3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Java Based Configuration Concept

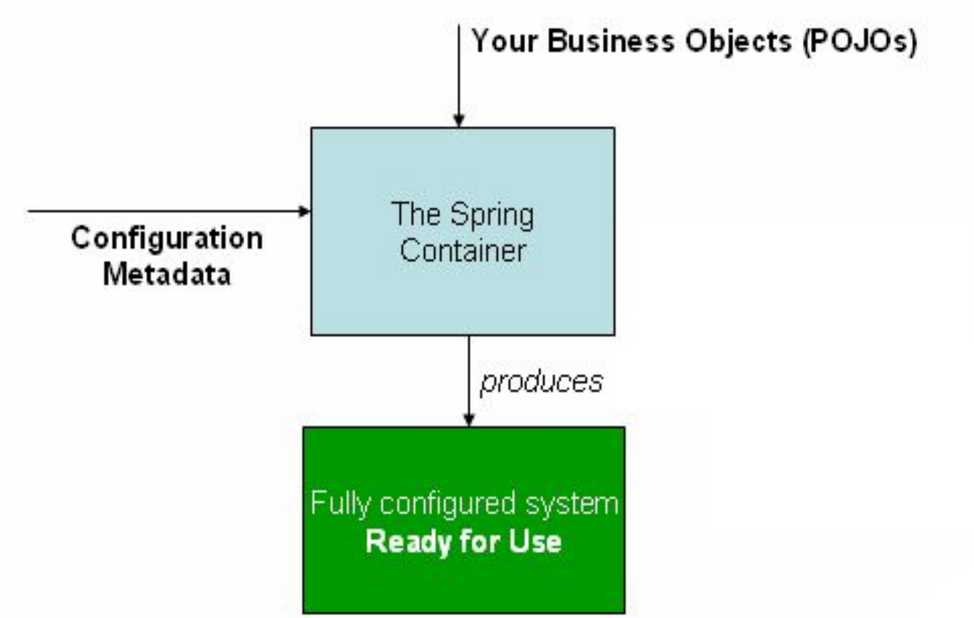


Figure 3.1: *The following diagram shows a high-level view of how Spring works. Application classes are combined with configuration metadata so that, after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.*

- How can we initialize POJOs?
- How can we configure the dependencies they have?

Java Based Configuration Concept

No new operator
No service located
AccountRepository dependency

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

Dependency: Needed to perform money transfers between accounts

```
public class JdbcAccountRepository implements AccountRepository {
    public JdbcAccountRepository(DataSource ds) {
        this.dataSource = ds;
    }
    ...
}
```

Dependency: Needed to access account data in the database

No new operator
No service located
DataSource dependency

What we need?
Decoupling these components
Configuring our own components

Configuration Instructions with Dependencies

@Configuration: Tells the framework that we have definitions inside this class.
@Bean: Tells spring that defines a bean in ApplicationContext.

Method Invocation

```
@Configuration
public class ApplicationConfig {
  @Bean public TransferService transferService() {
    return new TransferServiceImpl( accountRepository() );
  }
  @Bean public AccountRepository accountRepository() {
    return new JdbcAccountRepository( dataSource() );
  }
  @Bean public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUrl("jdbc:postgresql://localhost/transfer" );
    dataSource.setUsername("transfer-app");
    dataSource.setPassword("secret45");
    return dataSource;
  }
}
```

Represents a dependency

Pass as Reference

```
@Configuration
public class ApplicationConfig {
  @Bean public TransferService transferService(AccountRepository repository) {
    return new TransferServiceImpl( repository );
  }
  @Bean public AccountRepository accountRepository(DataSource dataSource) {
    return new JdbcAccountRepository( dataSource );
  }
  @Bean public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    ...
    return dataSource;
  }
}
```

