Spring:

Inversion of Control: using interface
Dependency Injection : using helper objects
Two types:
Constructor: constructor-arg
Setter : property (name same as method)
In setter injection, we can injection literal values or using property file.

LifeScope:
*Singleton: shared single instance of the bean.
*Prototype: new instance for each container request.
BeanLifeCycleHooks: init-method; destroy-method
(note: these method don't return any value and doesnot take arguments)
For "prototype" scoped beans, Spring does not call the destroy method.

Annotation:

@Component

Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

If nothing is specified it takes classname with starting letter as lower case.

@Autowired

Spring @Autowired annotation is used for automatic injection of beans. Spring @Qualifier annotation is used in conjunction with Autowired to avoid confusion when we have two of more bean configured for same type

*Constructor injection-
As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean only defines one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.
*Setter injection/Method injection- using methods
*Field injection- directly on fields(uses java reflection)

@Qualifier

However, for the special case of when BOTH the first and second characters of the class name are upper case, then the name is NOT converted.
For the case of RESTFortuneService

RESTFortuneService --> RESTFortuneService

No conversion since the first two characters are upper case.

Example for using Qualifier in constructor:

```
@Autowired
   public TennisCoach(@Qualifier("randomFortuneService") FortuneService
theFortuneService) {
      System.out.println(">> TennisCoach: inside constructor using @autowired and
@qualifier");
```

```
            fortuneService = theFortuneService;
        }
```

*@Scope("prototype")
*@PostConstruct
*@PreDestroy
@PreDestroy and @PostConstruct are alternative way for bean initMethod and destroyMethod. It can be used when the bean class is defined by us.

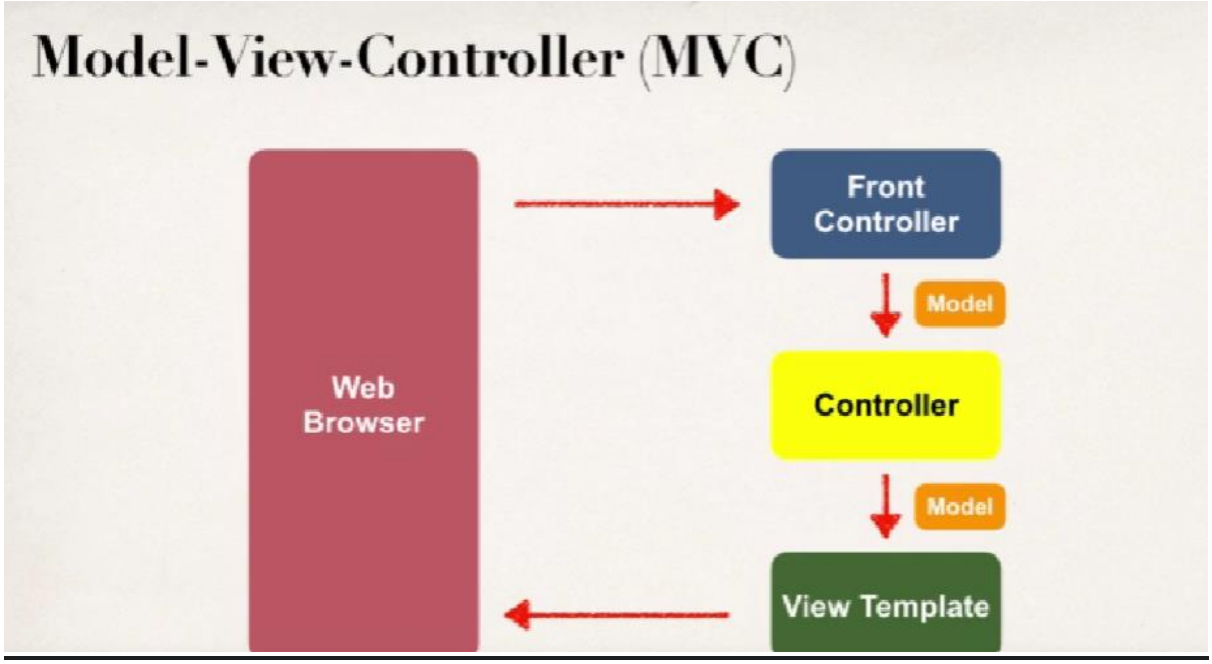
To config with no xml, Configuration class uses
***@Configuration**
@Configuration: Used to indicate that a class declares one or more @Bean methods. These classes are processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

***@ComponentScan**
Configures component scanning directives for use with @Configuration classes. Here we can specify the base packages to scan for spring components.

***@Bean** – to define beans
Indicates that a method produces a bean to be managed by the Spring container. This is one of the most used and important spring annotation. @Bean annotation also can be used with parameters like name, initMethod and destroyMethod.

***@PropertySource** – to link the property file
provides a simple declarative mechanism for adding a property source to Spring's Environment. There is a similar annotation for adding an array of property source files i.e @PropertySources.

Spring MVC(Model-View-Controller)



Front controller also known as DispatcherServelet
Part of Spring Framework
Already built by Spring Dev team.

Controller:
Handles business logic
As in handles the request, Store and retrieve data, place data in model
Send to appropriate template

Model:
Contains your data.(Data can be any java object/ collection)
Store and retrieve data via backend systems ;database, web service, or spring bean.

View:
Display the data.

*@Controller
*@RequestMapping
*@RequestParam
*@ModelAttribute

Validation:
@Size
@NotNull
@Valid
@InitBinder
@Min
@Max
@Pattern
Custom Validation:
@interface
@Constraint
@Target
@Retention
ConstraintValidator interface for class

Hibernate:

Minimizes JDBC code
Handles all of low level SQL
Object to Relational Mapping(ORM)
CRUD operation

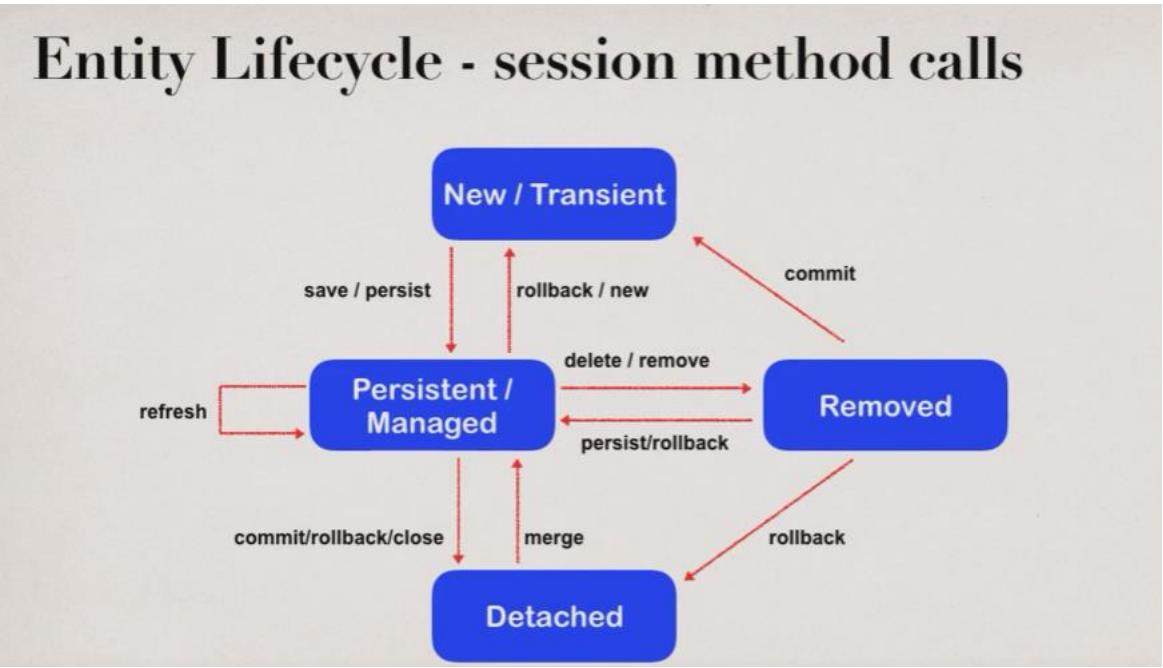
```
                                          _________
YOUR   =>                            =>  |         |
JAVA          Hibernate    JDBC          | Database |
APP    <=                            <=  |_________|
```

Connection interface and DriverManager for testing connection to DB;
Hibernate cfg.xml for configuring connection to DB.

@Entity
@Table
@Column
@Id
SessionFactory and Session interfaces .
@GeneratedValue

save()-create
get()- read with Id
createQuery()- to write query
list()- to change to list
executeUpdate()- to update to sql
Cascade – apply same operations to related entities (but cascade delete depends on use case)
Eager – will retrieve everything
Lazy – will retrieve on request

Entity LifeCycle:



Cascade Type:

CascadeType.PERSIST : In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted.

CascadeType.MERGE : In this cascade operation, if the parent entity is merged then all its related entity will also be merged.

CascadeType.REFRESH : In this cascade operation, if the parent entity is refreshed then all its related entity will also be refreshed.

CascadeType.REMOVE : In this cascade operation, if the parent entity is removed then all its related entity will also be removed

CascadeType.DETACH : In this cascade operation, if the parent entity is detached then all its related entity will also be detached.

CascadeType.ALL : cascade type all is shorthand for all of the above cascade operations.

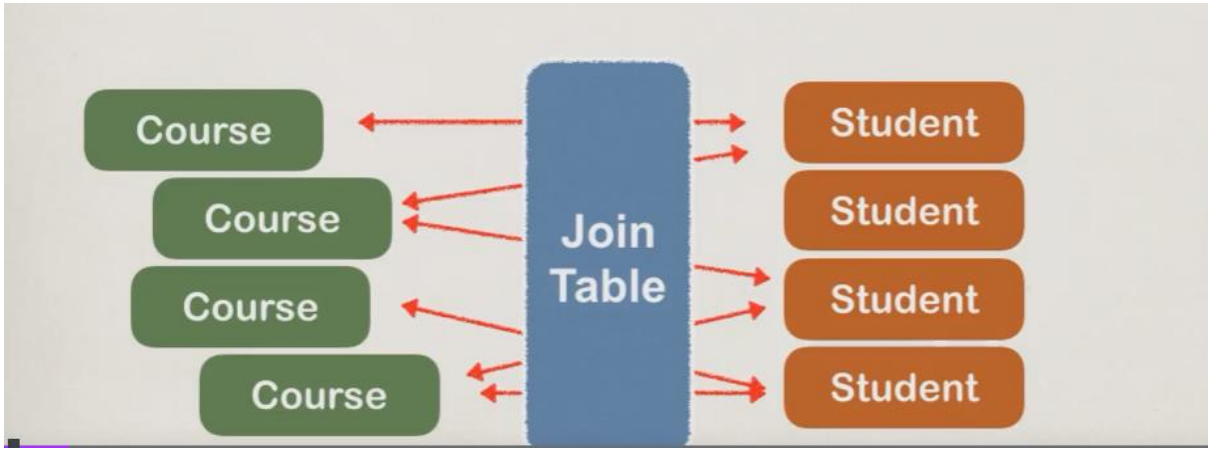
@OneToOne
mappedBy-field in other class
@JoinColumn(foreign key in other table)
@ManyToOne
@OneToMany

Prefer Lazy Loading Over Eager Loading

Default Fetch Types

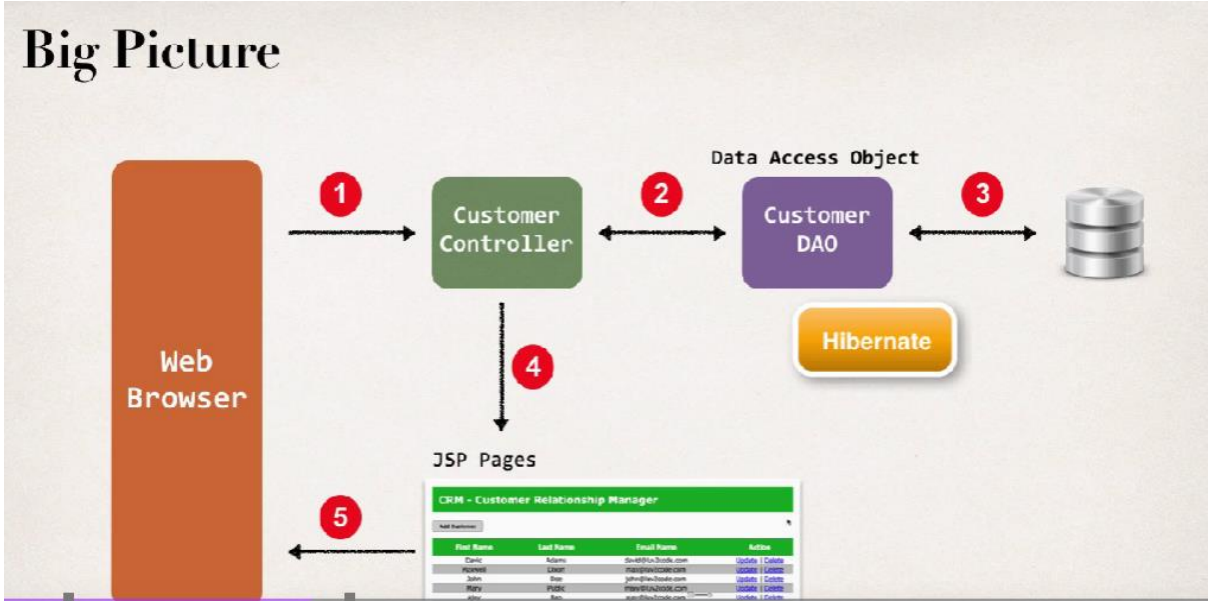
Mapping	Default Fetch Type
@OneToOne	FetchType.EAGER
@OneToMany	FetchType.LAZY
@ManyToOne	FetchType.EAGER
@ManyToMany	FetchType.LAZY

Lazy fetchtype exception:
Option 1: get before session is closed.
Option2: use HQL.



@ManyToMany
@JoinTable
JoinColumn
InverseColumn

Spring MVC + Hibernate:

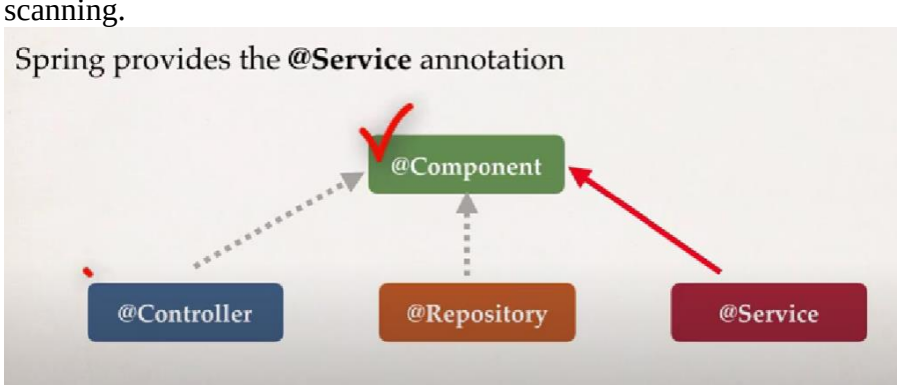


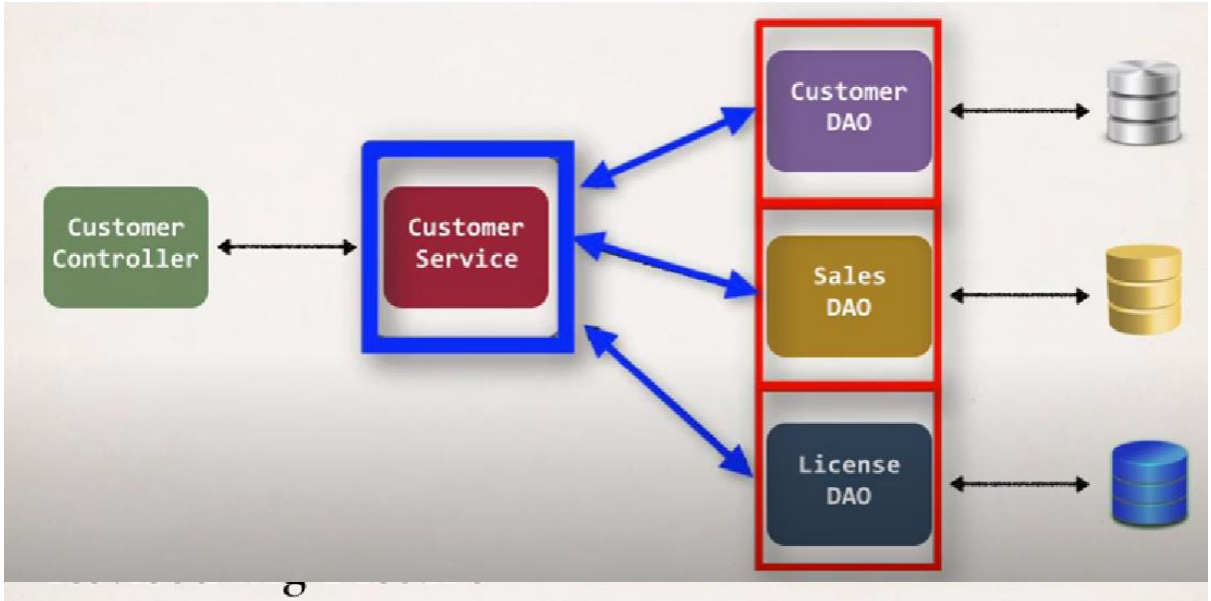
The Data Access Object is basically an object or an interface that provides access to an underlying database or any other persistence storage.

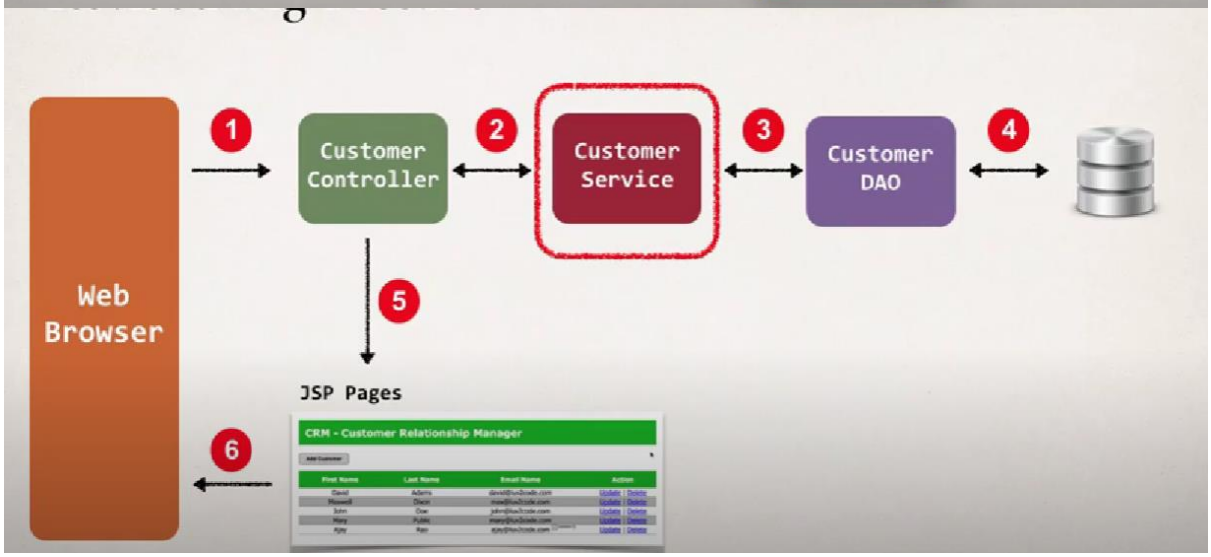
@Transactional- session (mostly used in service layer)

@Respository-applied to DAO implementation
Indicates that an annotated class is a “Repository”. This annotation serves as a specialization of @Component and advisable to use with DAO classes.

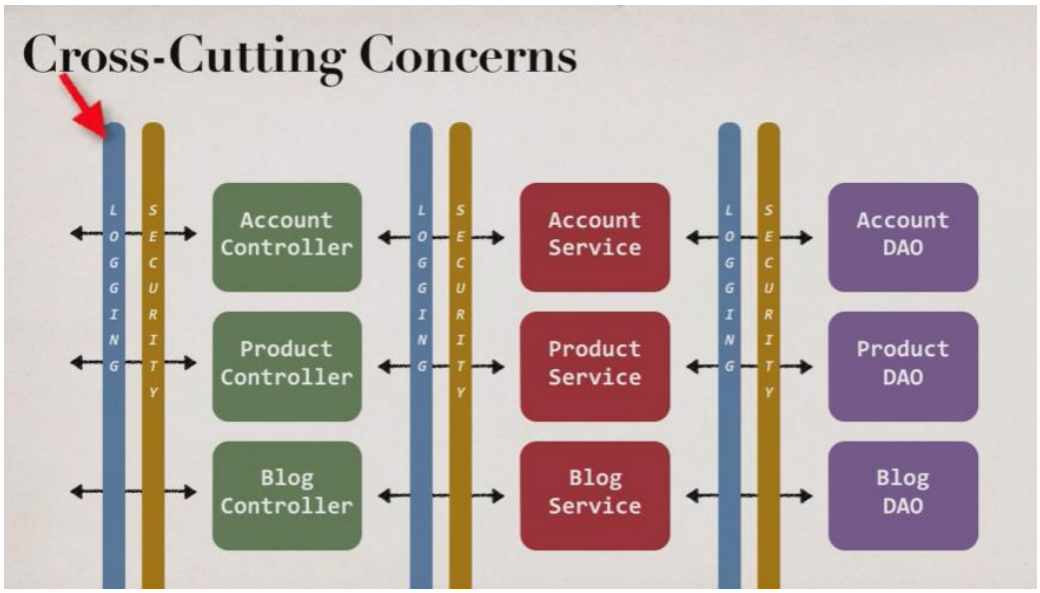
@GetMapping
@PostMapping
@Service
Indicates that an annotated class is a “Service”. This annotation serves as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.

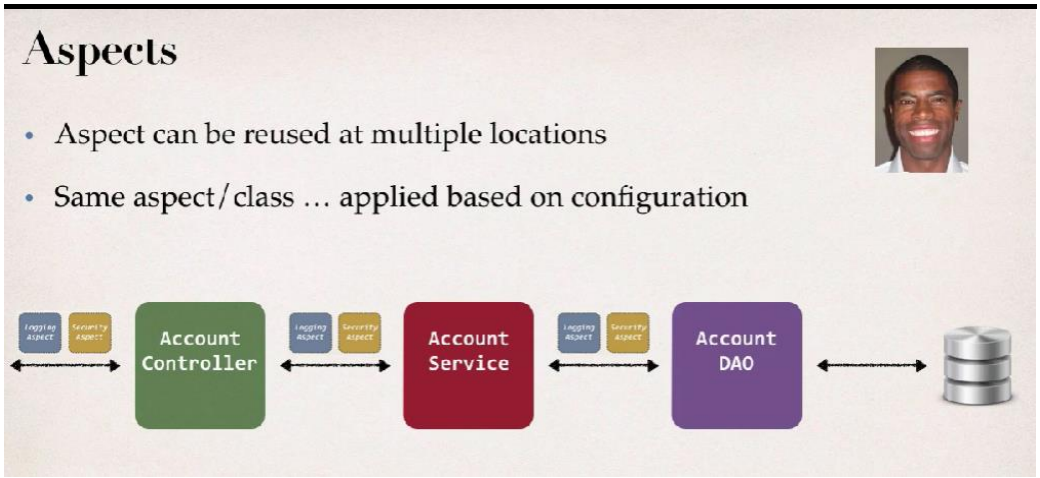


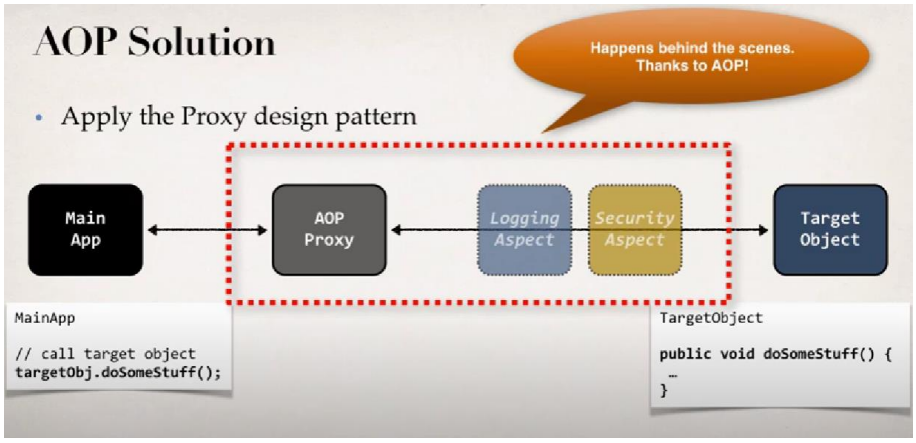

Customer Controller
Customer Service
Customer DAO
Sales DAO
License DAO


Web Browser
1
Customer Controller
2
Customer Service
3
Customer DAO
4
5
JSP Pages
6
CRM - Customer Relationship Manager

AOP(Aspect Oriented Programming):







AOP Terminology

- **Aspect**: module of code for a cross-cutting concern (logging, security, …)
- **Advice**: What action is taken and when it should be applied
- **Join Point**: When to apply code during program execution
- **Pointcut**: A predicate expression for where advice should be applied

Advice Types

- **Before advice**: run before the method
- **After finally advice**: run after the method (finally)
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **Around advice**: run before and after method

Comparing Spring AOP and AspectJ

- Spring AOP only supports
  - Method-level join points
  - Run-time code weaving (slower than AspectJ)
- AspectJ supports
  - join points: method-level, constructor, field
  - weaving: compile-time, post compile-time and load-time

@Configuration
@EnableAspectJAutoProxy
@ComponentScan
@Aspect
@Before

Pointcut Expression Language

```
execution(modifiers-pattern? return-type-pattern declaring-type-pattern?
          method-name-pattern(param-pattern) throws-pattern?)
```

- The pattern is optional if it has "?"

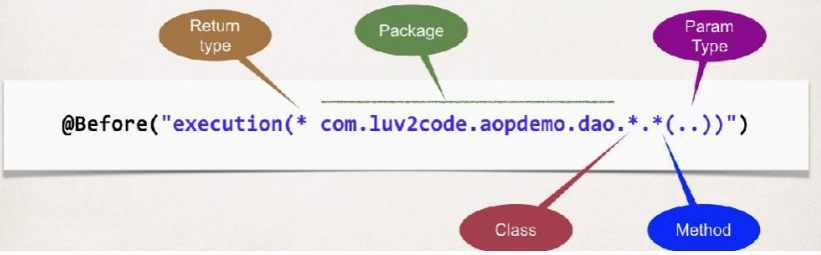
Parameter Pattern Wildcards

- For param-pattern
  - `()` - matches a method with no arguments
  - `(*)` - matches a method with one argument of any type
  - `(..)` - matches a method with 0 or more arguments of any type

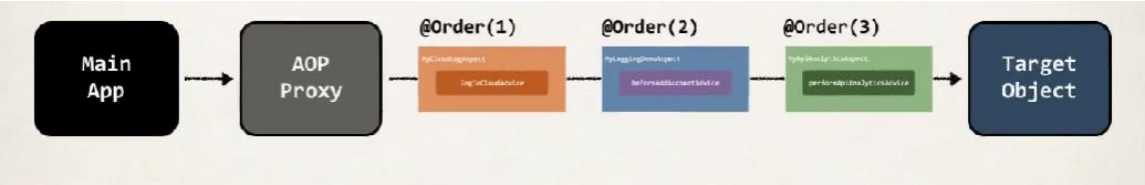
Package - Pointcut Expression Examples

Match on methods in a package

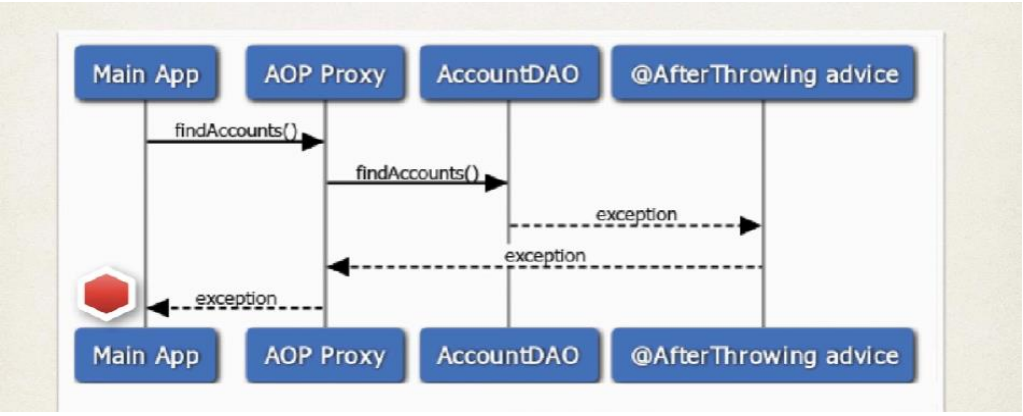
- Match any method in our DAO package: **com.luv2code.aopdemo.dao**



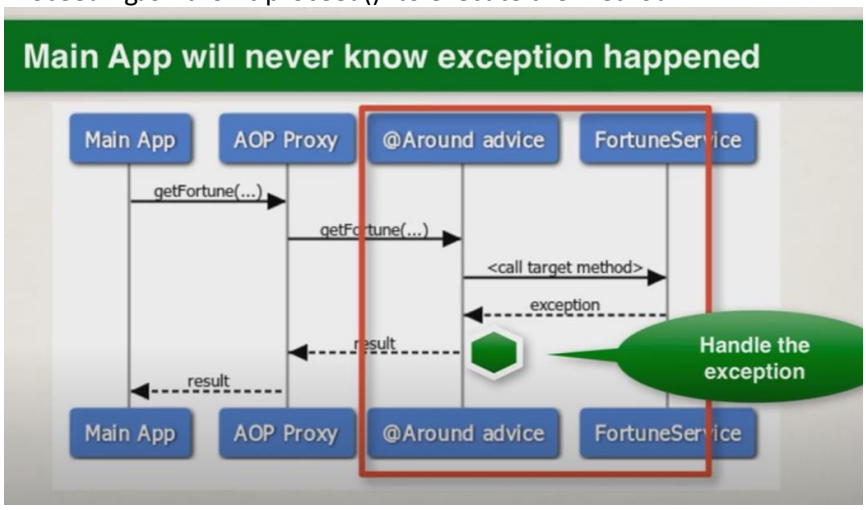
@Pointcut-expression
@Order-lower number has higher precedence



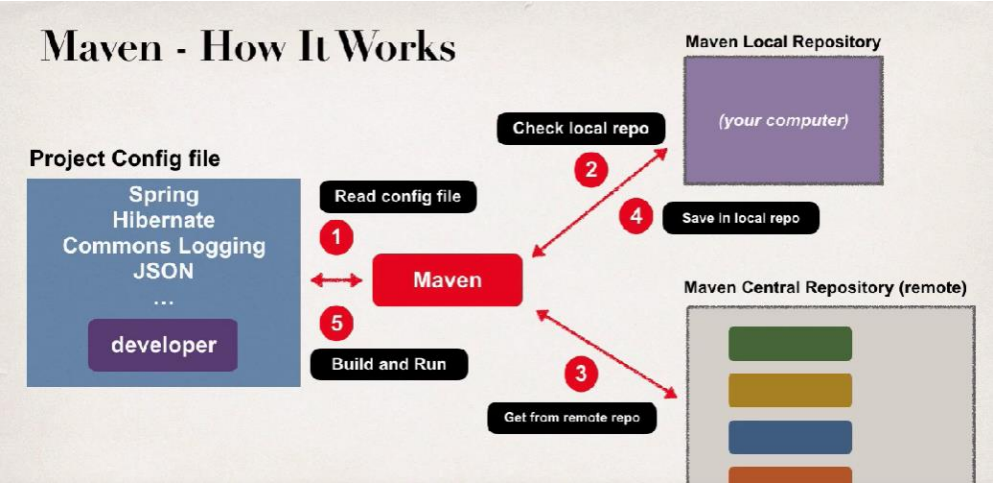
JoinPoint
MethodSignature
@AfterReturning(pointcut="", returning="")
@AfterThrowing(pointcut="", throwing="")

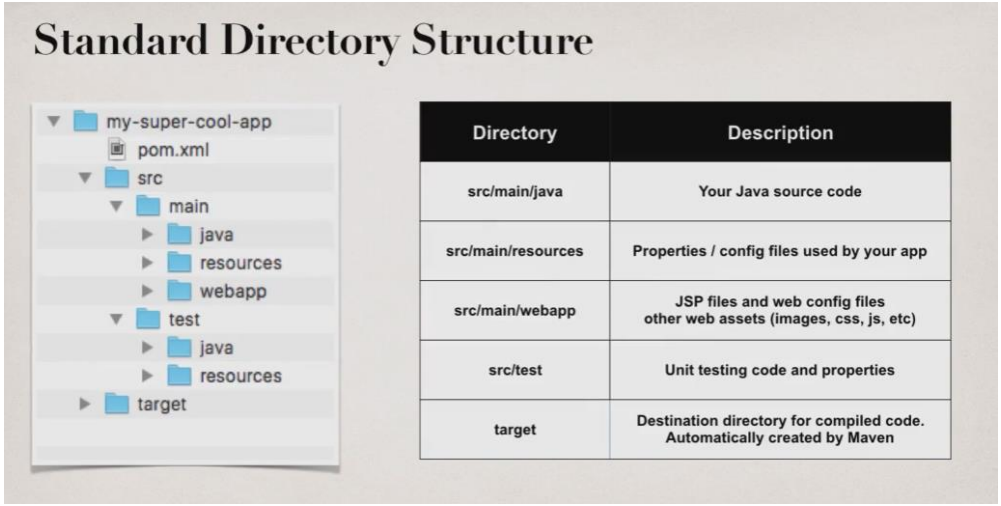


@After – will run for success or failure (finally)
@After will execute before@AfterThrowing
@Around
ProceedingJointPoint.proceed()- to execute the method

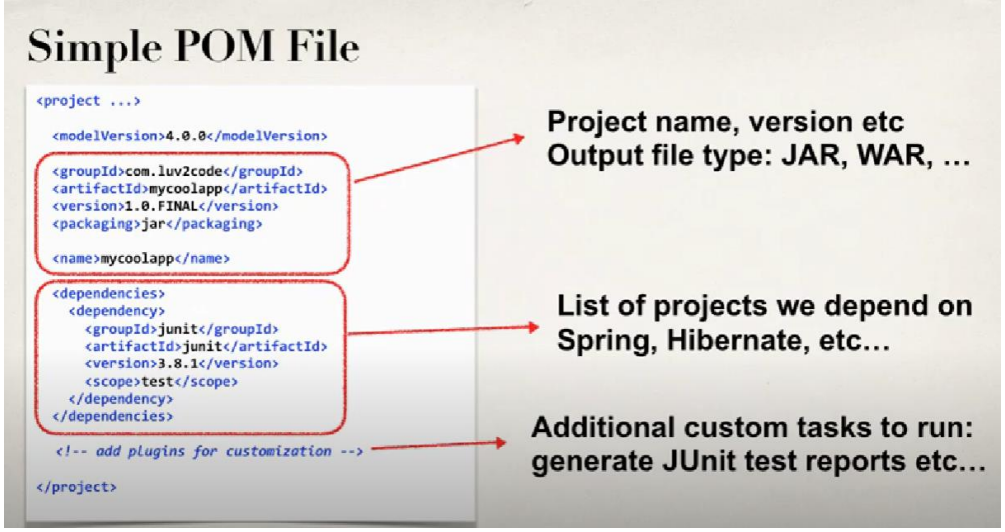


Maven:

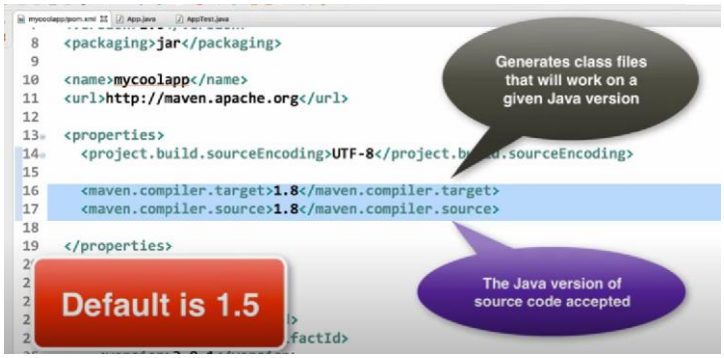




POM file (Project Object Model): Configuration file for project
-project meta data
-dependencies
-plugins



Project Coordinates uniquely identifies a project(GAV)
-groupId -artifactId -version
Archetypes can be used to create new Maven project. It contains template files for the given maven project.

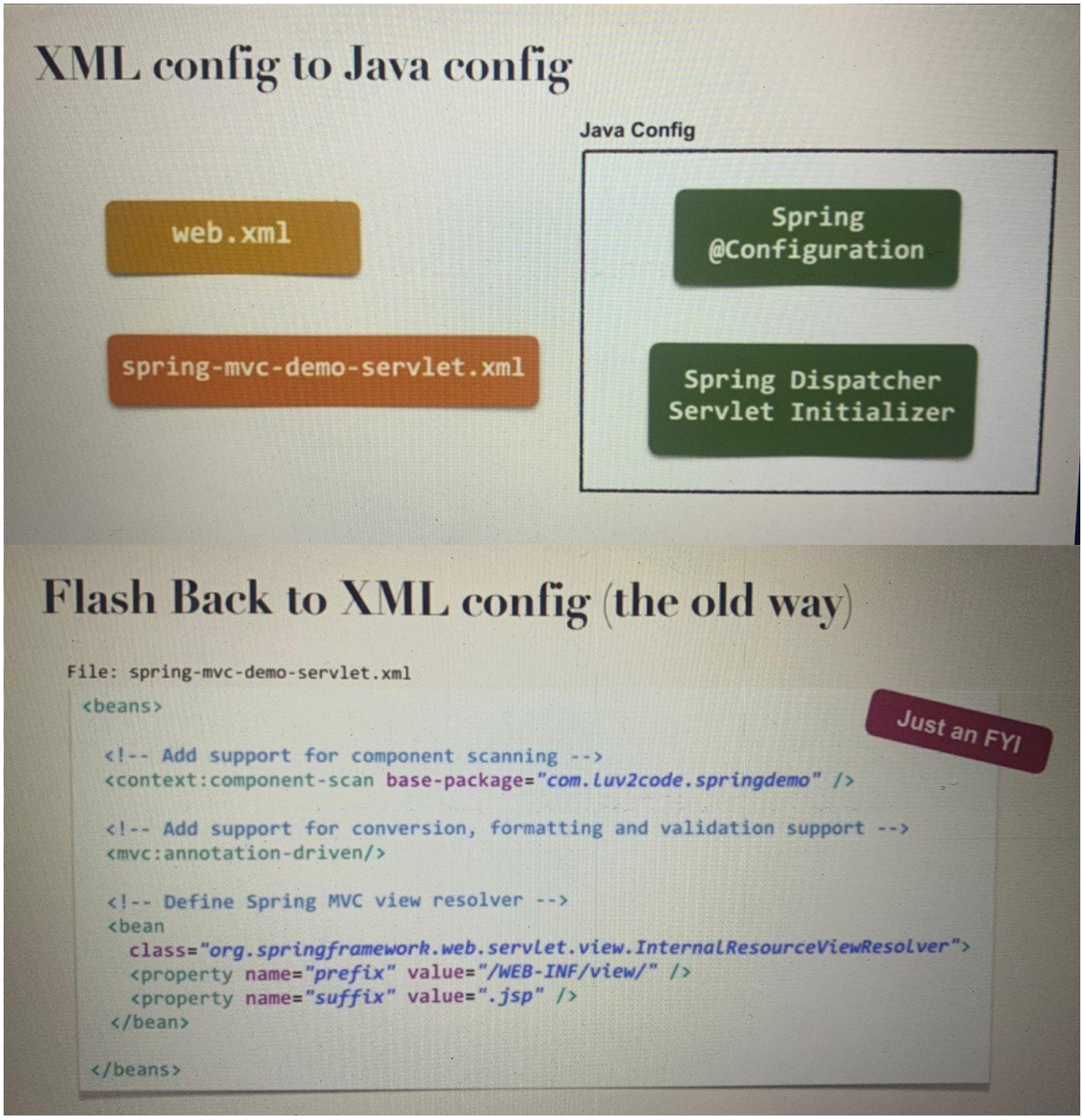


Maven Central repository(remote) requires internet.

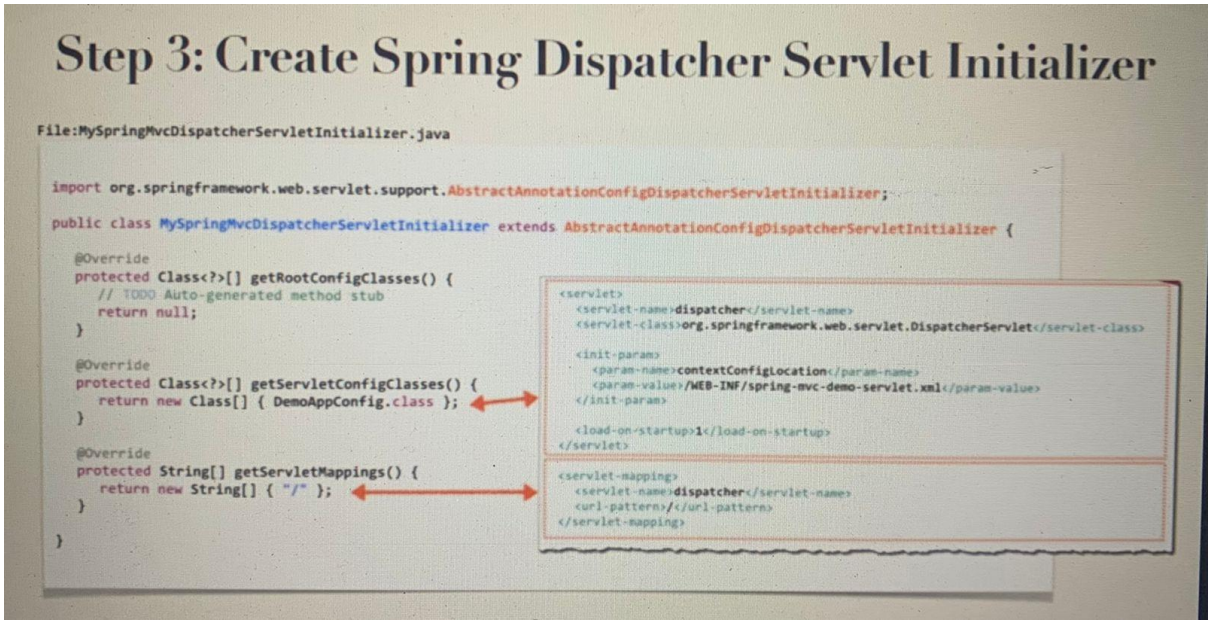
- Located on developer's computer
  - MS Windows: `c:\Users\<users-home-dir>\.m2\repository`
  - Mac and Linux: `~/.m2/repository`

Spring Security:

No XML in Spring MVC:

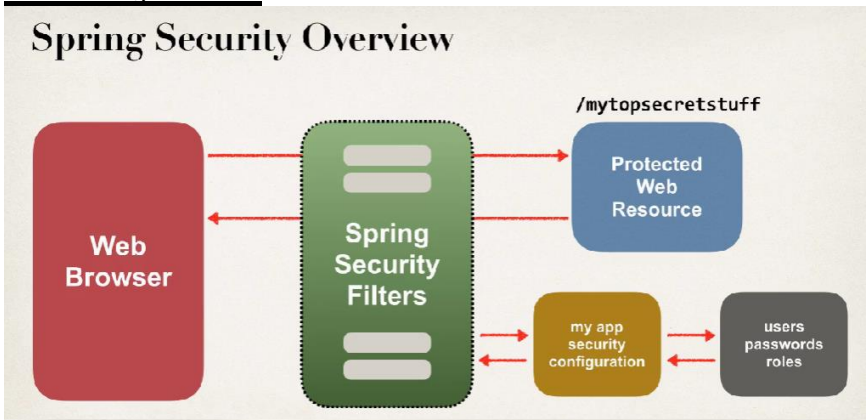


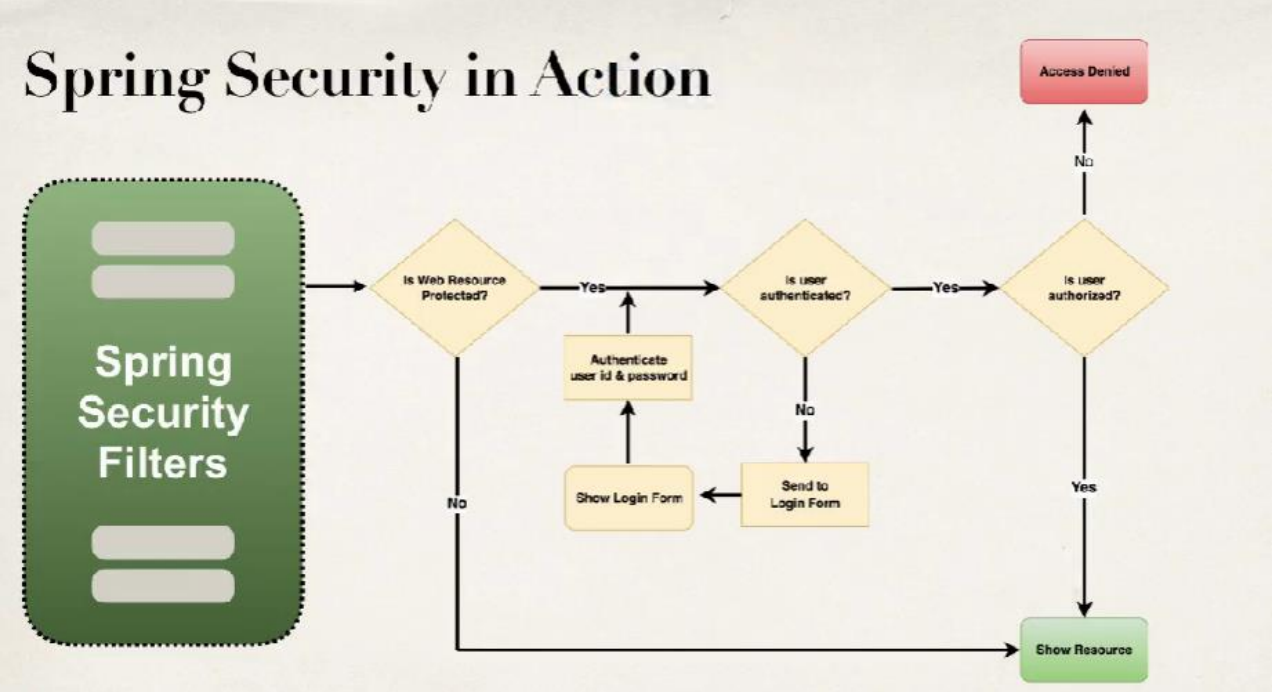
@EnableWebMvc - Provides similar support to <mvc:annotation-driven/> in XML.
Processing of @Controller classes and @RequestMapping etc



Makes sure your code is automatically detected. Your code is used to initialize the servlet container.

Security Part:



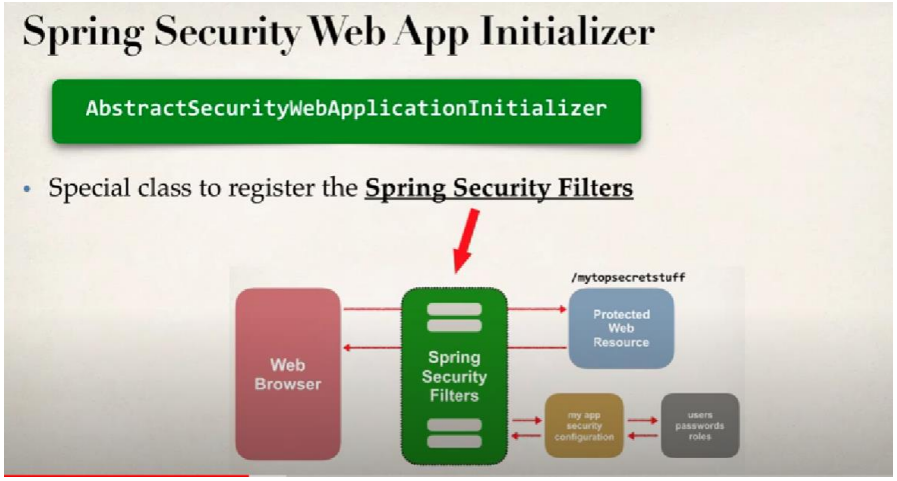


Provides two level security authentication and authorization.

Config Class:
@Configuration
@EnableWebMvc
@ComponentScan(basePackages=)
@Bean- return ViewResolver , InternalResourceViewResolver

MvcDispatcherServletInitailizer class:
AbstractAnnotationConfigDispatcherServletInitializer

Security Filter:



WebSecurityConfigureAdapter- security config file
@Configuration
@EnableWebSecurity

Adding csrf token:
<input type="hidden" name="${_csrf.parameterName}"
value="$(_csrf.token}"/>
<form:form> - it adds csrf token automatically.
Spring Security JSP tags

```
User: <security:authentication property="principal.username" />
<br><br>
Roler: <security:authentication property="principal.authorities" />
```

Step 3: Restricting Access to Roles

Any role in the list, comma-delimited list

```
antMatchers(<< add path to match on >>).hasAnyRole(<< list of authorized roles >>)
```

"ADMIN", "DEVELOPER", "VIP", "PLATINUM"

Content based on Roles:

```
<security:authorize access="hasRole('MANAGER')">

   <!-- Add a link to point to /leaders ... this is for the managers -->

   <p>
      <a href="${pageContext.request.contextPath}/leaders">Leadership
Meeting</a>
      (Only for Manager peeps)
   </p>

</security:authorize>
```

Step 4: Define DataSource in Spring Configuration

```
@Autowired
private Environment env;

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

@Bean
public DataSource securityDataSource() {

  // create connection pool
  ComboPooledDataSource securityDataSource = new ComboPooledDataSource();

  // set the jdbc driver
  try {
    securityDataSource.setDriverClass(env.getProperty("jdbc.driver"));
  }
  catch (PropertyVetoException exc) {
    throw new RuntimeException(exc);
  }

  ...
```

Read db configs

```
#
# JDBC connection properties
#
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_security_demo?useSSL=false
jdbc.user=springstudent
jdbc.password=springstudent
```

Step 4: Define DataSource in Spring Configuration

```
...

// for sanity's sake, let's log url and user ... just to make sure w
logger.info(">>>> jdbc.url=" + env.getProperty("jdbc.url"));
logger.info(">>>> jdbc.user=" + env.getProperty("jdbc.user"));

// set database connection props
securityDataSource.setJdbcUrl(env.getProperty("jdbc.url"));
securityDataSource.setUser(env.getProperty("jdbc.user"));
securityDataSource.setPassword(env.getProperty("jdbc.password"));

// set connection pool props
securityDataSource.setInitialPoolSize(Integer.parseInt(env.getProperty("connection.pool.initialPoolSize")));
securityDataSource.setMinPoolSize(Integer.parseInt(env.getProperty("connection.pool.minPoolSize")));
securityDataSource.setMaxPoolSize(Integer.parseInt(env.getProperty("connection.pool.maxPoolSize")));
securityDataSource.setMaxIdleTime(Integer.parseInt(env.getProperty("connection.pool.maxIdleTime")));

return securityDataSource;
}
```

```
#
# Connection pool properties
#
connection.pool.initialPoolSize=5
connection.pool.minPoolSize=5
connection.pool.maxPoolSize=20
connection.pool.maxIdleTime=3000
```

Step 5: Update Spring Security to use JDBC

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

   @Autowired
   private DataSource securityDataSource;

   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {

      auth.jdbcAuthentication().dataSource(securityDataSource);

   }

   ...

}
```

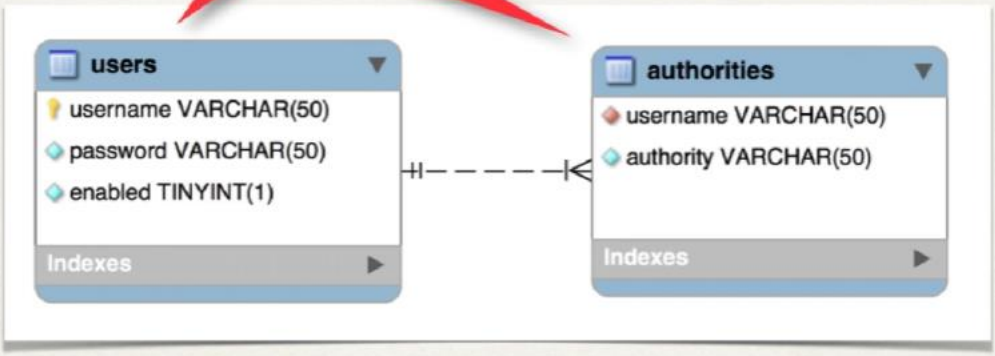
Inject our data source that we just configured

No longer hard-coding users :-)

Tell Spring Security to use JDBC authentication with our data source

Default Spring Security Database Schema

Need exact table names and column names



@PropertySource – reads property file in src/main/resources.
To add JDBC authentication

```
@Autowired
private DataSource securityDataSource;


@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {

   // use jdbc authentication ... oh yeah!!!

   auth.jdbcAuthentication().dataSource(securityDataSource);

}
```