SpringBoot CodingShuttle

Week 1:

1.3: Beans:

1.4 Dependency Injection:

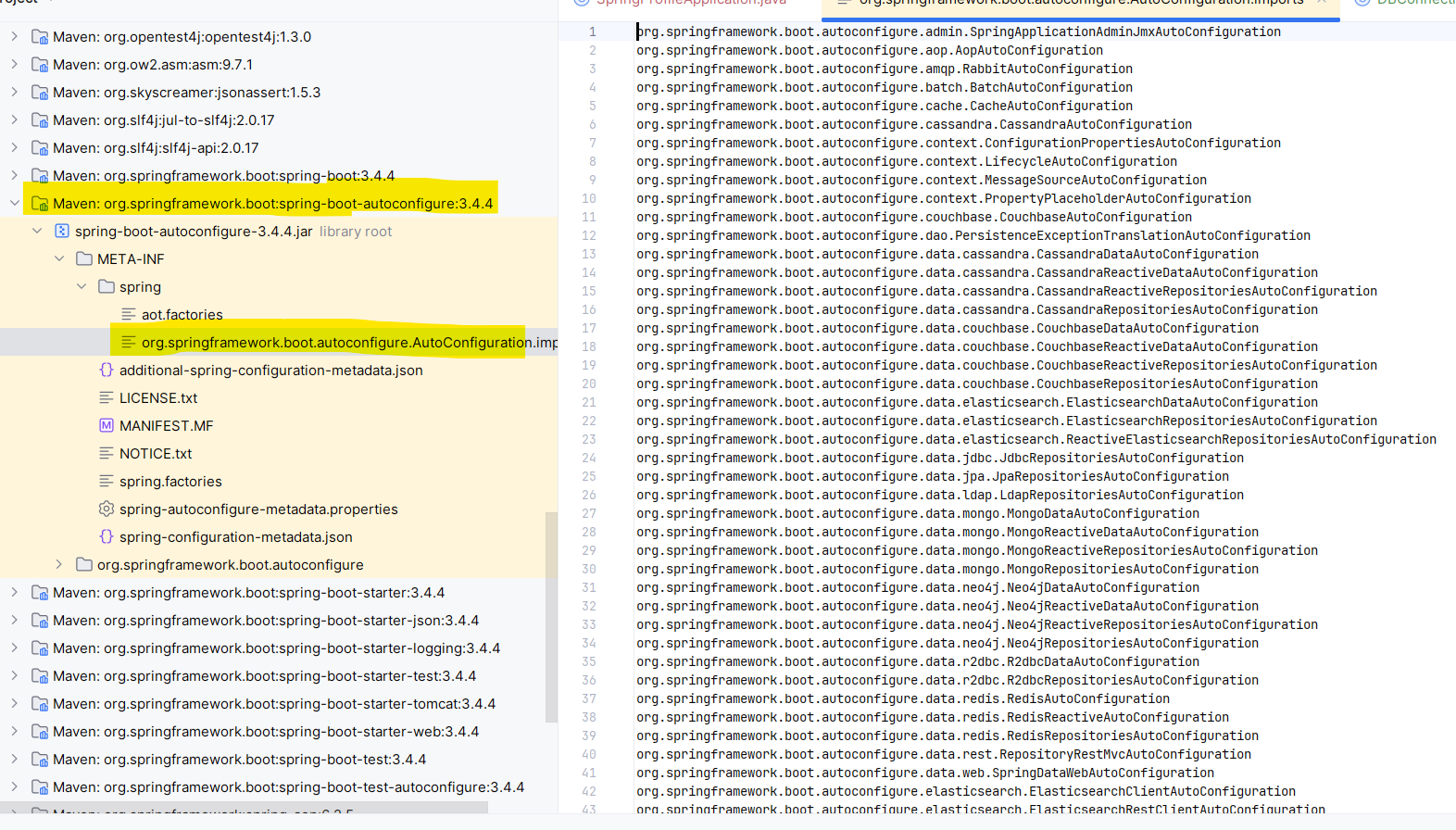
1.5 Spring VS SpringBoot

1.6 AutoConfiguration and Spring Boot Internal Flow:

In pom.xml all dependencies are specified.

