Autowiring Collaborators

The Spring container can autowire relationships between collaborating beans. You can let Spring resolve collaborators (other beans) automatically for your bean by inspecting the contents of the `ApplicationContext`. Autowiring has the following advantages:

- Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other mechanisms such as a bean template discussed elsewhere in this chapter are also valuable in this regard.)

Autowiring Collaborators
Limitations and Disadvantages of Autowiring
Excluding a Bean from Autowiring

Edit this Page
GitHub Project
Stack Overflow

be satisfied automatically without you needing to modify the configuration. This autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.

When using XML-based configuration metadata (see Dependency Injection), you can specify the autowire mode for a bean definition with the `autowire` attribute of the `<bean/>` element. The autowiring functionality has four modes. You specify autowiring per bean and can thus choose which ones to autowire. The following table describes the four autowiring modes:

Table 1. Autowiring modes

Mode	Explanation
`no`	(Default) No autowiring. Bean references must be defined by `ref` elements. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
`byName`	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name and it contains a `master` property (that is, it has a `setMaster(..)` method), Spring looks for a bean definition named `master` and uses it to set the property.
`byType`	Lets a property be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use `byType` autowiring for that bean. If there are no matching beans, nothing happens (the property is not set).
`constructor`	Analogous to `byType` but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

With `byType` or `constructor` autowiring mode, you can wire arrays and typed collections. In such cases, all autowire candidates within the container that match the expected type are provided to satisfy the dependency. You can autowire strongly-typed `Map` instances if the expected key type is `String`. An autowired `Map` instance's values consist of all bean instances that match the expected type, and the `Map` instance's keys contain the corresponding bean names.

Limitations and Disadvantages of Autowiring

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

- Explicit dependencies in `property` and `constructor-arg` settings always override autowiring. You cannot autowire simple properties such as primitives, `Strings`, and `Classes` (and arrays of such simple properties). This limitation is by-design.
- Autowiring is less exact than explicit wiring. Although, as noted in the earlier table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results. The relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.
- Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or `Map` instances, this is not necessarily a problem. However, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

In the latter scenario, you have several options:

- Abandon autowiring in favor of explicit wiring.
- Avoid autowiring for a bean definition by setting its `autowire-candidate` attributes to `false`, as described in the next section.
- Designate a single bean definition as the primary candidate by setting the `primary` attribute of its `<bean/>` element to `true`.
- Implement the more fine-grained control available with annotation-based configuration, as described in Annotation-based Container Configuration.

Excluding a Bean from Autowiring

On a per-bean basis, you can exclude a bean from autowiring. In Spring's XML format, set the `autowire-candidate` attribute of the `<bean/>` element to `false`; with the `@Bean` annotation, the attribute is named `autowireCandidate`. The container makes that specific bean definition unavailable to the autowiring infrastructure, including annotation-based injection points such as `@Autowired`.

NOTE

The `autowire-candidate` attribute is designed to only affect type-based autowiring. It does not affect explicit references by name, which get resolved even if the specified bean is not marked as an autowire candidate. As a consequence, autowiring by name nevertheless injects a bean if the name matches.

You can also limit autowire candidates based on pattern-matching against bean names. The top-level `<beans/>` element accepts one or more patterns within its `default-autowire-candidates` attribute. For example, to limit autowire candidate status to any bean whose name ends with `Repository`, provide a value of `*Repository`. To provide multiple patterns, define them in a comma-separated list. An explicit value of `true` or `false` for a bean definition's `autowire-candidate` attribute always takes precedence. For such beans, the pattern matching rules do not apply.

These techniques are useful for beans that you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured by using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

NOTE

As of 6.2, `@Bean` methods support two variants of the autowire candidate flag: `autowireCandidate` and `defaultCandidate`.

When using qualifiers, a bean marked with `defaultCandidate=false` is only available for injection points where an additional qualifier indication is present. This is useful for restricted delegates that are supposed to be injectable in certain areas but are not meant to get in the way of beans of the same type in other places. Such a bean will never get injected by plain declared type only, rather by type plus specific qualifier.

In contrast, `autowireCandidate=false` behaves exactly like the `autowire-candidate` attribute as explained above: Such a bean will never get injected by type at all.

Prev
Lazy-initialized Beans

Next
Method Injection