Maven means accumulator of knowledge. This process was introduced to build the application

1. Making the build process easy
2. Providing a uniform build process
3. Providing quality project information
4. Encouraging better development practice
5. Maven is reusable ANT scriptlet.

Maven CLI is available

* To create project
* To Clean
* To compile
* To package
* To do Unite Test
* To run application

Spring Tool Suites IDE or Plugged-in be used for eclipse to bootstrap the process of project creation with pre-defined dependencies

OR Click here to create a project <https://start.spring.io>

Graphical user interface, text, application, email

Description automatically generated

A zip file with the name Artifact Name will be downloaded.

Unzip it

and place it to appropriate location and open this project from eclipse

in Eclipse – File - > Open Project from File System

create a Employee class and under com.example.model package

annotate it @Component

Go to main class

ApplicationContext ctx=SpringApplication.*run*(Jor2HelloWordApplication.**class**, args);

Employee e=(Employee) ctx.getBean(Employee.**class**);

e.setName("Nitin");

System.***out***.println(e.getName());

And run the java application

**Spring Boot Web**

* It has an embedded web server. It supports the following servers:

1. Tomcat [ default ]
2. Jetty
3. Undertow

You no longer need any server to be installed and therefore no need to “run on server” means deployment.

You can simply run the application using **run as Java application** from eclipse. Spring Boot Web Application has main method that you need to run it using the java command of your JRE

java -jar EmonicsHrm.jar

* Java runtime will start the embedded tomcat server at port 8080 [ default ] and the application’s jar will be deployed automatically.

**Spring Rest**

* De-facto standard for building web services on the web because it is easy to build and consume
* REST embraces the precepts of the web including architecture, benefits and everything else.
* REST supports interoperable communication between client and server over the HTTP/s protocol.
* The web & its core protocol HTTP.
  + GET ------- Read
  + POST------- Create
  + PUT -------- Update
  + DELETE ------- Delete
  + OPTION
  + HEAD
  + TRACE
* Caching
* Redirection & Forwarding
* Security [ Encryption & Authentication ]
* Backward compatibility
* Evolving APIs
* Scalable Services
* Stateless Service

**Spring Framework + Tomcat/Jetty/Undertow – XML Configuration = Spring Boot.**

* java-based framework used to create microservices. Microservices is an architecture that allows the developers to develop n deploy services independently.
* Not only microservices but various Spring module-based applications can be developed.
  + Spring JPA
  + Spring JDBC
  + Spring AOP
  + Spring Web
  + Spring REST
  + Spring Batch
* It supports Java, Kotlin, and Groovy.
* You can get started with minimum configuration without the need for an entire Spring Configuration setup.
* There is no need to go with XML configurations. It uses a Java-Based Configuration.
* Spring Boot Web will come with an embedded web server. Java runtime [ java ] will be used to run the program
* Easy to understand and develop Spring application
* Increase productivity
* Reduces the development time.

**How does it work?**

* It automatically configures your application based on dependencies.
* **@EnableAutoConfiguration** annotation is used for this purpose.
* **@ComponentScan** annotation is used to scan all the beans in the root package and its sub-packages.
* **@SpringBootApplication** = **@EnableAutoConfiguration + @ComponentScan**

**java HelloWorld**

@SpringBootApplication

class HelloWorld{

p s v m(String[[ args){

}

}

Difference between Controller & RestController

@Controller + @ResponseBody = @RestController

@RestController @RequestMapping("/rest")

**public** **class** GreetingRS {

//@RequestMapping(path="/greeting", method=RequestMethod.GET)

@GetMapping("/greeting")

**public** Employee greet(@RequestParam("name") String name) {

//create utility class to return appropriate greet message depending on time

//dao...service.....

Employee e=**new** Employee(); e.setName(name);

**return** e;

}

}

Graphical user interface, text, application

Description automatically generated

**Alternatively , dependencies can be added in pom.xml under <dependencies > </dependencies>**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<http://................................../hr-rest?id=1>

Query String ----------------------------------------------- @RequestParam(“id”)

<http://................................../hr-rest/1>

@GetMapping(“/{id}”)

PathParam @PathVariable

Interface CrudRespository<T,ID> extends Repository<T,ID>{

long count();

Optional<T> findById(ID id); //find

<S extends T> save(S entity) ; //insert and update

void deleteById(ID id);

Boolean existById(ID id);

