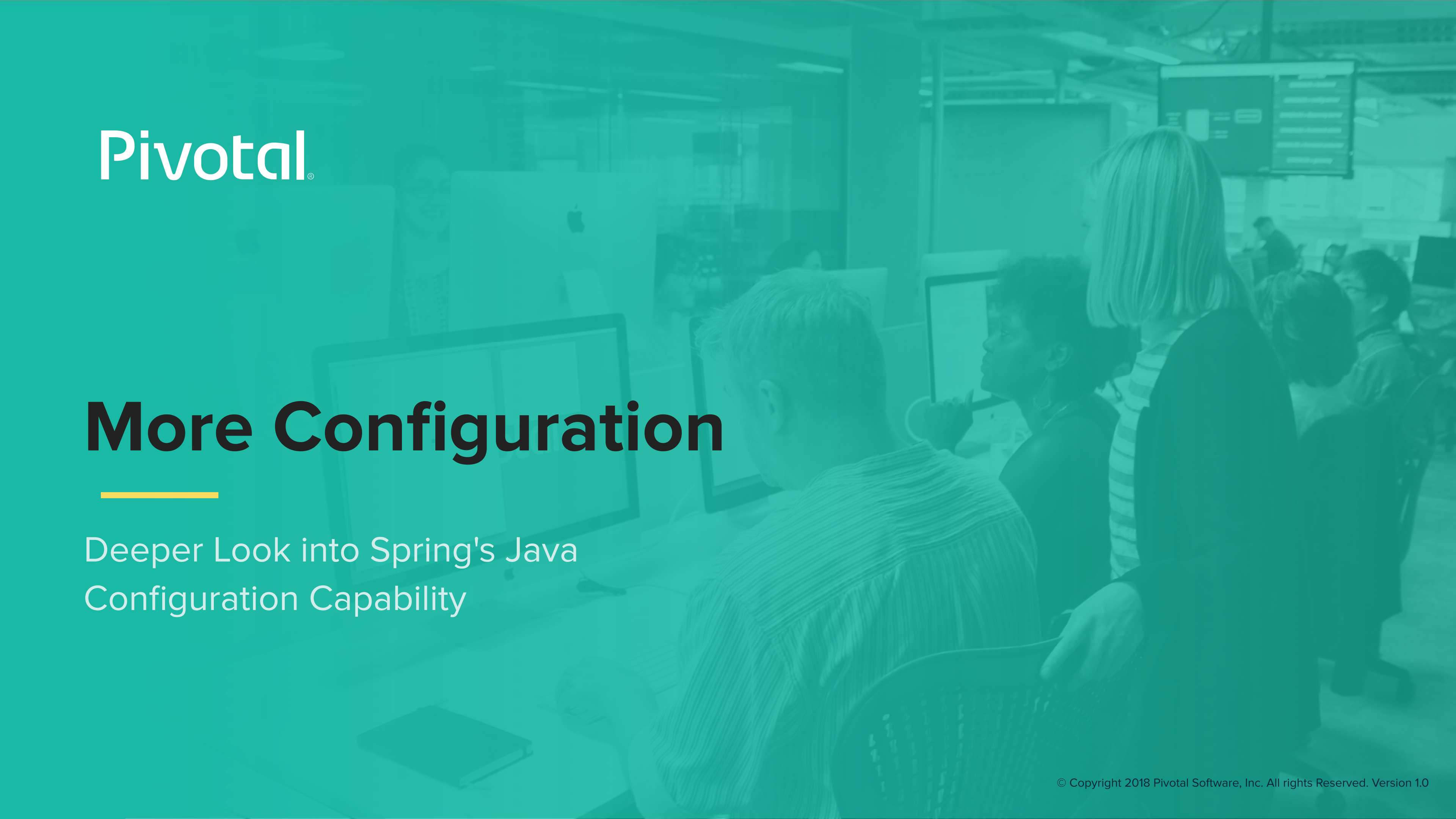


Pivotal

More Configuration

Deeper Look into Spring's Java Configuration Capability

© Copyright 2018 Pivotal Software, Inc. All rights Reserved. Version 1.0

Objectives

After completing this lesson, you should be able to

- Use External Properties to control Configuration
- Demonstrate the purpose of Profiles
- Use the Spring Expression Language (SpEL)
- Explain How Configuration classes are Manipulated at Runtime

