Dependency Injection In Spring

- Reduces coupling between classes
- Classes do not obtain their dependencies (collaborators), Spring container provides proper dependency based on configuration provided
- Enables easy switching of dependencies based en current needs - local development vs production environment, easy mocking for unit tests, in memory database vs production one, etc.
- Promotes programming to interfaces
- Spring-managed objects are called "beans"
- Beans are written as POJOs, no need to inherit Spring base classes or implement spring interfaces
- Classes can be completely independent of Spring framework (although for convenience Spring annotations are often used)
- Lifecycle of beans is managed centrally by Spring container (eg. enforces singleton state of bean/ one instance pers session. one instance per HTTP request/...)
- Spring configuration (including DI) is possible either via XML or Java configuration files (newer, more popular approach)

Spring Application Context

- Spring beans are managed by Application Context (container)
- The application context is initialized with one or more configuration files (java or XML), which contains setting for the framework, DI, persistence, transactions,...
- Application context can be also instantiated in unit tests
- Ensures beans are always created in right order based on their dependencies
- Ensures all beans are fully initialized before the first use
- Each bean has its unique identifier, by which it can be later referenced (should not contain specific implementation details, based on circumstances different implementations can be provided)
- No need for full fledged java EE application server

Creating app context

```
    //From single configuration class
    ApplicationContext context =
SpringApplication.run(ApplicationConfiguration.class);

    //With string args
    ApplicationContext context =
SpringApplication.run(ApplicationConfiguration.class, args);

    //With additional configuration
    SpringApplication app = new
SpringApplication(ApplicationConfiguration.class);
    // ... customize app settings here
    context = app.run(args);
```

[A] Additional ways to create app context

- new AnnotationConfigApplicationContext(AppConfig.class);
- new ClassPathXmlApplicationContext("com/example/app-config.xml");
- new FileSystemXmlApplicationContext("C:/Users/vojtech/app-config.xml");
- new FileSystemXmlApplicationContext("./app-config.xml");

Obtaining beans from Application Context

```
//Obtain bean by type
applicationContext.getBean(MyBean.class);
```

Bean Scopes

- Spring manages lifecycle of beans, each bean has its scope
- Default scope is singleton - one instance per application context
- If none of the Spring scopes is appropriate, custom scopes can be defined
- Scope can be defined by @Scope (eg. @Scope(BeanDefinition.SCOPE_SINGLETON)) annotation on the class-level of bean class

Available Scopes

- Singleton - One instance per application context, default if bean scope is not defined
- Prototype - New instance is created every time bean is requested
- Session - One instance per user session - Web Environment only
- [A]Global-session - One global session shared among all portlets - Only in Web Portlet Environment
- Request - One instance per each HTTP Request - Web Environment only
- Custom - Can define multiple custom scopes with different names and lifecycle
- Additional scopes available for Spring Web Flow applications (not needed for the certification)

Spring Configuration

- can be XML or java based
- Externalized from the bean class → separation of concerns

Externalizing Configuration Properties

- Configuration values (DB connection, external endpoints, ...) should not be hard-coded in the configuration files
- Better to externalize to, eg. to .properties files
- Can then be easily changed without a need to rebuild application
- Can have different values on different environments (eg. different DB instance on each environment)
- Spring provides abstraction on top of many property sources
    - JNDI
    - Java .properties files
    - JVM System properties

    - System environment variables
    - Servlet context params

Obtaining properties using Environment object

- Environment can be injected using @Autowired annotation
- properties are obtained using environment.getProperty("propertyName")

Obtaining properties using @Value annotation

- @Value("${propertyName}") can be used as an alternative to Environment object
- can be used on fields on method parameters

Property Sources

- Spring loads system property sources automatically (JNDI, System variables, ...)
- Additional property sources can be specified in configuration using @PropertySource annotation on @Configuration class
- If custom property sources are used, you need to declare PropertySourcesPlaceholderConfigurer as a static bean

```
@Configuration
@PropertySource("classpath:/com/example/myapp/config/application.properties
")
public class ApplicationConfiguration {

    @Bean
    public static PropertySourcesPlaceholderConfigurer
propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    //Additional config here
}
```

- Can use property placeholders in @PropertySource names (eg. different property based on environment - dev, staging, prod)
- placeholders are resolved against know properties

```
@PropertySource("classpath:/com/example/myapp/config/application-
{$ENV}.properties")
```

Spring Expression language

- Acronym SpEL
- can be used in @Value annotation values
- enclosed in #{}
- ${} is for properties, #{} is for SpEL
- Access property of bean #{beanName.property}
- Can reference systemProperties and systemEnvironment
- Used in other spring projects such as Security, Integration, Batch, WebFlow,...

Spring profiles

- Beans can have associated profile (eg. "local", "staging", "mock")
- Using @Profile("profileName") annotation - either on bean level or on @Configuration class level - then it applies for all the beans inside
- Beans with no profile are always active
- Beans with a specific profile are loaded only when given profile is active
- Beans can have multiple profiles assigned
- Multiple profiles can be active at the same time
- Active profiles can be set in several different ways
    - @ActiveProfiles in spring integration tests
    - in web.xml deployment descriptor as context param ("spring.profiles.active")
    - By setting "spring.profiles.active" system property
    - Active profiles are comma-separated
- Profiles can be used to load different property files under different profiles

```
@Configuration
@Profile("local")
@PropertySource("classpath:/com/example/myapp/config/application-
local.properties")
public class LocalConfig {
    //@PropertySource is loaded only when "local" profile is active

    //Config here
}
```

Composite configuration

- Preferred way is to divide configuration to multiple files and then import them when used
- Usually based on application layers - web layer, service layer, DAO layer,...
- Enables better configuration organization
- Enables to separate layers, which often change based on environment, etc. - eg. switching services for mocks in unit tests, having an in-memory database, ...
- Can import other config files in java config using @Import annotation

```
@Configuration
@Import({WebConfiguration.class, InfrastructureConfiguration})
public class ApplicationConfiguration {
    //Config here
}
```

- Beans defined in another @Configuration file can be injected using @Autowired annotation on field or setter level, cannot use constructors here
- Another way of referencing beans from another @Configuration is to declare them as method parameters of defined beans in current config file

```
@Configuration
@Import({WebConfiguration.class, InfrastructureConfiguration})
public class ApplicationConfiguration {

    @Bean //Object returned by this method will be spring managed bean
    public MyBean myBean(BeanFromOtherConfig beanFromOtherConfig) {
        //beanFromOtherConfig bean is automatically injected by Spring
        ...
        return myBean;
    }
```

- In case multiple beans of the same type are found in configuration, spring injects the one, which is discovered as the last

Java Configuration

- In Java classes annotated with @Configuration on class level
- Beans can be either specifically declared in @Configuration using @Bean annotation or automatically discovered using component scan

Component Scan

- On @Configuration class, @ComponentScan annotation can be present with packages, which should be scanned
- All classes with @Component annotation on class level in scanned packages are automatically considered spring managed beans
- Applies also for annotations annotated with @Component (@Service, @Controller, @Repository, @Configuration)
- Beans declare their dependencies using @Autowired annotation - automatically injected by spring
  - Field level - even private fields
  - Method Level
  - Constructor level
- Autowired dependencies are required by default and result in exception when no matching bean found
- Can be changed to optional using @Autowired(required=false)
- Dependencies of type `Optional<T>` are automatically considered optional
- Can be also used for scanning jar dependencies
- Recommended to declared as specific scanned packages as possible to speed up scan time

```
@Configuration
@ComponentScan( { "org.foo.myapp", "com.bar.anotherapp" })
public class ApplicationConfiguration {
    //Configuration here
}
```

@Value annotation

- Is used to inject value either from system properties (using ${}) or SpEL (using #{})
- Can be on fields, constructor parameters or setter parameters
- On constructors and setters must be combined with @Autowired on method level, for fields @Value alone is enough
- Can specify default values
  - ${minAmount:100}"
  - #{environment['minAmount'] ?: 100}

Setter vs Constructor vs Field injection

- All three types can be used and combined

- Usage should be consistent
- Constructors are preferred for mandatory and immutable dependencies
- Setters are preferred for optional and changeable dependencies
- Only way to resolve circular dependencies is using setter injection, constructors cannot be used
- Field injection can be used, but mostly discouraged

Autowired conflicts

- If no bean of given type for @Autowired dependency is found and it is not optional, exception is thrown
- If more beans satisfying the dependency is found, NoSuchBeanDefinitionException is thrown as well
- Can use @Qualifier annotation to inject bean by name, the name can be defined inside component annotation - @Component("myName")

```
@Component("myRegularDependency")
public class MyRegularDependency implements MyDependency {...}

@Component("myOtherDependency")
public class MyOtherDependency implements MyDependency {...}

@Component
public class MyService {
    @Autowired
    public MyService(@Qualifier("myOtherDependency") MyDependency
myDependency) {...}
}
```

Autowired resolution sequence

1. Try to inject bean by type, if there is just one
2. Try to inject by @Qualifier if present
3. Try to inject by bean name matching name of the property being set

- Bean 'foo' for 'foo' field in field injection
- Bean 'foo' for 'setFoo' setter in setter injection
- Bean 'foo' for constructor param named 'foo' in constructor injection

When a bean name is not specified, one is auto-generated: De-capitalized non-qualified class name

