Pivotal

Annotations and Component Scanning

Annotation-based configuration

spring by Pivotal

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

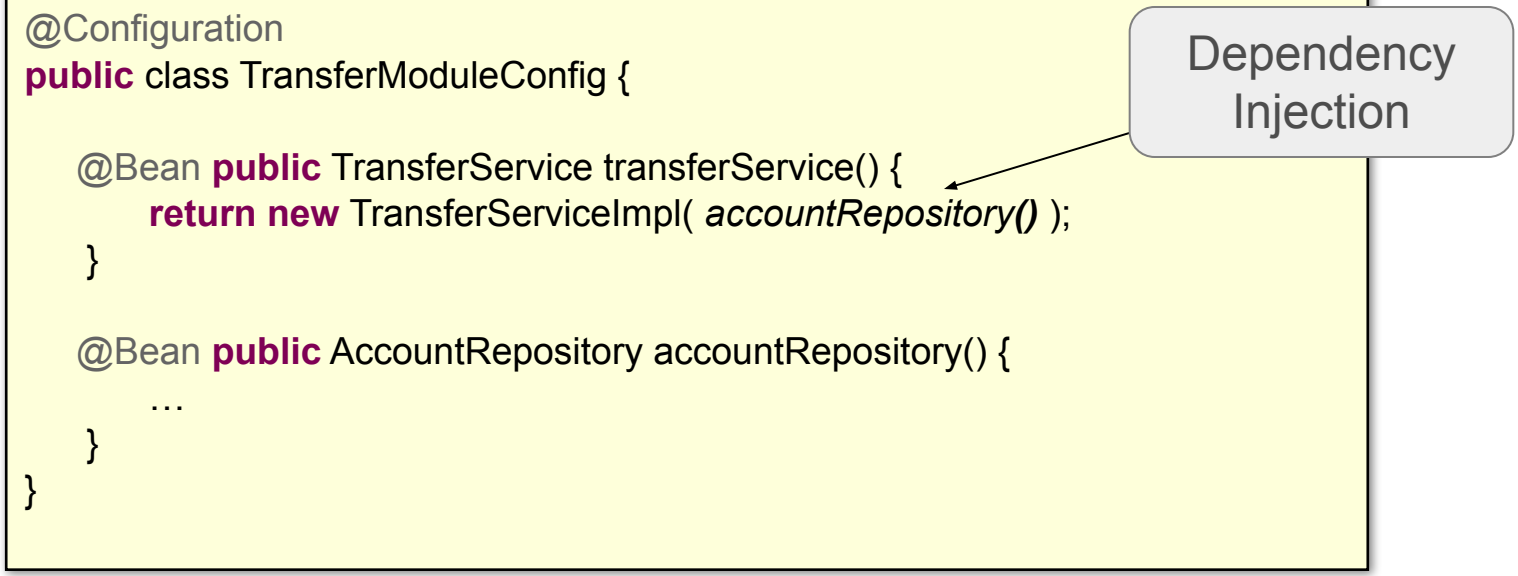
Before – *Explicit* Bean Definition

- 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() {
        …
    }
}
```



After - *Implicit* Configuration

- Annotation-based configuration w*ithin* bean-class
- Component-scanning

Bean id derived from classname: *transferServiceImpl*

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

Annotations embedded with POJOs

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

Find @Component classes within designated (sub)packages

Usage of @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

Java 8 Optional<T>

- Another way to inject optional dependencies
  - **`Optional<T>`** introduced to reduce null pointer errors

```
@Autowired(required=false)
public MyClass(AccountService
                    accountService){
}

public void doSomething() {
   if (accountService != null) {
      // do something
   }
}
```

```
@Autowired
public MyClass(Optional<AccountService>
                    accountService){
}

