Spring JavaConfig Reference Guide

Version 1.0.0.m3

About this document

1. Document structure

Part I, “Introduction” Introduces JavaConfig, including a quick start tutorial. *The impatient should start here!*

Part II, “API Reference” Provides a feature-by feature walk through the JavaConfig API

Part III, “Appendices” Covers tooling and other practical considerations not well-suited for the core documentation

2. Target audience

This document assumes at least a general familiarity with the Spring IoC container. If you are brand new to Spring (i.e.: have never used Spring's XML configuration language, it is suggested that you first read Chapter 3. IoC of the Core Spring reference manual documentation. You will also notice that this chapter is frequently referenced throughout this document.

3. Code conventions

Code artifacts will be referred to in `fixed-with` font.

Preface to the third milestone release

We're proud to announce this third milestone (M3) of the JavaConfig project. Much has happened in the long months since M2, including a healthy dose of user feedback. The good news is that we've learned users seem to like JavaConfig and the programming model works well for most needs. Thanks to everyone that has participated in the process thus far, including Rick Hightower, Solomon Duskis, Ben Rowlands, Craig Walls, Jim Moore, and many others.

M3 includes 20+ bug fixes and feature additions. See Chapter 2, *New & Noteworthy in 1.0.0.m3* for details. As a special note to existing users, M3 introduces one fairly major change to the existing public API - the deprecation of `AnnotationApplicationContext` in favor of its replacement, `JavaConfigApplicationContext`. `AnnotationApplicationContext` will remain deprecated in the codebase up until the 1.0 GA release, at which time it will be deleted permanently. So take heed and swap out references as soon as you can.

Future milestones and RCs on the path to 1.0 may include other breaking API changes. The core public API (annotations, application contexts, and key util classes) will not likely change radically, but internal APIs almost certainly will. It is important to the team that the packaging structure and internals of JavaConfig are 'just right' in order to facilitate flexible evolution and solid backward compatibility post-1.0.

Thanks for taking a look at JavaConfig! Your feedback will drive future revisions, so please don't hesitate to contact the team via JIRA issue tracking with anything you see missing.

Part I. Introduction

In Part I we explain what JavaConfig is, how it fits into the larger Spring configuration landscape, and make the case as to why it is a compelling option. After reading this part you'll be ready to create a basic configuration and evaluate JavaConfig for yourself. Finally, you'll be ready to move on to Part II and explore the JavaConfig API in detail.

Chapter 1. Overview

1.1. What is JavaConfig?

Spring JavaConfig is a product of the Spring community that provides a pure-Java approach to configuring the Spring IoC Container. While JavaConfig aims to be a feature-complete option for configuration, it can be (and often is) used in conjunction with the more well-known XML-based configuration approach.

See Section 8.1, “JavaConfig and XML” for more information on using JavaConfig and XML together

1.2. Why JavaConfig?

The Spring IoC Container is the leading dependency injection framework. It provides sophisticated dependency injection capabilities as well as advanced features for aspect-oriented programming. Today the majority of Spring users configure the Spring container using XML. This works very well, and in many cases is ideal. However, some developers find configuring the Spring container via Java a more natural or appropriate approach, for a variety of reasons.

Motivations of JavaConfig include:

Object-oriented configuration. Because configurations are defined as classes in JavaConfig, users can take full advantage of object-oriented features in Java. One configuration class may subclass another, overriding its `@Bean` methods, etc.

Reduced or eliminated XML configuration. The benefits of externalized configuration based on the principles of dependency injection have been proven. However, many developers would prefer not to switch back and forth between XML and Java. JavaConfig provides developers with a pure-Java approach to configuring the Spring container that is conceptually similar to XML configuration. It is technically possible to configure the container using only JavaConfig configuration classes, however in practice many have found it ideal to mix-and-match JavaConfig with XML. See Section 8.1, “JavaConfig and XML” for details.

Type-safe and refactoring-friendly. JavaConfig provides a type-safe approach to configuring the Spring container. Thanks to Java 5.0's support for generics, it is now possible to retrieve beans *by type* rather than by name, free of any casting or string-based lookups. See Section 4.3.2.1, “Type-safe access” for details.

1.3. Requirements

1.3.1. Runtime Environment

JavaConfig takes full advantage of Java 5.0 language features, especially annotations and generics. *A Java 5.0+ runtime environment is a requirement for using JavaConfig.*. For an introduction to annotations and other Java 5.0 features, see http://java.sun.com/docs/books/tutorial/java/javaOO/annotations.html and http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#lang

1.3.2. Project Dependencies

Every effort has been made to reduce the number of dependencies Spring JavaConfig requires. For very simple

configurations, the dependencies are limited to CGLIB, commons-logging, spring-core, spring-beans, and spring-context. If you use more advanced features such as AOP, you'll need aspectj, spring-aop, etc. Maven2 use is recommended, as it helps greatly in reducing the burden of dependency management.