Explicit bean declaration

- Class is explicitly marked as spring managed bean in @Configuration class (similar concept is in XML config)
- All settings of the bean are present in the @Configuration class in the bean declaration
- Spring config is completely in the @Configuration, bean is just POJO with no spring dependency
- Cleaner separation of concerns
- Only option for third party dependencies, where you cannot change source code

```
@Configuration
public class ApplicationConfiguration {

    /***
    *  - Object returned by this method will be spring managed bean
    *  - Return type is the type of the bean
    *  - Method name is the name of the bean
    */
    @Bean
    public MyBean myBean() {
        MyBean myBean = new MyBean();
        //Configure myBean here
        return myBean;
    }
```

Component scan vs Explicit bean declaration

- Same settings can be achieved either way
- Both approaches can be mixed
  - Can use component scan for your code, @Configuration for third party and legacy code
  - Use component scan only for Spring classes, Explicit configuration for others
- Explicit bean declaration
  - Is more verbose
  - Achieves cleaner separation of concerns
  - Config is centralized in on or several places
  - Configuration can contain complex logic
- Component scan
  - Config is scattered across many classes
  - Cannot be used for third party classes
  - Code and configuration violate single responsibility principle, bad separations of concerns
  - Good for rapid prototyping and frequently changing code

@PostConstruct, @PreDestroy

- Can be used on @Component's methods
- Methods can have any access modifier, but no arguments and return void
- Applies to both beans discovered by component scan and declared by @Bean in @Configuration
- Defined by JSR-250, not Spring (javax.annotation package)
- Also used by EJB3
- Alternative in @Configuration is @Bean(initMethod="init", destroyMethod="destroy") - can be used for third party beans

@PostConstruct

- Method is invoked by Spring after DI

@PreDestroy

- Method is invoked before bean is destroyed

- Is not called for prototype scoped beans!
- After application context is closed
- Only if JVM exits normally

Stereotypes and Meta-annotations

Stereotypes

- Spring has several annotations, which are themselves annotated by @Component
- @Service, @Controller, @Repository, @Configuration, ... more in other Spring Projects
- Are discoverable using component scan

Meta-annotations

- Used to annotate other annotations
- Eg. can create custom annotation, which combines @Service and @Transactional

[A] @Resource annotation

- From JSR-250, supported by EJB3
- Identifies dependencies for injection by name, not by type like @Autowired
- Name is spring bean name
- Can be used for field and setter DI, not for constructors
- Can be used with name @Resource("beanName") or without
    - If not provided, name is inferred from field name or tries injection by type if name fails
    - setAmount() → "amount" name

[A] JSR-330 - Dependency Injection for Java

- Alternative DI annotations
- Spring is valid JSR-330 implementation
- @ComponentScan also scans for JSR-330 annotations
- @Inject for injecting dependencies
    - Provides subset of @Autowired functionality, but often is enough
    - Always required
- @Named - Alternative to @Component
    - With or without bean name - @Named / @Named("beanName")
    - Default scope is "prototype"
- @Singleton - instead of @Named for singleton scoped beans

XML Configuration

- Original form of configuration in Spring
- External configuration in XML files
- Beans are declared inside <beans> tag
- Can be combined with java config
    - Xml config files can be imported to @Configuration using @ImportResource

    - @ImportResource({"classpath:com/example/foo/config.xml","classpath:com/example/foo/other-config.xml"})
    - For importing java config files, @Import is used
    - In xml, `<import resource="config.xml" />` can be used to import other xml config files, path is relative to current xml config file

```
<beans>
    <import resource="other-config.xml" />
    <bean id="fooBeanName" class="com.example.foo.Foo" />
    <bean id="barBeanName" class="com.example.bar.Bar" />
</beans>
```

Conponent scan can be also used in XML

```
<context:component-scan base-package="com.example.foo, com.example.bar"/>
```

Constructor injection

- using <constructor-arg> tag
- `ref` attribute is used for injecting beans
- `value` attribute is used for setting primitive values, primitives are auto-converted to proper type
    - supports numeric types, BigDecimal, boolean, Date, Resource, Locale
- Parameters can be in any order, are matched by type
- If necessary, order can be defined - `<constructor-arg ref="someBean" index="0"/>`
- Or using named constructor parameters - `<constructor-arg ref="someBean" name="paramName"/>`
    - Java 8+
    - OR compiled with debug-symbols enabled to preserve param names
    - OR @ConstructorProperties({ "firstParam", "secondParam" }) on constructor - order of items matches order of constructor params

```
<bean id="fooBeanName" class="com.example.foo.Foo">
    <constructor-arg ref="someBean"/>
    <constructor-arg val="42"/>
</bean>
```

Setter injection

- Using `<property\>` tag
- `ref` or `value` like with constructor injection
- Setter and constructor injection can be combined
- Can use `<value\>` tag inside of `<property\>`

```
<bean id="fooBeanName" class="com.example.foo">
    <constructor-arg ref="someBean"/>
    <property ref="someOtherBean"/>
    <property name="someProperty" val="42"/>
    <property name="someOtherProperty">
        <value>42</value>
    </property>
</bean>
```

Additional bean config

- @PostConstruct is equivalent to `init-method` attribute
- @PreDestroy is init to `destroy-method` attribute
- @Scope is equivalent to `scope` attribute, singleton if not specified
- @Lazy is equivalent to `lazy-init` attribute
- @Profile is equivalent to `profile` attribute on `<beans>` tag, `<beans>` tags can be nested

```
<beans profile="barProfile">
    <beans profile="fooProfile">
        <bean id="fooBeanName" class="com.example.foo.Foo"
scope="prototype" />
    </beans>
    <bean id="barBeanName" class="com.example.bar.Bar" lazy-init="true"
init-method="setup"
          destroy-method="teardown" />
</beans>
```

XML Namespaces

- Default namespace is usually beans

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
    <!--Config here-->
</beans>
```

- Many other namespaces available
  - context
  - aop (Aspect Oriented Programming)
  - tx (transactions)
  - util
  - jms
  - ...
- Provides tags to simplify configuration and hide detailed bean configuration
  - `<context:property-placeholder location="config.properties" />` instead of declaring PropertySourcesPlaceholderConfigurer bean manually
  - `<aop:aspectj-autoproxy />` Enables AOP (5+ beans)
  - `<tx:annotation-driven />` Enables Transactions (15+ beans)
- XML Schema files can have version specified (spring-beans-4.2.xsd) or not (spring-beans.xsd)
  - If version not specified, most recent is used
  - Usually, no version number is preferred as it means easier upgrade to newer framework version

Accessing properties in XML

- Equivalent of @Value in XML
- Still need to declare PropertySourcesPlaceholderConfigurer

- `<context:property-placeholder location="config.properties" />` from `context` namespace

```
<beans ...beans and context namespaces here...>
    <context:property-placeholder location="config.properties" />
    <bean id="beanName" class="com.example.MyBean">
        <property name="foo" value="${foo}" />
        <property name="bar" value="${bar}" />
    </bean>
</beans>
```

Can load different property files based on spring profiles

```
<context:property-placeholder properties-ref="config"/>
<beans profile="foo">
    <util:properties id="config" location="foo-config.properties"/>
</beans>
<beans profile="bar">
    <util:properties id="config" location="bar-config.properties"/>
</beans>
```

Spring Application Lifecycle

- Three main phases - Initialization, Use, Destruction
- Lifecycle independent on configuration style used (java / XML)

Initialization phase

- Application is prepared for usage
- Application not usable until init phase is complete
- Beans are created and configured
- System resources allocated
- Phase is complete when application context is fully initialized

Bean initialization process

1. Bean definitions loaded
2. Bean definitions post-processed
3. For each bean definition
    1. Instantiate bean
    2. Inject dependencies
    3. Run Bean Post Processors
        1. Pre-Initialize
        2. Initialize
        3. Post-Initialize

Loading and post-processing bean definitions

- Explicit bean definitions are loaded from java @Configuration files or XML config files

- Beans are discovered using component scan
- Bean definitions are added to BeanFactory with its id
- Bean Factory Post Processor beans are invoked - can alter bean definitions
- Bean definitions are altered before beans are instantiated based on them
- Spring has already many BFPPs - replacing property placeholders with actual values, registering custom scope,...
- Custom BFPPs can be created by implementing BeanFactoryPostProcessor interface

Instatiating beans

- Spring creates beans eagerly by default unless declared as @Lazy (or lazy-init in XML)
- Beans are created in right order to satisfy dependencies
- After a bean is instantiated, it may be post-processed (similar to bean definitions post-processing)
- One kind of post-processors are Initializers - @PostConstruct/init-method
- Other post processors can be invoked either before initialization or after initialization
- To create custom Bean Post Processor, implement interface BeanPostProcessor