public void doSomething() {
   accountService.ifPresent( s ->  {
      // s is the AccountService instance,
      // use s to do something
   });
}
```

Note the use of the lamda

Constructor vs Setter Dependency Injection

- Spring doesn't care (can use either)
  - But which is better?

Constructors	Setters
Mandatory dependencies	Circular dependencies possible
Dependencies can be immutable	Dependencies are mutable
Concise (pass several params at once)	Could be verbose for several params
	Inherited automatically

- Follow the same rules as standard Java
  - Constructor injection is generally preferred
  - Be consistent across your project team
  - Many classes use both

Autowiring and Disambiguation – 1

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

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

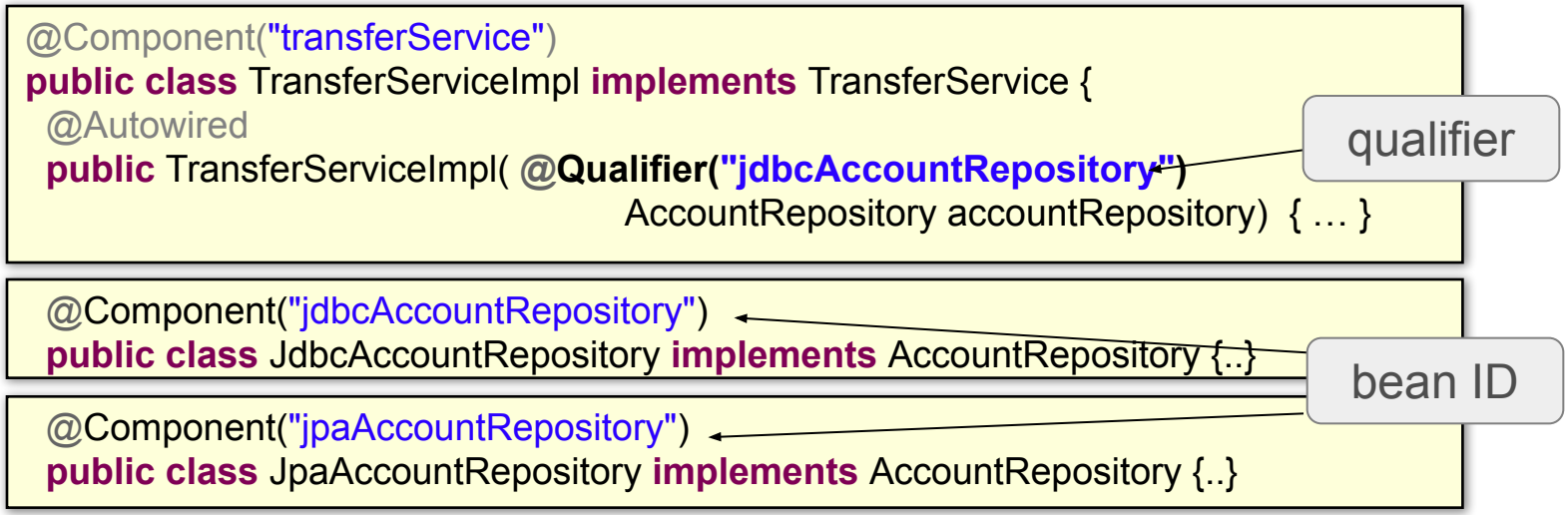
Which one should get injected?

```
@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



@Qualifier also available with method injection and field injection

Component names should *not* show implementation details *unless* there are 2 implementations of the *same* interface (as here)

Autowiring and Disambiguation – 3

- Autowired resolution rules
  - Look for unique bean of required *type*
  - Use @Qualifier if supplied
  - 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 classname by default
    - *But* will pick up implementation details from classname
  - *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

Using @Value to set Attributes

- Constructor-injection

Can use $ variables *or* SpEL

```
@Autowired // Optional if this is the only constructor
public TransferServiceImpl(@Value("${daily.limit}") int max) {
    this.maxTransfersPerDay = max;
}
```

- Method-injection

```
@Autowired
public void setDailyLimit(@Value("${daily.limit}") int max) {
   this.maxTransfersPerDay = max;
}
```

- Field-injection

```
@Value("#{environment['daily.limit']}")
int maxTransfersPerDay;
```

Not private so we can initialize in a unit-test

Delayed Initialization

Careful – often misused.
Most beans are *not* lazy.

- Beans normally created on startup when application context created
- Lazy beans created first time used
  - When dependency injected
  - By **`ApplicationContext.getBean`** methods
- Useful if bean's dependencies *not* available at startup

```
@Lazy @Component
public class MailService {
    public MailService(@Value("smtp:...") String url) {
        // connect to mail-server
    }
    ...
}
```

SMTP server may not be running when this process starts up

Annotations syntax vs Java Config

- Similar options are available

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

}
```

Annotations

```
@Configuration
public class TransferConfiguration
    @Bean(name="transferService")
    @Scope("prototype")
    @Profile("dev")
    @Lazy(true)
    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
    - Especially for large applications
    - A few seconds slower in the worst case
- What are the best practices?

