-up and readily provisioned

- These application frameworks have the following common features:

  ◇ An embedded HTTP web server (Tomcat, Jetty or Undertow) that helps eliminate the need for building and deploying WAR (Web ARchive) or other deployment bundles

  ◇ Built-in production-ready application support in the form of run-time metrics collection, health checks, and externalized configuration

  ◇ et al.

- The above features, in many cases, obviate the need for an external application server by transparently installing and wiring the required system plumbing around the application file(s) (e.g. jar file(s) in the Java world)

14.17 Embedding Databases in Java

- Java software designers and developers are being offered an array of embedded databases that can share the Java Virtual Machine (JVM) with the main application: you are only required to add the database's JAR (Java ARchive) file and optionally specify the location of the database file(s)

- Some of the more popular choices are listed below:

  ◇ **HSQLDB**

    - Used by OpenOffice

    - Gets slower with more data

  ◇ **Apache Derby (a.k.a. Java DB)**

    - Shipped with the Java Development Kit

    - Uses IBM DB2 SQL syntax

    - Client-server deployment mode supports ODBC/CLI, Perl and PHP

    - Regarded as being somewhat slow

  ◇ **H2**

- Claims better performance than the other two database engines

- Can run in embedded or client-server mode

- Supports the PostgreSQL ODBC driver

- Full text search implemented: a) natively; b) via Lucene

Notes:

For more information, visit *http://sayrohan.blogspot.ca/2012/12/choosing-light-weight-java-database.html*

14.18 Microservice-Oriented Application Frameworks and Platforms

- Popular with Java developers are:

  ◇ Spring Boot (*http://projects.spring.io/spring-boot/*)

  ◇ Dropwizard (*http://dropwizard.io/*)

  ◇ Spark (*http://sparkjava.com/*)

    - Do not confuse with Apache Spark!

  ◇ ...with more coming up …

- Popular with open-source community is Node.js (*https://nodejs.org*) used as the platform for building and running microservices

14.19 Summary

- Microservices are a subset of a broader SOA context

- There is a number of motivating forces that drive the adoption of microservices:

  ◇ An increased demand for faster development and deployment cycles

  ◇ The need for better testability of complex systems

  ◇ Resilience to system failures

  ◇ System scalability

- There are a number of application frameworks and platforms that facilitate the creation and deployment of microservices, e.g. Spring Boot, Dropwizard, and Node.js

---

Objectives
Key objectives of this chapter

- ■ The Spring Cloud Configuration Server
- ■ Separation of Configuration from Code
- ■ Configuration Service
- ■ Git Integration
- ■ Configuration Client
- ■ Dynamic Property Updates

15.1 The Spring Cloud Configuration Server

- ■ Externalized configuration support for distributed applications
- ■ Similar to Spring *Environment* and *PropertySource*
- ■ Supports encrypted properties
  - ◇ symmetric and asymmetric
- ■ Consists of:
  - ◇ Server framework
  - ◇ Client API
- ■ Configuration can be stored in Git
  - ◇ Allows for revision control of config files

15.2 Why Configuration Management is Important

- ■ Every application needs to manage configuration information
- ■ Maintaining consistency in configuration is difficult ...
  - ◇ in a distributed application
  - ◇ in a micro services environment

◇ when an application is promoted through different environments (DEV, UAT, PROD)

15.3 Configuration Management Challenges in Microservices

- Larger number of smaller deployment units

- Maintaining consistency across services

- Propagating configuration changes to multiple services

15.4 Separation of Configuration from Code

- Allows code and configuration to be deployed separately

- Simplifies deploying configuration changes

- Centralizes configuration

- Helps insure consistency across distributed services

15.5 Configuration Service

- Is a SpringBoot application

- Uses the **@EnableConfigServer** annotation

```
@EnableConfigServer
@SpringBootApplication
public class CloudConfigServiceApplication {

  public static void main(String[] args) {
    SpringApplication.run(
        CloudConfigServiceApplication.class,
        args);
  }
}
```

Note

Other than call SpringApplication.run() this code doesn't seem to do much.

15.6 How the Configuration Service Works

- The CloudConfigServiceApplication from the previous slide didn't have much in it

- The server portion of the Cloud Config Server uses its own configuration files to find config data and serve it to clients

- Configuration service supports global properties:

  ◇ Shared among SpringBoot applications

  ◇ Stored in a central location

- Configuration service supports application-level properties:

  ◇ Unique to a specific SpringBoot application

15.7 Cloud Config Server Properties File

- Stored in the Cloud Config Server's resources directory

- Named application.properties

  ◇ This file *does not* contain property pairs for other applications

  ◇ This file tells the Cloud Config Server where to find application properties eg:

```
spring.cloud.config.server.git.uri=~/microservices/config
```

Note

The property example above is for properties stored in git and references the repository directory

15.8 Git Integration

- The Cloud Config Server can use Git to store both global and application-specific configuration

- Set up a directory (server-side) and initialize with *git init*

- Create an *application.properties* file for global properties

- Create *{application-name}.properties* files for each supported SpringBoot application

- Add and commit the property files to Git

15.9 Properties

- Standard Java properties files

- Name/Value pairs

- May be updated dynamically

- May be stored in git

- Application-specific properties *override* global properties

15.10 Configuration Client

- Microservice applications will use the Cloud Config application to get configuration information

- Configuration file:
  - ◇ Define application name
  - ◇ Specify Cloud Config Server URI

- SpringBootApplication
  - ◇ Annotate properties with *@Value*

15.11 Sample Client Config File

- The client config info goes in bootstrap.properties so it will be loaded early in the application lifecycle

```
spring.application.name=my-client-app
spring.cloud.config.uri=http://localhost:9999
```

15.12 Sample Client Application

```
@SpringBootApplication
public class ConfigClientApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigClientApplication.class,
args);
  }
}

@RestController
class MessageRestController {

  @Value("${app-specific-propertie:APP}")
  private String appProp;

  @Value("${global-property:GLOBAL}")
  private String globalProp;

  @RequestMapping("/properties")
  String getMessage() {
    return appProp + ":" globalProp;
  }
}
```

Note

This example uses the @Value annotation to map an instance member to a named property. The strings "APP" and "GLOBAL" are default values in case the application cannot access the Cloud Config Service. In this example we did not have to explicitly access the Cloud Config Server. Because we specified its URI in bootstrap.properties, Spring was able to make the request and inject the appropriate property.

