Spring Professional: Course Summary

Table of Contents

1. Introduction

1.1. What is the Spring Framework?

Spring is an **Open Source**, **Lightweight**, **Container** and **Framework** for building Java enterprise applications.

1.1.1. Lightweight

- Spring applications do not require a Java EE application server (but can be deployed on one).
- Not invasive
  - Does not require you to extend framework classes or implement framework interfaces for most usage
  - You write your code as POJOs
- Small jars

1.1.2. Container

- Spring as a container for your application objects.
  - Objects do not have to worry how to find/connect to each other.
- Spring instantiates and dependency injects your objects
  - Serves as a lifecycle manager

1.1.3. Framework

- Spring provides framework classes to simplify working with lower-level technologies e.g.,
  - JDBC, JMS, AMQP, Transactions, ORM / JPA, NoSQL, Security, Web, Tasks, Scheduling, Mail, Files, XML/JSON Marshalling, Remoting, REST services, SOAP services, Mobile, Social, …
- Framework classes for abstraction of lower-level technologies.

1.2. Spring is a Container

Spring provides support for assembling such an application system from its parts.

- Parts do not worry about finding each other.
- Any part can easily be swapped out.
- Parts are POJOs.
- Interface usage allows swapping backing implementations.

1.3. What is Spring Used For?

Spring provides infrastructural support for developing enterprise Java applications. It deals with the plumbing so you can focus on solving the domain problem. It provides:

- Web Interfaces
- Messaging
- Persistence
- Batch
- Integration

2. Dependency Injection - Part 1

Introducing the Spring Application Context and Spring's Java Configuration capability.

2.1. Spring quick start

1. Application classes get injected into Spring Application Context
2. Configuration Instructions get injected into Application Context.
3. Spring Application Context creates Fully configured App.

Application classes.

```
public class TransferServiceImpl implements TransferService {
    public TransferServiceImpl(AccountRepository ar) { ①
        this.accountRepository = ar; }
    }
    ...
}
```

① Needed to perform money transfers between accounts.

Configuration Instructions.

```
@Configuration
public class ApplicationConfig {
    @Bean
    public TransferService transferService() {
        return new transferServiceImpl(accountRepository());
    }

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

    @Bean
    public DataSource dataSource() {
        return new new BasicDataSource());
    }
}
```

Creating and using the app.

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

// Look up the application service interface
TransferService service = (TransferService) context.getBean("transferService"); ①

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

① Bean ID based on method name.

Accessing a Bean.

```
ApplicationContext context = SpringApplication.run(...); ①

// Classic way: cast is needed
TransferService ts1 = (TransferService) context.getBean("transferService");

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

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

① Inject configuration classes into the run method e.g. `AppConfig.class`

Summary

- Spring manages the lifecycle of the application
  - All beans are fully initialized before use
- Beans are always created in the right order
  - Based on their dependencies
- Each bean is bound to a unique id
  - The id reflects the service or role the bean provides to clients
  - Bean id should not contain implementation details

2.2. Creating an application context

Spring application contexts can be **bootstrapped in any environment**, including JUnit, a Web application and Standalone applications.

Using an Application Context Inside a JUnit System Test.

```
class TransferServiceTests {
    private TransferService service;

    @Before public void setUp() { ①
        // Create the application from the configuration
        ApplicationContext context = SpringApplication.run( ApplicationConfig.class )
        // Look up the application service interface
        service = context.getBean(TransferService.class);
    }

    @Test public void moneyTransfer() { ②
        Confirmation receipt = service.transfer(new MonetaryAmount(□300.00□), □1□,
□2□));
        Assert.assertEquals(receipt.getNewBalance(), □500.00□);
    }
}
```

① Bootstraps the system to test.

② Tests the system.

2.3. Multiple Configuration Files

2.3.1. General

Devide your configuration into **multiple configuration** classes using `@Import`. This defines a **single Application Context** with beans sourced from **mutliple files**.

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

@Configuration
public class InfrastructureConfig { ... }

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

Best Practice

> Separate out “application” beans e.g., services from “infrastructure” beans e.g., datasources. Infrastructure often changes between environments.

2.3.2. Referencing beans from another file

Either use `@Autowired` to reference bean defined in a separate configuration file or define Define `@Bean` method parameters.

Autowired.

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

    @Autowired DataSource dataSource;

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

@Configuration
public class ApplicationConfig {

    @Bean
    public DataSource dataSource() {
        return ...;
    }
}
```

Bean.

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

    @Bean
    public AccountRepository accountRepository( DataSource dataSource ) { ①
        return new JdbcAccountRepository( dataSource );
    }
}

@Configuration
public class InfrastructureConfig {

    @Bean
    public DataSource dataSource() {
        return ...;
    }
}
```

① Spring matches this by type.

WARNING	It is not illegal to define the same bean more than once. **You get the last bean Spring sees** defined.

2.4. Bean scope

singleton *(default)*

A single instance is used.

prototype

A new instance is created each time the bean is referenced.

session *(web environment)*

A new instance is created once per user session.

request *(web environment)*

A new instance is created once per request.

custom scope name *(advanced feature)*

You define your own rules and a new scope name.

2.5. Summary

- Your object is handed 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

3. Dependency Injection - Part 2

3.1. External Properties

Bad practice to use hard coded properties e.g., db username. Instead externalize properties. Either use the **Environment object** or **@Value** annotation.

3.1.1. Environment Object

- Environment object used to obtain properties from runtime environment
- Properties from many sources:
  - JVM System Properties
  - Java Properties Files

- Servlet Context Parameters
- System Environment Variables
- JNDI

Using the Environment object to get the properties.

```
@Autowired public Environment

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

3.1.2. @Value

Using the `@Value` annotation to get the properties.

```
@Configuration
public class ApplicationConfig {

    @Bean
    public DataSource dataSource(
            @Value("${db.driver}") String dbDriver,
            @Value("${db.url}") String dbUrl,
            @Value("${db.user}") String dbUser,
            @Value("${db.password}") String dbPassword) {
        DataSource ds = new BasicDataSource();
        ds.setDriverClassName( dbDriver);
        ds.setUrl( dbUrl);
        ds.setUser( dbUser);
        ds.setPassword( dbPassword ));
        return ds;
    }
}
```

3.1.3. Property Sources

- Environment obtains values from “property sources”
  - System properties & Environment variables populated automatically
  - Use `@PropertySources` to contribute additional properties
  - Available resource prefixes: `classpath:` `file:` `http:`

```
@Configuration
@PropertySource ( classpath:/com/organization/config/app.properties )
public class ApplicationConfig {
    ...
    @Bean
    public static PropertySourcesPlaceholderConfigurer
propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer(); ①
    }
}
```

① `@PropertySource` ignored unless this bean declared.

Loading

- Property sources are loaded by a dedicated Spring bean
  - The PropertySourcesPlaceholderConfigurer

WARNING	This is a static bean. Such beans are created first. It ensures property-sources are read before any `@Configuration` bean using `@Value` is initialized.

Placeholders

- `${…}` placeholders in a @PropertySource are resolved against existing properties.
  - Such as System properties & Environment variables.

```
@PropertySource ( classpath:/com/acme/config/app-${ENV}.properties )
```

1. app-dev.properties

```
db.user=transfer-app
db.password=secret45
```

1. app-prod.properties

```
db.user=transfer-app
db.password=secret46
```

3.2. Profiles

Grouping

Beans can be grouped into Profiles.

- Profiles can **represent purpose**: “web”,“offline”
- **Or environment**: “dev”, “qa”, “uat”, “prod”

- Beans **included/excluded** based on profile membership
  - Beans with **no profile are always available**.

Simply use the `@Profile("sample_profile")` annotation either on:

- `@Configuration` classes: All beans in that class belong to that profile.
- `@Bean` methods: Only this bean belongs to that profile.

```
@Bean(name="dataSource") ①
@Profile("dev")②
public DataSource dataSourceForDev() { ... }

@Bean(name="dataSource") ①
@Profile("prod") ②
public DataSource dataSourceForProd() { ... }
```

① Same bean name.

② Different profiles.

Activate Profiles

Profiles must be activated at run-time.

- Integration Test: Use @ActiveProfiles
- System property: `-Dspring.profiles.active=dev,jpa`
- In `web.xml`:

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>jpa,web</param-value>
</context-param>
```

- Programmatically on ApplicationContext: Simply set `spring.profiles.active` system property before instantiating `ApplicationContext`

3.3. Spring Expression Language (SPEL)

3.3.1. @Value

Usage

```
@Value("#{ systemProperties['user.region'] }")
String region; ①

@Bean public TaxCalculator taxCalculator1() {
    TaxCalculator tc = new TaxCalculator();
    tc.setDefaultLocale(region); ①
    return tc;
}

@Bean public TaxCalculator taxCalculator2(
            @Value("#{ systemProperties['user.region'] }") String region) { ②
    TaxCalculator tc = return new TaxCalculator();
    tc.setDefaultLocale(region);
    return tc;
}
```

① On field.

② Or method argument.

Accessing beans.

```
class StrategyBean {
    private KeyGenerator gen = new UuidGenerator();
    public KeyGenerator getKeyGenerator() {
        return gen;
    }
}

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

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

- EL Attributes can be:
  - Spring beans (like strategyBean)
  - Implicit references
    - systemProperties and systemEnvironment available by default
- SpEL allows to create custom functions and references

3.4. Proxying

Spring works with singletons.

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

@Bean
public TransferService transferService() {
    TransferServiceImpl service = new TransferServiceImpl();
    service.setAccountRepository(accountRepository()); ①
    return service;
}

@Bean
public AccountService accountService() {
    return new AccountServiceImpl( accountRepository()); ①
}
```

① Two invocations but spring makes it a single instance.

Reason

- At startup time, a child class is created
  - For each bean, an instance is cached in the child class
  - Child class only calls super at first instantiation

Child class is the entry point. Java Configuration uses cglib for inheritance-based proxies.

```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {
    public AccountRepository accountRepository() {
        // if bean is in the applicationContext, then return bean
        // else call super.accountRepository(), store bean in context, return bean
    }

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

3.5. Summary

- Property values are easily externalized using Spring's Environment abstraction
- Profiles are used to group sets of beans
- Spring Expression Language

- Spring proxies your `@Configuration` classes to allow for scope control.

4. Annotations

Annotations for Dependency Injection and Interception - Component scanning and auto-injection.

4.1. Annotation-based Configuration

4.1.1. Before

- Separation of concerns
- Java-based dependency injection.

Configuration is external to bean-class.

```
@Configuration
public class TransferModuleConfig {
    @Bean public TransferService transService() {
        TransferServiceImpl service = new TransferServiceImpl();
        service.setAccountRepository(accountRepository());
        return service;
    }
}
```

4.1.2. After

Annotation-based configuration within bean-class

```
@Component
public class TransferServiceImpl implements TransferService { ①
    @Autowired
    public TransferServiceImpl(AccountRepository repo) {
        this.accountRepository = repo; ②
    }
}

@Configuration @ComponentScan ( □com.bank□ ) ③
public class AnnotationConfig {
    // no bean definition needed anymore
}
```

① Bean id derived from class name.

② Annotations embedded with POJO.

③ Find `@Component` classes within designated (sub)packages.

4.1.3. @Autowired

Where a **unique dependency of correct type must exist**. Injection via. **Constructor**, **Method** and **Field** (even `private`, but hard to unit test). Injection can either be **required (default) or not**.

Use required attribute to override default behaviour.

```
@Autowired(required=false)
public void setAccountRepository(AccountRepository a) { ①
    // Use required attribute to override default behavior
    this.accountRepository = a;
}
```

① Only inject if dependency exists.

Optional<T> (Java 8 only). Was introduced to reduce null pointer errors.

```
@Autowired
Optional<AccountService> accountService;

public void doSomething() {
    accountService.ifPresent(s -> {...});
}
```

4.1.4. Constructor vs Setter Dependency Injection

Constructor	Setter
Mandatory dependencies	Optional / changeable dependencies
Immutable dependencies	Circular dependencies
Concise (pass several params at once)	Inherited automatically

IMPORTANT Be consistent.

4.1.5. Disambiguation (@Qualifier)

Prevent disambiguities. `NoSuchBeanDefinitionException` is thrown at **start-up** in case no unique bean is defined. Use the `@Qualifier` annotation and explicit bean ids to refer to these. Also available with **method injection and field injection**.

Autowired resolution rules

- Look for unique bean of required type
- Use `@Qualifier` if supplied
- Try to find a matching bean by name

Use of `@Qualifier` annotation.

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl(@Qualifier("jdbcAccountRepository")
            AccountRepository accountRepository) {...}
    ...
}

@Component("jdbcAccountRepository")
public class JdbcAccountRepository implements AccountRepository {...}
```

4.1.6. @Value

Use `$` variables or SpEL.

Constructor injection

```
@Autowired
public TransferServiceImpl(@Value("${daily.limit}") int max) {...}
```

Method injection

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

Field injection

```
@Value("#{systemEnvironment['DAILY_LIMIT']}")
private int maxTransfersPerDay;
```

Provide **fall-back** values.

Use colon with `$` variables.

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

Use ?: (Elvis operator) for SpEL.

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

4.1.7. Component Names

When not specified

- Names are auto-generated
  - De-capitalized non-qualified classname by default
  - But will pick up implementation details from classname
- Recommendation: never rely on generated names!

When specified

- Allow disambiguation when 2 bean classes implement the same interface

IMPORANT: Avoid using qualifiers when possible. Usually rare to have 2 beans of same type in ApplicationContext.

4.2. Best practices for component-scanning

Components are scanned at startup

- Jar dependencies also scanned!
- Could result in slower startup time if too many files scanned
  - Especially for large applications
  - A few seconds slower in worst case

Use the most specific packages when using `@ComponentScan`.

```
@ComponentScan({"com.bank.app.repository", "com.bank.app.service",
"com.bank.app.controller"})
```

4.3. Java Config versus annotations

4.3.1. Annotation

Nice for frequently changing beans.

Pros

- Single place to edit (just the class)
- Allows for very rapid development

Cons

- Configuration spread across your code base
  - Harder to debug/maintain
- Only works for your own code
- Merges configuration and code (bad sep. of concerns)

Annotation.

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

4.3.2. Java Configuration

Pros

- Is centralized in one (or a few) places
- Write any Java code you need
- Strong type checking enforced by compiler (and IDE)
- Can be used for all classes (not just your own)

Cons

- More verbose than annotations

Java Configuration.

```
@Bean
@Scope("prototype")
@Profile("dev")
@Lazy(false)
public TransferService transferService() {
    return new TransferServiceImpl(accountRepository());
}
```

4.4. @PostConstruct and @PreDestroy

Add behavior at startup with `@PostConstruct` and shutdown with `@PreDestroy` method annotations.

IMPORTANT	Annotated methods can have any visibility but must take no parameters and only return void.

Example.

```
public class JdbcAccountRepository {
    @PostConstruct ①
    void populateCache(){...}

    @PreDestroy ②
    void clearCache(){...}
}
```

① Called after startup and after all dependencies got injected.

② Called at shutdown prior to destroying the bean instance.

Beans can be created in the normal way

- Returned from @Bean methods
- Found and created by the component-scanner
- Spring invokes them automatically

These are not Spring annotations

- Defined by JSR-250, part of Java since Java 6
- In javax.annotation package
- Also supported by EJB3

PostConstruct

- Called after setter methods got called
- Start → Constructor called → Setter(s) called → @PostConstruct called

PreDestroy

- Called when an ApplicationContext is closed
  - If application (JVM) exits normally
  - Useful for releasing resources & 'cleaning up'
  - Not called for prototype beans

NOTE `ApplicationContext.close()` triggers the `@PreDestroy`.

Bean alternative

- @Bean properties for classes you didn't write and can't annotate
- @PostConstruct/@PreDestroy for your own classes

Bean Alternative: `initMethod` *and* `destroyMethod`.

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

4.5. Stereotypes and meta annotations

Component scanning also checks for annotations that are themselves annotated with @Component (stereotype annotations).

@Component

@Service

Service classes

@Repository

Data access classes

@Controller

Web classes (Spring Mvc)

@Configuration

Java Configuration

Meta Annotations

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

Meta Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Service @Transactional(timeout=60)
public @interface MyTransactionalService {
    String value() default "";
}
```

4.6. @Resource

Identifies dependencies by name (Spring Bean name), not by type. If no name is provided, it first checks the property/field name and the falls back to type checking. It supports only setter and field injection.

@Autowired

type then name

@Resource

name then type

4.7. Standard Annotations (JSR 330)

Also known as `@Inject`. Is a subset of the @Autowired annotation. `@Named` annotations are also scanned by the component scan.

4.8. Summary

- Spring's configuration directives can be written using Java, annotations, or XML (next)
- You can mix and match Java, annotations, and XML as you please
- Autowiring with @Component allows for almost empty Java configuration files

5. Dependency Injection Using XML

5.1. Writing bean definitions in XML

Example

```
<beans profile="prod"> ①
    <bean id="transferService1" class="com.acme.TransferServiceImpl">
        <property name="repository" ref="accountRepository" /> ②
    </bean>

    <bean id="transferService2" class="com.acme.TransferServiceImpl">
        <constructor-arg ref="accountRepository"/> ③
        <constructor-arg ref="customerRepository"/> ③
    </bean>

    <bean id="service1" class="com.acme.ServiceImpl">
        <property name="stringProperty" value="foo" /> ④
    </bean>

    <import resource="db-config.xml" /> ⑤

    <bean id="service2" class="com.acme.ServiceImpl"
            init-method="setup" destroy-method="clear" /> ⑥

    <bean id="accountService1" class="com.acme.ServiceImpl" scope="prototype" /> ⑦

    <bean id="accountService2" class="com.acme.ServiceImpl" lazy-init="true"> ⑧
    ...
</beans>
```

① @Profile

② Setter injection.

③ Constructor injection. Use `index` if type is ambiguous.

④ Scalar values.

⑤ Import other XML files.

⑥ @PostConstruct @PreDestroy

⑦ @Scope

⑧ @Lazy("true")

Both, setter and constructor injection is combinable. When using scalar values, spring automatically converts types accordingly (Numeric types, BigDecimal, boolean, Date, Locale, Resource).

5.2. Creating an application context

Use a Java Configuration class, then use `@ImportResource` to specify XML files. When using `SpringApplication.run(MainConfig.class);` just do:

```
@Configuration
@ImportResource( { ①
        classpath:com/acme/application-config.xml,
        file:C:/Users/alex/application-config.xml
})
@Import(DatabaseConfig.class) ②
public class MainConfig { ... }
```

① Multiple files allowed (with prefix).

② Also combine with @Configuration imports.

5.3. Controlling Bean Behavior

Instead of @PostConstruct and @PreDestroy use `init-method` and `destroy-method`. The profile in a `<beans>` tag applies to all beans in the element. To use multiple profiles use nested `<beans>`

IMPORTANT	Same rules: method can have any visibility, must take no parameters, must return void. Called after dependency injection.

Using multiple profiles.

```
<beans xmlns=http://www.springframework.org/schema/beans ...>
    <bean id="rewardNetwork" ... /> <!-- Available to all profiles -->
    <beans profile="dev"> ... </beans>
    <beans profile="prod"> ... </beans>
</beans>
```

5.4. Namespaces

The **default namespace** in a Spring configuration file is typically the **"beans"** namespace. Other namespaces are available: aop (Aspect Oriented Programming), tx (transactions), util, jms, context.

They allow **hiding of actual bean definitions**. Use the **STS XML editor namespaces tab** to prevent typos. **Do not use Schema Version Numbers** to ease the migration to the next spring version.

Namespace example.

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

<context:property-placeholder location="db-config.properties" /> ①

<bean class="org.springframework...PropertySourcesPlaceholderConfigurer"> ②
    <property name="location" value="db-config.properties"/>
</bean>

<bean id="dataSource" class="com.oracle.jdbc.pool.OracleDataSource">  ③
    <property name="URL" value="${dbUrl}" />
    <property name="user" value="${dbUserName}" />
</bean>
```

① This instead of...

② ...this

③ Use it here

Equivalent to `@PropertySource ( "classpath:/com/acme/config/app-${ENV}.properties" )`

```
<import resource="classpath:config/${current.env}-config.xml"/>

<context:property-placeholder properties-ref="configProps"/>

<beans profile="dev">
    <util:properties id="configProps" location="config/app-dev.properties">
</beans>

<beans profile="prod">
    <util:properties id="configProps" location="config/app-prod.properties">
</beans>
```

Equivalent to `@ComponentScan( { "com.acme.app.repository", com.acme.app.service", "com.acme.app.controller" })`

```
<context:component-scan base-package="com.acme.app.repository, com.acme.app.service,
com.acme.app.controller" />
```

6. Bean Lifecycle

Using Bean Pre- and Post-Processors.

6.1. Introduction

Initialization → Use → Destruction

Initialization

- Prepares for use
- Application services
  - Are created
  - Configured
  - May allocate system resources
- Application is not usable until this phase is complete

Use

- Used by clients
- Application services
  - Process client requests
  - Carry out application behaviors
- 99.99% of the time is spent in this phase

Destruction

- Shuts down
- Application services
  - Release any systemresources
  - Are eligible for garbage collection

6.2. The initialization phase

When `SpringApplication.run(⋯)` returns the initialization phase completes.

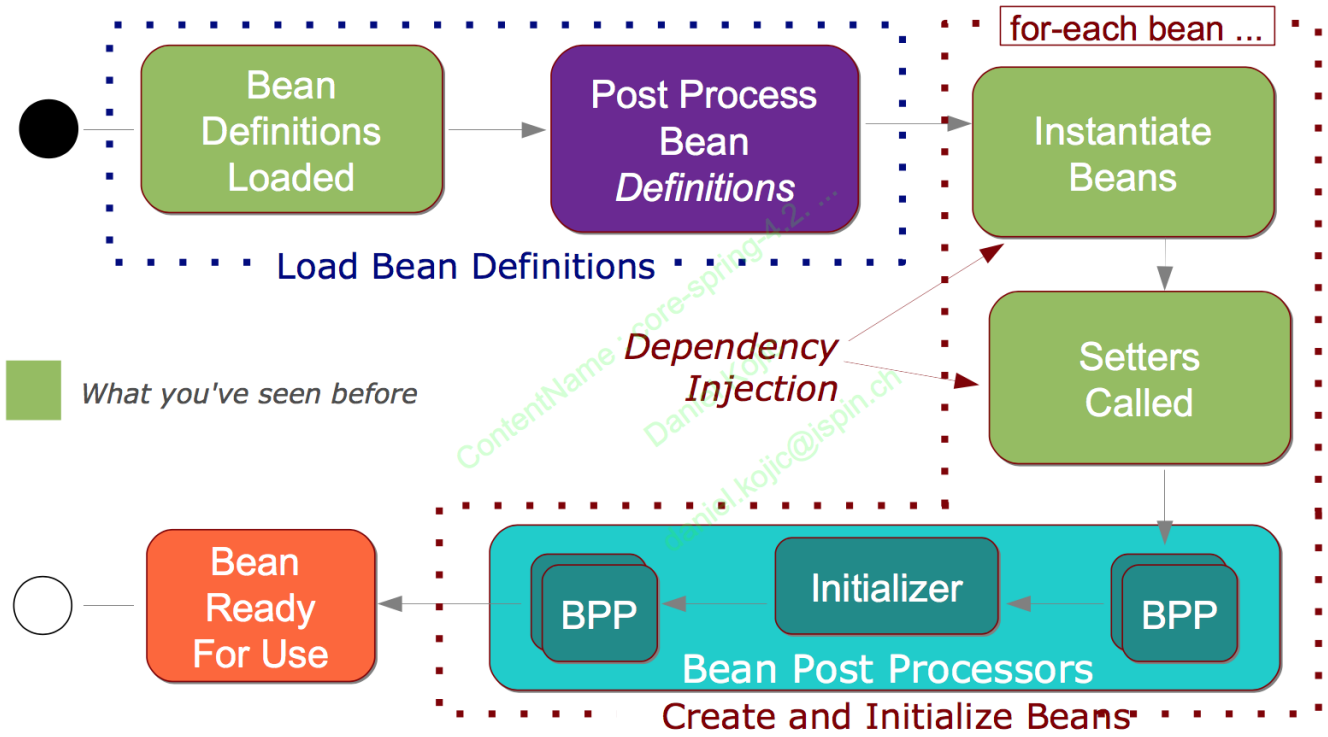


Figure 1. Bean initialization steps

6.2.1. Load Bean Definitions

The @Configuration classes are processed: **@Components are scanned and/or XML files parsed**. Bean definitions added to BeanFactory under its id. Then `BeanFactoryPostProcessor` beans get invoked. These may change any bean definition.

IMPORTANT

BeanFactoryPostProcessor Extension Point

- Applies transformations to bean definitions
  - Before objects are actually created
- Several useful implementations provided in Spring
  - Reading properties, registering a custom scope ...
- You can write your own.
  - Implement `BeanFactoryPostProcessor` interface

6.2.2. Initialize Bean Instances

Each bean is **eagerly instantiated** by default in **right order** with its **dependencies injected**. After dependency injection each bean is post-processed (further configured and initialized. After post processing the bean is fully initialized and ready for use.

There are **two types of bean post processors**: Initializers & the rest. Initializers call init methods e.g., @PostConstruct and init-method. All others can be used for additional configuration and may run **before or after the initialize step**.

IMPORTANT

BeanPostProcessor Extension Point

- An important extension point in Spring
  - Can modify bean instances in any way
  - Powerful enabling feature
- Spring provides several implementations
  - You can write your own (not common)
  - Must implement the `BeanPostProcessor` interface

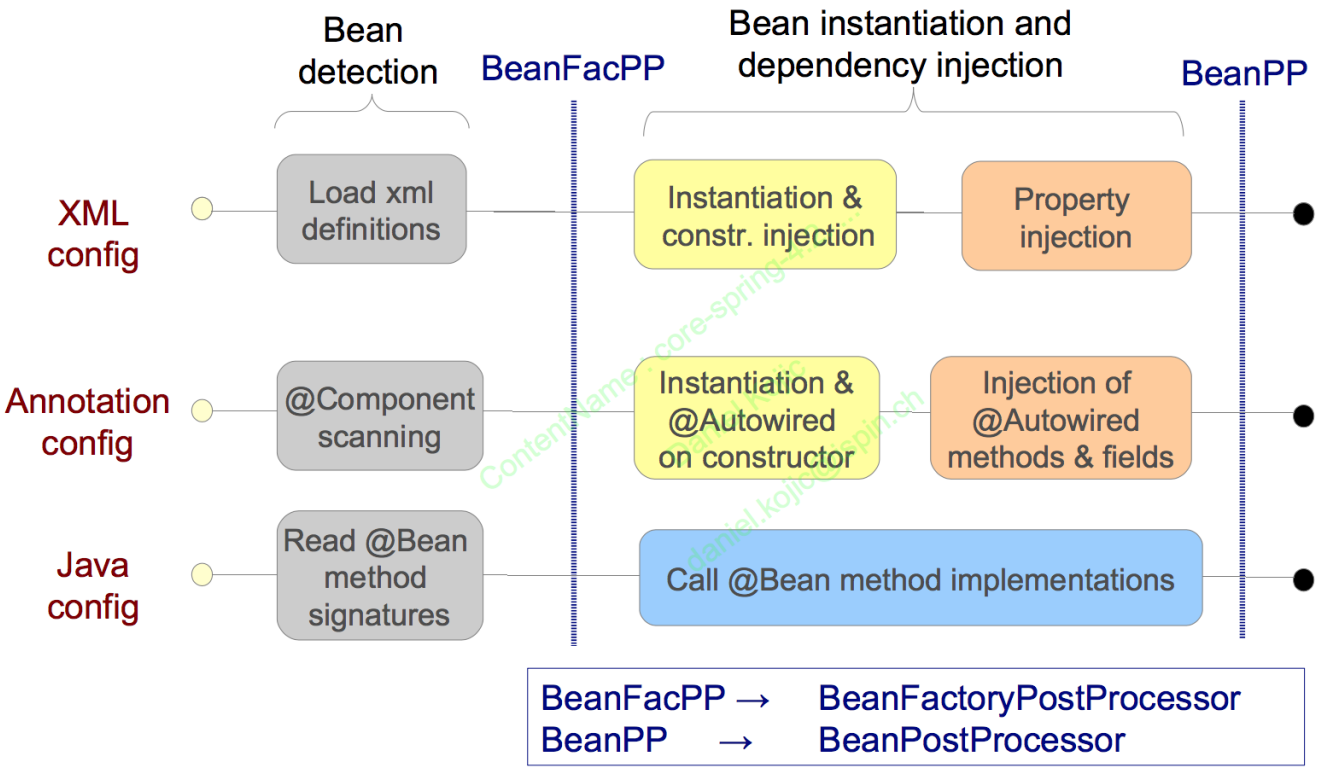
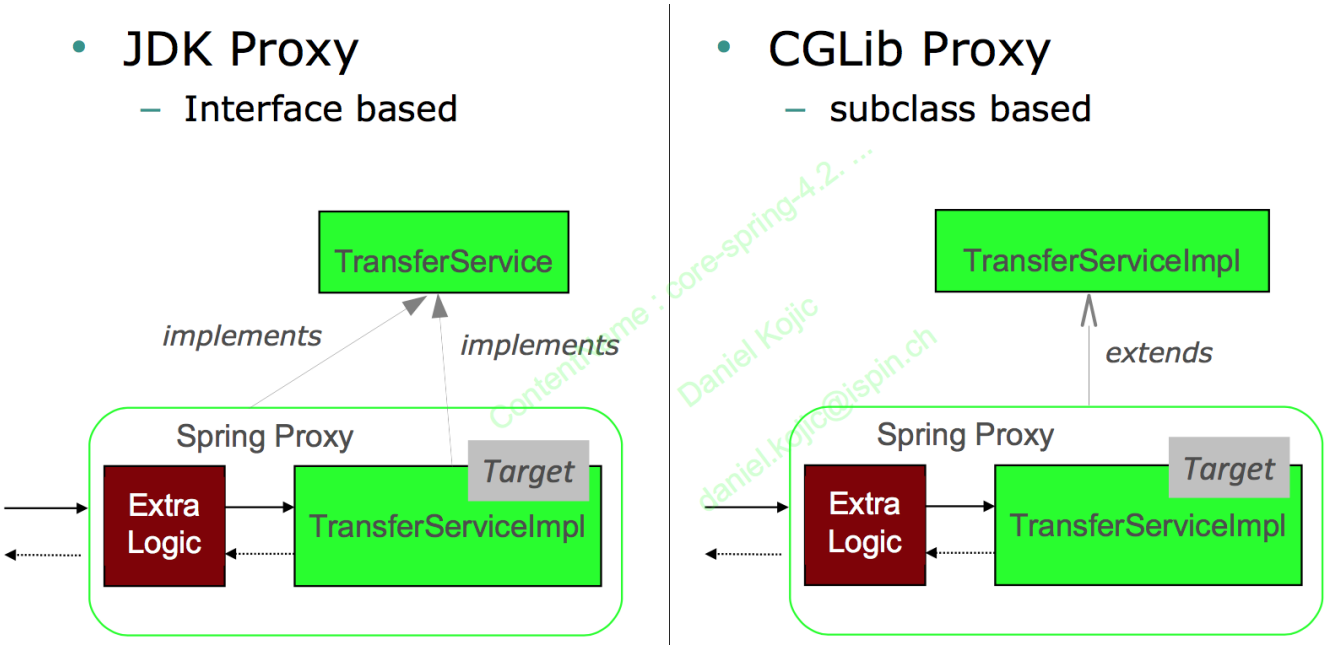


Figure 2. Configuration lifecycle.

6.3. The use phase

When you invoke a bean obtained from the context the application is used. Beans get **wrapped in dynamic proxys** which are created in the init phase by dedicated BeanPostProcessors. Spring will create two kinds of proxys: **JDK** and **CGLib** Proxies.



JDK Proxy (Interface based)

(Recommended)

- Also called dynamic proxies
- API is built into the JDK
- Requirements: Java interface(s)
- All interfaces proxied

CGLib Proxy (Subclass based)

- NOT built into JDK
- Included in Spring jars
- Used when interface not available
- Cannot be applied to final classes or methods

6.4. The destruction phase

When you close a context the destruction phase completes. Bean instances get destroyed by invoking their clean-up/destroy methods if the JVM exits normally. After closing, the context itself will no longer be usable.

7. Testing Spring Applications

Testing in General, Spring and JUnit, Profiles, Database Testing

7.1. Test Driven Development

TDD is about writing automated tests that verify code actually works and driving development with well defined requirements in the form of tests. Tested code allows faste and more confident

refactoring → essential to agile development. If your code is hard to test then the design should be reconsidered.

The cost to fix a bug grows exponentially in proportion to the time before it is discovered.

7.2. Unit testing without Spring

- Tests one unit of functionality
- Keeps dependencies minimal
- Isolated from the environment (including Spring)
- Uses simplified alternatives for dependencies
  - Stubs and/or Mocks

7.3. Integration Testing with Spring

Integration testing **tests the interaction of multiple units working together**. All should work individually (unit tests showed this). They test application classes in **context of their surrounding infrastructure**:

- Out-of-container testing, no need to run up full JEE system
- Infrastructure may be "scaled down"
  - Use Apache DBCP connection pool instead of container- provider pool obtained through JNDI
  - Use ActiveMQ to save expensive commercial JMS licenses

Production Mode Flow

1. **caller** (Injected by Spring)
2. **TransferServiceImpl** (Injected by Spring)
3. **JpaAccountRepo**
4. **Production DB**

Integration Test Flow

1. **TransferServiceTest** (Injected by Spring)
2. **TransferServiceImpl** (Injected by Spring)
3. **JpaAccountRepo**
4. **Test DB**

Spring's integration test support comes packaged as a separate module **spring-test.jar**. It consists of several JUnit test support classes. The central support class is `SpringJUnit4ClassRunner` (caches a shared ApplicationContext across test methods). The **ApplicationContext is instantiated only once** for all tests that **use the same set of config files**. (even across test classes)

Spring JUnit runner example.

```
@RunWith(SpringJUnit4ClassRunner.class) ①
@ContextConfiguration(classes=SystemTestConfig.class) ②
@ContextConfiguration("classpath:com/acme/system-test-config.xml") ③
public final class TransferServiceTests {

    @Autowired ④
    private TransferService transferService;

    @Test
    public void successfulTransfer(){
        TransferConfirmation conf = transferService.transfer(...); ⑤
    }

    @Configuration ⑥
    public static class TestConfiguration {
        @Bean public DataSource dataSource() { ... }
    }
}
```

① Run with spring support.

② Either point to config file (loads TransferServiceTests-context.xm if empty)…

③ … or to XML config file.

④ Inject Bean to test.

⑤ Test the system as usual.

⑥ Alternative to specifying external classes. Must be `static`.

TIP Annotate test method with `@DirtiesContext` to force recreation of the cached ApplicationContext if method changes the contained beans.

Benefits of Testing with Spring

- No need to deploy to an external container to test application functionality
  - Run everything quickly inside your IDE
- Allows reuse of your configuration between test and production environments
  - Application configuration logic is typically reused
  - Infrastructure configuration is environment-specific (DataSources, JMS Queues)

7.4. Testing with Profiles

Aktivate one or more profiles with `@ActiveProfiles` inside the test class. Beans associated with that profile are instantiated. Also beans not associated with any profile. Works with Java config (with @Profile(…) annotated @Configuration) and Annotations (with @Profile(…) annotated @Component) and XML configuration (profile attribute in beans tag).

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=DevConfig.class)
@ActiveProfiles( { "jdbc", "dev" } )
public class TransferServiceTest {...}
```

7.5. Testing with Databases

Integration testing against SQL database is common. **In-memory databases** useful for this kind of testing (no prior install). **Populate DB** before test runs with the `@Sql` annotation.

@Sql options

- executionPhase
  - BEFORE_TEST_METHOD
  - AFTER_TEST_METHOD

@Sql config

- errorMode
  - `FAIL_ON_ERROR`
  - `CONTINUE_ON_ERROR`
  - `IGNORE_FAILED_DROPS`
  - `DEFAULT` (whatever @Sql defines at class level, otherwise FAIL_ON_ERROR)

commentPrefix & separator

- Syntax control

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(...)
@Sql({ /testfiles/schema.sql, /testfiles/general-data.sql } )  ①
public final class MainTests {

    @Test
    @Sql ②
    public void success(){...}

    @Test
    @Sql ( /testfiles/error.sql ) ③
    @Sql ( scripts=/testfiles/cleanup.sql, ④
            executionPhase=Sql.ExecutionPhase.AFTER_TEST_METHOD )
    public void transferError() {...}
}
```

① Run these scripts before each @Test method.

② Run script named (by default) MainTests.success.sql in same package.

③ Run before @Test method.

④ Run after @Test method.

7.6. Summary

- Testing is an essential part of any development
- Unit testing tests a class in isolation
  - External dependencies should be minimized
  - You don't need Spring to unit test
  - Consider creating stubs or mocks to unit test
- Integration testing tests the interaction of multiple units working together
  - Spring provides good integration testing support
  - Profiles for different test & deployment configurations
  - Built-in support for testing with Databases

8. Aspect Oriented Programming (AOP)

Using and Implementing Spring Proxies.

8.1. What Problem Does AOP Solve?

Aspect-Oriented Programming (AOP) enables **modularization** of **cross-cutting concerns**. An example requirement could be:

> Perform a role-based security check before **every** application method.

Cross-Cutting Concern

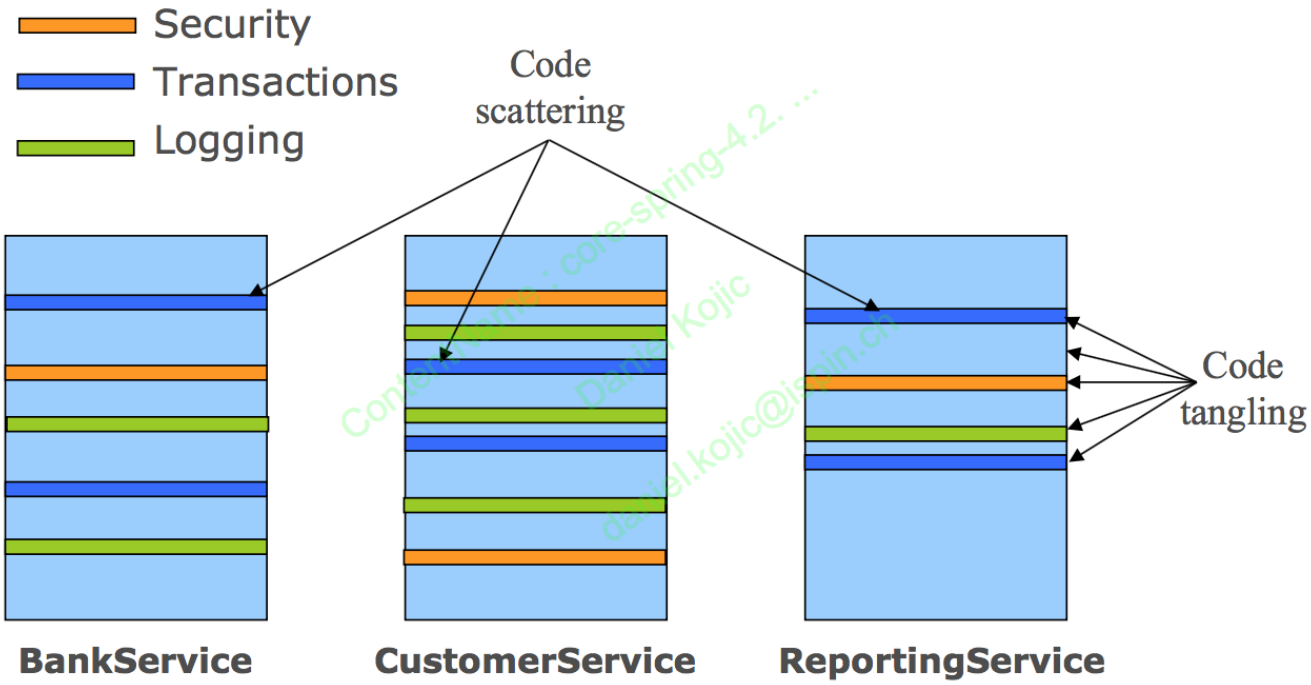
Generic functionality that is needed in many places in your application e.g., Logging and Tracing, Transaction Management, Security, Caching, Error Handling, Performance Monitoring, Custom Business Rules.

Failing to modularize cross-cutting concerns leads to two things: **Code tangling** (coupling of concerns) and **Code scattering** (the same concern spread across modules ⇒ code duplication).

Tangling example.

```
public class RewardNetworkImpl implements RewardNetwork {

    public RewardConfirmation rewardAccountFor(Dining dining) {
        if (!hasPermission(SecurityContext.getPrincipal()) {
            throw new AccessDeniedException(); ①
        }

        Account a = accountRepository.findByCreditCard(...); ①
        Restaurant r = restaurantRepository.findByMerchantNumber(...)
        MonetaryAmount amt = r.calculateBenefitFor(account, dining);
        ...
    }
}
```

① Mixed concerns.



8.1.1. How AOP works

Implement your mainline application logic

- Focusing on the core problem

Write aspects to implement your cross-cutting concerns

- Spring provides many aspects out-of-the-box

Weave your aspects into the application

- Adding the cross-cutting behaviours to the right places

8.1.2. AspectJ vs. Spring AOP

AspectJ

A full-blown Aspect Oriented Programming language. Uses byte code modification for aspect weaving.

Spring AOP

Java-based AOP framework with AspectJ integration. Uses dynamic proxies for aspect weaving. Focuses on using AOP to solve enterprise problems.

8.2. Core AOP Concepts

Join Point

A point in the execution of a program such as a method call or exception thrown.

Pointcut

An expression that selects one or more Join Points.

Advice

Code to be executed at each selected Join Point.

Aspect

A module that encapsulates pointcuts and advice.

Weaving

Technique by which aspects are combined with main code.

8.3. Quick Start

Lets consider this simple requirement:

> Log a message every time a property is about to change.

Define the aspect.

```
@Aspect
@Component
public class PropertyChangeTracker {

    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
    public void trackChange(JointPoint point) {  ①
        String name = point.getSignature().getName();
        Object newValue = point.getArgs()[0];
        logger.info(name + " about to change to " + newValue + " on " +
point.getTarget());
    }
}
```

① Context information on the intercepted point.

Then configure the aspect as a Bean.

```
@Configuration
@EnableAspectJAutoProxy ①
@ComponentScan(basePackages="com.example")
public class AspectConfig {...}
```

① Configures Spring to apply @Aspect to you beans. XML: `<aop:aspectj-autoproxy />`

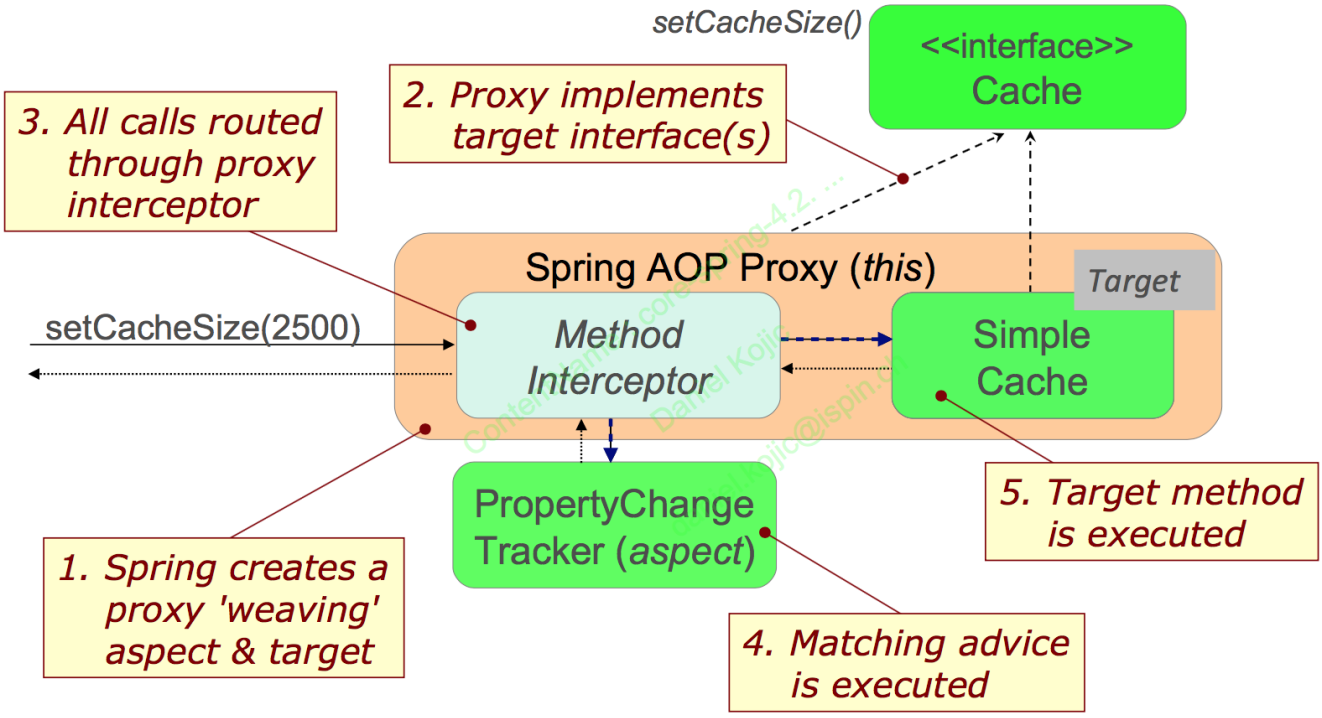
Include aspect configuration.

```
@Configuration
@Import(AspectConfig.class) // here
public class MainConfig{

    @Bean
    public Cache cacheA() { return new SimpleCache("cacheA"); }

    @Bean
    public Cache cacheB() { return new SimpleCache("cacheB"); }

    @Bean
    public Cache cacheC() { return new SimpleCache("cacheC"); }
}
```

And then test it.

```
@Autowired
@Qualifier("cacheA");
private Cache cache;
...
cache.setCacheSize(2500); ①
```

① `INFO: setCacheSize about to change to 2500 on cacheA`



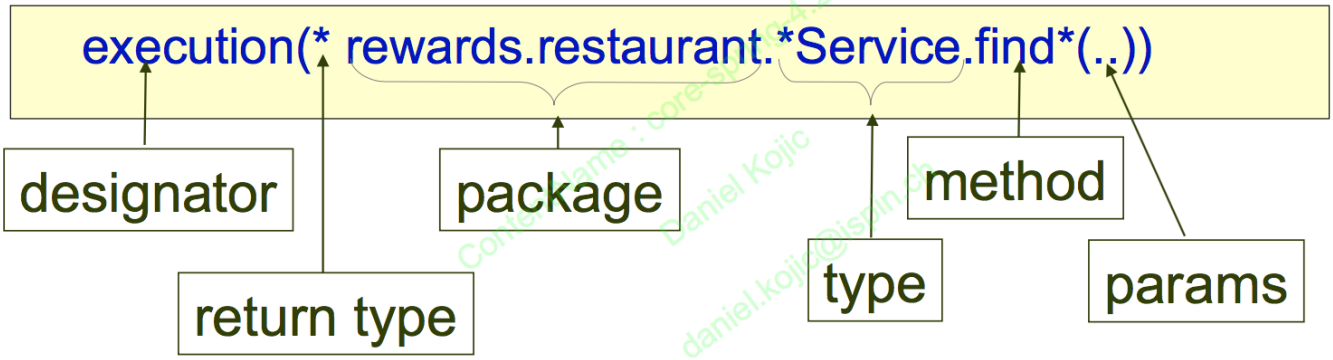
8.4. Defining Pointcuts

Spring AOP uses a **subset** of AspectJ's pointcut expression language for selecting where to apply advice.

8.4.1. Common Pointcut Designator

- execution(<method pattern>)
  - The method must match the pattern
- Can chain together to create composite pointcuts
  - && (and), || (or), ! (not)
- Method Pattern
  - `[Modifiers] ReturnType [ClassType] MethodName ([Arguments]) [throws ExceptionType]`

8.4.2. Examples



`execution(void send*(String))`

Any method starting with send that takes a single String parameter and has a void return type.

`execution(* send(*))`

Any method named send that takes a single parameter

`execution(* send(int, ..))`

Any method named send whose first parameter is an int (the ".." signifies 0 or more parameters may follow)

`execution(void example.MessageServiceImpl.*(..))`

Any visible void method in the MessageServiceImpl class. Will fail if a different implementation is used

`execution(void example.MessageService.send(*))`

Any void send method taking one argument, in any object of type MessageService (including sub-classes or implementations of MessageService). More flexible choice – still works if implementation changes

`execution(@javax.annotation.security.RolesAllowed void send*(..))`

Any void method starting with send that is annotated with the @RolesAllowed annotation

`execution(* rewards.*.restaurant.*.*(..))`

There is one directory between rewards and restaurant

`execution(* rewards..restaurant..(..))`

There may be several directories between rewards and restaurant

`execution(* *..restaurant.*.*(..))`

Any sub-package called restaurant

8.5. Implementing Advice

There are different types of advices: @Before, @AfterReturning, @AfterThrowing, @After, and @Around.

8.5.1. @Before

Proxy ⇒ BeforeAdvice ⇒ Target

```
@Aspect
public class PropertyChangeTracker {

    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
    public void trackChange() {
        logger.info("Property about to change...");
    }
}
```

WARNING If the advice throws an exception, target will not be called.

8.5.2. @AfterReturning

Proxy ⇒ Target(success) ⇒ AfterAdvice

Audit all operations in the service package that return a Reward object.

```
@AfterReturning(value="execution(* service..*.*(..))", returning="reward")
public void audit(JoinPoint jp, Reward reward) {
    auditService.logEvent(jp.getSignature() + " returns the following reward object :"
+ reward.toString() );
}
```

8.5.3. @AfterThrowing

Proxy ⇒ Target (exception thrown) ⇒ AfterThrowingAdvice

Send an email every time a Repository class throws an exception of type DataAccessException.

```
@AfterThrowing(value="execution(* *..Repository.*(..))", throwing="e")
public void report(JoinPoint jp, DataAccessException e) {
    mailService.emailFailure("Exception in repository", jp, e);
}
```

The @AfterThrowing advice will not stop the exception from propagating but it can throw a **different type of exception**.

NOTE If you wish to stop the exception from propagating any further, you can use an @Around advice.

8.5.4. @After

Proxy ⇒ Target (successful or exception) ⇒ AfterAdvice

Called regardless of whether an exception has been thrown by the target or not.

Tracks calls to all update methods.

```
@Aspect
public class PropertyChangeTracker {

    private Logger logger = Logger.getLogger(getClass());

    @After("execution(void update*(..))")
    public void trackUpdate() {
        logger.info("An update has been attempted ...");
    }
}
```

8.5.5. @Around

Proxy ⇒ AroundAdvice ⇒ Target ⇒ AroundAdvice

Provides a `ProceedingJoinPoint` parameter. Inherits from JoinPoint and adds the proceed() method.

```
@Around("execution(@example.Cacheable * rewards.service..*.*(..))")
public Object cache(ProceedingJoinPoint point) throws Throwable {

    Object value = cacheStore.get(cacheKey(point));

    if (value == null) { ①
        value = point.proceed();
        cacheStore.put(cacheKey(point), value);
    }

    return value;
```

① Proceed only if not already cached.

8.5.6. Limitations

- Can only advise non-private methods
- Can only apply aspects to Spring Beans
- Limitations of weaving with proxies
    - When using proxies, suppose method a() calls method b() on the same class/interface
        - advice will never be executed for method b()

8.5.7. Summary

- Aspect Oriented Programming (AOP) modularizes cross-cutting concerns.
- An aspect is a module containing cross-cutting behavior.
  - Behavior is implemented as “advice”
  - Pointcuts select where advice applies
  - Five advice types: Before, AfterThrowing, AfterReturning, After and Around

9. Data Access

Implementing Data Access and Caching.

9.1. The Role of Spring in Enterprise Data Access

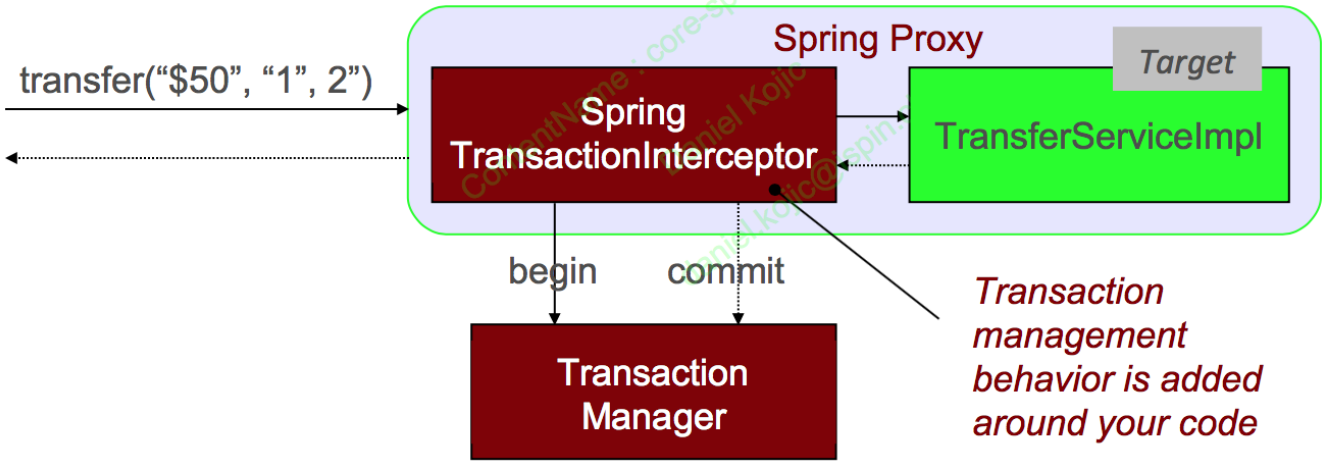
Works consistently with leading data access technologies. Resource management usually requires access (establishing or closing a connection) or transaction management. Spring manages this with **no additional code**, **prevents session leakage**, and **throws own exceptions**, independent from the underlying api. Transaction management can be done in a simple, declarative way.

Template Design Pattern link

- Define the outline or skeleton of an algorithm
  - Leave the details to specific implementations later
  - Hides away large amounts of boilerplate code
- Spring provides many template classes
  - JdbcTemplate
  - JmsTemplate
  - RestTemplate
  - WebServiceTemplate

9.1.1. Declarative Transaction Management

Every thread needs its own transaction. Spring’s transaction manager handles these within the **thread-local storage**. Data-access code, like JdbcTemplate, finds it automatically or do it yourself: `DataSourceUtils.getConnection(dataSource)`.



Many enterprise applications consist of three logical layers.

Service Layer (or application layer)

Exposes high-level application functions. Use-cases and business logic is defined here.

Data access Layer

Defines interface to the application's data repository (such as a Relational or NoSQL database).

Infrastructure Layer

Exposes low-level services to the other layers.

9.2. Spring's DataAccessExceptionHierarchy

Checked Exceptions

- Force developers to handle errors. But if you can't handle it, must declare it.
- **Bad**: Intermediate methods must declare exception(s) from all methods below (form of tight-coupling).

Unchecked Exceptions

- Can be thrown up the call hierarchy to the best place to handle it.
- **Good**: Methods in between don't know about it.
  - BetterinanEnterpriseApplication
- Spring throws Runtime (unchecked) Exceptions

SQLExceptions are **too general**. There is only one exception for every database error. Calling class 'knows' you are using JDBC (tight coupling). Spring provides the `DataAccessException` hierarchy. It **hides** whether you are using JPA or anything else and provides several **unchecked** exceptions which are consistent between different technologies.

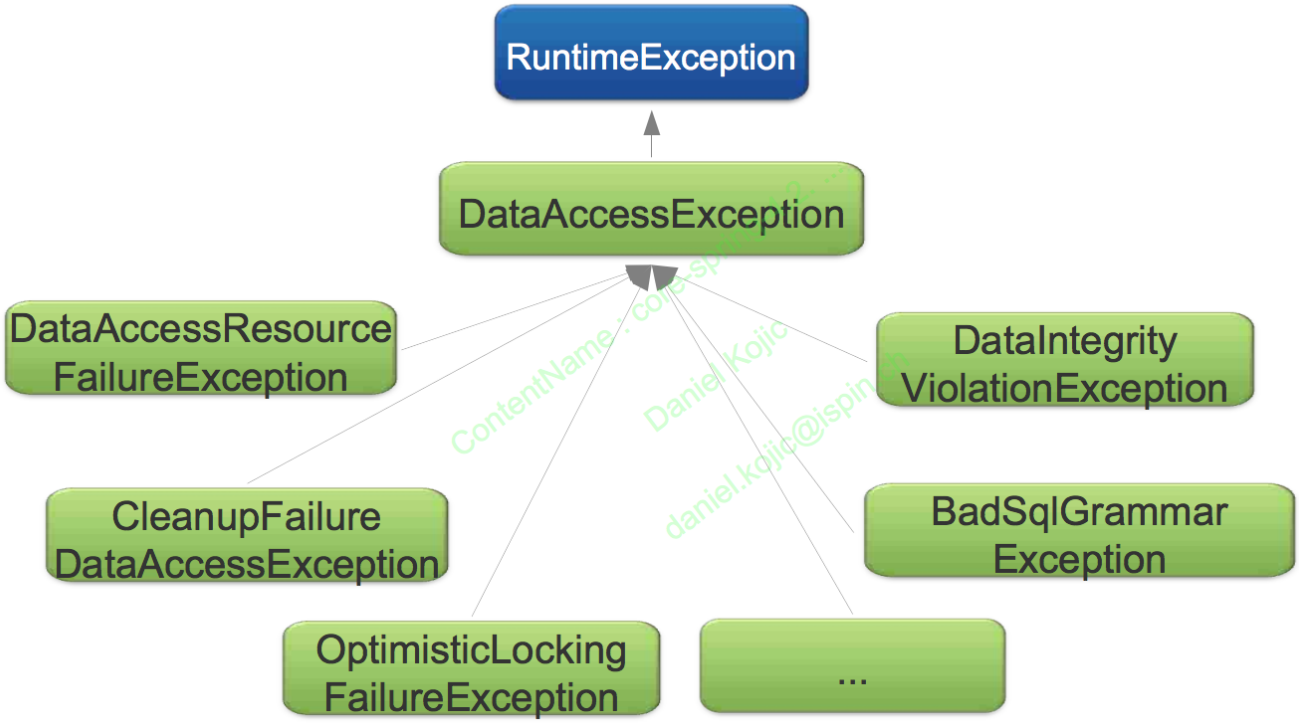


Figure 3. Spring Data Access Exceptions.

9.3. Using Test Databases

Spring provides an **Embedded Database Builder** to conveniently define a new **(empty) in-memory database**. HSQL, H2 and Derby are supported.

In java...

```
@Bean
public DataSource dataSource() {

    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setName("testdb")
            .addScript("classpath:/testdb/schema.db")
            .addScript("classpath:/testdb/test-data.db").build();
}
```

and xml possible.

```
<bean class="example.order.JdbcOrderRepository" >
    <property name="dataSource" ref="dataSource" />
</bean>

<jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:test-data.sql" />
</jdbc:embedded-database>
```

Also allows populating other (existing) data sources.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="url" value="${dataSource.url}" />
    <property name="username" value="${dataSource.username}" />
    <property name="password" value="${dataSource.password}" />
</bean>

<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:schema.sql" /> ①
    <jdbc:script location="classpath:test-data.sql" />
</jdbc:initialize-database>
```

① Initializes an external database.

Also in java.

```
@Configuration
public class DatabaseInitializer {
    @Value("classpath:schema.sql")
    private Resource schemaScript;

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

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

    @Bean
    public DataSourceInitializer anyName(final DataSource dataSource) {
        final DataSourceInitializer initializer = new DataSourceInitializer();
        initializer.setDataSource(dataSource);
        initializer.setDatabasePopulator(databasePopulator()); ①
        return initializer;
    }
```

① Explicitly create a database initializer which will do the work in its post-construct method.

9.4. Implementing Caching

A cache is a key-value store (map) and can be applied to methods often returning the same value for equal inputs. A unique key (cache key) must be generated from the arguments.

Spring transparently applies caching to Spring beans (AOP). Just mark methods `@Cacheable`, provide a caching key(s) and the name of cache to use (multiple caches supported, define in spring config). Enable caching via. `@EnableCaching` or `<cache:annotation-driven />`. You will also have to **specify a**

cache-manager (some provided in `org.springframework.cache`). Consider 3rd party cache-managers like ehCache or Gemfire.

In-memory cache.

```
@Cacheable
public Country[] loadAllCountries() { ... }
```

Implementing a custom cache-manager: a concurrent map cache.

```
@Bean
public CacheManager cacheManager() {
    SimpleCacheManager cmgr = new SimpleCacheManager();
    Set<Cache> caches = new HashSet<Cache>();
    caches.add(new ConcurrentMapCache("topAuthors"));
    caches.add(new ConcurrentMapCache("topBooks"));
    cmgr.setCaches(caches);
    return cmgr;
}
```

`@Cacheable` marks a method for caching its result is stored in a cache. Subsequent invocations (with the same arguments) fetch data from cache using key (method not executed).The `key` for each cached data-item uses SpEL and argument(s) of method.

```
public class BookService {

    @Cacheable(value="topBooks", key="#title", condition="#title.length < 32") ①
    public Book findBook(String title, boolean checkWarehouse) { ... }

    @Cacheable(value="topBooks", key="#author.name") ②
    public Book findBook2(Author author, boolean checkWarehouse) { ... }

    @Cacheable(value="topBooks", key="T(example.KeyGen).hash(#author)") ③
    public Book findBook3(Author author, boolean checkWarehouse) { ... }

    @CacheEvict(value="topBooks") ④
    public void loadBooks() { ... }

    ...
}
```

① Only cache if condition true.

② Use object property.

③ Custom key generator.

④ Clear cache before method invoked.

9.5. NoSQL databases

Spring does not only support SQL databases. It supports many more such as document-based, graph-based, big-data and column stores.

9.6. Summary

- Enables layered architecture principles
  - Higher layers should not know about data management below
- Isolate via Data Access Exceptions
  - Hierarchy makes them easier to handle
- Provides consistent transaction management
  - Supports most leading data-access technologies Relational and non-relational (NoSQL)
- A key component of the core Spring libraries
- Automatic caching facility

10. JDBC

Using the JDBC Template.

10.1. Problems with traditional JDBC

Traditional JDBC means **redundant, error prone** code with **poor exception handling**.

10.2. Spring's JdbcTemplate

JDBC Template Responsibilities

- Acquisition/release of the connection
- Transaction management
- Execution of the statement
- Processing of the result set
- Handling any exceptions

Greatly simplifies use of the JDBC API. Eliminates repetitive boilerplate code, hence alleviating a common causes of bugs. It also handles SQLExceptions properly without sacrificing power. It provides full access to the standard JDBC constructs.

Creating a JdbcTemplate requires a **datasource**. Once instantiated, **reuse** it. It can query **simple types, generic maps and domain objects**.

JDBC template example.

```
public class JdbcCustomerRepository implements CustomerRepository {

    private JdbcTemplate jdbcTemplate;

    public JdbcCustomerRepository(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int getCustomerCount() {
        String sql = "select count(*) from customer";
        return jdbcTemplate.queryForObject(sql, Integer.class); ①
    }

    public int getCountOfNationalsOver(Nationality nationality, int age) {
        String sql = "select count(*) from PERSON " +
                "where age > ? and nationality = ?";
        return jdbcTemplate.queryForObject(sql, Integer.class, age,
nationality.toString()); ②
    }
}
```

① Throws unchecked exceptions only.

② Bind variables.

10.3. Query Execution

10.3.1. Generic Queries

JdbcTemplate returns each row of a ResultSet as a Map. When expecting a single row, use `queryForMap(..)`. Use `queryForList(..)` for multiple rows. The data fetched does not need mapping to a Java object. Be careful with very large data-sets.

10.3.2. Querying Generic Maps

Query for a single row. Returns a map of column name|value pairs.

```
public Map<String,Object> getPersonInfo(int id) {
    String sql = "select * from PERSON where id=?";
    return jdbcTemplate.queryForMap(sql, id);
}
```

Query for multiple rows. Returns a list of maps of column name|value pairs.

```
public Map<String,Object> getPersonInfo(int id) {
    String sql = "select * from PERSON where id=?";
    return jdbcTemplate.queryForMap(sql, id);
}
```

10.3.3. Querying Domain Objects

Spring allows mapping relational data into domain objects e.g., a ResultSet to an Account using a callback approach. You may prefer to use ORM for this (JdbcTemplate vs. JPA or similar mappings). Some tables may be too hard to map with JPA.

10.3.4. Working with result sets

RowMapper

Maps a single row of a ResultSet to an object and can be used for single- and multiple- row queries.

```
public Person getPerson(int id) {
    return jdbcTemplate.queryForObject("select first_name,
            last_name from PERSON where id=?",
            new PersonMapper(),
            id);
}

...

class PersonMapper implements RowMapper<Person> {
    public Person mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Person(rs.getString("first_name"),
                rs.getString("last_name"));
    }
}
```

RowCallbackHandler

- Spring provides a simpler RowCallbackHandler interface when there is no return object
  - Streaming rows to a file
  - Converting rows to XML
  - Filtering rows before adding to a Collection (filtering in SQL is much more efficient)
  - Faster than JPA equivalent for big queries (avoids result-set to object mapping)

```
public class JdbcOrderRepository {

    public void generateReport(Writer out) {
        // select all orders of year 2009 for a full report
        jdbcTemplate.query("select * from order where year=?",
                new OrderReportWriter(out), 2009);
    }
}

class OrderReportWriter implements RowCallbackHandler {

    public void processRow(ResultSet rs) throws SQLException {
        // parse current row from ResultSet and stream to output
    }
}
```

ResultSetExtractor

Spring provides a ResultSetExtractor interface for processing an entire ResultSet at once. You are responsible for iterating the ResultSet. Useful e.g. for mapping entire ResultSet to a single object.

```
public class JdbcOrderRepository {

    public Order findByConfirmationNumber(String number) {
        // execute an outer join between order and item tables
        return jdbcTemplate.query("select...from order o, item i...conf_id = ?",
                new OrderExtractor(), number);
    }
}

class OrderExtractor implements ResultSetExtractor<Order> {

    public Order extractData(ResultSet rs) throws SQLException {
        Order order = null;
        while (rs.next()) {
            if (order == null) {
                order = new Order(rs.getLong("ID"), rs.getString("NAME"), ...);
            }
            order.addItem(mapItem(rs));
        }
        return order;
    }
}
```

10.3.5. Summary of Callback Interfaces

RowMapper

Best choice when each row of a ResultSet maps to a domain object.

RowCallbackHandler

Best choice when no value should be returned from the callback method for each row.

ResultSetExtractor

Best choice when multiple rows of a ResultSet map to a single object.

10.4. Inserts and Updates

```
public int insertPerson(Person person) {
    return jdbcTemplate.update( ①
            "insert into PERSON (first_name, last_name, age) values (?, ?, ?)",
            person.getFirstName(),
            person.getLastName(),
            person.getAge();
}

public int updateAge(Person person) {
    return jdbcTemplate.update(
            "update PERSON set age=? where id=?",
            person.getAge(),
            person.getId());
}
```

① Update method returns number of rows modified.

10.5. Exception handling

The JdbcTemplate transforms SQLExceptions into `DataAccessExceptions`.

11. Transactions

Transactional proxies and @Transactional.

11.1. Why use Transactions?

A set of tasks which take place as a single, indivisible action. It follows the **ACID** principles.

Atomic

Each unit of work is an all-or-nothing operation.

Consistent

Database integrity constraints are never violated.

Isolated

Isolating transactions from each other.

Durable

Committed changes are permanent.

11.2. Java Transaction Management

Java has several APIs which **handle transactions differently**: JDBC, JMS, JTA, Hibernate, JPA, etc. Each uses **program code to mark the start and end of the transaction** (Transaction Demarcation) where there are different APIs for **Global vs Local transactions**.

Local Transactions

Transactions managed by underlying resource.
App ⇒ Database

Global (distributed) Transactions

Transaction managed by separate, dedicated transaction manager.
App ⇒ TRX manager ⇒ PSQL, RabbitMQ, …

11.2.1. Drawbacks

- Multiple APIs for different local resources
- Programatic transaction demarcation
  - Typically performed in the repository layer (wrong place)
  - Usually repeated (cross-cutting concern)
- Service layer more appropriate
  - Multiple data access methods often called within a single transaction
  - But: don't want data-access code in service-layer Orthogonal concerns
- Transaction demarcation should be independent of transaction implementation

Transactions which are already in progress cannot be joined and code cannot be used with global transactions.

```
 try {
    conn = dataSource.getConnection(); ①
    conn.setAutoCommit(false); ②
    ...
    conn.commit(); ②
} catch (Exception e) { ③
    conn.rollback();
    ...
}
```

① Specific to JDBC API.

② Programatic trx demarcation.

③ Checked exceptions.

11.3. Spring Transaction Management

- Spring separates transaction demarcation from transaction implementation
  - Demarcation expressed declaratively via AOP
    - Programatic approach also available
  - PlatformTransactionManager abstraction hides implementation details.
    - Several implementations available
- Spring uses the same API for global vs. local.
  - Change from local to global is minor
    - Just change the transaction manager

There are 2 steps: declare a `PlatformTransactionManager` bean and the transactional methods (annotationm, XML or programatic).

11.3.1. PlatformTransactionManager

Spring's `PlatformTransactionManager` is the base interface for the abstraction. Several implementations are available e.g., DataSourceTransactionManager, HibernateTransactionManager and more.

Manager for a datasource.

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

In the code.

```
public class RewardNetworkImpl implements RewardNetwork {

    @Transactional
    public RewardConfirmation rewardAccountFor(Dining d) {
        // atomic unit-of-work
    }
}
```

In the configuration.

```
@Configuration
@EnableTransactionManagement ①
public class TxnConfig {

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

① Defines a Bean Post-Processor.

What happens

1. Target object wrapped in a proxy (Around advice)
2. Proxy implements the following behavior
    a. Transaction started before entering the method
    b. Commit at the end of the method
    c. Rollback if method throws a RuntimeException (default, can be overridden)
3. Transaction context bound to current thread.
4. All controlled by configuration

11.3.2. @Transactional

On class the @Transactional applies to all methods declared in the interface(s). You can also declare it on both, class- and method- level but with different settings e.g., timeouts.

11.4. Isolation Levels

There are 4 isolation levels where some DBMS may not support all levels.

```
@Transactional (isolation=Isolation.XXX)
```

READ_UNCOMMITTED

- Lowest level - allows dirty reads
- Current transaction can see the results of another uncommitted unit-of-work
- Typically used for large, intrusive read-only transactions
- And/or where the data is constantly changing

READ_COMMITTED

- Does not allow dirty reads (only committed information can be accessed)
- Default strategy for most databases

REPEATABLE_READ

- Does not allow dirty reads
- Non-repeatable reads are prevented
  - If a row is read twice in the same transaction, result will always be the same
  - Might result in locking depending on the DBMS

SERIALIZABLE

- Prevents non-repeatable reads and dirty-reads
- Also prevents phantom reads

11.5. Transaction Propagation

Transaction propagation is calling a `@Transactional` method within another `@Transactional` method. There are 7 **levels of propagation**. Two will be presented here.

```
@Transactional( propagation=Propagation.XXX )
```

REQUIRED

Default. Execute within a current transaction, create a new one if none existing.

REQUIRES_NEW

Create a new transaction, suspending the current transaction of one exists.

MANDATORY

Throw exception if none exists; otherwise use current transaction.

NEVER

Don't create a transaction if none exists. Throw exception if one existing.

NOT_SUPPORTED

Don't create a transaction if none exists. Suspend transaction, if one exists, then run method outside of a transaction.

SUPPORTS

Don't create a transaction if none exists; otherwise use existing transaction.

11.6. Rollback rules

By default, a transaction is rolled back if a RuntimeException has been thrown. Default settings can be overridden with `rollbackFor` and `noRollbackFor` attributes.

```
@Transactional(rollbackFor=MyCheckedException.class,
noRollbackFor={JmxException.class, MailException.class})
```

11.7. Testing

Annotate test method (or class) with `@Transactional`. This runs the test method in a transaction and **rolls back afterwards**. Using the `@Commit` annotation, the transaction won't be rolled back. Test methods with `@BeforeTransaction` get executed before the transaction is created.

12. Object Relational Mapping (ORM)

TODO.

13. Java Persistence API (JPA)

Object Relational Mapping with Spring & Java Persistence API.

13.1. Introduction to JPA

The Java Persistence API is designed for operating on domain objects: **Pojos, no interfaces**. It is a common API for object-relational mapping.

13.1.1. General Concepts

The key concepts comprise the Entity Manager, Entity Manager Factory and the Persistence Context.

EntityManager

- Manages a unit of work and persistent objects therein: the `PersistenceContext`.
- Lifecycle often bound to a Transaction (usually container- managed).

```
persist(Object o); // SQL: insert into table ... ①
remove(Object o); // SQL: delete from table ... ②
find(Class entity, Object primaryKey); // SQL: select * from table where id = ? ③
Query createQuery(String jpqlString); ④
flush(); ⑤
```

① Adds the entity to the Persistence Context.

② Removes the entity from the Persistence Context.

③ Find by primary key.

④ Create a JPQL query.

⑤ Force changed entity state to be written to database immediately.

EntityManagerFactory

- thread-safe, shareable object that represents a single data source / persistence unit.
- Provides access to new application-managed EntityManagers.

Persistence Unit

- Describes a group of persistent classes (entities)
- Defines provider(s)
- Defines transactional types (local vs JTA)
- Multiple Units per application are allowed

NOTE	The configuration can be in the Persistence Unit in the Spring bean-file or a combination of the two.

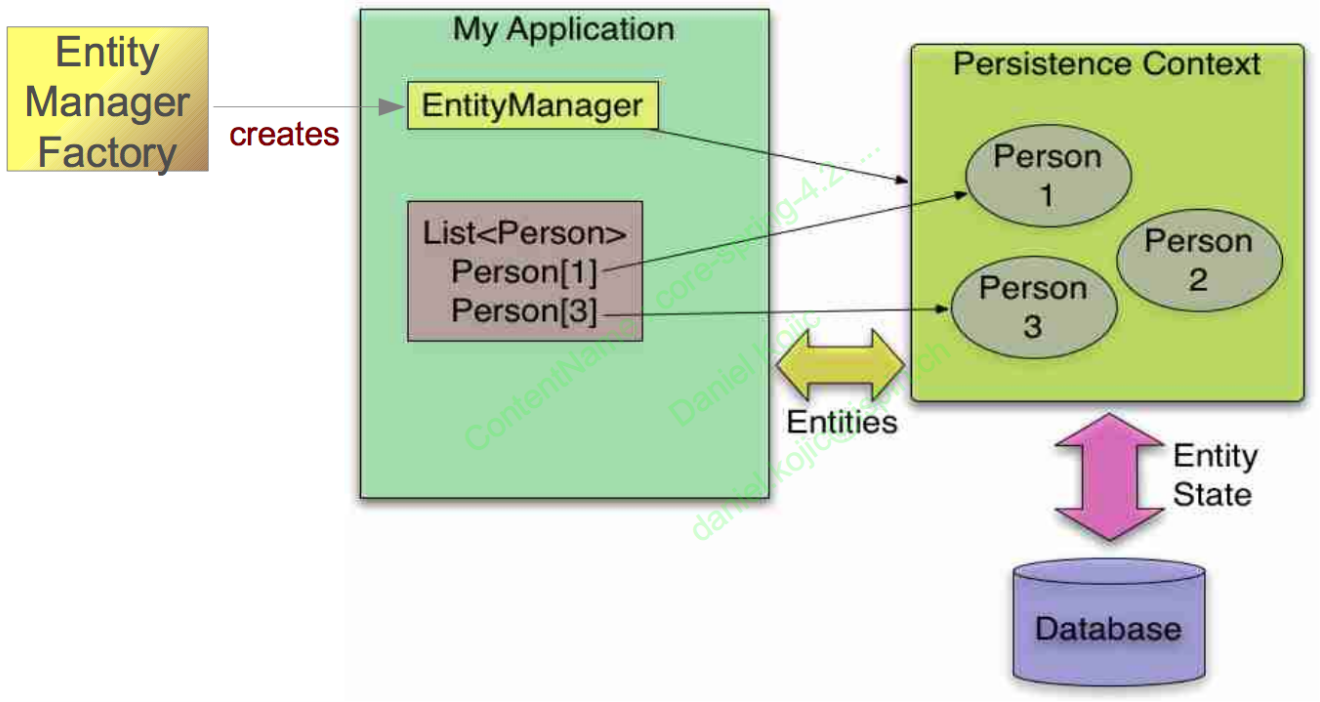


Figure 4. Persistence Context and EntityManager.

13.1.2. Mapping

JPA requires metadata for mapping classes/fields to database tables/columns, usually provided as **annotations** (XML mappings also supported for overrides). JPA metadata relies on defaults → do not specify the obvious. Annotate classes (table), fields (column, persistent by default vs. transient) or getters (also columns).

Common relationship mappings e.g., single and collection of entities are supported. **Associations can be uni- or bi-directional**. You can also **map a row to multiple classes** with `@AttributeOverride(…)`.

Mapping using fields. `@Entity` and `@Id` are mandatory.

```
@Entity ①
@Table(name= "T_CUSTOMER") ①
public class Customer {

    @Id ②
    @Column(name="cust_id") ③
    private Long id;

    @Column(name="first_name") ③
    private String firstName;

    @Transient ④
    private User currentUser;

    @OneToMany
    @JoinColumn (name="cid") ⑤
    private Set<Address> addresses;

    @Embedded
    @AttributeOverride (name="postcode", column=@Column(name="ZIP")) ⑥
    private Address office;
    ...
}

@Embeddable
public class Address {
    private String street;
    private String suburb;
    private String city;
    private String postcode; ⑥
    private String country;
}
```

① Mark as an entity Optionally override table name.

② Mark id-field (primary key).

③ Optionally override column names.

④ Not stored in database.

⑤ Foreig key in Address table.

⑥ Maps to "ZIP" column in "T_CUSTOMER".

Mapping using accessors.

```
@Entity @Table(name= "T_CUSTOMER")
public class Customer {
    private Long id;
    private String firstName;

    @Id ①
    @Column (name="cust_id")
    public Long getId() { return this.id; }

    @Column (name="first_name") ②
    String getFirstName() { return this.firstName; }

    public void setFirstName(String fn) { this.firstName = fn; }
}
```

① Must place `@Id` on the getter.

② Other annotations also place on getter methods.

13.1.3. Querying

- Retrieve an object **by primary key**.
- Query for objects using **JPA Query Language (JPQL)**.
  - Similar to SQL and HQL.
- Query for objects using **Criteria Queries**.
  - API for creating ad hoc queries.
- Execute **SQL directly** to underlying database.
  - "Native" queries, allow DBMS-specific SQL to be used.
  - Consider JdbcTemplate instead when not using managed objects – more options/control, more efficient.

Querying by Primary Key

To retrieve an object by its database identifier simply call find() on the EntityManager. Returns null if no object exists.

Querying with JPQL

Query for objects with `SELECT` based on properties or associations (cannot use `*`).

```
// Query with named parameters
TypedQuery<Customer> query = entityManager.createQuery(
        "select c from Customer c where c.address.city = :city",
        Customer.class); query.setParameter("city", "Chicago");
List<Customer> customers = query.getResultList();

// ... or using a single statement
List<Customer> customers2 = entityManager.createQuery("select c from Customer c ...",
Customer.class)
        .setParameter("city", "Chicago")
        .getResultList();

// ... or if expecting a single result
Customer customer = query.getSingleResult();
```

13.2. Configuring JPA in Spring

Steps to using JPA with Spring

1. Define an EntityManagerFactory bean.
2. Define a DataSource bean
3. Define a Transaction Manager bean
4. Define Mapping Metadata
5. Define DAOs

```
@Beean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(){

    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
    adapter.setShowSql(true);
    adapter.setGenerateDdl(true);
    adapter.setDatabase(Database.HSQL);

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

    LocalContainerEntityManagerFactoryBean emfb = new
LocalContainerEntityManagerFactoryBean();
    emfb.setDataSource(dataSource);
    emfb.setPackagesToScan("rewards.internal");
    emfb.setJpaProperties(props); emfb.setJpaVendorAdapter(adapter);
    return emfb;
}

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

@Bean
public DataSource dataSource() { // Lookup via JNDI or create locally. }
```

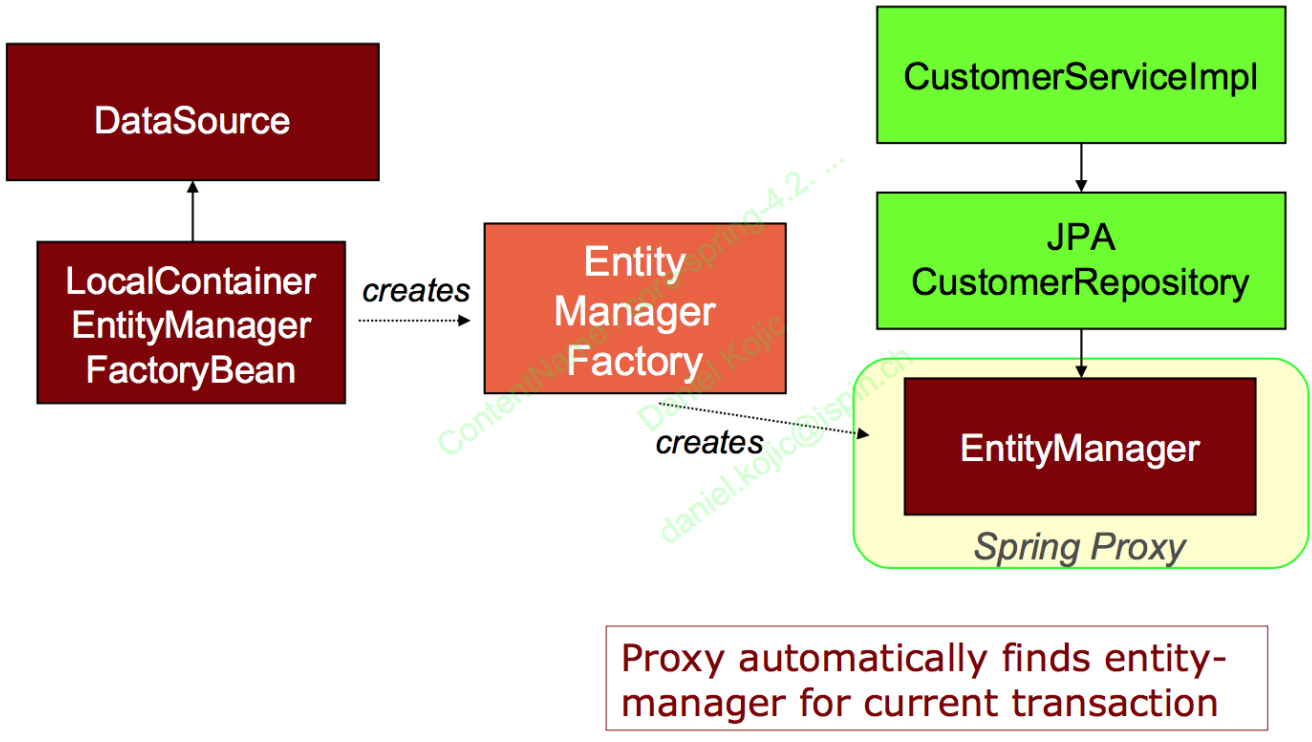


Figure 5. EntityManagerFactoryBean Configuration.

13.3. Implementing JPA DAOs

JPA provides configuration options so Spring can manage transactions and the EntityManager. There are **no Spring dependencies in your DAO** implementations. Optional: Use AOP for transparent exception translation: rethrows JPA PersistenceExceptions as Spring's DataAccessExceptions

Spring-Managed Transactions & EntityManager

- Transparently participate in Spring-driven transactions:
  - Use a Spring FactoryBean for building the EntityManagerFactory
  - Inject an EntityManager reference with @PersistenceContext
- Define a transaction manager
  - JpaTransactionManager / JtaTransactionManager

The repository.

```
public class JpaCustomerRepository implements CustomerRepository {
    private EntityManager entityManager;

    @PersistenceContext ①
    public void setEntityManager (EntityManager entityManager) {
        this. entityManager = entityManager;
    }

    public Customer findById(long orderId) {
        return entityManager.find(Customer.class, orderId); ②
    }
}
```

① Automatic injection of EM proxy.

② Proxy resloves to EM when used.

The configuration.

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() { ... }

@Bean
public CustomerRepository jpaCustomerRepository() {
    return new JpaCustomerRepository();
}

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

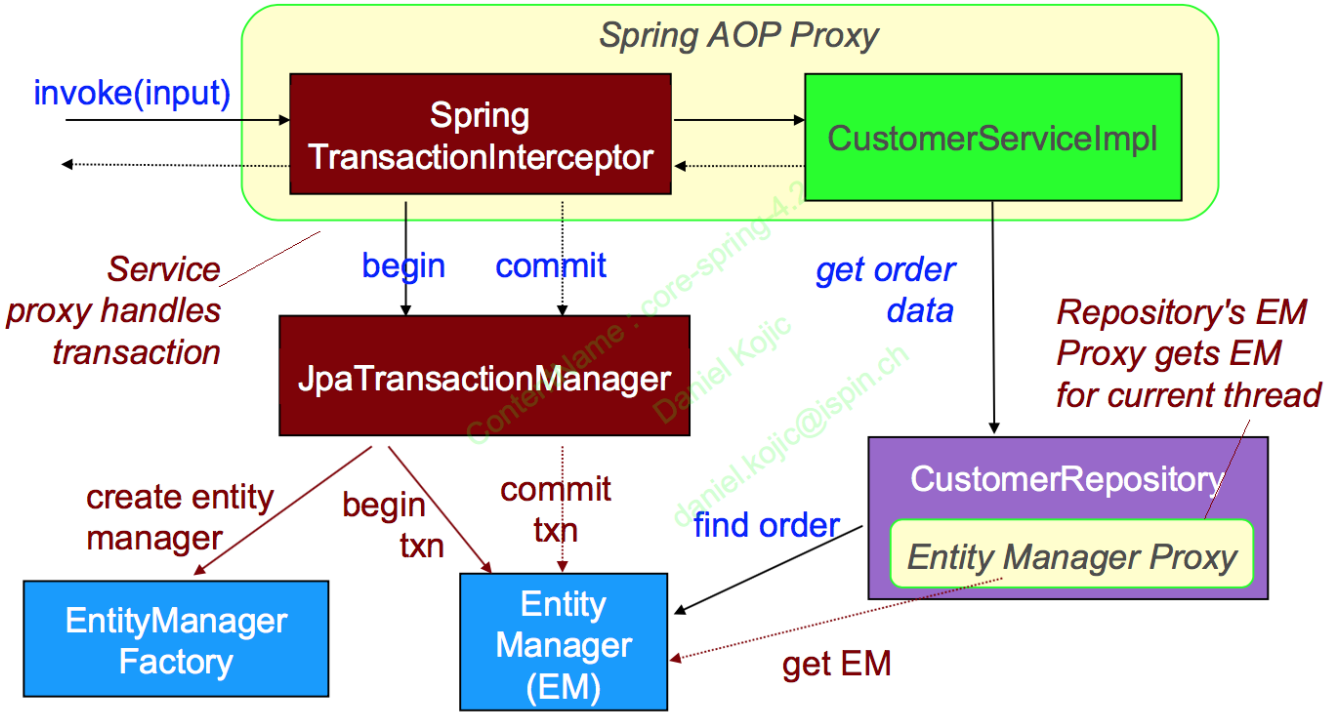


Figure 6. How JPA works.

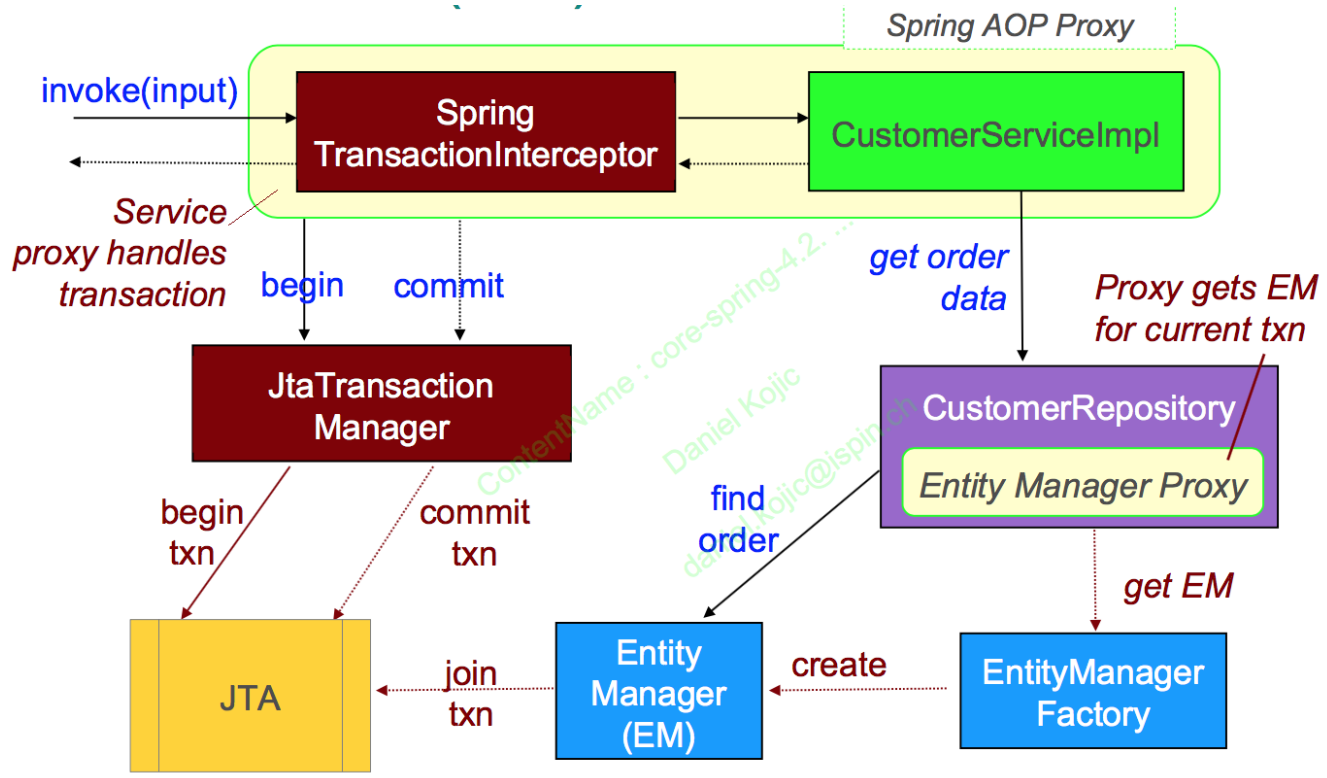


Figure 7. How JTA works.

13.4. Spring Data – JPA

Spring Data reduces boiler plate code for data access. It provides **instant repositories** by **annotating domain classes**. Repositories are **just interfaces**. Spring implements it at runtime by scanning for `Repository<T,K>` interfaces. **CRUD-methods are then auto-generated**. It allows

paging, custom queries and sorting.

To use it with JPA, annotate JPA Domain object as usual and define `@EnableJpaRepositories(basePackages=com.acme.**.repository")` in your configuration class. Spring Data provides similar annotations to JPA e.g., `@Document`, `@Region`, `@NodeEntity` as well as templates e.g., `MongoTemplate`, `GemfireTemplate`, `RedisTemplate`. Extend predefined repository interfaces with custom finders or any method from the `CrudRepository`. Implement `PagingAndSortingRepository<T, K>` to add `Iterable<T> findAll(Sort)` and `Page<T> findAll(Pageable)`.

Auto-generated finders obey naming convention: `findBy<DataMember><Op>` where `<Op>` can be Gt, Lt, Ne, Between, Like, etc.

Extend predefined interfaces.

```
public interface CrudRepository<T, ID> extends Serializable> extends Repository<T, ID>
{
    public <S extends T> save(S entity);
    public <S extends T> Iterable<S> save(Iterable<S> entities);

    public T findOne(ID id);
    public Iterable<T> findAll();
    public void delete(ID id);
    public void delete(T entity);
    public void deleteAll();
}

public interface CustomerRepository extends CrudRepository<Customer, Long> {
    public Customer findByEmail(String someEmail); // No <Op> for Equals
    public Customer findByFirstOrderDateGt(Date someDate);
    public Customer findByFirstOrderDateBetween(Date d1, Date d2);

    @Query("select u from Customer u where u.emailAddress = ?1")
    Customer findByEmail(String email); // ?1 replaced by method param
}
```

13.5. Summary

- Use 100% JPA to define entities and access data
  - Repositories have no Spring dependency
  - Spring Data Repositories need no code!
- Use Spring to configure JPA entity-manager factory
  - Smart proxy works with spring-driven transactions
  - Optional translation to DataAccessExceptions

14. Spring Web

Developing Modern Web Applications: Servlet Configuration, Product Overview

14.1. Introduction

Spring provides support in the Web layer (Spring MVC, Spring WebFlow, ...) and **integrates well** with any Java web framework.

14.2. Using Spring in Web Applications

- Spring can be initialized within a webapp.
  - start up business services, repositories, etc.
- Uses a standard servlet listener.
  - Initialization occurs befire any servlets execute.
  - application ready for user requests.
  - ApplicationContext.close() is called when the application is stopped.
- Configuration either via. WebApplicationInitializer or web.xml (see Configuration via. WebApplicationInitializer. and Configuration via. web.xml.)

Configuration via. WebApplicationInitializer.

```
public class MyWebAppInitializer extends AbstractContextLoaderInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        // Create the 'root' Spring application context
        AnnotationConfigWebApplicationContext rootContext
                = new AnnotationConfigWebApplicationContext();
        rootContext.getEnvironment().setActiveProfiles("jpa"); // optional
        rootContext.register(RootConfig.class);
        return rootContext;
    }
}
```

Configuration via. web.xml.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/merchant-reporting-webapp-config.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

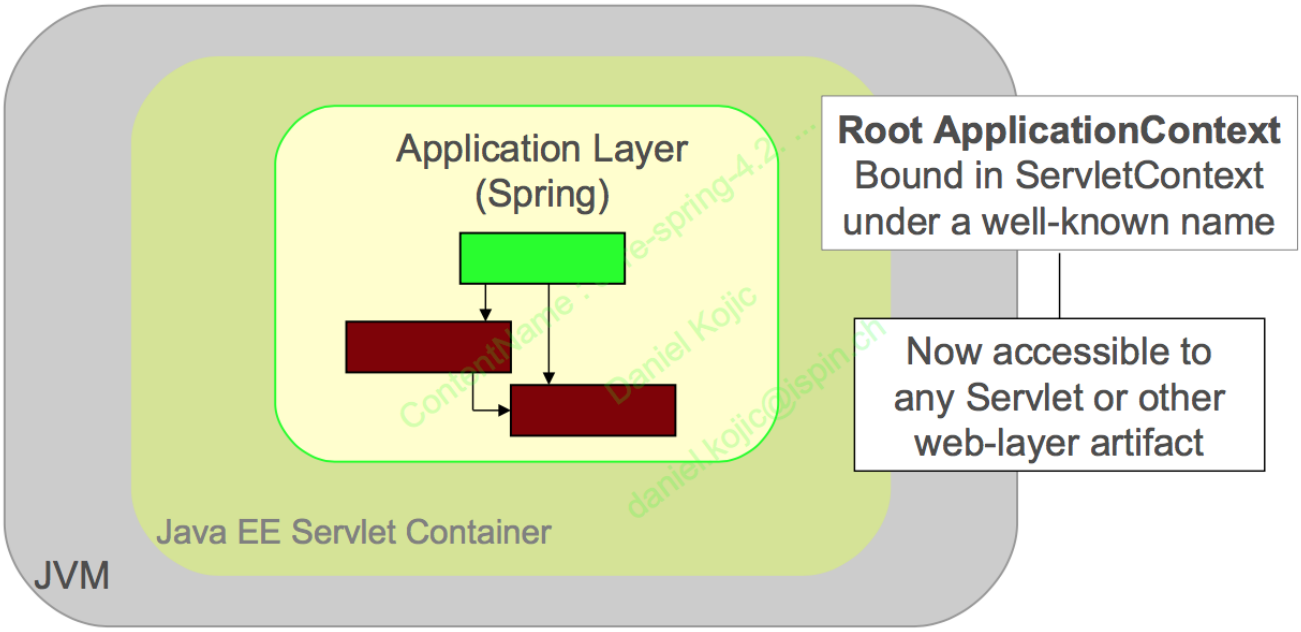


Figure 8. Servlet container after starting up.

Override `onStartup()` *method to define servlets. You cannot access spring beans yet.*

```
public class MyWebAppInitializer extends AbstractContextLoaderInitializer {
    protected WebApplicationContext createRootApplicationContext() {
        // ...Same configuration.
   }

    public void onStartup(ServletContext container) {
        super.onStartup(container);
        // Register and map a servlet
        ServletRegistration.Dynamic svlt = container.addServlet(
                "myServlet",
                new TopSpendersReportGenerator());
        svlt.setLoadOnStartup(1);
        svlt.addMapping("/");
}
```

- Dependency Injection of Servlets
    - Suitable for web.xml or AbstractContextLoaderInitializer

    - Use `WebApplicationContextUtils` to get the Spring ApplicationContext via ServletContext.
- Spring MVC Supports Dependency Injection

14.3. Overview of Spring Web

- Spring MVC
    - Web framework bundled with Spring
- Spring WebFlow
    - Plugs into Spring MVC
    - Implements navigation flows
- Spring Mobile
    - Routing between mobile / non-mobile versions of site
- Spring Social
    - Easy integration with Facebook, Twitter, etc.

14.4. Summary

- Spring can be used with any web framework
    - Spring provides the ContextLoaderListener that can be declared in web.xml
- Spring MVC is a lightweight web framework where controllers are Spring beans
- WebFlow plugs into Spring MVC as a Controller technology for implementing stateful "flows"

15. REST

REpresentational State Transfer

15.1. REST introduction

General

- Web apps not just usable by browser clients.
- REST is an architectural style that describes best practices to expose web services over HTTP.

Principles

- Expose resources through URIs.
- Resources support limited set of operations.
    - GET, PUT, POST, DELETE in case of HTTP.
    - All have well-defined semantics.
- Clients can request particular representation.
    - Resources can support multiple representations.

    - HTML, XML, JSON, …
- Representations can link to other resources.
    - Allows for extensions and discovery, like with web sites
- Rest extended: Hypermedia As The Engine of Application State (HATEOAS)
- Stateless architecture
    - No HttpSession usage
    - GETs can be cached on URL
    - Requires clients to keep track of state
    - Part of what makes it scalable
    - Looser coupling between client and server
- HTTP headers and status codes communicate result to clients.
    - All well-defined in HTTP Specification.

Benefits

- Every platform/language supports HTTP.
- Scalability.
- Support for redirect, caching, different representations, resource identification, …
- Support for multiple formats e.g., XML.

15.2. REST and Java

General

- Spring-MVC provides REST support as well.
- RestTemplate for building programmatic clients in Java.
- Single web-application for everything.

15.3. Spring MVC support for RESTful applications

15.3.1. Request/Response Processing

Request Mapping based on HTTP Method

- Can map HTTP requests based on method
- Allows same URL to be mapped to multiple methods
- Often used for form-based controllers (GET & POST)
    - Essential to support RESTful resource URLs
        - incl. PUT and DELETE

HTTP Status Code Support

- Web apps just use a handful of status codes.

    - Success: 200 OK
    - Redirect: 302/303 for Redirects
    - Client Error: 404 Not Found
    - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many additional codes to communicate with their clients.
- Add `@ResponseStatus`to controller method.
    - Don't have to set status on HttpServletResponse manually.

IMPORTANT	When using @ResponseStatus, void methods no longer imply a default view name. The reponse body will be empty.

Determining Location Header

- Location header value must be full URL.
    - Determine based on request URL.
        - Controller shouldn't know host name or servlet path.
- URL of created child resource usually a sub-path.
    - POST to http://www.myshop.com/store/orders gives http://www.myshop.com/store/orders/123
    - Use Spring's `UriTemplate`to ensure new URL is valid.

Exception Handling

- Define @ResponseStatus directly on a exception class.
- For existing exceptions you cannot annotate, use @ExceptionHandler method on controller.

Request mappings.

```
@RequestMapping(value="/orders", method=RequestMethod.GET)
public void listOrders(Model model) {
    // find all Orders and add them to the model
}

@RequestMapping(value="/orders", method=RequestMethod.POST)
public void createOrder(HttpServletRequest request, Model model) {
    // process the order data from the request
}

@RequestMapping(value="/orders", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public void createOrder(HttpServletRequest request, HttpServletResponse response) {
    Order order = createOrder(request);
    // determine full URI for newly created Order based on request
    response.addHeader("Location", getLocationForChildResource(request,
order.getId()));
}
```

Determining the location header.

```
String getLocationForChildResource(HttpServletRequest request, Object childIdentifier)
{
    StringBuffer url = request.getRequestURL();
    UriTemplate template = new UriTemplate(url.append("/{childId}").toString());
    return template.expand(childIdentifier).toASCIIString();
}
```

Can also annotate exception classes with @ResponseStatus.

```
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
public class OrderNotFoundException extends RuntimeException { ... }
```

Exception handler.

```
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler({DataIntegrityViolationException.class}) public void conflict() {
    // could add the exception, response, etc. as method params
}
```

15.3.2. Using MessageConverters

HttpMessageConverter

- Converts between HTTP request/response and object.
- Various implementations registered by default when using `@EnableWebMvc` or `<mvc:annotation-driven/>`.
- Define HandlerAdapter explicitly to register other HttpMessageConverters.

@RequestBody

- Enables converters for request data.
- Right converter chosen automatically.
  - Based on content type of request.
  - Order could be mapped from XML with JAXB2 or from JSON with Jackson, for example.

@ResponseBody

- Use converters for response data by annotating method with @ResponseBody
  - Converter handles rendering to response
  - No ViewResolver and View involved anymore!

Automatic Content Negotiation

- HttpMessageConverter selected automatically.
  - For @Request/ResponseBody annotated parameters.
  - Each converter has list of supported media types.

- Allows multiple representations for a single controller method.
  - Without affecting controller implementation.
  - Alternative to using Views.
- Flexible media-selection.
  - Based on Accept header in HTTP request, or URL suffix, or URL format parameter.

15.4. Summary

- REST is an architectural style that can be applied to HTTP-based applications.
  - Useful for supporting diverse clients and building highly scalable systems.
  - Java provides JAX-RS as standard specification.
- Spring-MVC adds REST support using a familiar programming model.
  - Extended by @Request-/@ResponseBody.
- Use RestTemplate for accessing RESTful apps.

16. Spring MVC

Web framework based on the Model/View/Controller pattern.

16.1. Request Processing Lifecycle

- Web request handling based on an incoming URL…
  - …we need to call a method…
  - …after which the return value (if any)…
  - …needs to be rendered using a view

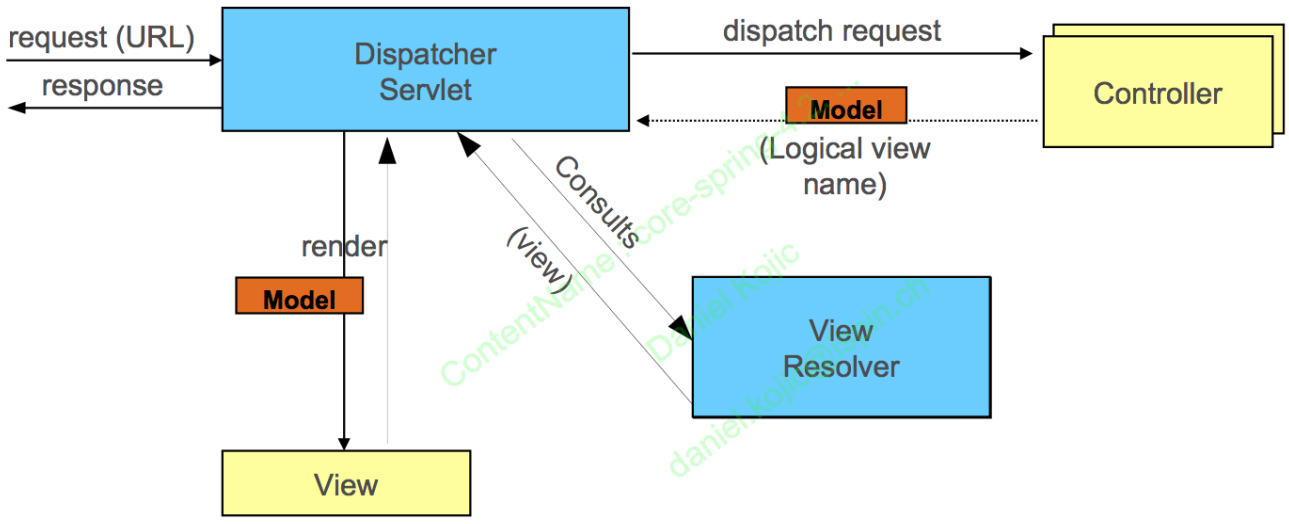


Figure 9. Request Processing Lifecycle.

16.2. Key Artifacts

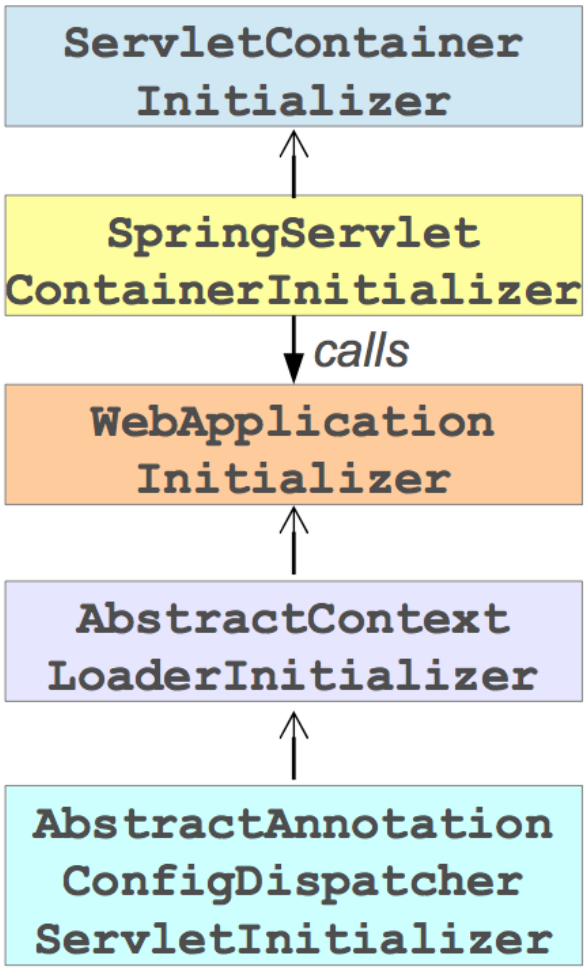
16.2.1. DispatcherServlet

General

- A “front controller”
  - coordinates all request handling activities
  - analogous to Struts ActionServlet / JSF FacesServlet
- Delegates to Web infrastructure beans
- Invokes user Web components
- Fully customizable
  - interfaces for all infrastructure beans
  - many extension points

Configuration

- Defined in web.xml or WebApplicationInitializer
- Uses Spring for its configuration
  - programming to interfaces + dependency injection
  - easy to swap parts in and out
- Creates separate “servlet” application context
  - configuration is private to DispatcherServlet
- Full access to the parent “root” context
  - instantiated via ContextLoaderListener
    - shared across servlets



Web Initializer

ServletContainerInitializer

Interface from Servlet 3 specification, implement to initialize servlet system.

SpringServletContainerInitialize

Spring's implementation which, in turn, delegates to one or more ...

WebApplicationInitializer

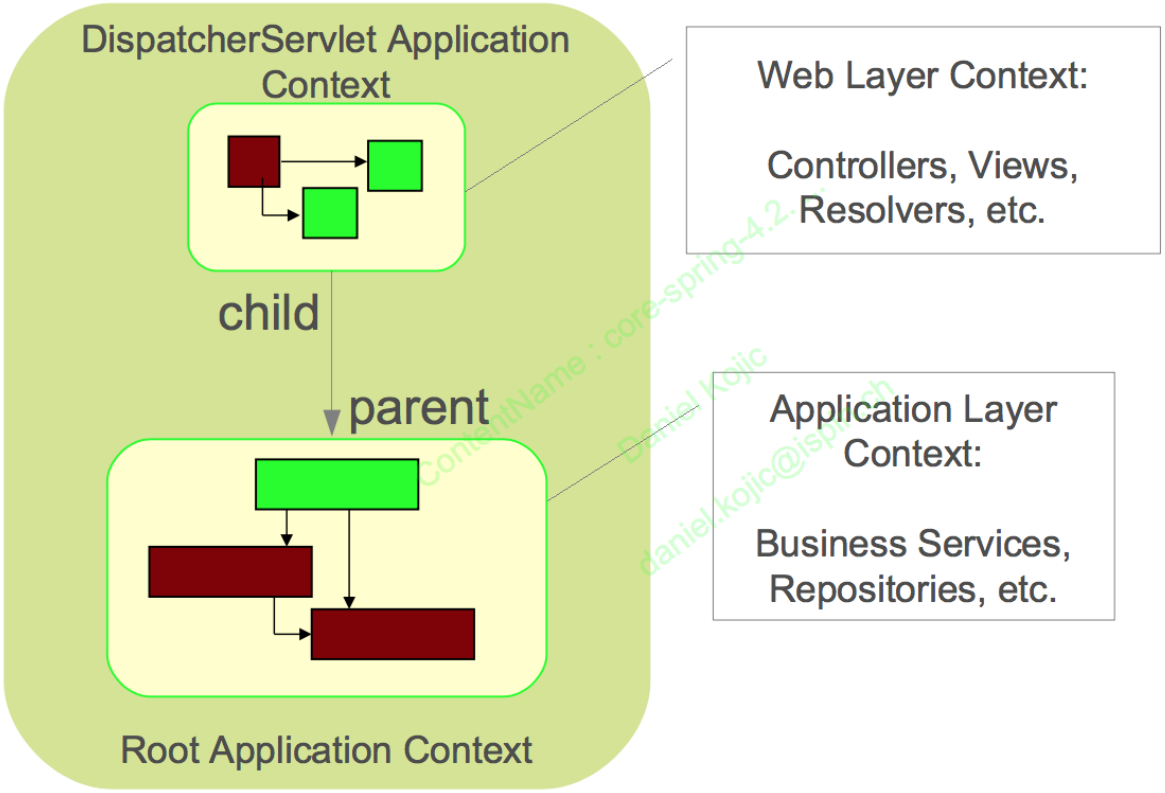
Base-class for all Spring MVC apps to implement for servlet configuration without web.xml.

AbstractContextLoaderInitializer

Sets up a ContextLoaderListener, you provide root ApplicationContext.

AbstractAnnotationConfigDispatcherServletInitializer

Defines a DispatcherServlet, assumes Java Config. You provide root and servlet Java config classes.



Java Configuration. Beans defined in MVC context have acess to root context beans.

```
public class MyWebInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    // Tell Spring what to use for the Root context:
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{ RootConfig.class };
    }

    // Tell Spring what to use for the DispatcherServlet context:
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{ MvcConfig.class };
    }

    // DispatcherServlet mapping:
    @Override
    protected String[] getServletMappings() {
        return new String[]{"/main/*"};
    }
}
```

16.2.2. Controllers

General

- Annotate controllers with @Controller

Request Mapping

- `@RequestMapping` tells Spring what method to execute when processing a particular request
  - Mapping rules typically URL-based, optionally using wild cards:
    - `/login`
    - `/editAccount`
    - `/listAccounts.htm`
    - `/reward//*`

Controller Method Parameters

- Pick parameters as you want.
  - HttpServletRequest, HttpSession, Principal ...
  - Model for sending data to the view.

Extracting Request Parameters

- Use @RequestParam annotation
  - Extracts parameter from the request
  - Performs type conversion

Uri Templates

- Values can be extracted from request URLs
  - Based on URI Templates
  - not Spring-specific concept, used in many frameworks
  - Use `{···}` placeholders and `@PathVariable`
- Allows clean URLs without request parameters

```
@Controller
public class AccountController {

    @RequestMapping("/listAccounts")
    public String list(Model model) { ... } ①

    @RequestMapping("/showAccount")
    public String show(@RequestParam("entityId") long id, Model model) { ... } ②

    @RequestMapping("/accounts/{accountId}")
    public String show(
            @PathVariable("accountId") long id, ③
            @RequestHeader("user-agent") String agent, ④
            Model model) { ... }
}
```

① Returns view name and holds data for view in method parameter.

② Extract request parameters.

③ Extract path variables.

④ Or even request headers.

16.2.3. Views

General

- A View renders web output.
  - Many built-in views available for JSPs, XSLT, templating approaches (Velocity, FreeMarker), etc.
  - View support classes for creating PDFs, Excel spreadsheets, etc.
- Controllers typically return a 'logical view name' String.
- ViewResolvers select View based on view name.

View Resolvers

- The DispatcherServlet delegates to a ViewResolver to obtain View implementation based on view name.
- The default ViewResolver treats the view name as a Web Application-relative file path
  - i.e. a JSP: /WEB-INF/reward/list.jsp
- Override this default by registering a ViewResolver bean with the DispatcherServlet
  - We will use InternalResourceViewResolver
  - Several other options available.

16.3. Quick Start

Steps to developing a Spring MVC application

1. Deploy a Dispatcher Servlet (one-time only)
2. Implement a controller
3. Register the Controller with the DispatcherServlet
4. Implement the View(s)
5. Register a ViewResolver (optional, one-time only)
6. Deploy and test

17. Spring Boot

Starter POMS and Auto-Configuration

17.1. What is Spring Boot?

An **opinionated runtime** for Spring Projects which **handles most low-level setup** with **support for different project types** like web and batch. You only need 3 files to get a running Boot app:

- Dependency management e.g., `pom.xml`
- Spring MVC controller
- Application launcher

17.1.1. Definition and Hello World example

Opinionated Runtime

- Spring Boot uses sensible defaults, “opinions”, mostly based on the classpath contents.
- For example
  - Sets up a JPA Entity Manager Factory if a JPA implementation is on the classpath.
  - Creates a default Spring MVC setup, if Spring MVC is on the classpath.
- Everything can be overridden easily.
  - But most of the time not needed

Deployment

- Example bundles Tomcat inside the application and runs as **executable jar**.
- Spring Boot apps can be deployed into an existing app server.
  - As familiar WAR file.

Example maven descriptor. Can also use gradle or Ant/Ivy.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId> ①
    <version>1.3.0.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId> ②
        <artifactId>spring-boot-starter-web</artifactId> ③
    </dependency>
    <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId> ④
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

① Defines properties e.g., `${spring.version=4.2}`

② Resolves ~16 JARs e.g., spring-boot-**.jar, spring-core-**.jar, even various web dependencies e.g., tomcat-*.jar

③ Spring MVC embedded tomcat. Version not needed. Defined by parent.

④ Resolves spring-test-**.jar, junit-**.jar, mockito-*.jar and various other test frameworks.

Spring MVC controller.

```
@RestController
public class HelloController {

    @RequestMapping("/")
    public String hello() {
        return "Greetings from Spring Boot!"; ①
    }
}
```

① Returns a String as the HTTP response body.

Annotation class.

```
@SpringBootApplication ①
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

① Runs embedded tomcat.

17.2. Spring Boot Explained

17.2.1. Dependency Management

How to use Spring Boot

- Add the appropriate Spring Boot dependencies.
- The easiest is to use a dependency management tool Spring Boot works with Maven, Gradle, Ant/Ivy.

General

- Spring Boot parent pom dfines key versions of dependencies and maven plugin.
- Various starter POMs available for common Java enterprise framworks e.g., spring-boot-starter-[jdbc|jpa|batch].

17.2.2. Auto Configuration

@EnableAutoConfiguration

- Annotation on a Spring Java configuration class.
  - Causes Spring Boot to automatically create beans it thinks you need.
  - Usually based on classpath contents, can easily override.

@SpringBootApplication

- Very common to use @EnableAutoConfiguration, @Configuration, and @ComponentScan together.
  - @ComponentScan, with no arguments, scans the current package and its sub-packages.

```
// this...
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class MyAppConfig { ... }

// ... vs this.
@SpringBootApplication
public class MyAppConfig { ... }
```

17.2.3. Containerless Applications

Spring Boot as a runtime

- Spring Boot can startup an embedded web server
  - You can run a web application from a JAR file!
  - Tomcat included in Web Starter
- Jetty can be used instead of Tomcat
  - In pom:
    - Exclude `spring-boot-starter-tomcat`
    - Include `spring-boot-starter-jetty`

Why run Web-Application outside of a Container?

- No separation of container config and app config.
  - They depend on each other anyway (like JNDI DS names, security config).
- Apps mostly target to a specific container.
  - Why not include that already?
- Easier debugging and profiling.
- Easier hot code replacement.
- No special IDE support needed.
- Familiar model for non-Java developers .

- Recommended for Cloud Native applications.

17.2.4. Packaging

- Spring Boot creates a single archive.
  - Jar or War.
  - Can also include the Application Server.
- Can be executed with “java -jar yourapp.war”.
- Gradle and Maven plugins available.
- Produces:
  - `*.jar`: Contains your code and all libs - executable.
  - `*.jar.original`: Contains only your code.

Maven Packaging. Add the boot maven plugin to pom.xml.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

17.3. Spring Boot inside of a Servlet Container

Your choice whether containerless or not

- Embedded container is just one feature of Spring Boot.
- Traditional WAR also benefits a lot from Spring Boot.
  - Automatic Spring MVC setup, including DispatcherServlet.
  - Sensible defaults based on the classpath content.
  - Embedded container can be used during development.

Spring Boot in a Servlet Container

- Spring Boot can also run in any Servlet 3.x container
  - e.g., Tomcat 7+, Jetty 8+
- Only small changes required
  - Change artifact type to WAR (instead of JAR).
  - Extend SpringBootServletInitializer.
  - Override configure method.
- Still no web.xml required

Spring Boot WAR file

- Spring Boot produces hybrid WAR file.
- Can still be executed with embedded Tomcat.
  - Using “java -jar yourapp.war”.
- Traditional WAR file produced as well.
  - Without embedded Tomcat.
  - Just drop it in your application server web app directory

Servlet Container and Containerless.

```
@ComponentScan
@EnableAutoConfiguration
public class Application extends SpringBootServletInitializer {

    protected SpringApplicationBuilder configure( SpringApplicationBuilder
application) {
        return application.sources(Application.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

17.4. Ease of Use Features

Externalized Properties

- Easily consumable with Spring `PropertySource`
- Spring boot automatically looks for `application.properties` in the classpath root.
- Starter POMs declare the properties to use (see reference documentation).
- Override location of a file:

```
System.setProperty("spring.config.name", "myserver");
SpringApplication.run(Application.class, args);
```

- Spring boot supports YAML configuration (application.yml).

Controlling Log Levels

- Boot can control the logging level
  - Just set it in application.properties
- Works with most frameworks e.g., Java Util Logging, Logback, Log4J and Log4J2.

Data Source Configuration

- Use either spring-boot-starter-jdbc or spring-boot-starter- data-jpa and include a JDBC driver on classpath.
- Declare poroperties:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Web Application Convenience

- Boot automatically configures Spring MVC DispatcherServlet and @EnableWebMvc defaults.
  - When spring-webmvc*.jar on classpath.
- Static resources served from classpath.
  - /static, /public, /resources or /META-INF/resources.
- Templates served from /templates.
  - When Velocity, Freemarker, Thymeleaf, or Groovy on classpath.
- Provides default /error mapping.
  - Easily overridden.

17.5. Summary

- Spring Boot speeds up Spring application development
- You always have full control and insight
- Nothing is generated
- No special runtime requirements
- No servlet container needed (if you want)
  - E.g. ideal for microservices

18. Spring Security

Addressing common Security Requirements.

18.1. High-Level Security Overview

Security Concepts

- **Principal**: User, device or system that performs an action.
- **Authentication**: Verifying that a principal's credentials are valid.
- **Authorization**: Deciding whether a principal is allowed to perform an operation.

- **Secured Item**: Resource that is being secured.

Authentication

- There are many authentication mechanisms.
  - e.g. basic, digest, form, X.509.
- There are many storage options for credential and authority information.
  - e.g. Database, LDAP, in-memory (development).

Authorization

- Authorization depends on authentication
  - Before deciding if a user can perform an action, user identity must be established.
- The decision process is often based on roles.
  - *ADMIN* can cancel orders.
  - *MEMBER* can place orders.
  - *GUEST* can browse the catalog.

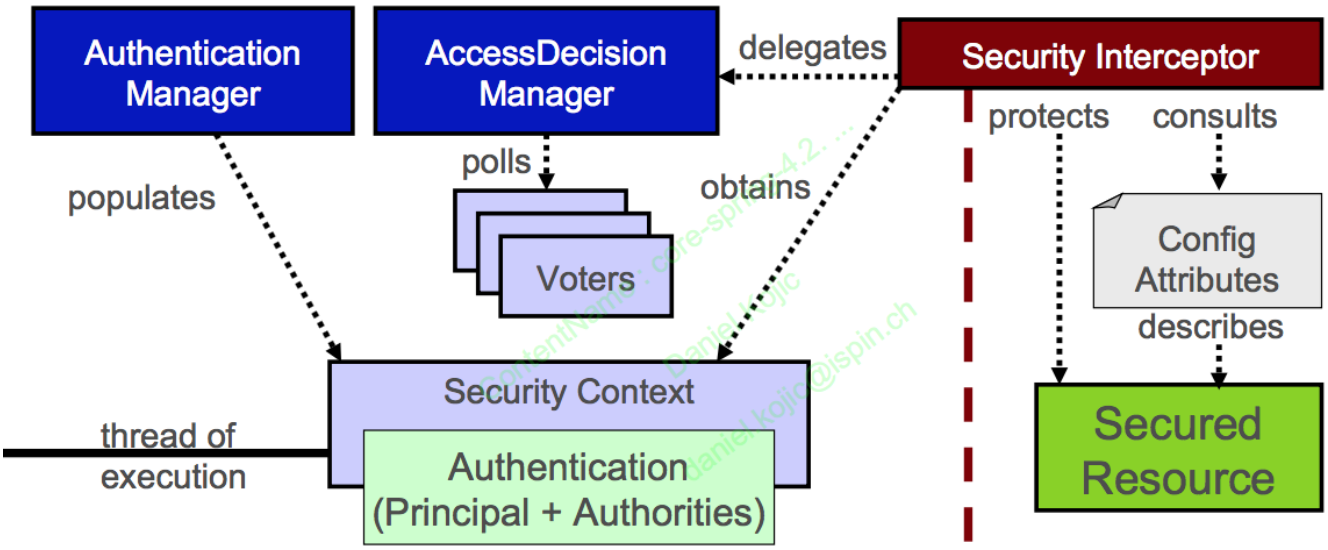
18.2. Motivations of Spring Security

Motivation

- **Spring Security is portable across containers**
  - Secured archive (WAR, EAR) can be deployed as-is.
  - Also runs in standalone environments.
  - Uses Spring for configuration.
- **Separation of Concerns**
  - Business logic is decoupled from security concerns.
  - Authentication and Authorization are decoupled.
    - Changes to the authentication process have no impact on authorization.
- **Flexibility**
  - Supports all common authentication mechanisms.
    - Basic, Form, X.509, Cookies, Single-Sign-On, etc.
  - Configurable storage options for user details (credentials and authorities).
    - RDBMS, LDAP, custom DAOs, properties file, etc.
- **Extensible**
  - All the following can be customized.
    - How a principal is defined.
    - How authorization decisions are made.
    - Where security constraints are stored.

Consistency of Approach

- The goal of authentication is always the same regardless of the mechanism.
  - Establish a security context with the authenticated principal's information.
  - Out-of-the-box this works for web applications.
- The process of authorization is always the same regardless of resource type.
  - Consult the attributes of the secured resource.
  - Obtain principal information from security context.
  - Grant or deny access.



18.3. Spring Security in a Web Environment

Extend the WebSecurityConfigurerAdapter for easiest use.

```
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception { ... } ①

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
... } ②
}
```

① Web-specific security settings.

② General security settings (authentication manager, ...).

Authorize requests.

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers("/css/**","/images/**","/javascript/**").permitAll()
            .antMatchers("/accounts/edit*").hasRole("ADMIN")
            .antMatchers("/accounts/account*").hasAnyRole("USER",ADMIN)
            .antMatchers("/accounts/**").authenticated()
            .antMatchers("/customers/checkout*").fullyAuthenticated()
            .antMatchers("/customers/**").anonymous();
```

Login and logout.

```
protected void configure(HttpSecurity http) throws Exception {
    http
            .authorizeRequests()
            .antMatchers("/aaa*").hasRole("ADMIN")
            .and() // method chaining

            .formLogin() // form-based authentication
            .loginPage("/login.jsp") // login url
            .permitAll() // any user can access
            .and()

            .logout() // logout for...
            .permitAll(); // ... any user
}
```

Example login page.

```
<c:url var=loginUrl value=/login.jsp /> ①
<form:form action=${loginUrl} method=POST>
    <input type=text name=username/> ②
    <br/>
    <input type=password name=password/> ②
    <br/>
    <input type=submit name=submit value=LOGIN/>
</form:form>
```

① URL that indicates an authentication request. Default: POST against URL used to display the page.

② The expected keys for generation of an authentication request token.

18.4. Configuring Web Authentication

Authentication Provider

*** DAO Authentication provider (default)**

- Expects a UserDetailsService implementation to provide credentials and authorities

    - Built-in: In-memory (properties), JDBC (database), LDAP
    - Custom
        - Custom Authentication provider
- Example: to get pre-authenticated user details when using single sign-on
    - CAS, TAM, SiteMinder …
        - Use a UserDetailsManagerConfigurer
- Three built in options:
    - LDAP, JDBC, in-memory (for quick testing)
- Or use your own UserDetailsService implementation

Sourcing Users from a Database

- Queries RDBMS for users and their authorities.
    - Provides default queries.
        - `SELECT username, password, enabled FROM users WHERE username = ?`
        - `SELECT username, authority FROM authorities WHERE username = ?`
- Groups also supported.
    - groups, group_members, group_authorities tables.
- Advantage.
    - Can modify user info while system is running

Password Encoding

- Can encode passwords using a hash.
    - sha, md5, bcrypt
- Always secure passwords with salts.
    - Makes brute force attacks harder.

Other Authentication Options

- Implement a custom UserDetailsService .
    - Delegate to an existing User repository or DAO.
- LDAP
- X.509 Certificates
- JAAS Login Module
- Single-Sign-On
    - OAuth, SAML
    - SiteMinder, Kerberos
    - JA-SIG Central Authentication Service

Profile with Security Configuration.

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {

    protected void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests().antMatchers("/resources/**").permitAll();
    }
}

@Configuration
@EnableWebSecurity
@Profile("development") // use in-memory provider
public class SecurityDevConfig extends SecurityBaseConfig {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("hughie").password("hughie").roles("GENERAL");
    }
}
```

In-memory vs. database authentication.

```
// Either in-memory e.g., for testing...
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth
            .inMemoryAuthentication() ①
                    .withUser("hughie").password("hughie").roles("GENERAL").and() ②
                    .withUser("dewey").password("dewey").roles("ADMIN").and()
                    .withUser("louie").password("louie").roles("SUPPORT");
}

// ... or sourcing from a database
@Autowired DataSource dataSource;

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.
            jdbcAuthentication()  ③
            .dataSource(dataSource)
            .passwordEncoder(new StandardPasswordEncoder("sodium-chloride")); ④
}
```

① Adds a `UserDetailsManagerConfigurer`.

② User, password and supported roles.

③ Can customize queries using methods: `usersByUsernameQuery()`, `authoritiesByUsernameQuery()` , `groupAuthoritiesByUsername()`

④ SHA-256 encoded passwords with a salt.

18.5. Using Spring Security's Tag Libraries

Tag Library Declatation

- In JSP: ` <%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>`
- Facelet tags for JSF are also available.
  - You need to define and install them manually
  - See "Using the Spring Security Facelets Tag Library" in the Spring Webflow documentation.
  - Principal is available in SpEL: #{principal.username}.

SpringSecurity's Tag Library

- Display properties of the Authentication object
- Hide sections of output based on role.
  - Role must be prefixed ROLE_ here.

```
You are logged as:
<security:authentication property="principal.username"/>

<security:authorize access="hasRole('ROLE_MANAGER')"> ①
    TOP-SECRET INFORMATION
    Click <a href="/admin/deleteAll">HERE</a> to delete all records.
</security:authorize>
```

① Alternatively define this **centralized as intercept-URL**: `.antMatchers("/admin/*").hasAnyRole("MANAGER","ADMIN")`

18.6. Method security

General

- Spring Security uses AOP for security at the method level.
  - Annotations based on Spring annotations or JSR-250 annotations.
  - Java configuration to activate detection of annotations.
- Typically secure your services.
  - Do not access repositories directly, bypasses security (and transactions).

JSR-250 (Method Security). Not limited to roles. SpEL not supported.

```
@EnableGlobalMethodSecurity(jsr250Enabled=true) // on config class

...

import javax.annotation.security.RolesAllowed;

public class ItemManager {

    @RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})
    public Item findItem(long itemNumber) { ... }
}
```

Method security with SpEL. Use Pre/Post annotations.

```
@EnableGlobalMethodSecurity(prePostEnabled=true)

...

import org.springframework.security.annotation.PreAuthorize;

public class ItemManager {

     @PreAuthorize("hasRole('ROLE_MEMBER')")
    public Item findItem(long itemNumber) { ... }
}
```

18.7. Advanced security: Filters

Spring Security in a Web Environment

- SpringSecurityFilterChain is declared in `web.xml`.
- This single proxy filter delegates to a chain of Spring- managed filters..
  - Drive authentication.
  - Enforce authorization.
  - Manage logout.
  - Maintain SecurityContext in HttpSession.

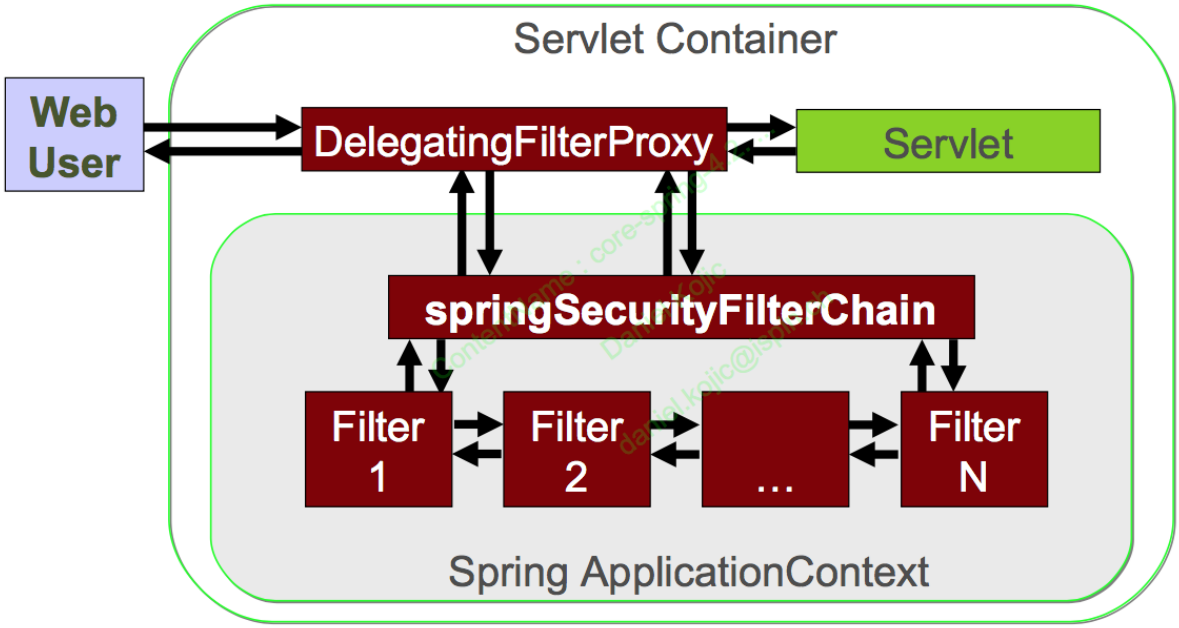


Figure 10. Web Security Filter Configuration.

Filter Chain

- With ACEGI Security 1.x
  - Filters were manually configured as individual <bean> elements.
  - Led to verbose and error-prone XML.
- Spring Security 2.x and 3.x
  - Filters are initialized with correct values by default.
  - Manual configuration is not required **unless you want to customize** Spring Security's behavior.

Custom Filter Chain

- Filter can be added to the chain.
  - Before or after existing filter.
  - `http.addFilterAfter ( myFilter, UsernamePasswordAuthenticationFilter.class );`
  - Just implement a `Filter` bean.
- Filter on the stack may be replaced by a custom filter
  - Replacement must extend the filter being replaced.
  - `http.addFilterAfter ( myFilter, UsernamePasswordAuthenticationFilter.class );`
  - `public class MySpecialFilter extends UsernamePasswordAuthenticationFilter {}`

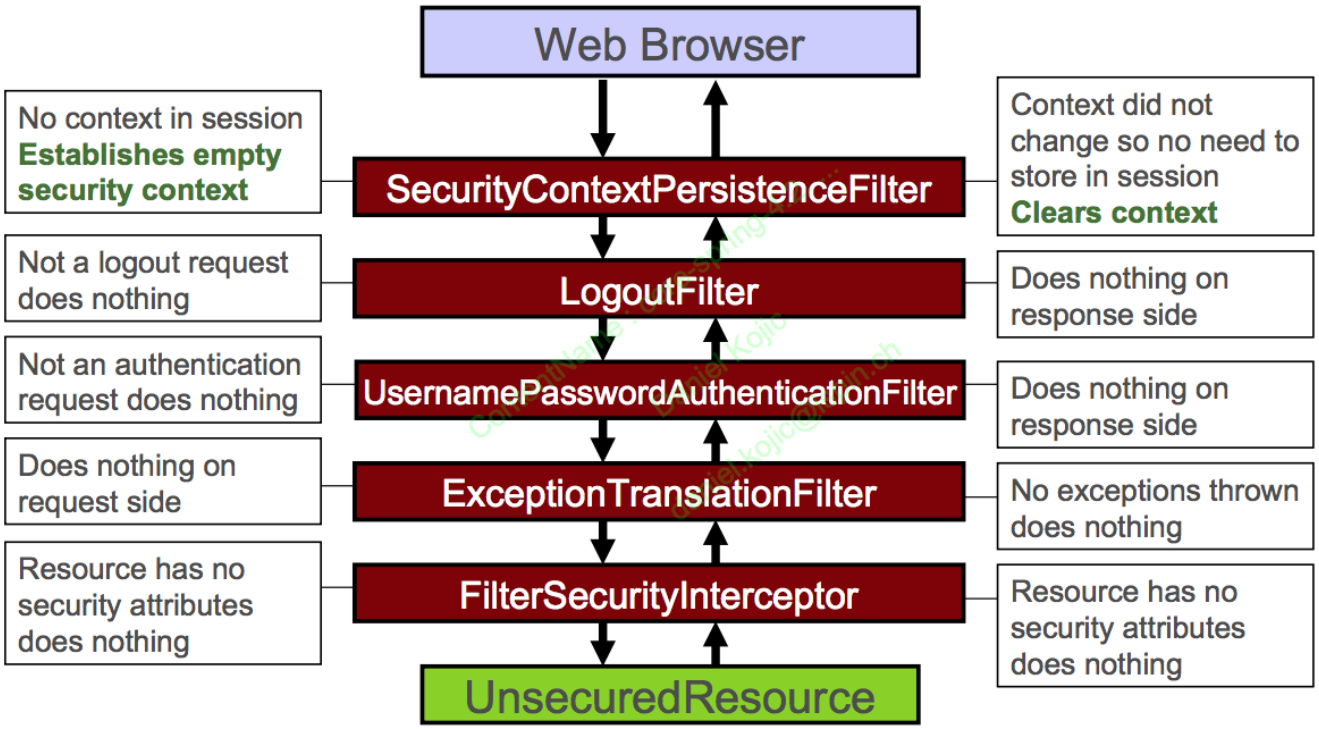


Figure 11. Access unsecured resource prior to login.

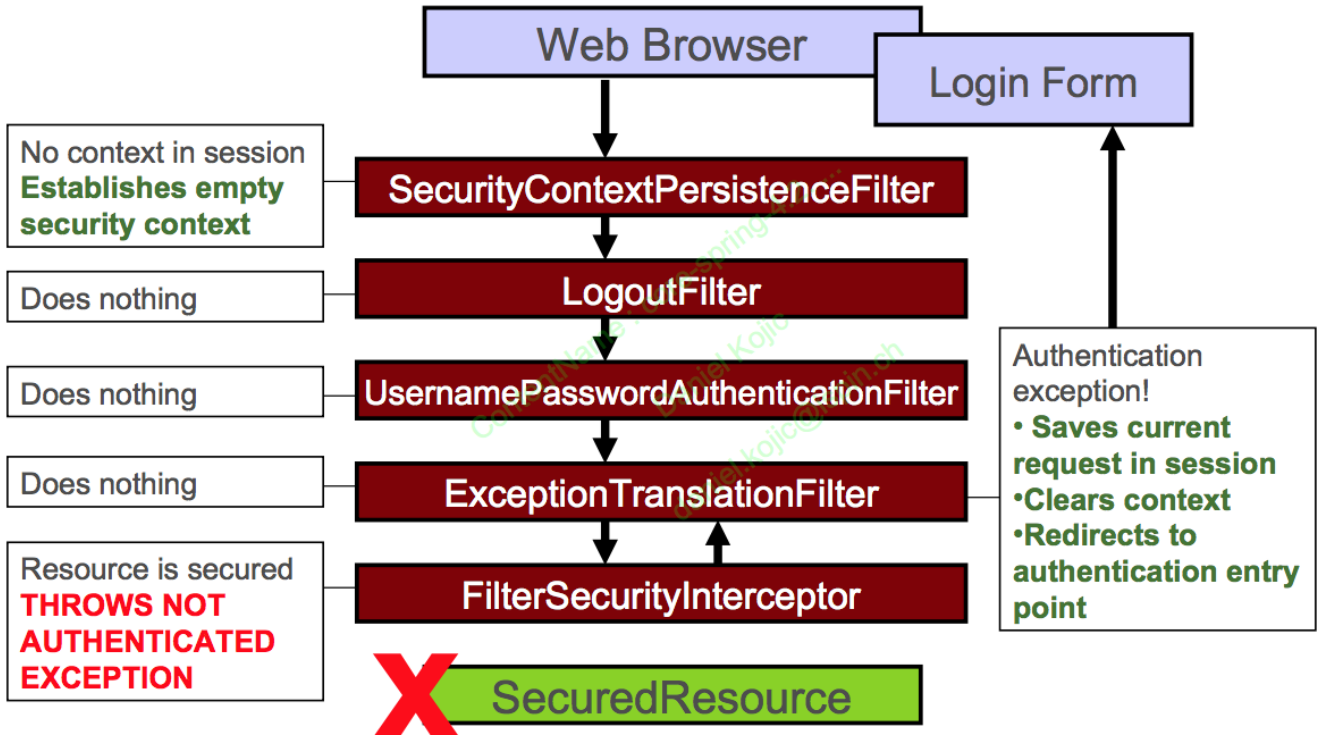


Figure 12. Access secured resource prior to login.

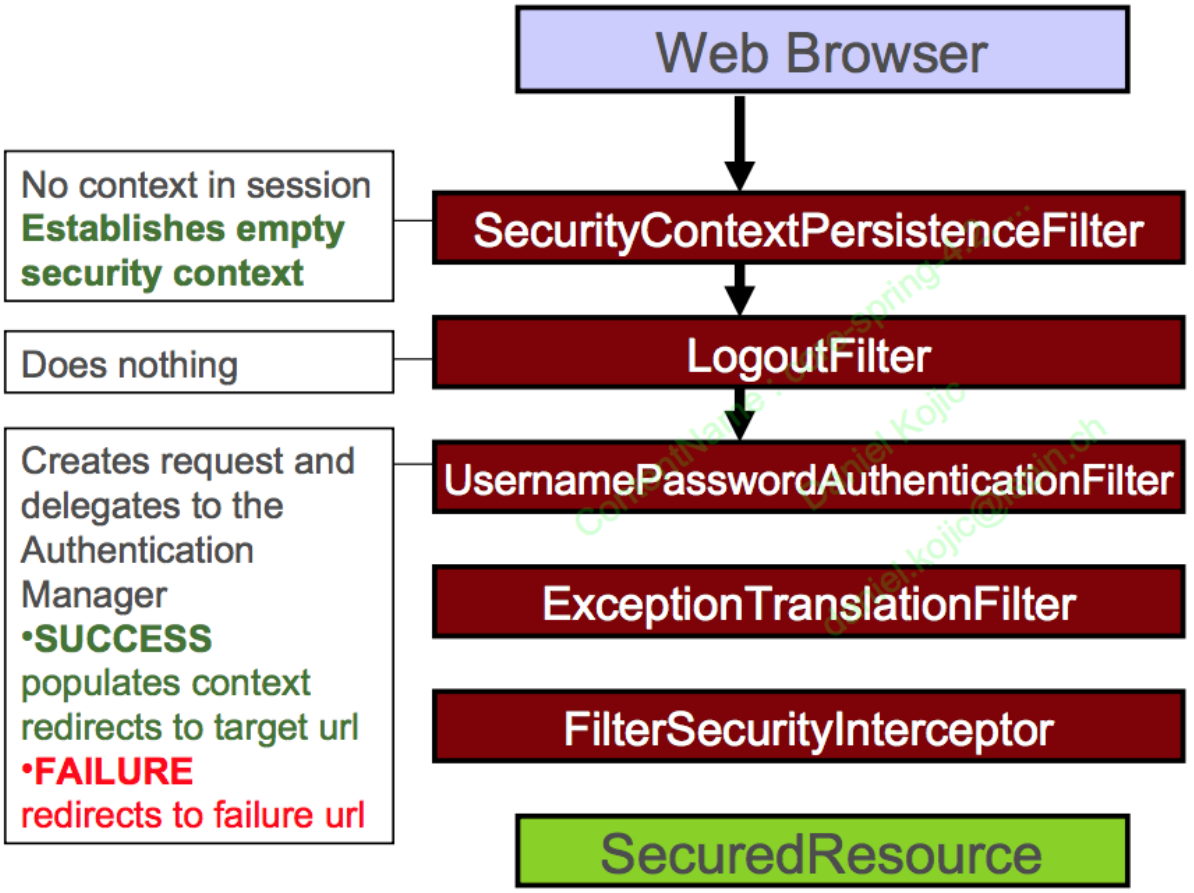


Figure 13. Submit Login Request.

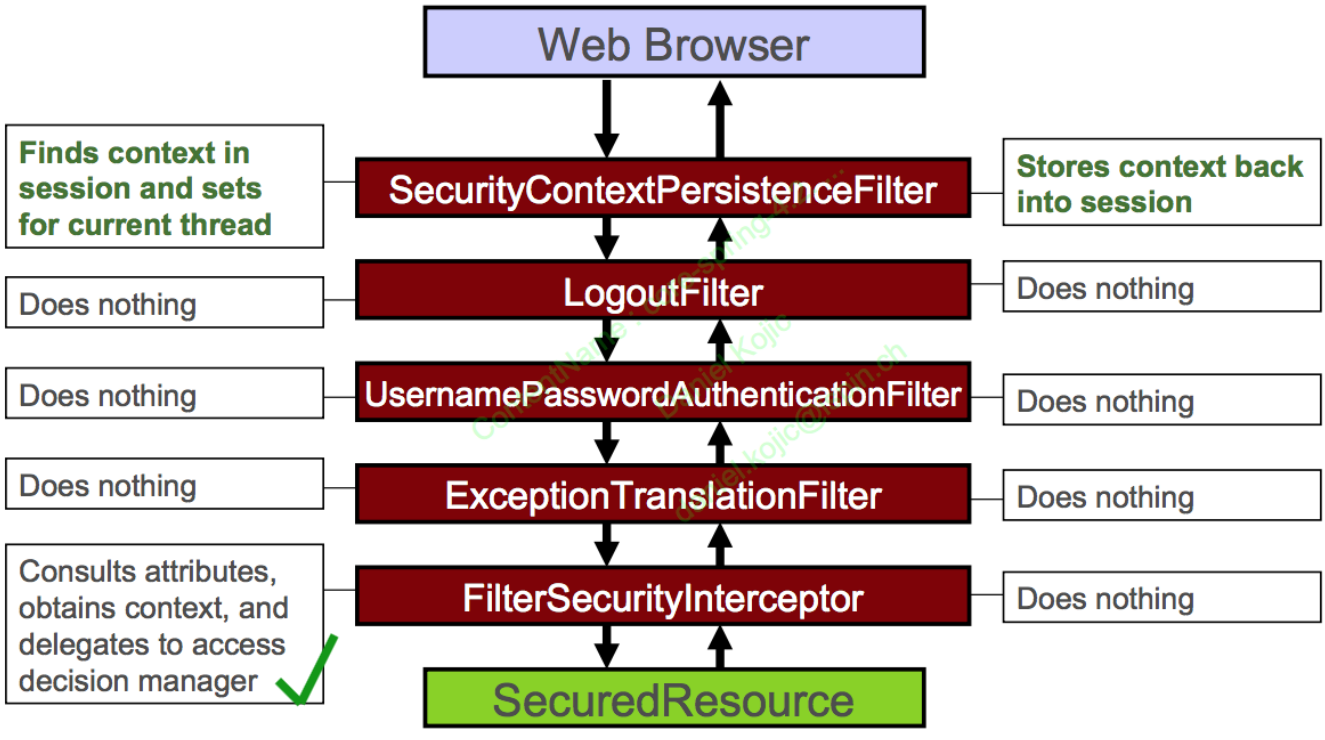


Figure 14. Access Resource With Required Role.

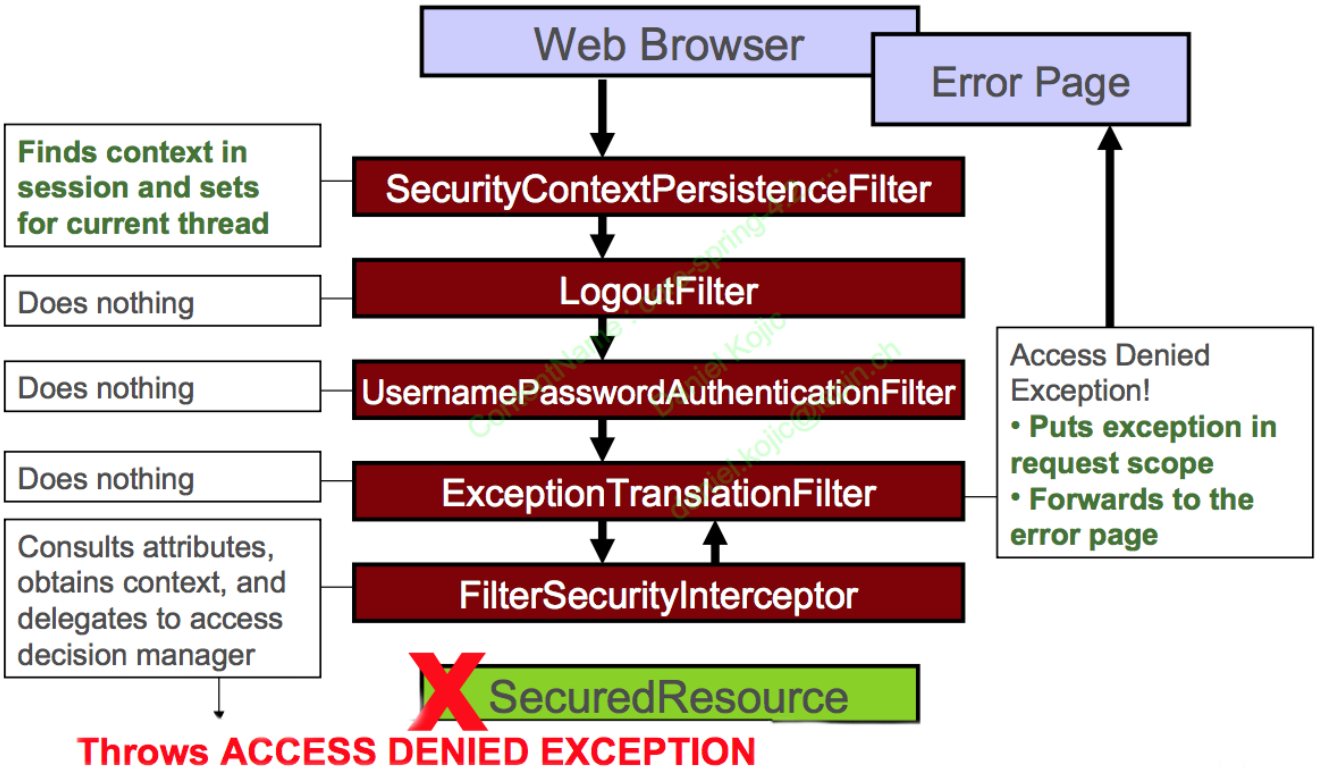


Figure 15. Access Resource Without Required Role.

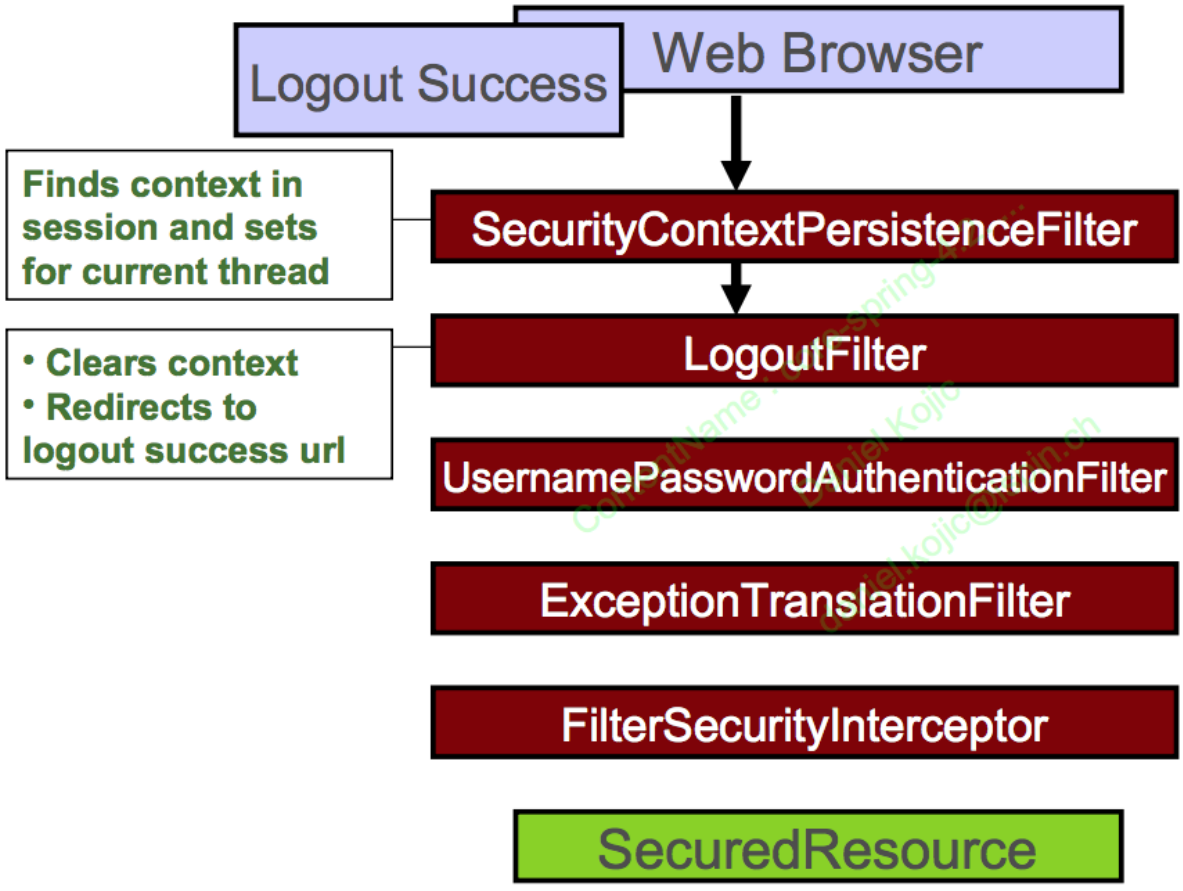


Figure 16. Submit Logout Request.

Table 1. Summary.

#	Filter Name	Main Purpose
1	SecurityContextIntegrationFilter	Establishes SecurityContext and maintains between HTTP requests (formerly: HttpSessionContextIntegrationFilter)
2	LogoutFilter	Clears SecurityContextHolder when logout requested
3	UsernamePassword AuthenticationFilter	Puts Authentication into the SecurityContext on login request (formerly: AuthenticationProcessingFilter)
4	Exception TranslationFilter	Converts SpringSecurity exceptions into HTTP response or redirect
5	FilterSecurity Interceptor	Authorizes web requests based on on config attributes and authorities

19. Microservices and Cloud embeded Systems

Building Cloud Native Applications.

19.1. What are Microservices?

Three Tier Architecture

- Single “monolithic” application that does everything.
- Single, large development team.
- Separate Ops, DBAs, Dev teams.
- All data in single relational database.

Microservice Features

- Multi-Language
  - Not all the services need to be in the same language / framework
- Polyglot Persistence
  - Each service uses the most suitable storage system.
- Stand-alone Development
  - Develop, Test and Deploy each service independently.
  - Separate teams, leverage Dev Ops.

Table 2. Trade-Offs.

Single App	Microservices
Easier to build	Harder to build

Single App	Microservices
Large process to deploy	Network Overheads
Ultimately more complex to enhance and maintain	Ultimately simpler to enhance and maintain
Scaling Up (bigger processors) limited	Scaling Out (more processes) easier

Core Spring Architectural Concepts

- Spring emphasizes
  - Loose Coupling
    - Applications are built from collaborating services (processes).
    - Similar to Service Oriented Architectures (SOA).
  - Tight Cohesion
    - An application (service) that deals with a single view of data.
    - Also known as “Bounded Contexts” (Domain-Driven Design).

Why Cloud, Why PaaS (Platform as a Service)?

- Deploying multiple processes is complicated.
  - Security, resilience, redundancy, load-balancing.
- A Cloud (PaaS) provides the necessary tools.
  - Natural fit for deploying a microservice-based system.
  - Application is the unit of deployment.
  - Application instances are the unit of scaling.
  - Start, stop and restart apps independently, on-demand.
  - Provide dynamic load-balancing, scaling and routing.

Cloud Native Applications

- Applications
  - Designed to run “in the cloud”.
  - In isolated, disposable containers.
    - Fast to scale-up and scale-down.
- Make no assumptions about the underlying infrastructure.
  - Local file-system transient.
  - Sessions lost on restart.
- Should design applications to suit this environment.
  - Use Twelve Factor Application design patterns.

Why Spring Boot?

- Faster to develop than traditional Spring.

- Java apps become as easy as Grails or Rails apps.
  - But with JVM robustness and scalability.
- Easy to incorporate other Spring modules.
  - As listed on previous slide.
- Spring Cloud requires Spring Boot.

19.2. Challenges and Implementation

Create a new Microservice App

- Start with Monolith
- As it grows:
  - Decompose into micro-service(s).
  - Enables separately manageable and deployable units.
  - Each can use own storage solution (polyglot persistence).

Decompose an existing Monolith

- Develop new functionality as microservice(s) around existing single-process application.
- Refactor existing monolith functionality into new microservice(s).
  - Refactor at service layer.
  - Monolith service communicates over an agreed protocol e.g., HTML with the new microservice.

19.3. Spring Cloud

19.3.1. Overview

- Building blocks for Cloud and Microservice applications.
  - Microservices Infrastructure.
    - Wraps up and makes available useful services.
    - Several based on other Open Source projects e.g., Netfix, HashiCorp's Consul
  - Cloud Independence.
    - Access cloud-specific information and services.
    - Support for Cloud Foundry, AWS and Heroku.
    - Or run without any cloud at all.
- Uses Spring Boot starters.
  - Requires Spring Boot to work.

Spring Cloud Projects

- IaaS Integration

- Dynamic Reconfiguration
- Service Discovery / Load Balancing
- Utilities
- Data Ingestion

Communication between Microservices

- Rest / JSON typically used.
  - How do Services find each other?
  - What happens if we run mutliple instances?

Registry Servers for Microservices

- Two popular open-source registry services.
  - Eureka (Netflix)
  - Consul.io (Vagrant)
- Spring cloud makes it easy to use either of or switch between them.

Spring Cloud Maven descriptor.

```
<parent>
    <groupId>org.springframework.cloud</groupId> // Parent
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Angel.SR3</version> // Consolidated set of releases.
</parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId> // Spring Cloud
            <artifactId>spring-cloud-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka-server</artifactId> // Eureka
regostry server.
        </dependency>
    </dependencies>
...
```

Building a Microservice System

1. Run a Discovery Service.
2. Run a Microservice.
   - Ensure it registers itself with the Discovery Service.
3. How do Microservice clients find the service?

- Inject a “smart” RestTemplate.
  - Spring performs service lookup for you.
  - Uses logical service names in URLs.

19.3.2. Eureka Service Discovery

Eureka application class.

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

Eureka configuration.

```
server:
    port : 8761
eureka:
    instance:
        hostname: localhost
    client:  # Not a client
        registerWithEureka: false
        fetchRegistry: false
```

19.3.3. Service Registration

- Each microservice declares itself a discovery-client.
  - Using `@EnableDiscoveryClient`.
  - Registers using its application name.

Service implementation.

```
@SpringBootApplication
@EnableDiscoveryClient
public class AccountsApplication {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Service discovery configuration.

```
spring:
    application:
        name: accounts-microservice ①
eureka:
    client:
        serviceUrl:
            defaultZone: http://localhost:8761/eureka/ ②
```

① Service name.

② Eureka server URL.

19.3.4. Service Discovery Client

Discovery client with a smart Rest-Template.

```
@SpringBootApplication
@EnableDiscoveryClient
public class FrontEndApplication {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public AccountManager accountManager() {
        return new RemoteAccountManager();
    }
}

@Service
public class RemoteAccountManager {

    // Spring injects a smart service-aware template
    // configured with RibbonHttpRequestClient to do a
    // load-balanced lookup
    @Autowired
    @LoadBalanced
    RestTemplate restTemplate;

    public Account findAccount(String id) {
        // Fetch data
        return restTemplate.getForObject(
                "http://accounts-microservice/accounts/{id}", // service name
                Account.class,
                id);
    }
}
```

Intelligent Routing

- Spring Cloud automatically integrates two Netflix utilities.
  - “Eureka” service-discovery.
  - “Ribbon” load-balancer.
- End result
  - Determines the best available service to use (when there are multiple instances of a microservice).
  - Just inject the load-balanced RestTemplate.
  - Automatic lookup by logical service-name.

NOTE

Check out

- Project Homepage
- Matt Stine’s presentation from CF Summit
- Spring Blog Article
- Spring Cloud/Boot Demos

19.4. Summary

- After completing this lesson, you should have learnt:
  - What is a Microservice Architecture?
  - Advantages and Challenges of Microservices.
  - Implementation using Spring Cloud projects.