Spring Dependency Injection Styles – Why I love Java based configuration (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/)

07/22/12 by **Tobias Flohre (https://blog.codecentric.de/en/author/tobias-flohre/)**

20 Comments (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comments)

I must admit, when looking at Spring 3.0's feature list for the first time I didn't see one thing I wanted to use right away in my next project.
There was the Spring Expression Language,
the stereotype annotation model,
there was some Spring MVC stuff which I didn't care about because I had nothing to do with Spring MVC back then,
and there was Java based configuration.

I saw potential in SpEL which indeed is very powerful. Now, almost three years later, I have used it here and then, but only very simple expressions, because anything a little more complicated always went into a Java component. Expressing complicated things through an expression never felt right. The stereotype annotation model is nice – but I never used it. It makes sense if you build up your own architecture relying on Spring's component scanning / autowiring injection style – never happened in three years. And Spring MVC – I got to know it by now, and I really liked it, but you know, it's a web framework, and you need a web framework when you need a web framework, you cannot use it always in every project using Spring.

When I look back now the **biggest impact** on my daily work with Spring definitely had the Java based configuration style, and I would never have thought that when looking at the feature list of Spring 3.0 back in 2009. Now I think that Java based configuration is one of those features that again **proves the significance of the Spring Framework**, and I'll explain why.

The world before Spring 3.0

Before Spring 3.0 came out there were two dependency injection styles available in Spring Core, one based on XML and one based on annotations. The annotation style is very similar to the way JEE 5/6 handles DI, and the XML style felt somehow, I don't know, outdated. At that time there were a lot of "why do we still need Spring" – discussions going on between Spring and JEE fans, and I read a lot of long discussions in comments under certain blog posts showing how serious people can get if you tell something bad about their toy.

Anyway, a little bit I felt like that as well. Do I still need Spring? Now, three years later, a definite "yes" is the answer, and of course it's not just the Java based configuration that makes me feel like that, it's the whole eco system that improved over the years, and you can still see the innovation (take Spring Data for example). But for me personally, a good part of that "yes" comes from Java based configuration.

But let's get back to 2009.

Component scanning and autowiring

To be honest, I don't like component scanning and autowiring that much, neither in Spring nor in JEE 5/6. Of course it always depends on the circumstances, the application, co-workers and so on, but in a bigger application it's a little bit too much magic for my taste. And I believe there's a violation of the dependency injection rule that a component should not know the bigger picture, in fact the **whole configuration is spread** among the business components.
Another disadvantage: there is not the one and only place where you can look for the configuration. And regarding Spring: there was still at least a little bit of XML necessary.

Okay, XML then?

We all know the disadvantages of XML by now, don't we? Here are some of them:

- It's not type-safe, you won't get errors before starting the Spring ApplicationContext, and sometimes even later. Typing errors may slow you down.
- XML is verbose, so configuration files get big. It's a good thing to split them up.
- Regarding splitting the configuration up: it's not possible to navigate between different XML-files. If you wanna know where Spring bean xyService is defined, you'll have to rely on full-text-search, like in the medieval age of programming.
- If you wanna build up libraries for usage (and you do that in big companies, where architecture teams provide libraries for other developers), it's really hard to find XML configuration files in jars on the classpath, and it's even harder to detect references in those files.

Some of those disadvantages may be covered up somehow when you've got the right tooling, but often you cannot choose your IDE. And not everything can be covered up.
So, both styles, annotation and XML based, have their advantages and disadvantages. They work well, of course, and will do in the future, I just wasn't very enthusiastic about them anymore.
And then, with Spring 3.0, came Java based configuration, completed in Spring 3.1, and made me enthusiastic again.

Tooling

First of all, tooling support is perfect in any Java IDE. Out of the box you get
– type-safety check by compiling
– code completion
– refactoring support
– support for finding references in the workspace (even on jars in the classpath)
That's one important part.

Language

The second part is about the language. I like Java, so why should I use different language construct for configurations? It feels really natural not to switch between Java and XML anymore. And of course, you can program anything you want directly in Java when creating Spring beans, like calling init-methods or static factory methods. No need for complicated ways to express that in XML.

Patterns

The third part is about patterns. Let's take a look at some elegant patterns for Java based configuration.

