What are the benefits of dependency injection containers?

Asked 16 years, 3 months ago Modified 9 years, 3 months ago Viewed 35k times

105

I understand benefits of dependency injection itself. Let's take Spring for instance. I also understand benefits of other Spring featureslike AOP, helpers of different kinds, etc. I'm just wondering, what are the benefits of XML configuration such as:

```
<bean id="Mary" class="foo.bar.Female">
  <property name="age" value="23"/>
</bean>
<bean id="John" class="foo.bar.Male">
  <property name="girlfriend" ref="Mary"/>
</bean>
```

compared to plain old java code such as:

```
Female mary = new Female();
mary.setAge(23);
Male john = new Male();
john.setGirlfriend(mary);
```

which is easier debugged, compile time checked and can be understood by anyone who knows only java. So what is the main purpose of a dependency injection framework? (or a piece of code that shows its benefits.)

UPDATE:
In case of

```
IService myService;// ...
public void doSomething() {
  myService.fetchData();
}
```

How can IoC framework guess which implementation of myService I want to be injected if there is more than one? If there is only one implementation of given interface, and I let IoC container automatically decide to use it, it will be broken after a second implementation appears. And if there is intentionally only one possible implementation of an interface then you do not need to inject it.

It would be really interesting to see small piece of configuration for IoC which shows it's benefits. I've been using Spring for a while and I can not provide such example.

And I can show single lines which demonstrate benefits of hibernate, dwr, and other frameworks which I use.

UPDATE 2:
I realize that IoC configuration can be changed without recompiling. Is it really such a good idea? I can understand when someone wants to change DB credentials without recompiling - he may be not developer. In your practice, how often someone else other than developer changes IoC configuration? I think that for developers there is no effort to recompile that particular class instead of changing configuration. And for non-developer you would probably want to make his life easier and provide some simpler configuration file.

UPDATE 3:

> External configuration of mapping between interfaces and their concrete implementations

What is so good in making it extenal? You don't make all your code external, while you definitely can - just place it in ClassName.java.txt file, read and compile manually on the fly - wow, you avoided recompiling. Why should compiling be avoided?!

> You save coding time because you provide mappings declaratively, not in a procedural code

I understand that sometimes declarative approach saves time. For example, I declare only once a mapping between a bean property and a DB column and hibernate uses this mapping while loading, saving, building SQL based on HSQL, etc. This is where the declarative approach works. In case of Spring (in my example), declaration had more lines and had the same expressiveness as corresponding code. If there is an example when such declaration is shorter than code - I would like to see it.

> Inversion of Control principle allows for easy unit testing because you can replace real implementations with fake ones (like replacing SQL database with an in-memory one)

I do understand inversion of control benefits (I prefer to call the design pattern discussed here as Dependency Injection, because IoC is more general - there are many kinds of control, and we are inverting only one of them - control of initialization). I was asking why someone ever needs something other than a programming language for it. I definitely can replace real implementations with fake ones using code. And this code will express same thing as configuration - it will just initialize fields with fake values.

```
mary = new FakeFemale();
```

I do understand benefits of DI. I do not understand what benefits are added by external XML configuration compared to configuring code that does the same. I do not think that compiling should be avoided - I compile every day and I'm still alive. I think configuration of DI is bad example of declarative approach. Declaration can be useful if is declared once AND is used many times in different ways - like hibernate cfg, where mapping between bean property and DB column is used for saving, loading, building search queries, etc. Spring DI configuration can be easily translated to configuring code, like in the beginning of this question, can it not? And it is used only for bean initialization, isn't it? Which means a declarative approach does not add anything here, does it?

When I declare hibernate mapping, I just give hibernate some information, and it works based on it - I do not tell it what to do. In case of spring, my declaration tells spring exactly wht to do - so why declare it, why not just do it?