Pivotal

Agenda

- **External Properties**
- Profiles
- Spring Expression Language
- Singleton “Magic”

Pivotal

Setting property values

- Consider this bean definition from the last chapter:

```
@Bean
public DataSource dataSource() {
   BasicDataSource ds = new BasicDataSource();
   ds.setDriverClassName(“org.postgresql.Driver”);
   ds.setUrl(“jdbc:postgresql://localhost/transfer” );
   ds.setUser(“transfer-app”);
   ds.setPassword(“secret45” );
   return ds;
}
```

- Bad practice to hard-code these parameter Strings
  - Better practice is to “externalize” these to a properties file

Spring's Environment Abstraction – 1

- `Environment` object used to load properties from runtime environment

- Properties derived from various sources, in this order:
  - JVM System Properties - `System.getProperty()`
  - Servlet Context Parameters
  - JNDI
  - System Environment Variables - `System.getenv()`
  - Java Properties Files

Spring's Environment Abstraction – 2

```java
@Configuration
public class DbConfig {

   @Bean public DataSource dataSource(Environment env) {
     BasicDataSource ds = new BasicDataSource();
     ds.setDriverClassName(  env.getProperty( "db.driver" ));
     ds.setUrl( env.getProperty( "db.url" ));
     ds.setUser( env.getProperty( "db.user" ));
     ds.setPassword( env.getProperty( "db.password" ));
     return ds;
   }
}
```

Inject `Environment` bean like any other Spring Bean

Fetch property values from environment

```properties
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql:localhost/transfer
db.user=transfer-app
db.password=secret45
```

app.properties

Pivotal

Property Sources

- Environment bean obtains values from "property sources"
  - *Environment variables* and *Java System Properties* always populated automatically
  - **@PropertySource** contributes *additional* properties
  - Available resource prefixes: **`classpath: file: http:`**

```
@Configuration
@PropertySource ( "classpath:/com/organization/config/app.properties" )
@PropertySource ( "file:config/local.properties" )
public class ApplicationConfig {
   ...
}
```

Adds properties from these files *in addition* to environment variables and system properties

Pivotal

Accessing Properties using @Value

```
@Configuration
public class DbConfig {
  @Bean
  public DataSource dataSource(
        @Value("${db.driver}") String driver,
        @Value("${db.url}") String url,
        @Value("${db.user}") String user,
        @Value("${db.password}") String pwd) {
    BasicDataSource ds = new BasicDataSource();
    ds.setDriverClassName(  driver);
    ds.setUrl( url);
    ds.setUser( user);
    ds.setPassword( pwd ));
    return ds;
  }
}
```

