Spring	javax.inject.*	javax.inject restrictions / comments
ObjectFactory	Provider	`javax.inject.Provider` is a direct alternative to Spring's `ObjectFactory`, just with a shorter `get()` method name. It can also be used in combination with Spring's `@Autowired` or with non-annotated constructors and setter methods.

7.12 Java-based container configuration

Basic concepts: @Bean and @Configuration

The central artifacts in Spring's new Java-configuration support are `@Configuration`-annotated classes and `@Bean`-annotated methods.

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. For those familiar with Spring's `<beans/>` XML configuration the `@Bean` annotation plays the same role as the `<bean/>` element. You can use `@Bean` annotated methods with any Spring `@Component`, however, they are most often used with `@Configuration` beans.

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. The simplest possible `@Configuration` class would read as follows:

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }

}
```

The `AppConfig` class above would be equivalent to the following Spring `<beans/>` XML:

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

Full @Configuration vs 'lite' @Beans 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. For example, bean methods declared in a `@Component` or even in a *plain old class* will be considered 'lite'.

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

> Only using `@Bean` methods within `@Configuration` classes is a 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.

The `@Bean` and `@Configuration` annotations will be discussed in depth in the sections below. First, however, we'll cover the various ways of creating a spring container using Java-based configuration.

Instantiating the Spring container using AnnotationConfigApplicationContext

The sections below document Spring's `AnnotationConfigApplicationContext`, new in Spring 3.0. This versatile `ApplicationContext` implementation is capable of accepting not only `@Configuration` classes as input, but also plain `@Component` classes and classes annotated with JSR-330 metadata.

When `@Configuration` classes are provided as input, the `@Configuration` class itself is registered as a bean definition, and all declared `@Bean` methods within the class are also registered as bean definitions.

When `@Component` and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as `@Autowired` or `@Inject` are used within those classes where necessary.

Simple construction

In much the same way that Spring XML files are used as input when instantiating a `ClassPathXmlApplicationContext`, `@Configuration` classes may be used as input when instantiating an `AnnotationConfigApplicationContext`. This allows for completely XML-free usage of the Spring container:

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

As mentioned above, `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. For example:

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(MyServiceImpl.class,
 Dependency1.class, Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

The above assumes that `MyServiceImpl`, `Dependency1` and `Dependency2` use Spring dependency injection annotations such as `@Autowired`.

Building the container programmatically using register(Class<?>…)

An `AnnotationConfigApplicationContext` may be instantiated using a no-arg constructor and then configured using the `register()` method. This approach is particularly useful when programmatically building an `AnnotationConfigApplicationContext`.

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Enabling component scanning with scan(String…)

To enable component scanning, just annotate your `@Configuration` class as follows:

```
@Configuration
@ComponentScan(basePackages = "com.acme")
public class AppConfig  {
    ...
}
```

> **Tip**
>
> Experienced Spring users will be familiar with the XML declaration equivalent from Spring's `context:` namespace
>
> ```
> <beans>
>     <context:component-scan base-package="com.acme"/>
> </beans>
> ```

In the example above, the `com.acme` package will be scanned, looking for any `@Component`-annotated classes, and those classes will be registered as Spring bean definitions within the container. `AnnotationConfigApplicationContext` exposes the `scan(String…)` method to allow for the same component-scanning functionality:

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

> **Note**
>
> Remember that `@Configuration` classes are meta-annotated with `@Component`, so they are candidates for component-scanning! In the example above, assuming that `AppConfig` is declared within the `com.acme` package (or any package underneath), it will be picked up during the call to `scan()`, and upon `refresh()` all its `@Bean` methods will be processed and registered as bean definitions within the container.

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. Note the use of the `contextClass` context-param and init-param:

```
<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`-annotated or in a `@Component`-annotated class.

Declaring a bean

To declare a bean, simply 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. The following is a simple example of a `@Bean` method declaration:

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

}
```

The preceding configuration is exactly equivalent to the following Spring XML:

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

Both declarations make a bean named `transferService` available in the `ApplicationContext`, bound to an object instance of type `TransferServiceImpl`:

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

Bean dependencies

A `@Bean` annotated method can have an arbitrary number of parameters describing the dependencies required to build that bean. For instance if our `TransferService` requires an `AccountRepository` we can materialize that dependency via a method parameter:

```
@Configuration
public class AppConfig {

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

}
```

The resolution mechanism is pretty much identical to constructor-based dependency injection, see the relevant section for more details.

Receiving lifecycle callbacks

Any classes defined with the `@Bean` annotation support the regular lifecycle callbacks and can use the `@PostConstruct` and `@PreDestroy` annotations from JSR-250, see JSR-250 annotations for further details.

The regular Spring lifecycle callbacks are fully supported as well. If a bean implements `InitializingBean`, `DisposableBean`, or `Lifecycle`, their respective methods are called by the container.

The standard set of `*Aware` interfaces such as BeanFactoryAware, BeanNameAware, MessageSourceAware, ApplicationContextAware, and so on are also fully supported.

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

```
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();
    }

}
```

Note

By default, beans defined using Java config that have a public `close` or `shutdown` method are automatically enlisted with a destruction callback. If you have a public `close` or `shutdown` method and you do not wish for it to be called when the container shuts down, simply add `@Bean(destroyMethod="")` to your bean definition to disable the default `(inferred)` mode.

You may want to do that by default for a resource that you acquire via JNDI as its lifecycle is managed outside the application. In particular, make sure to always do it for a `DataSource` as it is known to be problematic on Java EE application servers.

```
@Bean(destroyMethod="")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

Also, with `@Bean` methods, you will typically choose to use programmatic JNDI lookups: either using Spring's `JndiTemplate`/`JndiLocatorDelegate` helpers or straight JNDI `InitialContext` usage, but not the `JndiObjectFactoryBean` variant which would force you to declare the return type as the `FactoryBean` type instead of the actual target type, making it harder to use for cross-reference calls in other `@Bean` methods that intend to refer to the provided resource here.

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**
>
> When you work directly in Java, you can do anything you like with your objects and do not always need to rely on the container lifecycle!

Specifying bean scope

Using the @Scope annotation

You can specify that your beans defined with the `@Bean` annotation should have a specific scope. You can use any of the standard scopes specified in the Bean Scopes section.

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

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }

}
```

@Scope and scoped-proxy

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. Configuring your beans in Java with a @Scope annotation offers equivalent support with the proxyMode attribute. The default is no proxy (`ScopedProxyMode.NO`), but you can specify `ScopedProxyMode.TARGET_CLASS` or `ScopedProxyMode.INTERFACES`.

If you port the scoped proxy example from the XML reference documentation (see preceding link) to our `@Bean` using Java, it would look like the following:

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
public UserPreferences userPreferences() {
    return new UserPreferences();
}

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

Customizing bean naming

By default, configuration classes use a `@Bean` method's name as the name of the resulting bean. This functionality can be overridden, however, with the `name` attribute.

```
@Configuration
public class AppConfig {

    @Bean(name = "myFoo")
    public Foo foo() {
        return new Foo();
    }

}
```

Bean aliasing

As discussed in the section called "Naming beans", it is sometimes desirable to give a single bean multiple names, otherwise known as*bean aliasing*. The `name` attribute of the `@Bean` annotation accepts a String array for this purpose.

```
@Configuration
public class AppConfig {

    @Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }

}
```

Bean description

Sometimes it is helpful to provide a more detailed textual description of a bean. This can be particularly useful when beans are exposed (perhaps via JMX) for monitoring purposes.

To add a description to a `@Bean` the `@Description` annotation can be used:

```
@Configuration
public class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    public Foo foo() {
        return new Foo();
    }

}
```

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 `@Bean` annotated methods. Calls to `@Bean` methods on `@Configuration` classes can also be used to define inter-bean dependencies. See the section called "Basic concepts: @Bean and @Configuration" for a general introduction.

Injecting inter-bean 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 receives a reference to `bar` via constructor injection.

> **Note**
>
> This method of declaring inter-bean dependencies only works when the `@Bean` method is declared within a `@Configuration` class. You cannot declare inter-bean dependencies using plain `@Component` classes.

Lookup method injection

As noted earlier, lookup method injection is an advanced feature that you should use rarely. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. Using Java for this type of configuration provides a natural means for implementing this pattern.