See also: Appendix C, *Maven2 POM configurations*

1.4. History

JavaConfig was first conceived in 2005, at which time initial code was laid down. While it has remained in pre-1.0 status since that time, it has enjoyed a fair bit of use and positive user feedback. JavaConfig is now a fully supported effort and is moving toward 1.0 release.

Chapter 2. New & Noteworthy in 1.0.0.m3

2.1. `AnnotationApplicationContext` deprecated

`AnnotationApplicationContext` presented a naming conflict with Spring 2.5's Annotation-Driven Configuration. To avoid this problem it has been renamed to `JavaConfigApplicationContext`. Beyond the renaming, `JavaConfigApplicationContext` has several new features worth discussing. See below for details. *`AnnotationApplicationContext` will be removed entirely in JavaConfig 1.0.0.rc1*

See Section 4.3, “ `JavaConfigApplicationContext` ”

2.2. Type-safety improvements

`JavaConfigApplicationContext`, `JavaConfigWebApplicationContext` and the `ConfigurationSupport` base class now all expose type-safe `getBean` methods, allowing for looking up beans by type, rather than by name.

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

    @Bean
    public NotificationService notificationService() {
        return new NotificationServiceImpl();
    }
}

public class SimpleApp {
    public static void main(String... args) {
        JavaConfigApplicationContext context = new JavaConfigApplicationContext(AppConfig.class);

        // use the type-safe getBean method to avoid casting and string-based lookups
        TransferService transferService = context.getBean(TransferService.class);
        TransferService notificationService = context.getBean(NotificationService.class);

        // ...
    }
}
```

See Section 4.3.2.1, “Type-safe access” for details.

2.3. First-class support for JavaConfig in the web tier

By popular demand, a `WebApplicationContext` variant of `JavaConfigApplicationContext` has been created. This allows for seamless bootstrapping of JavaConfig bean definitions within web.xml requiring no Spring XML whatsoever.

```
<web-app>
    <!-- Bootstrap the root application context as usual using ContextLoaderListener -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <!-- Configure ContextLoaderListener to use JavaConfigWebApplicationContext
         instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>org.springframework.config.java.JavaConfigWebApplicationContext</param-value>
    </context-param>
```

```
    <!-- Configuration locations must consist of one or more comma- or space-delimited
         fully-qualified @Configuration classes -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>example.RootApplicationConfig</param-value>
    </context-param>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
        <servlet-name>test</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- Configure DispatcherServlet to use JavaConfigWebApplicationContext
             instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.config.java.JavaConfigWebApplicationContext</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>example.web.WebBeansConfig</param-value>
        </init-param>
    </servlet>
</web-app>
```

For basic information regarding initialization parameters to `DispatcherServlet` and use of `ContextLoaderListener`, see Chapter 13. Web MVC framework in the Core Spring Framework documentation.

For complete details on using JavaConfig in the web tier, see Chapter 7, *Developing web applications with JavaConfig*.

2.4. New semantics for nested `@Configuration` classes

Consider the case of nested `@Configuration` classes:

```
@Configuration
public class OuterConfig {
    @Bean
    public Foo outerBean() {
        // ...
    }
}

@Configuration
public static class InnerConfig {
    @Bean
    public Bar innerBean() {
        // ...
    }
}
```

In previous milestones, any nested `@Configuration` classes (such as `InnerConfig` above) were treated as just another source of bean definitions and were processed inline with the rest of the beans in the declaring `@Configuration` class. Ultimately, all bean definitions ended up in the same `BeanFactory`/`ApplicationContext`.

Now and going forward, nested `@Configuration` classes will be processed as child `ApplicationContexts`. Perhaps better said, any declaring (outer) `@Configuration` classes will be processed as parent `ApplicationContexts`. Using the example above, Instantiate an `ApplicationContext` using `InnerConfig` as an argument

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(InnerConfig.class);
context.getBean("innerBean"); // locally defined beans are available
context.getBean("outerBean"); // as are beans defined in the declaring OuterConfig class.
```

Note that when supplying `OuterConfig` as the argument, `InnerConfig` is ignored entirely. If it were to be processed, it would become a child context, but its beans would would be inaccessible (parent contexts have no access to child context beans).

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(OuterConfig.class);
context.getBean("outerBean"); // works fine
context.getBean("innerBean"); // throws NoSuchBeanDefinitionException!
```

See Section 5.6, “Nesting `@Configuration` classes” for full details.

2.5. Modularization improvements with `@Import`

JavaConfig now has the equivalent of XML configuration's `<import/>` element. One configuration class can import any number of other configuration classes, and their bean definitions will be processed as if locally defined.