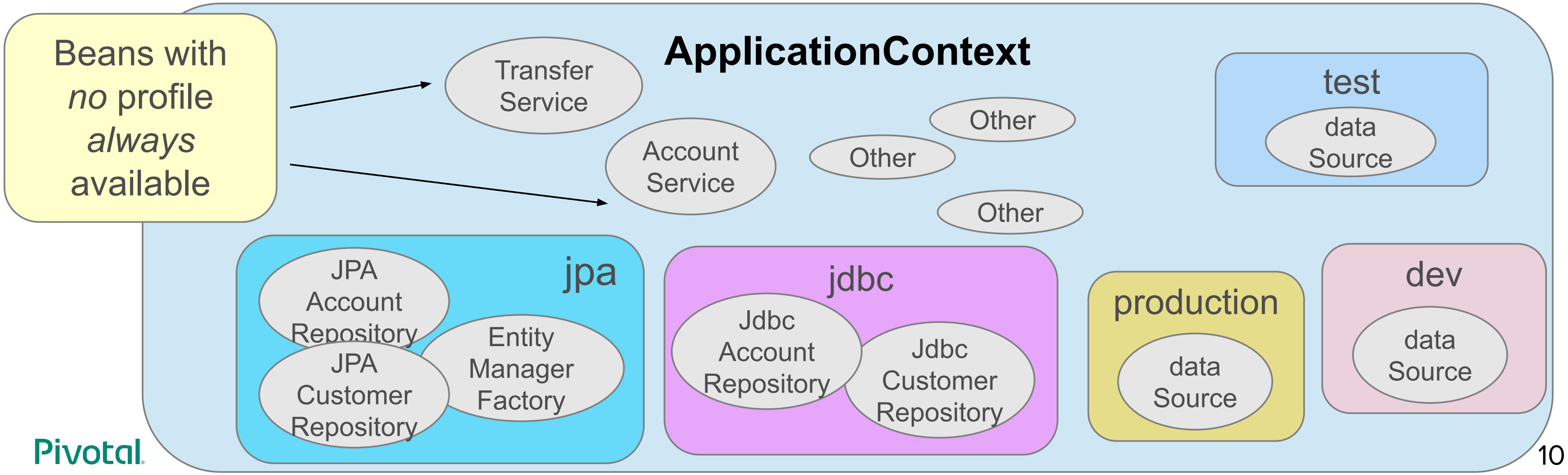
Convenient alternative to explicitly using Environment bean

Agenda

- External Properties
- **Profiles**
- Spring Expression Language
- Singleton “Magic”

Profiles - Beans can be grouped into Profiles

- Profiles can represent environment: dev, test, production
- Or implementation: "jdbc", "jpa"
- Or deployment platform: "on-premise", "cloud"
- Beans included / excluded based on profile membership



Pivotal

Defining Profiles – 1

- Using @Profile annotation on configuration class
  - Everything in Configuration belong to the profile

```
@Configuration
@Profile("embedded")
public class DevConfig {

    @Bean
    public DataSource dataSource() {
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        return builder.setName("testdb")
                      .setType(EmbeddedDatabaseType.HSQL)
                      .addScript("classpath:/testdb/schema.db")
                      .addScript("classpath:/testdb/test-data.db").build();
    }
    …
```

Nothing in this configuration will be used *unless* "embedded" is activated

H2, Derby are also supported

Pivotal

Defining Profiles - 2

- Using @Profile annotation on @Bean methods

```
@Configuration
public class DataSourceConfig {

  @Bean(name="dataSource")
  @Profile("embedded")
  public DataSource dataSourceForDev() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setName("testdb") ...
  }

  @Bean(name="dataSource")
  @Profile("!embedded")
  public DataSource dataSourceForProd() {
    BasicDataSource dataSource = new BasicDataSource();
    ...
    return dataSource;
  }
```

Explicit bean-name overrides method name

Both profiles define *same* bean id, so only *one* profile should be activated at a time.

Pivotal

Defining Profiles - 3

- Beans when a profile is *not* active

```
@Configuration
@Profile("cloud")
public class DevConfig {
    …
}
```

If *cloud* is **active** profile

```
@Configuration
@Profile("!cloud")
public class ProdConfig {
    …
}
```

If *cloud* is **inactive** profile

Not cloud – use exclamation !

