Spring Framework

Name : Ameya Joshi
Email : ameya.joshi@vinsys.com
Cell No. : +91 98 506 76160

Agenda

* What is Spring?
* The Spring Way
* Spring Framework
* Spring FrameWork Modules
* Spring IOC
* XML based Spring configuration
* Dependency Injection
* Dealing with properties
* Spring Collections Support
* Bean Wiring and Autowiring

Agenda

* Bean Scopes
* Annotation Based Spring Configuration
* Stereotype Annotations
* Java Based Configuration using Annotations
* Spring AOP
* AOP terms
* AOP Advices
* AOP Using ProxyFactory
* AOP with XML
* AOP with AspectJ annotations

Agenda

* Spring JDBC and Transactions Support
* DAOSupport
* DAO Templates
* TransactionManages
* Spring MVC
* Concepts
* Configuring MVC
* MVC Annotations
* Writing Controllers
* RequestMapping

Agenda

* ModelMap
* ModelAndView
* Spring REST Support
* HTTP Messages
* RequestBody
* ResponseBody
* HttpMessageConverters
* ResponseStauts
* Dealing with Exceptions
* RestTemplate

What is Spring?

* Lightweight container framework
  * Lightweight - minimally invasive
  * Container - manages app component lifecycle
  * Framework - basis for enterprise Java appsOpen source
* Apache licensed

The Spring Way

* Spring's overall theme:
* Simplify Enterprise Java Development
  * Loose coupling with dependency injection
  * Declarative programming with AOP
  * Boilerplate reduction with templates
  * Minimally invasive and POJO-oriented

Spring framework

* Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.
* Spring handles the infrastructure so you can focus on your application.
* Spring enables you to build applications from

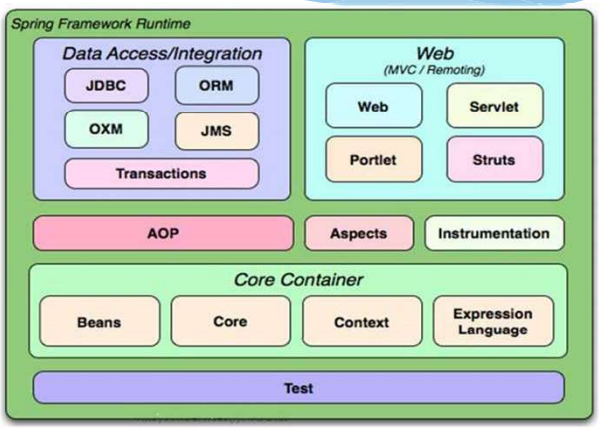
  "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs.

Spring Framework

* Examples of how you, as an application developer, can use the Spring platform advantage:
  * Make a Java method execute in a database transaction without having to deal with transaction APIs.
  * Make a local Java method a remote procedure without having to deal with remote APIs.
  * Make a local Java method a management operation without having to deal with JMX APIs.
  * Make a local Java method a message handler without having to deal with JMS APIs.

Spring Framework modules

* Spring IOC container(core container)
* Spring AOP
* Spring DAO
* Spring ORM
* JEE Module
* Spring web MVC



Spring Core Container

* The Spring core container provides an implementation for IOC (Inversion of control) supporting dependency injection.
* IOC is an architectural pattern that describes to have an external entity to perform Dependency injection (DI) at creation time, that is, object creation time.
* Dependency Injection is a process of injecting/pushing the dependencies into an object.

Spring AOP

* Spring AOP module provides an implementation of AOP (Aspect Oriented Programming).
* Spring is a proxy-based framework implemented in Java.
* AOP integrates the concerns dynamically into the system.
* Concerns: A part of the system divided based on the functionality (transaction, security, logging, caching).

DAO

* Data Access Object(DAO) is a design pattern the describes to separate the persistence logic from the business logic.
* The Spring DAO includes a support for the common infrastructures required in creating the DAO.
* Spring includes DAO support classes to take this responsibility such as:
  * Jdbc DAO support
  * Hibernate DAO support
  * JPA DAO support

ORM

* The Object/Relation Mapping(ORM) module of Spring framework provides a high level abstraction for well accepted object-relational mapping API's such as Hibernate, JPA, OJB and iBatis.
* The Spring ORM module is not a replacement or a competition for any of the existing ORM's, instead it is designed to reduce the complexity by avoiding the boilerplate code from the application in using ORMs.

JEE

* The JEE module of Spring framework is build on the solid base provided by the core package.
* This provides a support for using the remoting services in a simplified manner.
* This supports to build POJOs and expose them as remote objects without worrying about the specific remoting technology given rules.

Web

* This part of Spring framework implements the infrastructure that is required for creating web based MVC application in java.
* The Spring Web MVC infrastructure is built on top of the Servlet API, so that it can be integrated into any Java Web Application server.
* This uses the Spring IOC container to access the various framework and application objects.

Spring IOC

What is IOC ?

* IOC is also known as dependency injection (DI).
* Dependency Injection is the act of injecting dependencies into an Object.
* Inversion of Control is the general style of using Dependency injection to wire together application layers.
* Hence Spring is an Inversion of Control container. That is, it is a container that handles Dependency Injection for you.

Configuration Metadata

* XML-based configuration metadata.
* Annotation-based configuration
* Java-based configuration

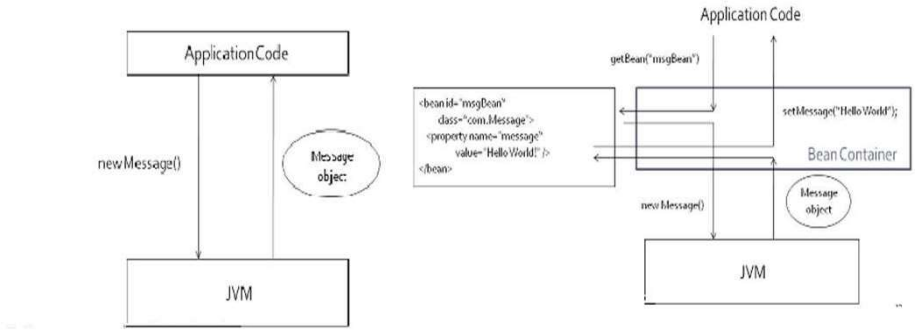
XML-based configuration

```
<beans>
        <bean id="msgBean" class="com.ameya.Message">
                <property name="message"
                value=" Hello World ! "/>
        </bean>
</beans>
```

* The string " Hello World " is injected to the bean msgBean

Understanding Bean Container

- Without a bean container
- With a bean container



Bootstrapping the IOC container

- To start an app using IOC :
  - Create an ApplicationContext object and tell it where applicationContext.xml is.
  - ApplicationContext appContext = newClassPathXmlApplicationContext("applicationContext.xml");
  - This just has to be done once on startup, and can be done in the main method or whatever code bootstraps the application.