Maven is popular build tool used in many java projects. In a Spring Boot Project, Dependencies are specified inside pom.xml file. Maven then resolves dependencies ( it load from mvn repository)and includes them in the class path.

This is where all AutoConfiguration are present: Around 152 autoconfiguration.



When you run the main method of Spring Boot Application Spring Boot will automatically register 17 of the property sources for us. Ex: application.properties , env-variables and many more .

So basically, SpringBoot is just a bunch of AutoConfiguration classes (== normal spring @Configurations), that creates @Beans for you if certain conditions are met!!

Spring Boot Internal Flow :

1. Initialization: when you start your spring boot application, the main entry point is typically a class annotated with @SpringBootApplication (or its meta-annotations). This annotation combines several other annotations such as @Configuration , @EnableAutoConfiguration and @ComponentScan.
2. Spring Application Context Creation: Spring Boot creates an application context, which serves as the container for managing beans and their dependencies. It scans the classpath for components, configurations, and auto-configurations and initializes the application context based on the detected classes and dependencies.
3. Auto-Configuration: Spring boot auto-configures beans and components based on the classpath and detected dependencies. It uses conditional annotations (@ConnditionalOnClass, @CondtionalOnBean etc.) to conditionally configure beans only if certain conditions are met.
4. Externalized Configuration: Spring Boot loads configuration properties from various sources such as property files, YAML files, env vars and command line arg. It provides sensible default values for configuration properties and allows them to be easily overridden or customized.
5. Embedded web server initialization
6. Application Startup: Spring Boot invokes lifecycle callbacks such as @PostConstruct methods and initialization callbacks on beans as the application context is being initialized. Beans are instantiated, dependencies are injected, and any necessary initialization logic is executed.
7. Application Ready: Once initialization process is complete, the application context is fully initialized and ready to handle requests. The embedded server is up and running and the application is ready to serve incoming HTTP requests.

1.7Maven Build Tool:

Week 2: Spring Boot MVC and RESTful APIs:

2.1: Introduction To Spring Boot Web MVC , MVC Architecture :

API’s : REST, SOAP, Web Sockets.

REST (Representational State Transfer) APIs (Application Programming Interfaces) are set of rules and conventions for building and interacting with web services.

REST API’s is stateless.

We can different request in REST API’s :

* GET /users : Retreve a list of all users.
* GET /users/{id} : Retrieve a specific user by ID
* POST /users : create a new user
* PUT /users/{id} : Update an existing user by ID
* PATCH /users/{id}: Partially update an existing user by ID
* DELETE /users/{id}: Delete a user by ID.

For REST API’s we need user Spring-boot-starter-web dependency:

Which contain following dependencies :

* Spring boot starter
* Jackson (JSON ⬄ Java Object)
* Spring-core
* Spring-mvc
* Spring-boot-starter-tomcat

Why to use MVC architecture:

1.Seperation of concerns 2. Reusability 3. Testability 4. Scalability (as code increases here it’s easy to expand)

2.2: Presentation Layer, DTO and Controllers:

1. Presentation Layer: It will write all logic to communicate between client and application. First point of contact between our client and spring boot application with the help of HTTP connection.

How client talk with application? As soon as you run your application server will starts . It goes to all the controllers and tell servlet dispatcher .

DispatcherServlet is a core component of the Spring MVC framework, acting as the front controller that handles all incoming HTTP requests. It receives requests and delegates them to other components for processing, such as controllers, and then returns the result as a response.

The workflow of DispatcherServlet involves the following steps:

* Receives an HTTP request.
* Determines the appropriate controller to handle the request using HandlerMapping.
* Invokes the selected controller through HandlerAdapter.
* Processes the controller's response, which might be a model and view.
* Resolves the view using ViewResolver.
* Renders the view and sends the response back to the client.

Annotated Controllers :

1. @RestController: It is used on top of controller class which accepts all the http request and return response. It is combination of @Controller + @ResponseBody: It means all method in the controller will return JSON/XML directly to the response body.
2. @RequestMapping : Annotation to map requests to controllers methods. It have various attributes to match by URL, HTTP method , request parameters ,headers and media types .