LAST UPDATE:
Guys, a lot of answers are telling me about dependency injection, which I KNOW IS GOOD. The question is about purpose of DI configuration instead of initializing code - I tend to think that initializing code is shorter and clearer. The only answer I got so far to my question, is that it avoids recompiling, when the configuration changes. I guess I should post another question, because it is a big secret for me, why compiling should be avoided in this case.

xml spring dependency-injection

Share
Improve this question
Follow

edited Apr 29, 2009 at 22:45

community wiki
16 revs, 4 users 70%
Pavel Feldman

---

21 Finally someone had the courage to ask this question. Why indeed would you want avoid recompilation when your implementation changes at the cost of sacrificing (or at least degrading) tool/IDE support? – Christian Klauser Aug 31, 2010 at 17:10

3 It seems like the title isn't quite right. The author has said that IOC containers are fine, but seems to take issue with using XML config instead of configuring via code (and fair enough too). I'd suggest perhaps "What are the benefits of configuring IOC containers via XML or other non-code approaches?" – Orion Edwards Aug 31, 2010 at 22:15

@Orion Examples I provided (with Male and Female) do not require any IOC container. I'm fine with IOC; using container whether it is configured using XML or not, is still an open question for me. – Pavel Feldman Sep 3, 2010 at 0:23

@Orion 2: While I use some form of IOC in most of projects, some of them benefit from IOC container as much as they benefit from Variable Assignment Container or If Statement Container - just language is often enough for me. I have no problem recompiling projects I'm working on, and having dev/test/production initializing code conveniently separated. So for me title is fine. – Pavel Feldman Sep 3, 2010 at 0:30

I see trouble with sample. On of principles is **Inject services, not data** – Jacek Cz Sep 3, 2015 at 18:26

16 Answers

Sorted by: Highest score (default)

41

For myself one of the main reasons to use an IoC (and make use of external configuration) is around the two areas of:

- Testing
- Production maintenance

Testing

If you split your testing into 3 scenarios (which is fairly normal in large scale development):

1. Unit testing
2. Integration testing
3. Black box testing

What you will want to do is for the last two test scenarios (Integration & Black box), is not recompile any part of the application.

If any of your test scenarios require you to change the configuration (ie: use another component to mimic a banking integration, or do a performance load), this can be easily handled (this does come under the benefits of configuring the DI side of an IoC though.

Additionally if your app is used either at multiple sites (with different server and component configuration) or has a changing configuration on the live environment you can use the later stages of testing to verify that the app will handle those changes.

Production

As a developer you don't (and should not) have control of the production environment (in particular when your app is being distributed to multiple customers or seperate sites), this to me is the real benefit of using both an IoC and external configuration, as it is up to the infrastructure/production support to tweak and adjust the live environment without having to go back to developers and through test (higher cost when all they want to do is move a component).

Summary

The main benefits that external configuration of an IoC come from giving others (non-developers) the power to configure your application, in my experience this is only useful under a limited set of circumstances:

- Application is distributed to multiple sites/clients where environments will differ.
- Limited development control/input over the production environment and setup.
- Testing scenarios.

In practice I've found that even when developing something that you do have control over the environment it will be run on, over time it is better to give someone else the capabilities to change the configuration:

- When developing you don't know when it will change (the app is so useful your company sells it to someone else).
- I don't want to be stuck with changing the code every time a slight change is requested that could have been handled by setting up and using a good configuration model.

Note: Application refers to the complete solution (not just the executable), so all files required for the application to run.

Share
Improve this answer
Follow

edited Aug 14, 2013 at 19:43
Edgar Catalán
65 ● 8

answered Oct 8, 2008 at 12:20
Andrew Bickerton

15

Dependency injection is a coding style that has its roots in the observation that object delegation is usually a more useful design pattern than object inheritance (i.e., the object has-a relationship is more useful than the object is-a relationship). One other ingredient is necessary however for DI to work, that of creating object interfaces. Combining these two powerful design patterns software engineers quickly realized that they could create flexible loosely coupled code and thus the concept of Dependency Injection was born. However it wasn't until object reflection became available in certain high level languages that DI really took off. The reflection component is core to most of today's DI systems today because the really cool aspects of DI require the ability to programmatically select objects and configure and inject them into other objects using a system external and independent to the objects themselves.

