Introduction to Hibernate and CRUD [3199]

What is JDBC?

JDBC stands for Java Database Connectivity, which provides low-level, database-dependent connectivity between the Java programming language and a wide range of databases.

Advantages of JDBC:

- Send queries and update statements to the database.
- Enables a Java application to communicate with a database.
- Retrieve and process the results received from the database in answer to your query.

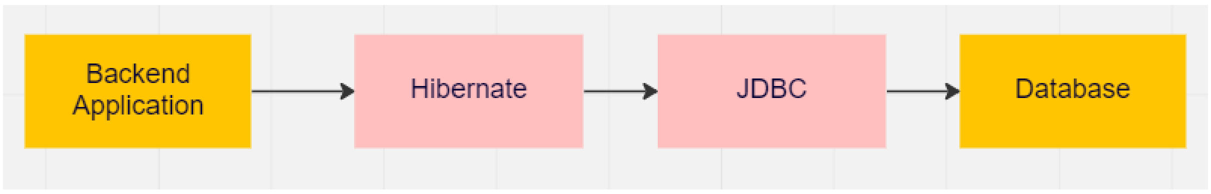
What is Hibernate in JAVA?

Hibernate provides a standardized and efficient way of storing and retrieving data in a relational database using Java. It facilitates mapping Java objects to database tables through **Object-Relational Mapping (ORM)**. Hibernate implements the **Java Persistence API (JPA)** specification, which sets standards for persisting data in Java applications.

What is Persistence?

Persistence refers to the ability of data to be stored and retrieved from a database even after the Java application that created it has been closed. Persistence is achieved through various technologies such as ORM, JPA, and frameworks like Hibernate.

Working of Hibernate



When an application requests data from the database, Hibernate retrieves the data using JDBC and maps it to **Java objects**, which are then returned to the application. When an application saves data to the database, Hibernate takes the Java objects, maps them to database tables, and persists the data using JDBC.

Overall, Hibernate simplifies database interactions by providing a higher-level API and abstracting the low-level details of JDBC and SQL.

What is the need for Hibernate?

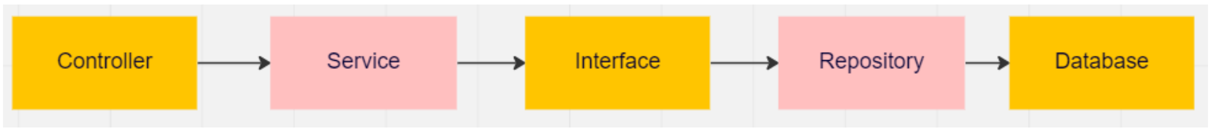
Some of the advantages of Hibernate are

1. Hibernate offers automated and **efficient mapping** of Java objects to database tables through ORM.
2. It also generates **database-independent queries**. So you don't need to write database-specific queries like you will have the same query for MySQL database as for PostgreSQL Database.
3. It simplifies database operations and reduces the boilerplate code needed for database interactions, making it easier to develop and maintain database-driven applications.
4.

Hibernate Annotations

These annotations define the mapping between Java objects and database tables and are placed on the Java classes, fields, and methods.

Components of a SpringBoot Application



The flow of components in a SpringBoot application is as follows:

Controller: Receives a request from the user, passes it to the service layer for processing, and returns the response to the user.

Service: Processes the request from the controller, interacts with the repository to retrieve or persist data, and returns the result to the controller.

Repository: Acts as an interface to the database, performing operations such as retrieving, updating, and deleting data and returning the result to the service layer.

Interface: Defines the methods and properties the repository and service layer must implement, serving as a blueprint for these components.

Database: Persists the data for the application and provides a way for the repository to store and retrieve data.

We will now discuss the different annotations used across the four layers of a SpringBoot Application, i.e., Model, Service, Controller, and DAL Layer.

Database: Model Layer

Model layer is the part of the application that interacts with the database using Hibernate. It comprises Java objects representing the application's data entities, such as items, orders, or products. These objects are known as **entities.**

We will now go through the various annotations used in the Model layer to map a class in java to a table in the database.