```
@Configuration
public class DataSourceConfig {
    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(...);
    }
}

@Configuration
@Import(DataSourceConfig.class)
public class AppConfig extends ConfigurationSupport {
    @Bean
    public void TransferService transferService() {
        return new TransferServiceImpl(getBean(DataSource.class);
    }
}
```

Importing multiple configurations can be done by supplying an array of classes to the `@Import` annotation

```
@Configuration
@Import({ DataSourceConfig.class, TransactionConfig.class })
public class AppConfig extends ConfigurationSupport {
    // bean definitions here can reference bean definitions in DataSourceConfig or TransactionConfig
}
```

See ??? for full details.

2.6. Support for externalizing string values with `@ExternalValue`

Important: This functionality may change!. We are currently evaluating approaches for best supporting externalized values. See SJC-74 for more details.

Somewhat similar to the way that `@ExternalBean` makes it possible to reference beans defined outside of the current `@Configuration`, `@ExternalBean` allows for accessing externalized string values in properties files. This is ideal for use when configuring up infrastructural resources with properties that may change at deployment time, or otherwise need to be externally accessible for modification.

```
@Configuration
@ResourceBundles("classpath:/com/myapp/datasource")
public abstract class AppConfig {
    @Bean
    public DataSource myDataSource() {
        DataSource dataSource = new MyDataSource();
        dataSource.setUsername(username());
        dataSource.setPassword(password());
```

```
        dataSource.setUrl(url());

        return dataSource;
    }

    @ExternalValue
    public abstract String username();

    @ExternalValue
    public abstract String password();

    @ExternalValue("jdbc.url")
    public abstract String url();
}
```

`com/myapp/datasource.properties`:

```
username=scott
password=tiger
jdbc.url=...
```

See Section 5.5, “Externalizing values with `@ExternalValue` and `@ResourceBundles`” for full details

Chapter 3. Quick start

This chapter provides a basic tutorial for getting started with JavaConfig. For full details on JavaConfig's capabilities, please refer to Part II, “API Reference”

3.1. Download JavaConfig

Like any Java library, you'll first need to get the JavaConfig jar (and the jars it depends on) into your classpath.

3.1.1. Manual download

Please visit http://www.springframework.org/javaconfig where you'll find links to zip file distributions. Three distributions are available:

1. `spring-javaconfig-1.0.0.m3.zip`
2. `spring-javaconfig-1.0.0.m3-with-minimal-dependencies.zip`
3. `spring-javaconfig-1.0.0.m3-with-dependencies.zip`

The `-with-minimal-dependencies` zip contains only those jars that are required for basic JavaConfig functionality, while the `-with-dependencies` zip contains all dependencies, including those required for AOP support.

3.1.2. Maven 2

Assuming your project uses a Maven2 build infrastructure, using JavaConfig is as simple as adding the following to your POM

```
<dependencies>
    <dependency>
        <groupId>org.springframework.javaconfig</groupId>
        <artifactId>spring-javaconfig</artifactId>
        <version>1.0.0.m3</version>
    </dependency>
</dependencies>
```

Note

Please note that this release is not published at the central Maven repository. Instead it is published on Amazon's S3 service, like all Spring milestones. To use it, add the following repository to your POM:

```
<repository>
    <id>spring-milestone</id>
    <name>Spring Milestone Repository</name>
    <url>http://s3.amazonaws.com/maven.springframework.org/milestone</url>
</repository>
```

Tip

See Appendix C, *Maven2 POM configurations* for more information about using Maven2 with Spring JavaConfig

3.2. Create bean definitions

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

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(...);
    }
}
```

3.3. Retrieve bean instances

Let's create a very basic, command-line application to allow users to transfer money from one account to another.

```
public class SimpleTransferApplication {
    public static void main(String... args) {
        double amount = new Double(args[0]);
        int sourceAcctId = new Integer(args[1]);
        int destAcctId = new Integer(args[2]);

        JavaConfigApplicationContext context = new JavaConfigApplicationContext(ApplicationConfig.class);
        TransferService transferService = context.getBean(TransferService.class);
        transferService.transfer(300.00, sourceAcctId, destAccountId);
    }
}
```

3.4. Summary

That's it! You're now ready for Part II, “API Reference”, where you'll walk through each of the features of Spring JavaConfig.

Part II. API Reference

This part of the Reference Guide explains the core functionality of Spring JavaConfig

Chapter 4, *Creating and using bean definitions* describes the fundamental concepts in Spring JavaConfig: the `@Configuration` and `@Bean` annotations, and gives the reader information on how to use them.

Chapter 5, *Modularizing configurations*. For non-trivial uses of Spring JavaConfig, users will want the ability to modularize configurations for reuse, readability and deployment concerns. This section documents the various techniques and features available for meeting these needs.

Chapter 6, *Using aspects*. One of the most powerful features of the Spring IoC container is the ability to add aspect-oriented behavior to plain beans using Spring AOP. JavaConfig also makes the use of aspects possible, and this section details how to do it.