A language must provide good support for both normal Object Oriented programming techniques as well as support for object interfaces and object reflection (for example Java and C#). While you can build programs using DI patterns in C++ systems its lack of reflection support within the language proper prevents it from supporting application servers and other DI platforms and hence limits the expressiveness of the DI patterns.

Strengths of a system built using DI patterns:

1. DI code is much easier to reuse as the 'depended' functionality is extrapolated into well defined interfaces, allowing separate objects whose configuration is handled by a suitable application platform to be plugged into other objects at will.
2. DI code is much easier to test. The functionality expressed by the object can be tested in a black box by building 'mock' objects implementing the interfaces expected by your application logic.
3. DI code is more flexible. It is innately loosely coupled code -- to an extreme. This allows the programmer to pick and choose how objects are connected based exclusively on their required interfaces on one end and their expressed interfaces on the other.
4. External (Xml) configuration of DI objects means that others can customize your code in unforeseen directions.
5. External configuration is also a separation of concern pattern in that all problems of object initialization and object interdependency management can be handled by the application server.
6. Note that external configuration is not required to use the DI pattern, for simple interconnections a small builder object is often adequate. There is a tradeoff in flexibility between the two. A builder object is not as flexible an option as an externally visible configuration file. The developer of the DI system must weigh the advantages of flexibility over convenience, taking care that small scale, fine grain control over object construction as expressed in a configuration file may increase confusion and maintenance costs down the line.

Definitely DI code seems more cumbersome, the disadvantages of having all of those XML files that configure objects to be injected into other objects appears difficult. This is, however, the point of DI systems. Your ability to mix and match code objects as a series of configuration settings allows you to build complex systems using 3rd party code with minimal coding on your part.

The example provided in the question merely touches on the surface of the expressive power that a properly factored DI object library can provide. With some practice and a lot of self discipline most DI practitioners find that they can build systems that have 100% test coverage of application code. This one point alone is extraordinary. This is not 100% test coverage of a small application of a few hundred lines of code, but 100% test coverage of applications comprising hundreds of thousands of lines of code. I am at a loss of being able to describe any other design pattern that provides this level of testability.

You are correct in that an application of a mere 10s of lines of code is easier to understand than several objects plus a series of XML configuration files. However as with most powerful design patterns, the gains are found as you continue to add new features to the system.

In short, large scale DI based applications are both easier to debug and easier to understand. While the Xml configuration is not 'compile time checked' all application services that this author is aware of will provide the developer with error messages if they attempt to inject an object having an incompatible interface into another object. And most provide a 'check' feature that covers all known objects configurations. This is easily and quickly done by checking that the to-be-injected object A implements the interface required by object B for all configured object injections.

Share

Improve this answer

Follow

edited Sep 25, 2008 at 9:04

answered Sep 25, 2008 at 8:21

Kevin S.
891 ● 6 ● 9

4 understand benefits of DI. I do not understand what benefits are added by external XML configuration compared to configurating code that does the same. Benefits you mentioned are provided by DI design pattern. The question was about benefits of DI configuration compared to plain initializing code. – Pavel Feldman Sep 25, 2008 at 9:08

>External configuration is also a separation... Separation of configuration is the heart of DI which is good. And it can de dome using initializing code. What cfg adds compared to initializing code? For me it seems that each line of cfg has corresponding initializing code line. – Pavel Feldman Sep 25, 2008 at 9:23

7

This is a bit of a loaded question, but I tend to agree that huge amounts of xml configuration doesn't really amount to much benefit. I like my applications to be as light on dependencies as possible, including the hefty frameworks.

They simplify the code a lot of the times, but they also have an overhead in complexity that makes tracking down problems rather difficult (I have seen such problems first hand, and straight Java I would be a lot more comfortable dealing with).

I guess it depends on style a bit, and what you are comfortable with... do you like to fly your own solution and have the benefit of knowing it inside out, or bank on existing solutions that may prove difficult when the configuration isn't just right? It's all a tradeoff.

However, XML configuration is a bit of a pet peeve of mine... I try to avoid it at all costs.