@Entity

The @Entity annotation in Hibernate indicates that a class represents a database table. When Hibernate encounters an @Entity annotated class, it maps the class and its properties to a database table with the same name as the class.

```
package com.cn.cnkart.entity;
import javax.persistence.*;
@Entity
public class Item {
}
```

@Id

The @Id annotation in Hibernate is used to indicate the primary key column of a database table. The primary key is a unique identifier for each record in a database table and is used to locate a specific record.

For example

```
import javax.persistence.Id;
@Entity
public class Item {
    @Id
    private int id;
}
```

In this example, the id field is annotated with @Id, indicating that it represents the primary key column of the "Item" table in the database.

@GeneratedValue & @GenerationType

The @GeneratedValue annotation in Hibernate automatically generates **unique values** for a primary key column in a database table. It is used in conjunction with the @Id annotation.

Generation strategies can use four **@GenerationType**:

- **AUTO**: Hibernate selects the strategy based on the used dialect.

- **IDENTITY**: Hibernate relies on an auto-incremented database column to generate the primary key,
- **SEQUENCE**: Hibernate requests the primary key value from a database sequence.
- **TABLE**: Hibernate uses a database table to simulate a sequence.

For example,

```
import javax.persistence.*;
import javax.persistence.*;
@Entity
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
}
```

In this example, @GeneratedValue indicates that Hibernate should automatically generate values of the id column.

@Column

The @Column annotation in Hibernate maps **a column** in a Java class to a field in a database table. The annotation is used to specify the column's name in the database table.

```
package com.cn.cnkart.entity;
import javax.persistence.*;
@Entity
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column
    private int id;
    @Column(name="item_name")
    private String name;
    @Column(name="item_description")
    private String description;
}
```

The name field is annotated with @Column, indicating that it should be mapped to a column in the "Item" table in the database. The annotation specifies the name of the column as "item_name."

Note: The @Column annotation is optional; if not specified, Hibernate will use the field's name as the column's name.

Controller Layer

Controller layer is responsible for receiving user requests and coordinating with the Hibernate layer to perform database operations, for example, creating, reading, updating, or deleting records in the database. The Controller layer acts as a mediator between the user interface and the database.

@RestController

This annotation indicates that this class is a Spring controller and is responsible for handling RESTful web requests. It is a combination of two annotations:

1. **@Controller** - This @Controller annotation marks the class as a controller class.
2. **@ResponseBody** - This annotation indicates that the response from the controller method should be sent to the client as the body of the HTTP response.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;

@RestController
public class ItemController {
}
```

@RequestMapping

This annotation is used to map HTTP requests to handler methods of MVC and REST controllers. To configure the mapping of web requests, we use the @RequestMapping annotation. The @RequestMapping annotation is applied to the controller class.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/item")
public class ItemController {
}
```

@RequestMapping maps HTTP requests to the "/item" URL to the methods in this class. This class's methods will handle requests that start with "/item."

@GetMapping

@GetMapping annotation maps HTTP GET requests onto specific handler methods. It is a composed annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.GET)**. The @GetMapping annotated methods handle the HTTP GET requests matched with the given URI expression.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/item")
public class ItemController {
    @GetMapping("/id/{id}")
    public Item getItemById(@PathVariable int id) {
    }
}
```

Here, @GetMapping maps the HTTP GET request with the specified URL pattern to this method. The pattern "/id/{id}" includes a path variable {id}, which represents a placeholder for a dynamic value that can be passed in the request URL.

The **@PathVariable** annotation tells Spring to extract the value of the id path variable from the URL and pass it as a parameter to the method.

@PostMapping

The @PostMapping is a specialized version of @RequestMapping annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.POST)**. The @PostMapping annotated methods handle the HTTP POST requests matched with the given URI expression.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/item")
public class ItemController {
    @PostMapping("/save")
    public void saveItem(@RequestBody Item item){
    }
}
```

@PostMapping annotation maps the HTTP POST request with the specified URL pattern "/save" to this method. The **@RequestBody** annotation tells Spring to extract the request body of the incoming HTTP POST request and map it to the item parameter.