Represents a dependency

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Application Context

```
// Create application context from the configuration
ApplicationContext context =
    SpringApplication.run( ApplicationConfig.class );

// Look up a bean from the application context
TransferService service =
        context.getBean("transferService", TransferService.class);

// Use the bean
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```

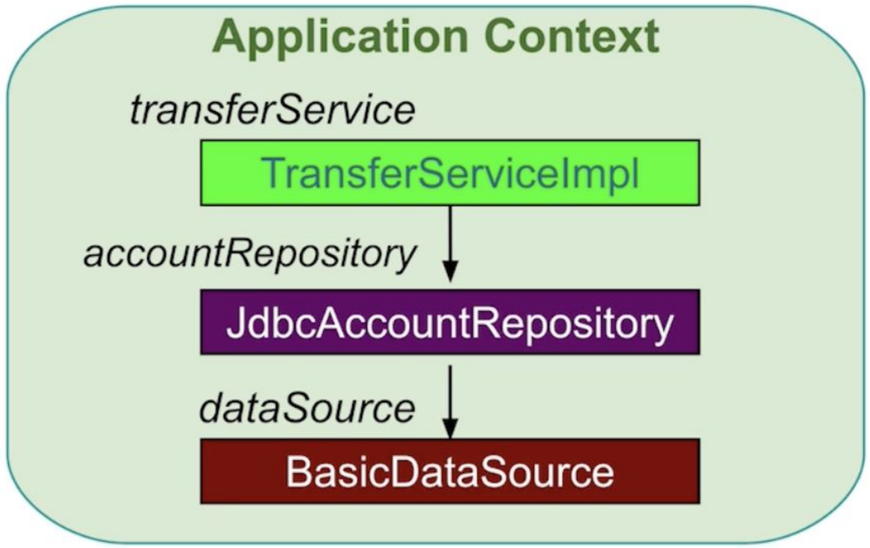
What configuration to use to define beans

Bean ID
Based in method name

use SpringApplication.run()

Inside the Spring ApplicationContext

```
// Create application context from the configuration
ApplicationContext context = SpringApplication.run( ApplicationConfig.class )
```



Application Context: Configuring with XML

```
// Create application context with ClassPathXmlApplicationContext
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans
           .xsd">

    <bean id="transferService" class="com.trendyol.bootcamp.service
        .TransferService"/>
</beans>
```

use ClassPathXmlApplicationContext with xml file

Accessing a Bean From Application Context

Multiple options

```
ApplicationContext context = SpringApplication.run(...);

// Use bean id, a cast is needed
TransferService ts1 = (TransferService)  context.getBean("transferService");

// Use typed method to avoid casting
TransferService ts2 =  context.getBean("transferService", TransferService.class);

