Table of Contents

Step 1: Understand Spring Boot

1. View the pom.xml file that has spring boot starter parent.

   It is a special starter project that provides default configurations for our application and a complete dependency tree to quickly build our Spring Boot project.
   It also provides default configurations for Maven plugins, such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, and maven-war-plugin.
   Beyond that, it also inherits dependency management from spring-boot-dependencies, which is the parent to the spring-boot-starter-parent.

2. @SpringBootApplication on the class with the main method:

   This annotation is used to enable three features, that is:
   a. @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
   b. @ComponentScan: enable @Component scan on the package where the application is located (see the best practices)
   c. @Configuration: allow to register extra beans in the context or import additional configuration classes

3. Run the application, you should see the error with respect to database configuration. This is the default behaviour of spring boot. It sees mysql dependency but did not find database connection parameter information.

4. Open application.properties file within the resources folder and add the database related connection parameters:

   # windows users use 3306 as port number
   #spring.datasource.url=jdbc:mysql://localhost:3306/java
   spring.datasource.url=jdbc:mysql://localhost:8889/java
   spring.datasource.username=root
   spring.datasource.password=root
   spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

   # below property specifies the dialect to use
   spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect

   spring.sql.init.platform=mysql

   # below property if used will perform DDL operations to create table
   spring.jpa.generate-ddl=true

   # below property if used will show the sql queries generated by hibernate
   spring.jpa.show-sql=true

5. Following the package structure is very important with spring boot for it to follow the default configurations.

Step 2: Understand DI in spring boot

6. Copy the NotificationService, IMessageProvider and StringMessageProvider along with annotations within com.boot.demo.service folder.

7. Now run the application and you will notice the beans created as output from constructor will be printed

8. To get the reference of spring context update the main method as follows:

   ApplicationContext context = SpringApplication.run(SpringBootDemo.class, args);

9. Now once you have the context, get the bean reference using context.getBean and will be able to call respective methods

Step 3: Create java entities and tables

1. Create below classes in the com.demo.entity folder with respective annotations to tell hibernate to map with database tables :

```
@Entity
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int aid;
    private String city;
    private String country;
    @Column(name=="pincode")
    private String zipcode;

    @OneToOne
    @JoinColumn(name="custid")
    private Customer customer;

    // getters and setters
    // constructor
}


@Entity
public class Customer {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @NotNull
        private int cid;
        private String cname;
        private String email;
        private String phone;
        @Size(min = 8, max=15)
```

```
	private String password;

	@OneToOne(mappedBy = "customer" , cascade = {CascadeType.REMOVE})
	private Address address;

	// getters and setters
	// constructor
}

@Entity
public class Book {

	@Id
	@GeneratedValue(strategy =  GenerationType.IDENTITY)
	private int bookid;
	private String title;
	private int year;

	@ManyToOne
	@JoinColumn(name="authorid")
	Author author;

	// getters and setters
	// constructor

	@Override
	public String toString() {
		return "Book [bookid=" + bookid + ", title=" + title + ", year=" + year + "]";
	}
}

@Entity
public class Author {

	@Id
	private int aid;
	private String name;
	@OneToMany(mappedBy = "author")
	Set<Book> books;
}
```

2. Once the entities are created, run the main class, see the console where it shows the queries generated by spring boot and see the database where tables are created

Step 4: CrudRepository, JpaRepository, PagingAndSortingRepository

1. By extending the interface, we get the most relevant CRUD methods for standard data access available in a standard DAO.

```
public interface AddressRepository extends JpaRepository<Address, Integer> {
}

public interface CustomerRepository extends JpaRepository<Customer, Integer>
{
}

public interface BookRepository extends JpaRepository<Book, Integer> {
}

public interface AuthorRepository extends JpaRepository<Author, Integer> {
}
```

Step 5: service layer

1. Create CustomerService class in service folder and inject respective dependencies for crud operations

```
@Service
public class CustomerService {

        @Autowired
        private CustomerRepository customerRepository;

        @Autowired
        private AddressRepository addressRepository;
}
```

Step 6: Insert Query

2. Create method in CustomerService class to insert customer record:

```
public Customer insertCustomer(Customer customer) {
        Customer savedCustomer = this.customerRepository.save(Customer);
        return savedCustomer;
}

// since save does both insert and update, if you need to use save only for insert
// then check if id exists and only then say save
public Customer insertCustomer(Customer Customer) {
    if (! CustomerRepository.existsById(Customer.getEid())) {
       Customer savedCustomer = this.CustomerRepository.save(Customer);
       return savedCustomer;
```

```
        }
        throw new EntityExistsException("Customer with id "+Customer.getEid()+"
already exists, hence cannot save");
    }
```