@Delete Mapping

The @DeleteMapping is a specialized version of @DeleteMapping annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.DELETE)**. The @DeleteMapping annotated methods handle the HTTP DELETE requests matched with the URI expression.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/item")
public class ItemController {
    @DeleteMapping("/delete/id/{id}")
    public void deleteItem(@PathVariable int id){
    }
}
```

@DeleteMapping annotation maps the HTTP DELETE request with the specified URL pattern "/delete/id/{id}" to this method. The pattern "/delete/id/{id}" includes a path variable {id}, which represents a placeholder for a dynamic value that can be passed in the request URL. **@PathVariable** annotation tells Spring to extract the value of the id path variable from the URL and pass it as a parameter to the method.

@Put Mapping

The @PutMapping is a specialized version of @PutMapping annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.PUT)**. The @PutMapping annotated methods handle the HTTP PUT requests matched with the given URI expression.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/item")
public class ItemController {
    @PutMapping("/update")
    public void updateItem(@RequestBody Item updateItem)
    {
        itemService.update(updateItem);
    }
}
```

@PutMapping annotation maps the HTTP PUT request with the specified URL pattern "/update" to this method. The **@RequestBody** annotation tells Spring to extract the request body of the incoming HTTP POST request and map it to the item parameter.

Service Layer

In Hibernate, the service layer manages **the persistence** of data to and from the database using Hibernate's ORM capabilities. The service layer acts as a bridge between the database and the application's business logic.

The service layer provides logic to operate on the data sent to and from the DAO and the client. Another user of the service layer is **security**, and if you provide a service layer that has no relation to the DB, then it is more difficult to gain access to the DB from the client except through the service.

For example,

```java
package com.cn.cnkart.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import com.cn.cnkart.service.ItemService;

@RestController
@RequestMapping("/item")
public class ItemController {

    @Autowired
    ItemService itemService;

    @GetMapping("/id/{id}")
    public Item getItemById(@PathVariable int id) {
        return itemService.getItemById(id);
    }
}
```

@Autowired annotation to injects an instance of ItemService into the controller. ItemService object calls its “getItemById” method, which is now responsible for retrieving an item from some persistent storage, such as an Item entity by its id.

@Service

The @Service annotation is a marker annotation in the SpringBoot framework that indicates that a class is a service component. By annotating a class with @Service, you indicate to Spring that it should consider the class as a candidate for component scanning.

For example,

```
package com.cn.cnkart.service;

import org.springframework.beans.factory.annotation.*;

@Service
public class ItemService {
}
```

@Transactional

This annotation is used to declare a method or class as transactional in the SpringBoot framework. By annotating a method or class with @Transactional, you indicate to Spring that the operations performed within the method or class should be executed as a single atomic transaction.

This means that either all operations will be executed and committed to the database, or none will be executed, and the database will be rolled back to its previous state.

For example,

```
package com.cn.cnkart.service;
import javax.transaction.Transactional;
import org.springframework.beans.factory.annotation.*;

@Service
public class ItemService {
    @Transactional
    public Item getItemById(int id) {
        return itemDAL.getById(id);
    }
}
```

In this example, when the “getItemById” method is called, it will be executed within the scope of a database transaction, ensuring that If any of the database operations fail, the entire transaction will be **rolled back**. The database will be returned to its previous state.

Interface: DAL Layer

The Data Access Layer (DAL) in Hibernate refers to the components responsible for communicating with the database to perform CRUD (Create, Read, Update, and Delete) operations.

We will create the interface for DAL Layer,

```
package com.cn.cnkart.dal;
```

```
import com.cn.cnkart.entity.Item;

public interface ItemDAL {

    Item getById(int id);

    void save(Item item);

    void delete(int id);

    void update(Item updateItem);

}
```

Repository: DAL Implementation Layer

The implementation class of DAL Interface typically contains the classes and methods that handle the mapping of objects to database tables, the execution of database queries, and the management of transactions.

