Table of Contents

Modify the code to fetch data from database

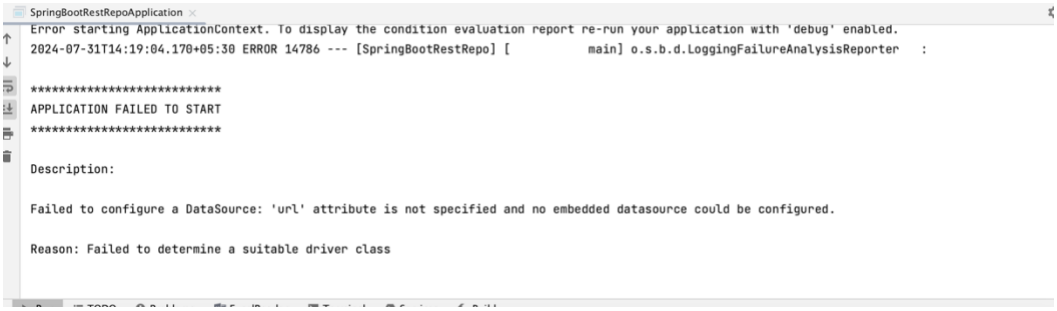
Step 1: Update properties file for Db connection parameters

1. **Open database and make sure to create database named "neueda"**

2. Add below dependency in pom.xml . **DO NOT FORGET TO RELOAD MAVEN**

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>com.mysql</groupId>
   <artifactId>mysql-connector-j</artifactId>
   <scope>runtime</scope>
</dependency>
```

3. Now just run the main method and you should see below error:



Spring needs the connection parameters for database.

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

MAKE SURE CREDENTIALS ARE AS PER YOUR DATABASE

```
spring.datasource.url=jdbc:mysql://localhost:3306/neueda
spring.datasource.username=root
spring.datasource.password=c0nygre
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

# below property if used will perform DDL operations to create table
# create will always drop and recreate
# update will create only if none exists
spring.jpa.hibernate.ddl-auto=create

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

// will decorate the console output
spring.output.ansi.enabled=ALWAYS

Step 2: Map java class to database table so that it becomes **"DATABASE MANAGED ENTITY"**

1. To let hibernate [which is part of spring boot data jpa] know how to map this class to a database table and create the table accordingly, update the Book class for respective annotations:

```
// tells to treat this class as database table.
@Entity

//[Optional ] if used will create the table with the name as book_table else maps with classname
//@Table(name = "book_table")

public class Book {
// Used for primary key identifier
  @Id
 // to generate auto-increment values use Identity strategy
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int bookid;
  // rest remains the same
}
```

2. Run the main method and you should get below error:

```
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:1352) ~[spring-boot-3.3.2.jar:3.3.2]
    at com.training.SpringBootDatabaseRepo.SpringBootDatabaseRepoApplication.main(SpringBootDatabaseRepoApplication.java:10) ~[classes/:na]
Caused by: java.sql.SQLSyntaxErrorException Create breakpoint : You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version fo
 the right syntax to use near 'desc varchar(255), title varchar(255), primary key (bookid)) engine=InnoDB' at line 1
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:121) ~[mysql-connector-j-8.3.0.jar:8.3.0]
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122) ~[mysql-connector-j-8.3.0.jar:8.3.0]
    at com.mysql.cj.jdbc.StatementImpl.executeInternal(StatementImpl.java:770) ~[mysql-connector-j-8.3.0.jar:8.3.0]
    at com.mysql.cj.jdbc.StatementImpl.execute(StatementImpl.java:653) ~[mysql-connector-j-8.3.0.jar:8.3.0]
    at com.zaxxer.hikari.pool.ProxyStatement.execute(ProxyStatement.java:96) ~[HikariCP-5.1.0.jar:na]
    at com.zaxxer.hikari.pool.HikariProxyStatement.execute(HikariProxyStatement.java) ~[HikariCP-5.1.0.jar:na]
    at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase.java:80) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
    ... 38 common frames omitted
```

3. The reason is desc is a keyword in database. Hence use @Column to specify the name of column to be created in database table. Update class as follows:

```
// default it creates column with length 255 and allows null. Modify using attributes as below
@Column(length = 100, nullable = false)
private String title;
@Column(length = 100, nullable = false)
private String author;
@Column(name = "description")
private String desc;
// Other parts remain the same
}
```