Ways to Activate Profiles

- Profiles must be activated at run-time
  - System property via command-line

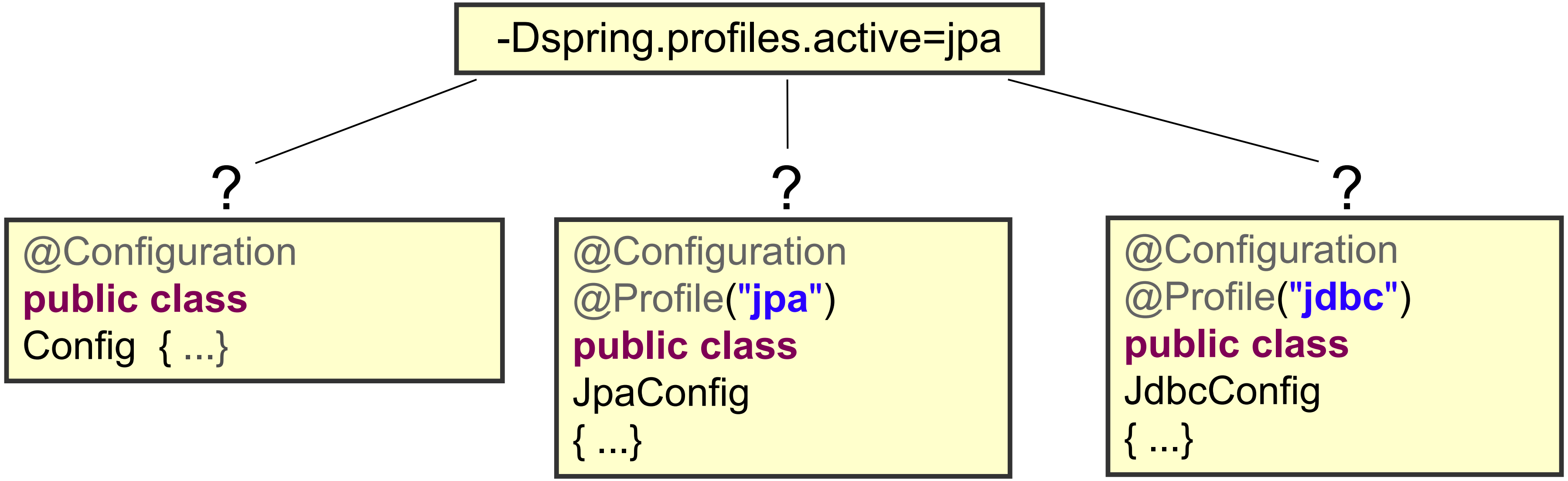
```
-Dspring.profiles.active=embedded,jpa
```

  - System property programmatically

```
System.setProperty("spring.profiles.active", "embedded,jpa");
SpringApplication.run(AppConfig.class);
```

  - Integration Test *only*: **`@ActiveProfiles`** (later section)

Quiz: Which of the Following is/are Selected?



```
-Dspring.profiles.active=jpa
```

?

```
@Configuration
public class
Config  { ...}
```

?

```
@Configuration
@Profile("jpa")
public class
JpaConfig
{ ...}
```

?

```
@Configuration
@Profile("jdbc")
public class
JdbcConfig
{ ...}
```

Pivotal

Property Source selection

- `@Profile` can control which `@PropertySources` are included in the Environment

```
@Configuration
@Profile("local")
@PropertySource ( "local.properties" )
class DevConfig { … }
```

```
@Configuration
@Profile("cloud")
@PropertySource ( "cloud.properties" )
class ProdConfig { … }
```

```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://localhost/transfer
db.user=transfer-app
db.password=secret45
```

local.properties

```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://prod/transfer
db.user=transfer-app
db.password=secret99
```

cloud.properties

Agenda

- External Properties
- Profiles
- **Spring Expression Language**
- Singleton "Magic"

Pivotal

Spring Expression Language

- SpEL for short
  - Inspired by the Expression Language used in Spring WebFlow
  - Based on Unified Expression Language used by JSP and JSF
- Pluggable/extendable by other Spring-based frameworks

This is just a brief introduction, for full details see:
http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html

SpEL examples – Using @Value

```
@Configuration
class TaxConfig
{
  @Value("#{ systemProperties['user.region'] }") String region;

  @Bean
  public TaxCalculator taxCalculator1() {
    return new TaxCalculator( region );
  }

  @Bean
  public TaxCalculator taxCalculator2
         (@Value("#{ systemProperties['user.region'] }") String region, …)  {
    return  new TaxCalculator( region );
  }
  …
```

