# Spring General Questions

* What is spring.

The Spring Framework is an [application framework](https://en.wikipedia.org/wiki/Application_framework) and [inversion of control](https://en.wikipedia.org/wiki/Inversion_of_control) [container](https://en.wikipedia.org/wiki/Servlet_container) for the [Java platform](https://en.wikipedia.org/wiki/Java_platform). The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the [Java EE](https://en.wikipedia.org/wiki/Java_EE) platform.

* What is dependency injection in Spring.

https://stackoverflow.com/questions/1061717/what-exactly-is-spring-framework-for

Basically Spring is a framework for [dependency-injection](https://stackoverflow.com/questions/tagged/dependency-injection) which is a pattern that allows to build very decoupled systems. I'll try to explain you the simplest I can (this isn't a short answer).

## **The problem**

For example, suppose you need to list the users of the system and thus declare an interface called UserLister:

public interface UserLister {

List<User> getUsers();

}

And maybe an implementation accessing a database to get all the users:

public class UserListerDB implements UserLister {

public List<User> getUsers() {

// DB access code here

}

}

In your view you'll need to access an instance (just an example, remember):

public class SomeView {

private UserLister userLister;

public void render() {

List<User> users = userLister.getUsers();

view.render(users);

}

}

Note that the code above doesn't have initialized the variable userLister. What should we do? If I explicitly instantiate the object like this:

UserLister userLister = new UserListerDB();

...I'd couple the view with my implementation of the class that access the DB. What if I want to switch from the DB implementation to another that gets the user list from a comma-separated file (remember, it's an example)? In that case I would go to my code again and change the last line by:

UserLister userLister = new UserListerCommaSeparatedFile();

This has no problem with a small program like this but... What happens in a program that has hundreds of views and a similar number of business classes. The maintenance becomes a nightmare!

## **Spring (Dependency Injection) approach**

What Spring does is to wire the classes up by using a XML file, this way all the objects are instantiated and initialized by Spring and injected in the right places (Servlets, Web Frameworks, Business classes, DAOs, etc, etc, etc...).

Going back to the example in Spring we just need to have a setter for the userLister field and have an XML like this:

<bean id="userLister" class="UserListerDB" />

<bean class="SomeView">

<property name="userLister" ref="userLister" />

</bean>

This way when the view is created it magically will have a UserLister ready to work.

List<User> users = userLister.getUsers(); // This will actually work

// without adding any line of code

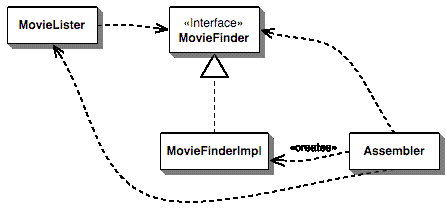
It is great! Isn't it?

* What if you want to use another implementation of your *UserLister* interface? Just change the XML
* What if don't have a *UserLister* implementation ready? Program a temporal mock implementation of UserLister and ease the development of the view
* What if I don't want to use Spring anymore? Just don't use it! Your application isn't coupled to it. [Inversion of Control](http://en.wikipedia.org/wiki/Inversion_of_control) states: "The application controls the framework, not the framework controls the application".
* What are the different types of dependency injection in Spring or In how many ways we can achieve dependency injection in spring.

https://martinfowler.com/articles/injection.html

## Forms of Dependency Injection

The basic idea of the Dependency Injection is to have a separate object, an assembler, that populates a field in the lister class with an appropriate implementation for the finder interface, resulting in a dependency diagram along the lines of [Figure 2](https://martinfowler.com/articles/injection.html#injector.gif)

**

*Figure 2: The dependencies for a Dependency Injector*

There are three main styles of dependency injection. The names I'm using for them are Constructor Injection, Setter Injection, and Interface Injection. If you read about this stuff in the current discussions about Inversion of Control you'll hear these referred to as type 1 IoC (interface injection), type 2 IoC (setter injection) and type 3 IoC (constructor injection). I find numeric names rather hard to remember, which is why I've used the names I have here.

### Constructor Injection with PicoContainer

I'll start with showing how this injection is done using a lightweight container called [PicoContainer](http://picocontainer.com/). I'm starting here primarily because several of my colleagues at ThoughtWorks are very active in the development of PicoContainer (yes, it's a sort of corporate nepotism.)

PicoContainer uses a constructor to decide how to inject a finder implementation into the lister class. For this to work, the movie lister class needs to declare a constructor that includes everything it needs injected.

*class MovieLister...*

public MovieLister(MovieFinder finder) {

this.finder = finder;

}

The finder itself will also be managed by the pico container, and as such will have the filename of the text file injected into it by the container.

*class ColonMovieFinder...*

public ColonMovieFinder(String filename) {

this.filename = filename;

}

The pico container then needs to be told which implementation class to associate with each interface, and which string to inject into the finder.

private MutablePicoContainer configureContainer() {

MutablePicoContainer pico = new DefaultPicoContainer();

Parameter[] finderParams = {new ConstantParameter("movies1.txt")};

pico.registerComponentImplementation(MovieFinder.class, ColonMovieFinder.class, finderParams);

pico.registerComponentImplementation(MovieLister.class);

return pico;

}

This configuration code is typically set up in a different class. For our example, each friend who uses my lister might write the appropriate configuration code in some setup class of their own. Of course it's common to hold this kind of configuration information in separate config files. You can write a class to read a config file and set up the container appropriately. Although PicoContainer doesn't contain this functionality itself, there is a closely related project called NanoContainer that provides the appropriate wrappers to allow you to have XML configuration files. Such a nano container will parse the XML and then configure an underlying pico container. The philosophy of the project is to separate the config file format from the underlying mechanism.