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" } )
```

When to use what?

Java

Java Configuration

- Pros:
  - Is centralized in one (or a few) places
  - Write any Java code you need
  - Can unit-test the configuration class
  - Can be used for all classes (not just your own)
- Cons:
  - More verbose than annotations

When to use what?

Component Scanning

- Nice for your own beans
- Pros:
  - Single place to edit (just the class)
  - Allows for very rapid development
- Cons:
  - Configuration spread across your code base
    - Harder to debug/maintain
  - Only works for your own code
  - Mixing configuration and code (bad sep. of concerns)

Mixing Java Config and Annotations

- You can mix and match in many ways
- Common approach:
  - Use annotations whenever possible
    - Your classes
  - But still use Java Configuration for
    - Third-party beans that aren't annotated
    - Legacy code that can't be changed

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.*

About @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

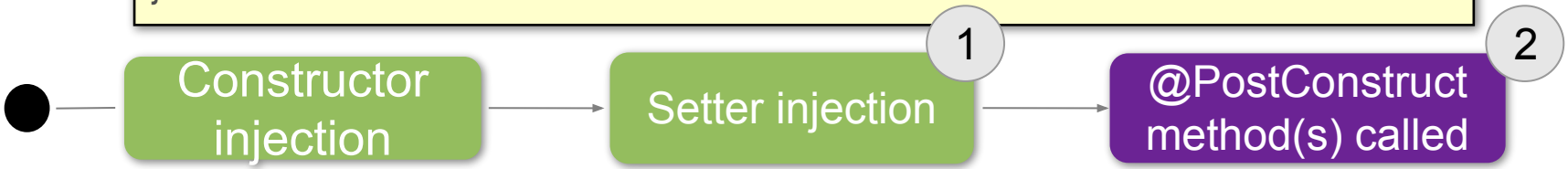
- Called after setter injections are performed

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

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

1

2



@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() { … }
    ...
```

Lifecycle Methods via @Bean

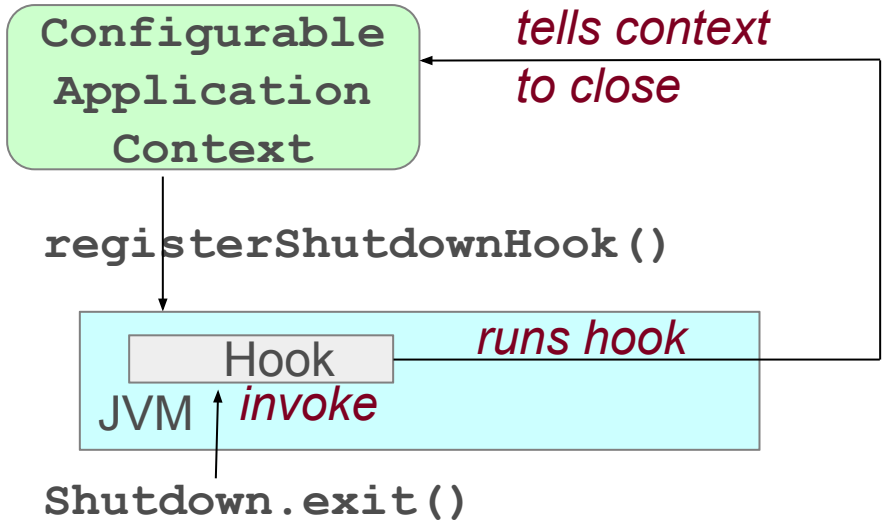
- Alternatively, **`@Bean`** has options to define these *life-cycle* methods

```
@Bean (initMethod="populateCache”, destroyMethod="flushCache")
public AccountRepository accountRepository() {
    // ...
}
```

- Common Usage:
  - **`@PostConstruct`**/**`@PreDestroy`** for your own classes
  - **`@Bean`** properties for classes you didn't write and can't annotate

Use a JVM Shutdown Hook

- Shutdown hooks
  - Automatically run when JVM shuts down
- **`SpringApplication.run`**
  - Does this automatically
  - Returns a **`Configurable-ApplicationContext`**



