Annotations and Component Scanning

Annotation-based configuration

1.18.5

Objectives

After completing this lesson, you should be able to do the following

- Explain and use Annotation-based Configuration
- Discuss Best Practices for Configuration choices
- Use **`@PostConstruct`** and **`@PreDestroy`**
- Explain and use "*Stereotype*" Annotations

Agenda

- **Annotation-based Configuration**
- Best Practices
- @PostConstruct, @PreDestroy
- Stereotypes, Meta Annotations
- Lab
- Optional topics:
  @Resource, JSR 330

Before – *Explicit* Bean Definition (covered in the Previous Module)

- Configuration is external to bean-class
  - *Separation of concerns*
  - Java-based dependency injection

```
@Configuration
public class TransferModuleConfig {

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

   @Bean public AccountRepository accountRepository() {
         …
   }
}
```

Dependency Injection

After - *Implicit* Configuration (Covered in this module)

- Annotation-based configuration *within* bean-class
- Component-scanning

Annotations embedded with POJOs

```
@Component
public class TransferServiceImpl implements TransferService {

      public TransferServiceImpl(AccountRepository repo) {
          this.accountRepository = repo;
      }
}
```

Dependency Injection

```
@Configuration
@ComponentScan ( “com.bank” )
public class AnnotationConfig {
   // No bean definition needed any more
}
```

Find @Component annotated classes within designated (sub)packages

Dependency Injection via @Autowired

Unique dependency of correct **type** *must* exist

- Constructor-injection (recommended practice)

```
@Autowired  // Optional if this is the only constructor
public TransferServiceImpl(AccountRepository a) {
    this.accountRepository = a;
}
```

- Method-injection

```
@Autowired
public void setAccountRepository(AccountRepository a) {
   this.accountRepository = a;
}
```

- Field-injection

```
@Autowired
private AccountRepository accountRepository;
```

Even when field is private!!
– *but* hard to unit test, see URL

http://olivergierke.de/2013/11/why-field-injection-is-evil/

@Autowired Dependencies: Required or Optional?

- Default behavior: required

```
@Autowired
public void setAccountRepository(AccountRepository a) {
   this.accountRepository = a;
}
```

Exception if no dependency found

- Use *required* attribute to override default behavior

```
@Autowired(required=false)
public void setAccountRepository(AccountRepository a) {
   this.accountRepository = a;
}
```

Only inject *if* dependency exists

Autowiring and Disambiguation – 1

```
@Component
public class TransferServiceImpl implements TransferService {
  @Autowired // optional if there is a single no-arg constructor
  public TransferServiceImpl(AccountRepository accountRepository)   { … }
}
```

```
  @Component
  public class JpaAccountRepository implements AccountRepository {..}
```

Which one should get injected?

```
  @Component
  public class JdbcAccountRepository implements AccountRepository {..}
```

At startup: *NoSuchBeanDefinitionException*, no unique bean of type [AccountRepository] is defined: expected single bean but found 2...

Autowiring and Disambiguation – 2

- Use of the @Qualifier annotation

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
  @Autowired
  public TransferServiceImpl( @Qualifier("jdbcAccountRepository")
                                  AccountRepository accountRepository)  { … }
```

qualifier

```
  @Component("jdbcAccountRepository")
  public class JdbcAccountRepository implements AccountRepository {..}
```

```
  @Component("jpaAccountRepository")
  public class JpaAccountRepository implements AccountRepository {..}
```

bean ID

@Qualifier also available with method injection and field injection

Autowiring and Disambiguation – 3

Autowired resolution rules

1. Look for unique bean of required *type*
2. Use @Qualifier if supplied
3. Try to find a matching bean by *name*

Example

We have multiple *Queue* beans

Spring finds bean with id matching what is being set: “**ack**”

```
@Autowired
public MyBean(Queue ack) {
    …
}
```

```
@Autowired
public void setQueue(Queue ack) {
    …
}
```

```
@Autowired
private Queue ack;
```

Looks for Queue bean with id = “**ack**”

Component Names

- When not specified
  - Names are auto-generated
    - De-capitalized non-qualified class name by default
    - *But* will pick up implementation details from class name
  - *Recommendation:* never rely on generated names!

- When specified
  - Allow disambiguation when 2 bean classes implement the same interface

Common strategy: avoid using qualifiers when possible.
Usually rare to have 2 beans of same type in ApplicationContext

Annotations syntax vs Java Config

- Similar options are available

```
@Component("transferService")
@Scope("prototype")
public class TransferServiceImpl
        implements TransferService {
  @Autowired
  public TransferServiceImpl
        (AccountRepository accRep) { … }

