Spring Framework Note

IoC Container and Beans

Inversion of Control

Inversion of Control(a.k.a dependency injection) is a process whereby objects define their dependencies only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean.

`org.springframework.beans`
`org.springframework.context`

are basis for Spring Framework's IoC container.

BeanFactory provides configuration mechanism, the **configuration framework** and basic functionality, capable of managing any type of object. *ApplicationContext* extends BeanFactory, which adds:

- Easier integration with Spring's AOP
- Message resource handling(for use in internationalization)
- Applicatioon-layer specific contexts

ApplicationContext adds more **enterprise-specific** functionality.

Beans

An object that is instantiated, assembled and managed by a Spring IoC container.

Beans, and dependencies among them are reflected in the configuration metadata used by a container.

IoC Container

ApplicationContext represents the Spring IoC container and is responsible for manage beans. It is instructions by reading configuration metadata, represented in XML, java annotations or java code.

Instances of beans can be retrieved from the ApplicationContext using getBean. Yet, it should not be used in the Spring project.

Bean Overview

Inside of the container, there are *BeanDefinition* objects which contains the metadata of beans:

- A package-qualified class name: typically the actual implementation class of the bean being defined.

- Bean behavioral configuration elements, e.g. scope, lifecycle callbacks and so forth.
- Reference to other beans, a.k.a collaborators or dependencies
- Other configuration settings, the arguments needed for setting a object.

The external objects can also be registrated. getBeanFactory() can retrieve AplicationContext's BeanFactory, the *DefaultListableBeanFactory. It can register through* registerSingleton()* and *registerBeanDefinition().*

The class attribute(which is a Class property on a *BeanDefinition*) is usually mandatory.

You can instantiate a bean with:

- Constructor
- Static Factory Method: Specify the class's name.
- Instance Factory Method: similar to static one. However, since the bean was initialized by another bean inside of the container, you need to specify its name.

Dependencies

The container injects dependancies when it creates the bean. This process is fundamentally the inverse of the bean itself controlling the instantiation or location of its dependencies on its own.

Dependancy injection makes the code cleaner, more decoupled.

Dependency Resolution Process

1. ApplicationContext created and initialized with configuration metadata that describes all the beans.
2. Bean's dependencies are specified in properties, constructor arguments or arguments to the static factory method. Dependencies are provided to the bean, when the bean is actually created.
3. Each property or arugments is an actually definition of the value to set, or a reference to another bean in the container.
4. Each property or constructor argument that is a value is converted from its specified format to the actual type of that property or constructor argument.

Circular Dependencies: to solve circular dependencies, one of the object should be configured by setters rather than constructors.

Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. The container loaded correctly can later generate an

exception. Hence, ApplicationContext implementations by default pre-instatiate singleton beans.

If there is no circular dependencies, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being inject.

Annotation-based Container Configuration

Annotation injection is performed before XML injection. Thus, the XML configuration overrides the annotations for properties wired throuh both approaches.

@Required

Required anothation applies to bean property setter methods, which indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. However, assertion is still preferable since it is also effective outside of the container.

@Autowired

@Inject annotation can be used to replace @Autowired

If @Autowired added to a field or method that expects an array of that type, Spring will provide all beans of that type from the ApplicationContext.

Annotated methods and fiedls are treated required by default, but setting required field to false make it non-required. Also, you can use Optional or @Nullable to denote that.

@Autowire, @Inject, @Value and @Resource annotations are handled by Spring BeanPostProcessor implimentations. So, those annotations cannot be applied in BeanPostProcess or BeanFactoryPostProcessor.

@Primary, @Qualifier can be used to tuning the priority.

For @annotation-driven injection by name, use @Resouce instead, which is semantically defined to identify a specific targe component by its unique name with decalred type being irrelevant for matching.

@Resource

@Resouece takes a name attribute, and Spring interprets that value as the bean name to be injected.

The exacution paths of @Resource is:

1. Match by Name
2. Mathc by Type
3. Match by Qualifier

@Value

Inject externalized properties.

```
@Component
public class MovieRecommender {
    private final String Catalog;

    public MovieRecommender(@Value("${catalog.name}") String catalog) {
        this.catalog = catalog;
    }
}
```

Configuration:

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig { }
```

To maintain strict control over nonexistent values, a PropertySourcesPlaceholderConfigurer bean can be declared, and it should be static.

Classpath Scanning

@Filter

Extend and modify behavior by applying custom filters. Each filter element requires the **type** and **expression** attributes. There are four types: annotation(default), assignable, aspectj, regex, custom.

@Named and @ManagedBean

@Named and @ManagedBean can be used as replacement for @Component.

Java-based Container Configuration

@Bean and @Configuration

`@Bean` is used to indicate that a method instantiates, configures and initializes a new **object** to be managed by the Spring IoC container. `@Bean` can be with any Spring `@Component`, but used with `@Configuration` in most cases.

`@Configuration` indicates that the class's primary purpose is as a source of bean definitions.

`@Bean` used without `@Configuration` will define a lite bean, where the proxy is not applied(cross-method references get redirected to the containers' lifecycle management).

Using `@AnnotationConfigApplicationContext`

`ApplicaionContext` can accept both `@Configuration` and `@Component` classes.

`@Configuration`: the class itself is regitered as a bean definition and all decalred `@Bean` methods within the class are also registered as bean definitions.

`@Component`(Or other JSR-330 classes): itself is registered as bean definitions, and it is assumed that DI metadata are used.

Using `@Bean`

`@Bean` is a method-level annotation of the XML `<bean/>`. When you annotate a method with it, you are using this method to register a bean definition within an ApplicationContext. By default, its type will be the method's return type and its name will be the same as the method name.

Arguments of the method will be regarded as dependencies.

Using `@Configuration`

`@Configuration` is a class level annotaion indicating an object is a source of bean definitions. It declares beans through public `@Bean` annotated methods.

Environment Abstraction

There are two key aspects of the application environment: **profiles** and **properties**.

Profile

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

The @Profile string may contain a simple profile name or a profile expression, which contains logic expression of profiles. (Must have a parentheses if mix the & and |)

@Profile can also be declared at the method level to include specific bean of a configuration class.

In the case of overloaded @Bean methods of the same Java method name, @Profile confition needs to be consistently declared on all overloaded methods. If not, the first method will be picked.

Activate a Profile

To activate a profile you can:

- Using Environment API which is available through ApplicationContext.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();
```

- Using property `spring.profiles.active`, which accept several arguments seperated by comma.

Default Profile

Profile definition for one or more beans if no profile is active. If any profiles is enabled, the default profiles does not apply.

```
@Profile("default")
```

It can be configured using `setDefaultProfiles()` in the Environment or through `spring.profiles.default` property.

Property