15.13 Dynamic Property Updates – Server

- On the server side
  - ◊ Update the property file
  - ◊ Commit the update in git

- ◇ Server detects the commit

- ◇ Server reads and starts serving new/modified properties

15.14 Dynamic Property Update – Client

- Normally the client connects only once to the Config Server

- Add *spring-boot-starter-actuator* to your dependencies to enable client updates

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- Add the @RefreshScope annotation to the class that contains the dynamic properties

15.15 Dynamic Property Update – Execute

- The *spring-boot-starter-actuator* dependency created the */refresh* endpoint in the client application

- Post an empty message to the */refresh* endpoint

```
curl -X POST http://YOUR-HOST:YOUR-PORT/refresh
```
- The client will reload properties from the server

15.16 Summary

In this module we examined:

- The Spring Cloud Configuration Server

- Separation of Configuration from Code

- Configuration Service

- Git Integration

- Configuration Client
- Dynamic Property Updates

<table>
<tr><td>Objectives</td></tr>
<tr><td>

Key objectives of this chapter

- Service Discovery in Microservices

- Netflix Eureka

- Eureka Server

- Eureka Client

- Eureka and the AWS Ecosystem

</td></tr>
</table>

16.1 Service Discovery in Microservices

- In a dynamic environment the collection of service endpoints may change frequently due to:

  ◇ Service failure

  ◇ Auto-scaling

  ◇ Normal service lifecycle (updates, maintenance, retirement)

- At *cloud scale* service discovery must be

  ◇ Highly available

  ◇ Fault tolerant

  ◇ Low latency

16.2 Load Balancing in Microservices

- Microservices are *fine-grained*

- Individual services may be scaled *elastically*

- Traditional load balancers are not well-suited to a microservices environment

 ◇ Configured with known IP addresses and/or hostname

16.3 Netflix Eureka

- Developed by Netflix

- Open source service discovery and load balance solution

    ◇ Internally Netflix wraps Eureka with a proprietary load balance framework that provides more sophisticated load-balancing

- Designed to work in Amazon's AWS cloud

    ◇ May be deployed outside of AWS

- Supports cloud scale applications

- Used internally by microservice applications

    ◇ Not designed for *edge* or client-facing service discovery or load balance

Note

Netflix's internal load balance solution does more than *round-robin* requests. The Netflix solution takes service and network state (and more) into consideration in routing requests.

Even though Amazon and Netflix compete in the streaming video space, Netflix is deployed on Amazon's AWS cloud.

16.4 Eureka Architecture
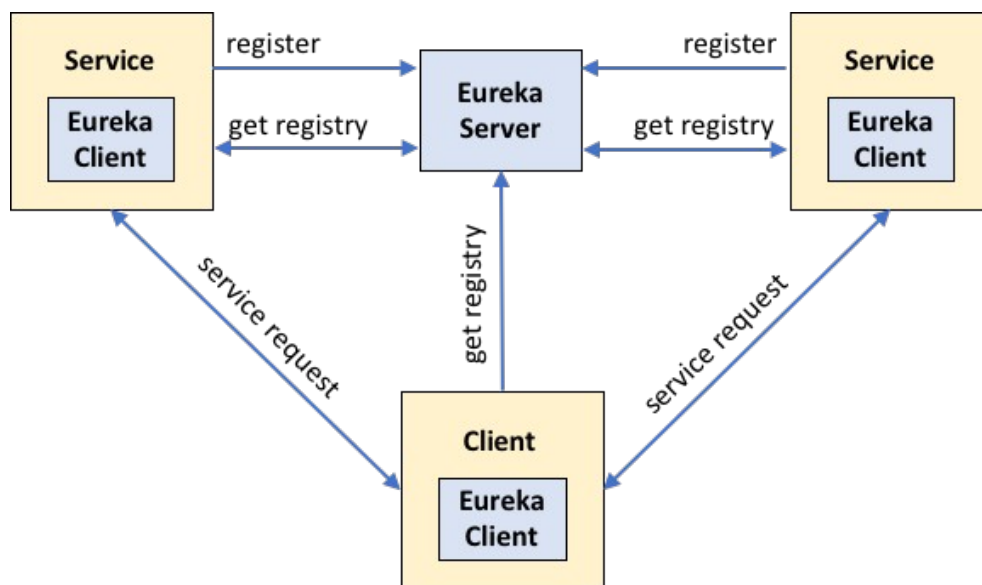
Note

In this drawing the services and the client all get the registry from Eureka. Services may be clients of other services and clients may be services as well.

Not all communications are shown.

16.5 Communications in Eureka

- Register
  - ◇ Clients/Services register with Eureka
- Renew
  - ◇ Heartbeat
  - ◇ Every 30s
  - ◇ Servers that fail to send three consecutive heartbeats (90s) are removed
- Get Registry
  - ◇ Clients/Services get registry info and cache it locally
  - ◇ Registry deltas retrieved at client every 30s
- Cancel
  - ◇ Client informs server to remove it from the registry

16.6 Time Lag

- Heartbeats and Registry updates are sent at a rate of 0.33 Hz (every 30 seconds)
- Changes may take up to 2 minutes to propagate within the system

0:00	Service fails
0:30	Eureka server misses 3 heartbeats
1:30	Server cached registry expires

2:00	Clients receive registry updates

Note

This is the worst case scenario for time lag.

16.7 Eureka Deployment

- Eureka should not be a *single point of failure*

    ◇ Multiple Eureka servers should be deployed in production environments

- Peer Eureka servers communicate the same way that Eureka clients and servers do

- At startup Eureka servers attempt to get current registry info from existing servers

    ◇ If necessary the server attempts to retrieve registry info from multiple peers

- A server that cannot communicate with its peers will try to preserve registry information longer than usual

    ◇ In this case clients may receive outdated registry copies

    ◇ Clients must be able to gracefully handle receiving info for *dead* nodes from Eureka

Note

In case Eureka sends outdated information, clients should set short service request timeouts and failover quickly.

16.8 Peer Communication Failure between Servers

- When a server stops receiving heartbeats from other servers

    ◇ The server enters *self-preservation* mode

    ◇ Preserves current registry state

    ◇ Registry info may be *stale*

- If a service registers with a server in *self-preservation* mode
  - Only clients connected to this server node will acknowledge the new registration
  - Different clients may have different registries
- When peer communication resumes the servers will normalize and synchronize their registries

16.9 Eureka Server Configuration

- SpringBootApplication
- Add the @EnableEurekaServer annotation
- Configure application.properties
  - Specify listen port
  - Disable *normal* eureka client registration
  - Disable *normal* registry fetch

```
server.port=9999
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

16.10 Eureka Client/Service

- SpringBootApplication
- Add the @EnableDiscoveryClient annotation
- Use the @Autowired annotation to inject the DiscoveryClient

```
@EnableDiscoveryClient
@SpringBootApplication
public class EurekaClient {
  public static void main(String[] args{
    SpringApplication.run(
        EurekaClientApplication.class, args);
  }
}
```