}
```

Annotations

```
@Configuration
public class TransferConfiguration

  @Bean(name="transferService")
  @Scope("prototype")
  public TransferService tsvc() {
    return
      new TransferServiceImpl(
              accountRepository());
  }
}
```

Java Configuration

Agenda

- Annotation-based Configuration
- **Best Practices**
- @PostConstruct, @PreDestroy
- Stereotypes, Meta Annotations
- Lab
- Optional topics:
  @Resource, JSR 330

Autowiring Constructors

- If a class *only* has a default constructor
  - Nothing to annotate
- If a class has *only one* non-default constructor
  - It is the only constructor available, Spring will call it
  - **`@Autowired`** is optional
- If a class has *more than one* constructor
  - Spring invokes zero-argument constructor by default (if it exists)
  - Or you *must* annotate with **`@Autowired`** the one you want Spring to use

In our examples we use @Autowired, *even when it is optional*, so that you can see Dependency Injection happening.

About Component Scanning

- Components are scanned at startup
  - JAR dependencies also scanned!
  - Could result in slower startup time if too many files scanned

Component Scanning **Best Practices**

- Really bad:

```
@ComponentScan ( { "org", "com" } )
```

All "org" and "com" packages in the classpath will be scanned!!

- Still bad:

```
@ComponentScan ( "com" )
```

- OK:

```
@ComponentScan ( "com.bank.app" )
```

- Optimized:

```
@ComponentScan ( { "com.bank.app.repository",
                   "com.bank.app.service", "com.bank.app.controller" } )
```

Agenda

- Annotation-based Configuration
- Best Practices
- **@PostConstruct, @PreDestroy**
- Stereotypes, Meta Annotations
- Lab
- Optional topics:
  @Resource, JSR 330

@PostConstruct and @PreDestroy

- Add behavior at startup and shutdown

```
public class JdbcAccountRepository {
    @PostConstruct
    void populateCache() { }

   @PreDestroy
    void flushCache() { }
}
```

Method called at *startup* after all dependencies are injected

Method called at *shutdown* prior to destroying the bean instance

Annotated methods can have any visibility but *must* take *no* parameters and *only* return *void.*

@PostConstuct & @PreDestroy

- Beans are created in the usual ways:
  - Returned from @Bean methods
  - Found and created by the component-scanner
- Spring then invokes these methods *automatically*
  - During bean-creation process

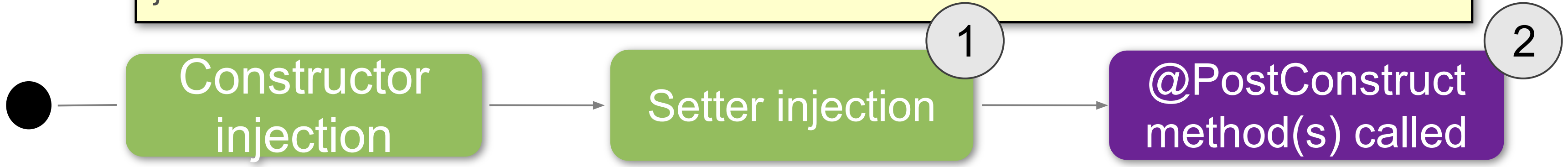
- These are not Spring annotations
  - Defined by JSR-250, part of Java since Java 6
  - In **`javax.annotation`** package
  - Supported by Spring, *and* by Java EE

@PostConstruct

- Called after setter injections are performed

```
public class JdbcAccountRepository {
      private DataSource dataSource;
      @Autowired
      public void setDataSource(DataSource dataSource)            1
         {  this.dataSource = dataSource;   }


      @PostConstruct
      public void populateCache()                                     2
         {   Connection conn = dataSource.getConnection(); //...   }
}
```



@PreDestroy

NOTE: PreDestroy methods called if application shuts down *normally*. ***Not** if the process dies or is killed.*

- Called when a ***`ConfigurableApplicationContext`*** is *closed*
  - Useful for releasing resources & 'cleaning up'
  - Not called for prototype beans

