Spring Boot
+
Introduction to
Spring Framework Annotations & Spring Data JPA

By

Pichet Limvajiranan

Spring Framework Annotations

- Basically, there are 6 types of annotation available in the whole spring framework.
  1. Spring Web Annotations
  2. Spring Core Annotations
  3. Spring Boot Annotations
  4. Spring Scheduling Annotations
  5. Spring Data Annotations
  6. Spring Bean Annotations

1) Spring Web Annotations

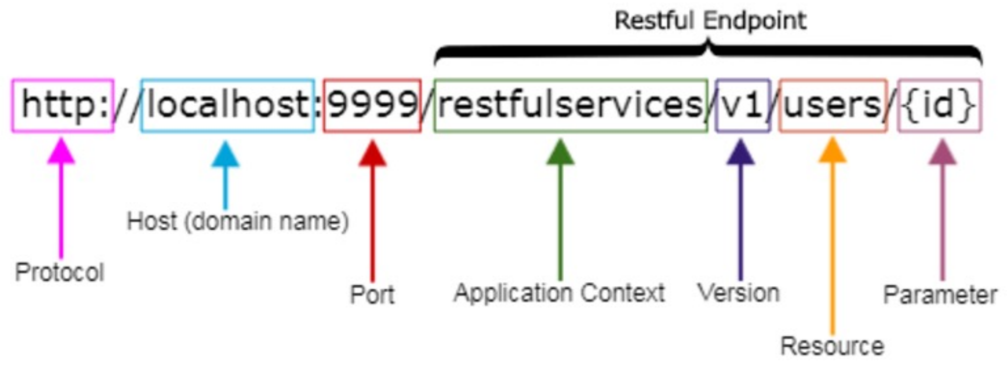
- Present in the org.springframework.web.bind.annotation
- Some of the annotations that are available in this category are:
  - **@RequestMapping**
  - **@RequestBody**
  - **@PathVariable**
  - @RequestParam
  - Response Handling Annotations
    - @ResponseBody @ExceptionHandler @ResponseStatus
  - @Controller
  - **@RestController**
  - @ModelAttribute
  - @CrossOrigin

Spring Web Annotation example

```
@RestController
@RequestMapping("/api/offices")
public class OfficeController {
    :
    :
    @GetMapping("/{officeCode}")
    public Office getOfficeById(@PathVariable String officeCode) {
        return service.getOffice(officeCode);
    }


    @PostMapping("")
    public Office addNewOffice(@RequestBody Office office) {
         return service.createNewOffice(office);
   }
```