Class level:

You can use @RequestMapping on a class to define a base URI for all methods within that class. For instance, @RequestMapping("/myapp") at the class level would mean all methods inside that class would handle requests starting with /myapp/.

Method-Level:

You can also use @RequestMapping on a method to further refine the mapping for that specific method. For example, if you had @RequestMapping("/myapp") at the class level and @RequestMapping("/hello") on a method, then requests to /myapp/hello would be handled by that method.

@GetMapping:

To get the data two ways :

* 1. @PathVariable : /employees/123

Use path variables when the parameter is **essential** part of the URL path that defines the resource

*@GetMapping(path = "/employees/{employeeId}")*

*public EmployeeDTO getEmployeeByID(@PathVariable(required = false) Long employeeId){*

*return new EmployeeDTO(employeeId,"Sachin","sachin@gmail.com",24, LocalDate.of(2023,9,7),true);*

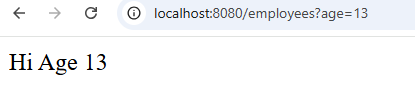
*}*

* 1. @RequestParam : /employees?id=123 (Query Parameter)

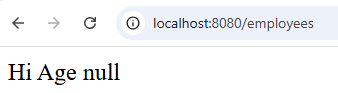
Use query parameters when the parameter is **optional** and used for filtering ,sorting or other modifications to the request.

http://localhost:8080/employees?age=13

@GetMapping(path = "/employees")  
public String getAllEmployees(@RequestParam(required = false) Integer age){  
 return "Hi Age "+ age;  
}



But is its optional if you don’t give then also it works:



Note: When you hit the browser all requests are type of get only. For post put patch you need to have some UI / frontend / client or Postman (web client) to test web API’s .

* 1. @RequestBody : It is used to bind the HTTP request body to Java Object . When a client sends data in the body of a request ( e.g. JSON or XML),@RequestBody maps this data to a java object.