```
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();
}
```

Using Java-configuration support , you can create a subclass of `CommandManager` where the abstract `createCommand()` method is overridden in such a way that it looks up a new (prototype) command object:

```
@Bean
@Scope("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() overridden
    // to return a new prototype Command object
    return new CommandManager() {
        protected Command createCommand() {
            return asyncCommand();
        }
    }
}
```

Further information about how Java-based configuration works internally

The following example shows a `@Bean` annotated method being called twice:

```
@Configuration
public class AppConfig {

    @Bean
    public ClientService clientService1() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientService clientService2() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientDao clientDao() {
        return new ClientDaoImpl();
    }

}
```

`clientDao()` has been called once in `clientService1()` and once in `clientService2()`. Since this method creates a new instance of `ClientDaoImpl` and returns it, you would normally expect having 2 instances (one for each service). That definitely would be problematic: in Spring, instantiated beans have a `singleton` scope by default. This is where the magic comes in: All `@Configuration` classes are subclassed at startup-time with `CGLIB`. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance. Note that as of Spring 3.2, it is no longer necessary to add CGLIB to your classpath because CGLIB classes have been repackaged under org.springframework and included directly within the spring-core JAR.

> **Note**
>
> The behavior could be different according to the scope of your bean. We are talking about singletons here.

> **Note**
>
> There are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time:
>
> - Configuration classes should not be final
> - They should have a constructor with no arguments

Composing Java-based configurations

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);
}
```

This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring the developer to remember a potentially large number of `@Configuration` classes during construction.

Injecting dependencies on imported @Bean definitions

The example above works, but is simplistic. In most practical scenarios, beans will have dependencies on one another across configuration classes. When using XML, this is not an issue, per se, because there is no compiler involved, and one can simply declare `ref="someBean"` and trust that Spring will work it out during container initialization. Of course, when using `@Configuration` classes, the Java compiler places constraints on the configuration model, in that references to other beans must be valid Java syntax.

Fortunately, solving this problem is simple. As we already discussed, `@Bean` method can have an arbitrary number of parameters describing the bean dependencies. Let's consider a more real-world scenario with several `@Configuration` classes, each depending on beans declared in the others:

```
@Configuration
public class ServiceConfig {

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

}

@Configuration
public class RepositoryConfig {

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

}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

There is another way to achieve the same result. Remember that `@Configuration` classes are ultimately just another bean in the container: This means that they can take advantage of `@Autowired` and `@Value` injection etc just like any other bean!

> **Warning**
>
> Make sure that the dependencies you inject that way are of the simplest kind only. `@Configuration` classes are processed quite early during the initialization of the context and forcing a dependency to be injected this way may lead to unexpected early initialization. Whenever possible, resort to parameter-based injection as in the example above.
>
> Also, be particularly careful with `BeanPostProcessor` and `BeanFactoryPostProcessor` definitions via `@Bean`. Those should usually be declared as `static` `@Bean` methods, not triggering the instantiation of their containing configuration class. Otherwise, `@Autowired` and `@Value` won't work on the configuration class itself since it is being created as a bean instance too early.

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }

}

@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    @Autowired
    public RepositoryConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

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

}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

> **Tip**
>
> Constructor injection in `@Configuration` classes is only supported as of Spring Framework 4.3. Note also that there is no need to specify `@Autowired` if the target bean defines only one constructor; in the example above, `@Autowired` is not necessary on the `RepositoryConfig` constructor.

In the scenario above, using `@Autowired` works well and provides the desired modularity, but determining exactly where the autowired bean definitions are declared is still somewhat ambiguous. For example, as a developer looking at `ServiceConfig`, how do you know exactly where the `@Autowired` `AccountRepository` bean is declared? It's not explicit in the code, and this may be just fine. Remember that the Spring Tool Suite provides tooling that can render graphs showing how everything is wired up - that may be all you need. Also, your Java IDE can easily find all declarations and uses of the `AccountRepository` type, and will quickly show you the location of `@Bean` methods that return that type.

In cases where this ambiguity is not acceptable and you wish to have direct navigation from within your IDE from one `@Configuration` class to another, consider autowiring the configuration classes themselves:

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        // navigate 'through' the config class to the @Bean method!
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }

}
```

In the situation above, it is completely explicit where `AccountRepository` is defined. However, `ServiceConfig` is now tightly coupled to `RepositoryConfig`; that's the tradeoff. This tight coupling can be somewhat mitigated by using interface-based or abstract class-based `@Configuration` classes. Consider the following:

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }
}

@Configuration
public interface RepositoryConfig {

    @Bean
    AccountRepository accountRepository();

}

@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig {

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

}

@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) // import the concrete config!
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

Now `ServiceConfig` is loosely coupled with respect to the concrete `DefaultRepositoryConfig`, and built-in IDE tooling is still useful: it will be easy for the developer to get a type hierarchy of `RepositoryConfig` implementations. In this way, navigating `@Configuration` classes and their dependencies becomes no different than the usual process of navigating interface-based code.