// No need for bean id if type is unique - recommended (use type whenever possible)
TransferService ts3 =  context.getBean(TransferService.class );
```

cast because it is not type safe

a safe way with name and type

an ambiguity situation

Summary 

Spring manages your application objects

- ❑ Creating them in the correct dependency order
- ❑ Ensuring they are fully initialized before use

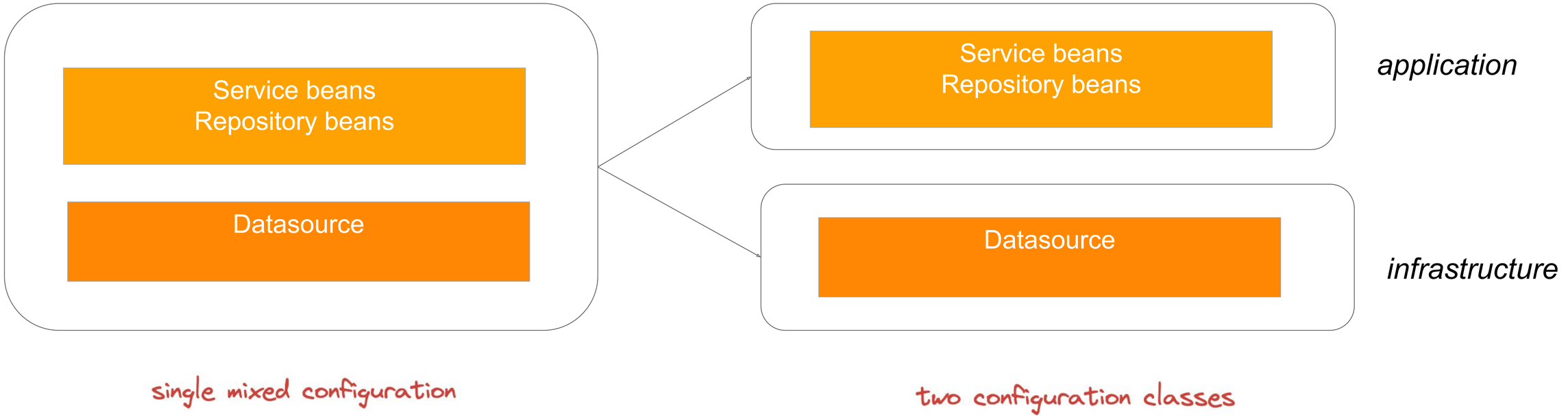
Ice Breaker 🥶

If you could travel anywhere in the world, where would you go?

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Creating an Application Context From Multiple Files



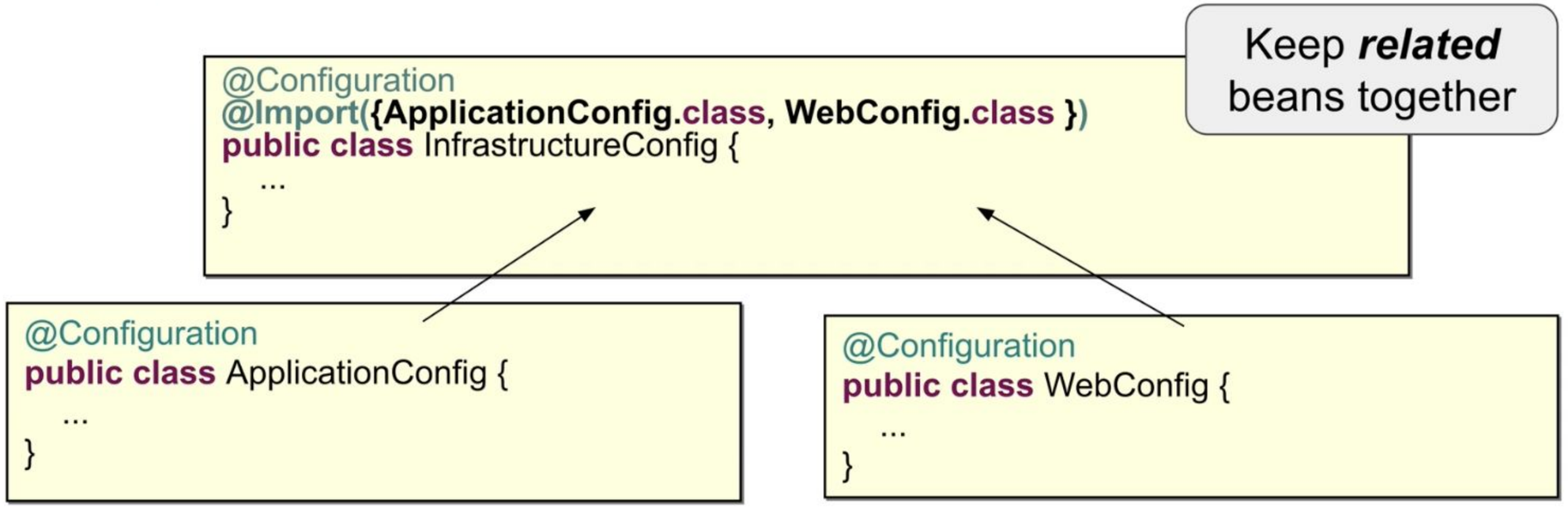
Separation of Concerns principle

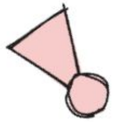
- ❏ Keep related beans in the same **@Configuration** file

Best Practice: separate “application” and “infrastructure”

- ❏ In real world, infrastructure often changes between environments

Handling Multiple Configurations





Files combined with @Import

Defines a single Application Context

Beans sourced from multiple files

Mixed Configuration

```
@Configuration
public class ApplicationConfig {

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

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

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

application beans

infrastructure bean

Coupled to a local Postgres environment

```
@Configuration
public class ApplicationConfig {
   @Bean public TransferService transferService(AccountRepository repository) {
      return new TransferServiceImpl ( repository );
   }

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