Chapter 7, *Developing web applications with JavaConfig*. Documents JavaConfig's first-class support for use within the web tier with `JavaConfigWebApplicationContext`.

Chapter 8, *Combining configuration approaches*. Describes how to use JavaConfig's `ConfigurationPostProcessor` bean post-processor for integrating `@Configuration`-annotated classes within an XML configuration file.

Chapter 4. Creating and using bean definitions

4.1. `@Configuration`

Annotating a class with the `@Configuration` annotation indicates that the class will be used by JavaConfig as a source of bean definitions.

An application may make use of just one `@Configuration`-annotated class, or many. `@Configuration` can be considered the equivalent of XML's `<beans/>` element. Like `<beans/>`, it provides an opportunity to explicitly set defaults for all enclosed bean definitions.

```
@Configuration(defaultAutowire = Autowire.BY_TYPE, defaultLazy = Lazy.FALSE)
public class DataSourceConfiguration {
    // bean definitions follow
}
```

Because the semantics of the attributes to the `@Configuration` annotation are 1:1 with the attributes to the `<beans/>` element, this documentation defers to the beans-definition section of Chapter 3, IoC from the Core Spring documentation.

4.2. `@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`.

4.2.1. 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 that of the method name (see bean naming for details on how to customize this behavior).

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

The above is exactly equivalent to the following `appConfig.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.TransferService
```

See Section 4.3, “ `JavaConfigApplicationContext` ” for details about instantiating and using an `ApplicationContext` with JavaConfig.

4.2.2. 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
    }
}
```

Also, the lifecycle callback methods are fully supported.

4.2.3. Bean visibility

A feature unique to JavaConfig feature is *bean visibility*. JavaConfig uses standard Java method visibility modifiers to determine if the bean ultimately returned from a method can be accessed by an owning application context / bean factory.

Consider the following configuration:

```
@Configuration
public abstract class VisibilityConfiguration {

  @Bean
  public Bean publicBean() {
     Bean bean = new Bean();
     bean.setDependency(hiddenBean());
     return bean;
  }

  @Bean
  protected HiddenBean hiddenBean() {
     return new Bean("protected bean");
  }

  @Bean
  HiddenBean secretBean() {
     Bean bean = new Bean("package-private bean");
     // hidden beans can access beans defined in the 'owning' context
     bean.setDependency(outsideBean());
  }

  @ExternalBean
  public abstract Bean outsideBean()
}
```

Let's bootstrap the above configuration within a traditional XML configuration (for more information on mixing configuration strategies see Chapter 8, *Combining configuration approaches*). The application context being instantiated agaist the XML file will be the 'owning' or 'enclosing' application context, and will not be able to 'see' the hidden beans:

```
<beans>
 <!-- the configuration above -->
 <bean class="my.java.config.VisibilityConfiguration"/>

 <!-- Java Configuration post processor -->
```

```
 <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

 <bean id="mainBean" class="my.company.Bean">
    <!-- this will work -->
    <property name="dependency" ref="publicBean"/>
    <!-- this will *not* work -->
    <property name="anotherDependency" ref="hiddenBean"/>
 </bean>
</beans>
```

As JavaConfig encounters the `VisibilityConfiguration` class, it will create 3 beans : `publicBean`, `hiddenBean` and `secretBean`. All of them can see each other however, beans created in the 'owning' application context (the application context that bootstraps JavaConfig) will see only `publicBean`. Both `hiddenBean` and `secretBean` can be accessed only by beans created inside `VisibilityConfiguration`.

Any `@Bean` annotated method, which is not `public` (i.e. with `protected` or default visibility), will create a 'hidden' bean. Note that due to technical limitations, `private` `@Bean` methods are not supported.

In the example above, `mainBean` has been configured with both `publicBean` and `hiddenBean`. However, since the latter is (as the name imply) hidden, at runtime Spring will throw:

```
org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'hiddenBean' is defined
 ...
```

To provide the visibility functionality, JavaConfig takes advantage of the application context hierarchy provided by the Spring container, placing all hidden beans for a particular configuration class inside a child application context. Thus, the hidden beans can access beans defined in the parent (or owning) context but not the other way around.

4.2.4. Bean scoping

4.2.4.1. 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() {
        // ...
    }
}
```

4.2.4.2. `@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 as alternative 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;
}
```

4.2.4.3. Method injection

As noted in the Core documentation, method injection is an advanced feature that should be comparatively rarely used. When using XML configuration, it is required in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. In JavaConfig, however, it is a (somewhat) simpler proposition:

```
@Bean
public MyAbstractSingleton mySingleton(){
    return new MyAbstractSingleton(myDependencies()){
        public MyPrototype createMyPrototype(){
            return new MyPrototype(someOtherDependency());
            // or alternatively return myPrototype() -- this is some @Bean or @ExternalBean method...
        }
    }
}
```

4.2.5. 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.

```
<beans>
    <bean class="org.springframework.config.java.process.ConfigurationPostProcessor">
        <property name="namingStrategy">
            <bean class="my.custom.NamingStrategy"/>
        </property>
    </bean>
</beans>
```

Note

Overriding the bean naming strategy is currently only supported by XML configuration of `ConfigurationPostProcessor`. In future revisions, it will be possible to specify BeanNamingStrategy directly on `JavaConfigApplicationContext`. Watch SJC-86 for details.