```
@RestController
class ServiceInstanceRestController {
  @Autowired
  private DiscoveryClient discoveryClient;
```

16.11 Eureka Client Properties

- Configured in bootstrap.properties
  - ◇ This is the first property file loaded during startup
- Specify application name

```
spring.application.name=foo-client
```

16.12 Spring Cloud DiscoveryClient Interface

- Access to the local copy of the registry
- Methods
  - ◇ String description()
    - Returns the name/description of the implementation
  - ◇ List<ServiceInstance> getInstances(String serviceId)
    - Returns a list of ServiceInstances containing the information to access *each* instance of the specified service
  - ◇ List<String> getServices()
    - Returns a list of service names
    - Names will only be in the list once even if multiple instances are deployed

16.13 ServiceInstance JSON

[{"host":"192.168.10.102","port":8080,"metadata":
{"management.port":"8080"},"uri":"http://
192.168.10.102:8080","secure":false,"serviceId":"FOO-
SERVICE","instanceInfo":{"instanceId":"192.168.10.102:foo-
service:8080","app":"Foo-
SERVICE","appGroupName":null,"ipAddr":"192.168.10.102","sid":"na","ho

```
mePageUrl":"http://192.168.10.102:8080/","statusPageUrl":"http://
192.168.10.102:8080/info","healthCheckUrl":"http://
192.168.10.102:8080/
health","secureHealthCheckUrl":null,"vipAddress":"foo-
service","secureVipAddress":"foo-service", "countryId":1,
"dataCenterInfo"{"@class":"com.netflix.appinfo.InstanceInfo$DefaultDa
taCenterInfo","name":"MyOwn"},"hostName":"192.168.10.102","status":"U
P","leaseInfo":
{"renewalIntervalInSecs":30,"durationInSecs":90,"registrationTimestam
p":1518591759032,"lastRenewalTimestamp":1518592028960,"evictionTimest
amp":0,"serviceUpTimestamp":1518591758520},"isCoordinatingDiscoverySe
rver":false,"metadata":
{"management.port":"8080"},"lastUpdatedTimestamp":1518591759032,"last
DirtyTimestamp":1518591758443,"actionType":"ADDED","asgName":null,"ov
erriddenStatus":"UNKNOWN"}}]
```

Note

This is a lot of information but if you examine it closely you will see some important information

- Directory info: host port URI

- renewalIntervalInSecs: rate at which heartbeats are sent

- durationInSecs: length of time the serve will wait before assuming a service is offline

16.14 ServiceInstance Interface

- ServiceInstance has methods to extract service information

- Information accessed through ServiceInstance is used to access the service

  - Host

  - Port

  - isSecure – HTTPS/TLS

  - URI

  - Metadata

  - etc

16.15 What about Services

- Services are also clients of Eureka and are configured as shown on the precious slides

- In a microservices environment individual services are often clients of other services

16.16 Eureka and the AWS Ecosystem

- Eureka was designed to run in Amazon AWS

- Eureka clusters are deployed within a single AWS *region*

- Multi-region deployment

  - ◇ Each region has its own Eureka cluster

  - ◇ Eureka does not replicate registry information across regions

- Each *zone* within a region should have at least on Eureka server

- In the case of server failure, clients can get registry updates across zones

16.17 Summary

In this module we examined:

- Service Discovery in Microservices

- Netflix Eureka

- Eureka Server

- Eureka Client

- Eureka Service Registration

- Eureka and the AWS Ecosystem

Chapter 17 - Load-Balancing with Netflix Ribbon

Objectives
Key objectives of this chapter

- Load Balancing
- Netflix Ribbon
- Load Balance Rules
- Using Ribbon
- Integration with Eureka (Service Discovery)
- Using Ribbon in the Amazon AWS Cloud

17.1 Load Balancing in Microservices

- Load balancing is a necessary part of any horizontally scalable environment
- Fine-grained elasticity is a key advantage of microservices
  - Load-balancing must be implemented a more fine-grained level
- Must handle dynamic endpoint allocation
- Must handle node failure
- Must be fault tolerant

17.2 Netflix Ribbon

- Load balancer for microservices
- Client-side
- Integrates with Netflix Eureka
- Supports multiple load balance algorithms (rules)
- Monitors target services

17.3 Server-side load balance

- Traditional Load balancers operate on the server-side

    ◇ Appliances (BigIP, Barracuda Load Balancer ADC)

    ◇ Software (Apache, Nginx)

    ◇ Cloud (AWS Elastic Load Balance, Google Cloud HTTP(S) Load Balance

- May be single point of failure

- Load is concentrated in a single node

- Failure of a load-balance node affects many client nodes

17.4 Client-side Load Balance

- Load-balancing distributed throughout the system

- Failure of a node affects smaller portion of the system

- Removes a network *hop*

- Every node does it's own load-balancing

- Ribbon is a client-side load-balance implementation

17.5 Architecture
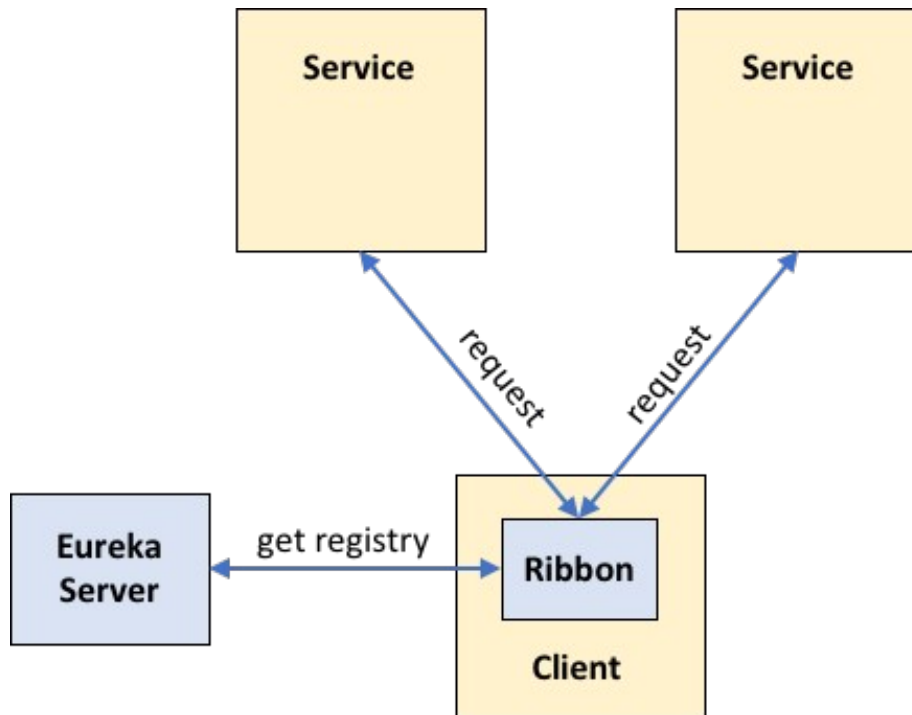
17.6 Load Balance Rules

- Rules are an important part of any load balancer
- Implement IRule interface
- Ribbon offers a useful set of built-in rules
  - ◇ RoundRobinRule
  - ◇ AvailabilityFilteringRule
  - ◇ WeightedResponseTimeRule
  - ◇ RandomRule
  - ◇ ZoneAvoidanceRule

17.7 RoundRobinRule

- Server selection by simple round robin

- Acceptable for many applications

- Does not take service load into account

- Default Rule

17.8 AvailabilityFilteringRule

- Skips servers if:
  - ◇ Circuit is tripped
  - ◇ Concurrent connection count too high

- Circuit tripped
  - ◇ A server is in a *circuit tripped* state after three consecutive connection attempt failures
  - ◇ Circuit tripped state lasts 30 seconds
  - ◇ Each subsequent trip lasts exponentially longer

17.9 WeightedResponseTimeRule

- Servers are weighted by response time

- Requests are routed to servers on a weighted random basis
  - ◇ Servers with the least response time will receive the most requests
  - ◇ All servers will receive some requests

- Server weight is updated dynamically

17.10 RandomRule

- Selects server randomly
  - ◇ Chooses from active servers

- Similar to round-robin

- Most effective when target nodes are configured alike

17.11 ZoneAvoidanceRule

- Designed for Amazon AWS

- Avoids AWS Zones with the most failures

- Applies AvailabilityFilteringRule

17.12 IPing Interface (Failover)

- Allows client-side Ribbon to ping to test if servers are alive

- Multiple implementations

  - NoOpPing – does nothing, assumes server is always available

  - PingURL – HTTP/S connection to service looking for HTTP status 200

17.13 Using Ribbon

In order to use simple Ribbon load balancing (with a static server list)

- Define configuration (YAML file)

- Create configuration class

- Create client class

17.14 YAML Configuration

- Define service name

- Disable Eureka integration (since we are using a static server list)

- Specify server list