```
@Configuration
@Import(ApplicationConfig.class )
public class TestInfrastructureConfig {
   @Bean public DataSource dataSource() {
      ...
   }
}
```

```
ApplicationContext ctx = SpringApplication.run( TestInfrastructureConfig.class )
```

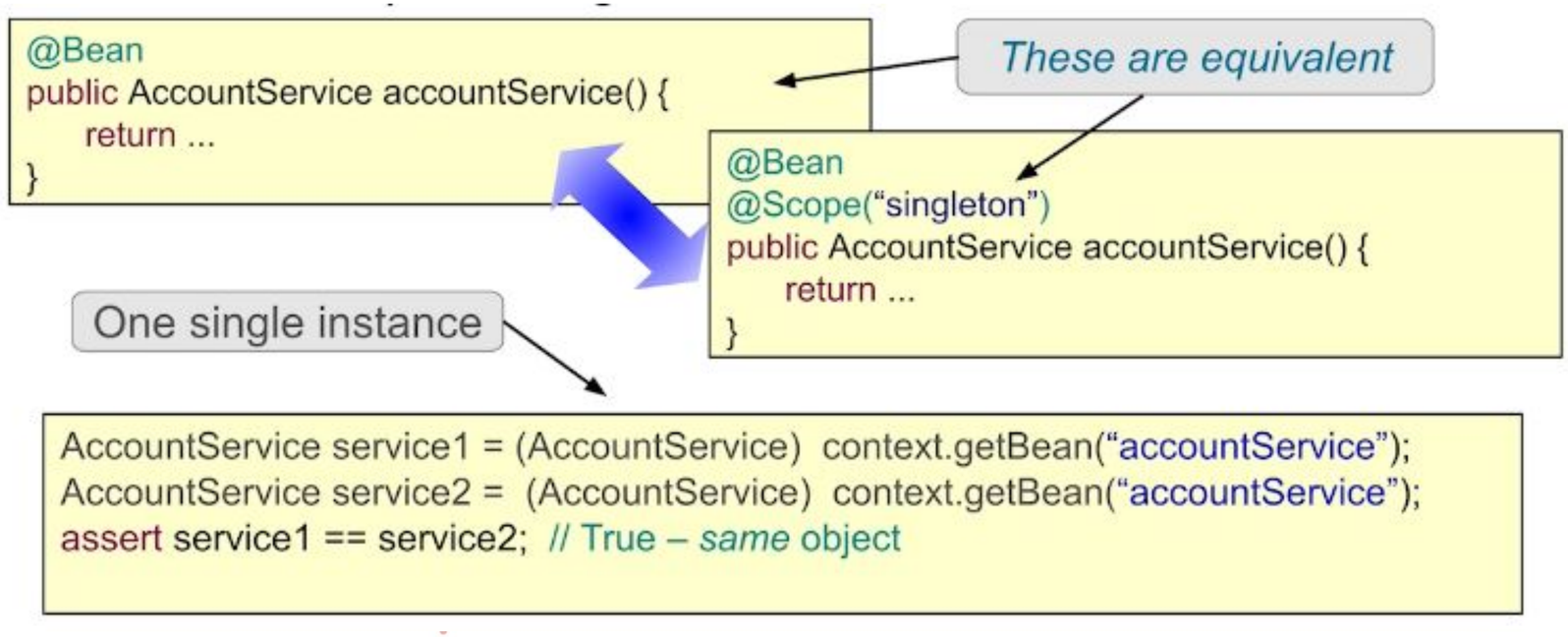
3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Bean Scope: *singleton*

The **scope** of a bean defines the life cycle and visibility of that bean in the contexts we use it.

- Default scope is *singleton.*
- **Singleton**: same instance used every time bean is referenced



Multiple requests problem:

Multiple threads accessing singleton beans at the same time

Bean Scope: *prototype*

- **Prototype**: new instance created every time bean is referenced

```
@Bean
@Scope("prototype")
public Action deviceAction() {
    return …
}
```

@Scope(scopeName="prototype")