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

For more information on integrating JavaConfig and XML, see Chapter 8, *Combining configuration approaches*

4.3. `JavaConfigApplicationContext`

`JavaConfigApplicationContext` provides direct access to the beans defined by `@Configuration`-annotated classes. For more information on the ApplicationContext API in general, please refer to the Core Spring

documentation.

4.3.1. Construction Options

Instantiating the `JavaConfigApplicationContext` can be done by supplying `@Configuration`-annotated class literals to the constructor, and/or strings representing packages to scan for `@Configuration`-annotated classes.

4.3.1.1. Construction by class literal

Each of the class literals supplied to the constructor will be processed, and for each `@Bean`-annotated method encountered, JavaConfig will create a bean definition and ultimately instantiate and initialize the bean.

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(AppConfig.class);
Service service = context.getBean(Service.class);
```

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(AppConfig.class, DataConfig.class);
Service service = context.getBean(Service.class);
```

4.3.1.2. Construction by base package

Base packages will be scanned for the existence of any `@Configuration`-annotated classes. Any candidate classes will then be processed much as if they had been supplied directly as class literals to the constructor.

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext("**/configuration/**/*.class");
Service service = (Service) context.getBean("serviceA");
```

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext("**/configuration/**/*.class", "**/other/*Config.class);
Service service = (Service) context.getBean("serviceA");
```

4.3.1.3. Post-construction configuration

When one or more classes/packages are used during construction, a `JavaConfigApplicationContext` cannot be further configured. If post-construction configuration is preferred or required, use either the no-arg constructor, configure by calling setters, then manually refresh the context. After the call to `refresh()`, the context will be 'closed for configuration'.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext();
context.setParent(otherConfig);
context.setConfigClasses(AppConfig.class, DataConfig.class);
context.setBasePackages("**/configuration/**/*.class");
context.refresh();
Service service = (Service) context.getBean("serviceA");
```

Note

Whenever multiple packages and/or classes are used to instantiate a `JavaConfigApplicationContext`, *order matters*. This is important when considering what happens if two configuration classes define a bean with the same name. The last-specified class wins.

4.3.2. Accessing beans with `getBean()`

`JavaConfigApplicationContext` provides several variants of the `getBean()` method for accessing beans.

4.3.2.1. Type-safe access

The preferred method for accessing beans is with the type-safe `getBean()` method.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = context.getBean(Service.class);
```

4.3.2.1.1. Disambuguation options

If more than one bean of type `Service` had been defined in the example above, the call to `getBean()` would have thrown an exception indicating an ambiguity that the container could not resolve. In these cases, the user has a number of options for disambiguation:

4.3.2.1.1.1. Indicating a `@Bean` as `primary`

Like Spring's XML configuration, JavaConfig allows for specifying a given `@Bean` as `primary`:

```
@Configuration
public class MyConfig {
    @Bean(primary=Primary.TRUE)
    public Service myService() {
        return new Service();
    }

    @Bean
    public Service backupService() {
        return new Service();
    }
}
```

After this modification, all calls to `getBean(Service.class)` will return the `primary` bean

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
// returns the myService() primary bean
Service service = context.getBean(Service.class);
```

4.3.2.1.1.2. Disambiguation by bean name

JavaConfig provides a `getBean()` variant that accepts both a class and a bean name for cases just such as these.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = context.getBean(Service.class, "myService");
```

Because bean ids must be unique, this call guarantees that the ambiguity cannot occur.

4.3.2.1.1.3. Retrieve all beans of a given type

It is also reasonable to call the `getBeansOfType()` method in order to return all beans that implement a given interface:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Map matchingBeans = context.getBeansOfType(Service.class);
```

Note that this latter approach is actually a feature of the Core Spring Framework's `AbstractApplicationContext` (which `JavaConfigApplicationContext` extends) and is not type-safe, in that the returned `Map` is not parameterized.

4.3.2.2. String-based access

Beans may be accessed via the traditional string-based `getBean()` API as well. Of course this is not type-safe and requires casting, but avoids any potential ambiguity entirely:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = (Service) context.getBean("myService");
```

Chapter 5. Modularizing configurations

While the simplest configuration may be expressed as a single class that exposes several beans, it is often desirable to *modularize* configurations for reuse and clarity.