REST API URI Naming Conventions and Best Practices



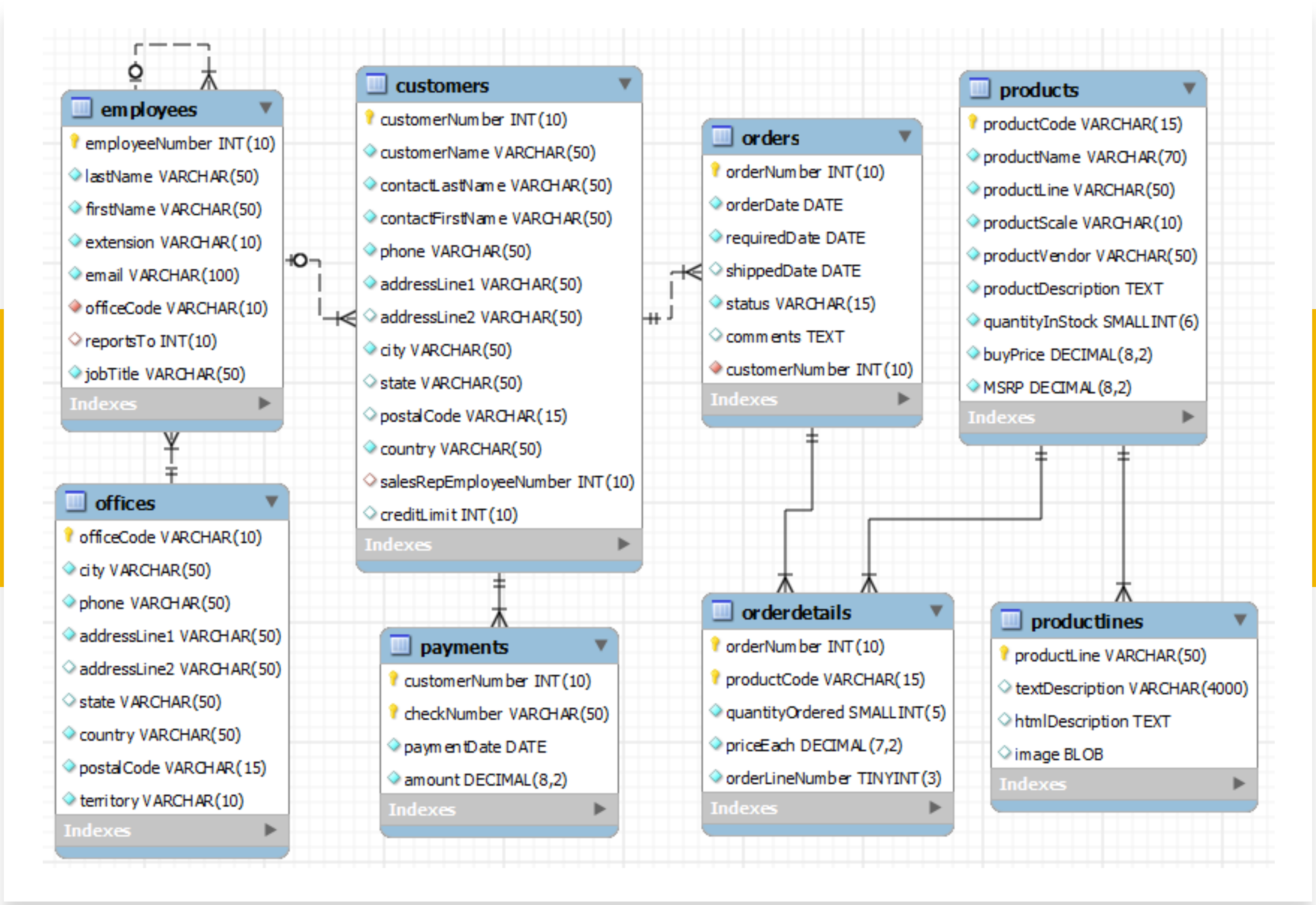
- Singleton and Collection Resources

```
/customers                  // is a collection resource
/customers/{id}             // is a singleton resource
```

- Sub-collection Resources

```
/customers/{id}/orders                  // is a sub-collection resource
```

- Best Practices
  - https://medium.com/@nadinCodeHat/rest-api-naming-conventions-and-best-practices-1c4e781eb6a5
  - https://restfulapi.net/resource-naming



Excercise

Design REST Resources representation for the Classicmodels

2) Spring Core annotations

- Spring Core annotations are present in the 2 packages
  - org.springframework.beans.factory.annotation
  - org.springframework.context.annotation
- We can divide them into two broad categories:
  - **DI-Related Annotations**
    - **@Autowired**   @Qualifier   @Primary
    - @Bean   @Lazy   @Required
    - @Value   @Scope
    - @Lookup, etc.
  - Context Configuration Annotations
    - @Profile   @Import
    - @ImportResource   @PropertySource, etc.

@Autowired

- We use @Autowired to mark the dependency that will be injected by the Spring container.
- Applied to the fields, **setter** methods, and **constructors**. It injects object dependency implicitly.

```java
public class OfficeService {
    @Autowired
    private OfficeRepository repository;
```

```java
public class OfficeService {
    private final OfficeRepository repository;
    @Autowired
    public OfficeService(OfficeRepository repository) {
        this.repository = repository;
    }
```

```java
public class OfficeService {
    private final OfficeRepository repository;
    @Autowired
    public void setRepository(OfficeRepository repository) {
        this.repository = repository;
    }
```

3) Spring Boot Annotations

- @SpringBootApplication Combination of three annotations
  - @EnableAutoConfiguration
  - @ComponentScan
  - @Configuration.