Share Improve this answer Follow

answered Sep 25, 2008 at 7:44
Mike Stone
44.6k ● 30 ● 114 ● 140

5

Any time you can change your code to data you're making a step in the right direction.

Coding anything as data means that your code itself is more general and reusable. It also means that your data may be specified in a language that fits it exactly.

Also, an XML file can be read into a GUI or some other tool and easily manipulated pragmatically. How would you do that with the code example?

I'm constantly factoring things most people would implement as code into data, it makes what code is left MUCH cleaner. I find it inconceivable that people will create a menu in code rather than as data--it should be obvious that doing it in code is just plain wrong because of the boilerplate.

Share
Improve this answer
Follow

edited Aug 28, 2015 at 20:51

community wiki
2 revs
Bill K

makes sense, did not think about it in this perspective – Pavel Feldman Apr 16, 2009 at 20:48

7 then again, people often go the other way and try and put logic into data which just means you end up coding your application in a substandard programming language – Casebash Sep 20, 2009 at 0:45

@Casebash That's an interesting viewpoint--I would be amazingly interested in an example. I find that anything I can move into data helps. I also find that if I do just what you are saying, the language is actually improved because it's a DSL--but even then it takes serious justification to create an entirely new language. – Bill K Sep 20, 2009 at 16:51

1 "Any time you can change your code to data you're making a step in the right direction." Welcome to the Soft Coding anti-pattern. – Raedwald Apr 11, 2012 at 12:29

@Raedwald What you are talking about is externalizing which can be really difficult if you don't know what you are doing (and the reason someone incompetent tried it, failed and called it an anti-pattern) More positive examples would be Injection, Iterators, nearly anything with annotations, anything that initializes with arrays. Most great programming structures are an attempt to separate out the differences in your code and combine what's left, driving it with something that can be grouped and managed better. – Bill K Apr 12, 2012 at 1:28

3

The reason for using a DI container are that you don't have to have a billion properties pre-configured in your code that are simply getters and setters. Do you really want to hardcode all those with new X()? Sure, you can have a default, but the DI container allows the creation of singletons which is extremely easy and allows you to focus on the details of the code, not the miscellaneous task of initializing it.

For example, Spring allows you to implement the InitializingBean interface and add an afterPropertiesSet method (you may also specify an "init-method" to avoid coupling your code to Spring). These methods will allow you to ensure that any interface specified as a field in your class instance is configured correctly upon startup, and then you no longer have to null-check your getters and setters (assuming you do allow your singletons to remain thread-safe).

Furthermore, it is much easier to do complex initializations with a DI container instead of doing them yourself. For instance, I assist with using XFire (not CeltiXFire, we only use Java 1.4). The app used Spring, but it unfortunately used XFire's services.xml configuration mechanism. When a Collection of elements needed to declare that it had ZERO or more instances instead of ONE or more instances, I had to override some of the provided XFire code for this particular service.

There are certain XFire defaults defined in its Spring beans schema. So, if we were using Spring to configure the services, the beans could have been used. Instead, what happened was that I had to supply an instance of a specific class in the services.xml file instead of using the beans. To do this, I needed to provide the constructor and set up the references declared in the XFire configuration. The real change that I needed to make required that I overload a single class.

But, thanks to the services.xml file, I had to create four new classes, setting their defaults according to their defaults in the Spring configuration files in their constructors. If we had been able to use the Spring configuration, I could have just stated:

```
<bean id="base" parent="RootXFireBean">
    <property name="secondProperty" ref="secondBean" />
</bean>

<bean id="secondBean" parent="secondaryXFireBean">
    <property name="firstProperty" ref="thirdBean" />
</bean>

<bean id="thirdBean" parent="thirdXFireBean">
    <property name="secondProperty" ref="myNewBean" />
</bean>
```

```
<bean id="myNewBean" class="WowItsActuallyTheCodeThatChanged" />
```