Dependency Injection Types

* Setter Injection
* Constructor Injection

Create the Account.java:

```
package com.ameya.models;
public class Account {
private String firstName, lastName;
private double balance;
private Address address;
public Account(){
address = new Address();
}
public Account(String firstName, String lastName, double balance) {
this.firstName = firstName;
this.lastName = lastName;
this.balance = balance;
this.address = new Address();
}
public Account(String firstName, String lastName, double balance, Address address)
{this.firstName=firstName;
this.lastName=lastName;
this.balance = balance;
this.address = address;
}
//Setter & Getter
}
```

Create the Address.java:

```
package com.ameya.models;
public class Address {
private String building, street, city, state, country;
public Address() {}
public Address(String building, String street, String city, String state,
String country) {this.building = building;
this.street = street;
this.city = city;
this.state = state;
this.country = country;
}
//Setter & Getter
}
```

Constructor-based dependency Injection

```
<beans....>
        <bean id="account"
        class="com.ameya.models.Account">
                <constructor-arg value="Ameya"/>
                <constructor-arg value="10000"/>
                <constructor-arg value="Joshi"/>
        </bean>
</beans>
```

```
<beans ...>
        <bean id="account3" class=
        "com.ameya.models.Account">
                <constructor-arg type="java.lang.String"
                value="Ameya"/>
                <constructor-arg type="double"
                value="10000"/>
                <constructor-arg type="java.lang.String"
                value=”Joshi”/>
</bean>
```

```
<beans… .>
        <bean id="account4"
        class="com.ameya.models.Account">
                <constructor-arg value="Ameya"
                index="0"/>
                <constructor-arg value="Joshi"
                index="1"/>
                <constructor-arg value="10000"
                index="2"/>
        </bean>
</beans>
```

```
<beans… >
        <bean id="account5"
        class="com.ameya.models.Account">
                <constructor-arg name="firstName"
                value="Ameya"/>
                <constructor-arg name="lastName" value=
                "Joshi"/>
                <constructor-arg name="balance" value=
                "lOOOO"/>
        </bean>
</beans>
```

Setter-based dependency Injection

```
<bean id="account6" class="com.ameya.models.Account">
        <property name="firstName" value="Ameya"/>
        <property name="lastName" value="Joshi"/>
        <property name="balance" value="10000"/>
</bean>
<bean id="address" class=com.ameya.models.Address>
        <property name="building" value="Champions"/>
        <property name="street" value= "JM Road">
        <property name="city" value="Pune"/>
        <property name="country" value="India"/>
        <property name="state" value="MH"/>
</bean>
```

```
<bean id="account7"
class="com.ameya.models.Account">
        <property name="firstName" value="Ameya"/>
        <property name="lastName" value="Joshi"/>
        <property name="balance" value="10000"/>
        <property name="address" ref="address"/>
</bean>
```

Factory Methods

```
package com.ameya.services;
public class AccountService {
private static AccountService service = new AccountService();
private AccountService() {}
public static AccountService getInstance(){return service;
}
public static AccountService getInstance(String serviceName){
System.out.println("Service Name :"+serviceName);
return service;
}
}
```

```
<bean id="accountService"
class="com.ameya.services.AccountService"
factory-method="getInstance"/>

<bean id="accountService2“
class="com.ameya.services.AccountService"
factory-method= "getInstance">
        <constructor-arg name=”serviceName”
        value="Account"/>
</bean>
```

Instance Factory Methods

```
package com.ameya.utils;
public class AccountServiceLocator {
private static AccountService service = new AccountService();
private AccountServiceLocator(){}
public AccountService  createAccountServiceInstance(){
return service;
}
}
```

```
<bean id="serviceLocator"
class="com.ameya.utils.AccountServiceLocator"/>
<bean id="accountService3"
factory-bean="serviceLocator“
factory-method="createAccountServiceInstance"/>
```

Assigning Values To Properties

* In Spring, there are multiple ways to inject values into bean properties.
  * Normal way
  * shortcut
  * "p" namespace (Spring 2.0 and later)
  * "c" namespace (Spring 2.0 and later)

* **Normal way :**

```
<bean id="account" class="com.ameya.models.Account">
        <property name= "firstName">
                <value>Ameya</value>
        </property>
        <property name= "lastName">
                <value >Joshi</value>
        </property)
        <property name= "balance">
                <value>10000</value>
        </property>
</bean>
```

* **Shortcut Way :**

```
<bean id="account" class="com.ameya.models.Account">
        <property name="firstName" value="Ameya"/>
        <property name="lastName" value="Joshi"/>
        <property name="balance" value="10000"/>
</bean>
```

* **"p" namespace :**

```
<bean id="account"class="com.ameya.models.Account"
        p:firstName="Ameya"
        p:lastName="Joshi"
        p:balance="10000">
</bean>
```

* **“c" namespace :**

```
<bean id="account"class="com.ameya.models.Account"
        c:firstName="Ameya"
        c:lastName="Joshi"
        c:balance="10000">
</bean>
```

Collections: List, Set, Map

* Create the CollectionObject.java:

```
package com.ameya.collections;
import java. util. List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;
public class CollectionObject {
private List<String> firstNames;
private Set<String> lastNames;
private Map<String, Float> accountDetails;
private Properties adminEmails;
//Setter & Getter methods
}
```

```
<bean id="collectionObject" class="com.ameya.collections.CollectionObject">
<property name="firstNames">
        <list>
                <value>Ameya</value>
                <value>Sanjay</value>
                <value>Amol</value>
        </list>
</property>
<property name="lastNames">
        <set>
                <value >Joshi</value>
                <value>Patil</value>
                <value>Pawar</value>
        </set>
</property>
```

```
<property name="adminEmails">
        <props>
                <prop key="sales">sales@example.com</prop>
                <prop key="admin">admin@example.com</prop>
        </props>
</property>

<property name="accountDetails">
        <map>
                <entry key="A101" value="123.22"/>
                <entry key="A102" value="213.22"/>
                <entry key="A103" value="211.33"/>
        </map>
</property>
</bean>
```

FactoryBean Interface

* org.springframework.beans.factory.FactoryBean
  * Interface to be implemented by objects used within a BeanFactory that are themselves factories.
  * If a bean implements this interface, it is used as a factory, not directly as a bean.
* A bean that implements this interface cannot be used as a normal bean.
* A FactoryBean is defined in a bean style, but the object exposed for bean references is always the object that it creates.