```
Action action1 = (Action)  context.getBean("deviceAction");
Action action2 =  (Action)  context.getBean("deviceAction");
assert action1 != action2;  // True – different objects
```

TWO instances

Common Spring Scopes

Singleton

A single instance is used

Prototype

A new instance is created each time the bean is referenced

Session

A new instance is created per user session

Request

A new instance is created once per request

web environment only

Summary 

- In Java configuration there are two important annotation: **@Configuration** and **@Bean**
- You should define beans in configuration files.
- You can use single configuration file or multiple configuration file.
- You can use multiple configuration files with **@Import** annotation**.**

- Dependency injection aim is to decrease coupling
- Components don’t need to find their dependencies
- Container will inject those dependencies for them

- Application context manages the lifecycle of components
- The default scope is **Singleton**. It creates one instance and uses it
- In **Prototype** scope, a new instance is created every time

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Setting Property Values

Hard-coding these properties is a Bad practice

- ❏ Better practice is to “externalize” these properties.
- ❏ One way to “externalize” them is by using property files.

Environment bean represents loaded properties from runtime environment.
Properties derived from various sources:

- ❏ JVM System properties → **System.getProperty()**
- ❏ System Environment Variables → **System.getenv()**
- ❏ Java Properties Files


```
@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;
}
```

Spring Environment Abstraction

Inject **Environment**

- ❏ package: *org.springframework.core.env*
- ❏ Call *getProperty()* method
- ❏ If you want to change any property value, change it from application.yml file

```
@Configuration
public class DbConfig {

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

}
```

Inject Environment bean like any other Spring bean

```
spring:
  datasource:
    driver: org.postgresql.Driver
    url: jdbc:postgresql://localhost/transfer
    username: transfer-app
    password: secret45
```

application.yml file

Property Source

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/trendyol/bootcamp/config/app.properties”)
@PropertySource (“file:config/local.properties”)
public class ApplicationConfig {

  ...
}
```

Add properties to these files in addition to environment variables and system properties

Accessing Properties Using @Value

use **@Value** annotation

- ❏ *package*: org.springframework.beans.factory.annotation
- ❏ No need to reference Environment
- ❏ 💥The *most* used way in Trendyol for accessing properties

```
@Configuration
   public class DbConfig {

      @Value("${spring.datasource.driver}")
       private String driver;

      @Value("${spring.datasource.url}")
       private String url;

      @Value("${spring.datasource.username}")
       private String username;

      @Value("${spring.datasource.password}")
       private String password;

      @Bean
      public DataSource dataSource() {
         BasicDataSource ds = new BasicDataSource();
         ds.setDriverClassName(driver);
         ds.setUrl(url);
         ds.setUser(username);
         ds.setPassword(password);
         return ds;
      }
}
```

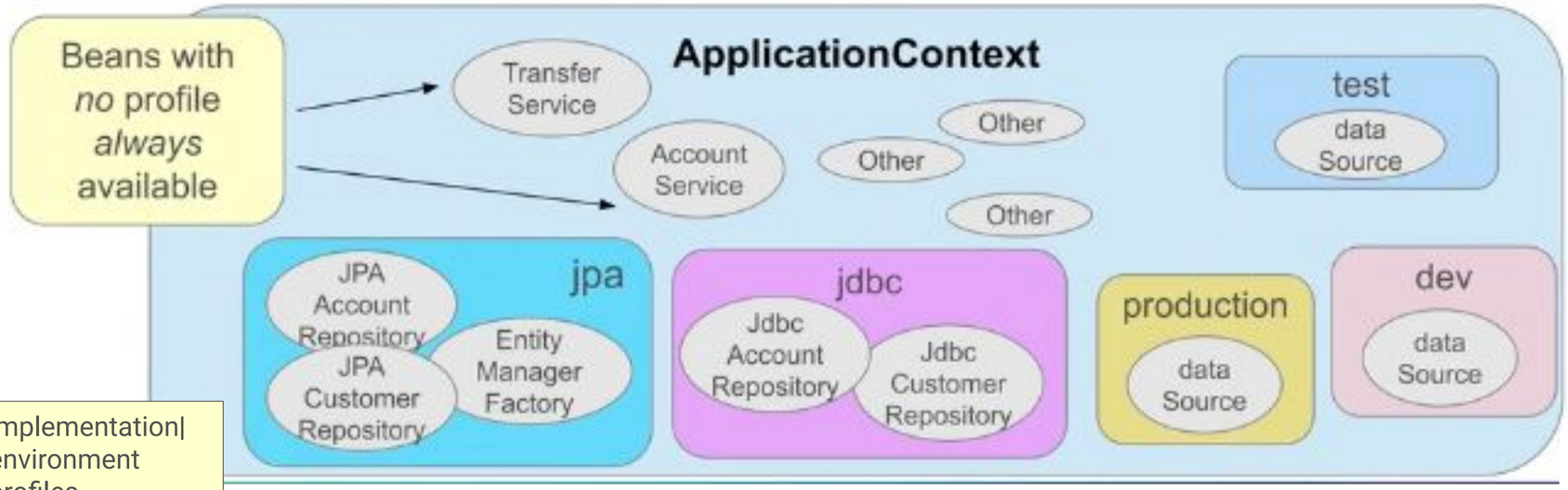
3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Coding Section: *Java Based Configuration*