3. To test the above method in the main class, update with foll code:

```
CustomerService eservice = context.getBean(CustomerService.class);
Customer emp = new Customer();
emp.setEname("Shalini");
emp.setEmail("shalini@test.com");
emp.setPassword("shalini123");
emp.setPhone("9898989898");
eservice. insertCustomer(emp);
```

4. Create method in CustomerService class to insert address record. Since address cannot exist without Customer, hence need to check if Customer exist whose address need to be saved. If Customer exists, then address object should be associated with Customer id before insert.

```
public Address insertAddress(Address address, int empid) {
        Optional<Customer> optional = CustomerRepository.findById(empid);
        if(optional.isPresent()){
                // associate Customer and address
                address.setCustomer(optional.get());
                Address sAddress = this.addressRepository.save(address);
                return sAddress;
        }
        throw new EntityNotFoundException("Customer with id "+empid+" does not
                exist and hence cannpt save the addres");
}
```

5. To test the above method add below code in main method:

```
Address addr = new Address();
addr.setCity("Delhi");
addr.setCountry("India");
addr.setZipcode("989898");
// make sure Customer with id  1already exists, else it will throw exception
CustomerService.insertAddress(addr,1);
```

Step 7: Transaction Management

1. Add @EnableTransactionManagement in the class with main method
2. Below method is created if need to save Customer and address. If address is null then Customer record should not be inserted.

```
@Transactional
public Customer insertCustomerAndAddress(Customer Customer) {

if(this.CustomerRepository.existsById(Customer.getEid()))
```

```
                throw new EntityExistsException("Customer "+Customer.getEid()+" already
        exist");

                Customer savedCustomer = this.CustomerRepository.save(Customer);

                Address address = Customer.getAddress();
                if(address == null)
                        throw new RuntimeException("ERROR could not insert address");
                address.setCustomer(savedCustomer);
                Address savedAddress = this.addressRepository.save(address);
                savedCustomer.setAddress(savedAddress);

                return savedCustomer;
        }
```

test the above code in main method as follows:

```
Address addr = new Address();
addr.setCity("Delhi");
addr.setCountry("India");
addr.setZipcode("989898");

Customer emp = new Customer();
emp.setEname("Sia");
emp.setEmail("sia@test.com");
emp.setPassword("sia12323");
emp.setPhone("9898989898");
// Uncomment below line, Customer and address both will be saved. If commented since address
null, Customer will also be not inserted since we used @Transactional
//emp.setAddress(addr);
try {
        Customer e = CustomerService.insertCustomer(emp);
        }catch(Exception e) {
                System.out.println(e.getMessage());
                }
```

Step 8: Select Query

6. Update CustomerService class for select:

```
    public Customer getCustomerById(int eid) {
            System.out.println("Emp service "+eid);
            return this.CustomerRepository.findById(eid)
                    .orElseThrow(()-> new EntityNotFoundException("Customer "+eid+"
    not found"));
    }
```

test the above code in main method

```
    public List<Customer> getCustomers(){
```

```
        List<Customer> Customers = new ArrayList<Customer>();
        this.CustomerRepository.findAll().forEach(Customers::add);
        return Customers;
    }
```

test the above code in main method

Step 9: Update Query

7. Update CustomerService class for update:

```
    public Customer updateCustomer(Customer Customer) {
        if(!this.CustomerRepository.existsById(Customer.getEid()))
            throw new EntityNotFoundException("Customer
"+Customer.getEid()+" not found and cannot be updated");
        return this.CustomerRepository.save(Customer);
    }
```

test the above code in main method

Step 10: Delete Query

8. Update CustomerService class for delete:

```
    public boolean deleteCustomer(int eid) {
        if(!this.CustomerRepository.existsById(eid))
            throw new EntityNotFoundException("Customer "+eid+" not found
and cannot be deleted");
        this.CustomerRepository.deleteById(eid);
        return true;
    }
```

test the above code in main method

Step 11: JPA Query methods

1. To create your own queries based on different columns apart from the id , go through below link:

   https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

2. Lets create a jpa query to get Customer by email. Add below code in CustomerRepository interface

   ```
   public Customer findByEmail(String email);
   ```

3. Update CustomerService class to invoke the above method and return Customer by email

```
public Customer findCustomerByEmail(String email){
        return this.CustomerRepository.findByEmail(email);
}
```

4. To search for Customer address by Customer email, add below code in AddressRepository

```
public Address findByCustomerEname(String ename);
```

5. Create method in service class to test above method:

```
public Address findCustomerAddressByEmail(String email){
        return this.addressRepository. findByCustomerEname (email);
}
```

test the above code in main method

Step 12: @Query annotation

1. @Query annotation is used to create custom queries for which method signature is not available