ListFactoryBean

* org.springframework.beans.factory.config.ListFactoryBean
  * The ListFactoryBean is a simple factory for shared List instance.
  * The list element is defined in the xml bean definitions.
  * The setSourceList method sets the source List which is written in xml list element.
  * .The SetTargetList Class method sets the class to use as target List.

SetFactoryBean

* org.springframework.beans.factory.config.SetFactoryBean
  * The SetFactoryBean is a simple factory for shared Set instance.
  * The set element is defined in the xml bean definitions.
  * The setSourceSet method sets the source Set which is written in xml set element.
  * The SetTargetSet Class method sets the class to use as target Set.

MapFactoryBean

* org.springframework.beans.factory.config.MapFactoryBean
* The MapFactoryBean is a simple factory for shared Map instance.
* The map element is defined in the xml bean definitions.
* The setSourceMap method sets the source Map which is written in xml map element.
* The SetTargetMapClass method set the class to use as target Map.

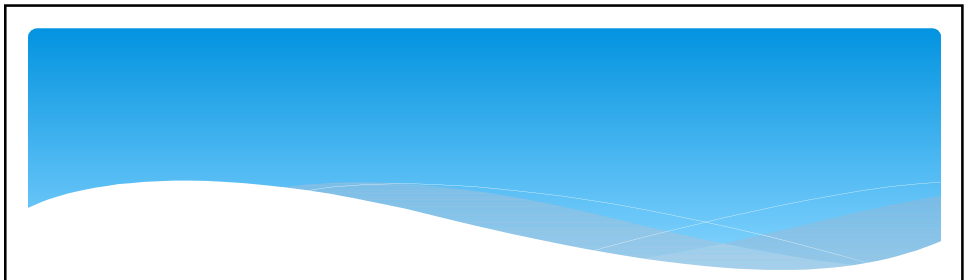
```
<bean id="cObj" class="com.ameya.collections.CollectionObject">
        <property name="firstNames">
                <bean class="org. springframework. beans.factory.
                        config.ListFactoryBean">
                        <property name="targetListClass">
                                <value>java.util. ArrayList</value>
                        </property>
                        <property name="sourceList">
                                <list>
                                <value>Ameya</value>
                                <value>Sanjay</value>
                                <value>Amol</value>
                                </list>
                        </property>
                </bean>
        </property>
```

```
<property name="lastNames">
        <bean class="org.springframework.beans.factory.config
                .SetFactoryBean”>
                <property name="targetSetClass">
                        <value>java.util.TreeSet</value>
                </property>
                <property name="sourceSet">
                        <set>
                                <value>Joshi</value>
                                <value>Patil</value>
                                <value>Pawar</value>
                        </set>
                </property>
        </bean>
</property>
```

```
<property name="accountDetails">
        <bean class="org.springframework.beans.factory.config
                .MapFactoryBean”>
        <property name=”targetMapClass”>
                <value>java.util.HashMap</value>
        </property>
        <property name="sourceMap">
                <map>
                        <entry key="A101" value="123.22">
        </entry>
                        <entry key="A102" value="213.22">
        </entry>
                        <entry key="A103" value="211.33">
        </entry>
                </map>
        </property>
        </bean>
</property>
</bean>
```

util namespeace for collections

```
<util:map id= "ad" map-class="java.util.HasMap">
        <entry key="A101" value="123.22"></entry>
        <entry key="A102" value="213.22"></entry>
        <entry key="A103" va1ue="211.33"></entry>
</util:map>
<bean id="cObj" class="com.ameya.collections.CollectionObject">
        <property name="accountDetails" value="#{ad}"/>
        <property name="firstNames>
                <util:list list-class=”java.utit.LLnkedLLst”>
                        <value>Ameya</value>
                        <value>Sanjay</value>
                        <value>Amol</value>
                </util:list>
        </property>
```



```
        <property name="lastNames">
                <util:set set-class="java.util.HashSet">
                        <value>Joshi</value>
                        <value>Patil</value>
                        <value>Pawar</value>
                </util:set>
        </property>
        <property name="adminEmails">
                <util:properties>
                        <prop key=”manager”>manager@example.com
                        </prop>
                </util:properties>
        </property>
</bean>
```

depends-on

* Using depend-on for example,

a static initializer in a class needs to be triggered, such as database driver registration. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed.

```
<bean id="customer" class="com.ameya.models.Customer"
depends-on= “address”>
        <property name="custId" value="101"/>
        <property name="custName" value="Ameya"/>
        <property name="address" ref="address"/>
</bean>
```

Lazy-initialized beans

* A lazy-initialized bean tells the IOC container to create a bean instance when it is first requested, rather than at startup.

```
<bean id="customer" class="com.ameya.models.Customer"
depends-on="address" lazy-init="true">
        <property name="custId" value="101"/>
        <property name="custName" value="Ameya"/>
        <property name="address" ref="address"/>
</bean>
```

Autowiring

* The Spring container can autowire relationships between collaborating beans.
* You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext.