MAKE SURE TO RESTART THE SERVER

4. Now if you run main method again, you should see table created in database.
Check description : default 255 and null

5. Once the entities are created, run the main class, see the console where it shows the queries generated by spring boot since we used show-sql as true and see the database where tables are created

```
Hibernate: drop table if exists book
Hibernate: create table book (bookid integer not null auto_increment, price float(53) not null, author varchar(100) not null, title varchar(100) not null,
 description varchar(255), primary key (bookid)) engine=InnoDB
2024-07-30T12:23:50.772+05:30  INFO 48765 --- [SpringBootDatabaseRepo] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA
 EntityManagerFactory for persistence unit 'default'
```

Step 3: JpaRepository

1. Once the tables are created you need to perform CRUD operations. Spring boot provides an interface for the same.
   [GOOD NEWS!!! – NO NEED TO WRITE SQL QUERIES 😊]

2. Just extend the interface and we get the most relevant CRUD methods for standard data access available in a standard DAO.

```
public interface BookRepo extends JpaRepository<Book, Integer> {
}
```

3. JpaRepository<T,ID> is generic interface where
   T : represents the database managed entity name
   ID: represents the type of identifier.

Step 4: service layer

1. Create BookServiceRepo class to perform business operations on the book table as follows:

```
package com.training.SpringBootDatabaseRepo.service;

import com.training.SpringBootDatabaseRepo.entity.Book;
import com.training.SpringBootDatabaseRepo.repo.BookRepo;
import jakarta.persistence.EntityExistsException;
import jakarta.persistence.EntityNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class BookServiceRepo {
  @Autowired
  private BookRepo bookRepo;

  public long getTotalBookCount(){
    return bookRepo.count();
  }
public List<Book> getAllBooks(){
    return bookRepo.findAll();
  }
  public Book addNewBook(Book book){
    if(bookRepo.existsById(book.getBookid()))
      throw new EntityExistsException("Cannot add "+book.getBookid()+" already exists");
    return bookRepo.save(book);

  }
  public Book updateBook(Book book){
    if(!bookRepo.existsById(book.getBookid()))
      throw new EntityNotFoundException("cannot update "+book.getBookid()+" does not exist");
    return bookRepo.save(book);
  }
  public boolean deleteBook(int id){
    if(!bookRepo.existsById(id))
      throw new EntityNotFoundException("cannot delete "+id+" does not exist");
    bookRepo.deleteById(id);
    return true;
  }
  public List<Book> getBooksByAuthor(String author){
    return null;
  }
```

```
    public Book getBookById(int id){
      if(!bookRepo.existsById(id))
        throw new EntityNotFoundException(id+" not found");
      return bookRepo.findById(id).get();
    }
  }
```

2. Just update the BookRestController class Autowired dependency from BookService to BookServiceRepo; everything should work as before. Just that this time it is communicating with a database and not stale data.
   MAKE SURE TO EITHER ADD SOME DATA MANUALLY OR VIA POST REQUEST

Step 5: Custom method:

1. By default the JpaRepository provides all CRUD operations using the primary key. If we want to fetch data by a column other than primary key ex:
   select * from book where author =’’;

2. Spring boot provides with conventions to follow for custom methods to execute custom queries:
   https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

3. Just add below method in BookRepo file to fetch all books by author.
   List<Book> findByAuthor(String author);

4. Update below method in BookServiceRepo class as follows:

```
public List<Book> getBooksByAuthor(String author){
    return bookRepo.findByAuthor(author);
  }
```

5. Test using url http://localhost:8081/books?author=<authorname> and now you should get list of authors.

Step 6: OneToOne:

1. Database tables have relationships using primary and foreign key.
2. Create below 2 classes for implementing one to one relationship:

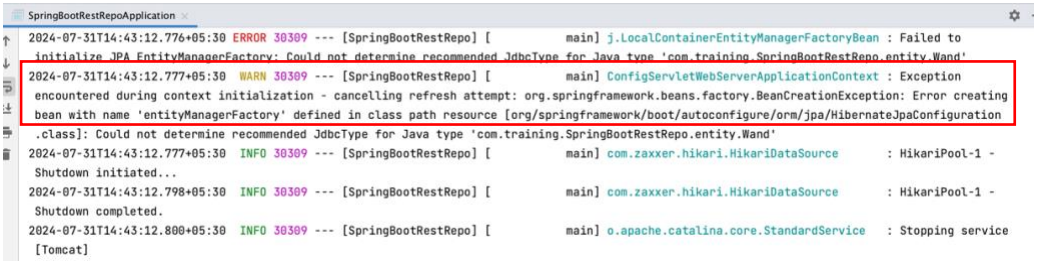
```
package com.training.SpringBootRESTRepo.entity;

import com.fasterxml.jackson.annotation.JsonIgnore;
import jakarta.persistence.*;
/@Entity
@Table(name = "wand")
public class Wand {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;
  private String wood;
  private String core;
  private String length;

  public Wand() {

  }

  public Wand(String wood, String core, String length) {
    this.wood = wood;
    this.core = core;
    this.length = length;
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getWood() {
    return wood;
  }

  public void setWood(String wood) {
    this.wood = wood;
```

```
    }

    public String getCore() {
        return core;
    }

    public void setCore(String core) {
        this.core = core;
    }

    public String getLength() {
        return length;
    }

    public void setLength(String length) {
        this.length = length;
    }

    @Override
    public String toString() {
        return "Wand{" +
                "id=" + id +
                ", wood='" + wood + '\'' +
                ", core='" + core + '\'' +
                ", length='" + length + '\'' +
                '}';
    }
}
```

```
package com.training.SpringBootRESTRepo.entity;

import jakarta.persistence.*;

@Entity
@Table(name = "fictional_character")
public class FictionalCharacter {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(length = 150, unique = true)
    private String name;
    @Column(length = 150, nullable = false)
    private String house;

    private Wand wand;

    @Column(length = 150, nullable = false)
    private String bio;
    private String imageurl;

    public FictionalCharacter() {

    }

    public FictionalCharacter(String name, String house, Wand wand, String bio, String imageurl) {
        this.name = name;
        this.house = house;
        this.wand = wand;
        this.bio = bio;
        this.imageurl = imageurl;
    }

    @Override
    public String toString() {
        return "FictionalCharacter{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", house='" + house + '\'' +
                ", wand=" + wand +
                ", bio='" + bio + '\'' +
                ", imageurl='" + imageurl + '\'' +
                '}';
    }
```

```
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getHouse() {
        return house;
    }

    public void setHouse(String house) {
        this.house = house;
    }

    public Wand getWand() {
        return wand;
    }

    public void setWand(Wand wand) {
        this.wand = wand;
    }

    public String getBio() {
        return bio;
    }

    public void setBio(String bio) {
        this.bio = bio;
    }

    public String getImageurl() {
        return imageurl;
    }

    public void setImageurl(String imageurl) {
        this.imageurl = imageurl;
    }

}
```

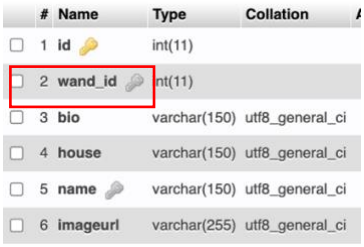
3. Rerun the project and you should get below error: Spring is unaware of how to map wand object to a column in the database table.



4. For references, spring provides OneToOne annotation. Update the Wand reference in FictionalCharacter class as follows:

```
@OneToOne
private Wand wand;
```

5. Now rerun and see the structure of fictional_character table in the database :
It creates the column name by default using convention <classname_idname>



#	Name	Type	Collation
1	id	int(11)	
2	wand_id	int(11)	
3	bio	varchar(150)	utf8_general_ci
4	house	varchar(150)	utf8_general_ci
5	name	varchar(150)	utf8_general_ci
6	imageurl	varchar(255)	utf8_general_ci

6. To give a custom name to the column use JoinColumn as follows:

```
@JoinColumn(name = "wandid")
@OneToOne
private Wand wand;
```

7. Now rerun and see the structure of fictional_character table in the database:

Table structure | Re

#	Name	Type
1	id	int(11)
2	wandid	int(11)
3	bio	varchar(150) u
4	house	varchar(150) u
5	name	varchar(150) u
6	imageurl	varchar(255) u

Step 7: Repo Layer:

1. Create Repos for character and wand as follows:

```
package com.training.SpringBootRESTRepo.repo;

import com.training.SpringBootRESTRepo.entity.Wand;
import com.training.SpringBootRESTRepo.entity.Wand;
import org.springframework.data.jpa.repository.JpaRepository;

public interface WandRepo extends JpaRepository<Wand, Integer> {
}
```

```
package com.training.SpringBootRESTRepo.repo;

import com.training.SpringBootRESTRepo.entity.FictionalCharacter;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface CharacterRepo extends JpaRepository<FictionalCharacter,Integer > {

    public List<FictionalCharacter> findAllByHouse(String house);
    public FictionalCharacter findByName(String name);
    public boolean existsByName(String name);
}
```

