



# DI Styles: Choosing the Right Tool for the Job

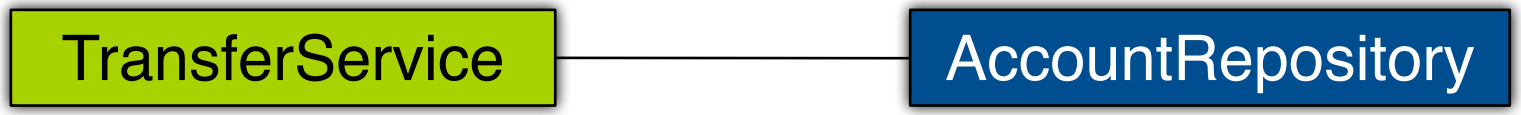
Chris Beams - SpringSource

Mark Pollack - SpringSource

# Hello!

- Mark Pollack
  - Principal at SpringSource
  - Founder Spring.NET
  - Core Spring committer
- Chris Beams
  - Senior Consultant at SpringSource
  - Lead Spring JavaConfig
  - Core Spring committer
  - Trained hundreds to use Spring

# Dependency Injection: *Simple, Right?*




```
public TransferService(AccountRepository ar) {
        this.accountRepository = ar;
}
```

# Choices

@Configuration

@Autowired

<context:annotation-config/>

<context:component-scan/> @Inject @Bean

@Component

@Configurable

<bean/>

# Where we're headed...

- A brief history of Dependency Injection
- Seven characteristics of a DI style
- A demo-centric tour of DI styles
  - What's new in Spring 3.0 for DI
  - All demo sources will be available at
  - https://src.springsource.org/svn/springone2gx/distyles

# A one-slide history of DI

- 2000: Fowler, et al coin 'POJO'
- 2002: Johnson, et al: 1:1 J2EE; J2EE w/o EJB
- 2002-3: Spring and other 'lightweight IoC containers' emerge
- 2004: Fowler coins 'Dependency Injection' as a specialization of the Inversion of Control principle
  - Defined three 'types' of DI
    - Constructor Injection
    - Setter Injection
    - Interface Injection
- 2004-present: Spring evolves; DI is widely adopted

# DI: Why?

- Dependency Injection enables 'POJO programming'
- POJO programming facilitates
  - Simplicity
  - Effective separation of concerns
  - Proper unit testing
- Well-factored and well-tested code
  - Tends to be well-designed code
  - Evolves well into supple, maintainable systems

# DI: Where to Inject?

- Three possible 'Injection Points'
  - Constructor
    - Good for mandatory dependencies
  - Setter
    - Good for optional dependencies
  - Field
    - Good for injecting system under test into JUnit test

# DI: How to Configure?

- Styles for expressing DI metadata and instructions
  - External
    - Configuration files (XML, properties, ...)
    - Code (Java)
    - DSL (Spring XML namespaces, Groovy)
  - Internal
    - Annotations embedded within POJOs
    - Usually requires at least some external configuration

# DI: Evolution

- We've come a long way since Fowler first defined DI
- Today, developers are faced with many choices
  - The introduction of annotations changed the game
  - The rise of non-Java languages introduces new possibilities

# DI: Today's Choices

- Open-source projects
  - Spring
  - Grails BeanBuilder
  - Google Guice
  - *Many* other projects across languages
- Standards efforts
  - JSR-250 (Common Annotations)
  - JSR-299 (Java Contexts and Dependency Injection)
  - JSR-330 (Dependency Injection for Java)
  - OSGi 4.2 Blueprint Container


# Choosing a DI Style

# DI configuration:
# *What matters to you?*

- Let's begin with defining the characteristics that matter when thinking about DI
- Provide a framework for making decisions about your own applications

# Seven Characteristics of a DI Configuration Style

1. External vs. Internal
2. Explicit vs. Implicit
3. Type-safety
4. Invasiveness
5. Portability (of POJOs)
6. Configurability of 3rd party components
7. Toolability