2. To delete address of Customer by Customer id, need to retrieve the address id for the Customer and delete address. Update AddressRepository as follows:

```
@Query("select aid from Address where empid=:id")
  public Integer findAddressIdByCustomerEid(int id);
```

3. Update CustomerService as follows:

```
public boolean deleteAddressByCustomerId(int eid) {

    if(!this.CustomerRepository.existsById(eid))
            throw new EntityNotFoundException("Customer "+eid+" not found and
  cannot be deleted");

    Integer addressid = addressRepository.findAddressIdByCustomerEid(eid) ;
    System.out.println("address id "+addressid);
    if(addressid!= null){
            this.addressRepository.deleteById(addressid);
    return true;
}
return false;
}
```

4. To validate Customer credenitials valid or not when email and password is passed to the program, we need query that returns password for email id if it exists else returns null. Update the CustomerRepository as follows:

```
    @Query("select password from Customer u where email=:email")
```

```
public String findPasswordByEmail(String email);
```

5. Add below class in exception package:

```
public class InvalidCredentialsException extends Exception{
        public InvalidCredentialsException(String message) {
                super(message);
        }
}
```

6. UpdateCustomerService as follows:

```
public boolean loginCustomer(String email, String password) throws
InvalidCredentialsException
        {
                String pwd = this.CustomerRepository.findByEmail(email);
                if(pwd!=null)
                {
                        //Customer emp = opt.get();
                        if(pwd.equals(password))
                                return true;
                }
                throw new InvalidCredentialsException("Invalid credentials, Please try
        again");
        }
```

test the above code in main method

Step 13: Pagination and Sorting

1. To fetch Customers pagewise, update CustomerService as follows:

```
public Page<Customer> getFilteredCustomers(Integer pageno, Integer size){

        Pageable pageable = PageRequest.of(pageno, size);

        // for sorting
        //Pageable pageable = PageRequest.of(pageno, size,
//Sort.by(Sort.Direction.DESC, "email"));

        List<Customer> Customers = new ArrayList<Customer>();
        return this.CustomerRepository.findAll(pageable);

}
```

2. In main method just insert few dummy emp records as follows:

```
for(int i=1;i<=50;i++)
{
        Customer e1 = new Customer();
        e1.setEname("Emp "+i);
```

```
            e1.setEmail("emp"+i+"@test.com");
            e1.setPassword("emp"+i);
            e1.setPhone(i+"234567890");
            Address a1 = new Address();
            a1.setCity("city "+i);
            a1.setCountry("country "+i);
            a1.setZipcode(i+"234");
            e1.setAddress(a1);
            CustomerService.insertCustomer(e1);
        }
```

test the above code in main method as follows:

```
Page<Customer> pages = CustomerService.getFilteredCustomers(0, 5);
            System.out.println(pages.getNumberOfElements());
            System.out.println(pages.getTotalPages());
            System.out.println(pages.getTotalElements());
            for(Customer emp :pages.getContent())
                System.out.println(emp);
```

Step 14: Appendix

JPA ANNOTATIONS

1. @Entity : used at the class level and marks the class as a persistent entity. It signals to the JPA provider that the class should be treated as a table in the database

2. @Id: indicating the member field below is the primary key of the current entity.

3. @ GeneratedValue : This annotation is generally used in conjunction with @Id annotation to automatically generate unique values for primary key columns within our database tables.

4. The @GeneratedValue annotation provides us with different strategies for the generation of primary keys which are as follows :
    a. GenerationType.IDENTITY: This strategy will help us to generate the primary key value by the database itself using the auto-increment column option. It relies on the database's native support for generating unique values.

    b. GenerationType.AUTO: This is a default strategy and the persistence provider which automatically selects an appropriate generation strategy based on the database usage.

    c. GenerationType.TABLE: This strategy uses a separate database table to generate primary key values. The persistence provider manages this table and uses it to allocate unique values for primary keys.

    d. GenerationType.SEQUENCE: This generation-type strategy uses a database sequence to generate primary key values. It requires the usage of database sequence objects, which varies depending on the database which is being used.

5. @Column : used to customize the mapping of a specific field to a database column. While JPA automatically maps most fields based on naming conventions, the @Column annotation provides developers with greater control over the mapping process.

6. @OneToOne: is used to associate one JAVA object with another JAVA object.

7. @JoinColumn annotation is used to define a foreign key with the specified name

8. @JoinTable annotation in JPA is used to customize the association table that holds the relationships between two entities in a many-to-many relationship. This annotation is often used in conjunction with the @ManyToMany annotation to define the structure of the join table.