Autowiring

* The autowiring functionality has four modes:
* **no** (Default) : No autowiring. Bean references must be defined via ref element.
* **byName** : Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired.
* **byType** : Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
* **constructor** : Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

Bean Scopes

* When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition.
* The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

Bean Scopes

* Singleton
* Prototype
* Request
* Session
* Global

Bean definition inheritance

* A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.
* A child bean definition inherits configuration data from a parent definition.
* The child definition can override some values, or add others, as needed.
* Using parent and child bean definitions can save a lot of typing.
* Effectively, this is a form of templating

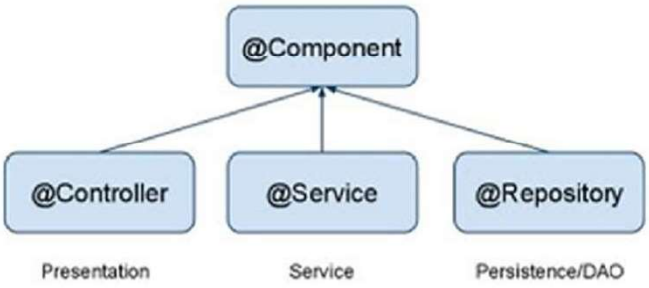
```
<beans ...>
        <bean id="customer1“  class="com.ameya.models.Customer">
                <property name="person" ref="person"/>
        </bean>

        <bean id="customer2" parent="customer1">
                <property name="customerBill" ref="bill"/>
        </bean>
</beans>
```

Annotation-based container configuration

* Starting from Spring 2.5 it became possible to configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.
* Annotation injection is performed before XML injection, thus the later configuration will override the former for properties wired through both approaches.

* Annotation wiring is not turned on in the Spring container bydefault. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file.
* To do this You need to add the context schema and the below tag to xml file
  * **<context: annotation-config />**

StereoType Annotations



@Component

* Annotate your other components (for example REST resource classes) with @Component.

```
@Component
public class ContactResource {
        ......
}
```

@Component is a generic stereotype for any Spring-managed component.

@Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.

@Service

* Annotate all your service classes with @Service.
* All your business logic should be in Service classes.

```
@Service
public class CompanyServiceImpl implements
CompanyService {
        ……
}
```

@Repository

* Annotate all your DAO classes with @Repository.
* All your database access logic should be in DAO classes.

```
@Repository
public class CompanyDAOImpl implements CompanyDAO {
        .....
}
```

@Required

* The @Required annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time otherwise the container throws a BeanInitializationException exception.

```
import org.springframework.beans.factory.annotation.Required;
public class Student {
        private Integer age;
        private String name;

        @Required
        public void setAge(Integer age) {
                this.age = age;
        }
        ....
}
```

@Autowired

* The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished.

* The @Autowired annotation can be used to auto wire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

@Autowired on Setter Methods

* You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file.
* When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

```
import org.springframework.beans.factory.annotation.Autowired;
public class TextEditor {
        private SpellChecker spellChecker;
        @Autowired
        public void setSpellChecker( SpellChecker spellChecker ){
                this.spellChecker = spellChecker;
        }
.....
}
```

@Autowired on Properties

* You can use @Autowired annotation on properties to get rid of the setter methods.
* When you will pass values of auto wired properties using @Autowired Spring will automatically assign those properties with the passed values or references.

```
import org.springframework.beans.factory.annotation.Autowired;
public class TextEditor {
        @Autowired
        private SpellChecker spellChecker;
....
}
```

@Autowired on Constructor

* You can apply @Autowired to constructors as well.
* A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file.

```
import org. springframework.beans.factory.annotation.Autowired;
public class TextEditor {
        private SpellChecker spellChecker;
        @Autowired
        public TextEditor(SpellChecker spellChecker){
                System.out.println("Inside TextEditor constructor.");
                this.spellChecker = spellChecker;
        }
....
}
```

@Qualifier

* There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use @Qualifier annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
public class Profile {
        @Autowired
        @Qualifier("student1")
        private Student student;
        ....
}
```

JSR-250 Annotations Support

* Spring also supports JSR-250 based annotations which include @PostConstruct, @PreDestroy annotations.

```
import javax.annotation.*;
public class HelloWorld {
        @PostConstruct
        public void init(){
                System.out.println("Bean is going through init.");
        }
        @PreDestroy
        public void destroy(){
                System.out.println("Bean will destroy now.");
        }
....
}
```

Java Based Configuration

* Java based configuration option enables you to write most ofyour Spring configuration without XML but with the help of few Java-based annotations.
* @Configuration
* @Bean
* @Import
* @Scope

@Configuration & @Bean

* Annotating a class with the @Configuration indicates that the class can be used by the Spring IOC container as a source for bean definitions.
* The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

```
import org.springframework.context.annotation.*;
@Configuration
public class HelloWorldConfig {
        @Bean
        public HelloWorld helloWorld{
                return new HelloWorld();
        }
}
```

* Above code will be equivalent to the following XML configuration:

```
<beans>
        <bean id="helloWorld" class="com.ameya.Helloworld"/>
</beans>
```

AnnotationConfigApplicationContext

```
public static void main(String[] args){
        ApplicationContext ctx = new
        AnnotationConfigApplicationContext(HelloWorldConflg.class);
        HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
        helloWorld.setMessage("Hello World!");
        helloWorld.getMessage();
}
```

@Import

* The @Import annotation allows for loading Bean definitions from another configuration class.
* Consider a ConfigA class as follows:

```
@Configuration
public class ConfigA {
        @Bean
        public A a() { return new A(); }
}
```

* You can import above Bean declaration in another Bean Declaration as follows

```
@Configuration
@Import(ConfigA.class)
public class ConfigB {
        @Bean
        public B a() { return new A(); }
}
```

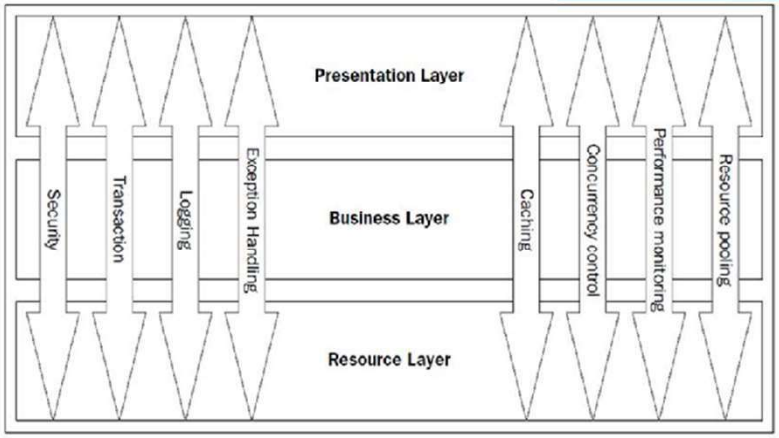
@Scope

* The default scope is singleton, but you can override this with the @Scope annotation as follows:

```
@Configuration
public class AppConfig {
        @Bean
        @Scope("prototype")
        public Employee employee {
                return new Employee();
        }
}
```

Spring AOP

Object Oriented Programming-Limits



Aspect-oriented programming

* Aspect-oriented programming supports object-oriented programming by de-coupling modules that implement cross-cutting concerns.
* Its purpose is the separation of concerns.
* In object-oriented programming the basic unit is the Class, whereas in aspect-oriented programming it's the Aspect.

AOP terms: (Aspect)

* A modularization of a concern that cuts across multiple classes.
* Transaction management is a good example of a cross-cutting concern in enterprise Java applications.

AOP terms: (Join point)

* A point during the execution of a program, such as the execution of a method or the handling of an exception.
* In Spring AOP a join point always represents a method execution.

AOP terms: (Advice)

* This is action taken by an aspect at a particular join point.
* Different types of advice include "around", "before" and "after" advice.
* Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.

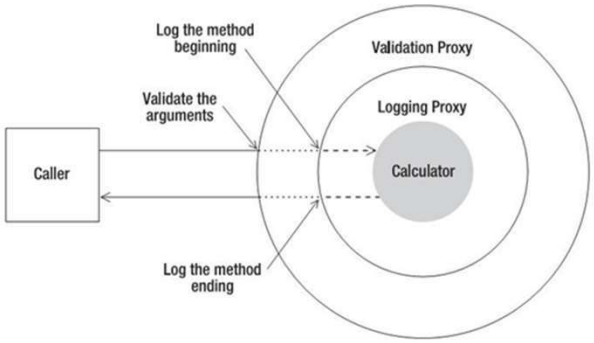
AOP terms: (Pointcut)

* A predicate that matches join points.
* Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).

Dynamic Proxy

* Can be implemented writing static proxies using proxy design pattern or through the dynamic proxy support offered by JDK version 1.3 or higher. Only restriction is must use at least one interface or else you can use CG LIB proxy.
* jdk proxy requres invocation handler which is an interface having invoke method.

Proxy



Spring AOP Architecture

* The core architecture of Spring AOP is based around proxies.
* Using ProxyFactory is a purely programmatic approach to creating AOP proxies.
* For the most part, you don't need to use this in your application; instead, you can rely on the declarative AOP configuration mechanisms provided by Spring(the ProxyFactoryBean class, the aop namespace, and @AspectJ-style annotations) to provide declarative proxy creation.

* At runtime, Spring will analyze the cross-cutting concerns defined for the beans in the ApplicationContext and generate proxy beans(which wraps the underlying target bean) dynamically.
* Instead of calling the target bean directly, callers are injected with the proxied bean, and any calls to the target are received by the proxy bean. The proxy bean will then analyze the running condition (i.e., join point, pointcut, advice, etc.) and weave in the appropriate advice accordingly.

* Internally, Spring has two proxy implementations: the JDK dynamic proxy and the CG LIB proxy.
* By default, when the target object to be advised implements some sort of an interface, Spring will use a JDK dynamic proxy to create proxy instances of the target.
* However, when the advised target object doesn't implement an interface (e.g., it's a concrete class), CGLIB will be used for proxy instance creation.
* One major reason is that JDK dynamic proxy only supports proxying of interfaces.

* The ProxyFactory class controls the weaving and proxy creation process in Spring AOP.
* ProxyFactory class provides the addAdvice(), addAdvisor(), removeAdvice(), removeAdvisor() & adviceInclude() methods.

Types Of Advice

Type	Interface
Around	org.aopalliance.intercept.MethodInterceptor
Before	org.springframework.aop.BeforeAdvice
After	org.springframework.aop.AfterReturningAdvice
Throws	org.springframework.aop.ThrowsAdvice

Custom Aspects Implementation

* Spring supports the @AspectJ annotation style approach and the XML schema-based approach to implement custom aspects.

Approach	Description
XML Schema based	Aspects are implemented using regular classes along with XML based configuration.
@AspectJ based	@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

Enable AspectJ

* The AspectJ support is enabled by including the following element inside your spring configuration:
  * <aop:aspectj-autoproxy/>.
* If you are using the DTD, it is still possible to enable AspectJ support by adding the following definition to your application context:
  * <bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"/>

AspectJ Advices

* @Before
* @After
* @AfterReturning
* @AfterThrowing
* @Around

Spring JDBC And Transactions

Spring DAOSupport

* Extend your DAO classes with the proper xxxDAOSupport class that matches your persistence mechanism.
* JdbcDaoSupport
  * Super class for JDBC data access objects.
  * Requires a DataSource to be set, providing a JdbcTemplate based on it to subclasses.

* HibernateDaoSupport
  * Super class for Hibernate data access objects.
  * Requires a SessionFactory to be set, providing a HibernateTemplate based on it to subclasses.

* JdoDaoSupport
  * Super class for JDO data access objects.
  * Requires a PersistenceManagerFactory to be set, providing a JdoTemplate based on it to subclasses.

* SqlMapDaoSupport
  * Super class for iBATIS SqlMap data access object.
  * Requires a DataSource to be set, providing a SqlMapTemplate

Spring DAO Templates

* Built in code templates that support JDBC, Hibernate, JDO, and iBatis SQL Maps.
* Simplifies data access coding by reducing redundant code and helps avoid common errors.
* Alleviates opening and closing connections in your DAO code.
* No more ThreadLocal or passing Connection/Session objects.
* Transaction management is handled by a wired bean.
* You are dropped into the template with the resources you need for data access – Session, PreparedStatement, etc..
* Optional separate JDBC framework.

Consistent Exception Handling

* Spring has it's own exception handling hierarchy for DAO logic.
* No more copy and pasting redundant exception logic!
* Exceptions from JDBC, or a supported ORM, are wrapped up into an appropriate, and consistent, DataAccessException and thrown.
* This allows you to decouple exceptions in your business logic.
* These exceptions are treated as unchecked exceptions that you can handle in your business tier if needed. No need to try/catch in your DAO.
* Define your own exception translation by subclassing classes.

Transactions

* A transaction comprises a unit of work performed within a database management system (or similar system) against a database.
* Transactions provide "all-or-nothing" proposition. Each unit of work performed should be entirely committed or rolled back.

Transactions - ACID

* Atomic: Atomicity ensures that all operations in the transaction happen or none of them happen.
* Consistent: The system should be left in a consistent state even if the transaction succeeds or fails.
* Isolated: Should allow multiple users to work. Transactions should be isolated from each others work. This prevents concurrent reads and writes.
* Durable: After successful transaction, the system should store the details permanently.

Before Spring transactions

* EJB was a powerful API available offering Bean Managed(BMT) and Container Managed Transactions(CMT).
* It offers both programmatic and declarative based transaction management.
* Problems:
  * Needs an application server to run the CMT.
  * EJB relies on JTA for transactions.
  * Using EJB itself is a heavy choice.

Spring Transactions Advantages

* Offers both programmatic and Declarative transactions.
* Declarative transactions are equivalent of the Container managed transactions and there is no need for an application server.
* No need for using JTA.
* Wide range of Transactional Managers are defined in spring transactions API.

Spring Transactions Limitations

* Spring transactions are good at applications using single database.
* If you need to access multiple databases, distributed transactional(XA) management is needed.
* Spring supports any kind of JTA implementation.

Programmatic Vs Declarative Transactions

* Programmatic transactions give precise control on the boundaries of the transaction.
* E.g.