```
ConfigurableApplicationContext context = SpringApplication.run( ... );
...
// Trigger call of all @PreDestroy annotated methods
context.close();
```

Causes Spring to invoke this method

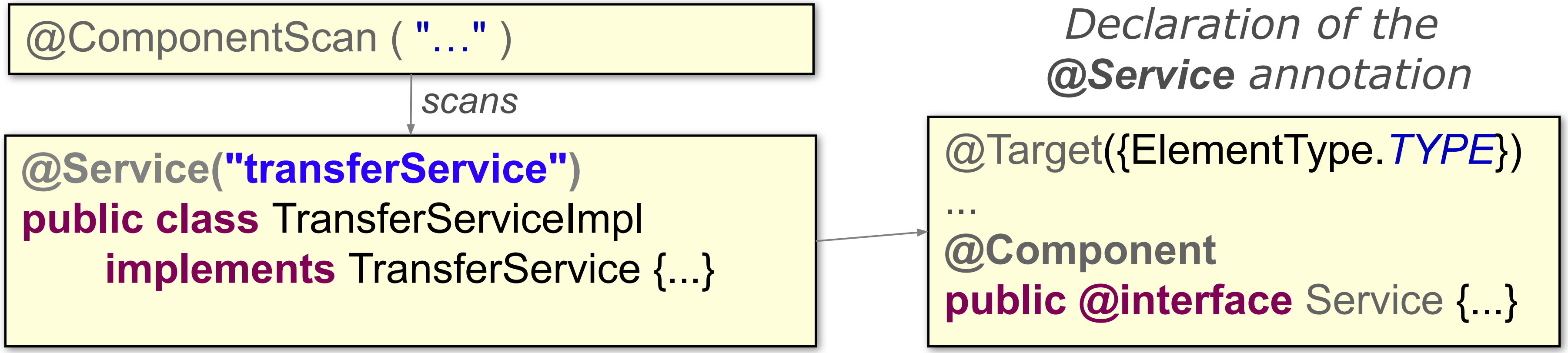
```
public class JdbcAccountRepository {
    @PreDestroy
    public void flushCache() { … }
    ...
```

Agenda

- Annotation-based Configuration
- Best Practices
- @PostConstruct, @PreDestroy
- **Stereotypes, Meta Annotations**
- Lab
- Optional topics:
  @Resource, JSR 330

Stereotype Annotations

- Component scanning also checks for annotations that are themselves annotated with @Component
  - So-called stereotype annotations



```
@ComponentScan ( "…" )
```

```
@Service("transferService")
public class TransferServiceImpl
      implements TransferService {...}
```

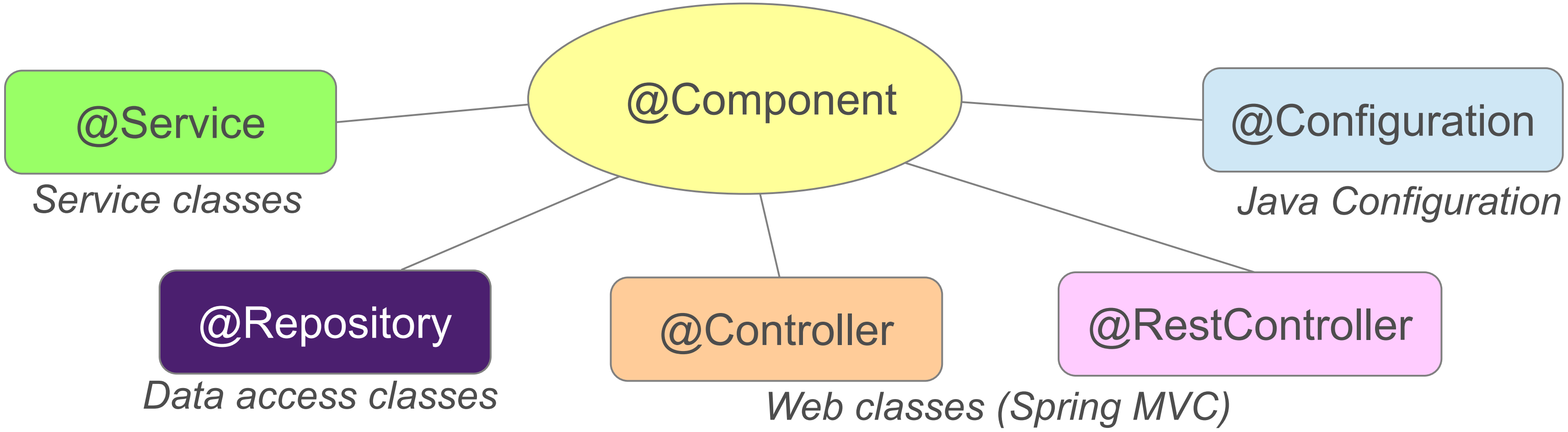
Declaration of the
***@Service** annotation*

```
@Target({ElementType.TYPE})
...
@Component
public @interface Service {...}
```

@Service annotation is part of the Spring framework

Predefined Stereotype Annotations

- Spring framework stereotype annotations



Other Spring projects provide their own stereotype annotations (Spring Integration, Spring Batch ...)

Summary

- Spring beans can be defined:
  - Explicitly using @Bean methods inside configuration class
  - Implicitly using @Component and component-scanning
- Applications can use both
  - Implicit for your classes
  - Explicit for the rest - prefer for large apps
- Can perform initialization and clean-up
  - Use @PostConstruct and @PreDestroy
- Use Spring's stereotypes and/or define your own meta annotations