Java Configuration

Dependency Injection using Spring

1.18.5

Objectives

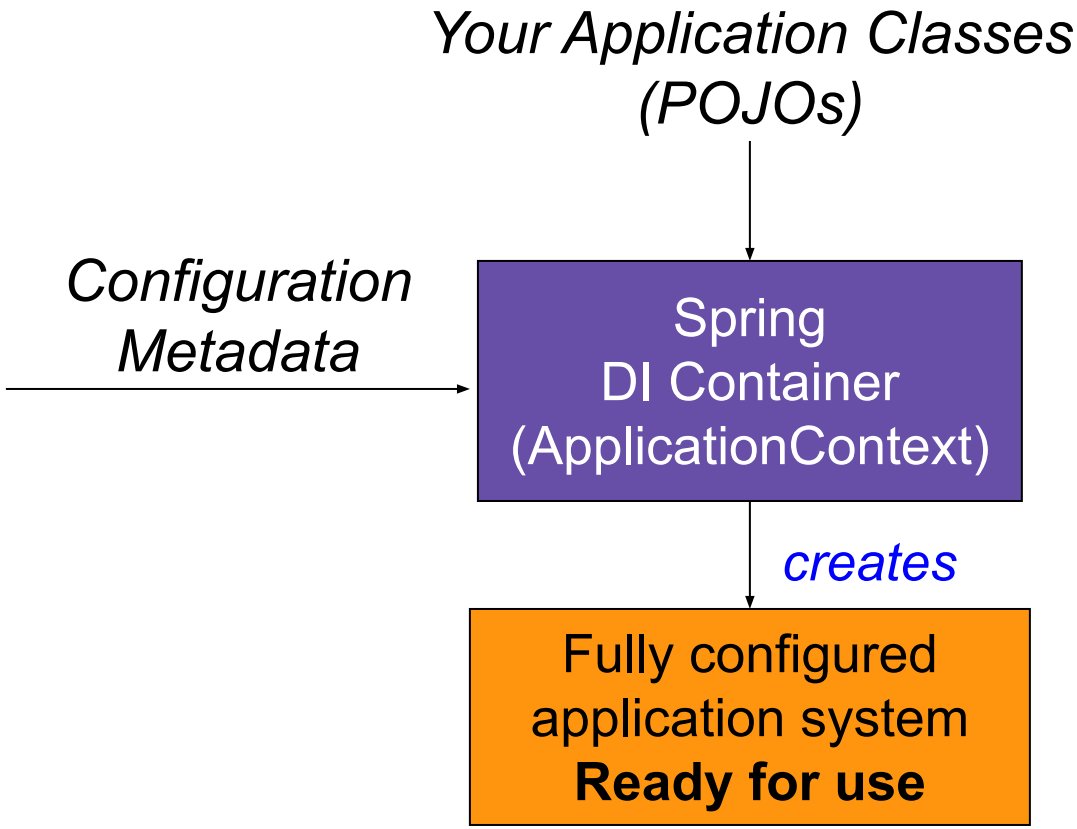
After completing this lesson, you should be able to do the following

- Define Spring Beans using Java code
- Access Beans in the Application Context
- Handle multiple Configuration files
- Handle Dependencies between Beans
- Explain and define Bean Scopes

Agenda

- **Spring quick start**
- Creating an application context
- Multiple Configuration Files
- Bean scope

How Spring DI Container Works



Your Application Classes as POJO's with Dependencies

```
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

You do not have to use *interfaces* in your classes, but it is a good Java practice as they encourage loose-coupling.

Configuration Instructions With Dependencies

```
@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

Creating and Using the Application

```
// 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

Note that Spring will create *four* beans: **`ApplicationConfig`** is *also* a Spring Bean - it is used to create the others.

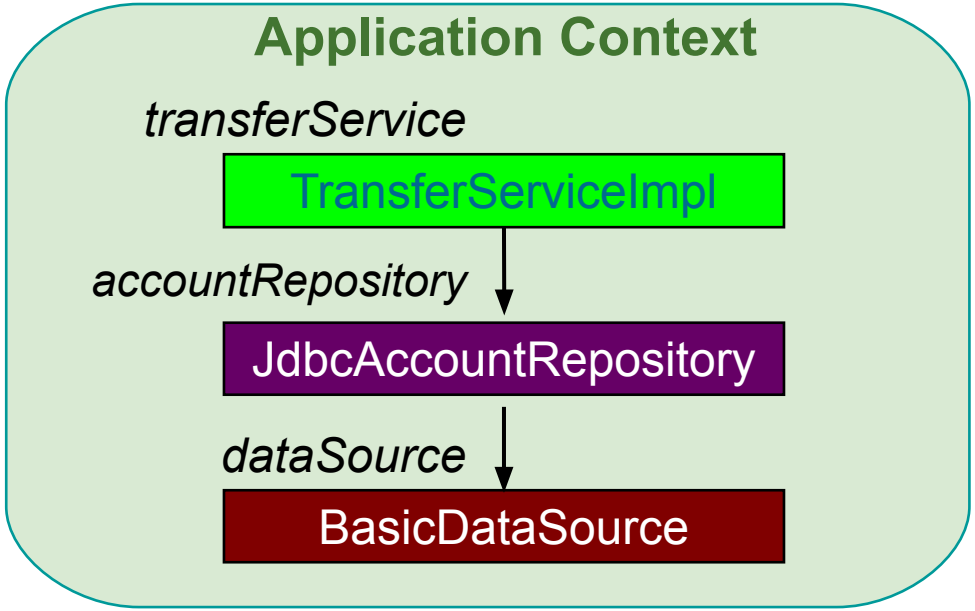
Accessing a Bean Programmatically

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 );
```