```
ticEketBooking(){
        T1.start();
                checkAvailability();
        T1.Commit();
        T2.start();
                selectSeats();
                payment();
                ticketConfirmation();
        T2.commit();
}
```

* Declarative transactions on the other hands are less intrusive and are defined in a Configuration file.
* Developed based on the AOP concepts. This gives an advantage of keeping the cross-cutting concerns like transactions out of our DAO layer code.

Spring TransactionManagers

```
<bean id="transactionManager“
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref=”dataSource’/>
</bean>

<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">    <property
name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref=      "entityManagerFactory"/>
</bean>

<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager"> <property
name="transactionManagerName"        value="java:/TransactionManager"/>
</bean>
```

Transaction Attributes

* Propagation
* Isolation
* Read only
* Rollback rules
* Timeout

Propogation

* Propagation behavior defines the boundaries of a transaction.
* Propagation configuration tells the transaction manager if a new transaction should be started or it can use the transaction which already exists.

Propogation Rules

* **PROPAGATION_MANDATORY**: Method should run in a transaction and ifno transaction exists — exception will be thrown.
* **PROPAGATION_NESTED**: Method should run in a nested transaction.
* **PROPAGATION_NEVER**: Current method should not run in a transaction. If exists an exception will be thrown.
* **PROPAGATION_NOT_SUPPORTED**: Method should not run in a transaction. Existing transaction will be suspended till method completes the execution.
* **PROPAGATION_REQUIRED**: Method should run in a transaction. If already exists, method will run in that and if not, a new transaction will be created.
* **PROPAGATION_REQUIRES_NEW**: Method should run in a new transaction. If already exists, it will be suspended till the method finishes.
* **PROPAGATION_SUPPORTS**: Method need not run in a transaction. But if already exists, it supports one which is already in progress.

ISOLATION Levels

* Dirty reads occur when transaction reads an uncommitted data.
* Non-repeatable Reads occurs when a transaction reads the same data multiple times but gets different results each time.
* Phantom Reads occur when two transactions work on a same row where one updates and other reads.The reading transaction get new data.

* **ISOLATION_DEFAULT**: default isolation specific to the data source.
* **ISOLATION_READ_UNCOMMITTED**: Read changes that are uncommitted. Leads to dirty reads, Phantom reads and non repeatable reads.
* **ISOLATION_READ_COMMITTED**: Reads only committed data. Dirty read is prevented but repeatable and non repeatable reads are possible.
* **ISOLATION_REPEATABLE_READ**: Multiple reads of same field yield same results unless modified by same transaction. Dirty and non repeatable reads are prevented but phantom reads are possible as other transactions may edit the fields.
* **ISOLATION_SERIALIZABLE**: Dirty, phantom and non repeatable reads are prevented. But hampers the performance of application.

Read-Only

* A read-only transaction does not modify any data.
* Read-only transactions can be a useful optimization in some cases

Transaction Timeout

* By declaring the Timeout attribute, we can ensure that long running transactions are rolled back after certain number of seconds.
* The count down starts when transaction is started. So, Timeout attribute can be meaningful when applied for
  * PROPAGATION REQUIRED
  * PROPAGATION_REQUIRES_NEW
  * PROPAGATION_NESTED

Rollback

* Rollback tells a transaction manager when to rollback a transaction when an exception occurs.
* By default the transactions will be rolled back when runtime exceptions occurs.
* But, by specifically mentioning the checked exceptions, transaction manager will be able to rollback the transactions.

<tx:annotation-driven/>

* @Transactional annotation is simply metadata
* <tx:annotation-driven/> element switches on the transactional behavior.

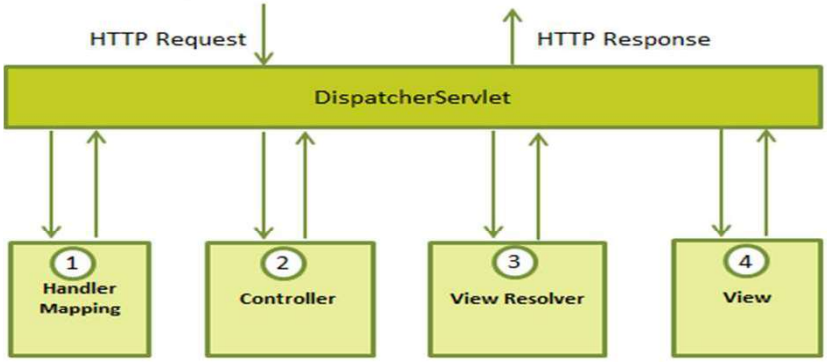
Spring MVC

Spring MVC

* The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.
* The Model encapsulates the application data and in general they will consist of POJO.
* The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.
* Thus MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

The DispatcherServlet

* The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.



web.xml

* You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the web.xml file.