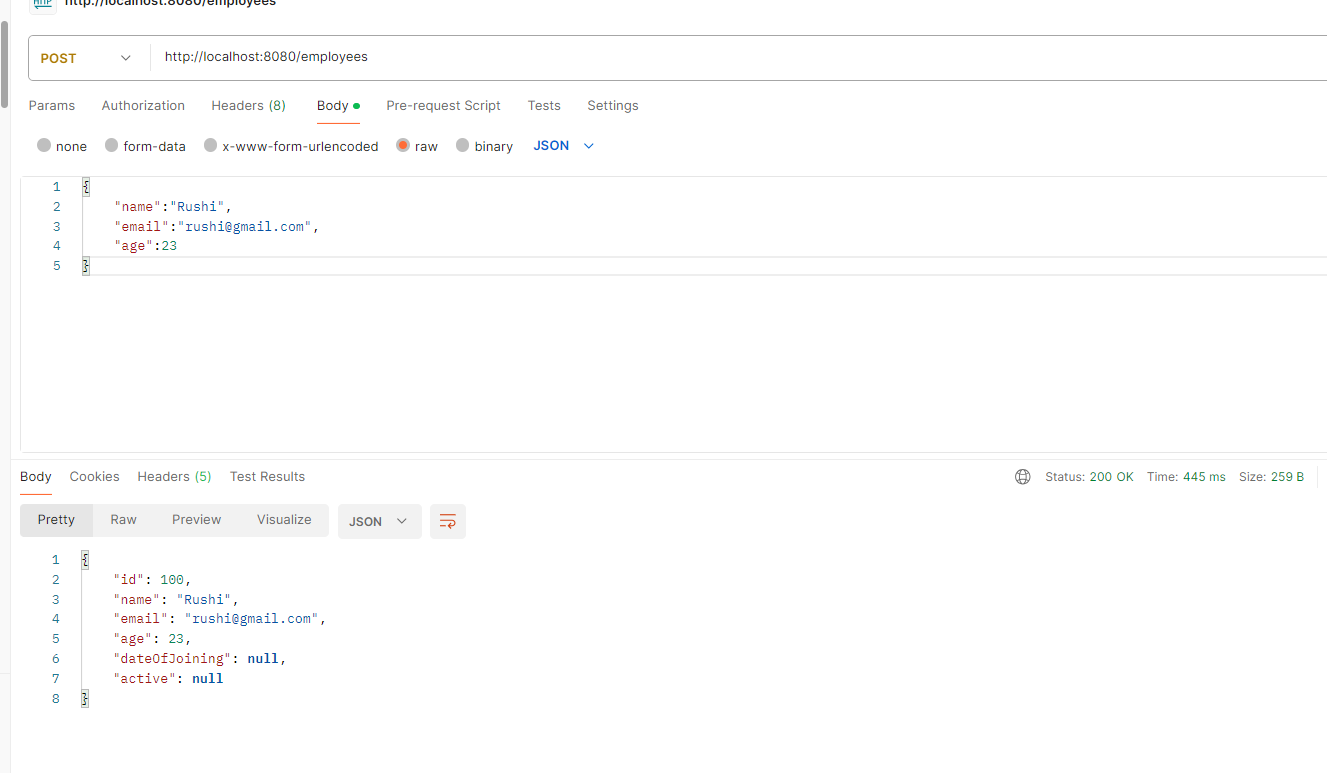
@PostMapping

public EmployeeDTO createNewEmployee(@RequestBody EmployeeDTO employeeDTO){

employeeDTO.setId(100L) ;//we are saying don't give id I will give

return employeeDTO;

}



-----------------------------------------------------------------------------------------------------------------

* 1. Persistence Layer & JPA:

See the architecture of JPA in Dig :

H2 is in memory database: Mainly, H2 database can be configured to run as inmemory database, which means that data will not persist on the disk. Because of embedded database it is not used for production development, but mostly used for development and testing.

We will need : starter-data-jpa dependency .

<dependency>

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

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

</dependency>

For Database we need use H2-database dependency :   
<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>  
</dependency>

Two interfaces : 1) CrudRepository<T,ID> 2)JpaRepository

**✅ 1. CrudRepository**

**Basic Interface** for CRUD (Create, Read, Update, Delete) operations.

**🔸 Key Features:**

* Basic operations:
  + save()
  + findById()
  + findAll()
  + deleteById()
* Minimal functionality — enough for simple use cases.

**🔹 Example:**

public interface EmployeeRepository extends CrudRepository<Employee, Long> {

}

**✅ 2. JpaRepository**

**Extends CrudRepository** and **PagingAndSortingRepository** with **extra JPA-specific features**.

**🔸 Key Features:**

* Inherits all CrudRepository methods.
* Adds:
  + flush() – sync changes immediately to DB
  + saveAllAndFlush()
  + findAll(Sort sort) – for sorting
  + findAll(Pageable pageable) – for pagination
  + deleteInBatch() and deleteAllInBatch()
* Better suited for complex applications and production systems.

**🔹 Example:**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

**🆚 Comparison Summary:**

| **Feature** | **CrudRepository** | **JpaRepository** |
| --- | --- | --- |
| Basic CRUD operations | ✅ Yes | ✅ Yes |
| Sorting | ❌ No | ✅ Yes |
| Pagination | ❌ No | ✅ Yes |
| Batch operations | ❌ No | ✅ Yes |
| JPA-specific methods | ❌ No | ✅ Yes |
| Recommended for real projects | ❌ Limited use | ✅ Preferred choice |

**✅ Conclusion:**

✅ Use **JpaRepository** if you want **full functionality**, especially for **pagination, sorting, and batch operations**.  
⚠️ Use **CrudRepository** only for **very simple** use cases with limited operations.

In Pom.xml Spring-boot-starter-parent will handle dependency and versions.

By default H-2 database is in memory database : It means once you close the application it will clear all the data present in that .