Instead, it looked more like this:

```
public class TheFirstPointlessClass extends SomeXFireClass {
    public TheFirstPointlessClass() {
        setFirstProperty(new TheSecondPointlessClass());
        setSecondProperty(new TheThingThatWasHereBefore());
    }
}

public class TheSecondPointlessClass extends YetAnotherXFireClass {
    public TheSecondPointlessClass() {
        setFirstProperty(TheThirdPointlessClass());
    }
}

public class TheThirdPointlessClass extends GeeAnotherXFireClass {
    public TheThirdPointlessClass() {
        setFirstProperty(new AnotherThingThatWasHereBefore());
        setSecondProperty(new WowItsActuallyTheCodeThatChanged());
    }
}

public class WowItsActuallyTheCodeThatChanged extends
TheXFireClassIActuallyCareAbout {
    public WowItsActuallyTheCodeThatChanged() {
    }

    public overrideTheMethod(Object[] arguments) {
        //Do overridden stuff
    }
}
```

So the net result is that four additional, mostly pointless Java classes had to be added to the codebase to achieve the affect that one additional class and some simple dependency container information achieved. This isn't the "exception that proves the rule", this IS the rule...handling quirks in code is much cleaner when the properties are already provided in a DI container and you're simply changing them to suit a special situation, which happens more often than not.

Share Improve this answer Follow

answered Oct 1, 2008 at 2:12

MetroidFan2002
29.9k ● 16 ● 65 ● 81

3

I have your answer

There are obviously trade offs in each approach, but externalized XML configuration files are useful for enterprise development in which build systems are used to compile the code and not your IDE. Using the build system, you may want to inject certain values into your code - for example the version of the build (which could be painful to have to update manually each time you compile). The pain is greater when your build system pulls code off of some version control system. Modifying simple values at compile time would require you to change a file, commit it, compile, and then revert each time for each change. These aren't changes that you want to commit into your version control.

Other useful use cases regarding the build system and external configs:

- injecting styles/stylesheets for a single code base for different builds
- injecting different sets of dynamic content (or references to them) for your single code base
- injecting localization context for different builds/clients
- changing a webservice URI to a backup server (when the main one goes down)

Update: All the above examples were on things that didn't necessarily require dependencies on classes. But you can easily build up cases where both a complex object and automation is necessary - for example:

- Imagine you had a system in which it monitored the traffic of your website. Depending on the # of concurrent users, it turns on/off a logging mechanism. Perhaps while the mechanism is off, a stub object is put in its place.
- Imagine you had a web conferencing system in which depending on the # of users, you want to switch out the ability to do P2P depending on # of participants

Share
Improve this answer
Follow

edited Feb 24, 2011 at 1:48

community wiki
2 revs
badunk

+1 for highlighting the enterprise aspect right at the top. Testing poorly written legacy code can be a days-long nightmare at times. – Richard Le Mesurier Dec 10, 2013 at 8:45

2

You don't need to recompile your code each time you change something in configuration. It will simplify program deployment and maintenance. For example you can swap one component with another with just 1 change in config file.

Share Improve this answer Follow

answered Sep 25, 2008 at 7:43

aku
124k ● 33 ● 176 ● 203

deployment? probably... maintenance of deployment? probably... maintenance of code? I tend to think no... debugging through frameworks tends to be a big headache, and pojos are a lot easier to deal with in that regard. – Mike Stone Sep 25, 2008 at 7:47

1 Mike, I said nothing about code. We all know that XML configuration sucks :) – aku Sep 25, 2008 at 7:50

Hmm.. how often do you change components without recompiling and what for? I understand if someone changes DB credentials and does not want to recompile the program - he may be not the one who develops it. But I can hardly imagine someone other than developer changing spring configuration – Pavel Feldman Sep 25, 2008 at 7:50

Pavel usually this situation occurs when you have to deploy program to hundreds of clients. In such situation it's much easier to change config rather than deploy new version of the product. You're right saying about devs. Usually developer creates new cfg and admin deploys it. – aku Sep 25, 2008 at 9:44

2

You can slot in a new implementation for girlfriend. So new female can be injected without recompiling your code.

```
<bean id="jane" class="foo.bar.HotFemale">
  <property name="age" value="19"/>
</bean>
<bean id="mary" class="foo.bar.Female">
  <property name="age" value="23"/>
</bean>
<bean id="john" class="foo.bar.Male">
  <property name="girlfriend" ref="jane"/>
</bean>
```