```
my-service:
  ribbon:
    eureka:
      enabled: false
    listOfServers: hostA:7001,hostB:7001,hostC:7001
    ServerListRefreshInterval: 15000
```

17.15 Configuration Class

- Override default Ribbon beans (IPing, IRule)

```
public class MyClientConfiguration {

  @Autowired IClientConfig ribbonClientConfig;

  // select IPing implementation
  @Bean public IPing ribbonPing(IClientConfig cfg) {
    return new PingUrl();
  }

  // select IRule implementation
  @Bean public IRule ribbonRule( (IClientConfig cfg) {
    return new AvailabilityFilteringRule();
  }

}
```

Note

These beans are may be overridden in the configuration class:

- IClientConfig
- IRule
- IPing
- ServerList<Server>
- ServerListFilter<Server>
- ILoadBalancer
- ServerListUpdater

17.16 Client Class

- Apply @RibbonClient annotation to class
  - ◇ Specify configuration
- Apply @LoadBalanced annotation to RestTemplate()

- Use RestTemplate and service name to call service via load balancer

17.17 Client Class Implementation

```
@SpringBootApplication @RestController
@RibbonClient( name = "my-service",
    configuration = MyClientConfiguration.class)
public class UserApplication {

  @LoadBalanced @Bean RestTemplate restTemplate(){
    return new RestTemplate();
  }

  @Autowired RestTemplate restTemplate;

  ...

    String result = this.restTemplate.getForObject(
                      "http://my-service/greeting",
                      String.class);
```

17.18 Integration with Eureka (Service Discovery)

- Override ribbonServerList with DiscoveryEnabledNIWServerList
- IPing implementation: NIWSDiscoveryPing
- ServerList implementation: DomainExtractingServerList

17.19 Using Ribbon in the Amazon AWS Cloud

- Ribbon is an open source project from Netflix
- Netflix runs in the Amazon AWS Cloud
- Ribbon can use a *ZoneAffinityServerListFilter*
  - Routes only to servers in a single AWS zone
  - If there are no available servers in the zone routes to available servers

across zones

17.20 Summary

In this module we examined

- Load Balancing
- Netflix Ribbon
- Load Balance Rules
- Using Ribbon
- Integration with Eureka (Service Discovery)
- Using Ribbon in the Amazon AWS Cloud

Chapter 18 - Application Hardening with Netflix Hystrix

Objectives
Key objectives of this chapter

- Netflix Hystrix
- Patterns
- Circuit Breaker Configuration
- Fallback Configuration
- Collapser Configuration
- The Monitor

18.1 Netflix Hystrix

- Hystrix was designed by Netflix to run in the Amazon AWS Cloud
- Designed to :
  - Handle latency across service
  - Handle failures across services
  - Stop cascading failures
  - Provide fallbacks
  - Improve system resiliency
- Hystrix is named after a kind of endangered porcupine

18.2 Design Principles

These are Netflix's published design principles for Hystrix

- Preventing any single dependency from using up all container (such as Tomcat) user threads.

- Shedding load and failing fast instead of queueing.

- Providing fallbacks wherever feasible to protect users from failure.

- Using isolation techniques (such as bulkhead, swimlane, and circuit breaker patterns) to limit the impact of any one dependency.

- Optimizing for time-to-discovery through near real-time metrics, monitoring, and alerting

18.3 Design Principles (continued)

- Optimizing for time-to-recovery by means of low latency propagation of configuration changes and support for dynamic property changes in most aspects of Hystrix, which allows you to make real-time operational modifications with low latency feedback loops.

- Protecting against failures in the entire dependency client execution, not just in the network traffic.

18.4 Cascading Failures

- Failure of a single service causes failure of upstream or downstream services

- Hystrix uses the following to prevent cascading failures:
  - ◇ Bulkhead pattern
  - ◇ Circuit Breaker pattern
  - ◇ Thread Pooling

18.5 Bulkhead Pattern

Problem(s)

- Failure of a service affects all consumers of that service

- Failure of a service may affect a client's ability to connect to other services

- A client may make multiple request to a service, exhausting resources and causing other clients to fail

Solution(s)

- Partition services into groups for different clients

- Assign connection pools to clients for each service

Benefit(s)

- Isolate services (and clients to prevent cascading failures)

- System provides partial functionality in the event of service failure

- Different bulkheads may offer services with different QOS

Note

Bulkhead is a nautical term. In this case the pattern is named after the dividers in a ship that create watertight compartments so that a failure or leak in one compartment will not flood the entire ship.

18.6 Circuit Breaker Pattern

Problem(s)

- Clients may continue to attempt to access a service after the service has failed

- Clients may be unaware of a service failure until a request fails by timeout

- Clients accessing a failed server should be able to recover quickly and gracefully

Solution(s)

- Route client requests to servers with the highest probability of success

- Avoid recently failed servers

- Avoid servers under high load

Benefit(s)

- Fewer request get routed to failed services

- Failures may be detected earlier (not have to wait for back-end service)

Note

Circuit breaker is an electrical term. An electrical circuit breaker is *tripped* when the load on a circuit is too high. It is a defensive mechanism to prevent the circuit from physically overheating and causing a fire or worse.

18.7 Thread Pooling

- Rather than create new threads for every request
  - ◇ Threads are allocated from a pool
  - ◇ Prevents thread starvation
  - ◇ Prevents CPU over-utilization

18.8 Request Caching

- Reduces calls from Hystrix to backend service
  - ◇ Improves performance
- In memory cache
-

18.9 Request Collapsing

- Multiple requests are made for a *non-cached* service
- Hystrix only makes one request to the service
  - ◇ Other requests wait until the single request is made and cached
  - ◇ Reduces the overhead of multiple calls
  - ◇ Reduces load on target service

18.10 Fail-Fast

- Some requests should fail quickly and return
  - ◇ No retires

- ◇ No fallback or alternatives
- ■ A Circuit Breaker Open Circuit fails fast
- ■ Example
  - ◇ Writes when a database is not available
  - ◇ Time-critical reads

18.11 Fallback

- ■ Some requests can and should be retried in case of failure
- ■ When a service is down requests can fallback to an in-memory cache
- ■ Example
  - ◇ Read when a database is not available

18.12 Using Hystrix

- ■ Ribbon and Zuul use Hystrix
- ■ Applications can use Hystrix without using Ribbon or Zuul
- ■ Hystrix provides a monitoring console
  - ◇ Works with Turbine and Eureka to identify and aggregate service data

18.13 Circuit Breaker Configuration

- ■ **@HystrixCommand** annotation enables Hystrix Circuit Breaker
  - ◇ Synchronous methods
  - ◇ Asynchronous methods may return **Future<T>** or **Observable<T>**
- ■ Annotation properties configure Short Circuit behavior
  - ◇ Failure threshold (%) to trip circuit,
  - ◇ Length of time to ignore open circuit

◇ ...