5.1. Partitioning bean definitions across multiple `@Configuration` classes

The simplest technique for modularizing configurations is to simply split up large `@Configuration` classes containing many `@Bean` definitions into multiple smaller classes:

```
// monolithic configuration
@Configuration
public class AppConfig {
    @Bean
    public ServiceA serviceA() {
       // ...
    }

    @Bean
    public ServiceB serviceB() {
       // ...
    }

   // assume many bean definitions follow
}
```

The above configuration class might be supplied as a parameter to `JavaConfigApplicationContext`:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(AppConfig.class);
ServiceA serviceA = context.getBean(ServiceA.class);
ServiceB serviceB = context.getBean(ServiceB.class);
```

We can easily partition this configuration such that bean definitions are spread across two classes, instead of one:

```
// partitioned configuration
@Configuration
public class AppConfigA {
    @Bean
    public ServiceA serviceA() {
       // ...
    }
}

@Configuration
public class AppConfigB {
    @Bean
    public ServiceB serviceB() {
       // ...
    }
}
```

Now simply supply both configuration classes to the constructor of `JavaConfigApplicationContext`:

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(AppConfigA.class, AppConfigB.class);
// both beans are still available in the resulting application context
ServiceA serviceA = context.getBean(ServiceA.class);
ServiceB serviceB = context.getBean(ServiceB.class);
```

5.2. Referencing externally defined beans with `@ExternalBean`

One configuration class may need to reference a bean defined in another configuration class (or in XML, for that matter). The `@ExternalBean` annotation provides just such a mechanism. When JavaConfig encounters a method annotated as `@ExternalBean`, it replaces that method definition with a lookup to the enclosing bean factory for a bean with the same name as the method name.

```
@Configuration
public class ConfigOne {
    @Bean
    public AccountRepository accountRepository() {
        // create and return an AccountRepository object
    }
}

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

    @ExternalBean
    public abstract AccountRepository accountRepository();
}
```

Given that both these configuration classes are supplied to the application context at runtime, JavaConfig will be able to resolve the '`accountRepository`' `@ExternalBean` by name and everything will 'wire-up' accordingly.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(ConfigOne.class, ConfigTwo.class);
```

5.3. Importing `@Configuration` classes with `@Import`

`@Import` represents JavaConfig's equivalent of XML configuration's `<import/>` element. One configuration class can import any number of other configuration classes, and their bean definitions will be processed as if locally defined.

```
@Configuration
public class DataSourceConfig {
    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(...);
    }
}

@Configuration
@Import(DataSourceConfig.class)
public class AppConfig extends ConfigurationSupport {
    @Bean
    public void TransferService transferService() {
        return new TransferServiceImpl(getBean(DataSource.class);
    }
}
```

Importing multiple configurations can be done by supplying an array of classes to the `@Import` annotation

```
@Configuration
@Import({ DataSourceConfig.class, TransactionConfig.class })
public class AppConfig extends ConfigurationSupport {
    // bean definitions here can reference bean definitions in DataSourceConfig or TransactionConfig
}
```

5.4. `ConfigurationSupport`

As a convenience, `@Configuration` classses can extend `ConfigurationSupport`, primarily in order to facilitate easy lookup of beans from the enclosing `BeanFactory` / `ApplicationContext`.

```
@Configuration
public class AppConfig extends ConfigurationSupport {
    @Bean
    public Service serviceA() {
        DataSource dataSource = this.getBean(DataSource.class); // provided by base class
        // ...
    }
}
```

5.5. Externalizing values with `@ExternalValue` and `@ResourceBundles`

Tip

Important: This functionality may change! We are currently evaluating approaches for best supporting externalized values. See SJC-74 for more details.

What about `PropertyOverrideConfigurer`?

Those familiar with XML configuration will notice that there is not a direct equivalent for the popular `PropertyOverrideConfigurer`. However, the combination of `@ResourceBundles` and `@ExternalValue` is roughly equivalent to the use of the `PropertyPlaceholderConfigurer`. Just as many users do not wish to store usernames, passwords and other sensitive or environment-specific values in XML, neither would we want to do so in Java configurations. Interestingly, the functionality of `PropertyOverrideConfigurer` is not hard to emulate, either. Simply give the method a default return value.

Access externally defined string values such as usernames, passwords, and the like using `@ResourceBundles` and `@ExternalValue`.