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

5) Spring Data Annotations

Spring Data provides an abstraction over data storage.

- Common Spring Data Annotations

  @Transactional

  @NoRepositoryBean

  @Param

  @Id

  @Transient

  @CreatedBy, @LastModifiedBy, @CreatedDate, @LastModifiedDate

- Spring Data JPA Annotations

  @Query

  @Procedure

  @Lock

  @Modifying

  @EnableJpaRepositories

- Spring Data Mongo Annotations

  @Document

  @Field

  @Query

  @EnableMongoRepositories

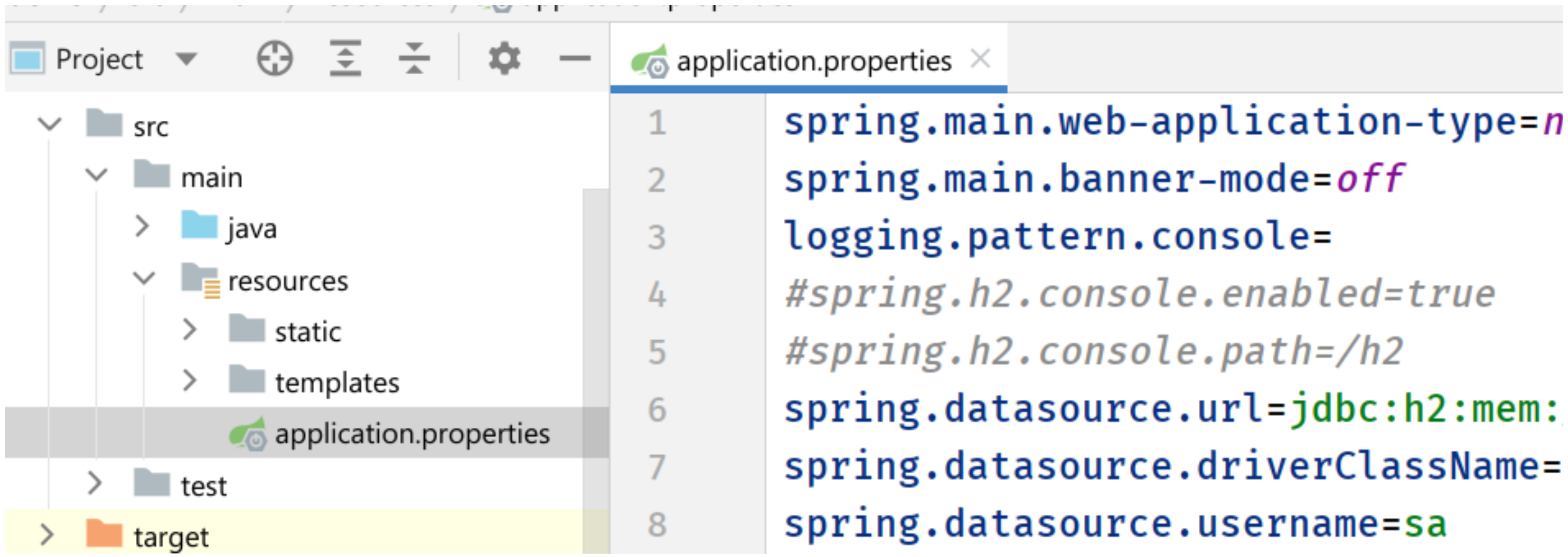
6) Spring Bean Annotations

- Some of the annotations that are available in this category are:
  - @ComponentScan
  - @Configuration
  - Stereotype Annotations
    - @Component
      - **@Service**
      - **@Repository**
      - **@Controller**

@Service: We specify a class with @Service to indicate that they're holding the business logic. Besides being used in the service layer, there isn't any other special use for this annotation. The utility classes can be marked as Service classes.
@Repository: We specify a class with @Repository to indicate that they're dealing with **CRUD operations**, usually, it's used with DAO (Data Access Object) or Repository implementations that deal with database tables.
@Controller: We specify a class with @Controller to indicate that they're front controllers and responsible to handle user requests and return the appropriate response. It is mostly used with REST Web Services.