```
@HystrixCommand
@RequestMapping(value="cities/{zip}")
public String getCity() {
  String city = // query database for city by zip
   return city;
}
```

18.14 Fallback Configuration

- Fallback is related to Circuit Breakers

- When a Circuit Breaker is open the operation can either:

  ◇ Fail Fast: return immediately

  ◇ Fallback: attempt to satisfy request using another method

- Fallback is a property of the **@HystrixCommand**

```
@HystrixCommand(fallback="getCachedCity")
@RequestMapping(value="cities/{zip}")
public String getCity() {
   ...
}

// invoked as fallback if getCity() fails
@HystrixCommand
public String getCachedCity() {
   ...
}
```

Note

Spring will call **getCity()** for requests like this one: h**ttp://host:port/cities/90210**. If requests start failing and the Circuit Breaker trips open, then requests will be handled by **getCachedCity()**.

18.15 Collapser Configuration

- Define a handler method in the RestController

- Create a *collapser* service

  ◦ Define a simple *dummy method* to be called by the handler method

  ◦ Define a method to do aggregation

18.16 Rest Controller and Handler

```
@RestController
class OrderHistoryController {

  @Autowired
  private OrderHistoryCollapserService ohcService;

  @RequestMapping(value = "/order-history/{account}")
  public MessageWrapper getHistory(@PathVariable int account)
        throws ExecutionException, InterruptedException {
    return ochService.getHistory(account).get();
  }
}
```

Note

The handler method calls the *dummy* method of the service. On the next slide you will see that the *dummy* method returns a Future<MessageWrapper> object. The handler returns the result of the Future object's *get()* method.

18.17 Collapser Service (Part 1)

```
@Service
public class OrderHistoryCollapserService {

 // dummy method
 @HystrixCollapser(
   scope=com.netflix.hystrix.HystrixCollapser.Scope.GLOBAL,
   batchMethod="getHistories")
  public Future<MessageWrapper> getHistory(Integer account) {
    throw new RuntimeException("Should not be invoked");
  }

  // aggregator method
```

```
@HystrixCommand
public List<MessageWrapper> getHistories(
                                List<Integer> accounts) {
    List<MessageWrapper> orderHistories =
                                new ArrayList<>(accounts.size());

    // make calls to populate orderHistories

    return orderHistories;
    }
}
```

Note

The *dummy* method need the **@HystrixCollapser** annotation. It specifies that the batch or aggregation method id **getHistories()**. Notice that the dummy method throws an exception if it ever gets invoked.

The aggregator method, **getHistories** takes a List of accounts and requires the **@HistirxCommand** annotation.

18.18 How the Collapser Works

- Multiple RESTful GET requests to **/order-history/{account}**

- Hystrix queues up the calls

- When the time threshold is reached the queued account number are passed to the aggregator method

- The aggregator method then makes a single coarse grained request to fulfill *all* the individual RESTful requests

    ◇ The coarse-grained request doesn't have to be a Web service it could be a SQL query

- To callers making RESTful requests the collapser is invisible

18.19 Hystrix Monitor

- Spring Boot web application

- **@EnableHystrixDashboard**

■ Monitors circuits

```
@EnableHystrixDashboard
@SpringBootApplication
public class DashboardApplication {
  public static void main(String[] argv){
    SpringApplication.run(
            DashboardApplication.class,
            argv);
  }
}
```

18.20 Enable Monitoring

■ Every Hystrix enabled app streams data

http://app-host/hystrix.stream

■ Start

http://hystrix-host/hystrix/monitor?stream=http://app-host:port/hystrix.stream

■ Using this mechanism Hystrix can only monitor one service

■ In the next slide we will introduce Turbine

18.21 Turbine

■ Turbine is an aggregator for Hystrix stream data

  ◇ Discovers Hystrix services

  ◇ Turbine streams aggregate data at http://turbine-host/hystrix.stream

■ Use the **@EnableTurbine** annotation

