2.2. @Bean

@Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports most of the attributes offered by <bean/>, such as: init-method, destroy-method, autowiring, lazy-init, dependency-check, depends-on and scope.

2.2.1. Declaring a bean

To declare a bean, simply annotate a method with the @Bean annotation. When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a BeanFactory. By default, the bean name will be the same as the method name (see bean naming for details on how to customize this behavior). The following is a simple example of a @Bean method declaration:

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

For comparison sake, the configuration above is exactly equivalent to the following Spring XML:

```
<beans>
    <bean name="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

Both will result in a bean named transferService being available in the BeanFactory / ApplicationContext, bound to an object instance of type TransferServiceImpl:

```
transferService -> com.acme.TransferServiceImpl
```

2.2.2. Injecting dependencies

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another:

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        return new Foo(bar());
    }

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

In the example above, the `foo` bean recevies a reference to `bar` via constructor injection.

2.2.3. Receiving lifecycle callbacks

2.2.3.1. Using JSR-250 annotations

JavaConfig, like the core Spring Framework, supports use of JSR-250 "Common Annotations". For example:

```
public class FooService {
    @PostConstruct
    public void init() {
        // custom initialization logic
    }
}

@Configuration
@AnnotationDrivenConfig
public class ApplicationConfig {
    @Bean
    public FooService fooService() {
        return new FooService();
    }
}
```

In the above example, `FooService` declares `@PostConstruct` . By declaring JavaConfig's `@AnnotationDrivenConfig` on The `@Configuration` class, this annotation will be respected by the container and called immediately after construction. See The core framework documentation on support for JSR-250 annotations for further details.

2.2.3.2. Using Spring interfaces

Spring's lifecycle callbacks are fully supported. If a bean implements `InitializingBean`, `DisposableBean`, or `Lifecycle`, their respective methods will be called by the container in accordance with their Javadoc.

2.2.3.3. Using `@Bean` `initMethodName` / `destroyMethodName` attributes

The `@Bean` annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's `init-method` and `destroy-method` attributes to the `bean` element:

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

Of course, in the case of `Foo` above, it would be equally as valid to call the `init()` method directly during construction:

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.init();
        return foo;
    }

    // ...
}
```

Tip

Remember that because you are working directly in Java, you can do anything you like with your objects, and do not always need to rely on the container!

2.2.4. Using `*Aware` interfaces

The standard set of `*Aware` interfaces such as [`BeanFactoryAware`], [`BeanNameAware`], [`MessageSourceAware`], [`ApplicationContextAware`], etc. are fully supported. Consider an example class that implements `BeanFactoryAware`:

```
public class AwareBean implements BeanFactoryAware {

    private BeanFactory factory;

    // BeanFactoryAware setter (called by Spring during bean instantiation)
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.factory = beanFactory;
    }

    public void close(){
        // do clean-up
    }
}
```

If the class above were declared as a bean as follows:

```
@Configuration
public class AppConfig {
    @Bean
    public AwareBean awareBean() {
        return new AwareBean();
    }
}
```

its `setBeanFactory` method will be called during initialization, providing the bean with access to its enclosing `BeanFactory`.

2.2.5. Specifying bean scope

2.2.5.1. Using `@Bean`'s `scope` attribute

JavaConfig makes available each of the four standard scopes specified in Section 3.4, "Bean Scopes" of the Spring reference documentation.

The `DefaultScopes` class provides string constants for each of these four scopes. SINGLETON is the default, and can be overridden by supplying the `scope` attribute to `@Bean` annotation:

```
@Configuration
public class MyConfiguration {
    @Bean(scope=DefaultScopes.PROTOTYPE)
    public Encryptor encryptor() {
        // ...
    }
}
```

2.2.5.2. `@ScopedProxy`

Spring offers a convenient way of working with scoped dependencies through scoped proxies. The easiest way to create such a proxy when using the XML configuration is the `<aop:scoped-proxy/>` element. JavaConfig offers equivalent support with the `@ScopedProxy` annotation, which provides the same semantics and configuration options.