```
<web-app>
<display-name>Example</display-name>
<context:component-scan base-
package="com.proj.web" />
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
/WEB-INF/context/example-general-context.xml
</param-value>
</context-param>
<listener>
<listener-class>
org.springframework.web.context.
ContextLoaderListener
</listener-class>
</listener>
```

```
<servlet>
<servlet-name>spring</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/example-servlet.xml</param-
value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>spring</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

springmvc-servlet.xml

* Now, let us check the required configuration for example-servlet.xml file, placed in your web application's WebContent/WEB-INF directory:

```
<context:component-scan base-package="com.proj.web.controller" />
<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
<property name="viewClass"
value="org.springframework.web.servlet.view.JstlView" />
<property name="prefix" value="/WEB-INF/view/" />
<property name="suffix" value=".jsp" />
</bean>
<mvc:annotation-driven />
</beans>
```

Controller & View

* HomeController.java

```
@Controller
public class HomeController {
@RequestMapping(value = { "/", "/home" },
method = RequestMethod.GET)
public String showHomePage(ModelMap
model) {
model.addAttribute("message", "Hello
Spring MVC Framework!");
return "home";
}
}
```

* home.jsp

```
<html>
<head>
<title>Here is home page</title>
</head>
<body>
<h2>${message}</h2>
</body>
</html>
```

Request Mappings

* RequestMappings are really flexible
* You can define a @RequestMapping on a class and all methods
* @RequestMappings will be relative to it.
* There are a number of ways to define them:
  * URI Patterns
  * HTTP Methods (GET, POST, etc)
  * Request Parameters
  * Header values

RequestMapping - Class Level

```
package com.ameya.controllers;

@RequestMapping ("/portfolio")
@Controller
public class PortfolioController {
        @RequestMapping("/create")
        public String create() {
                return "create";
        }
}
```

* The URL for this (relative to your context root)would be:
  /portfolio/create

RequestMapping - HTTP Methods

```
package com.ameya.controllers;

@RequestMapping("/portfolio")
@Controller
public class PortfolioController {
        @RequestMapping(value="/create",method=RequestMethod.POST)
        public String save() {
                return "view";
        }
}
```

* Same URL as the previous example, but responds to POSTs

RequestMapping - Request Params

```
package com.ameya.controllers;

@RequestMapping("/portfolio")
@Controller
public class PortfolioController {
@RequestMapping(value="/view",params="details=all")
public String viewAll() {
return "viewAll";
}
}
```

* This will respond to
  /portfolio/view?details=all

RequestMapping - URI Templates

```
package com.ameya.controllers;

@RequestMapping("/portfolio")
@Controller
public class PortfolioController {
        @RequestMapping ("/viewProject/{projectId}")
        public String viewProject() {
                return "viewProject";
        }
}
```

* The URL for this (relative to your context root) would be:
  /portfolio/viewProject/10

Controller Method Arguments

* Sometimes you need access to the request,session, request body, or other items
* If you add them as arguments to your controller method, Spring will pass them in.

```
@RequestMapping(value="/")
public String getProject(HttpServletRequest request,
                HttpSession session,
                @RequestParam("projectId") Long projectId,
                 @RequestHeader ("content-type") String contentType)
{
        return "index";
}
```

Supported Method Arguments

* Request/Response objects
* Session object
* Spring's WebRequest object.
* java.util.Locale.
* java.io.Reader (access to request content)
* java.io.Writer (access to response content)
* java.security.Principal
* ModelMap
* org.springframework.validation.Errors
* org.springframework.validation.BindingResult

Supported Annotations on params

* @PathVariable
* @RequestParam
* @RequestHeader
* @RequestBody

The Model

* You populate the view with data via the **ModelMap** or **ModelAndView** (which has a ModelMap underneath).
  * This is basically a Map
  * All attributes are added to the request so that they can be picked up by JSPs

ModelMap

```
public String doController(ModelMap modelMap) {
        modelMap.addAttribute(user);
        modelMap.addAttribute("otherUser",user);
        return "index";
}
```

* We could consume it on JSP as follows :

```
User: ${user}<br/><br/>
Other User: ${otherUser}
```

ModelandView

```
public ModelAndView doController() {
        ModelAndView mav = new ModelAndView("index");
        mav.addObject(user);
        mav.addObject("otherUser",user);
        return mav;
}
```

* We could consume it on JSP as follows :

```
User: ${user}<br/><br/>
Other User: ${otherUser}
```

Spring REST

HTTP Message

* What does an HTTP message look like?

```
GET /view/1 HTTP/1.1          <- Request Line
User-Agent: Chrome            <- Headers
Accept: application/json
[CRLF]
```

```
POST /save HTTP/1.1           <- Request Line
User-Agent: IE
Content-Type: application/x-www-form-urlencoded   <- Headers
[CRLF]
name=x&id=2                   <- Request Body
```

HTTP Message - Responses

```
HTTP/1.1 200 OK               <- Status Line
Content-Type: text/html       <- Headers
Content-Length: 1337
[CRLF]
<html>
Some HTML Content.            <- Response Body
</html>
```

```
HTTP/1.1 500 Internal Server Error   <- Status Line
```

```
HTTP/1.1 201 Created          <- Status Line
Location: /view/7             <- Headers
[CRLF]
Some message goes here.       <- Response Body
```

RequestBody

* Annotating a handler method parameter with @RequestBody will bind that parameter to the request body

```
@RequestMapp±ng("/echo/string")
public void writeString(@RequestBody String input) { }

@RequestMapping("/echo/json")
public void writeJson(@RequestBody SomeObject input) { }
```

ResponseBody

* Annotating a return type with @ResponseBody tells Spring MVC that the object returned should be treated as the response body

```
@RequestMapping("/echo/str±ng")
public @ResponseBody String readString() { }

@RequestMapping("/echo/json")
public @ResponseBody SomeObject readJson() { }
```

HttpMessageConverters

* How does Spring MVC know how to turn a JSON string into SomeObject, or vice-versa?