```
@SpringBootApplication
@EnableTurbine
public class TurbineApplication {
  public static void main(String[] args) {
      SpringApplication.run(TurbineApplication.class,
      args);
  }
}
```
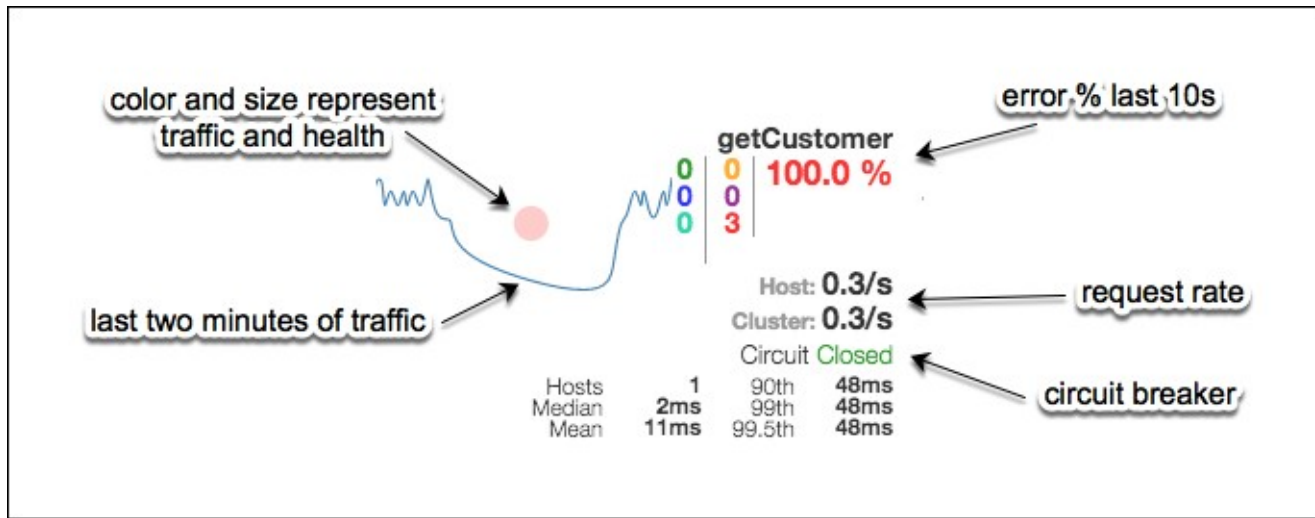
■   Invoke the Hystrix monitor at:

http://hystrix-host/hystrix/monitor?stream=http://turbine-host/hystrix.stream

18.22 The Monitor
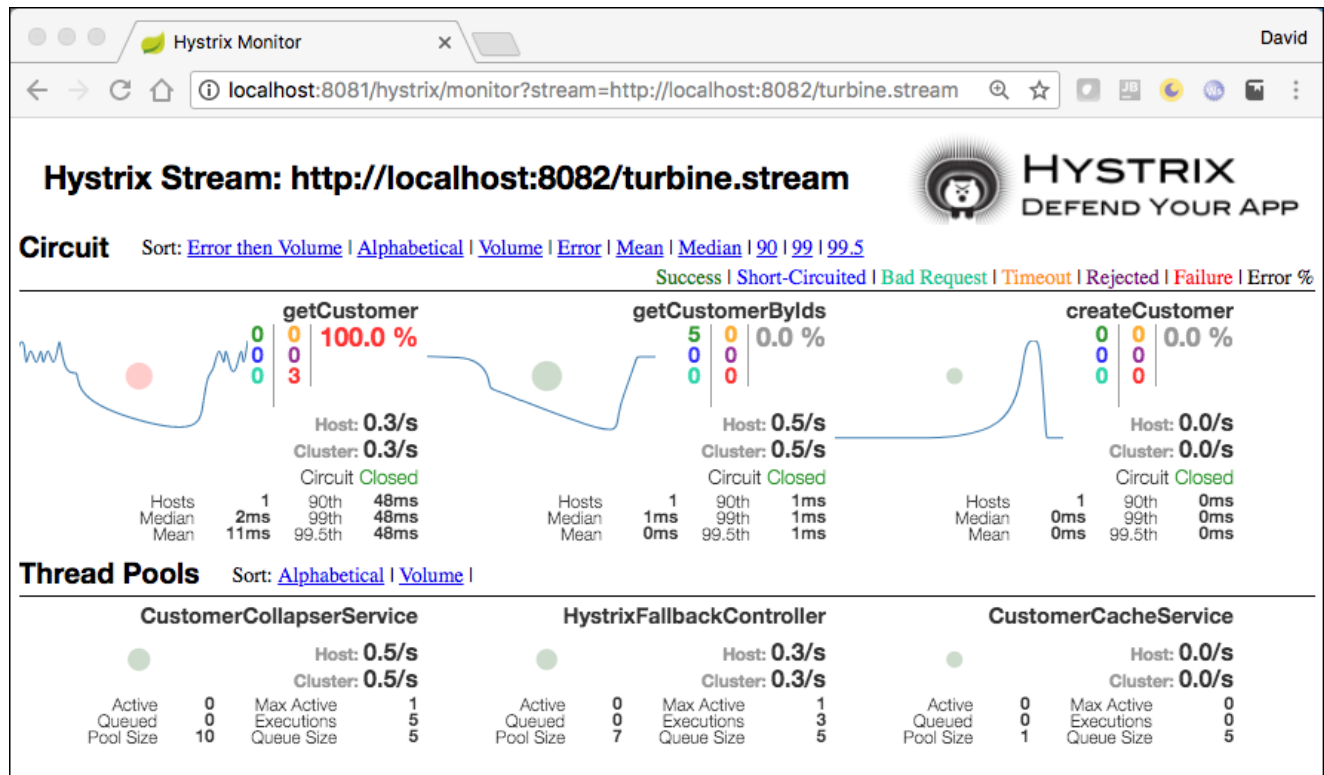
18.23   Monitor details

18.24   Summary

In this module we examined

- Netflix Hystrix
- Patterns
- Circuit Breaker Configuration
- Fallback Configuration
- Collapser Configuration
- The Monitor

Chapter 19 - Edge Components with Netflix Zuul

Objectives
Key objectives of this chapter

- Zuul
- Filters
- Filter Communications
- Zuul Routing with Eureka and Ribbon

19.1 Zuul is the Gatekeeper

- Ghostbusters
- Created in 2012
- Open Source project from Netflix
- Zuul is a collection of filters
  ◇ Written in Groovy so it can be dynamically re-compiled
- At Netflix Zuul is deployed just inside AWS Elastic Load Balancer

Note

According to the *Ghostbusters Wiki, "*Zuul the Gatekeeper of Gozer is a demigod and minion of Gozer, The Destructor, alongside Vinz Clortho the Keymaster."

19.2 Request Handling

- All requests go through Zuul
- Ideal location to perform:
  ◇ Authentication
  ◇ Logging
  ◇ Routing
  ◇ Load-balancing

◇ Deep Insights

19.3 Filters

■ Handle lifecycle events as requests are processed by Zuul

■ Request lifecycle

◇ Pre-routing

◇ Routing

◇ Post-routing

◇ Error

19.4 Filter Architecture
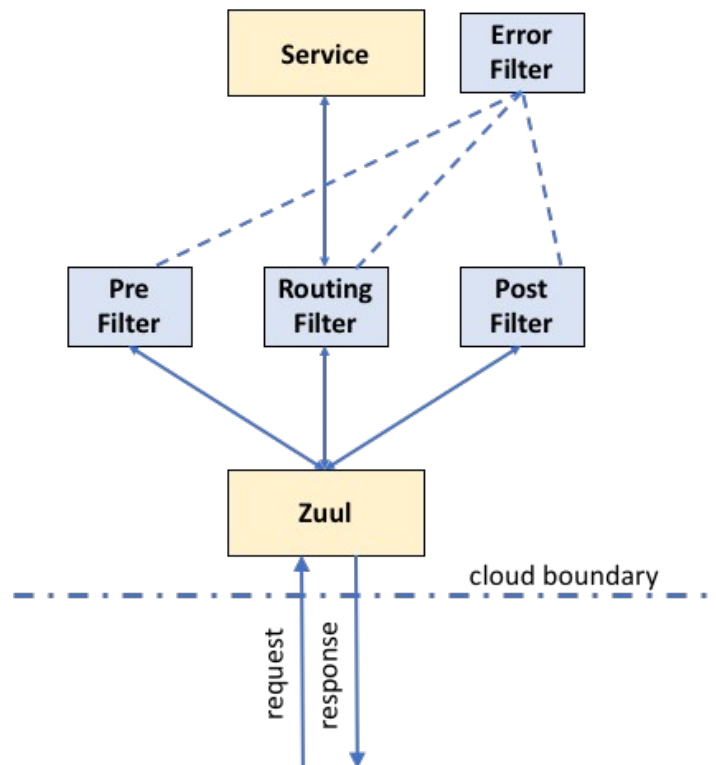
19.5 Filter Properties

■ Key Filter properties

- Filter Type
  - ◇ Defines filter lifecycle stage
- Filter Order
  - ◇ Execution order among filters within the lifecycle stage
- Should Filter
  - ◇ Conditions necessary for filter execution
- Run
  - ◇ Operation to be performed

19.6 filterType()

- Defines lifecycle stage at which filter is applied
- Returns String
- Values
  - ◇ "pre"
  - ◇ "routing"
  - ◇ "post"
  - ◇ "error"