// many more – refer api documentation

}

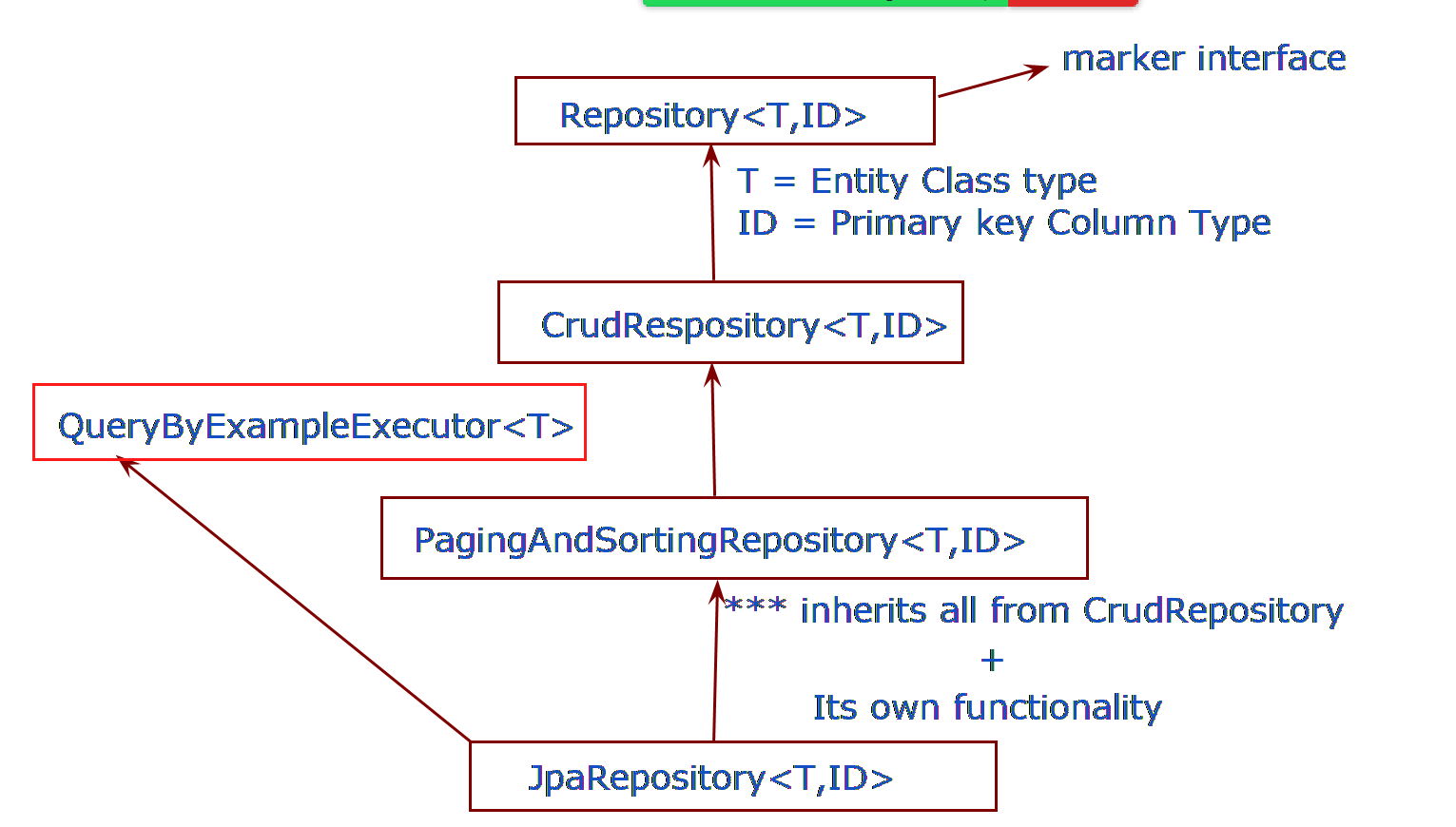
Interface PagingAndSortingRepository<T,ID> extends CrudRespository<T,ID>

{

Page<T> findAll(Pageable pageable)

Iterable<T> findAll(Sort sort)

}



public interface **JpaRepository<T,ID>**

extends [PagingAndSortingRepository](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html?is-external=true)<T,ID>, [QueryByExampleExecutor](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/query/QueryByExampleExecutor.html?is-external=true)<T>{

}

@Required

@Autowired

@Configuration

@ComponentScan

@Bean at method level

Spring Boot Stereotype annotations

1. @Component : it is used to mark a class as a bean. This class will be used in the application context as Spring Bean.
2. @Controller / @RestController: It is a specialization of @Component and the logic in this class for managing web requests n responses. @RequestMapping annotation will be used by the controller.
3. @Repository: It is a specialization of @Component and the logic in this class for DAO.
4. @Service: It is a specialization of @Component and the logic in this class for managing the business operation.

**Spring AOP**

Aspect Oriented Programming (AOP) uses aspects known as concerned. It is used to increase modularity by cross-cutting concerns.

A cross-cutting concern is a concern that can affect the whole application and should be centralize at one location.