(The above assumes Female and HotFemale implement the same GirlfFriend interface)

Share Improve this answer Follow

answered Sep 25, 2008 at 8:02

Paul Whelan
16.8k ● 12 ● 52 ● 85

Why logic modifications without recompiling are considered a good idea? – Pavel Feldman Sep 25, 2008 at 8:05

I definitely can do HotFemale jane = new HotFmale(); jane.setAge(19); john.setGirlfriend(jane); So the only difference is that cfg can be changed without recompiling? It seems to be common answer when Spring is discussed. Why?! Why is it good to avoid compiling? – Pavel Feldman Sep 25, 2008 at 8:14

Well I can test the code better I can Mock the Female Object. – Paul Whelan Sep 25, 2008 at 9:58

@Pavel Feldman : because if you already have the app deployed at the client this is easier. – Andrei Rînea Nov 12, 2010 at 15:52

1

In the .NET world, most of IoC frameworks provide both XML and Code configuration.

StructureMap and Ninject, for example, use fluent interfaces to configure containers. You are no longer constrained to use XML configuration files. Spring, which also exists in .NET, heavily relies on XML files since it is his historical main configuration interface, but it is still possible to configure containers programmatically.

Share Improve this answer Follow

answered Sep 25, 2008 at 7:43
Romain Verdier
13k ●7 ●59 ●77

That's great that I finally can use code for what it intended to be used :) But why do I even need anything other than programming language to do such things? – Pavel Feldman Sep 25, 2008 at 7:46

I think that is because XML allows runtime changes, or at least, configuration changes without having to recompile the project. – Romain Verdier Sep 25, 2008 at 7:57

1

Ease of **combining partial configurations** into a final complete configuration.

For example, in web applications, the model, view and controllers are typically specified in separate configuration files. Use the declarative approach, you can load, for example:

```
UI-context.xml
Model-context.xml
Controller-context.xml
```

Or load with a different UI and a few extra controllers:

```
AlternateUI-context.xml
Model-context.xml
Controller-context.xml
ControllerAdditions-context.xml
```

To do the same in code requires an infrastructure for combining partial configurations. Not impossible to do in code, but certainly easier to do using an IoC framework.

Share Improve this answer Follow

answered Sep 26, 2008 at 4:01
flicken

15.5k ● 4 ● 31 ● 29

1

Often, the important point is *who* is changing the configuration after the program was written. With configuration in code you implicitly assume that person changing it has the same skills and access to source code etc as the original author had.

In production systems it is very practical to extract some subset of settings (e.g. age in you example) to XML file and allow e.g. system administrator or support personal to change the value without giving them the full power over source code or other settings - or just to isolate them from complexities.

Share
Improve this answer
Follow

answered Jun 17, 2010 at 22:22

community wiki
Miro A.

That's a valid point, but spring configuration often tends to be rather complex. While it's easy to change age, sysadmin still has to deal with large xml file he does not necessary understand entirely. Isn't it better to extract portion that is supposed to be configured into something even more simple, than spring XML config? Like properties file, with single line "age=23" and not let admin change other details, like class names, etc, that require knowledge of internal program structure. – Pavel Feldman Jun 18, 2010 at 19:10

I was recently working on a project that had a mixture of Java code and XSLT. The team was a mixture of people who were strong in Java (and maybe less comfortable working with XML and XSLT); and people who were very strong working with XML and XSLT (and less comfortable with Java). Since the configuration was going to be managed by the second group, it made sense to use Spring, and have XML configs. In other words, Spring solved a division-of-labour problem in the team. It didn't solve a "technical" problem; in the sense that the config could just as easily have been done with Java code. – Dawood ibn Kareem Dec 30, 2011 at 10:00

When would 'support personnel' know anything about having to change some classes being created within a dependency injection container?? Really under the assumption that it's a developer's job to do this? – Jimbo May 9, 2013 at 13:04

That is exactly the reason why extracting the configuration values (e.g. URL of system you integrate with) makes sense: the support personel edits properties file or (in worst case) XML file, the compiled Java class remains the same. – Miro A. May 12, 2013 at 1:49

1

From a Spring perspecitve I can give you two answers.