```
public interface BeanPostProcessor {
    public Object postProcessAfterInitialization(Object bean, String
beanName);
    public Object postProcessBeforeInitialization(Object bean,String
beanName);
}
```

Note that return value is the post-processed bean. It may be the bean with altered state, however, it may be completely new object. It is very important - post processor can return proxy of bean instead of original one - used a lot in spring - AOP, Transactions, ... A proxy can add dynamic behavior such as security, logging or transactions without its consumers knowing.

Spring Proxies

- Two types of proxies
- JDK dynamic proxies
    - Proxied bean must implement java interface
    - Part of JDK
    - All interfaces implemented by the class are proxied
    - Based on proxy implementing interfaces
- CGLib proxies
    - Is not part of JDK, included in spring
    - Used when class implements no interface
    - Cannot be applied to final classes or methods
    - Based on proxy inheriting the base class

Use phase

- App is in this phase for the vast majority of time
- Beans are being used, business logic performed

Destruction phase

- Application shuts down
- Resources are being released
- Happens when application context closes
- @PreDestroy and destroyMethod methods are called
- Garbage Collector still responsible for collecting objects

Testing Spring Applications

Test Types

Unit Tests

- Without spring
- Isolated tests for single unit of functionality, method level
- Dependencies interactions are not tested - replaced by either mocks or stubs
- Controllers; Services; ...

Integration Tests

- With spring
- Tests interactions between components
- Dependencies are not mocked/stubbed
- Controllers + Services + DAO + DB

Spring Test

- Distributed as separate artifact - spring-test.jar
- Allows testing application without the need to deploy to external container
- Allows to reuse most of the spring config also in tests
- Consist of several JUnit support classes
- SpringJUnit4ClassRunner - application context is cached among test methods - only one instance of app context is created per test class
- Test class needs to be annotated with @RunWith(SpringJUnit4ClassRunner.class)
- Spring configuration classes to be loaded are specified in @ContextConfiguration annotation
    - If no value is provided (@ContextConfiguration), config file `${classname}-context.xml` in the same package is imported
    - XML config files are loaded by providing string value to annotation - `@ContextConfiguration("classpath:com/example/test-config.xml")`
    - Java @Configuration files are loaded from `classes` attribute - `@ContextConfiguration(classes={TestConfig.class, OtherConfig.class})`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={TestConfig.class, OtherConfig.class})
public final class FooTest  {
    //Autowire dependencies here as usual
    @Autowired
    private MyService myService;

    //...
}
```

@Configuration inner classes (must be static) are automatically detected and loaded in tests

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(...)
public final class FooTest  {
    @Configuration
    public static class TestConfiguration {...}
}
```

Testing with spring profiles

- @ActiveProfiles annotation of test class activates profiles listed
- @ActiveProfiles({ "foo", "bar" })

Test Property Sources

- @TestPropertySource overrides any existing property of the same name.
- Default location is "[classname].propeties"
- @TestPropertySource(properties= { "username=foo" })

Testing with in-memory DB

- Real DB usually replaced with in-memory DB for integration tests
- No need to install DB server
- In-memory DB initialized with scripts using @Sql annotation
- Can be either on class level or method level
- If on class level, SQL is executed before every test method in the class
- If on method level, SQL is executed before invoking particular method
- Can be declared with or without value
- When value is present, specified SQL script is executed - @Sql({ "/sql/foo.sql", "/sql/bar.sql" })
- When no value is present, defaults to ClassName.methodName.sql
- Can be specified whether SQL is run before (by default) or after the test method
- @Sql(scripts="/sql/foo.sql", executionPhase=Sql.ExecutionPhase.AFTER_TEST_METHOD)
- Can provide further configuration using `config` param - error mode, comment prefix, separator, ...

Aspect Oriented Programming

- AOP solves modularization of cross-cutting concerns
- Functionality, which would be scattered all over the app, but is not core functionality of the class
    - Logging
    - Security

    - Performance monitoring
    - Caching
    - Error handling
    - ...
- Keeps good separation of concerns, does not violate single responsibility principle
- Solves two main problems
    - Code Tangling - Coupling of different concerns in one class - eg. business logic coupled with security and logging
    - Code scattering - Same code scattered among multiple classes, code duplication - eg. same security logic in each class

Writing AOP Apps

1. Write main application logic
2. Write Aspects to address cross-cutting concerns
3. Weave aspects into the application to be applied in right places

AOP Technologies

- AspectJ
    - Original AOP Technology
    - Uses bytecode modification for aspect weaving
    - Complex, lot of features
- Spring AOP
    - AspectJ integration
    - Subset of AspectJ functionality
    - Uses dynamic proxies instead of bytecode manipulation for aspect weaving
    - Limitations
        - Can advise non-private methods only
        - Only applicable to Spring Managed Beans
        - AOP is not applied when one method of the class calls method on the same class (will bypass the proxy)

AOP Concepts

- Join Point - A point in the execution of the app, where AOP code will be applied - method call, exception thrown
- Pointcut - Expression that matches one or multiple Join Points
- Advice - Code to be executed at each matched Join Point
- Aspect - Module encapsulating Pointcut and Advice
- Weaving - Technique, by which aspects are integrated into main code

Enabling AOP

- In XML - `<aop:aspectj-autoproxy />`
- In Java Config - @EnableAspectJAutoProxy on @Configuration class
- Create @Aspect classes, which encapsulate AOP behavior
    - Annotate with @Aspect

- Must be spring managed bean - either explicitly declare in configuration or discover via component-scan
- Class methods are annotated with annotation defining advice (eg. Before method call, After, if exception), the annotation value defines pointcut

```
@Aspect
public class MyAspect {
    @Before("execution(void set*(*))")
    public void beforeSet() {
        //Do something before each setter call
    }
}
```

Pointcuts

- Spring AOP uses subset of AspectJ's expression language for defining pointcuts
- Can create composite conditions using ||, && and !
- `execution(<method pattern>)` - method must match the pattern provided
- `[Modifiers] ReturnType [ClassType] MethodName ([Arguments]) [throws ExceptionType]`
    - `execution(void com.example.*Service.set*(..))`
    - Means method call of Services in com.example package starting with "set" and any number of method parameters
- More examples
    - `execution(void find*(String))` - Any methods returning void, starting with "find" and having single String argument
    - `execution(* find(*))` - Any method named "find" having exactly one argument of any type
    - `execution(* find(int, ..))` - Any method named "find", where its first argument is int (.. means 0 or more)
    - `execution(@javax.annotation.security.RolesAllowed void find(..))` - Any method returning void named "find" annotated with specified annotation
    - `execution(* com.*.example.*.*(..))` - Exactly one directory between com and example
    - `execution(* com..example.*.*(..))` - Zero or more directoryies between com and example
    - `execution(* *..example.*.*(..))` - Any sub-package called "Example"

Advice Types

Before

- Executes before target method invocation
- If advice throws an exception, target is not called
- @Before("expression")

After returning

- Executes after successful target method invocation
- If target throws an exception, advice is not called

- Return value of target method can be injected to the annotated method using `returning` param
- @AfterReturning(value="expression", returning="paramName")

```
    @AfterReturning(value="execution(* service..*.*(..))", 
returning="retVal")
    public void doAccessCheck(Object retVal) {
        //...
    }
```

After throwing

- Called after target method invocation throws an exception
- Exception object being thrown from target method can be injected to the annotated method using `throwing` param
- It will not stop exception from propagating but can throw another exception
- Stopping exception from propagating can be achieved using @Around advice
- @AfterThrowing(value="expression", throwing="paramName")

```
    @AfterThrowing(value="execution(* service..*.*(..))", 
throwing="exception")
    public void doAccessCheck(Exception exception) {
        //...
    }
```

After

- Called after target method invocation, no matter whether it was successful or there was an exception
- @After("expression")

Around

- ProceedingJoinPoint parameter can be injected - same as regular JoinPoint, but has proceed() method
- By calling proceed() method, target method will be invoked, otherwise it will not be

[A] XML AOP Configuration

- Instead of AOP annotations (@Aspect, @Before, @After, ...), pure XML onfig can be used
- Aspects are just POJOs with no spring annotations or dependencies whatsoever
- `aop` namespace

```
<aop:config>
    <aop:aspect ref="myAspect">
        <aop:before pointcut="execution(void set*(*))" method="logChange"/>
    </aop:aspect>

    <bean id="myAspect" class="com.example.MyAspect" />
</aop:config>
```

[A]Named Pointcuts

- Pointcut expressions can be named and then referenced
- Makes pointcut expressions reusable
- Pointcuts can be externalized to a separate file
- Composite pointcuts can be split into several named pointcuts

XML

```
<aop:config>
    <aop:pointcut id="pointcut" expression="execution(void set*(*))"/>

    <aop:aspect ref="myAspect">
        <aop:after-returning pointcut-ref="pointcut" method="logChange"/>
    </aop:aspect>

    <bean id="myAspect" class="com.example.MyAspect" />
</aop:config>
```

Java

```
//Pointcut is referenced here by its ID
@Before("setters()")
public void logChange() {
    //...
}

//Method name is pointcut id, it is not executed
@Pointcut("execution(void set*(*))")
public void setters() {
    //...
}
```

[A] Context Selecting Pointcuts

- Data from JoinPoint can be injected as method parameters with type safety
- Otherwise they would need to be obtained from JoinPoint object with no type safety guaranteed
- `@Pointcut("execution(void example.Server.start(java.util.Map)) && target(instance) && args(input)")`
    - target(instance) injects instance on which is call performed to "instance" parameter of the method
    - args(input) injects target method parameters to "input" parameter of the method

REST

- Representational State Transfer
- Architectural style
- Stateless (clients maintains state, not server), scalable → do not use HTTP session
- Usually over HTTP, but not necessarily
- Entities (e.g. Person) are resources represented by URIs
- HTTP methods (GET, POST, PUT, DELETE) are actions performed on resource (like CRUD)
- Resource can have multiple representations (different content type)

- Request specifies desired representation using HTTP Accept header, extension in URL (.json) or parameter in URL (format=json)
- Response states delivered representation type using Content-Type HTTP header

HATEOAS

- Hypermedia As The Engine of Application State
- Response contains links to other items and actions → can change behavior without changing client
- Decoupling of client and server

JAX-RS

- Java API for RESTful web services
- Part of Java EE6
- Jersey is reference implementation

