`@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:

- working against an in-memory datasource in development vs looking up that same datasource from JNDI when in QA or production
- registering monitoring infrastructure only when deploying an application into a performance environment
- registering customized implementations of beans for customer A vs. customer B deployments

Let's consider the first use case in a practical application that requires a `DataSource`. In a test environment, the configuration may look like this:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("my-schema.sql")
        .addScript("my-test-data.sql")
        .build();
}
```

Let's now consider how this application will be deployed into a QA or production environment, assuming that the datasource for the application will be registered with the production application server's JNDI directory. Our `dataSource` bean now looks like this:

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

The problem is how to switch between using these two variations based on the current environment. Over time, Spring users have devised a number of ways to get this done, usually relying on a combination of system environment variables and XML `<import/>` statements containing `${placeholder}` tokens that resolve to the correct configuration file path depending on the value of an environment variable. Bean definition profiles is a core container feature that provides a solution to this problem.

If we generalize the example use case above of environment-specific bean definitions, we end up with the need to register certain bean definitions in certain contexts, while not in others. You could say that you want to register a certain profile of bean definitions in situation A, and a different profile in situation B. Let's first see how we can update our configuration to reflect this need.

@Profile

The `@Profile` annotation allows you to indicate that a component is eligible for registration when one or more specified profiles are active. Using our example above, we can rewrite the `dataSource` configuration as follows:

```
@Configuration
@Profile("dev")
public class StandaloneDataConfig {

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

```
@Configuration
@Profile("production")
public class JndiDataConfig {

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

Note

As mentioned before, with `@Bean` methods, you will typically choose to use programmatic JNDI lookups: either using Spring's `JndiTemplate`/`JndiLocatorDelegate` helpers or the straight JNDI `InitialContext` usage shown above, but not the `JndiObjectFactoryBean` variant which would force you to declare the return type as the `FactoryBean` type.

`@Profile` can be used as a meta-annotation for the purpose of creating a custom *composed annotation*. The following example defines a custom `@Production` annotation that can be used as a drop-in replacement for `@Profile("production")`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}
```

`@Profile` can also be declared at the method level to include only one particular bean of a configuration class:

```
@Configuration
public class AppConfig {

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

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

Tip

If a `@Configuration` class is marked with `@Profile`, all of the `@Bean` methods and `@Import` annotations associated with that class will be bypassed unless one or more of the specified profiles are active. If a `@Component` or `@Configuration` class is marked with `@Profile({"p1", "p2"})`, that class will not be registered/processed unless profiles 'p1' and/or 'p2' have been activated. If a given profile is prefixed with the NOT operator (`!`), the annotated element will be registered if the profile is **not** active. For example, given `@Profile({"p1", "!p2"})`, registration will occur if profile 'p1' is active or if profile 'p2' is not active.

XML bean definition profiles

The XML counterpart is the `profile` attribute of the `<beans>` element. Our sample configuration above can be rewritten in two XML files as follows:

```
<beans profile="dev"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="...">

    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

```
<beans profile="production"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

It is also possible to avoid that split and nest `<beans/>` elements within the same file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <!-- other bean definitions -->

    <beans profile="dev">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
    </beans>
</beans>
```

The `spring-bean.xsd` has been constrained to allow such elements only as the last ones in the file. This should help provide flexibility without incurring clutter in the XML files.

Activating a profile

Now that we have updated our configuration, we still need to instruct Spring which profile is active. If we started our sample application right now, we would see a `NoSuchBeanDefinitionException` thrown, because the container could not find the Spring bean named `dataSource`.

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

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

In addition, profiles may also be activated declaratively through the `spring.profiles.active` property which may be specified through system environment variables, JVM system properties, servlet context parameters in `web.xml`, or even as an entry in JNDI (see the section called “PropertySource abstraction”). In integration tests, active profiles can be declared via the `@ActiveProfiles` annotation in the `spring-test` module (see the section called “Context configuration with environment profiles”).

Note that profiles are not an "either-or" proposition; it is possible to activate multiple profiles at once. Programmatically, simply provide multiple profile names to the `setActiveProfiles()` method, which accepts `String`… varargs:

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

Declaratively, `spring.profiles.active` may accept a comma-separated list of profile names:

```
-Dspring.profiles.active="profile1,profile2"
```

Default profile

The *default* profile represents the profile that is enabled by default. Consider the following:

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}
```

If no profile is active, the `dataSource` above will be created; this can be seen as a way to provide a *default* definition for one or more beans. If any profile is enabled, the *default* profile will not apply.

The name of the default profile can be changed using `setDefaultProfiles()` on the `Environment` or declaratively using the `spring.profiles.default` property.

PropertySource abstraction

Spring’s `Environment` abstraction provides search operations over a configurable hierarchy of property sources. To explain fully, consider the following:

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsFoo = env.containsProperty("foo");
System.out.println("Does my environment contain the 'foo' property? " + containsFoo);
```

In the snippet above, we see a high-level way of asking Spring whether the `foo` property is defined for the current environment. To answer this question, the `Environment` object performs a search over a set of `PropertySource` objects. A `PropertySource` is a simple abstraction over any source of key-value pairs, and Spring’s `StandardEnvironment` is configured with two PropertySource objects — one representing the set of JVM system properties (*a la* `System.getProperties()`) and one representing the set of system environment variables (*a la* `System.getenv()`).