First the XML configuration isn't the only way to define the configuration. Most things can be configured using annotations and the things that must be done with XML are configuration for code that you aren't writing anyways, like a connection pool that you are using from a library. Spring 3 includes a method for defining the DI configuration using Java similar to the hand rolled DI configuration in your example. So using Spring does not mean that you have to use an XML based configuration file.

Secondly Spring is a lot more than just a DI framework. It has lots of other features including transaction management and AOP. The Spring XML configuration mixes all these concepts together. Often in the same configuration file I'm specifying bean dependencies, transaction settings and adding session scoped beans that actually handled using AOP in the background. I find the XML configuration provides a better place to manage all these features. I also feel that the annotation based configuration and XML configuration scale up better than doing Java based configuration.

But I do see your point and there isn't anything wrong with defining the dependency injection configuration in Java. I normally do that myself in unit tests and when I'm working on a project small enough that I haven't added a DI framework. I don't normally specify configuration in Java because to me that's the kind plumbing code that I'm trying to get away from writing when I chose to use Spring. That's a preference though, it doesn't mean that XML configuration is superior to Java based configuration.

Share
Improve this answer
Follow

answered Apr 5, 2011 at 21:04

community wiki
David W Crook

0

Spring also has a properties loader. We use this method to set variables that are dependant on the environment (e.g. development, testing, acceptance, production, ...). This could be for example the queue to listen to.

If there is no reason why the property would change, there is also no reason to configure it in this way.

Share Improve this answer Follow

answered Sep 25, 2008 at 7:45

Jeroen Wyseur
3,543 ● 3 ● 22 ● 16

0

Your case is very simple and therefore doesn't need an IoC (Inversion of Control) container like Spring. On the other hand, when you "program to interfaces, not implementations" (which is a good practice in OOP), you can have code like this:

```
IService myService;
// ...
public void doSomething() {
  myService.fetchData();
}
```

(note that the type of myService is IService -- an interface, not a concrete implementation). Now it can be handy to let your IoC container automatically provide the correct concrete instance of IService during initialization - when you have many interfaces and many implementations, it can be cumbersome to do that by hand. Main benefits of an IoC container (dependency injection framework) are:

- External configuration of mapping between interfaces and their concrete implementations
- IoC container handles some tricky issues like resolving complicated dependency graphs, managing component's lifetime etc.
- You save coding time because you provide mappings declaratively, not in a procedural code
- Inversion of Control principle allows for easy unit testing because you can replace real implementations with fake ones (like replacing SQL database with an in-memory one)

Share Improve this answer Follow

answered Sep 25, 2008 at 7:51
Borek Bernard
53.1k ● 62 ● 175 ● 246

0

Initializing in an XML config file will simplify your debugging / adapting work with a client who has your app deployed on their computers. (Because it doesn't require recompilation + binary files replacement)

Share
Improve this answer
Follow

answered Nov 12, 2010 at 15:58

community wiki
Andrei Rînea

-2

One of the most appealing reasons is the "Hollywood principle": don't call us, we'll call you. A component is not required to do the lookups to other components and services itself; instead they are provided to it automatically. In Java, this means that it is no longer necessary to do JNDI lookups inside the component.

It is also lots easier to unit test a component in isolation: instead of giving it an actual implementation of the components it needs, you simply use (possibly auto generated) mocks.

Share Improve this answer Follow

answered Sep 25, 2008 at 8:54
jan.vdbergh
2,119 ● 2 ● 20 ● 29

This answer is about dependency injection. I know what it is, I know it is good and I state it clearly in the first sentence of the question. The question was about benefits of DI configuration, compared to plain initializing code. – Pavel Feldman Sep 25, 2008 at 9:06

Doesn't really answer the question – Casebash Sep 20, 2009 at 0:43