Spring Boot Application Properties

- Spring Boot Framework comes with a built-in mechanism for application configuration using a file called application.properties.
- It is located inside the src/main/resources folder.
- The properties have default values.
- We can set a property(s) for the Spring Boot application.



```
spring.main.web-application-type=n
spring.main.banner-mode=off
logging.pattern.console=
#spring.h2.console.enabled=true
#spring.h2.console.path=/h2
spring.datasource.url=jdbc:h2:mem:
spring.datasource.driverClassName=
spring.datasource.username=sa
```

Spring Boot Application Properties - Examples

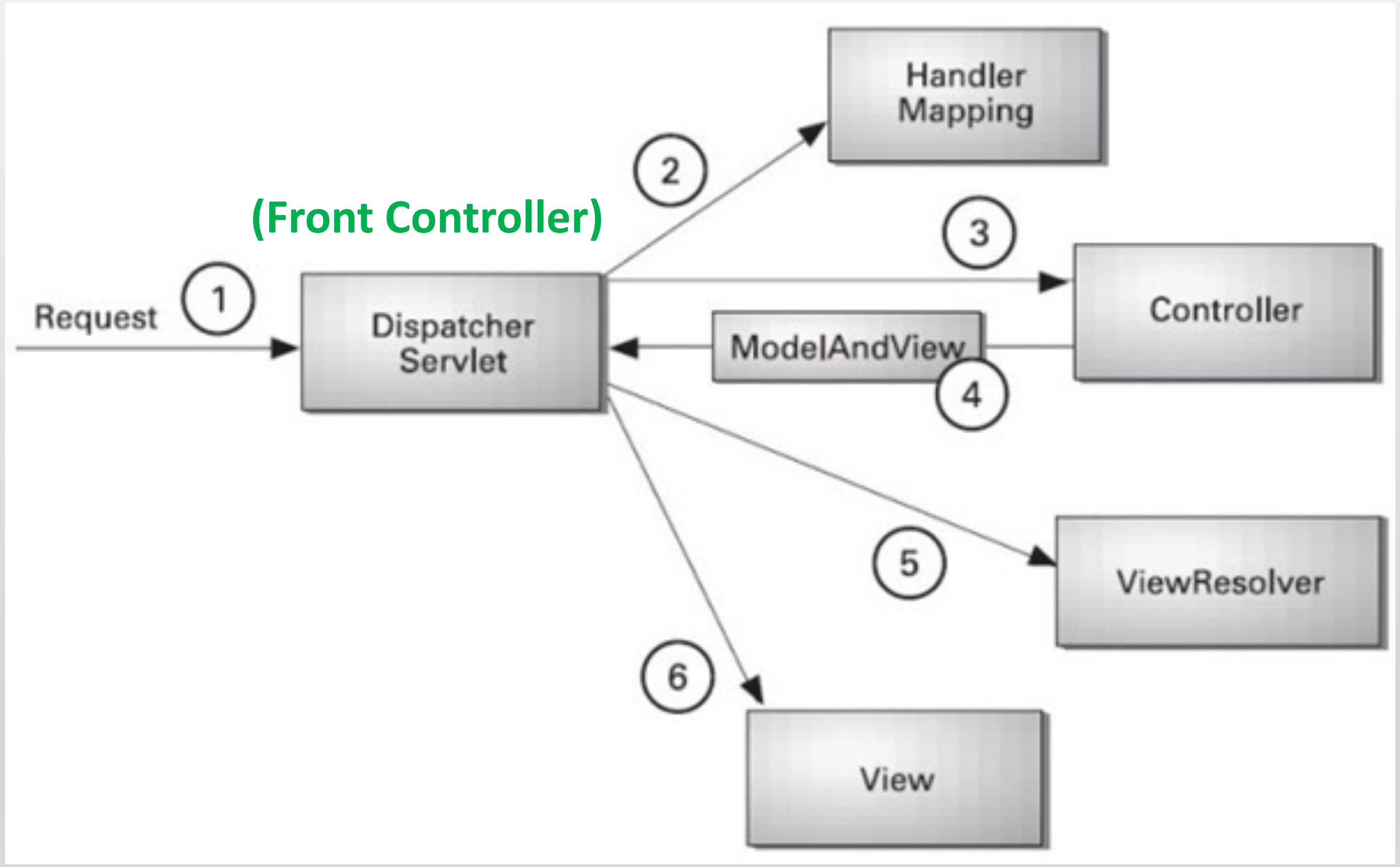
- spring.main.banner-mode
  - CONSOLE Print the banner to System.out.
  - LOG Print the banner to the log file.
  - OFF Disable printing of the banner.
- logging.level.<logger-name>=<level>
  - where level is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF.
  - Example
    - logging.level.root=warn
    - logging.level.org.springframework.web=off
    - logging.level.org.hibernate=error
- server.error.include-stacktrace
  - ALWAYS Always add stacktrace information.
  - NEVER Never add stacktrace information.
  - ON_PARAM
    Add stacktrace attribute when the appropriate request parameter is not "false".