If we were to port the the XML reference documentation scoped proxy example (see link above) to JavaConfig, it would look like the following:

```
// a HTTP Session-scoped bean exposed as a proxy
@Bean(scope = DefaultScopes.SESSION)
@ScopedProxy
public UserPreferences userPreferences() {
   return new UserPreferences();
}

@Bean
public Service userService() {
   UserService service = new SimpleUserService();
   // a reference to the proxied 'userPreferences' bean
   service.seUserPreferences(userPreferences());
   return service;
}
```

2.2.5.3. Lookup method injection

As noted in the core documentation, lookup method injection is an advanced feature that should be comparatively rarely used. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. JavaConfig provides a natural means for implementing this pattern. *Note that the example below is adapted from the example classes and configuration in the core documentation linked above.*

```
package fiona.apple;

public abstract class CommandManager {
    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();

        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

JavaConfig can easily create a subclass of `CommandManager` where the abstract `createCommand()` is overridden in such a way that it 'looks up' a brand new (prototype) command object:

```
@Bean(scope=DefaultScopes.PROTOTYPE)
public AsyncCommand asyncCommand() {
    AsyncCommand command = new AsyncCommand();
    // inject dependencies here as required
    return command;
}

@Bean
public CommandManager commandManager() {
    // return new anonymous implementation of CommandManager with command() overridde.
    // to return a new prototype Command object
    return new CommandManager() {
        protected Command command() {
            return asyncCommand();
        }
    }
}
```

2.2.6. Customizing bean naming

By default, JavaConfig uses a `@Bean` method's name as the name of the resulting bean. This functionality can be overridden, however, using the `BeanNamingStrategy` extension point.

```
public class Main {
    public static void main(String[] args) {
        JavaConfigApplicationContext ctx = new JavaConfigApplicationContext();
        ctx.setBeanNamingStrategy(new CustomBeanNamingStrategy());
        ctx.addConfigClass(MyConfig.class);
        ctx.refresh();
        ctx.getBean("customBeanName");
    }
}
```

Note

`JavaConfigApplicationContext` will be covered in detail in Chapter 3, *Using @Configuration classes*

For more details, see the API documentation for `BeanNamingStrategy`.

2.2.7. Working with Spring `FactoryBean` implementations

Spring provides many implementations of the `FactoryBean` interface. Usually these classes are used to support integrations with other frameworks. Take for example org.springframework.orm.hibernate3.LocalSessionFactoryBean. This class is used to create a Hibernate SessionFactory and requires as dependencies the location of Hibernate mapping files and a DataSource. Here's how it is commonly used in XML:

```
<beans>
    <bean id="sessionFactory"
          class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="mappingResources">
             <list>
                 <value>com/acme/Bank.hbm.xml</value>
                 <value>com/acme/Account.hbm.xml</value>
                 <value>com/acme/Customer.hbm.xml</value>
             </list>
        </property>
    </bean>

    <bean id="dataSource" class="...">
        <!-- ... -->
    </bean>
</beans>
```

The Spring container recognizes that LocalSessionFactoryBean implements the FactoryBean interface, and thus treats this bean specially: An instance of LocalSessionFactoryBean is instantiated, but instead of being directly returned, instead the getObject() method is invoked. It is the object returned from this call getObject() that is ultimately registered as the sessionFactory bean.

How then would we use LocalSessionFactoryBean in JavaConfig? The best approach is to extend the ConfigurationSupport base class and use the getObject() method:

```
@Configuration
public class DataAccessConfig extends ConfigurationSupport {
    @Bean
    public SessionFactory sessionFactory() {
        LocalSessionFactoryBean factoryBean = new LocalSessionFactoryBean();
        factoryBean.setDataSource(dataSource());
        ArrayList<String> mappingFiles = new ArrayList<String>();
        mappingFiles.add("com/acme/Bank.hbm.xml");
        mappingFiles.add("com/acme/Account.hbm.xml");
        mappingFiles.add("com/acme/Customer.hbm.xml");
        factoryBean.setMappingResources(mappingFiles);
        return this.getObject(SessionFactory.class, factoryBean);
    }
    // ... other beans, including dataSource() ...
}
```

Notice the call to this.getObject(Class, FactoryBean)? This call ensures that any container callbacks are invoked on the FactoryBean object, and then returns the value from the FactoryBean's getObject() in a type-safe fashion.