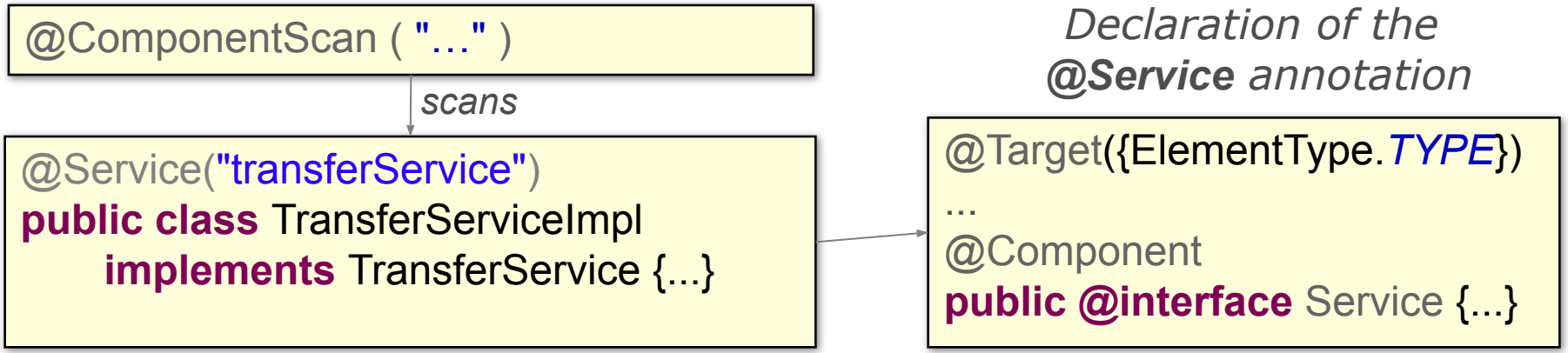

```
ConfigurableApplicationContext context = SpringApplication.run(...);
// Registered the shutdownHook for you
```

Agenda

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

Stereotype Annotations

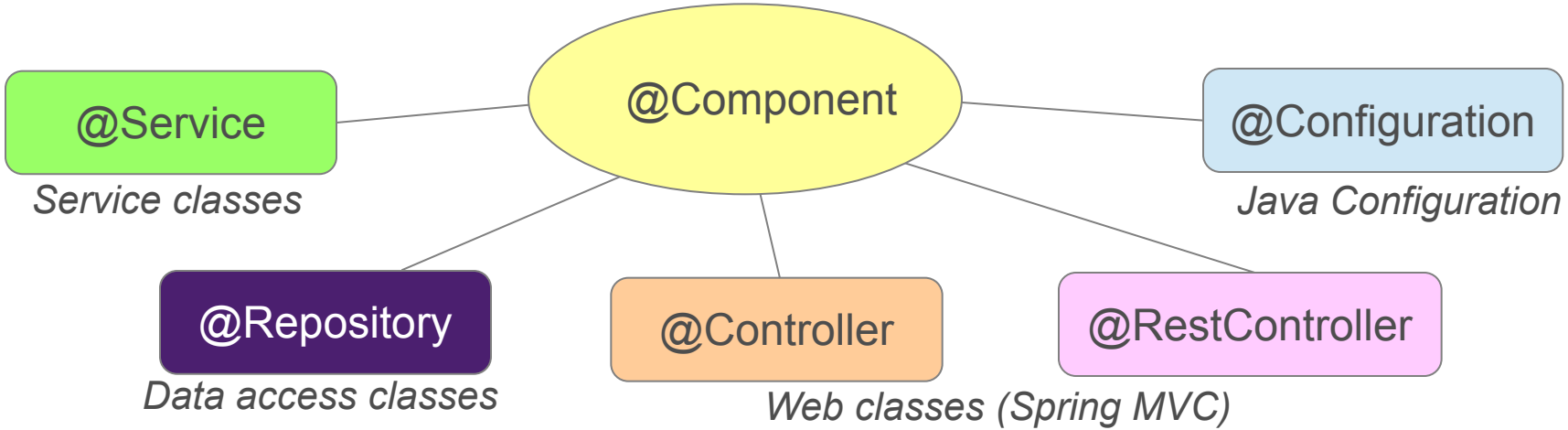
- Component scanning also checks for annotations that are themselves annotated with @Component
  - So-called stereotype annotations