https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

Spring Web MVC

- The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications.
- The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.
- A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet.
    - In Spring Web MVC, the **DispatcherServlet** class works as the **front controller**. It is responsible to manage the flow of the Spring MVC application.

The DispatcherServlet and Flow of Spring Web MVC



Defining a Controller

- The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it.
- The **@Controller** annotation indicates that a particular class serves the role of a controller.
- The @**RequestMapping** @**GetMapping** @**PostMapping** annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
public class HelloController {
   @RequestMapping("/hello")
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

Spring Boot Controller example

```java
@Controller
public class AppController {
    @Autowired
    private final StudentRepository studentRepository;

    @RequestMapping("/home")
    public String home() {
        return "home";
    }
    @GetMapping("/student-listing")
    public String students(Model model) {
        model.addAttribute("students", studentRepository.findAll());
        return "student-list";
    }
    @GetMapping("/student-list-plain-text")
    public ResponseEntity<String> students_list(Model model) {
        return new ResponseEntity<>(studentRepository.findAll()
             .toString(), HttpStatus.OK);
    }
```

Spring View Technology

- The Spring web framework is built around the MVC (Model-View-Controller) pattern, which makes it easier to separate concerns in an application.
- This allows for the possibility to use different view technologies, from the well established JSP technology to a variety of template engines.
  - Java Server Pages
  - Thymeleaf
  - FreeMarker
  - Groovy Markup Template Engine

Thymeleaf Template Engine example

```
<!DOCTYPE html>
<html lang="en“  xmlns=http://www.w3.org/1999/xhtml
         xmlns:th="http://www.thymeleaf.org">
<body>
<div class="container p4 m4">
    <h2>Student List:</h2><hr>
    <div class="row">
        <div class="col-2">Student Id</div>
        <div class="col-4">Name</div>
        <div class="col-2">GPAX</div>
    </div>
    <div class="row" th:each="student : ${students}">
        <div class="col-2" th:text="${student.id}"/>
        <div class="col-4" th:text="${student.name}"/>
        <div class="col-2" th:text="${student.gpax}"/>
    </div>
</div>
```