Navigable configurations

Java is verbose, too, so it makes sense to split up big configurations into several configuration classes. You can separate bean definitions by component and/or by layer, for example one configuration for low level infrastructure like datasources, transaction manager and co, one for high level infrastructure components, one for repositories, one for services etc.
When connecting those configurations, you should use this pattern:

```
@Configuration
public class PartnerConfig {

        @Bean
        public PartnerService partnerService() {
                return new PartnerServiceImpl();
        }

}
```

```
@Configuration
@Import(PartnerConfig.class)
public class CashingConfig {

	@Autowired
	private PartnerConfig partnerConfig;

	@Bean
	public CashingService cashingService() {
		return new CashingServiceImpl(partnerConfig.partnerService());
	}

}
```

We have two configuration files with the responsibility for different components. The *CashingService* depends on one of the components from *PartnerConfig*. By importing the *PartnerConfig* into the *CashingConfig*, all Spring beans from the *PartnerConfig* are available, but instead of autowiring those components directly, you wire the configuration class itself into the *CashingConfig* and use it to reference beans from it.
If you build up all your configuration files like this, it's easy to navigate through all dependencies by jumping directly into the method that defines the used component, even if the configuration class is in a jar on the classpath.

Abstract dependency definitions

It's easy to write a library and define through an abstract class or an interface needed components a user of the library has to add.
A simple example for this pattern is defining infrastructure components through an interface:

```
public interface InfrastructureConfig {

	public DataSource dataSource();

	public PlatformTransactionManager transactionManager();

}
```

As the author of that piece of software you create a configuration class like this:

```
@Configuration
public class ApplicationConfig {

	@Autowired
	private InfrastructureConfig infrastructureConfig;

	@Bean
	public JdbcTemplate jdbcTemplate() {
		return new JdbcTemplate(infrastructureConfig.dataSource());
	}

	@Bean
	public SomeService someService() {
		return new SomeServiceImpl(jdbcTemplate());
	}

}
```

When somebody wants to use the library, he has to create an implementation of *InfrastructureConfig* and add it to the ApplicationContext. This way the developer of the library doesn't need to think of environments in which the classes will run, it's up to the user.
There is much potential in this pattern, you can think of abstract configuration classes defining some Spring beans fully, some just as abstract methods and providing default Spring beans for some types. Somebody using the library extends this configuration class, defines the abstract beans and overrides some of the default Spring beans. Whenever you're developing some kind of framework, consider this pattern.

Multiple imports and profiles

Looking at the example code of the last paragraph, we can go a step further using Spring 3.1's profiles:

```
@Configuration
@Import({ JndiInfrastructureConfig.class, StandaloneInfrastructureConfig.class })
public class ApplicationConfig {

	@Autowired
	private InfrastructureConfig infrastructureConfig;

	@Bean
	public JdbcTemplate jdbcTemplate() {
		return new JdbcTemplate(infrastructureConfig.dataSource());
	}

	@Bean
	public SomeService someService() {
		return new SomeServiceImpl(jdbcTemplate());
	}
```

```
}
```

This way we provide two implementations of the *InfrastructureConfig* interface. Since we can only autowire one of them into the *ApplicationConfig*, just one may be active. With the *@Profile* annotation, configuration classes just get imported if the profile mentioned in the annotation is active.

```
@Profile("standalone")
@Configuration
public class StandaloneInfrastructureConfig implements InfrastructureConfig {

	@Bean
	public DataSource dataSource() {
		BasicDataSource dataSource = new BasicDataSource();
		dataSource.setUrl("someURL");
		dataSource.setUsername("username");
		dataSource.setPassword("password");
		return dataSource;
	}

	@Bean
	public PlatformTransactionManager transactionManager() {
		return new DataSourceTransactionManager(dataSource());
	}

}
```

Somebody using the library now has the option to activate one of two default profiles or to implement another implementation of *InfrastructureConfig* and add it to the application context.

Conclusion

Since Spring 3.0 we have three different dependency injection styles, and my favorite is clear: Java based configuration has the best tooling support, feels natural and offers some nice patterns, especially when building frameworks and libraries. So give it a try!
And regarding the significance of the Spring Framework: right then when I felt that the existing dependency injection styles are not that perfect, Spring provided me with a third option, perfectly fitting my taste. I guess that's innovation.