@Service annotation is part of the Spring framework

Predefined Stereotype Annotations

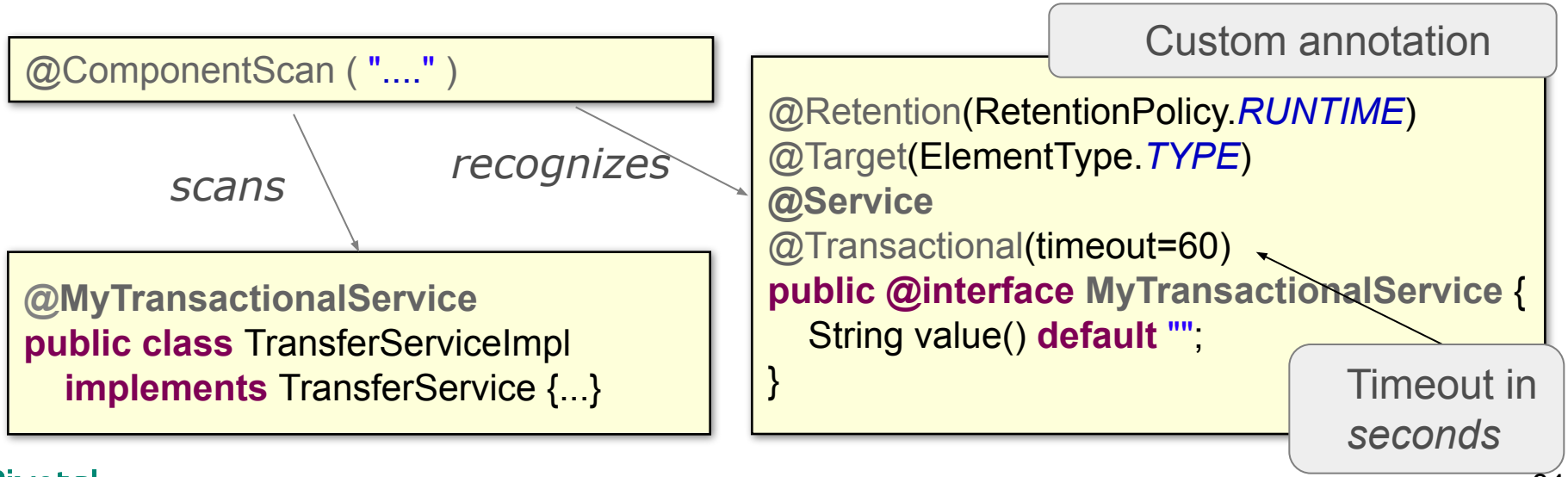
- Spring framework stereotype annotations



Other Spring projects provide their own stereotype annotations
(Spring Web-Services, Spring Integration...)

