1. @RestController annotation: tells Spring to render the resulting string directly back to caller and consider it when handling incoming web requests.
2. @RequestMapping annotation: provides “routing” information. It is telling Spring that ayn HTTP request with the path “/” should be mapped to the home method
3. @EnableAutoConfiguration class-level annotation: Spring Boot to guess how you will want to configure Spring, based on the jar dependencies that you have added.
4. Recommend that you locate your main application class in a root package above other classes. Using a root package also allows the @ComponentScan annotation to be used without needing to specify a basePackage attribute. Here is a typical layout:

com

+- example

+- myproject

+- Application.java

|

+- domain

| +- Customer.java

| +- CustomerRepository.java

|

+- service

| +- CustomerService.java

|

+- web

+- CustomerController.java

1. Importing additional configuration classes: The @Import annotation can be used to import additional configuration classes. Alternatively, you can use @ComponentScan to automatically pick up all Spring components, including configuration classes. You can use an additional @ImportResource annotation to load XML configuration files in a @Configuration class.
2. Automatic restart: in IntelliJ IDEA, building the project (Build->Make Project) will restart the application
3. The banner that is printed on start up can be changed by putting banner.txt into classpath: (Application.classs.getResource(“/”).toString())

# Spring IoC

1. In spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called *beans.* A bean is an object that is instantiated, assembled, and otherwise managed by a Spring Ioc container
2. ApplicationContext represents the Spring IoC container and gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata.

ClassPathXmlApplicationContext and FileSystemXmlApplicationContext implements the ApplicationContext interface.

## Configuration metadata

XML-based configuration metadata shows beans configured as <bean /> elements inside a top-level <beans />. Java configuration typically uses @Bean annotated methods within a @Configuration class.

We can use the application context constructor to load bean definitions from all XML fragments. Alternatively, use one or more occurrences of the <import/> element to load bean definitions from another file of files.

<import resource="dispatcher-servlet.xml" />

It is not recommended to reference files in parent directories using a relative “../” path. It is better form not to use the “/” at all.

If use absolute locations, it is preferable to keep an indirection for each absolute locations, for example, through “${...}” placeholders.

Whit component scanning in the classpath, Spring generates bean names for unnamed components, following the rules: taking the simple class name and turning tis initial character to lower-case.

Alias definition:

<alias name="internalResourceViewResolver" alias="viewResolver"/>

Static Inner class names: com.example.Foo$Bar

**Instantiation with a constructor**

<bean id="exampleBean" class="examples.ExampleBean" />

**Instantiation with a static factory method**

When defining a bean that you create with static factory method, you use the class attribute to specify the class containing the static factory method and an attribute named factory-method to specify the name of the factory method itself.

<bean id="clientService" class="com.wang.test.controller.Controller" factory-method="createInstance"/>

**Instantiation using an instance factory method**

Instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean.

<!-- the factory bean, which contains a method called createInstance() -->  
<bean id="serviceLocator" class="examples.DefaultServiceLocator">  
 <!-- inject any dependencies required by this locator bean -->  
</bean>  
<!-- the bean to be created via the factory bean -->  
<bean id="clientService"  
 factory-bean="serviceLocator"  
 factory-method="createClientServiceInstance"/>

public class DefaultServiceLocator {  
 private static ClientService *clientService* = new ClientServiceImpl();  
 private DefaultServiceLocator() {}  
 public ClientService createClientServiceInstance() {  
 return *clientService*;  
 }  
}

**Dependency Injection (DI)**

DI exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

It is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the @Required annotation on a setter method can be used to make the property a required dependency.

Instead of using a constructor, Spring is told to call a static factory method to return an instance of the object (the returned type not have to be the class type of the factory method in):

<bean id="example" class="examples.Example" factory-method="createInstance">  
 <constructor-arg ref="anotherExampleBean"/>  
 <constructor-arg ref="yetAnotherBean"/>  
 <constructor-arg value="1"/>  
</bean>  
<bean id="anotherExampleBean" class="examples.AnotherBean"/>  
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

For straight values (primitives, Strings, and so on), the value attribute of <property /> element specifies a property or constructor argument as a human-readable string representation.