```
@Path("/persons/{id}")
public class PersonService {
    @GET
    public Person getPerson(@PathParam("id") String id) {
        return findPerson(id);
    }
}
```

RestTemplate

- Can be used to simplify HTTP calls to RESTful api
- Message converters supported - Jackson, GSON
- Automatic input/output conversion - using HttpMessageConverters
    - StringHttpMessageConvertor
    - MarshallingHttpMessageConvertor
    - MappingJackson2XmlHttpMessageConverter
    - GsonHttpMessageConverter
    - RssChannelHttpMessageConverter
- AsyncRestTemplate - similar to regular one, allows asynchronous calls, returns ListenableFuture

Spring Rest Support

- In MVC Controllers, HTTP method consumed can be specified
    - @RequestMapping(value="/foo", method=RequestMethod.POST)
- @ResponseStatus can set HTTP response status code
    - If used, void return type means no View (empty response body) and not default view!
    - 2** - success (201 Created, 204 No Content,...)
    - 3** - redirect
    - 4** - client error (404 Not found, 405 Method Not Allowed, 409 Conflict,...)
    - 5** - server error

- @ResponseBody before controller method return type means that the response should be directly rendered to client and not evaluated as a logical view name
    - Uses converters to convert return type of controller method to requested content type
- @RequestHeader can inject value from HTTP request header as a method parameter
- @RequestBody - Injects body of HTTP request, uses converters to convert request data based on content type
    - public void updatePerson(@RequestBody Person person, @PathVariable("id") int personId)
    - Person can be converted from JSON, XML, ...
- HttpEntity<>
    - Controller can return HttpEntity instead of view name - and directly set response body, HTTP response headers
    - `return new HttpEntity<ReturnType>(responseBody, responseHeaders);`
- HttpMessageConverter
    - Converts HTTP response (annotated by @ResponseBody) and request body data (annotated by @RequestBody)
    - `@EnableWebMvc` or `<mvc:annotation-driven/>` enables it
    - XML, Forms, RSS, JSON,...
- @RequestMapping can have attributes produces and consumes to specify input and output content type
    - `@RequestMapping (value= "/person/{id}", method=RequestMethod.GET, produces = {"application/json"})`
    - `@RequestMapping (value= "/person/{id}", method=RequestMethod.POST, consumes = { "application/json" })`
- Content Type Negotiation can be used both for views and REST endpoints
- @RestController - Equivalent to @Controller, where each method is annotated by @ResponseBody
- @ResponseStatus(HttpStatus.NOT_FOUND) - can be used on exception to define HTTP code to be returned when given exception is thrown
- @ExceptionHandler({MyException.class}) - Controller methods annotated with it are called when declared exceptions are thrown
    - Method can be annotated also with @ResponseStatus to define which HTTP result code should be sent to the client in the case of the exception
- When determining address of child resources, UriTemplate should be used so absolute urls are not hardcoded

```
StringBuffer currentUrl = request.getRequestURL();
String childIdentifier = ...;//Get Child Id from url requested
UriTemplate template = new
UriTemplate(currentUrl.append("/{childId}").toString());
String childUrl = template.expand(childIdentifier).toASCIIString();
```

Microservices in Spring

Microservices

- Classic monolithic architecture
    - Single big app

    - Single persistence (usually relational DB)
    - Easier to build
    - Harder to scale up
    - Complex and large deployment process
- Microservices architecture
    - Each microservice has its own storage best suited for it (Relational DB, Non-Relation DB, Document Store, ...)
    - Each microservice can be in different language with different frameworks used, best suited for its purpose
    - Each microservice has separate development, testing and deployment
    - Each microservice can be developed by separate team
    - Harder to build
    - Easier to extend and maintain
    - Easier to scale up
    - Easy to deploy or update just one microservice instead of the whole monolith
- One microservice usually consists of several services in service layer and corresponding data store
- Microservices are well suited to run in the cloud
    - Easily manage scaling - number of microservice instances
    - Scaling can be done dynamically
    - Provided load balancing across all the instances
    - Underlying infrastructure agnostic
    - Isolated containers
- Apps deployed to cloud don't necessary have to be microservices
    - Enables to gradually change existing apps to microservice architecture
    - New features of monolith can be added as microservices
    - Existing features of monolith can be one by one refactored to microservices

Spring Cloud

- Provides building blocks for building Cloud and microservice apps
    - Intelligent routing among multiple instances
    - Service registration and discovery
    - Circuit Breakers
    - Distributed configuration
    - Distributed messaging
- Independent on specific cloud environment - Supports Heroku, AWS, Cloud foundry, but can be used without any cloud
- Built on Spring Boot
- Consist of many sub-projects
    - Spring Cloud Config
    - Spring Cloud Security
    - Spring Cloud Connectors
    - Spring Cloud Data Flow
    - Spring Cloud Bus
    - Spring Cloud for Amazon Web Services
    - Spring Cloud Netflix
    - Spring Cloud Consul
    - ...

Building Microservices

1. Create Discovery Service
2. Create a Microservice and register it with Discovery Service
3. Clients use Microservice using "smart" RestTemplate, which manages load balancing and automatic service lookup

Discovery Service

- Spring Cloud supports two Discovery Services
    - Eureka - by Netflix
    - Consul.io by Hashicorp (authors of Vagrant)
    - Both can be set up and used for service discovery easily in Spring Cloud
    - Can be later easily switched
- On top of regular Spring Cloud and Boot dependencies (spring-cloud-starter, spring-boot-starter-web, spring-cloud-starter-parent for parent pom), dependency for specific Discovery Service server needs to be provided (eg. spring-cloud-starter-eureka-server)
- Annotate @SpringBootApplication with @EnableEurekaServer
- Provide Eureka config to application.properties (or application.yml)

```
server.port=8761
#Not a client
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false

logging.level.com.netflix.eureka=OFF
logging.level.com.netflix.discovery=OFF
```

Microservice Registration

- Annotate @SpringBootApplication with @EnableDiscoveryClient
- Each microservice is a spring boot app
- Fill in application.properties with app name and eureka server URL
    - `spring.application.name` is name of the microservice for discovery
    - `eureka.client.serviceUrl.defaultZone` is URL of the Eureka server

Microservice Usage

- Annotate client app using microservice (@SpringBootApplication) with @EnableEurekaServer
- Clients call Microservice using "smart" RestTemplate, which manages load balancing and automatic service lookup
- Inject RestTemplate using @Autowired with additional @LoadBalanced
- "Ribbon" service by Netflix provides load balancing
- RestService automatically looks up microservice by logical name and provides load-balancing

```
@Autowired
@LoadBalanced
private RestTemplate restTemplate;
```

Then call specific Microservice with the restTemplate

```
restTemplate.getForObject("http://persons-microservice-name/persons/{id}",
Person.class, id);
```

Spring Security

- Independent on container - does not require EE container, configured in application and not in container
- Separated security from business logic
- Decoupled authorization from authentication
- Principal - User, device, or system that is performing and action
- Authentication
    - Process of checking identity of a principal (credentials are valid)
    - basic, digest, form, X.509, ...
    - Credentials need to be stored securely
- Authorization
    - Process of checking a principal has privileges to perform requested action
    - Depends on authentication
    - Often based on roles - privileges not assigned to specific users, but to groups
- Secured item - Resource being secured

Configuring Spring Security

- Annotate your @Configuration with @EnableWebSecurity
- Your @Configuration should extend WebSecurityConfigurerAdapter

```
@Configuration
@EnableWebSecurity
public class HelloWebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

  @Override
  public void configureGlobal(AuthenticationManagerBuilder
authManagerBuilder) throws Exception {
    //Global security config here
  }

  @Override
  protected void configure(HttpSecurity httpSecurity) throws Exception {
      //Web security specific config here
  }
}
```

web.xml - Declare spring security filter chain as a servlet filter

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
```

```
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Authorization

- Process of checking a principal has privileges to perform requested action
- Specific urls can have specific Role or authentication requirements
- Can be configured using HttpSecurity.authorizeRequests().*

```
@Configuration
@EnableWebSecurity
public class HelloWebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity httpSecurity) throws Exception {

httpSecurity.authorizeRequests().antMatchers("/css/**","/img/**","/js/**").
permitAll()

.antMatchers("/admin/**").hasRole("ADMIN")

.antMatchers("/user/profile").hasAnyRole("USER","ADMIN")

.antMatchers("/user/**").authenticated()

.antMatchers("/user/private/**").fullyAuthenticated()

.antMatchers("/public/**").anonymous();
  }
}
```

- Rules are evaluated in the order listed
- Should be from the most specific to the least specific
- Options
    - `hasRole()` - has specific role
    - `hasAnyRole()` - multiple roles with OR
    - `hasRole(FOO)` AND `hasRole(BAR)` - having multiple roles
    - `isAnonymous()` - unauthenticated
    - `isAuthenticated()` - not anonymous

Authentication

Authentication provider

- Processes authentication requests and returns Authentication object
- Default is DaoAuthenticationProvider
    - UserDetailsService implementation is needed to provide credentials and authorities
    - Built-in implementations: LDAP, in-memory, JDBC
    - Possible to build custom implementation

- Can provide different auth configuration based on spring @Profiles - in-memory for development, JDBC for production etc.

In-memory Authentication provider

```
@Configuration
@EnableWebSecurity
public class HelloWebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

  @Override
  public void configureGlobal(AuthenticationManagerBuilder
authManagerBuilder) throws Exception  {

authManagerBuilder.inMemoryAuthentication().withUser("alice").password("let
mein").roles("USER").and()

.withUser("bob").password("12345").roles("ADMIN").and();
  }
}
```

JDBC Authentication provider

- Authenticates against DB
- Can customize queries using .usersByUsernameQuery(), .authoritiesByUsernameQuery(), groupAuthoritiesByUsername()
- Otherwise default queries will be used

```
@Configuration
@EnableWebSecurity
public class HelloWebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

  @Autowired
  private DataSource dataSource;

  @Override
  public void configureGlobal(AuthenticationManagerBuilder
authManagerBuilder) throws Exception {
      authManagerBuilder.jdbcAuthentication().dataSource(dataSource)
                                             .usersByUsernameQuery(...)

.authoritiesByUsernameQuery(...)

.groupAuthoritiesByUsername(...);
  }
}
```

Password Encoding

- Supports passwords hashing (md5, sha, ...)
- Supports password salting - adding string to password before hashing - prevents decoding passwords using rainbow tables

```
@Configuration
@EnableWebSecurity
```

```
public class HelloWebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

  @Autowired
  private DataSource dataSource;

  @Override
  public void configureGlobal(AuthenticationManagerBuilder
authManagerBuilder) throws Exception {
      authManagerBuilder.jdbcAuthentication().dataSource(dataSource)
                                             .passwordEncoder(new
StandardPasswordEncoder("this is salt"));
  }
}
```

Login and Logout

```
@Configuration
@EnableWebSecurity
public class HelloWebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity httpSecurity) throws Exception {
      httpSecurity.authorizeRequests().formLogin().loginPage("/login.jsp")
.permitAll()
                                      .and()
                                      .logout().permitAll();
  }
}
```

Login JSP Form

```
<c:url var="loginUrl" value="/login.jsp" />
<form:form action="${loginUrl}" method="POST">
    <input type="text" name="username"/>
    <input type="password" name="password"/>
    <input type="submit" name="submit" value="LOGIN"/>
</form:form>
```

Spring Security Tag Libraries

- Add taglib to JSP

```
 <%@ taglib prefix="security"
uri="http://www.springframework.org/security/tags" %>
```

- Facelet fags for JSF also available
- Displaying properties of Authentication object - `<security:authentication property=“principal.username”/>`
- Display content only if principal has certain role

```
<security:authorize access="hasRole('ADMIN')">
    <p>Admin only content</p>
</security:authorize>
```

- Or inherit the role required from specific url (roles required for specific urls are centralized in config and not across many JSPs)

```
<security:authorize url="/admin">
    <p>Admin only content</p>
</security:authorize>
```

Method Security

- Methods (e.g. service layer) can be secured using AOP
- JSR-250 or Spring annotations or pre/post authorize

JSR-250

- Only supports role based security
- `@EnableGlobalMethodSecurity(jsr250Enabled=true)` on @Configuration to enable
- On method level `@RolesAllowed("ROLE_ADMIN")`

Spring @Secured Annotations

- `@EnableGlobalMethodSecurity(securedEnabled=true)` on @Configuration to enable
- `@Secured("ROLE_ADMIN")` on method level
- Supports not only roles - e.g. @Secured("IS_AUTHENTICATED_FULLY")
- SpEL not supported

Pre/Post authorize

- `@EnableGlobalMethodSecurity(prePostEnabled=true)` on @Configuration to enable
- Pre authorize - can use SpEL (@Secured cannot), checked before annotated method invocation
- Post authorize - can use SpEL, checked after annotated method invocation, can access return object of the method using returnObject variable in SPEL; If expression resolves to false, return value is not returned to caller
- `@PreAuthorize("hasRole('ROLE_ADMIN')")`

Spring Web

- Provides several web modules
    - Spring MVC - Model View Controller framework
    - Spring Web Flow - Navigation flows (wizard-like)
    - Spring Social - Integration with facebook, twitter, ...
    - Spring mobile - switching between regular and mobile versions of site
- Spring can be integrated with other web frameworks, some integration provided by Spring itself
    - JSF
    - Struts 2 (Struts 1 no longer supported as of Spring 4+)

    - Wicket
    - Tapestry 5
    - ...
- Spring web layer on top of regular spring application layer
- Initialized using regular servlet listener
- Can be configured either in web.xml or using AbstractContextLoaderInitializer - implements WebApplicationInitializer, which is automatically recognized and processed by servlet container (Servlets 3.0+)

Basic configuration

AbstractContextLoaderInitializer

```
public class WebAppInitializer extends AbstractContextLoaderInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        AnnotationConfigWebApplicationContext applicationContext = new
AnnotationConfigWebApplicationContext();
        //Configure app context here
        return applicationContext;
    }
}
```

- Servlet container looks for classes implementing ServletContainerInitializer (Spring provides SpringServletContainerInitializer)
- SpringServletContainerInitializer looks for classes implementing WebApplicationInitializer, which specify configuration instead of web.xml
- Spring provides two convenience implementations
    - AbstractContextLoaderInitializer - only Registers ContextLoaderListener
    - AbstractAnnotationConfigDispatcherServletInitializer - Registers ContextLoaderListener and defines Dispatcher Servlet, expects JavaConfig

web.xml

- Register spring-provided Servlet listener
- Provide spring config files as context param - contextConfigLocation
- Defaults to WEB-INF/applicationContext.xml if not provided
- can provide spring active profiles in spring.profiles.active

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/config.xml
        /WEB-INF/other-config.xml
    </param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>profile1,profile2</param-value>
</context-param>
```

Dependency injection in servlets

- in init() of HttpServlet cannot access spring beans as they are not available yet
- need to use `WebApplicationContextUtils` - provides application Context through Servlet Context
- Only needed in servlets - other components such as @Controllers can use regular Dependency Injection

```
public class MyServlet extends HttpServlet {
    private MyService service;

    public void init() {
        ApplicationContext context =
WebApplicationContextUtils.getRequiredWebApplicationContext(getServletConte
xt());
        service = (MyService) context.getBean("myService");
    }

    ...
}
```

Spring Web flow

- Provides support of stateful navigation flows
- Handles browser back button
- Handles double submit problem
- Support custom scopes of beans - flow scope, flash scope, ...
- Configuration of flows defined in XML
    - Defines flow states - view, action, end, ...
    - Defines transition between states

Spring MVC

- Spring Web Framework based on Model-View-Controller pattern
- Alternative to JSF, Struts, Wicket, Tapestry, ...
- Components such as Controllers are Spring-managed beans
- Testable POJOs
- Uses Spring Configuration
- Supports a wide range of view technologies - JSP, Freemarker, Velocity, Thymeleaf, ...

Dispatcher Servlet

- Front controller pattern
- Handles all incoming requests and delegates them to appropriate beans

- All the web infrastructure beans are customizable and replaceable
- DS is defined as servlet in web.xml or through WebApplicationInitializer interface
- Separate Web Application context on top of regular app context
- Web app context can access beans from root context but not vice versa

```
public class WebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    //@Configuration classes for root application context
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{ RootConfig.class };
    }

    //@Configuration classes for web application contet
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{ WebConfig.class };
    }

    //Urls handled by Dispatcher Servlet
    @Override
    protected String[] getServletMappings() {
        return new String[]{"/app/*"};
    }

    ...
}
```

Controllers

- Annotated with @Controller
- @RequestMapping annotation on controller's methods specifies which requests should the method handle

@RequestMapping

- Can specify URL, which annotated method should handle - @RequestMapping("/foo")
    - => server-url/app-context-root/servlet-mapping/**request-mapping**
    - can use wildcards @RequestMapping("/foo/*")
- Can specify HTTP method, which annotated method should handle

@RequestParam

- Can specify parameter from http request to be injected as method parameter

```
@RequestMapping("/foo")
public String foo(@RequestParam("bar") String bar) {
    ...
}
```

- This will extract value "xxx" from requested url `/foo?bar=xxx`

@PathVariable

- Can extract value as a method parameter from the url requested
- eg. Extract "1" from `/persons/1`

```
@RequestMapping("/persons/{personId}")
public String foo(@PathVariable("personId") String personId) {
    ...
}
```

- Other method parameters can be injected as well - @RequestHeader(), @Cookie,...
- Some parameters injects spring by type - HttpServletRequest, HttpServletResponse, Principal, HttpSession, ...

Views

- Views render output to the client based on data in the model
- Many built-in supported view technologies - Freemarker, Velocity, JSP, ...
- ViewResolver selects specific View based on logical view name returned by the controller methods

View Resolver

- Translates logical view name to actual View
- This way controller is not coupled to specific view technology (returns only logical view name)
- Default View resolver already configured is InternalResourceViewResolver, which is used to render JSPs (JstlView). Configures prefix and suffix to logical view name which then results in a path to specific JSP.
- Can register custom resolvers

View Resolution Sequence

1. Controller returns logical view name to DispatcherServlet.
2. ViewResolvers are asked in sequence (based on their Order).
3. If ViewResolver matches the logical view name then returns which View should be used to render the output. If not, it returns null and the chain continues to the next ViewResolver.
4. Dispatcher Servlet passes the model to the Resolved View and it renders the output.

Spring MVC Quick Start

1. Register Dispatcher servlet (web.xml or in Java)
2. Implement Controllers
3. Register Controllers with Dispatcher Servlet
    - Can be discovered using component-scan
4. Implement Views
    - eg. write JSP pages
5. Register View resolver or use the default one
    - Need to set prefix (eg. /WEB-INF/views/) and suffix (eg. .jsp)

6. Deploy

Spring Boot

Basics

- Convention over configuration - pre-configures Spring app by reasonable defaults, which can be overridden
- Maven and Gradle integration
- MVC enabled by having spring-boot-starter-web as a dependence
    - Registers Dispatcher servlet
    - Does same as @EnableWebMvc + more
    - Automatically serves static resources from /static, /public, /resources or /META-INF/resources
    - Templates are served from /templates (Velocity, FreeMarker, Thymeleaf)
    - Registers BeanNameViewResolver if beans implementing View interface are found
    - Registers ContentNegociatingViewResolver, InternalResourceViewResolver (prefix and suffix configurable in application.properties)
    - Customisation of auto-configured beans should be done using WebMvcConfigurerAdapter as usual

@SpringBootApplication

- Main Class annotated with @SpringBootApplication, can be run as a jar with embedded application server (Tomcat by default, can be changed for example to Jetty or Undertow)
- Actually consists of three annotations @Configuration, @EnableAutoConfiguration and @ComponentScan
- @EnableAutoConfiguration configures modules based on presence of certain classes on classpath - based on @Conditional
- Manually declared beans usually override beans automatically created by AutoConfiguration (@ConditionalOnMissingBean is used), usually bean type and not name matters
- Can selectively exclude some AutoConfigutation classes
  `@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)`

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
  //run in embedded container
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }

  //run as war, needs to have <packaging>war</packaging> in pom.xml
  @Override
  protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
    return application.sources(Application.class);
  }
}
```

Dependencies

- Need to add proper maven parent and dependencies
- Using "starter" module dependencies → using transitive dependencies bundles versions which are tested to work well together
    - Can override version of specific artifacts in pom.xml

```
<properties>
    <spring.version>4.2.0.RELEASE</spring.version>
</properties>
```

- spring-boot-starter, spring-boot-starter-web, spring-boot-starter-test, ...
- Parent pom defines all the dependencies using dependency management, specific versions are not defined in our pom.xml
- Only version which needs to be specifically declared is parent pom version

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.0.RELEASE</version>
</parent>
```

Application Configuration

- Application configuration is externalized by default to application.properties file
- Alternatively, can use YAML configuration - application.yml by default
- Located in workingdirectory/config or working directory or classpath/config or classpath
- PropertySource automatically created

Configuration resolution sequence

1. Check /config subdirectory of the working directory for application.properties file
2. Check working directory
3. Check config package in the classpath
4. Check classpath root
5. Create property source based on files found

Logging

- By default Logback over SLF4J
- By default logs to console, but can define log file

```
#Logging through SLF4J
logging.level.org.springframework=DEBUG
logging.level.com.example=INFO
logging.file=logfile.log
#OR spring.log file in to configured path
logging.path=/log
```

To change logging framework from default logback

1. Exclude logback dependency `ch.qos.logback.logback-classic`
2. Add desired logging framework dependency - `eg. org.slf4j.slf4j-log4j12`

DataSource

- either include spring-boot-starter-jdbc or spring-boot-starter- data-jpa
- JDBC driver required on classpath, datasource will be created automatically
- Tomcat JDBC as default pool, other connection pools can be used if present - eg.HikariCP

```
#Connection
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
spring.datasource.driver-class-name=

#Scripts to be executed
spring.datasource.schema=
spring.datasource.data=

#Connection pool
spring.datasource.initial-size=
spring.datasource.max-active=
spring.datasource.max-idle=
spring.datasource.min-idle=
```

Web Container

```
server.port=
server.address=
server.session-timeout=
server.context-path=
server.servlet-path=
```

[A]Web container can be configured in Java dynamically by implementing EmbeddedServletContainerCustomizer interface and registering resulting class as a @Component

```
@Override
public void customize(ConfigurableEmbeddedServletContainer container) {
  container.setPort(8081);
  container.setContextPath("/foo");
}
```

Or if needed more fine-grained configuration - declare bean of type EmbeddedServletContainerFactory

YAML

- YAML - Yaml Ain't a Markup Language
- Alternative to configuration in .properties file
- Configuration can be hierarchical

```
server:
    port:
    address:
    session-timeout:
    context-path:
    servlet-path:
```

- YAML can contain configuration based on spring profiles
    - Divided by ---

```
    ---
    spring.profiles: development
    database:
      host: localhost
    user: dev
    ---
    spring.profiles: production
    database:
      host: 198.18.200.9
      user: admin
```

- Or each profile can have its own dedicated file
    - `application-profilename.yml`

@ConfigurationProperties

- Class is annotated with @ConfigurationProperties(prefix= "com.example")
- Fields of the class are automatically injected with values from properties
- @ConfigurationProperties(prefix= "com.example") + com.example.foo → foo field injected

```
@ConfigurationProperties(prefix="com.example")
public class MyProperties {
    private String foo; //com.example.foo
    private int bar; //com.example.bar, type conversion applied

}
```

- Needs to be enabled on @Configuration class - `@EnableConfigurationProperties(MyProperties.class)`

Embedded container

- Spring boot can run embedded application server from a jar file
- `spring-boot-starter-web` includes embedded Tomcat, can change to Jetty
    - `spring-boot-starter-jetty` as a dependency
    - exclude spring-boot-starter-tomcat dependency from `spring-boot-starter-web`
- Embedded app server recommended for Cloud Native applications

- Spring boot can produce jar or war (change packaging to `war`, extend SpringBootServletInitializer, override configure method)

```
@ComponentScan
@EnableAutoConfiguration
public class MyApplication extends SpringBootServletInitializer {
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(MyApplication.class);
    }

    //Still executable through traditional main method
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

- war can still be executed `java -jar myapp.war` or deployed to app server
- when using spring-boot-maven-plugin, `mvn package` produces two jars
    - traditional jar without dependencies
    - executable jar with all dependencies included
- Both war and jar with embedded app server are valid options

@Conditional

- Enables bean instantiatiation only when specific condition is met
- Core concept of spring boot
- Only when specific bean found - `@ConditionalOnBean(type={DataSource.class})`
- Only when specific bean is not found - `@ConditionalOnMissingBean`
- Only when specific class is on classpath - `@ConditionalOnClass`
- Only When specific class is not present on classpath - `@ConditionalOnMissingClass`
- Only when system property has certain value - `@ConditionalOnProperty(name="server.host", havingValue="localhost")`
- @Profile is a special case of conditional
- org.springframework.boot.autoconfigure contains a lot of conditionals for auto-configuration
    - eg. @ConditionalOnMissingBean(DataSource.class) → Created embedded data source
    - Specifically declared beans usually disable automatically created ones
    - If needed, specific autoconfiguration classes can be excluded explicitly
    - `@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)`

Testing

- Can use same configuration as Spring Boot application in tests without invoking main method
- @SpringApplicationConfiguration(classes= MyApplication.class)

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@SpringApplicationConfiguration(classes=MyApplication.class)
public class FooServiceTest {
    ...
}
```

- Or for testing web app - @WebAppConfiguration
- Initializes web app context

Data Management With Spring

- Spring supports many data-access technologies
- No matter what technology is used, consistent way of using
- JDBC, JPA, Hibernate, MyBatis, ...
- Spring manages most of the actions under the hood
  - Accessing data source
  - Establishing connection
  - Transactions - begin, rollback, commit
  - Close the connection
- Declarative transaction management (transactions through AOP proxies)

Exception handling

- Provides own set of exception so exceptions are not specific to underlying technology used
- Replaces checked exceptions with unchecked
  - Checked exceptions provide a form of tight coupling between layers
  - If exception is not caught must be declared in method signature
  - Exceptions from specific data access technology leak to service layer of the app - layers no more loosely coupled
- Spring provides DataAccessException
  - Does not reveal underlying specific technology (JPA, Hibernate, JDBC, ...)
  - Is not single exception but hierarchy of exceptions
  - DataAccessResource FailureException, DataIntegrityViolationException, BadSqlGrammarException, OptimisticLocking FailureException, ...
  - Unchecked (Runtime)

In-Memory Database

- Spring can create embedded DB and run specified init scripts
- H2, HSQL, Derby supported
- Using EmbeddedDatabaseBuilder

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    builder.setName("inmemorydb")
           .setType(EmbeddedDatabaseType.HSQL)
           .addScript("classpath:/inmemorydb/schema.db")
           .addScript("classpath:/inmemorydb/data.db");
```

```
            return builder.build();
}
```

Or in XML

```
<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:data.sql" />
</jdbc:embedded-database>
```

Alternatively, existing DB can be initialized with DataSource provided

```
<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:data.sql" />
</jdbc:initialize-database>
```

Or the same in Java

```
@Configuration
public class DatabaseInitializer {

    @Value("classpath:schema.sql")
    private Resource schemaScript;

    @Value("classpath:data.sql")
    private Resource dataScript;

    private DatabasePopulator databasePopulator() {
        ResourceDatabasePopulator populator = new
ResourceDatabasePopulator();
        populator.addScript(schemaScript);
        populator.addScript(dataScript);

        return populator;
    }

    @Bean
    public DataSourceInitializer dataSourceInitializer(DataSource
dataSource) {
        DataSourceInitializer initializer = new DataSourceInitializer();
        initializer.setDataSource(dataSource);
        initializer.setDatabasePopulator(databasePopulator());

        return initializer;
    }

}
```

[A] Caching

- Spring bean's method results can be cached
- Cache here is a map of cache-key, cached value
- Multiple caches supported
- Caching enabled using
    - `@EnableCaching` on `@Configuration` class level

  - or in xml `<cache:annotation-driven />`

@Cacheable

- Marks method as cacheable
- Its result will be stored in the cache
- Subsequent calls of the method with the same arguments will result in data being fetched from the cache
- Defines `value` (name of the cache - multiple caches supported) and `key` (key in given cache)
- `key` can use SpEL expressions
- Can also use `condition`, which uses SpEL

```
@Cacheable(value="mycache", key="#personName.toUpperCase()",
condition="#personName.length < 50")
public Person getPerson(String personName) {
    ...
}
```

- `@CacheEvict(value="cacheName")` clears cache before annotated method is called

Or alternatively can define caching in XML

```
<bean id="myService" class="com.example.MyService"/>

<aop:config>
    <aop:advisor advice-ref="cacheAdvice" pointcut="execution(*
*..MyService.*(..))"/>
</aop:config>

<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="myCache">
        <cache:cacheable method="myMethod" key="#id"/>
        <cache:cache-evict method="fetchData" all-entries="true" />
    </cache:caching>
</cache:advice>
```

Caching Manager

- Cache manager must be specified
- SimpleCacheManager, EHCache, Gemfire, Custom, ...

SimpleCacheManager

```
@Bean
public CacheManager cacheManager() {
    SimpleCacheManager manager = new SimpleCacheManager();
    Set<Cache> caches = new HashSet<Cache>();
    caches.add(new ConcurrentMapCache("fooCache"));
    caches.add(new ConcurrentMapCache("barCache"));
    manager.setCaches(caches);

    return manager;
}
```

EHCache

```
@Bean
public CacheManager cacheManager(CacheManager ehCache) {
    EhCacheCacheManager manager = new EhCacheCacheManager();
    manager.setCacheManager(ehCache);
    return manager;
}

@Bean EhCacheManagerFactoryBean ehCacheManagerFactoryBean(String
configLocation) {
    EhCacheManagerFactoryBean factory = new EhCacheManagerFactoryBean();
    factory.setConfigLocation(context.getResource(configLocation));
    return factory;
}
```

Gemfire

- Distributed cache replicated across multiple nodes
- GemfireCacheManager
- Supports transaction management - GemfireTransactionManager

```
<gfe:cache-manager p:cache-ref="gemfire-cache"/>
<gfe:cache id="gemfire-cache"/>
<gfe:replicated-region id="foo" p:cache-ref="gemfire-cache"/>
<gfe:partitioned-region id="bar" p:cache-ref="gemfire-cache"/>
```

JDBC with Spring

- Using traditional JDBC connection has many disadvantages
    - Lot of boilerplate code
    - Poor exception handling (cannot really react to exceptions)
- Spring JDBC Template to the rescue
- Implementing template method pattern
- Takes care of all the boilerplate
    - Establishing connection
    - Closing connection
    - Transactions
    - Exception handling
- Handles SQLExceptions properly -
- Transforms SQLExceptions into DataAccessExceptions
- Great simplification - user just performs queries
- Create one instance and reuse it - thread safe
- Uses runtime exceptions - no try-catch needed
- Can query for
    - primitives, String, Date, ...
    - Generic maps
    - Domain objects

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```

Query for simple object

```
template.queryForObject(sqlStatement, Date.class);
```

- Can bind variables represented by `?` in the query
- Values provided as varargs in the queryFor method
- Order of `?` matches order of vararg arguments

```
String sqlStatement = "select count(*) from ACCOUNT where balance > ? and
type = ?";
accountsCount = jdbcTemplate.queryForObject(sqlStatement, Long.class,
balance, type);
```

Query for generic collection

- Each result row returned as map of (Column name / Field value) pairs
- `queryForList` - when expecting multiple results
- `queryForMap` - when expecting single result

Query For single row

```
jdbcTemplate.queryForMap("select * from ACCOUNT where id=?", accountId);
```

Results in `Map<String, Object>` {ID=1, BALANCE=75000, NAME="Checking account", ...}

Query for multiple rows

```
return jdbcTemplate.queryForList("select * from ACCOUNT");
```

Results in `List<Map<String,Object>> { Map<String, Object> {ID=1, BALANCE=75000, NAME="Checking account", ...} Map<String, Object> {ID=2, BALANCE=15000, NAME="Checking account", ...} ... }`

Query for domain object

- May consider ORM for this
- Must implement RowMapper interface, which maps row of ResultSet to a domain object

```
public interface RowMapper<T> {
    T mapRow(ResultSet resultSet, int rowNum) throws SQLException;
}
```

Query using query for object

```
//Single domain object
Account account = jdbcTemplate.queryForObject(sqlStatement, rowMapper,
param1, param2);
//Multiple domain objects
List<Account> = jdbcTemplate.query(sqlStatement, rowMapper, param1,
param2);
```

Alternatively, RowMapper can be replaced by lambda expression

RowCallbackHandler

- Should be used when no return object is needed
    - Converting results to xml/json/...
    - Streaming results
    - ...
- Can be replaced by lambda expression

```
public interface RowCallbackHandler {
    void processRow(ResultSet resultSet) throws SQLException;
}
jdbcTemplate.query(sqlStatement, rowCallbackHandler, param1, param2);
```

ResultSetExtractor

- Can map multiple result rows to single return object
- can be replaced by lambda

```
public interface ResultSetExtractor<T> {
    T extractData(ResultSet resultSet) throws SQLException,
DataAccessException;
}
jdbcTemplate.query(sqlStatement, resultExtractor, param1, param2);
```

Inserts and updates

- returns number of rows modified
- jdbcTemplate.update() used both for updates and inserts

```
numberOfAffectedRows = jdbcTemplate.update(sqlStatement, param1, param2);
```

Transactions

- Set of operations that take place as a single atomic unit - all or nothing
    - E.g. Money transfer operation consists of deducting money from source account and adding them to the target account. If one part fails, other cannot be performed.
- ACID
    - Atomic - All operations are performed or none of them
    - Consistent - DB integrity constraints are not violated
    - Isolated - Transactions are isolated from each other
    - Durable - Committed changes are permanent
- When in transaction, one connection should be reused for the whole transaction

Java Transaction Management

- Local transaction - One resource is involved, transaction managed by resource itself (eg. JDBC)
- Global transaction - Multiple different resources involved in transaction, transaction managed by external Transaction Manager (e.g. JMS + two different databases)

- Multiple modules supporting transactions - JDBC, JPA, JMS, JTA, Hibernate
- Different API for global transactions and local transactions
- Different modules have different transaction API
- Usually programmatic transaction management - not declarative as in spring - need to call eg. begin transactions, commit, etc.
- Transaction management is cross-cutting concern and should be centralized instead scattered across all the classes
- Transaction demarcation should be independent on actual transaction implementation
- Should be done in service layer instead of data access layer

Spring Transaction Management

- Declarative transaction management instead of programmatic
- Transaction declaration independent on transaction implementation
- Same API for global and local transactions
- Hides implementation details
- Can easily switch transaction managers
- Can easily upgrade local transaction to global
- To enable transaction management in spring
    - enable transaction management
        - Java - @EnableTransactionManagement
        - [A] `<tx:annotation-driven/>` in XML
    - Declare a PlatformTransactionManager bean - "transactionManager" is default bean name
    - Mark methods as transactional (java or XML)

Transaction Manager

- Spring provides many implementations of PlatformTransactionManager interface
    - JtaTransactionManager
    - JpaTransactionManager
    - HibernateTransactionManager
    - DataSourceTransactionManager
    - ...

```
@Configuration
@EnableTransactionManagement
public class TransactionConfiguration {

    @Bean
    public PlatformTransactionManager transactionManager(DataSource
dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
```

@Transactional

- Methods which require being run in transaction should be annotated with @Transactional
- Object is wrapped in a proxy
    - Around advice is used

    - Before entering method a transaction is started
    - If everything goes smoothly, there is commit once the method is finished
    - If method throws RuntimeException, rollback is performed, transaction is not committed
- Can be applied either to method level or class level. If on class level, applies to all the methods declared by implementing interfaces (!). Can be also applied to interface or parent class.
- Can combine method level and class level on the same class. Method level configuration then overrides class level.
- If a test method is annotated @Transactional, it is run in transaction and rolled back after finished
    - Can be both on method and class level
    - Can add @Commit annotation on method level to override @Transactional on test class level

Isolation Levels

- 4 isolation levels available (from least strict to the most strict)
    1. READ_UNCOMMITTED
    2. READ_COMMITTED
    3. REPEATABLE_READ
    4. SERIALIZABLE
- Not all isolation levels may be supported in all databases
- Different databases may implement isolation is slightly different ways

READ_UNCOMMITTED

- @Transactional (isolation=Isolation.READ_UNCOMMITTED)
- The lowest isolation level
- Dirty reads can occur - one transaction may be able to see uncommitted data of other transaction
- May be viable for large transactions or frequently changing data

READ_COMMITTED

- @Transactional (isolation=Isolation.READ_COMMITTED)
- Default isolation strategy for many databases
- Other transactions can see data only after it is properly committed
- Prevents dirty reads

REPEATABLE_READ

- @Transactional (isolation=Isolation.REPEATABLE_READ)
- Prevents non-repeatable reads - when a row is read multiple times in a single transaction, its value is guaranteed to be the same

SERIALIZABLE

- @Transactional (isolation=Isolation.SERIALIZABLE)

- Prevents phantom reads

Transaction Propagation

- Happens when code from one transaction calls another transaction
- Transaction propagation says whether everything should be run in single transaction or nested transactions should be used
- There are 7 levels of propagation
- 2 Basic ones - REQUIRED and REQUIRES_NEW

REQUIRED

- @Transactional(propagation=Propagation.REQUIRED)
- Default value if not specified otherwise
- If there is already transaction, new @Transactional code is run in existing transaction, otherwise a new transaction is created

REQUIRES_NEW

- @Transactional(propagation=Propagation.REQUIRES_NEW)
- If there is no transaction, a new one is created
- If there already is a transaction, it is suspended and a new transaction is created

Rollback

- By default rollback is performed if RuntimeException (or any subtype) is thrown in @Transactional method
- This can be overriden by rollbackFor and noRollbackFor attributes

```
@Transactional(rollbackFor=MyCheckedException.class,
noRollbackFor={FooException.class, BarException.class})
```

JPA with Spring

- JPA - Java Persistence API
- ORM mapping
- Domain objects POJOs
- Common API for object relational mapping
- Several implementations of JPA
  - Hibernate EntityManager
  - EclipseLink - Glassfish
  - Apache OpenJPA - WebLogic and WebSphere
  - Data Nucleus - Google App Engine
- Can be used in spring without app server

Spring JPA Configuration

1. Define EntityManagerFactory bean

2. Define DataSource bean
3. Define TransactionManagement bean
4. Define DAOs
5. Define metadata Mappings on entity beans

EntityManager

- Manages unit of work and involved persistence objects (Persistence context)
- Usually in transaction
- Some API
    - persist(Object object) - adds entity to persistence context - INSERT INTO ...
    - remove(Object object) - removes entity from persistence context - DELETE FROM ...
    - find(Class entity, Object primaryKey) - Find by primary key - SELECT * FROM ... WHERE id=primaryKey
    - Query createQuery(String queryString) - Used to create JPQL query
    - flush() - Current state of entity is written to DB immediately ...

Entity Manager Factory

- Provides access to new app managed Entity Managers
- Thread safe
- Represents single data source / persistence unit

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
    adapter.setShowSql(true);
    adapter.setGenerateDdl(true);
    adapter.setDatabase(Database.HSQL);

    Properties properties = new Properties();
    properties.setProperty("hibernate.format_sql", "true");

    LocalContainerEntityManagerFactoryBean emfb = new
LocalContainerEntityManagerFactoryBean();
    emfb.setDataSource(dataSource());
    emfb.setPackagesToScan("com.example");
    emfb.setJpaProperties(properties);
    emfb.setJpaVendorAdapter(adapter);

    return emfb;
}

@Bean
public DataSource dataSource() {
    ...
}

@Bean
public PlatformTransactionManager transactionManager(EntityManagerFactory
emf) {
    return new JpaTransactionManager(emf);
}
```

Or entityManagerFactory XML Equivalent

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan" value="com.example"/>
    <property name="jpaVendorAdapter">
        <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
            <property name="showSql" value="true"/>
            <property name="generateDdl" value="true"/>
            <property name="database" value="HSQL"/>
        </bean>
    </property>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.format_sql">true</prop>
        </props>
    </property>
</bean>
```

Persistence Unit

- Group of persistent classes - entities
- Has persistence provider
- Has assigned transaction type
- App can have multiple persistence units
- In spring configuration of persistence unit can be either in spring configuration, persistence unit itself or combination

JPA Mapping

- Metadata is required to determine mapping of class fields to DB columns
- Can be done using either annotations or XML (orm.xml)
- Many defaults provided - configuration needed only when differs from the defaults

Can annotate

- Classes
  - Configuration applies to entire class
  - Class annotated @Entity
  - Table name can be overridden using @Table(name="TABLE_NAME"), if not specified class name is used
- Fields
  - Usually mapped to columns
  - By default all considered persistent
  - Can annotate @Transient if should not be persistent
  - Accessed directly though reflection
  - @Id can be used to mark primary key field, can be placed also on getter
  - @Column(name="column_name") can be used to specify column name, if not specified, field name is used as default, can be also placed on getter
  - @Entity and @Id are mandatory, rest is optional
  - @JoinColumn (name="foreignKey")

    - @OneToMany
- Getters
    - Maps to columns like fields
    - Alternative to annotating fields directly

@Embeddable

- One row can be mapped to multiple entities
- Eg. Person, which is one DB table contains personal info as well as address info
- Address can be embedded in Person entity
    - Address as separate entity annotated with @Embeddable
    - In Person class, Address field is annotated with @Embedded
    - Can use @AttributeOverride annotation to specify mappings from columns to fields

JPA Queries

By Primary key

- entityManager.find(Person.class, personId)
- Returns null if no item found
- Uses generics - no cast needed

JPQL Queries

- JPQL - JPA Query Language

```
TypedQuery<Person> query = entityManager.createQuery("select p from Person
p where p.firstname = :firstName", Person.class);
query.setParameter("firstName", "John");
List<Person> persons = query.getResultList();

List<Customer> persons = entityManager.createQuery("select p from Person p
where p.firstname = :firstName", Person.class).setParameter("firstName",
"John").getResultList();

Person person = query.getSingleResult();
```

- Criteria Queries
- Native Queries - JDBC template preferred instead

JPA DAOs

- JPA DAOs in Spring
    - Transactions are handled in Service Layer - Services wrapped in AOP proxies - Spring TransactionInterceptor in invoked
    - TransactionInterceptor delegates to TransactionManager (JpaTransactionManager, JtaTransactionManager)
    - Services call DAOs with no spring dependency
    - Services are injected with AOP-proxied EntityManager

    - Proxied EntityManager makes sure queries are properly joined into active transaction
- DAO implementations have no spring dependencies
- Can use AOP for exception translation - JPA exceptions translated to Spring DataAccessExceptions
- Can use @PersistenceContext annotation to inject EntityManager proxy

```
public class JpaPersonRepository implements PersonRepository {

    private EntityManager entityManager;

    @PersistenceContext
    public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    //Use EntityManager to perform queries

    //If this repository is not annotated as spring component
(@Repository), and explicitly declared in @Configuration
    //file, there is no spring dependency in the DAO layer -
@PersistenceContext is from javax.persistence
}
```

JPA With Spring Data

- Simplifies data access by reducing boilerplate code
- Consists of core project and many subprojects for various data access technologies - JPA, Gemfire, MongoDB, Hadoop, ...

1. Domain classes are annotated as usual
2. Repositories are defined just as interfaces (extending specific Spring interface), which will be dynamically implemented by spring

Spring data Domain Classes

- For JPA nothing changes, classes are annotated as usual - @Entity
- Specific data stores have their own specific annotations
    - MongoDB - @Document
    - Gemfire - @Region
    - Neo4J - @NodeEntity, @GraphId

Spring data Repositories

- Spring searches for all interfaces extending `Repository<DomainObjectType, DomainObjectIdType>`
- Repository is just marker interface and has no method on its own
- Can annotate methods in the interface with @Query("Select p from person p where ...")
- Can extend `CrudRepository` instead of `Repository` - added methods for CRUD
    - Method names generated automatically based on naming convention
    - findBy + Field (+ Operation)

    - `FindByFirstName(String name)`, `findByDateOfBirthGt(Date date)`, ...
    - Operations - Gt, Lt, Ne, Like, Between, ...
- Can extend `PagingAndSortingRepository` - added sorting and paging
- Most Spring data sub-projects have their own variations of `Repository`
    - `JpaRepository` for JPA
- Repositories can be injected by type of their interface

```
public interface PersonRepository extends Repository<Person, Long> {}

@Service
public class PersonService {

    @Autowired
    private PersonRepository personRepository;

    ...
}
```

Repository lookup

- Locations, where spring should look for `Repository` interfaces need to be explicitly defined

```
@Configuration
@EnableJpaRepositories(basePackages="com.example.**.repository")
public class JpaConfig {...}

@Configuration
@EnableGemfireRepositories(basePackages="com.example.**.repository")
public class GemfireConfig {...}

@Configuration
@EnableMongoRepositories(basePackages="com.example.**.repository")
public class MongoDbConfig {...}
<jpa:repositories base-package="com.example.**.repository" />
<gfe:repositories base-package="com.example.**.repository" />
<mongo:repositories base-package="com.example.**.repository" />
```