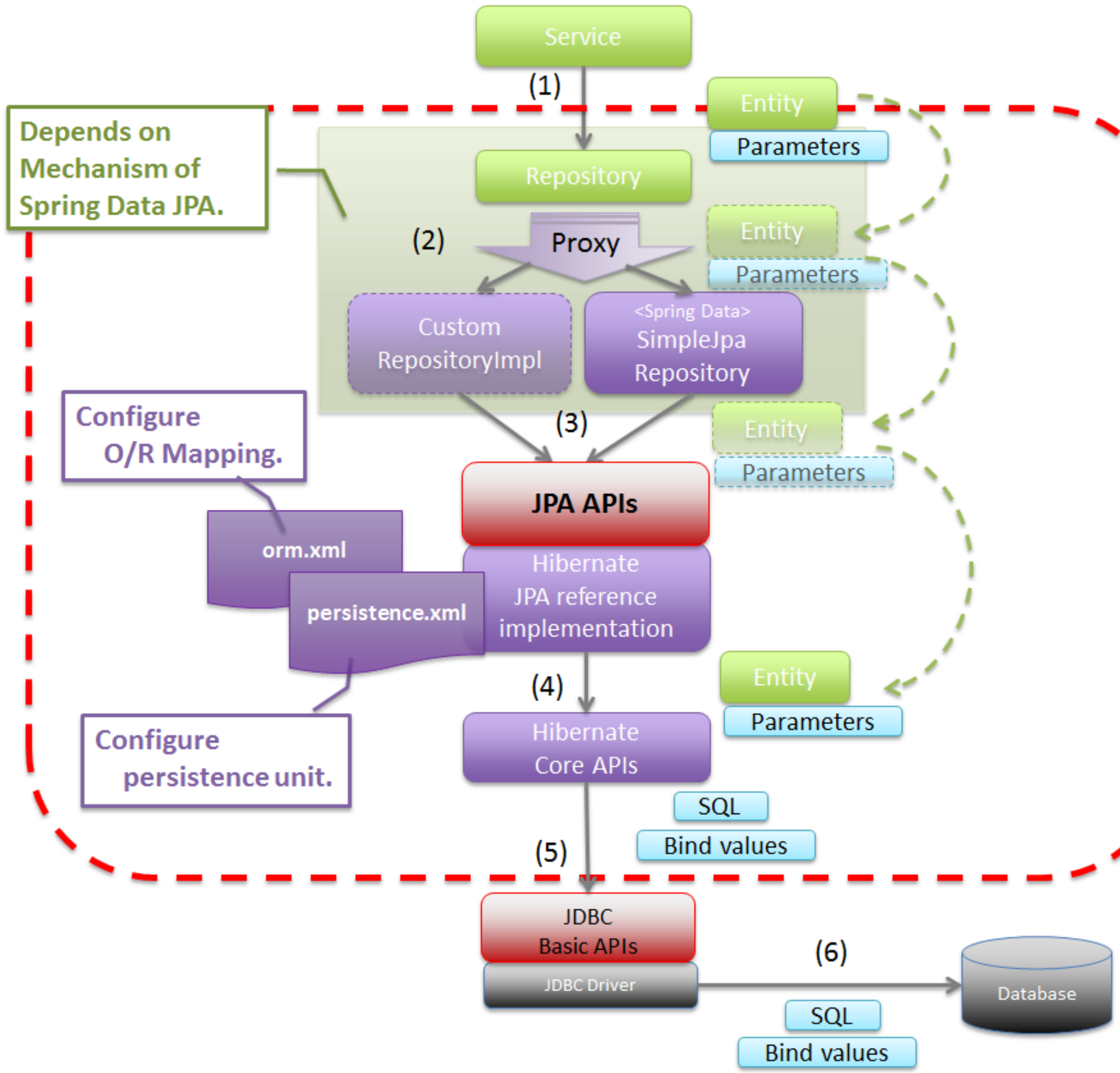
Spring Data JPA

- Managing data between java classes or objects and the relational database is a very cumbersome and tricky task.
- The DAO (Data Access Object) layer usually contains a lot of **boilerplate code** that should be simplified in order to reduce the number of lines of code and make the code reusable.
- Spring Data JPA:
  - This provides spring data repository interfaces which are implemented to create JPA repositories.
  - Spring Data JPA provides a solution to reduce a lot of boilerplate code.
  - Spring Data JPA provides an **out-of-the-box** implementation for all the required CRUD operations for the JPA entity so we don't have to write the same boilerplate code **again and again**.

```
public class CustomerRepository {
   private static final int PAGE_SIZE = 10;
   private EntityManager getEntityManager()
   public List<Customer> findAll() {..}
   public void save(Customer c) {..}
   public Product find(Integer cid) {..}
}
```

```
public class ProductRepository {
   private static final int PAGE_SIZE = 50;
   private EntityManager getEntityManager()
   public List<Product> findAll() {..}
   public void save(Product p) {..}
   public Product find(String pid) {..}
}
```