Option 1: Set an attribute then use it

Option 2: Pass as a bean method argument

Pivotal

SpEL – Accessing Spring Beans

```
class StrategyBean {
   private KeyGenerator gen = new KeyGenerator.getInstance("Blowfish");
   public KeyGenerator getKeyGenerator()  { return gen; }
}
```

```
@Configuration
class StrategyConfig
{
     @Bean public StrategyBean strategyBean() {
          return new StrategyBean();
     }
}
```

```
@Configuration
@Import(StrategyConfig.class)
class AnotherConfig
{
     @Value("#{strategyBean.keyGenerator}") KeyGenerator kgen;
     …
}
```

Pivotal

Accessing Properties

- Can access properties via the *environment*
  - These are equivalent

```
@Value("${daily.limit}")
int maxTransfersPerDay;
```

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

- Properties are Strings
  - May need to cast in expressions

```
@Value("#{new Integer(environment['daily.limit']) * 2}")

@Value("#{new java.net.URL(environment['home.page']).host}")
```

Cannot do this with properties

Fallback Values

:-) Elvis lives!

- Providing a fall-back value
  - If `daily.limit` undefined, use colon **:**

```
@Autowired
public TransferServiceImpl(@Value("${daily.limit : 100000}") int max) {
    this.maxTransfersPerDay = max;
}
```

Equivalent operators

  - For SpEL, use the "Elvis" operator **?:**

```
@Autowired
public setLimit(@Value("#{environment['daily.limit'] ?: 100000}") int max) {
    this.maxTransfersPerDay = max;
}
```

`x ?: y` is short for `x != null ? x : y`

Pivotal

SpEL

- EL Attributes can be:
  - Spring beans (like *strategyBean*)
  - Implicit references
    - Spring's *environment, systemProperties, systemEnvironment* available by default
    - Others depending on context
- SpEL allows to create custom functions and references
  - Widely used in Spring projects
    - Spring Security, Spring WebFlow
    - Spring Batch, Spring Integration
  - Each may add *their own* implicit references

Agenda

- External Properties
- Profiles
- Spring Expression Language
- **Singleton "Magic"**

Pivotal

Quiz

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

@Bean
public TransferService transferService1() {
    TransferServiceImpl service = new TransferServiceImpl();
     service.setAccountRepository(accountRepository());
     return service;
}

@Bean
public TransferService transferService2() {
    return new TransferServiceImpl( new JdbcAccountRepository()  );
}
```

Which is the best implementation?

1. Method call? ✓

2. New instance?

Prefer call to *dedicated* method. Let's discuss why ...

Pivotal

Working with Singletons

Recall: *Singleton* is default scope

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

@Bean
public TransferService transferService()  {
    TransferServiceImpl service = new TransferServiceImpl();
     service.setAccountRepository(accountRepository());
     return service;
}

@Bean
public AccountService accountService()  {
    return new AccountServiceImpl( accountRepository() );
}
```

Singleton??

Method called *twice* more

HOW IS IT POSSIBLE?

Singletons Require "Magic"

- At startup time, a *subclass* is created
  - Subclass performs *scope-control*
    - Only calls *super* on *first* invocation of singleton bean method
    - Singleton instance is cached by the *ApplicationContext*

```
@Configuration
public class AppConfig {
      @Bean public AccountRepository accountRepository() { ... }
      @Bean public TransferService transferService() { ... }
}
```

inherits from

```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {
  public AccountRepository accountRepository() {   // ... }
  public TransferService transferService() {   // ... }
}
```

Inheritance-based Subclasses

- Child class is the entry point

```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {

  public AccountRepository accountRepository() {
    // if bean is in the applicationContext, then return bean
    // else call super.accountRepository(), store bean in context, return bean
  }

  public TransferService transferService() {
    // if bean is in the applicationContext, then return bean
    // else call super.transferService(), store bean in context, return bean
  }
```

Java Configuration uses *cglib* for inheritance-based subclasses

Pivotal

Summary

- Property values are easily externalized using Spring's Environment abstraction
- Profiles are used to group sets of beans
- Spring Expression Language
- Spring proxies your **`@Configuration`** classes to allow for scope control