If you want to store that data in disk locally you can change the url :In application.properties : spring.datasource.url=jdbc:h2:file:/Users/sacshetk/SpringBoot\_0\_to\_1/week2\_2\_3/db

If you want to see h2 console :

spring.h2.console.enabled=true

If you want to add your custom username and password you can add :

spring.datasource.username=sa

spring.datasource.password=sa

If you want for first when it is running application you should do this :

spring.jpa.hibernate.ddl-auto=create

If you want from second time it should update existing table instead of creating new use following :

spring.jpa.hibernate.ddl-auto=update

Define this repository:

@Repository

public interface EmployeeRepository extends JpaRepository<EmployeeEntity,Long> {

}

To connect with repository :

1. findById(id) : return optional type entity : for get request
2. findAll() : for GET request
3. save(entity) : For post request

private final EmployeeRepository employeeRepository;

public EmployeeController(EmployeeRepository employeeRepository) {

this.employeeRepository = employeeRepository;

}

@GetMapping(path = "/{employeeId}")

public EmployeeEntity getEmployeeByID(@PathVariable(name = "employeeId") Long employeeId) {

return employeeRepository.findById(employeeId).orElse(null);

}

@GetMapping

public List<EmployeeEntity> getAllEmployees(@RequestParam(required = false) Integer age){

return employeeRepository.findAll();

}

@PostMapping

public EmployeeEntity createNewEmployee(@RequestBody EmployeeEntity employeeEntity){

return employeeRepository.save(employeeEntity);

}

For post request :

{

"name":"Subramanyam",

"email":"subramanyam@gmail.com",

"age":24,

"dateOfJoining":"2024-11-01",

"isActive":true

}

In entity class we need to do follow some things :

1. @Entity : To convert entity class into table
2. @Table : To change the name of table else it will give name to table as classname
3. @Id : To denote primary key
4. @GeneratedValue(strategy= GenrationType.AUTO)

@GeneratedValue is an annotation used in JPA to specify that the value of the primary key should **be automatically generated** (i.e., not set manually).

It works together with **@Id** (which marks a field as the primary key).

**⚙️ Strategy Options:**

|  |  |
| --- | --- |
|  |  |
| AUTO (default) | JPA picks the best strategy depending on the database |
| IDENTITY | Uses DB identity columns (e.g., AUTO\_INCREMENT in MySQL) |
| SEQUENCE | Uses database sequence object (best for Oracle/PostgreSQL) |
| TABLE | Uses a table to simulate sequence (less common, slower) |
| **💡 What Is Lombok?**  **Lombok** is a **Java library** that helps reduce **boilerplate code** in Java classes.  It does this by using **annotations** to automatically generate commonly used methods like:   * getters * setters * constructors * toString() * equals() and hashCode() * and more...   **✅ Why Use Lombok?**  Java code can be **verbose**. For example, if you have a class with 3 fields, you typically write:   * Getters * Setters * Constructor * toString()   That’s a lot of **repetitive code**.  With Lombok, you can avoid writing all that manually.  **🔧 Setup**   1. **Add Lombok to pom.xml** (if you're using Maven):   xml  CopyEdit  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <version>1.18.30</version> <!-- or latest -->  <scope>provided</scope>  </dependency>   1. **Enable annotation processing** in your IDE (important):    * In IntelliJ: Preferences → Build, Execution, Deployment → Compiler → Annotation Processors → ✅ Enable.   **🧪 Example Without Lombok:**  java  CopyEdit  public class User {  private String name;  private int age;  public User() {}  public String getName() { return name; }  public void setName(String name) { this.name = name; }  public int getAge() { return age; }  public void setAge(int age) { this.age = age; }  @Override  public String toString() {  return "User{name='" + name + "', age=" + age + "}";  }  }  **✅ Same Example With Lombok:**  java  CopyEdit  import lombok.Data;  @Data  public class User {  private String name;  private int age;  }  ➡️ This one annotation @Data automatically generates:   * Getters and setters * toString() * equals() and hashCode() * Required constructor   **📌 Common Lombok Annotations**   | **Annotation** | **Description** | | --- | --- | | @Getter | Generates getter methods | | @Setter | Generates setter methods | | @Data | Combines @Getter, @Setter, @ToString, etc. | | @NoArgsConstructor | Generates no-arg constructor | | @AllArgsConstructor | Generates constructor with all fields | | @Builder | Adds builder pattern support | | @ToString | Generates toString() | | @EqualsAndHashCode | Generates equals() and hashCode() |   **✅ Conclusion:**  Lombok helps write **clean, short, and readable Java code** by removing boilerplate. It's widely used in **Spring Boot projects** to keep entities, DTOs, and models simple. |  |