Spring Profiles

Profiles can represent

- ❏ Environment: *dev, test, production*
- ❏ Implementation: *jdbc, jpa*
- ❏ Deployment platform: *on-premise, cloud*
- ❏ Beans included / excluded based on profile membership



implementation| environment profiles

Defining Profiles

Using **@Profile** annotation on configuration class

- org.springframework.context.annotation
- Everything in Configuration belong to the profile
- If Profile is not activated, bean is not initialized

```
@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" profile is chosen as one of the active profiles

H2, Derby are also supported

Defining Profiles

You can use **@Profile** annotation on **@Bean** methods
You can use exclamation **!** with **@Profile**

```
@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 !

```
@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 name, so only *one* profile should be activated at a time.

Ways to Activate Profiles

Profiles must be activated at run-time

- System property via command-line

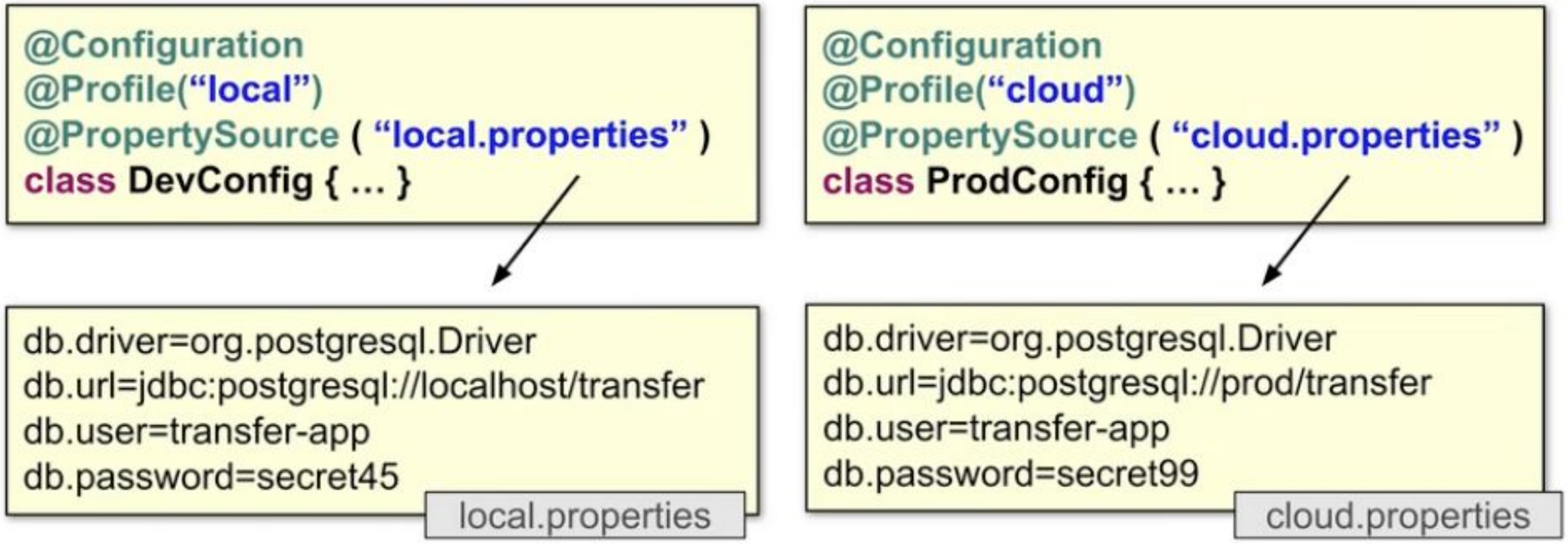
```
-Dspring.profiles.active=embedded,jpa
```

- System property programmatically

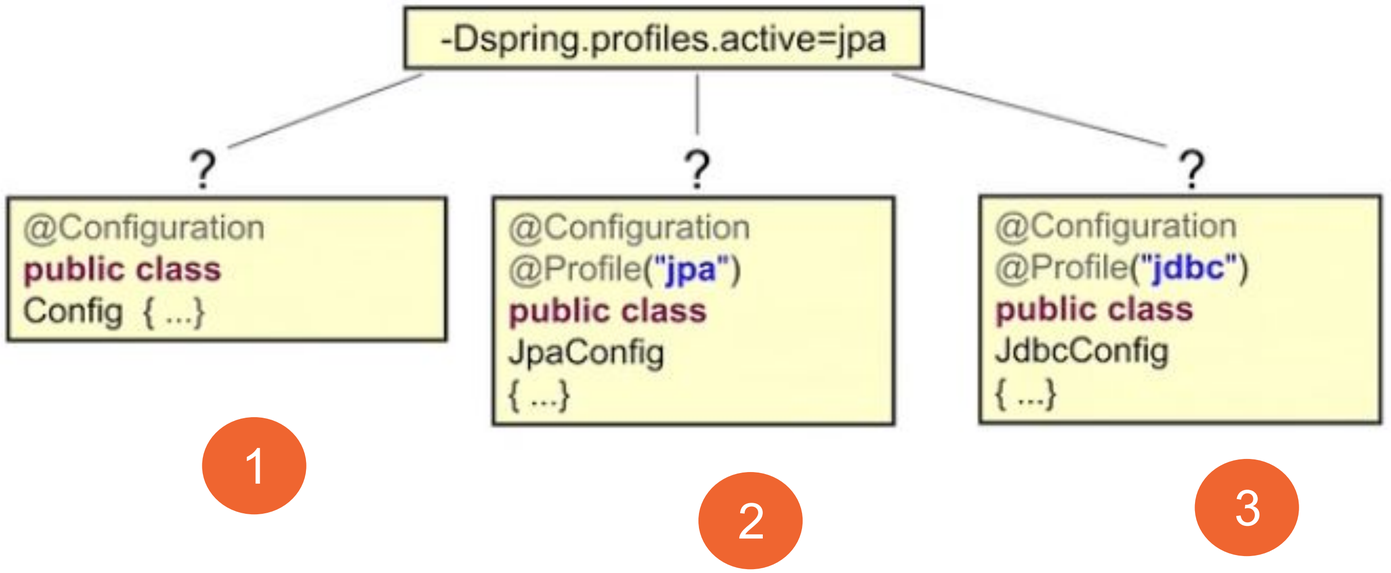
```
System.setProperty("spring.profiles.active", "embedded,jpa");
SpringApplication.run(AppConfig.class);
```

Property Source Selection

@Profile can control which **@PropertySource** are included in the Environment



Question



trendyol learning

3. Java Based Configuration

1. Java Based Configuration Concept
2. Application Context
3. Handling Multiple Configuration
4. Bean Scopes
5. Property Values
6. Spring Profiles
7. Lab Section: *Java Based Configuration*

Lab Section

What we will learn:

1. How to create a Spring ApplicationContext and get a bean from it
2. Spring Java configuration syntax

Todos:

1. Please, Switch branch to `feature/java-based-configuration`
2. There are **12 TODOs** in the project files. Look at these TODOs
3. Lets try to do each TODO together