Inside the Spring Application Context

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



Quick Start Summary

- Spring separates application configuration from application objects (beans)
- Spring manages your application objects
  - Creating them in the correct dependency order
  - Ensuring they are fully initialized before use
- Each bean is given a unique id / name

Agenda

- Spring quick start
- **Creating an application context**
- Multiple Configuration Files
- Bean scope

Creating a Spring Application Context

- Spring application context represents Spring DI container
  - Spring beans are managed through the application context
- Spring application context can be created in any environment, including
  - Standalone application
  - Web application
  - JUnit test

Application Context Example

Creating Application Context in a System Test

```
public class TransferServiceTests {
    private TransferService service;

    @BeforeEach public void setUp() {
        // Create application context from the configuration
        ApplicationContext context = SpringApplication.run( ApplicationConfig.class )

        // Look up a service
        service = context.getBean(TransferService.class);
    }

    @Test public void moneyTransfer() {
        Confirmation receipt = service.transfer(new MonetaryAmount("300.00"), "1", "2"));

        Assert.assertEquals("500.00", receipt.getNewBalance());
    }
```

Bootstraps the system to test

Tests the system

Using JUnit 5 – JUnit 4 or TestNG also supported

Agenda

- Spring quick start
- Creating an application context
- **Multiple Configuration Files**
- Bean scope

Creating an Application Context from Multiple Configurations

- Your **`@Configuration`** class can get too big
  - Instead use *multiple* config. files combined with **`@Import`**
  - Defines a <u>single</u> Application Context
    - Beans sourced from multiple files

```
@Configuration
@Import({ApplicationConfig.class, WebConfig.class })
public class InfrastructureConfig {
    ...
}
```

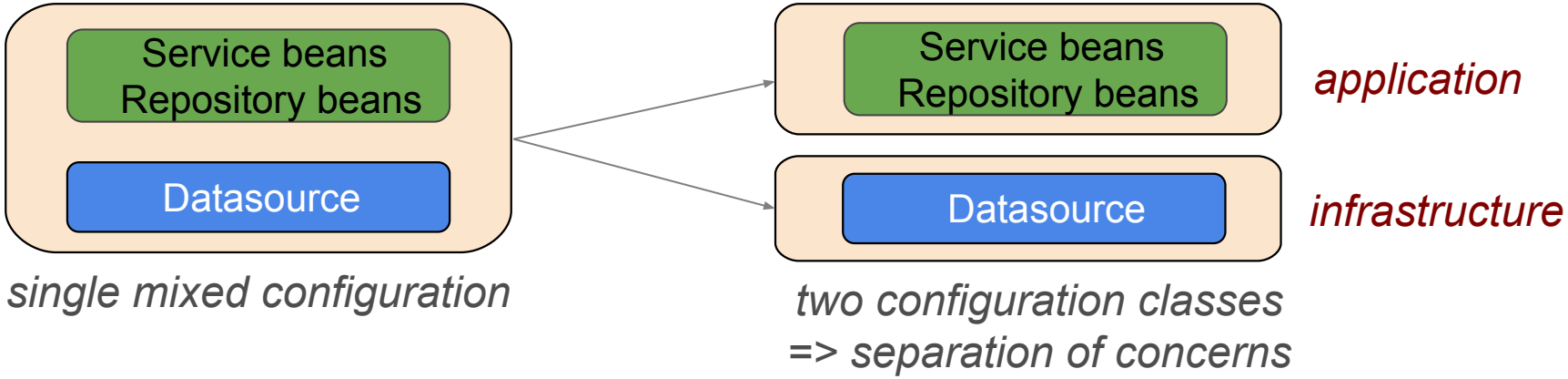
Keep ***related*** beans together

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

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

Creating an Application Context from Multiple Files

- *Separation of Concerns* principle
  - Keep related beans in the same @Configuration
- *Best Practice:* separate “application” & “infrastructure”
  - Infrastructure often changes between environments



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

Coupled to a local Postgres environment

infrastructure bean

Java Configuration with Dependency Injection

- Use @Autowired to inject a bean defined elsewhere

```
@Configuration
public class ApplicationConfig {
  private final DataSource dataSource;

  @Autowired
  public ApplicationConfig(DataSource ds) {
    this.dataSource = ds;
  }

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

```
@Configuration
@Import(ApplicationConfig.class )
public class InfrastructureConfig {
  @Bean
  public DataSource dataSource() {
    DataSource ds = new BasicDataSource();
    ...
    return ds;
  }
}
```

Agenda

- Spring quick start
- Creating an application context
- Multiple Configuration Files
- **Bean scope**

Bean Scope: Default

service1== service2

- Default scope is *singleton*

```
@Bean
public AccountService accountService() {
    return ...
}
```

These are equivalent

```
@Bean
@Scope("singleton")
public AccountService accountService() {
    return ...
}
```

One single instance

```
AccountService service1 = (AccountService)  context.getBean("accountService");
AccountService service2 =  (AccountService)  context.getBean("accountService");
assert service1 == service2;  // True – same object
```

Common Spring Scopes

- The most commonly used scopes are:

singleton	A single instance is used
prototype	A new instance is created each time the bean is referenced
session	A new instance is created once per user session - web environment only
request	A new instance is created once per request – web environment only

Dependency Injection Summary

- Your object is handed with what it needs to work
  - Frees it from the burden of resolving its dependencies
  - Simplifies your code, improves code reusability
- Promotes programming to interfaces
  - Conceals implementation details of dependencies
- Improves testability
  - Dependencies easily stubbed out for unit testing
- Allows for centralized control over object lifecycle
  - Opens the door for new possibilities