@Repository

The annotation indicates that a class is a repository, a component that provides access to the data stored in a database.

For example,

```
package com.cn.cnkart.dal;

import org.springframework.stereotype.Repository;

@Repository
public class ItemDALImpl implements ItemDAL{
}
```

Entity Manager

Is used to create **sessions** that interact with a database. The Entity Manager helps you manage interactions with the database by providing methods for performing database operations, creating queries, and managing transactions. The sessions created by the Entity Manager allow you to perform database operations consistently and reliably.

Session Methods: GET

```
package com.cn.cnkart.dal;

```

```java
import javax.persistence.EntityManager;
import org.hibernate.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import com.cn.cnkart.entity.Item;


@Repository
public class ItemDALImpl implements ItemDAL{
    @Autowired
    EntityManager entityManager;

    @Override
    public Item getById(int id) {
        Session session = entityManager.unwrap(Session.class);
        Item item = session.get(Item.class, id);
        return item;
    }

}
```

The EntityManager is injected into the class using the **@Autowired** annotation. This annotation tells Spring to automatically create an instance of the EntityManager and inject it into the entityManager field.

The "unwrap" method of EntityManager returns the underlying **Hibernate Session** from the EntityManager, which can be used to perform additional operations not available through EntityManager. Once the Hibernate Session is obtained, the get method is called on the Session, passing in the "**Item.class**" and the object's id to retrieve. This method retrieves the Item object with the specified id from the database and returns it.

Session Methods: GET-ALL

```java
@Override
public List<Item> getAllItems() {
     Session session = entityManager.unwrap(Session.class);
     //Pass your query, and name of class on which you want to perform   the query
in createQuery method.
     Query<Entity> query = session.createQuery("from Item",Entity.class);
     // Save all item in the list
     List<Item> itemList = query.getResultList();
     return itemList;
}
```

To fetch all Items from the database we can use createQuery() method. To learn more about this method, you can refer to this link.

Session Methods: POST

```
@Override
public void save(Item item) {
    Session session = entityManager.unwrap(Session.class);
    session.save(item);
}
```

Similarly, you can use the **"session.save(item)"** method to save an Item entity into the database.

Session Methods: DELETE

```
@Override
public void delete(int id) {
    Session session = entityManager.unwrap(Session.class);
    Item item = session.get(Item.class, id);
    session.delete(item);
}
```

The Entity Manager is used to obtain a Hibernate Session, which is then used to retrieve the Item object from the database. Then, you can use the "**session.delete(item)**" method to delete the Item entity from the database.

Session Methods: UPDATE

```
@Override
public void update(Item updateItem) {
    Session session = entityManager.unwrap(Session.class);
    //fetch the current item details from the DB
    Item currentItem = session.get(Item.class, updateItem.getId());

    //update the details in the current object
    currentItem.setDescription(updateItem.getDescription());
    currentItem.setName(updateItem.getName());
    //update the database
    session.update(currentItem);
}
```

The Entity Manager is used to obtain a Hibernate Session, which is then used to update the Item object from the database. First, the "get" method retrieves the current Item object with the specified id from the database. Finally, the update method is called on the Session, passing in the current Item object. The **"session.update(currentItem)"** method updates the Item object in the database with the new values.

Note: When you call the update method, Hibernate will synchronize the state of the persistent object with the database, updating the database record to reflect any changes made to the object. For this to work, the object must already be associated with a Hibernate session and have a corresponding database record.

Conclusion

In this lesson, we covered the basics of Hibernate and JDBC. We learned about the different layers of a Spring Boot application and the various annotations used in each layer. We also discussed the concept of sessions and how they can be used to perform CRUD (Create, Read, Update, and Delete) operations on a database.

Instructor Codes

- CNkart Application

References

1. JDBC
2. Hibernate
3. Hibernate Session Methods