```
@Override
public String filterType() {
  return "pre";
}
```

19.7 filterOrder()

- Defines execution order among filters of the same type
- Returns int
  - ◇ lowest number runs first
  - ◇ equal number, order is indeterminate

```
@Override
```

```
public int filterOrder() {
  return 2;
}
```

19.8 shouldFilter()

- Determines whether filter should be run

- Returns boolean

  - True if filter should run

  - False if filter should not run

```
@Override
public boolean shouldFilter() {
  return true;
}
```

19.9 Run()

- Does the work of the filter

- Returns Object

  - Current API ignores this value

  - Filter should return *null*

```
@Override
public Object run() {
  RequestContext ctx =RequestContext.getCurrentContext();
  HttpServletRequest request = ctx.getRequest();
  log.info(request.getRequestURI());
  return null;
}
```

19.10 Cancel Request

- Filters can short-circuit requests

  - Cancel a request in-flight

- In the **run()** method

```
RequestContext ctx = getCurrentContext();

// cancel the request
ctx.unset();

// set the HTTP status code
ctx.setResponseStatusCode(
        HttpStatus.UNAUTHORIZED.value());
```

19.11 Dynamic Filter Loading

- Filters can be written in Groovy

- Groovy filters may be loaded into Zuul at runtime

  - Behave just like Java filters

19.12 Filter Communications

- Filters cannot call each other

  - Filters are invoked directly by Zuul

- Filters communicate using the **RequestContext**

  - Data can be passed in the context

```
RequestContext ctx = RequestContext.getCurrentContext();
ctx.set("TIMEZONE", "UTC+1");
```

19.13 Routing with Eureka and Ribbon

- Zuul can be configured route and load balance with Eureka and Ribbon

- New service and service instances are discovered by Eureka-service

- Zuul routes to services by *service-name*

  - Using Ribbon

- Routing uses Hystrix and avoids failed servers

- Enable **ribbon** in **application.properties**