------------------------------------------------------------------------------------------------------------------

* 1. Service Layer, Writing our business logic:

Key role of service layer:

The service layer acts as bridge between the persistence layer (responsible for data access) and the presentation layer (handing user interaction).

It encapsulates the business logic of the application, shows interactions between different components, and provides a clean interface for external clients to interact with the system.

By abstracting away the complexities of data access and business operations, the service layer promotes modularity, maintainability, and scalability.

Important things : In service layer we don’t return entity to controller we return DTO for that we need to map Entity with DTO .

For that we need add dependency of Mapper in pom.xml and need to create Bean of that by using config :

Dependency :

<!-- https://mvnrepository.com/artifact/org.modelmapper/modelmapper -->

<dependency>

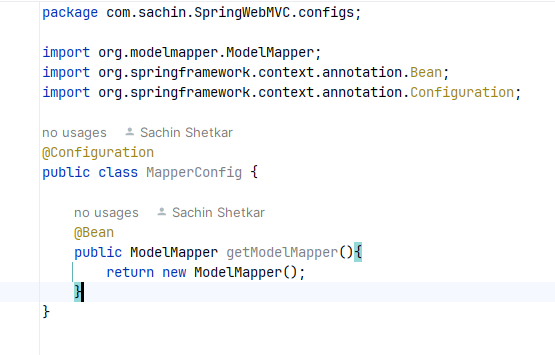
<groupId>org.modelmapper</groupId>

<artifactId>modelmapper</artifactId>

<version>3.0.0</version>

</dependency>

Config class :



We need use ModelMapper in now service :

ModelMapper has map method(source , target.class)

private final EmployeeRepository employeeRepository;  
  
**private final ModelMapper modelMapper;**  
public EmployeeService(EmployeeRepository employeeRepository, ModelMapper modelMapper) {  
 this.employeeRepository = employeeRepository;  
 this.modelMapper = modelMapper;  
}  
  
public EmployeeDTO getEmployeeById(Long employeeId) {  
 EmployeeEntity employeeEntity = employeeRepository.findById(employeeId).orElse(null);  
 return modelMapper.map(employeeEntity,EmployeeDTO.class);  
}  
  
public List<EmployeeDTO> getAllEmployees() {  
 List<EmployeeEntity> employeeEntities = employeeRepository.findAll();  
 return employeeEntities  
 .stream()  
 .map(employeeEntity -> modelMapper.map(employeeEntity,EmployeeDTO.class))  
 .collect(Collectors.*toList*());  
}  
  
public EmployeeDTO createNewEmployee(EmployeeDTO employeeDTO) {  
 EmployeeEntity toSaveEntity = modelMapper.map(employeeDTO,EmployeeEntity.class);  
 EmployeeEntity savedEmployeeEntity = employeeRepository.save(toSaveEntity);  
 return modelMapper.map(savedEmployeeEntity,EmployeeDTO.class);  
}  
  
  
public EmployeeDTO updateEmployeeById(EmployeeDTO employeeDTO, Long employeeId) {  
 EmployeeEntity employeeEntity = modelMapper.map(employeeDTO,EmployeeEntity.class);  
 employeeEntity.setId(employeeId);  
 EmployeeEntity savedEmployeeEntity = employeeRepository.save(employeeEntity);  
 return modelMapper.map(savedEmployeeEntity,EmployeeDTO.class);  
}

**When to Use ModelMapper:**

* To reduce boilerplate code for object mapping.
* In layered apps (like Spring Boot), where you separate:
  + Controllers (accept request DTO)
  + Services (use entity/domain objects)
  + Responses (return response DTO)

------------------------------------------------------------------------------------------------------------------