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
 <property name="driverClassName" value="${jdbc.driverClassName}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
</bean>

Also, can use the p-namespace for even more succinct XML configuration.

xmlns:p="http://www.springframework.org/schema/p"

<bean id="dataSourceP" class="org.springframework.jdbc.datasource.DriverManagerDataSource"  
 p:driverClassName="com.mysql.jdbc.Driver"  
 p:url="${jdbc.url}"  
 p:username="${jdbc.username}"  
 p:password="${jdbc.password}"  
/>

**Using depends-on**

If a bean is a dependency of another that usually means that one bean is set as a property of another. Sometimes dependencies between beans are less direct; the depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized.

<bean id="beanOne" class="ExampleBean" depends-on="manager"/>  
<bean id="manager" class="ManagerBean" />

**Limitations and disadvantages of autowiring**

Consider the limitations and disadvantages of autowiring:

* Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire so-called simple properties such as primitives, Strings and Classes
* Autowiring is less than explicit wiring.
* Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired.

**Compare Annotation-based container configuration with XML**

Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them.

Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

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

<context:annotation-config /> only looks for annotations on beans in the same application context in which it defined. This means that, if you put <context:annotation-config />in a WebApplicationContext for a DispatcherServlet, it only checks for @Autowired beans in your controllers, and not your services.

**@Required annotation applies to bean property setter methods**

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring.

**@Autowired annotation (@Inject)**

You can apply the @Autowired annotation to constructors. The annotation constructor is no longer necessary if the target bean only defines one constructor.

You can also apply the @Autowired annotation to “traditional” setter methods

Also can apply the @Autowired to fields as well and even mix it with constructors.

Can use @Autowired(required=false) if there is no candidate beans are available.

@Autowired’s required attribute is recommended over the `@Required annotation.

Can also use @Autowired for interfaces that are well-known resolvable dependencies: BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource

**@Primary**

Indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency.

@Configuration  
public class MovieConfiguration {  
 @Bean  
 @Primary  
 public MovieCatalog firstMovieCatalog() { ... }  
 @Bean  
 public MovieCatalog secondMovieCatalog() { ... }  
// ...  
}

The corresponding bean definitions:

<bean class="example.SimpleMovieCatalog" primary="true">  
 <!-- inject any dependencies required by this bean -->  
</bean>  
<bean class="example.SimpleMovieCatalog">  
 <!-- inject any dependencies required by this bean -->  
</bean>  
<bean id="movieRecommender" class="example.MovieRecommender"/>

**Fine-tuning annotation-based autowiring with qualifiers**

@Primary is an effective way to use autowiring by type with several instances when one primary candidate can be determined. When more control over the selection process is required, spring’s @Qualifier annotation can be used.

@Autowired  
@Qualifier("main")  
private MovieCatalog movieCatalog;

@Autowired  
public void prepare(@Qualifier("main")MovieCatalog movieCatalog,  
 CustomerPreferenceDao customerPreferenceDao) {  
 this.movieCatalog = movieCatalog;  
 this.customerPreferenceDao = customerPreferenceDao;  
}

The corresponding bean definitions appear as follows:

<bean class="example.SimpleMovieCatalog">  
<qualifier value="main"/>  
<!-- inject any dependencies required by this bean -->  
</bean>  
<bean class="example.SimpleMovieCatalog">  
<qualifier value="action"/>  
<!-- inject any dependencies required by this bean -->  
</bean>

Advice: if you intend to express annotation-driven injection by name, do not primarily use @Autowired. Instead, use @Resource annotation, which is semantically defined to identify a specific target component by its unique name.