```
ribbon.eureka.enabled=true
```

19.14 Summary

In this module we examined:

- Zuul
- Filters
- Filter Communications
- Zuul Routing with Eureka and Ribbon

Chapter 20 - Distributed Tracing with Zipkin

Objectives
Key objectives of this chapter

- ■ Zipkin
- ■ Architecture
- ■ Zipkin in Spring Boot
- ■ GUI Console

20.1 Zipkin

- ■ Open sourced by Twitter in 2012
- ■ Distributed tracing for microservices
  - ◇ Focused on performance data
- ■ Based on the Google *Dapper* paper (2010)

Note

The name Zipkin is Azerbaijani for 'Harpoon.' As in a weapon to kill 'fail whales.'

20.2 Zipkin Features

- ■ Service Dependency Diagrams
- ■ Detailed timing information for individual service calls
- ■ Searching/Filtering of trace information
  - ◇ GUI
  - ◇ API

20.3 Architecture

- ■ Zipkin consists of 4 major components
  - ◇ The Collector

- ◇ Storage

- ◇ API

- ◇ GUI Console

20.4 The Collector

- Instrumented services are called Reporters in Zipkin

- Reporters send information to the Collector

- The Collector initially processes this information and sends it to Storage

  - ◇ Validation

  - ◇ Indexing

20.5 Storage

- Trace information can be stored:

  - ◇ In-memory (default)

  - ◇ JDBC (MySQL/MariaDB)

  - ◇ Cassandra

  - ◇ Elastisearch

Note

In-memory storage is not persistent and does not scale sufficiently to support production environments.

20.6 API

- Zipkin Query Service

- RESTful API for accessing trace information form Storage

  - ◇ JSON format

- The GUI Console is built on this API

20.7   GUI Console

- Open in web browser (default port: 9411)

- Examine traces

  - ◇ 'Timeline' view gives detailed timing information

- View Dependencies

20.8   Zipkin Console Homepage

- Search Criteria

  - ◇ Service name

  - ◇ Start/End Time

  - ◇ Duration

  - ◇ Annotations

20.9 View a Trace

Note

In this trace you can see that the total duration for the *paula-nancy-millstone-service* is 614.760ms. Breaking it down further you see that this service called (serially) *derk-service*, *ford-service*, and *arthur-service*. The *arthur-service* then called (also serially) *marvin-service* and *trillian-service*. From this trace it would be worth examining the code to determine whether some of the service calls could be made concurrently.

20.10 Trace Details

derk-service.http:/derk-service/fire-and-ice/first-line: 280.000ms ✕

AKA: paula-nancy-millstone-service,derk-service

Date Time	Relative Time	Annotation	Address
2/25/2018, 10:45:23 PM	35.000ms	Client Send	192.168.43.228:9006 (paula-nancy-millstone-service)
2/25/2018, 10:45:23 PM	205.000ms	Server Receive	192.168.43.228:9002 (derk-service)
2/25/2018, 10:45:23 PM	309.622ms	Server Send	192.168.43.228:9002 (derk-service)
2/25/2018, 10:45:23 PM	315.000ms	Client Receive	192.168.43.228:9006 (paula-nancy-millstone-service)

| Key	Value
http.host	localhost
http.method	GET
http.path	/derk-service/fire-and-ice/first-line
http.url	http://localhost:8080/derk-service/fire-and-ice/first-line
mvc.controller.class	PoemService
mvc.controller.method	getFirstLine
spring.instance_id	hal:paula-nancy-millstone-service:9006
spring.instance_id	hal:derk-service:9002

More Info

Note

The trace details breaks down the timing information and displays detail about the request and the particular server instance that handled it.
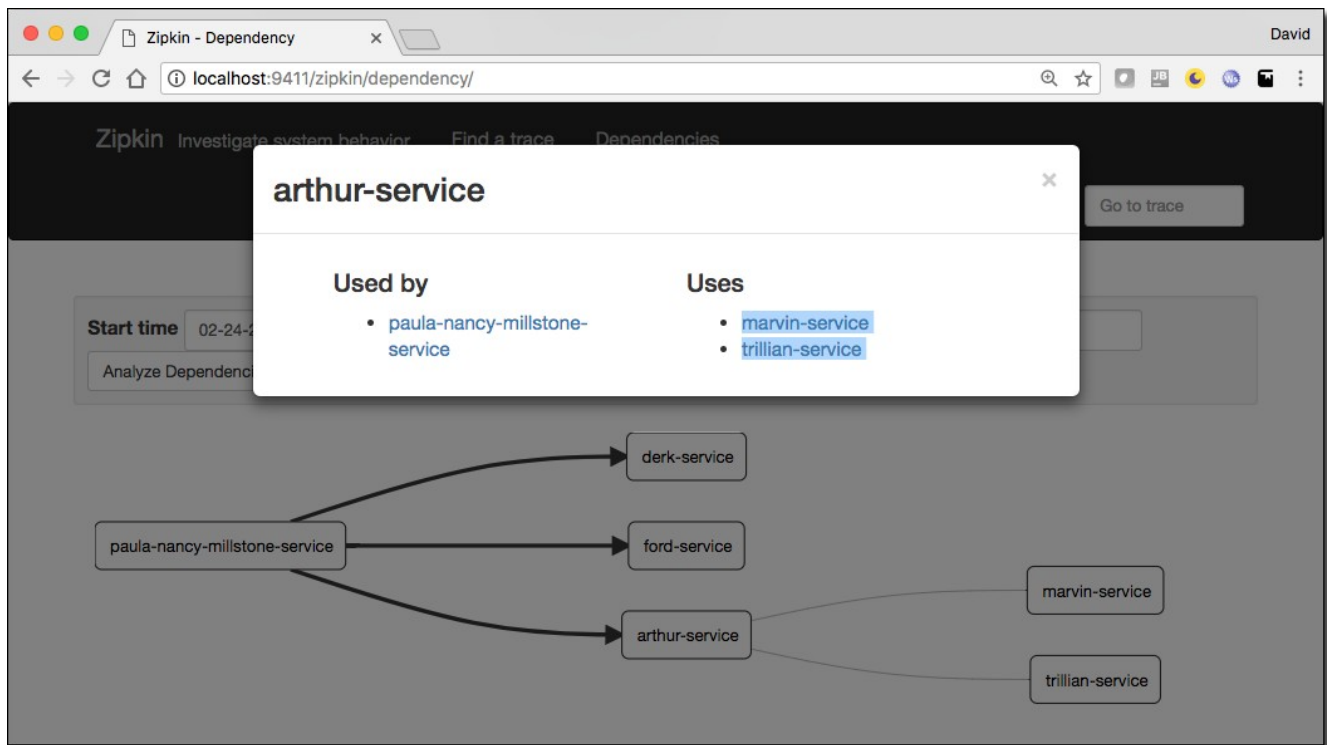
20.11 Dependencies

■ The Console displays a dependency graph
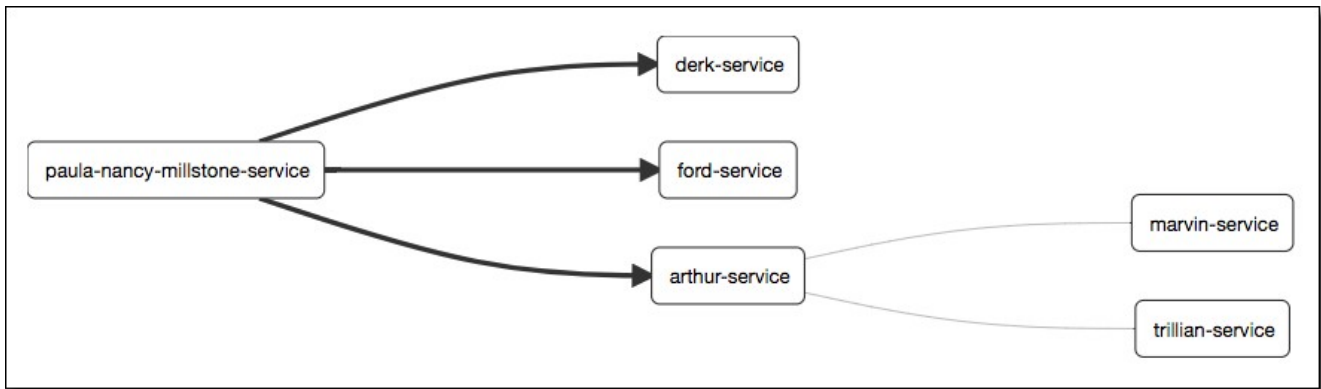
20.12 Dependency Details

■ Clicking on a service brings up a text version of the *upstream* and *downstream* dependencies

Note

This is a *very* simple dependency graph. The details view is helpful for complex dependencies.

20.13 Zipkin in Spring Boot

- Zipkin is available as a Spring Boot application

- Integrates with Eureka for service discovery

- Dependencies

  - Eureka (client)

  - Zipkin

- Time lag

  - Service discovery in Zipkin may take up to two minutes

20.14 Zipkin Configuration

```java
// ZipkinService.java
@EnableDiscoveryClient
@EnableZipkinServer
@SpringBootApplication
public class ZipkinService {
  public static void main(String[] args) {
    SpringApplication.run(ZipkinService.class, args);
  }
}
```

20.15 Summary

In this module we examined

- Zipkin

- Architecture

- Zipkin in Spring Boot

- GUI Console