```
@ResourceBundles("classpath:/org/springframework/config/java/simple")
@Configuration
public abstract static class ConfigWithPlaceholders {
    @ExternalValue("datasource.username")
    public abstract String username();

    @Bean
    public TestBean testBean() {
        return new TestBean(username());
    }
}
```

The '`org/springframework/config/java/simple`' argument to `@ResourceBundles` tells JavaConfig that there must be a file in the classpath at `org/springframework/config/java/simple[locale].[properties|xml]` from which key/value pairs can be read. For example:

```
datasource.username=scott
datasource.password=tiger
...
```

Multiple values can be supplied to `@ResourceBundles`:

```
@ResourceBundles({"classpath:/com/foo/myapp/simple", "classpath:/com/foo/myapp/complex"})
@Configuration
public class AppConfig {
    // ...
}
```

5.6. Nesting `@Configuration` classes

Consider the case of nested `@Configuration` classes:

```
@Configuration
public class OuterConfig {
    @Bean
    public Foo outerBean() {
        // ...
    }

    @Configuration
    public static class InnerConfig {
        @Bean
        public Bar innerBean() {
            // ...
        }
    }
}
```

Nested `@Configuration` classes will be processed as child `ApplicationContexts`. Perhaps better said, any declaring (outer) `@Configuration` classes will be processed as parent `ApplicationContexts`. Using the example above, Instantiate an `ApplicationContext` using `InnerConfig` as an argument

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(InnerConfig.class);
context.getBean("innerBean"); // locally defined beans are available
context.getBean("outerBean"); // as are beans defined in the declaring OuterConfig class.
```

Note that when supplying `OuterConfig` as the argument, `InnerConfig` is ignored entirely. If it were to be processed, it would become a child context, but its beans would would be inaccessible (parent contexts have no access to child context beans).

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(OuterConfig.class);
context.getBean("outerBean"); // works fine
context.getBean("innerBean"); // throws NoSuchBeanDefinitionException!
```

Chapter 6. Using aspects

AOP support in JavaConfig is a work in progress. It is documented here in order to solicit feedback. *Expect changes in forthcoming milestone releases*

6.1. Embedded aspects

Tip

For those unfamiliar with AOP and/or Spring AOP, you'll want to take a look Chapter 6 of the core Spring reference documentation, Aspect-oriented Programming with Spring.

A configuration class can serve 'double duty' as an aspect. By applying `@Aspect` to a `@Configuration` class, you can then add pointcuts and advice that will be applied against all beans in the container.

```
@Aspect
@Configuration
public class AppConfig {
    @Bean
    public Service service() {
        return new ServiceImpl(...);
    }

    @Before("execution(* service..Service+.set*(*))")
    public void trackServicePropertyChange() {
        logger.info("property changed on service!");
    }
}
```

This pointcut will match the all methods starting with 'set' on all implementations of the Service interface. Before any matching methods execute, the `trackServicePropertyChange()` method will be executed.

6.2. Reusable aspects

To create a reusable aspect, define a class annotated with `@Configuration` and `Aspect` containing advice methods and pointcuts.

```
@Aspect
@Configuration
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(PropertyChangeTracker.class);

    @Before("execution(* set*(*))")
    public void trackChange() {
        logger.info("property just changed...");
    }
}
```

Then include that aspect in the application context, either using the @Import annotation or by adding at as a constructor argument when instantiating `JavaConfigApplicationContext`. Examples of both approaches are below.

```
@Configuration
@Import(PropertyChangeTracker.class)
@Aspect // necessary in order to have transferService bean get aspects applied!
public class MyConfig {
    @Bean
    public TransferService myService() {
        return new TransferServiceImpl(...);
```

```
    }
}

...

JavaConfigApplicationContext context = new JavaConfigApplicationContext(MyConfig.class);
```

or...

```
@Aspect // necessary in order to have transferService bean get aspects applied!
@Configuration
public class MyConfig {
    @Bean
    public TransferService myService() {
        return new TransferServiceImpl(...);
    }
}

...

JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(MyConfig.class, PropertyChangeTracker.class);
```

Note

Annotating Configuration classes with `@Aspect` to enable AOP functionality is not consistent with the standard semantics around the `@Aspect` annotation, and would thus likely be un-intuitive to someone familiar with XML configuration. *This approach to applying aspects will change prior to JavaConfig's 1.0 GA release*. See SJC-55 for details.

Chapter 7. Developing web applications with JavaConfig

JavaConfig provides the `JavaConfigWebApplicationContext` class for bootstrapping your configurations into the web tier.

7.1. `JavaConfigWebApplicationContext`

`JavaConfigWebApplicationContext` allows for seamlessly bootstrapping JavaConfig bean definitions within your servlet container's web.xml. This process requires no Spring XML bean definitions whatsoever:

```
<web-app>
    <!-- Bootstrap the root application context as usual using ContextLoaderListener -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <!-- Configure ContextLoaderListener to use JavaConfigWebApplicationContext
         instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>org.springframework.config.java.JavaConfigWebApplicationContext</param-value>
    </context-param>
    <!-- Configuration locations must consist of one or more comma- or space-delimited
         fully-qualified @Configuration classes -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>example.RootApplicationConfig</param-value>
    </context-param>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
        <servlet-name>test</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- Configure DispatcherServlet to use JavaConfigWebApplicationContext
             instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.config.java.JavaConfigWebApplicationContext</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>example.web.WebBeansConfig</param-value>
        </init-param>
    </servlet>
</web-app>
```

For basic information regarding initialization parameters to `DispatcherServlet` and use of `ContextLoaderListener`, see Chapter 13. Web MVC framework in the Core Spring Framework documentation.

Chapter 8. Combining configuration approaches

JavaConfig can be used in conjunction with any or all of Spring's other container configuration approaches. The question is when and where it's appropriate to do so.

8.1. JavaConfig and XML

8.1.1. Bootstrapping JavaConfig from XML with `ConfigurationPostProcessor`

Currently, to use JavaConfig and XML config together, the configuration needs to be 'XML-driven', meaning that it will be the XML configuration that bootstraps JavaConfig.