# Characteristic 1: *External vs. Internal*

- External DI is noninvasive
  - But causes context switching during coding
  - More verbose
  - Provides a 'blueprint' of your application
- Internal DI is necessarily invasive
  - May or may not be portable
  - But requires less coding and maintenance
  - Ease of use during development

# Characteristic 1:
# *External vs. Internal*

**External**

```
<bean id="transferService" class="com.bank.TransferServiceImpl">
   <constructor-arg ref="accountRepository" />
</bean>
```

**Internal**

```
@Component
public class TransferServiceImpl implements TransferService {
   @Autowired
   public TransferServiceImpl(AccountRepository repo) {
      this.accountRepository = repo;
   }
   ...
}
```

# Characteristic 2:
# *Explicit vs. Implicit*

- Explicit DI comes at a greater verbosity cost
  - More tedious in simple cases
  - Easier when things get complicated
- Implicit DI introduces the possibility of ambiguity
  - If multiple implementations of a given type are scanned
  - Disambiguation strategies are required
  - But ambiguities may arise at any time

# ***Explicit*** *vs. Implicit*

```
<bean id="transferService" class="com.bank.TransferServiceImpl">
   <constructor-arg ref="accountRepository" />
   <constructor-arg ref="feePolicy" />
</bean>

<bean id="accountRepository" class="com.bank.JdbcAccountRepository">
<bean id="feePolicy" class="com.bank.FlatFeePolicy">
```

```
public class TransferServiceImpl implements TransferService {

   public TransferServiceImpl(AccountRepository accountRepository,
                              FeePolicy feePolicy) {
      …
   }
   …
}
```

# *Explicit vs.* ***Implicit***

```
<context:component-scan base-package="com.bank">

@Component
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl(AccountRepository accountRepository,
                               FeePolicy feePolicy) { … }
    …
}

@Component
public class JdbcAccountRepository implements AccountRepository{ … }

@Component
public class FlatFeePolicy implements FeePolicy{ … }
```

# *Explicit vs.* ***Implicit****: Ambiguity*

```
<context:component-scan base-package="com.bank">

@Component
public class TransferServiceImpl implements TransferService {
   @Autowired
   public TransferServiceImpl(AccountRepository accountRepository,
                              FeePolicy feePolicy) { … }
   …
}

@Component
public class JdbcAccountRepository implements AccountRepository{ … }

@Component
public class FlatFeePolicy implements FeePolicy{ … }

@Component
public class VariableFeePolicy implements FeePolicy{ … }
```

**Which one should get injected?**

# Implicit DI: *Disambiguation*

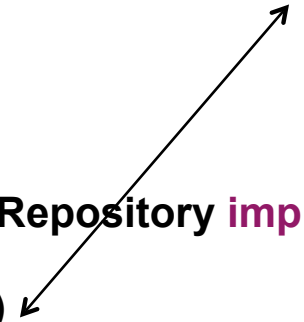
```
<context:component-scan base-package="com.bank">

@Component
public class TransferServiceImpl implements TransferService {
  @Autowired
   public TransferServiceImpl(AccountRepository repo,
                              @Qualifer("domestic") FeePolicy feePolicy) { … }
   …
}

@Component
public class JdbcAccountRepository implements AccountRepository{ … }

@Component("domestic")
public class FlatFeePolicy implements FeePolicy{ … }

@Component("international")
public class VariableFeePolicy implements FeePolicy{ … }
```



# Characteristic 3: *Type-safety*

- XML is inherently not type-safe
  - Tooling can mitigate this
    - STS/IDEA/Netbeans are Spring XML-aware and can contribute warnings/errors at development time
- How can get the compiler to catch errors in DI configuration?
  - Custom @Qualifier annotations
  - @Configuration classes

# Characteristic 4: *Invasiveness*

- Originally, noninvasiveness was a defining characteristic of DI and POJO programming
- Noninvasiveness matters because
  - An object should be usable independent of its environment and context
  - Especially important when it comes to testing
- but...
  - Annotations changed this

# Annotations and Invasiveness

- Annotations, by definition, are invasive
  - Requires modifying POJOs
- But we may say they are *minimally invasive*
  - Because annotations have no detrimental effect on the *utility* of a POJO
  - Java 6 allows for annotations to be missing at runtime
- *Non-standard* annotations impact POJO portability

# Characteristic 5: *Portability*

- Ideally, a POJO should be reusable across DI frameworks
- Non-standard annotations tie you to a framework
  - Hence the need for standardization
  - JSR-330!