Conditionally include @Configuration classes or @Bean methods

It is often useful to conditionally enable or disable a complete `@Configuration` class, or even individual `@Bean` methods, based on some arbitrary system state. One common example of this is to use the `@Profile` annotation to activate beans only when a specific profile has been enabled in the Spring `Environment` (see the section called “Bean definition profiles” for details).

The `@Profile` annotation is actually implemented using a much more flexible annotation called `@Conditional`. The `@Conditional` annotation indicates specific `org.springframework.context.annotation.Condition` implementations that should be consulted before a `@Bean` is registered.

Implementations of the `Condition` interface simply provide a `matches(…)` method that returns `true` or `false`. For example, here is the actual `Condition` implementation used for `@Profile`:

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    if (context.getEnvironment() != null) {
        // Read the @Profile annotation attributes
        MultiValueMap<String, Object> attrs =
 metadata.getAllAnnotationAttributes(Profile.class.getName());
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if (context.getEnvironment().acceptsProfiles(((String[]) value))) {
                    return true;
                }
            }
            return false;
        }
    }
    return true;
}
```

See the `@Conditional` javadocs for more detail.

Combining Java and XML configuration

Spring's `@Configuration` class support does not aim to be a 100% complete replacement for Spring XML. Some facilities such as Spring XML namespaces remain an ideal way to configure the container. In cases where XML is convenient or necessary, you have a choice: either instantiate the container in an "XML-centric" way using, for example, `ClassPathXmlApplicationContext`, or in a "Java-centric" fashion using `AnnotationConfigApplicationContext` and the `@ImportResource` annotation to import XML as needed.

XML-centric use of @Configuration classes

It may be preferable to bootstrap the Spring container from XML and include `@Configuration` classes in an ad-hoc fashion. For example, in a large existing codebase that uses Spring XML, it will be easier to create `@Configuration` classes on an as-needed basis and include them from the existing XML files. Below you'll find the options for using `@Configuration` classes in this kind of "XML-centric" situation.

Remember that `@Configuration` classes are ultimately just bean definitions in the container. In this example, we create a `@Configuration` class named `AppConfig` and include it within `system-test-config.xml` as a `<bean/>` definition. Because `<context:annotation-config/>` is switched on, the container will recognize the `@Configuration` annotation and process the `@Bean` methods declared in `AppConfig` properly.

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

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

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }

}
```

system-test-config.xml:

```
<beans>
    <!-- enable processing of annotations such as @Autowired and @Configuration -->
    <context:annotation-config/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="com.acme.AppConfig"/>

    <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>
```

jdbc.properties:

```
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```
public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:/com/acme/system-test-
config.xml");
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

> **Note**
>
> In `system-test-config.xml` above, the `AppConfig` `<bean/>` does not declare an `id` element. While it would be acceptable to do so, it is unnecessary given that no other bean will ever refer to it, and it is unlikely that it will be explicitly fetched from the container by name. Likewise with the `DataSource` bean - it is only ever autowired by type, so an explicit bean `id` is not strictly required.

Because `@Configuration` is meta-annotated with `@Component`, `@Configuration`-annotated classes are automatically candidates for component scanning. Using the same scenario as above, we can redefine `system-test-config.xml` to take advantage of component-scanning. Note that in this case, we don't need to explicitly declare `<context:annotation-config/>`, because `<context:component-scan/>` enables the same functionality.

system-test-config.xml:

```
<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

In applications where `@Configuration` classes are the primary mechanism for configuring the container, it will still likely be necessary to use at least some XML. In these scenarios, simply use

`@ImportResource` and define only as much XML as is needed. Doing so achieves a "Java-centric" approach to configuring the container and keeps XML to a bare minimum.

```
@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);
    }

}
```

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

```
jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

7.13 Environment abstraction

The `Environment` is an abstraction integrated in the container that models two key aspects of the application environment: *profiles* and *properties*.

A *profile* is a named, logical group of bean definitions to be registered with the container only if the given profile is active. Beans may be assigned to a profile whether defined in XML or via annotations. The role of the `Environment` object with relation to profiles is in determining which profiles (if any) are currently active, and which profiles (if any) should be active by default.

Properties play an important role in almost all applications, and may originate from a variety of sources: properties files, JVM system properties, system environment variables, JNDI, servlet context parameters, ad-hoc Properties objects, Maps, and so on. The role of the `Environment` object with relation to properties is to provide the user with a convenient service interface for configuring property sources and resolving properties from them.

Bean definition profiles

Bean definition profiles is a mechanism in the core container that allows for registration of different beans in different environments. The word *environment* can mean different things to different users and this feature can help with many use cases, including: