**JPA Annotations:**

@Entity should be there.

No Arg Constructor should be there with public or protected.

@ID for primary key

Primary Key: Generation types:

GenerationType.AUTO = it will pick appropriate strategy of database.

GenerationType.IDENTITY = Assign primary keys using database identity column

GenerationType.SEQUENCE = Assign primary keys using a database sequence

GenerationType.Table = Assign primary keys using an underlying database table for uniqueness.

***Instructor version:***

*Create custom implementation of org.hibernate.id.IdentifyGenerator and override method public Serializable generate(..)*

Custom ID generation: [Spring Data JDBC - How to use custom ID generation](https://spring.io/blog/2021/09/09/spring-data-jdbc-how-to-use-custom-id-generation)

**How does hibernate work? – Hibernate needs entity manager which interacts with data source to do DB operation.**

**Entitymanager and Data source will be created automatically by spring boot.**

**Steps:**

1. **DAO Interface**
2. **DAO Interface implementation**
3. **@Transactional**
4. **@Repository – provides CRUD operation and search, converts checked exceptions to unchecked exceptions**

**Update auto increment:**

**Alter table student auto\_increment 3000; //sets initial value to 3000**

**Reseting auto increment:**

**Truncate student;**

**Entity manager provides methods for finding, persisting, removing , for creating query etc.**

**Query:**

JPA has JPA query language (JPQL).

**JPQL has Like, where, OR , And etc like SQL language. It provides named parameters too like JDBC prepared statement.**

**When I was learning 2012, we will be getting data source from spring. We need to create JDBC template and then need to execute plain SQL queries with respect to preparedstatement or statement. All of them are removed with @Repository and JPQL.**

**We will be creating TypedQuery here as well. But it is more like ORM than SQL query.**

**Updating Object:**

**a) Find Object first and then use setter methods to update and call entitymanager.merge() to update**

**b) Using query to update all rows blindly too using TypedQuery or by filtering set of valid rows.**

**Deleting Object: (remove)**

**Delete by find by id or using typed query to delete all.**

**Creating Tables from Java Code:**

**Spring.jpa.hibernate.ddl-auto**

|  |  |
| --- | --- |
| **Property Value** | **Desc** |
| None | No action will be performed |
| Create-only | Database table only will be created |
| Drop | Database table will be dropped |
| Create | Database tables are dropped followed by database creation |
| create-drop | Database table are dropped followed by database creation. On application shutdown, drop the database tables. (good for unit testing) |
| Validate | Validates database table schemas |
| Update | Update given database table |

**Log statements for capturing sql statement:**

logging.level.org.hibernate.SQL=debug //log SQL statements  
logging.level.org.hibernate.orm.jdbc.bind=trace //log values for SQL statements

**REST CRUD APIs**

|  |  |
| --- | --- |
| **HTTP METHOD** | **Desc** |
| POST | Create new entity |
| GET | Retrieve new entity |
| PUT | Update existing entity |
| DELETE | Delete an existing entity |
| PATCH | Update partial entity. |

**@PostContruct** Method called only once when the given bean constructed**. (How will it behave on prototype scope?)**

**JAVA JSON Mapping:**

Data binding is the process of converting JSON data to Java POJO. (Replacement of DTO)

Spring uses Jackson for Data binding. (Included by default)

Even without providing annotation, Jackson takes care of converting data to and fro.

**Path variables:**

@PathVariable.

Note: PathVariable and method parameters should be same.

*Handling Exception:*

We are returning response-entity with is equivalent to HTTP response. If I want to throw proper exception for exception scenario, I need to handle it.

**Steps:**

1. **Array Index out of Bound Exception**
2. Create custom error response POJO class. (Jackson will convert POJO object to Json)
3. Create custom Exception to handle it.
4. Throw the custom exception.
5. Catch exception with Annotation using @ExceptionHandler.
6. **MethodArgumentTypeMismatchException or Bad Request: (Catch all)**
7. Create custom error response POJO class. (Jackson will convert POJO object to Json)
8. Throw Exception
9. Catch Exception using @ExceptionController

So far, we catch exception on each controller.

**How to Catch Exception Globally?**

Using **@ControllerAdvice.** (we have something called restcontroller advice too. Purpose?)

* Controller Advice is like Interceptor or Filter.
* We can use it as pre-process requests to controller
* We can use it as post-process responses to controller for handling exceptions
* Perfect for making use global controllers.
* Example of Realtime use of AOP

**Spring Rest API Design**

Questions:

1. Who will use your API?
2. How will they use your API?
3. Design the API based on our requirements.

API design Process:

1. Review your API requirements.
2. Identify main resource/entity
3. Use HTTP methods to assign action on resource

@Repository: define bean as repository. Same as @component. It is for clarity. We will be having entity manager.

@Service: Another annotation: (mostly for delegating stuff)

1. Service Façade Design Pattern
2. Intermediate Layer for Custom Business Logic
3. Integrate from Multiple Datasources(DAO/repositories)

Ex: Employee service will speak with EmployeeDao, SkillsDao, PayrollDao etc

==========================================

**Spring Boot Data JPA:**

Our DAO code itself boilerplate. We can fix it by using JpaRepository provided by Spring. This will reduce 70% of code based on use case.

All we must do is extend JPARepository Interface. ***We don’t need implementation class itself.***

**Advanced Features on Data JPA:**

1. *Extending and adding additional queries with JPQL*
2. *Query Domain Specific Language.*
3. *Defining custom methods (low-level coding)*

**Spring Data Rest: to avoid boilerplate code for CRUD operation.**

Leverage your existing JpaRepository.

Spring data rest will scan for JPARepository and create rest endpoints based on Pojo and primary key mentioned on JPA repository.

Spring Data Rest Endpoints are **HATEOAS** complaints.

HATEOAS – Hypermedia AS The Engine Of Application State.

**Hypermedia** driven site provides access to REST endpoints along with pagination data for Array of Elements.

No controller/no service required.

**Plural form:**

I can configure base path and provide plural form for certain entities (where appending ‘s’ won’t do good).

**@RepositoryRestResource(path=”members”) members is alternative path.**

**Pagination:**

By default, it supports 20 per page.

<http://localhost:8080/employees?page=0/1/2> etc

**properties:**

spring.data.rest.base-path

spring.data.rest.default-page-size

spring.data.rest.max-page-size

**Sorting:**

**Sorting can be done using queryparam on entity parameters.**

<http://localhost:8080/employees?sort=lastName>

<http://localhost:8080/employees?sort=lastName,desc>

<http://localhost:8080/employees?sort=lastName,firstName,asc>

**Spring Security:**

Implemented using Servlet filters in the background. We are having two methods of securing applications:

1. Declarative (application security constraints on configuration (@Configuration))
2. Programmatic (framework provides extensibility(via API) to customize security based on our need)

**Spring Security with Servlet Filters:**

* Servlet Filters are used to pre-process and post-process web requests.
* Servlet Filters can route web requests based on security logic.
* lot of functionalities provided by spring security framework using servlet filters.

**In Spring security, passwords are stored using specific format. {id}encodedpassword**

|  |  |
| --- | --- |
| **ID** | **Description** |
| Noop | Plain text |
| Bcrypt | Bcrypt password hashing |

**Example:**

**{noop}test123 | {bcrypt}assdfgee342(length: 8+60)**

**Spring Security works with Servlet filters. As you know, any request will pass through list of filters before it reaches servlet (in our case, it is dispatcherServlet).**

**Filter modify the request and response based on filters. Filter will affect the downstream filter access or servlet access. So, order of execution is very important.**

**Spring inserts itself as Filter via Delegate Proxy.**

All the work will be happen with **SecurityFilterChain**.

[SecurityFilterChain](https://docs.spring.io/spring-security/site/docs/6.1.3/api/org/springframework/security/web/SecurityFilterChain.html) is used by [FilterChainProxy](https://docs.spring.io/spring-security/reference/servlet/architecture.html" \l "servlet-filterchainproxy) to determine which Spring Security Filter instances should be invoked for the current request.

[Architecture :: Spring Security](https://docs.spring.io/spring-security/reference/servlet/architecture.html)

Note: Spring Security filters all of them are beans. It is all registered with FilterChainProxy. Spring allow us to have more than one SecurityFilterChain. When we are having multiple Security Filter chain, Filter Chain proxy determines which one to invoke using Request Matcher interfaces. (on J2EE, filters will be invoked based on URL)

SecurityFilterChain takes care of authentication, authorization, exploiting protection and more.

In SecurityFilterChain, order of filter execution is important. For knowing the order of filter execution, verify FilterOrderRegistration code.

For enabling logging on spring security logging.level.org.springframework.security=TRACE

Note: While adding custom filter to SecurityfilterChain, make sure that you are not declaring it as Spring bean. Because it will be executed by Spring bean as well as Spring security framework in different order.

Example:

import java.io.IOException;

import jakarta.servlet.Filter;

import jakarta.servlet.FilterChain;

import jakarta.servlet.ServletException;

import jakarta.servlet.ServletRequest;

import jakarta.servlet.ServletResponse;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import org.springframework.security.access.AccessDeniedException;

public class TenantFilter implements Filter {

@Override

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) servletRequest;

HttpServletResponse response = (HttpServletResponse) servletResponse;

String tenantId = request.getHeader("X-Tenant-Id"); (1)

boolean hasAccess = isUserAllowed(tenantId); (2)

if (hasAccess) {

filterChain.doFilter(request, response); (3)

return;

}

throw new AccessDeniedException("Access denied"); (4)

}

}

Followed by,

@Bean

SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

// ...

.addFilterBefore(new TenantFilter(), AuthorizationFilter.class);

return http.build();

}

In case, If I want to declare it as bean to leverage dependency injection, use FilterRegistrationBean

@Bean

public FilterRegistrationBean<TenantFilter> tenantFilterRegistration(TenantFilter filter) {

FilterRegistrationBean<TenantFilter> registration = new FilterRegistrationBean<>(filter);

registration.setEnabled(false);

return registration;

} – It won’t register with Spring container.

As part of UserName and Password Authentication, we need to get UserDetails which will be given by UserDetails service which act as input to DAOAuthenticationEngine.

**Spring security by default support defined schema for Authentication.**

For predefined table,

Spring security provided JDBC code too.

JdbcUserDetailsManager( with dataSource). Datasource will be created by default on Spring boot based on config

Table: members, authorities.

For custom table,

We need to return object of JdbcUserDetailsManager with cutom query for users and authorities as shown below:

userDetailsManager.setUsersByUsernameQuery(

"select user\_id, pw, active from members where user\_id=?"

);

//defined query to determine authorities/roles by username

userDetailsManager.setAuthoritiesByUsernameQuery(

"select user\_id, role from roles where user\_id=?"

);

If we need to define our JPA, service, repository way,

@Bean

public BCryptPasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

//authenticationProvider bean definition

@Bean

public DaoAuthenticationProvider authenticationProvider(UserService userService) {

DaoAuthenticationProvider auth = new DaoAuthenticationProvider();

auth.setUserDetailsService(userService); //set the custom user details service

auth.setPasswordEncoder(passwordEncoder()); //set the password encoder - bcrypt

return auth;

}

**Spring MVC:**

Spring MVC Validation uses Java Bean Validation API (provided by Java). It defines a metadata model and API for entity validation.

Spring boot and Thymeleaf also support bean validation APIs.

<http://www.beanvalidation.org>

**Bean Validation features: (spring boot start validator)**

* required
* validate length.
* Validate numbers
* Validate with regular expressions
* Custom validations

Ex:

@NotNull, @Min, @Max, @Size, @Pattern, @Future/@Past and so on.

**@InitBinder:**

@NotNull, @Size, @Min will consider space as valid values. For example, I am saying that LastName should not be empty, it will allow to have space. But it doesn’t make sense.

* This will work as pre-processor.
* It will process each web request to our controller.
* Method annotated with @InitBinder will be executed.

Ex: for lastName, I need to remove leading and trailing whitespaces, and it has only spaces, then treat it as null.

**Annotation and Usage:**

* @NotNull(make it required). ***(Note: It will work on Objects. Not on primitive types. Work for Integer too)***
* @Size for String.
* @InitBinder for handling empty space issues. We can have our custom implementation too.
* @Min, @Max for Numbers
* @Pattern for regular expressions

**Handling String input to Integer fields – Need to throw exceptions:**

1. **Custom error message from property file. (messages.properties)**

**Ex:** typeMismatch.<objectname>.<Field/variableName>=<custom error code>

**How do we come to this format? Let’s deep dive?**

*BindingResult has so much information related to validation process.*

*If you print binding result, it will say how the property picked up. (It is somewhat same as selecting template for API on Sterling)*

**Custom Validation:**

we need to create custom java annotation with Jakarta provided annotation @Constraint.

Ex:

@Constraint(validatedBy = CourseCodeConstraintValidator.class)

**Spring MVC – CRUD:**

Fall back page can be defined on Spring boot. If I declare index.html with in **static folder and there is no url match happens, it will display index.html file from static folder.**

We will be having form for providing input and we need to send model attributes. Thymeleaf takes care of doing getters(**load)** and setters(**submit**) on model object.

Th:action, th:object, th:fields and so on for building thymeleaf template.

Note: If I want to display list sort by specific column we don’t have to create new implementations.

findAllBy{fieldName}{sortType}(); - springboot parse method name and create appropriate Query.

Ex: findAllByOrderByLastNameAsc();

**Spring Security MVC**

Spring security uses web browser session i.e only on browser close, session will end. Until then Session will be active.

On spring security, we need to define which URLs are should be authenticated with roles. We need to configure securityFilterChain as shown below.

Public SecurityFilterChain filterChain(HttpSecurity httpSec) throws Exception{

httpSec.authorizeHttpRequests( reqMatcher ->

reqMatcher.anyrequest().authenticated()

)

.formLogin(

Form ->

Form

. loginpage(“/showLoginForm”)) ---- link to controller custom page

.loginProcessingUrl(“authenticateTheUser”) – given by spring security. **(url can be anything)**

.permitAll()); – allowed for everyone

Return httpSec.build();

Since authentication handled by Spring security, we need to use default field names. (username, password)

**Logout functionality:**

We need to use spring security logout operation.

On spring security filter chain, just add

.logout(formLogout -> formLogout.permitAll()

Default login – **/logout**

Spring security takes care invalidate session and redirect to login page with **logout param on url**.

**Display User ID and Roles:**

Spring security provides way to access user Id and roles.

We need to use thymeleaf/extras/spring-security.

xmlns:sec="http://www.thymeleaf.org/extras/spring-security" >

User: <span sec:authentication="principal.username" />

Role(s): <span sec:authentication="principal.authorities" />

**Custom access denied page:**

**On security filter chain:**

.exceptionHandling(customzier -> customzier.accessDeniedPage("/access-denied"))

Need to define controller and view page for URI path

**Spring and Hibernate**

**Entity Lifecycle:**

|  |  |
| --- | --- |
| Operations | Description |
| Detach | If entity is detached, it is not associated with hibernate session |
| Merge | If instance is detached from session, then merge will reattach the session |
| Persist | Transitions new instance to managed state. Next flush/commit will save in DB |
| Remove | Transitions managed entity to be removed. Next flush/commit will delete from DB |
| Refresh | Reload/Sync object with data from db. Prevents stable data |

A black background with white rectangles

Description automatically generated

Cascade: Apply same operations on related entities

|  |  |
| --- | --- |
| Cascade Type | Description |
| Persist | If entity is persisted or saved, related entity also be persisted |
| Remove | If entity is removed/deleted, related entity also be removed |
| Refresh | If entity is refreshed, related entity also be refreshed |
| Detach | If entity is detached, related entity also be detached |
| Merge | If Entity is merged, then related entity will also be merged |
| ALL | All of Above Cascade types |

Where we are having foreign key,

@OneToOne(cascade = cascadeType.ALL)

@JoinColumn

For Bi-directional,

@OneToOne(**mappedBy=”instructiorDetail”, cascade = cascadeType.ALL**) – object name on another class

**Dive Deep:**

Mappedby tells to Hibernate that

* Look at the Instructordetail property on instructor class.
* Use information from Instructor class (join column)
* To help find an associated Instructor.
* **Note: I can have different cascade type on bidirectional.**

**IF I need to break bi-directional link upon deleting, I need to exclude cascade type on appropriate class and also I need to set null on referential entity.**

**Ex:**

@Override  
@Transactional  
public void deleteInstructorDetailById(int id) {  
 InstructorDetail detail = findInstructorDetailById(id);  
 //breaking bi-directional link  
 detail.getInstructor().setInstructorDetail(null);  
 entityManager.remove(detail);  
}

**If we did not set null value, nothing will happen.**

**@OneToMany, @ManyToOne.**

Note: Join column will be there with ManyToOne since that will have foreign key. (we can have with ***one to many*** also where we want to maintain unidirectional.)

Important: Usually, when we are deleting references, we won’t delete all records.

Ex: one instructor mapped to multiple courses. Deleting instructor should not delete courses.

When we are having bi-directional links, we need to write extra code to honor/maintain link. (same as one to one bidirectional)

**Lazy Loading:**

Whenever we are loading list of entities (one to many, many to many), default fetching types is Lazy Loading.

**More About Lazy Loading:**

* We need to have an open hibernate session (database connection) for lazy loading whenever required.
* If hibernate session is closed and we try to retrieve data, hibernate will throw exception.

While executing below statements, we will get exceptions since Hibernate Session is already closed

private void findInstructorWithCourses(AppDao appDao) {  
 Instructor tmp = appDao.findInstructorById(1);  
 System.*out*.println(tmp);  
 System.*out*.println("Association");  
 System.*out*.println(tmp.getCourses()); //This one Is failed  
}

Two Solutions for exceptions:

1. Set fetch type as EAGER.
2. Find course lazily wherever required and set. (handle it in DAO, DTO) – Cons: multiple query
3. Join Fetch – there will be one query and fetch type Lazy still intact. (Eager loading – merely overriding options)

**@OneToMany – Uni-Directional**

Example req: Course will have multiple reviews. If Courses are deleted, all reviews should be deleted.

**@ManyToMany**

**Like @JoinColumn on oneToMany/ManyToOne, we are having @JoinTable on ManyToMany.**

**@JoinTable tells hibernate,**

**(joinColumns)Look at id of own entity primary key and for other(inverseJoinColumn) side, we will have primary key of other table.**

**Aspect Oriented Programming**

In Our APP, we need to add logging and security. But, we need to avoid code tangling (business problems should be isolated and code should be scatter).

AOP takes care of Two main problem:

1. Code tangling
2. Code Scattering

Possible solutions:

* Inheritance
* Code delegation

Good Option is Aspect Oriented Programming. **It is a programming technique based on aspects.** Aspect encapsulated cross-cutting logic.

**Aspects:**

It can be maintained as a layer without affecting business logic.

It can be reused at multiple locations.

Some aspect/class applied based on configuration.

**Aspect Concepts and Terminology:**

We will be using Proxy pattern beneath the code. Our app or business logic doesn’t know about AOP. Only configurations.

**Benefits:**

* Code for aspect designed in a single class.
  + Much better than being scattered everywhere.
  + Promotes code reuse and easier to change.
* Business Code in your application is cleaner
  + Only applies to business functionality.
  + Reduces code complexity
* Configurable
  + Based on configuration, apply aspects selectively to different parts of app
  + No need to make changes on main application code.

**Additional Use cases**

* **Most Common**
  + Logging, security, transactions
* **Audit Logging**
  + Who, what, when, where
* **Exception handling**
  + Log exception and notify Devops team via SMS/Email
* **API management**
  + How many times has a method been called user
  + Analytics – what are peak times, what is average load etc.(sterling statistics)

**AOP Terminology:**

**Aspect:** Module of Code for 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 method (like try catch finally)
* **After returning Advice:** run after method (success execution)
* **After throwing advice:** run after throwing exception. (if exception thrown)
* **Around Advice:** run before and after method.

**Weaving: (connecting two yarns to make clothes/**[நெசவு](https://www.bing.com/ck/a?!&&p=fe62592b26acee02JmltdHM9MTY5NjIwNDgwMCZpZ3VpZD0yNDYyZjliYi1lODRkLTZkOGMtMWIyNS1lYmU4ZTllMDZjOGUmaW5zaWQ9NTIxMw&ptn=3&hsh=3&fclid=2462f9bb-e84d-6d8c-1b25-ebe8e9e06c8e&psq=nesavu+tamil&u=a1aHR0cHM6Ly90YS53aWtpcGVkaWEub3JnL3dpa2kvJUUwJUFFJUE4JUUwJUFGJTg2JUUwJUFFJTlBJUUwJUFFJUI1JUUwJUFGJTgxJUUwJUFFJUE0JUUwJUFGJThEXyVFMCVBRSVBNCVFMCVBRiU4QSVFMCVBRSVCNCVFMCVBRSVCRiVFMCVBRSVCMiVFMCVBRiU4RCVFMCVBRSVBOCVFMCVBRiU4MSVFMCVBRSU5RiVFMCVBRiU4RCVFMCVBRSVBQSVFMCVBRSVBRSVFMCVBRiU4RA&ntb=1))

* Connecting aspects to target objects to create advised objects.

**Types:**

1. Compile time weaving
2. Runtime weaving
3. Load-time weaving

***Note: Runtime weaving performance is bad.***

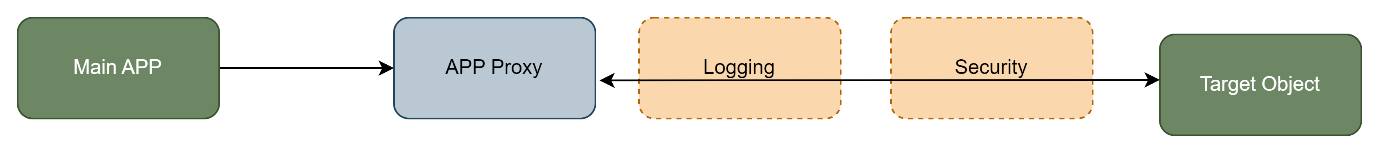
**AOP Frameworks:**

1. **Spring AOP**
   1. **Spring provides AOP support OOB.**
   2. **Key component of Spring – security, transactions, caching etc**
   3. **Used Runtime weaving concepts.**
2. **AspectJ**
   1. **Released originally in 2001. (**[**www.eclipse.org/aspectJ**](http://www.eclipse.org/aspectJ)**)**
   2. **Support full AOP specifications or full AOP stack.**
   3. **Rich Support for join points(method, constructor, field) and code weaving(load-time, compile-time and run-time)**

|  |  |  |
| --- | --- | --- |
| **Framework** | **Advantages** | **Disadvantages** |
| **Spring AOP** | 1. Simpler to use than AspectJ. 2. Uses Proxy Pattern 3. Can migrate to AspectJ when using @Aspect annotation | 1. Only supports method-level join points 2. Can only apply aspects to beans created by Spring app context 3. Minor performance cost for aspect execution |
| **AspectJ (Aspect for Java)** | 1. Support all join points. 2. Works with any POJO not just beans created from app context. 3. Faster performance compared to spring AOP. 4. Complete AOP support | 1. Compile time weaving requires extra compilation step. 2. AspectJ pointcut syntax can become complex |

**Spring AOP:**

**@Before advice:**

****

**Whenever, Main app want to execute methods from target object, Logging and security will execute before target method execution.**

@Aspect  
@Component  
public class MyDemoLoggingAspect {  
  
 //This is where we are all add our advises.  
  
 //we are mentioning pointcut expression within quotes  
 @Before("execution(public void addAccount())")  
 public void beforeAddAccountAdvice(){  
 System.*out*.println("\n=======>> Executing @Before Advice on adding Account");  
 }  
}

This method will be executed whenever we are calling this method.

**AOP – Pointcut Expressions:**

Point-cut Expressions: predicate expressions where advice should be applied.

Spring AOP uses AspectJ’s pointcut expression language. We have multiple ways:

1. **Execution Pointcuts**

Syntax: ***execution(modifiers-pattern? Return-type-pattern declaring-type-pattern? Method-name-pattern(param-pattern throws-pattern?)***

**Execution** – denotes execution pointcut.

**Modifiers-pattern** – public,private etc. (Spring AOP supports only public)

**Return-type-pattern** – void, Boolean, string, List<Customer> etc

**Declaring-type-patteren** – classname of the type that you’re gonna use for this given method

**Method-name-pattern** – method name or wildcards

**Param-name** – match on method for a given parameter-type list

**Throws pattern** – match on method based on throwing exception.

Pattern is optional if it has **?**

**Example:**

**@Before(“execution(public void com.praga.dao.AccountDao.addAccount())”)**

**Explanation:**

**Public** – access modifier

**Void** – return type

**Com.praga.dao.AccountDao** – declare type

**addAccount** – method name

**@Before(“execution(public void addAccount())”)** = match addAccount method in any class.

**@Before(“execution(public void add**\*())”) = match any method starts with add in any class.

**@Before(“execution(public VerificationResult processCreditCard\*()** = match method starting with processCreditCard in any class

**For param,**

() – no args

(\*) – one param with any type

(..) – any number of param with any type

(Type) – one param with given type(fully qualified or with wildcards to match)

(type,..) - one param with given type, any(0+) number of param with any type

**Point-cut declarations:**

@Aspect  
@Component  
public class MyDemoLoggingAspect {  
  
 //This is where we are all add our advises.  
 **@Pointcut("execution(\* com.praga.aopdemo.test..\*.\*(..))")** private void forDaoPackage(){}  
 //we are mentioning pointcut expression within quotes  
  
 @Before("forDaoPackage()")  
 public void beforeAddAccountAdvice(){  
 System.*out*.println("\n=======>> Executing @Before Advice on adding Account");  
 }  
  
}

Point-cut declarations are useful wherever it required. Basically reusing.

**Combining Point-cut declarations:**

**Problem:**

* How to apply multiple point cut declarations on single advice.
* Execute an advice only if certain conditions are met.

We will be using logical operators to combine point cut operations.

&&, ||, !

**Ordering Aspects:**

Let’s say I have advice which is matching on specific method. Execution of advice is not ordered (unpredictable). (Note: All advices are on same aspect)

**Solution:**

Refactor: place advices on separate Aspects

Order: apply @Order annotation. This guarantees when aspects are applied.