**@Resource(name=””)**

@PostConstruct and @PreDestroy

## Classpath scanning and managed components

**@component and further stereotype annotations**

The @Repository annotation is a marker for any class that fulfills the role or stereotype of a repository (DAO). Among the uses of this marker is the automatic translation of exceptions.

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

The use of <context:component-scan > implicitly enables the functionality of <context:annotation-config >.

**Using filters to customize scanning**

By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components. However, you can modify and extend this behavior simply by applying custom filters. Add them as includeFilters or excludeFilters parameters of the @ComponentScan annotation (or as include-filter or exclude-filter sub-elements of the compontent-scan element).

@Configuration  
@ComponentScan(basePackages = "org.example",  
 includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),  
 excludeFilters = @Filter(Repository.class))  
public class AppConfig {  
 ...  
}

And the equivalent using XML

<context:component-scan base-package="org.example">  
<context:include-filter type="regex"  
 expression=".\*Stub.\*Repository"/>  
<context:exclude-filter type="annotation"  
 expression="org.springframework.stereotype.Repository"/>  
</context:component-scan>

**Defining bean metadata within components**

This is same as the definition within @Configuration

@Component  
public class FactoryMethodComponent {  
 private static int *i*;  
 @Bean  
 @Qualifier("public")  
 public TestBean publicInstance() {  
 return new TestBean("publicInstance");  
 }  
 // use of a custom qualifier and autowiring of method parameters  
 @Bean  
 protected TestBean protectedInstance(  
 @Qualifier("public") TestBean spouse,  
 @Value("#{privateInstance.age}") String country) {  
 TestBean tb = new TestBean("protectedInstance", 1);  
 tb.setSpouse(spouse);  
 tb.setCountry(country);  
 return tb;  
 }  
 @Bean  
 private TestBean privateInstance() {  
 return new TestBean("privateInstance", *i*++);  
 }  
 @Bean  
 @RequestScope  
 public TestBean requestScopedInstance() {  
 return new TestBean("requestScopedInstance", 3);  
 }  
}

The example autowires the String method parameter country to the value of the Age property on another bean named privateInstance. A Spring Expression Language element defines the value of the property through the notation #{<expression>}. For @Value annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.

However, invoking a method or field in @Bean method within a plain @Component class has standard Java semantics, with no CGLIB processing or other constraints applying. In contrast, @Bean method within a @Configuration enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within @Bean methods in @Configuration classes creates bean metadata references to collaborating objects; such methods are not invoked with normal Java semantics but rather go through the container in order to provide the usual lifecycle management and proxying of Spring beans even when referring to other beans via programmatic calls to @Bean method.

使用@Configuration 的 @Bean方法，即使普通调用，也相当于通过Spring容器调用，而@Component中的@Bean方法，只是一个普通的Java调用。在@Configuration的@Bean方法会被覆盖，因此@Bean方法不能是private或者final。

### Naming autodetected components

When a component is autodetected as part of the scanning process, its bean name is generated by the BeanNameGenerator strategy known to that scanner.

If annotation (@Component, @Repository, @Service, and @Controller) contains no name value or for any other detected component, the default bean name generator returns the uncapitalized non-qualified class name.

@Service("myMovieLister") // name will be myMovieLister  
public class SimpleMovieLister {  
// ...  
}  
  
@Repository // name will be movieFinderImpl:  
public class MovieFinderImpl implements MovieFinder {  
// ...  
}

If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the BeanNameGenerator interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner.

@Configuration  
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)  
public class AppConfig {  
 ...  
}

The XML configuration

<context:component-scan base-package="org.example"  
 name-generator="org.example.MyNameGenerator" />

### Providing a scope for autodetected components

Simple provide the name of scope within the annotation:

**@Scope("prototype")**  
@Repository  
public class MovieFinderImpl implements MovieFinder {  
// ...  
}

@Qualifier should be review.