This concern can be for transaction management, security and logging etc.

**Methods name**

M1()

M2()

M3()

M4()

M5()

C1()

C2()

C3()

C4()

Use AOP to address concerns by writing code in one place and applying it to various operations.

Diagram

Description automatically generated

**AOP Terminology**

1. JoinPoint : It is point in your program [ e.g method execution, exception handling ]
2. Advice : Action taken by aspect at particular point.
   1. Before Advice
   2. After Returning Advice
   3. After Throwing Advice
   4. After [ finally ] Advice
   5. Around Advice: it executes before and after a join point.
3. Pointcut
   1. An expression language of AOP to match join point.
4. Target Object
   1. An object being advised by one or more aspects.
5. Aspect
   1. A class that contains advice, joinpoint etc.
6. Intercept
   1. An aspect that contains only one advice.
7. AOP Proxy
   1. Use by AOP framework. JDK based or CGLIB proxy in spring framework
8. Weaving
   1. Linking aspect with other application types or objects to create advised objects.
9. Add dependencies in the pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

1. Create an Aspect [ Create A class ]

@Aspect

@Component

**public** **class** EmplpoyeeServiceAspect {

@Before(value="execution(\* jor.empapp.service.EmployeeServiceImpl.\*(..))")

**public** **void** beforeAdvice(JoinPoint jp) {

System.***out***.println(jp.getSignature());

}

@After(value="execution(\* jor.empapp.service.EmployeeServiceImpl.\*(..))")

**public** **void** afterAdvice(JoinPoint jp) {

System.***out***.println(jp.getSignature());

}

@Around(value = "execution(\* jor.empapp.service.EmployeeServiceImpl.addEmployee(..))")

**public** **void** around(ProceedingJoinPoint jp) {

**long** intime=System.*currentTimeMillis*();

Object[] obj=jp.getArgs();

Employee e1=(Employee) obj[0];

e1.setName(e1.getName().toUpperCase());

System.***out***.println("Arg:"+obj[0]);

**try** {

jp.proceed();

} **catch** (Throwable e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**long** outtime=System.*currentTimeMillis*();

**long** duration=outtime-intime;

System.***out***.println(duration);

}

}

Pointcut Designators

A pointcut expression starts with a **pointcut designator (PCD)**, which is a keyword telling Spring AOP what to match. There are several pointcut designators, such as the execution of a method, type, method arguments, or annotations.

1. **execution**

@Pointcut("execution(public String com.example.pointcutadvice.dao.FooDao.findById(Long))")

This example pointcut will exactly match the execution of the *findById* method of the *FooDao* class. This works, but it's not very flexible. Suppose we'd like to match all the methods of the *FooDao* class, which may have different signatures, return types, and arguments. To achieve this, we can use wildcards:

@Pointcut("execution(\* com.example.pointcutadvice.dao.FooDao.\*(..))")

Here, the first wildcard matches any return value, the second matches any method name, and the *(..)* pattern matches any number of parameters (zero or more).

**2. *within***

Another way to achieve the same result as the previous section is by using the *within* PCD, which limits matching to join points of certain types:

@Pointcut("within(com.example.pointcutadvice.dao.FooDao)")

We can also match any type within the *com.example* package or a sub-package:

@Pointcut("within(com.example..\*)")

**3. *this* and *target***

*this* limits matching to join points where the bean reference is an instance of the given type, while *target*limits matching to join points where the target object is an instance of the given type. The former works when Spring AOP creates a CGLIB-based proxy, and the latter is used when a JDK-based proxy is created. Suppose that the target class implements an interface:

public class FooDao implements BarDao {

...

}

In this case, Spring AOP will use the JDK-based proxy, and we should use the *target* PCD because the proxied object will be an instance of the *Proxy* class and implement the *BarDao* interface:

@Pointcut("target(com.example.pointcutadvice.dao.BarDao)")

On the other hand, if *FooDao* doesn't implement any interface, or the *proxyTargetClass* property is set to true, then the proxied object will be a subclass of *FooDao*and we can use the *this* PCD:

@Pointcut("this(com.example.pointcutadvice.dao.FooDao)")

**4. *args***

We can use this PCD for matching particular method arguments:

@Pointcut("execution(\* \*..find\*(Long))")

This pointcut matches any method that **starts with find** and **has only one parameter of type *Long***. If we want to match a method with any number of parameters, but still having the fist parameter of type *Long*, we can use the following expression:

@Pointcut("execution(\* \*..find\*(Long,..))")