Basic Spring Data JPA Flow



JPA Repository Example

```java
@Getter @Setter @NoArgsConstructor
@AllArgsConstructor @ToString
@Entity
public class Student {
    @Id
    private Integer id;
    private String name;
    private Double gpax;
}
```

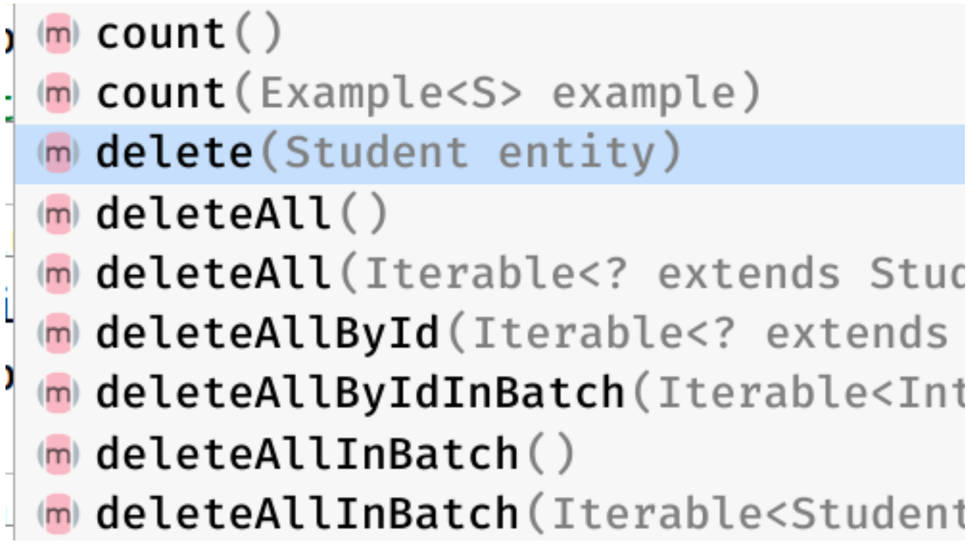
```java
import org.springframework.data.jpa.repository.JpaRepository;
import sit.int204.demo.entities.Student;

public interface StudentRepository extends JpaRepository<Student, Integer> {
    List<Student> findByNameContainsOrGpaxBetweenOrderByGpaxDesc(
            String name, double low, double high);
}
```