## Java-based container configuration

### Basic concepts: @Bean and @Configuration

The @Bean annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container.

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions. Furthermore, @Configuration classes allow inter-bean dependencies to be defined by simply calling other @Bean methods in the same class.

**Full @Configuration vs ‘lite’ @Bean mode**

When @Bean methods are declared within classes that are not annotated with @Configuration they are referred to as being processed in a ‘lite’ mode.

Unlike full @Configuration, lite @Bean methods cannot easily declare inter-bean dependencies. Usually one @Bean method should not invoke another @Bean method when operating the ‘lite’ mode.

Only using @Bean methods within @Configuration classes is recommended approach of ensuring that ‘full’ mode is always used. This will prevent the same @Bean method from accidentally being invoked multiple times and helps to reduce subtle bugs that can be hard to track down when operating in ‘lite’ mode.

### Instantiating the Spring container using AnnotationConfigApplicationContext

This allows for completely XML-free usage of the Spring container. AnnotationConfigApplicationContext is not limited to working only with @Configuration classes. Any @Component or JSR-330 annotated class may be supplied as input to the constructor.

### Support for web applications with AnnotationConfigWebApplicationContext

A WebApplicationContext variant of AnnotationConfigApplicationContext is available with AnnotationConfigWebApplicationContext. This implementation may be used when configuring the Spring ContextLoaderListener servlet listener, Spring MVC DispatcherServlet, etc. What follows is a web.xml snippet that configures a typical Spring MVC web application.

<web-app>  
 <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext  
 instead of the default XmlWebApplicationContext -->  
 <context-param>  
 <param-name>contextClass</param-name>  
 <param-value>  
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext  
 </param-value>  
 </context-param>  
 <!-- Configuration locations must consist of one or more comma- or space-delimited  
 fully-qualified @Configuration classes. Fully-qualified packages may also be  
 specified for component-scanning -->  
 <context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>com.acme.AppConfig</param-value>  
 </context-param>  
 <!-- Bootstrap the root application context as usual using ContextLoaderListener -->  
 <listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
 </listener>  
 <!-- Declare a Spring MVC DispatcherServlet as usual -->  
 <servlet>  
 <servlet-name>dispatcher</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext  
 instead of the default XmlWebApplicationContext -->  
 <init-param>  
 <param-name>contextClass</param-name>  
 <param-value>  
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext  
 </param-value>  
 </init-param>  
 <!-- Again, config locations must consist of one or more comma- or space-delimited  
 and fully-qualified @Configuration classes -->  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>com.acme.web.MvcConfig</param-value>  
 </init-param>  
 </servlet>  
 <!-- map all requests for /app/\* to the dispatcher servlet -->  
 <servlet-mapping>  
 <servlet-name>dispatcher</servlet-name>  
 <url-pattern>/app/\*</url-pattern>  
 </servlet-mapping>  
</web-app>

### Using the @Bean annotation

@Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports some of the attributes offered by <bean />, such as: init-method, destroy-method, autowiring and name.

You can use the @Bean annotation in a @Configuration or a @Component class.

### Declaring a bean

To declare a bean, simple annotate a method with the @Bean annotation. You use this method to register a bean definition within an ApplicationContext of the type specified as the method’s return value. By default, the bean name will be the same as the method name.

### Bean dependencies

A @Bean annotated method can have an arbitrary number of parameters describing the dependencies required to build that bean. (I think accountRepository is a bean).

@Configuration  
public class AppConfig {  
 @Bean  
 public TransferService transferService(AccountRepository accountRepository) {  
 return new TransferServiceImpl(accountRepository);  
 }  
}

Specify initMethod or destroyMethod

public class Foo {  
 public void init() {  
// initialization logic  
 }  
}  
public class Bar {  
 public void cleanup() {  
// destruction logic  
 }  
}  
@Configuration  
public class AppConfig {  
 @Bean(initMethod = "init")  
 public Foo foo() {  
 return new Foo();  
 }  
 @Bean(destroyMethod = "cleanup")  
 public Bar bar() {  
 return new Bar();  
 }  
}

Specifying bean scope

Use the @Scope annotation specify your beans with a specific scope.