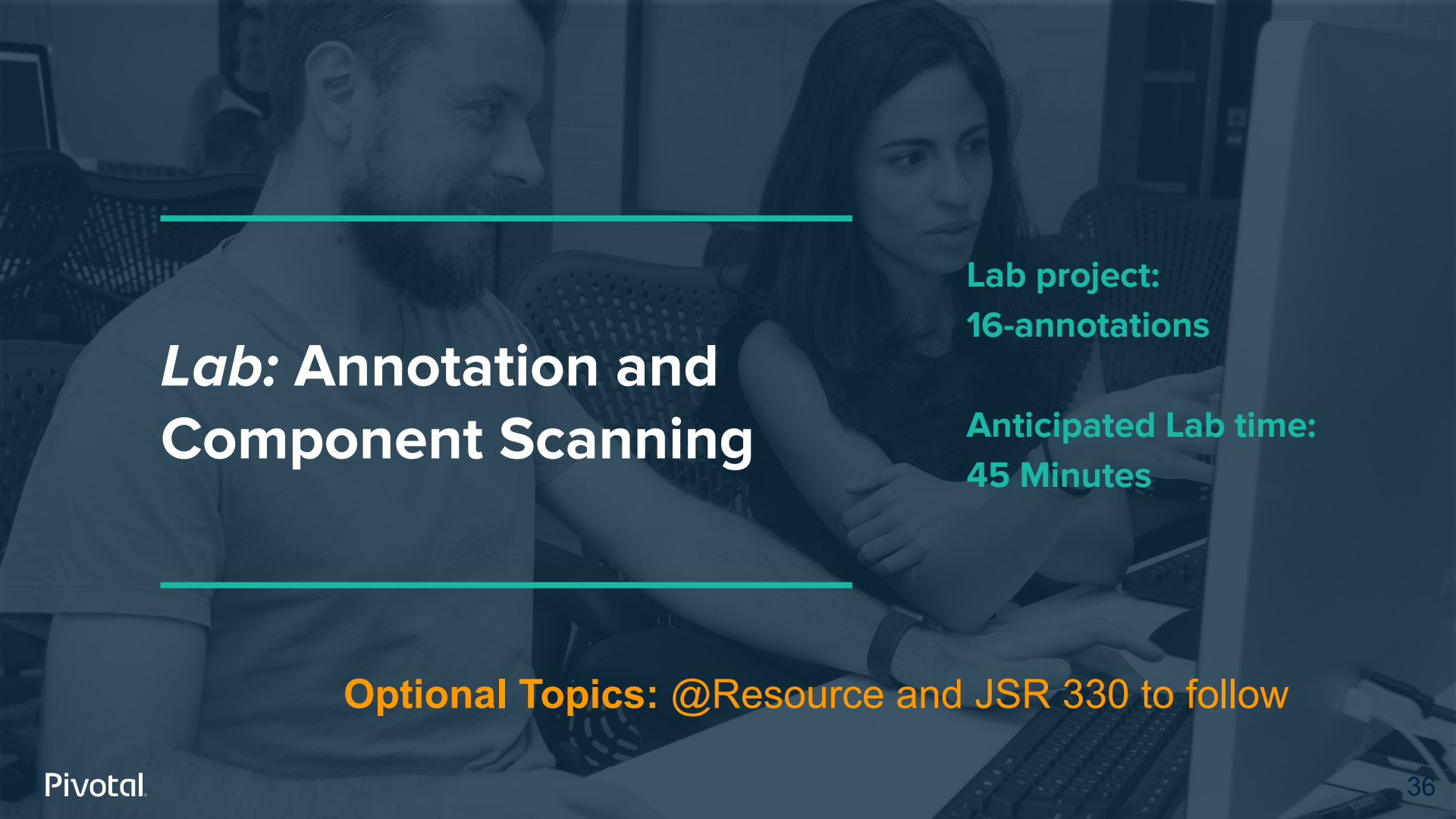
Meta-annotations

- Annotation which can be used to annotate other annotations
  - e.g. all service beans should be configurable using component scanning and be transactional



Summary

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



Lab: Annotation and Component Scanning

Lab project:
16-annotations

Anticipated Lab time:
45 Minutes

Optional Topics: @Resource and JSR 330 to follow

Agenda

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

Note: Advanced or Optional topics after Lab *not* required for Certification

Using @Resource

- From JSR-250, supported by EJB 3.0 and Spring
  - Identifies dependencies by *name*, not by *type*
    - *Name is Spring bean-name*
    - @Autowired matches by *type*
  - Supports setter and field injection *only*

```
@Resource(name="jdbcAccountRepository")
public void setAccountRepository(AccountRepository repo) {
  this.accountRepository = repo;
}
```

Setter Injection

```
@Resource(name="jdbcAccountRepository")
private AccountRepository accountRepository;
```

Field injection

Qualifying @Resource

@Autowired: type *then* name
@Resource: name *then* type

- When no name is supplied
  - Inferred from property/field name
  - Or falls back on injection by type
- Example
  - Looks for bean called *accountRepository*
    - because method is ***set****AccountRepository*
  - Then looks for bean of type *AccountRepository*

```
@Resource
public void setAccountRepository(AccountRepository repo) {
  this.accountRepository = repo;
}
```

JSR 330

- Java Specification Request 330
  - Also known as @Inject
  - Joint JCP effort by Google and SpringSource
  - Standardizes internal DI annotations
  - Published late 2009
    - Spring is a valid JSR-330 implementation
- Subset of functionality compared to Spring's @Autowired support
  - @Inject has 80% of what you need
  - Need @Autowired for the rest
  - Recommendation: use Spring annotations instead

JSR 330 annotations

```
@ComponentScan ( "...." )
```

Also scans JSR-330 annotations

```
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class TransferServiceImpl implements TransferService {
    @Inject
    public TransferServiceImpl( @Named("accountRepository")
            AccountRepository accountRepository)     { … }
}
```

Should be specified for component scanning (even without a name)

```
import javax.inject.Named;

@Named("accountRepository")
public class JdbcAccountRepository implements AccountRepository { ... }
```

From @Autowired to @Inject

Spring	JSR 330	Comments
@Autowired	@Inject	@Inject always mandatory, has no required option
@Component	@Named	Spring also scans for @Named
@Scope	@Scope	JSR 330 Scope for meta-annotation and injection points only
@Scope ("singleton")	@Singleton	JSR 330 default scope is like Spring's '*prototype*'
@Qualifer	@Named	
@Value	No equivalent	SpEL specific
@Required	Redundant	@Inject *always* required
@Lazy	No equivalent	Useful when needed, often abused