10 Application Performance Tuning Tips (https://blog.codecentric.de/en/2012/07/10-application-performance-tuning-tips/)

Tobias Flohre (https://blog.codecentric.de/en/author/tobias-flohre/)

Post by **Tobias Flohre**

CONTINUOUS DELIVERY

Continuous Delivery Patterns: Building your application inside a Docker container (https://blog.codecentric.de/en/2016/11/continuous-delivery-patterns-building-application-inside-docker-container/)

Contract Testing: Testen in einem Deploy-To-Production-Whenever-You-Want-Szenario (https://blog.codecentric.de/2016/09/contract-testing-testen-einem-deploy-production-whenever-want-szenario/)

More content about **Java**

JAVA

Scaling Spring Boot Apps on Docker Windows Containers with Ansible: A Complete Guide incl Spring Cloud Netflix and Docker Compose (https://blog.codecentric.de/en/2017/05/ansible-docker-windows-containers-scaling-spring-cloud-netflix-docker-compose/)

JAVA

Running Spring Boot Apps on Docker Windows Containers with Ansible: A Complete Guide incl Packer, Vagrant & Powershell (https://blog.codecentric.de/en/2017/04/ansible-docker-windows-containers-spring-boot/)

Kommentare

23. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73148) von **Tcharl (http://osgiliath.net)**

An other advantage is that the maven-bundle-plugin marks Java configuration imports into MANIFEST.MF, while it does not work with xml conf.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73148#respond)

23. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73229) von **J. Paul**

(http://javarevisited.blogspot.gr)

Your points on XML configuration files are perfectly valid and they are not as smooth as Java based configuration but by using XML to configure system has quite a standard now days and many people are familiar with that, so splitting configuration between XML and Java is not a good idea, I guess.

Javin
Top 10 Spring interview questions (http://javarevisited.blogspot.sg/2011/09/spring-interview-questions-answers-j2ee.html)

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73229#respond)

23. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73232) von Tobias Flohre

Hi Javin,

there's no need for splitting: since Spring 3.1 you can build a web application without a single line of XML (even no web.xml: http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/WebApplicationInitializer.html (http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/WebApplicationInitializer.html)).

Tobias

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73232#respond)

23. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73245) von Mitch Pronschinske

Hi Tobias,

Would you and the other blog authors of Codecentric be interested in being republished on Javalobby sometime? Send me an email if you're interested.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73245#respond)

23. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73324) von **Java Developer**

(http://www.fromdev.com)

I guess you may want to try STS for spring beans config navigation. It has really good features though the cross xml browsing is still not available, which should not be a difficult thing as a eclipse plugin though.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73324#respond)

24. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73439) von Tobias Flohre

I'm using STS as much as I can, but I don't always have the choice. And anyway, even if you use STS, XML is far away from being as smooth as Java based configuration. And I certainly don't wanna write my own eclipse plugin for something I get for free somewhere else.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73439#respond)

24. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73501) von **Peter**

(http://karussell.wordpress.com/)

6 months ago I tried spring again.

It was still a pain to work with the Java config if you need something beyond the normal DI configuration like authentication etc.

although I tried hard it was not possible at that time to find docs etc for it (again, the auth. part of the java configuration not the DI part)

And for a DI only solution: I'm still prefering a more lightweight solution like Guice. It is very easy to use, also in a web container environment with GuiceServlet. Also it is more transparent to test with IMO (as you don't need a magic subclass just call Guice.createInjector(modules))

This one simple setup call ala "Guice.createInjector" gives you the power to use guice in a command line app (batch job) and even on android – see roboguice!

As a side note: although it is a matter of taste I don't like annotating everything. And I'm not sure if you can configure your beans in Spring also in a Java fashion like in Guice.

Let me know what assumptions are incorrect 😉 !

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73501#respond)

25. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73842) von Tobias Flohre

Hi Peter, hard to say what you mean with authentication, but you said it was a problem not related to Java Config DI, so it's probably out of scope of this blog post. And regarding Guice: I don't know enough about it to compare it to Spring.
Regarding annotations: as I wrote I don't like annotations for DI, so my business components are free of those. I don't mind using @Configuration and @Bean in configuration classes, though.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73842#respond)

26. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-73969) von **Vishal (http://cloudspring.com)**

Hi Tobias,
Forgive my ignorance, but by moving the configuration into a Java class, navigation is definitely simpler. But if you have to change the configuration, you are changing a piece of code which needs "compile". What are your thoughts on this?

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=73969#respond)

26. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-74071) von Tobias Flohre

Hi Vishal,

yes, it's a piece of code and you need to recompile it when you change it. But I don't see where not needing to recompile XML is any advantage.
In a normal enterprise environment these days you have a build server like Jenkins doing your builds. Whenever you change something in your application and you want to deploy it to production, Jenkins will build it before, including Java compilation.
A Spring configuration (whether XML or Java) belongs to the heart of your application, you shouldn't change it on the fly without a build directly in production. If you need your application to be configurable after building, you should make it configurable by property-files and/or profiles, not by changing the XML configuration directly.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=74071#respond)

8. July 2014 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-193080) von Phillip

Wow, okay not to resurrect a 2 year old thread, but when people come across this later I want to express why I strongly disagree with this sentiment:

There are a number of groups, companies, and organizations, (e.g. banks, insurance, investment houses, retail outfits, etc.), that strongly control their deployed baseline. That is, in order to deploy a new baseline to the servers, there is a several-week-to-many-month process of reviews, meetings and paperwork. However, in these environments it's often trivial to get a 'config file tweak' approved, because hey, all you're doing is asking them to cycle servers after changing *one* text (xml) file. If there's going to be a change for regulations or setup you can see coming, and you have those classes already defined in software, then when those changes happen, you save yourself a world of pain by simply 'updating' this xml, and not trying to push a whole new software baseline that has to go through code reviews, acceptance testing, integration reviews, ops update impact statements, scheduling meetings, etc. etc.

Anything which requires a re-compile is great for silicon valley startups and university projects, but when big money or risk is in the balance, red tape renders it useless.

That's why it's an enormous advantage in some enterprise environments to never have to recompile to change out a set of classes, and why clunky old xml is great. It's not about the code, it's about politics and risk management.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=193080#respond)

10. July 2014 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-193640) von

Gregor Elke (http://greelgorke.tumblr.com)

I'm not sure, but somehow I have a feeling, that you think all that reviews, paperworks etc. are useless junk, which is understandable, because it's frustrating sometimes. But all this "junk" has some value, be it risk reduction or political convenience. I would agree on the point, that most of it might be obsolete, IF the organization is ready to get over it.

But what I'm concerned about is the mindset, in which a configuration change bears no risk. Ask your Ops if they think the same, either they don't or I'd be worried.

To be clear: i'm not saying java config is better than xml. I'm saying, it doesn't matter how, you should your config under version control and test it.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=193640#respond)

31. July 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-75571) von **SD (http://saarvi.com)**

I will give this a try. Thanks for this elaborated post on injection styles...

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=75571#respond)

3. August 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-76248) von Oussama ZOGHLAMI

Hi Tobias,

It's nice to use Java Based Configuration, but it has also some disadvantages. For example, if we want to change an implementation, we should recompile the project, which is not the case with xml configuration.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=76248#respond)

9. February 2014 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-165185) von DS

I completely agree. This is the main reason I think Java based configuration is crazy.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=165185#respond)

24. March 2015 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-328046) von Brooke

Ummm .... configuration files. They can hold the actual values and live outside the app if needed.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=328046#respond)

4. August 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-76577) von Michal

Tobias,

I had very similar opinion on XML configs while I was using Eclipse. Later I switched to IntelliJ and now I see that even XML config is perfectly OK with proper tooling. Eclipse/STS still have sooo much to improve here.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=76577#respond)

22. October 2012 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-106450) von **Java programmer (http://java67.blogspot.co.nz)**

Java Configuration is good, specially because you can navigate configuration using IDE but I guess production system may not have IDE installed, so you got to rely on plain old VI editor.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=106450#respond)

27. May 2014 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-178370) von Aby

Hello Tobias,

In your view point java based DI is good. But in real scenario I am not agree with that , I choose xml configuration since it is configurable one

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=178370#respond)

11. June 2014 (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/#comment-184098) von Binh Thanh Nguyen

Thanks, nice tips.

Reply (https://blog.codecentric.de/en/2012/07/spring-dependency-injection-styles-why-i-love-java-based-configuration/?replytocom=184098#respond)

Comment

Nachricht

Name

E-Mail

SUBMIT

Notify me of followup comments via e-mail

Newsletter

E-Mail Adresse

SUBMIT

(https://www.facebook.com/codecentric) (https://twitter.com/codecentric)

IMPRINT (HTTPS://WWW.CODECENTRIC.DE/IMPRESSUM-DATENSCHUTZ/) PRIVACY POLICY (HTTPS://WWW.CODECENTRIC.DE/TERMS-AND-CONDITIONS/)

CONTACT (HTTPS://WWW.CODECENTRIC.DE/UEBER-CODECENTRIC/KONTAKT/)