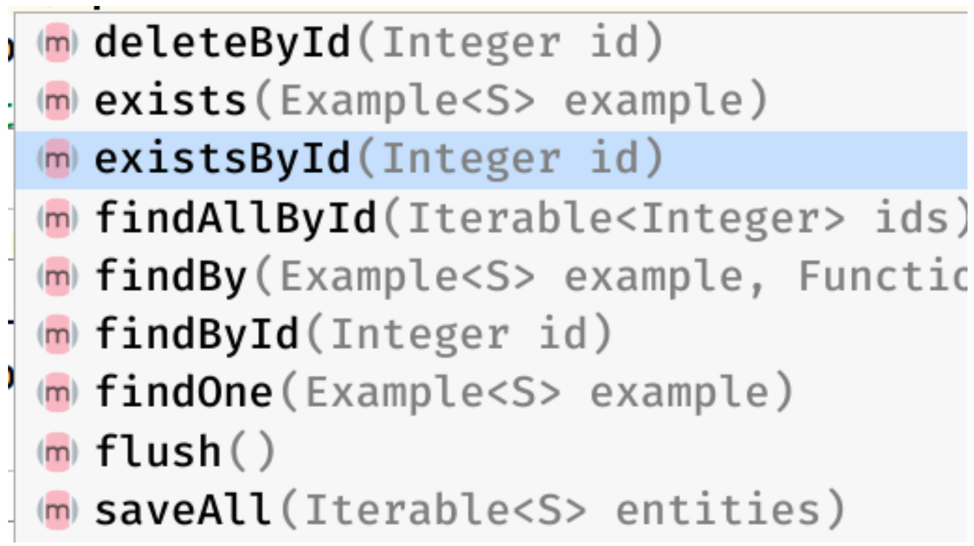
Query methods

Jpa Repository default methods

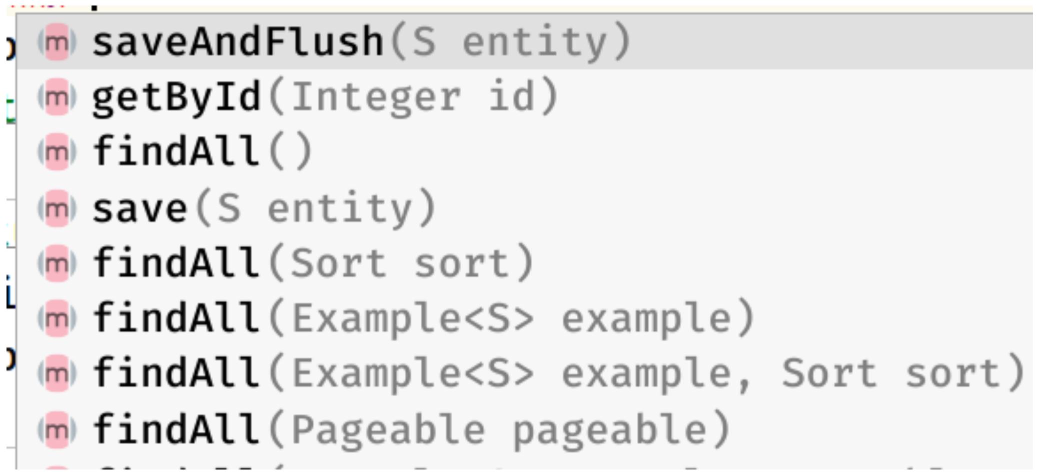
```
public class AppController {
    @Autowired
    private final StudentRepository
studentRepository;
```



```
count()
count(Example<S> example)
delete(Student entity)
deleteAll()
deleteAll(Iterable<? extends Stuc
deleteAllById(Iterable<? extends
deleteAllByIdInBatch(Iterable<Int
deleteAllInBatch()
deleteAllInBatch(Iterable<Student
```



```
deleteById(Integer id)
exists(Example<S> example)
existsById(Integer id)
findAllById(Iterable<Integer> ids)
findBy(Example<S> example, Functic
findById(Integer id)
findOne(Example<S> example)
flush()
saveAll(Iterable<S> entities)
```



```
saveAndFlush(S entity)
getById(Integer id)
findAll()
save(S entity)
findAll(Sort sort)
findAll(Example<S> example)
findAll(Example<S> example, Sort sort)
findAll(Pageable pageable)
```

Example: Data Jpa - count()

OfficeController

```
@GetMapping("/count")
public Count getOfficesCount() {
    return new Count(service.getOfficeCount());
}
```

OfficeService

```
public Long getOfficeCount() {
    return repository.count();
}
```

```
@Getter
@Setter
class Count {
    private long count;
    public Count(long n) {
        this.count = n;
    }
}
```

Exercise: Create REST API service for resources below

URI	HTTP Verb	Description
/customers	**GET**	**Get all customers**
/customers/{id}	**GET**	**Get a customer with id**
/customers/{id}/orders	**GET**	**Get all orders for customer id**
/customers	POST	Add new customers
/customers/{id}	PUT	Update a customers with id
/customers/{id}	DELETE	Delete a customers with id