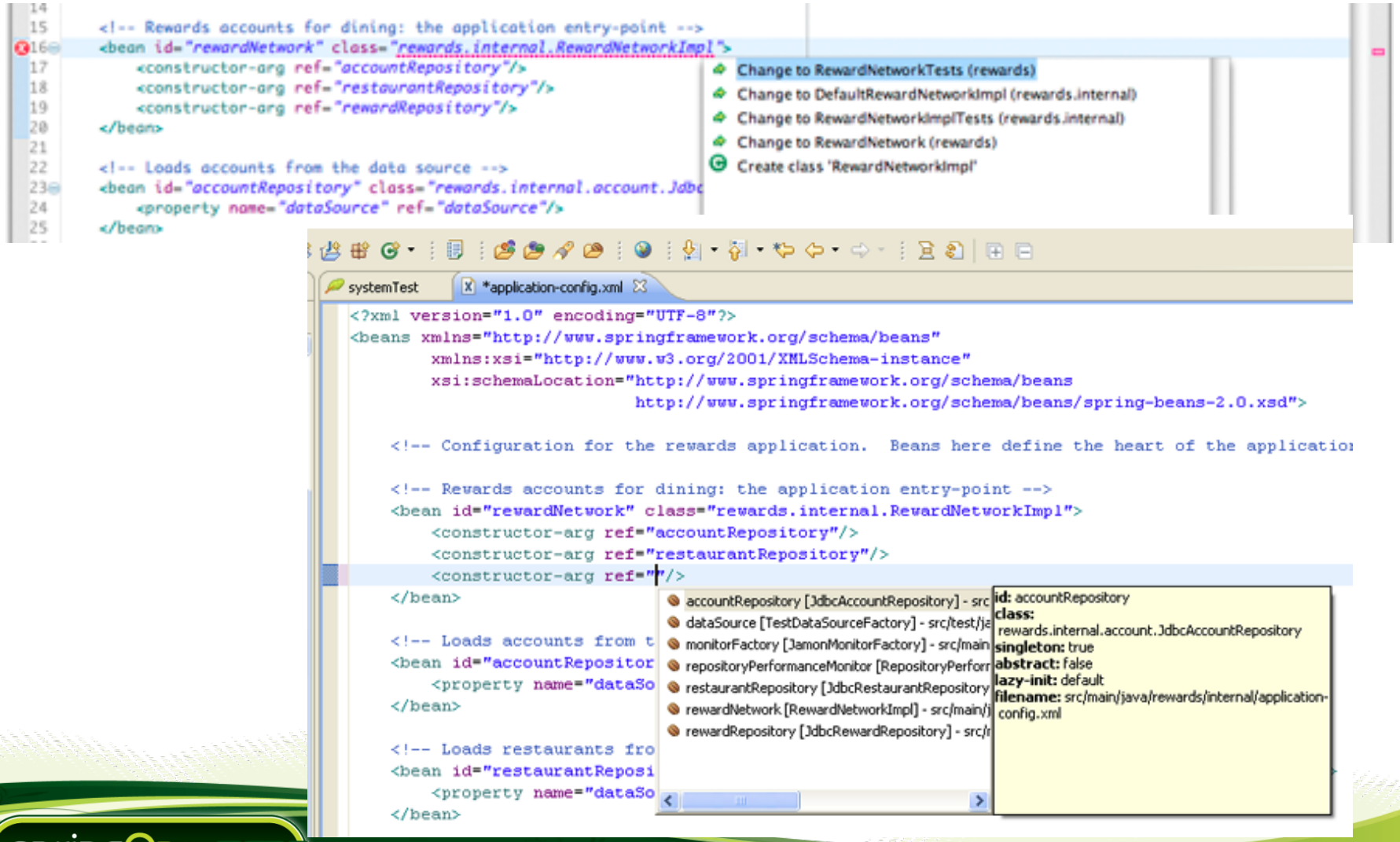
# Characteristic 6: *Configurability of 3rd Party Components*