Like @Scope(“prototype”)

Customizing bean naming

Use @Bean(name = “myFoo”) to specify a bean’ name, or @Bean(name={“dataSource”, “subsystem-dataSource”}) to use bean aliasing.

### Using the @Configuration annotation

@Configuration is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans via public annotated methods.

### Using the @Import annotation

Much as the <import /> element is used within Spring XML files to aid in modularizing configurations, the @Import annotation allows for loading @Bean definitions from another configuration class.

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

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied explicitly:

public static void main(String[] args) {  
 ApplicationContext ctx = new

AnnotationConfigApplicationContext(ConfigB.class);  
// now both beans A and B will be available...  
 A a = ctx.getBean(A.class);  
 B b = ctx.getBean(B.class);  
}

### Combining java and XML configuration

@ImportResource annotation to import XML as needed.

XML-centric use of @Configuration classes

Create @Configuration classes on an as-needed basis and include them from the existing XML files.

@Configuration  
public class AppConfig {  
 @Autowired  
 private DataSource dataSource;  
 @Bean  
 public AccountRepository accountRepository() {  
 return new JdbcAccountRepository(dataSource);  
 }  
 @Bean  
 public TransferService transferService() {  
 return new TransferService(accountRepository());  
 }  
}

XML will like:

<beans>  
<!-- picks up and registers AppConfig as a bean definition -->  
<context:component-scan base-package="com.acme"/>  
<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>  
<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
</bean>  
</beans>

### @Configuration class-centric use of XML with @ImportResource

@Configuration  
@ImportResource("classpath:/com/acme/properties-config.xml")  
public class AppConfig {  
 @Value("${jdbc.url}")  
 private String url;  
 @Value("${jdbc.username}")  
 private String username;  
 @Value("${jdbc.password}")  
 private String password;  
 @Bean  
 public DataSource dataSource() {  
 return new DriverManagerDataSource(url, username, password);  
 }  
}

XML configuration:

<beans>  
<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>  
</beans>

### Environment abstraction

A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active.

Properties play an important role in almost all applications, and may originate form a variety of sources: properties files, JVM system environment variables, servlet context parameters, Maps, and so on.

#### Bean definition profiles

Bean definition profiles is a mechanism in the core container that allows for registration of different beans in different environments.

To switch between two different variations based on the current environment, usually relying on a combination of system environment variables and XML <import /> statements containing @{placeholder} tokens that resolve to the correct configuration file path depending on the value of an environment variable.

#### @Profile

The @Profile annotation allows you to indicate that a component is eligible for registration when one or more specified profiles are active.

@Configuration  
@Profile("dev")  
public class StandaloneDataConfig {  
 @Bean  
 public DataSource dataSource() {  
 return new EmbeddedDatabaseBuilder()  
 .setType(EmbeddedDatabaseType.HSQL)  
 .addScript("classpath:com/bank/config/sql/schema.sql")  
 .addScript("classpath:com/bank/config/sql/test-data.sql")  
 .build();  
 }  
}

Another

@Configuration  
@Profile("production")  
public class JndiDataConfig {  
 @Bean(destroyMethod="")  
 public DataSource dataSource() throws Exception {  
 Context ctx = new InitialContext();  
 return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");  
 }  
}

#### Activation a profile

Activating a profile can be done in several ways, but the most straightforward is to do it programmatically against the Environment API which is available via an ApplicationContext:

AnnotationConfigApplicationContext ctx = new

AnnotationConfigApplicationContext();  
**ctx.getEnvironment().setActiveProfiles("dev");**  
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);  
ctx.refresh();

In addition, profiles may also be activated declaratively through the spring.profiles.active property which may be specified through system environment variables, JVM system properties, servlet context parameters in web.xml.