To use the container you write code something like this.

public void testWithPico() {

MutablePicoContainer pico = configureContainer();

MovieLister lister = (MovieLister) pico.getComponentInstance(MovieLister.class);

Movie[] movies = lister.moviesDirectedBy("Sergio Leone");

assertEquals("Once Upon a Time in the West", movies[0].getTitle());

}

Although in this example I've used constructor injection, PicoContainer also supports setter injection, although its developers do prefer constructor injection.

### Setter Injection with Spring

The [Spring framework](http://www.springsource.org/) is a wide ranging framework for enterprise Java development. It includes abstraction layers for transactions, persistence frameworks, web application development and JDBC. Like PicoContainer it supports both constructor and setter injection, but its developers tend to prefer setter injection - which makes it an appropriate choice for this example.

To get my movie lister to accept the injection I define a setting method for that service

*class MovieLister...*

private MovieFinder finder;

public void setFinder(MovieFinder finder) {

this.finder = finder;

}

Similarly I define a setter for the filename.

*class ColonMovieFinder...*

public void setFilename(String filename) {

this.filename = filename;

}

The third step is to set up the configuration for the files. Spring supports configuration through XML files and also through code, but XML is the expected way to do it.

<beans>

<bean id="MovieLister" class="spring.MovieLister">

<property name="finder">

<ref local="MovieFinder"/>

</property>

</bean>

<bean id="MovieFinder" class="spring.ColonMovieFinder">

<property name="filename">

<value>movies1.txt</value>

</property>

</bean>

</beans>

The test then looks like this.

public void testWithSpring() throws Exception {

ApplicationContext ctx = new FileSystemXmlApplicationContext("spring.xml");

MovieLister lister = (MovieLister) ctx.getBean("MovieLister");

Movie[] movies = lister.moviesDirectedBy("Sergio Leone");

assertEquals("Once Upon a Time in the West", movies[0].getTitle());

}

### Interface Injection

The third injection technique is to define and use interfaces for the injection. [Avalon](http://avalon.apache.org/) is an example of a framework that uses this technique in places. I'll talk a bit more about that later, but in this case I'm going to use it with some simple sample code.

With this technique I begin by defining an interface that I'll use to perform the injection through. Here's the interface for injecting a movie finder into an object.

public interface InjectFinder {

void injectFinder(MovieFinder finder);

}

This interface would be defined by whoever provides the MovieFinder interface. It needs to be implemented by any class that wants to use a finder, such as the lister.

*class MovieLister implements InjectFinder*

public void injectFinder(MovieFinder finder) {

this.finder = finder;

}

I use a similar approach to inject the filename into the finder implementation.

public interface InjectFinderFilename {

void injectFilename (String filename);

}

*class ColonMovieFinder implements MovieFinder, InjectFinderFilename...*

public void injectFilename(String filename) {

this.filename = filename;

}

Then, as usual, I need some configuration code to wire up the implementations. For simplicity's sake I'll do it in code.

*class Tester...*

private Container container;

private void configureContainer() {

container = new Container();

registerComponents();

registerInjectors();

container.start();

}

This configuration has two stages, registering components through lookup keys is pretty similar to the other examples.

*class Tester...*

private void registerComponents() {

container.registerComponent("MovieLister", MovieLister.class);

container.registerComponent("MovieFinder", ColonMovieFinder.class);

}

A new step is to register the injectors that will inject the dependent components. Each injection interface needs some code to inject the dependent object. Here I do this by registering injector objects with the container. Each injector object implements the injector interface.

*class Tester...*

private void registerInjectors() {

container.registerInjector(InjectFinder.class, container.lookup("MovieFinder"));

container.registerInjector(InjectFinderFilename.class, new FinderFilenameInjector());

}

public interface Injector {

public void inject(Object target);

}

When the dependent is a class written for this container, it makes sense for the component to implement the injector interface itself, as I do here with the movie finder. For generic classes, such as the string, I use an inner class within the configuration code.

*class ColonMovieFinder implements Injector...*

public void inject(Object target) {

((InjectFinder) target).injectFinder(this);

}

*class Tester...*

public static class FinderFilenameInjector implements Injector {

public void inject(Object target) {

((InjectFinderFilename)target).injectFilename("movies1.txt");

}

}

The tests then use the container.

*class Tester…*

public void testIface() {

configureContainer();

MovieLister lister = (MovieLister)container.lookup("MovieLister");

Movie[] movies = lister.moviesDirectedBy("Sergio Leone");

assertEquals("Once Upon a Time in the West", movies[0].getTitle());

}

The container uses the declared injection interfaces to figure out the dependencies and the injectors to inject the correct dependents. (The specific container implementation I did here isn't important to the technique, and I won't show it because you'd only laugh.)

* Give spring MVC.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.
* Give spring architecture.