- Internal configuration and 3rd party components don't mix
  - You can't annotate somebody else's code
- External configuration is the only way
- Hence, a complete DI solution must support both

# Characteristic 7: *Toolability*

- Natural, integrated refactoring
  - XML-driven DI requires Spring-aware tooling
  - Code-driven DI takes advantage of built-in tooling
- Content assist in configuration files
  - Generic XML tooling only gets you so far
  - Need XML tooling built to purpose
- Visualization
  - Seeing the 'blueprint' of your application
- Static analysis
- Obfuscation

# SpringSource ToolSuite




# A Tour of DI Styles

# DI Styles

- XML
  - <beans/>
  - <namespace:*/>
- @Autowired
- @Configuration
- Standards
  - JSR-250 (Common Annotations)
  - JSR-299 (Java Contexts and Dependency Injection)
  - JSR-330 (Dependency Injection for Java)

# DI Styles

- *Remember...*
  - DI styles need not be mutually exclusive!
  - You'll probably use more than one in a given app


# <beans/>

# <beans/> XML

- The original DI style for Spring
- Remains very widely used
- General-purpose, thus very powerful
- But can get verbose


# <beans/> demo

# <beans/>: *Summary*

- External vs. Internal: **External**
- Explicit vs. Implicit: **Explicit**
- Type-safe: **No**
- Invasive: **No**
- Portable: **Yes**
- Can configure 3rd party: **Yes**
- Has tooling support: **Yes**


# <namespace:*/>

# <namespace:*/> XML

- Introduced in Spring 2.0
- Expanded in Spring 2.5 and 3.0
- Widely adopted by Spring projects
  - Spring Integration
  - Spring Batch
  - Spring Web Flow
  - Spring DM
  - Spring Security
- Greatly reduces verbosity
- More expressive at the same time


# <namespace:*/> demo

# <namespace:*/>: *Summary*

- Same fundamental characteristics as <beans/>
- But eliminates the XML verbosity problem
  - Serves as a 'configuration DSL'
- Not just about DI
  - Helps manage many other aspects of the application
  - Scheduling, aop, etc.


# @Autowired

# @Autowired

- AKA "Annotation-driven injection"
- Introduced in Spring 2.5
- More annotations added in Spring 3.0
  - @Primary
  - @Lazy
  - @DependsOn
  - extended semantics for @Scope
- Widely used today
- Works in conjunction with @Component and <context:component-scan/> to streamline development lifecycle


# @Autowired demo

# @Autowired: *Summary*

- External vs. Internal: **Internal**
- Explicit vs. Implicit: **Implicit**
- Type-safe: **Yes**
- Invasive: **Yes**
- Portable: **No**
- Can configure 3rd party: **No**
- Has tooling support: **Yes** (as of STS 2.2.0)


# JSR-330 (@Inject)

# Introducing JSR-330

- AKA *@Inject*
- Packaged under javax.inject.*
- A joint effort by Google and SpringSource
- Provides portable DI annotations
- JSR went final two weeks ago
  - API is available in Maven central
- Spring 3.0 support passes the TCK ... *as of today!*

# Smallest. JSR. Ever.

| Interface Summary | |
|---|---|
| Provider<T> | Provides instances of T. |

| Annotation Types Summary | |
|---|---|
| Inject | Identifies injectable constructors, methods, and fields. |
| Named | String-based qualifier. |
| Qualifier | Identifies qualifier annotations. |
| Scope | Identifies scope annotations. |
| Singleton | Identifies a type that the injector only instantiates once. |