* HttpMessageConverters
* These are responsible for converting a request body to a certain type, or a certain type into a response body
* Spring MVC figures out which converter to use based on Accept and Content-Type headers, and the Java type
* Your Accept and Content-Type headers DON'T have to match. For example, you can send in JSON and ask for XML back

StringHttpMessageConverter

* Reads and writes Strings.
* Reads text/*
* Writes text/plain

StringHttpMessageConverter

```
@RequestMapping("/echo/string")
public @ResponseBody String echoString(@RequestBody String input) {
        return "Your Text Was: " + input;
}
```

REQUEST

```
POST /echo/string HTTP/1.1
Accept: text/plain
Content-Type: text/plain

Hello!
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 17

Your Text Was: Hello!
```

MappingJacksonHttpMessageConverter

* Maps to/from JSON objects using the Jackson library
* Reads application/json
* Writes application/json

MappingJacksonHttpMessageConverter

```
public Person {
// String name, int age;
}

@RequestMapping("/echo/json")
public @ResponseBody Person echoJson(@RequestBody Person person) {
        // Upper case name, square age
        return person;
}
```

REQUEST

```
POST /echo/string HTTP/1.1
Accept: application/json
Content-Type: application/json

{ "name" : "Spencer", "age" : 5 }
```

RESPONSE

```
HTTP/1.1 201 Created
Content-Type: application/json

{ "name" : "SPENCER", "age" : 25 }
```

Jaxb2RootElementHttpMessageConverter

* Maps to/from XML objects
* Must have your object at least annotated with @XmlRootElement
* Reads text/xml, appilcation/xml
* Writes text/xml, application/xml

Jaxb2RootElementHttpMessageConverter

```
@XmlRootElement
public Person {// String name, int age;}

@RequestMapping("/echo/xml")
public @ResponseBody Person echoXml(@RequestBody Person person) {
        // Upper case name, square age
        return person;
}
```

REQUEST

```
POST /echo/string HTTP/1.1
Accept: application/xml
Content-Type: application/xml

<thing><name>Spencer</name><age>5</age></thing>
```

RESPONSE

```
HTTP/1.1 201 Created
Content-Type: application/xml

<thing><name>SPENCER</name><age>25</age></thing>
```

Other parts of the HttpMessage

* What if you need to get/set headers?
* Or set the status code?

```
@RequestMapping("/echo/string")
public @ResponseBody String echoString(
                        @RequestBody String input,
                         HttpServletRequest request,
                         HttpServletResponse response) {
        String requestType = request.getHeader("Content-Type");
        response.setHeader("Content-Type", "text/plain");
        response.setStatus(200);
        return input
}
```

@ResponseStatus

* There is a convenient way to set what the default status for a particular handler should be

```
@RequestMapping("/create")
@ResponseStatus(HttpStatus.CREATED) // CREATED = 201
public void echoString(String input) {
}
```

HttpEntity

* Convenience class for dealing with bodies, headers, and status
* Converts messages with HttpMessageConverters

```
@RequestMapping("/image/upload")
public ResponeEntity<String> up1oad(HttpEntity<byte[]> rEntity) {
        String t = rEntity.getHeaders.getFirst("Content-Type");
        byte[] data = rEntity.getBody();// Save the file
        HttpHeader responeHeader = new HttpHeaders();
        responseHeaders.set("Location", "/image/1");
        return new ResponseEntity<String>
        ("Created",responseHeaders, HttpStatus.CREATED);
}
```

Dealing with Exceptions

* By default, Spring MVC will map certain exceptions to status codes
* You can implement your own HandlerExceptionResolver, which takes an exception and returns a ModelAndView
* You can register a SimpleMappingExceptionResolver to map exceptions to views
* You can annotate methods in the controller to handle specific exceptions

Default Exception Mappings

* These take effect if you have no other configuration

* ConversionNotSupported Exception => 500
* NoSuchMethodHandlingException => 404
* MissingServletRequestParameterException => 400
* HttpRequestMethodNotSupportedException =>405
* TypeMismatchException => 400
* HttpMediaTypeNotSupportedException => 415
* HttpMediaTypeNotAcceptableException => 406

@ExceptionHandler

* Create a method to handle the exception, annotate it with @ExceptionHandler, and pass the exception(s) that method can handle
* ExceptionHandler methods look a lot like normal handler methods

```
@RequestMapping("/error")
public void dosomething() {
        throw new DataAccessException("Unable to access the database");
}

@ExceptionHandler(DataAccessException.class)
public @ResponseBody String handleDataAccessError(DataAccessException ex) {
        return ex.getMessage();
}
```

@ResponseStatus (For Exceptions)

* We saw this annotation earlier
* It can also be placed on Exception classes or @ExeptionHandler methods to return a specific status code for a particular exception

```
@Responsestatus(HttpStatus.INTERNAL_SERVER_ERROR)
@ExceptionHandler(DataAccessException.class)
public void handleDataAccessError(DataAccessException ex) {}

@ResponseStatus(value=HttpStatus.PAYMENT_REQUIRED,
message="I need money.")
public class PaymentRequiredException {}
```

Spring REST Template

RestTemplate

* RestTemplate communicates with HTTP server using RESTful principals.
* RestTemplate provides different methods to communicate via HTTP methods.
* This class provides the functionality for consuming the REST Services in a easy and graceful manner.
* When using the said class the user has to only provide the URL, the parameters(if any) and extract the results received.
* The RestTemplate manages the HTTP connections.

RestTemplate Methods

HTTP method	RestTemplate methods
DELETE	delete(java.lang.String, java.lang.Object...)
GET	getForObject(java.lang.String, java.lang.Class<T>, java.lang.Object...)
	getForEntity(java.lang.String, java.lang.Class<T>, java.lang.Object...)
POST	postForLocation(java.lang.String, java.lang.Object, java.lang.Object...)
	postForObject(java.lang.String, java.lang.Object, java.lang.Class<T>, java.lang.Object...)
PUT	put(java.lang.String, java.lang.Object, java.lang.Object...)

HTTP GET Using RestTemplate

```
RestTemplate restTemplate = new RestTemplateO;

String url ="http://localhost:8o8o/demo/rest/employees/{id};

Map<String, String> map = new HashMap<String,String>();
map.put("id", "101");

ResponseEntity<Employee> entity =restTemplate.getForEntity(url,
Employee.class, map);

System. out.println(entity.getBody());
```

HTTP POST Using RestTemplate

```
RestTemplate restTemplate = new RestTemplate();

String url ="http://localhost:8080/demo/rest/employees";

Employee employee =restTemplate.postForObject(url,
newEmployee,Employee.class);

System.out.println(employee);
```

HTTP PUT Using RestTemplate

```
RestTemplate restTemplate = new RestTemplate();

String url ="http://localhost:8080/demo/rest/employees/{id}";

Map<String, String> map = new HashMap<>();
map.put("id", "80");

restTemplate.put(url, updatedEmployee, map);
```

HTTP DELETE Using RestTemplate

```
RestTemplate  restTemplate = new RestTemplate();

String url ="http://localhost:8080/demo/rest/employees/{id}";

Map<String, String> map = new HashMap<>();
map.put("id", "101");

restTemplate.delete(url, map);
```

THANK YOU.