**5. *@target***

The *@target* PCD (not to be confused with the *target* PCD described above) limits matching to join points where the class of the executing object has an annotation of the given type:

@Pointcut("@target(org.springframework.stereotype.Repository)")Copy

**6. *@args***

This PCD limits matching to join points where the runtime type of the actual arguments passed have annotations of the given type(s). Suppose that we want to trace all the methods accepting beans annotated with the *@Entity* annotation:

@Pointcut("@args(com.example.pointcutadvice.annotations.Entity)")

public void methodsAcceptingEntities() {}Copy

To access the argument, we should provide a *JoinPoint* argument to the advice:

@Before("methodsAcceptingEntities()")

public void logMethodAcceptionEntityAnnotatedBean(JoinPoint jp) {

logger.info("Accepting beans with @Entity annotation: " + jp.getArgs()[0]);

}Copy

**7. *@within***

This PCD limits matching to join points within types that have the given annotation:

@Pointcut("@within(org.springframework.stereotype.Repository)")

Which is equivalent to:

@Pointcut("within(@org.springframework.stereotype.Repository \*)")

**8. *@annotation***

This PCD limits matching to join points where the subject of the join point has the given annotation. For example, we can create a *@Loggable* annotation:

@Pointcut("@annotation(com.example.pointcutadvice.annotations.Loggable)")

public void loggableMethods() {}

Then we can log the execution of the methods marked by that annotation:

@Before("loggableMethods()")

public void logMethod(JoinPoint jp) {

String methodName = jp.getSignature().getName();

logger.info("Executing method: " + methodName);

}

4. **Combining Pointcut Expressions**

Pointcut expressions can be combined using **&&**, **||**, and **!** operators:

@Pointcut("@target(org.springframework.stereotype.Repository)")

public void repositoryMethods() {}

@Pointcut("execution(\* \*..create\*(Long,..))")

public void firstLongParamMethods() {}

@Pointcut("repositoryMethods() && firstLongParamMethods()")

public void entityCreationMethods() {}

1 seconds = 1000 miliseconds

1 micro-second = 1/6 seconds

1 nano-second = 1/9 seconds

1 pico-seconds = 1/12 seconds

***Spring Transaction***

Data should remain integrated when any changes are done. If the data integrity is compromised, the whole data will be get disturbed and corrupted.

There are 4 properties of Transactions known **as ACID properties**.

A = Atomicity

C= Consistency

I = Isolation D = Durability

**@Transactional**

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html>

|  |
| --- |
| [**MANDATORY**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#MANDATORY)  Support a current transaction, throw an exception if none exists. |
| [**NESTED**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#NESTED)  Execute within a nested transaction if a current transaction exists, behave like REQUIRED otherwise. |
| [**NEVER**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#NEVER)  Execute non-transactionally, throw an exception if a transaction exists. |
| [**NOT\_SUPPORTED**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#NOT_SUPPORTED)  Execute non-transactionally, suspend the current transaction if one exists. |
| [**REQUIRED**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#REQUIRED)  Support a current transaction, create a new one if none exists. |
| [**REQUIRES\_NEW**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#REQUIRES_NEW)  Create a new transaction, and suspend the current transaction if one exists. |
| [**SUPPORTS**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html#SUPPORTS)  Support a current transaction, execute non-transactionally if none exists. |

Recording information about execution flow, success o failure of operation while your program is running is known is logging.

Recording information can be done on Files or Console. Web Application Server they maintain the log for the activities being perform on their services. The file that stores the information on the server is server.log.

1. Java.util.logging API
2. Log4J

* **Log4J is highly configurable with external configurations files.**
* **It has three main components.**
  + **Loggers : capturing log information**
  + **Appenders : publishing log information to destination**
  + **Layouts: formatting log information**

1. Add dependency to pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>

**private** **static** **final** Logger ***logger***=LogManager.*getLogger*(EmployeeServiceImpl.**class**);

**log4j2.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<Configuration status=*"DEBUG"*>

<Appenders>

<Console name=*"logConsole"* target=*"SYSTEM\_OUT"*>

<PatternLayout pattern=*"%d{HH:mm:ss:SSS}[%t] %-5level %logger{36} - %msg%n"*/>

</Console>

<File name=*"logFile"* fileName=*"logs/app.log"*>

<PatternLayout>

<Pattern>%d %p %c{1.} [%t] %m%n</Pattern>

</PatternLayout>

</File>

</Appenders>

<Loggers>

<Logger name=*"jor.empapp"* level=*"debug"* additivity=*"false"*>

<AppenderRef ref=*"logConsole"*/>

<AppenderRef ref=*"logFile"*/>

</Logger>

<Logger name=*"org.springframework.boot"* level=*"error"* additivity=*"false"*>

<AppenderRef ref=*"logConsole"*/>

</Logger>

<Root level=*"error"*>

<AppenderRef ref=*"logConsole"*/>

<AppenderRef ref=*"logFile"*/>

</Root>

</Loggers>

</Configuration>