# JSR-330 demo

# JSR-330: *Summary*

- External vs. Internal: **Internal**
- Explicit vs. Implicit: **Undefined!** (can be either)
- Type-safe: **Yes**
- Invasive: **Yes**
- Portable: **Yes**
- Can configure 3rd party: **No**
- Has tooling support: **Not yet**

# @Autowired and @Inject: *The Bottom Line*

- JSR-330 standardizes internal DI annotations
  - Meaning: portable POJOs
- However, @Inject is a subset of the functionality provided by Spring's @Autowired
- Rule of thumb
  - You can get 80% of what you need with @Inject
  - Rely on @Autowired and friends for the other 20%

# From @Autowired to @Inject

| Spring | javax.inject.* | |
|---|---|---|
| @Autowired | @Inject * | @Inject has no 'required' attribute |
| @Component | @Named * | Spring supports scanning for @Named |
| @Scope | @Scope * | for meta-annotation and injection points only |
| @Scope ("singleton") | @Singleton * | jsr-330 default scope is like Spring's 'prototype' |
| @Qualifier | @Qualifier, @Named | |
| @Value | *no equivalent* | *see SPR-6251 for ideas on how Spring can help bridge this gap* |
| @Primary | *no equivalent* | |
| @Lazy | *no equivalent* | |
| @Required | *no equivalent* | |


# @Configuration

# @Configuration

- Formerly *Spring JavaConfig*
- Now included in core Spring Framework 3.0
- Annotation-driven, but is an *external* DI style
  - POJOs remain untouched by annotations
- Full programmatic control
- Allows for object-oriented configuration
- Integrates well with other Spring DI styles

# A @Configuration class

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource());
    }

    // ...
}
```

# Look familiar?

```
<bean id="transferService" class="com.bank.TransferServiceImpl">
   <constructor-arg ref="accountRepo" />
</bean>

<bean id="accountRepo" class="com.bank.JdbcAccountRepository">
   <constructor-arg ref="dataSource" />
</bean>
```

# Bootstrapping

```
public class Bootstrap {

    public static void main(String... args) {
        ApplicationContext ctx =
            new ConfigurationClassApplicationContext(AppConfig.class);

        TransferService transferService = ctx.getBean(TransferService.class);

        transferService.transfer(100.00, “A123”, “C456”);
    }

}
```


# @Configuration demo

# @Configuration

- External vs. Internal: **External**
- Explicit vs. Implicit: **Explicit**
- Type-safe: **Yes**
- Invasive: **No**
- Portable: **Yes**
- Can configure 3rd party: **Yes**
- Has tooling support: **Yes**


# Groovy/Grails BeanBuilder

# BeanBuilder

- DSL for creating Spring BeanDefinitions
- Currently part of Grails
- Work is underway to separate BeanBuilder from Grails for standalone use in any Groovy / Java app

# BeanBuilder at a Glance

```
import org.springframework.context.ApplicationContext
import grails.spring.BeanBuilder
...
def bb = new BeanBuilder()

bb.beans {
   transferService(TransferServiceImpl, accountRepository, feePolicy)
   accountRepository(JdbcAccountRepository, dataSource)
   dataSource(BasicDataSource) {
      driverClassName = "org.hsqldb.jdbcDriver"
      url = "jdbc:hsqldb:mem:grailsDB"
      username = "sa"
      password = ""
   }
}

ApplicationContext ctx = bb.createApplicationContext();
TransferService transferService = ctx.getBean(TransferService);
```


# BeanBuilder demo

# Groovy BeanBuilder

- External vs. Internal: **External**
- Explicit vs. Implicit: **Explicit**
- Type-safe: **No**
- Invasive: **No**
- Portable: **Yes**
- Can configure 3rd party: **Yes**
- Has tooling support: **No**

# Spring is about Choice

- Choice in many areas…
- All DI metadata (internal or external) contributes to the core *BeanDefinition* model
  - This layer is what makes adding different configuration styles possible!


# Q&A


# Thank you!

https://src.springsource.org/svn/springone2gx/distyles

http://twitter.com/javaconfig