#### @PropertySource

The @PropertySource annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring’s Environment.

Any ${...} placeholders present in a @PropertySource resource location will be resolved against the set of property sources already registered against the environment.

@Configuration  
@PropertySource("**classpath:/com/${my.placeholder:default/path}/app.properties**")  
public class AppConfig {  
 @Autowired  
 Environment env;  
 @Bean  
 public TestBean testBean() {  
 TestBean testBean = new TestBean();  
 testBean.setName(env.getProperty("testbean.name"));  
 return testBean;  
 }  
}

Assuming that “my.placeholder” is present in one of the property sources already registered, e.g system properties or environment variables, the placeholder will be resolved to the corresponding value. If not, the “default/path” will be used as default.

### Registering a LoadTimeWeaver

The LoadTimeWeaver is used by Spring to dynamically transform classes as they are loaded into the JVM.

### Standard and Custom Events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. If a bean that implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified. Essentially, this is a standard Observer design pattern.

### Convenient ApplicationContext instantiation for web application

You can create ApplicationContext instances declaratively by using, for example, a ContextLoader. Of course you can also create ApplicationContext instances programmatically by using one of the ApplicationContext implementations.

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>  
</context-param>  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>

The listener inspects the contextConfigLocation parameter. If the parameter does not exist, the listener uses /WEB-INF/applicationContext.xml as a default. When the parameter does exist, the listener separates the String by using predefined delimiters (comma, semicolon and whitespace) and uses the values as locations where application contexts will be searched. Ant-style path patterns are supported as well. Examples are /WEB-INF/\*Context.xml for all files with names ending with "Context.xml", residing in the "WEB-INF" directory, and /WEB-INF/\*\*/\*Context.xml, for all such files in any subdirectory of "WEB-INF".

## Resources

The InputStream cannot be read multiple times, and must be read once only and then closed to avoid resource leaks.

“classpath\*:conf/appContext.xml”: a location string may use the special

classpath\*: prefix. The special prefix specifies that all classpath resources that match the given name must be obtained, and then merged to form the final application context definition.

## Validation, Data Binding, and Type Conversion

Data binding is useful for allowing user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-called DataBinder to do exactly that. The validator and the DataBinder make up the Validation package, which is primarily used in but not limited to the MVC framework.

public class CustomerValidator {  
 private final Validator addressValidator;  
 public CustomerValidator(Validator addressValidator) {  
 if (addressValidator == null) {  
 throw new IllegalArgumentException("The supplied [Validator] is " +  
 "required and must not be null.");  
 }  
 if (!addressValidator.supports(Address.class)) {  
 throw new IllegalArgumentException("The supplied [Validator] must " +  
 support the validation of [Address] instances.");  
 }  
 this.addressValidator = addressValidator;  
 }  
 */\*\*  
 \* This Validator validates Customer instances, and any subclasses of Customer too  
 \*/* public boolean supports(Class clazz) {  
 return Customer.class.isAssignableFrom(clazz);  
 }  
 public void validate(Object target, Errors errors) {  
 ValidationUtils.*rejectIfEmptyOrWhitespace*(errors, "firstName", "field.required");  
 ValidationUtils.*rejectIfEmptyOrWhitespace*(errors, "surname", "field.required");  
 Customer customer = (Customer) target;  
 try {  
 errors.pushNestedPath("address");  
 ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);  
 } finally {  
 errors.popNestedPath();  
 }  
 }  
}

## Resolving views

可以使用order来指定view resolver的顺序。Order值越小，越先被使用。

重定向：**redirect:** prefix allows you to accomplish this. If a view name is returned that has the prefix redirect:, the UrlBasedViewResolver will recognize this as a special indication that a redirect is needed.

使用**redirect:** 浏览器的地址会改变，**forward:** 浏览器的地址不会改变。