```
<beans>
   <!-- first, define your individual @Configuration classes as beans -->
   <bean class="com.myapp.config.AppConfig"/>
   <bean class="com.myapp.config.DataConfig"/>

    <!-- be sure to include the JavaConfig bean post-processor -->
    <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>
</beans>
```

Then, bootstrap an XML ApplicationContext:

```
ApplicationContext context = new ClassPathXmlApplicationContext("application-config.xml");
```

The beans defined in `AppConfig` and `DataConfig` will be available via '`context`'.

8.1.1.1. Configuring configurations

An added benefit that comes along with bootstrapping JavaConfig from XML is that the configuration bean instances are eligible, just as any other bean, for configuration:

```
<beans>
    <!-- a possible "configurable configuration" -->
    <bean class="org.my.company.config.AppConfiguration">
        <property name="env" value="TESTING"/>
        <property name="monitoring" value="true"/>
        <property name="certificates" value="classpath:/META-INF/config/MyCompany.certs"/>
    </bean>
    <!-- JavaConfig post-processor -->
    <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>
</beans>
```

Part III. Appendices

Appendix A. Roadmap

See the JavaConfig JIRA roadmap for up-to date information

Appendix B. Visualizing configurations with Spring IDE

Spring IDE supports JavaConfig. At the time of this writing, however, support is limited to configurations that are 'bootstrapped' via XML (see Section 8.1, “JavaConfig and XML”). Stay tuned to Spring IDE's javaconfig.core component in JIRA for updates on progress. Follow (link to Spring IDE jira) for updates on progress here. Click for screenshot

Appendix C. Maven2 POM configurations

As mentioned in Chapter 3, *Quick start*, JavaConfig requires only a minimal set of dependencies for basic (non-AOP) configuration scenarios.

Note

Please note that this release is not published at the central Maven repository. Instead it is published on Amazon's S3 service, like all Spring milestones. To use it, add the following repository to your POM:

```
<repository>
  <id>spring-milestone</id>
  <name>Spring Milestone Repository</name>
  <url>http://s3.amazonaws.com/maven.springframework.org/milestone</url>
</repository>
```

C.1. Minimal

This minimal configuration will work for all JavaConfig features that do not involve AOP.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.javaconfig</groupId>
        <artifactId>spring-javaconfig</artifactId>
        <version>1.0.0.m3</version>
    </dependency>
</dependencies>
```

C.2. AOP

In an effort to keep the number of dependencies required when using JavaConfig to a minimum, AOP-related dependencies are excluded by default. Should you wish to use AOP with JavaConfig, modify your Maven2 POM configuration to include the following:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.javaconfig</groupId>
        <artifactId>spring-javaconfig</artifactId>
        <version>1.0.0.m3</version>
    </dependency>
    <dependency>
        <groupId>asm</groupId>
        <artifactId>asm</artifactId>
        <version>2.2.3</version>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>asm</groupId>
        <artifactId>asm-commons</artifactId>
        <version>2.2.3</version>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>aspectj</groupId>
        <artifactId>aspectjlib</artifactId>
        <version>1.5.3</version>
        <optional>true</optional>
```

```
    </dependency>
    <dependency>
        <groupId>aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>1.5.3</version>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.5.3</version>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Appendix D. Additional resources

D.1. Core Spring documentation

The Core Spring documentation is mentioned throughout this document, and it is recommended that JavaConfig users read and understand the IoC chapter of the reference guide. Although it discusses XML, its treatment of fundamental concepts remains directly applicable to JavaConfig.

D.2. Community forums

The first place to stop with questions regarding SJC is http://forums.springframework.org. As of this writing, Spring JavaConfig does not have its own dedicated forum, but you can post questions to 'Core Container' for the time being.

D.3. Issue tracking

After checking out the forums, any bugs, improvements or new features can be submitted via the Spring Framework issue tracker at http://jira.springframework.org/browse/SJC

D.4. SpringSource team blog

Keep up to date with JavaConfig progress by watching the SpringSource Team Blog at http://blog.springsource.com

D.5. Consulting and training services

Spring JavaConfig development is sponsored by SpringSource, which specializes in consuling, training and support for the Spring Portfolio of projects. Contact sales@springsource.com for details.