Step 8: Service Layer:

1. Let create CharacterService class for the business login layer.
2. **Update the class for spring specific annotations.**

```
package com.training.SpringBootRESTRepo.service;

import com.training.SpringBootRESTRepo.entity.FictionalCharacter;
import com.training.SpringBootRESTRepo.entity.Wand;
import com.training.SpringBootRESTRepo.repo.CharacterRepo;
import com.training.SpringBootRESTRepo.repo.WandRepo;
import jakarta.persistence.EntityExistsException;
```

```
import jakarta.persistence.EntityNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

public class CharacterService {


    private CharacterRepo characterRepo;


    private WandRepo wandRepo;

    public long getCount(){
        return this.characterRepo.count();
    }

    public List<FictionalCharacter> getCharacterByHouse(String house){
        return characterRepo.findAllByHouse(house);

    }
    public List<FictionalCharacter> getAllCharacters(){
        return characterRepo.findAll();
    }
    public FictionalCharacter getCharacterByName(String name){
        if(!characterRepo.existsByName(name))
            throw new EntityNotFoundException(name+" not found");
        return characterRepo.findByName(name);
    }
    public FictionalCharacter getCharacterById(int id){
        System.out.println("**********************************");
        System.out.println(characterRepo);
        System.out.println("**********************************");
        if(!characterRepo.existsById(id))
            throw new EntityNotFoundException(id+" not found");
        return characterRepo.findById(id).get();
    }

}
```

Step 9: Controller Layer:

1. You can execute below queries to populate your database with some seed data.

   PLEASE MAKE SURE TO CHANGE THE BELOW PROPERTY IN application.properties file to **update**:

   spring.jpa.hibernate.ddl-auto=update

```
INSERT INTO wand (  core, length, wood) VALUES( 'Unicorn Hair', '10 inches', 'Hawthorn');
INSERT INTO wand (  core, length, wood) VALUES( 'Phoenix feather', '13.5 inches', 'yew');
INSERT INTO wand (  core, length, wood) VALUES( 'Dragon heartstring', '10.75 inches', 'Vine');
INSERT INTO wand (  core, length, wood) VALUES( 'Phoenix Feather', '11 inches', 'Holly');
INSERT INTO wand ( core, length, wood) VALUES( 'Thestral tail hair', '15 inches', 'Elder');
INSERT INTO wand (  core, length, wood) VALUES( NULL, '16 inches(broken)', 'Oak');
INSERT INTO wand (  core, length, wood) VALUES( 'Some Core', 'Unknown', 'Unknown');
INSERT INTO wand (  core, length, wood) VALUES('Unicorn hair', '13 inches', 'Cherry');

INSERT INTO fictional_character (bio, house, name, imageurl,wandid) VALUES( 'The wise and powerful headmaster of Hogwarts, Dumbledore is known for his wisdom, kindness, and crucial role in the fight against Voldemort.', 'Gryffindor', 'Albus Dumbledore', 'https://ik.imagekit.io/hpapi/albus.jpg',5);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( 'A member of the Malfoy family, Draco is known for his rivalry with Harry and his complex journey throughout the series.', 'Slytherin', 'Draco Malfoy', 'https://ik.imagekit.io/hpapi/draco.jpg',1);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( '!! The Boy Who Lived, Harry is known for his bravery, his role in defeating Voldemort, and his strong sense of justice.', 'Gryffindor', 'Harry Potter', 'https://ik.imagekit.io/hpapi/harry.jpg',4);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( 'The brightest witch of her age, Hermione is known for her intelligence, dedication, and strong principles.', 'Gryffindor', 'Hermione Granger', 'https://ik.imagekit.io/hpapi/hermione.jpg',3);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( 'The Dark Lord, Voldemort is known for his fearsome power, his quest for immortality, and his reign of terror over the wizarding world.', 'Slytherin', 'Lord Voldemort', 'https://ik.imagekit.io/hpapi/voldemort.jpg',2);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( 'Known for her quirky personality and
```

```
unique perspective, Luna is a loyal friend and a member of Dumbledores Army.', 'Ravenclaw', 'Luna Lovegood',
'https://ik.imagekit.io/hpapi/luna.jpg',7);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( 'Initially timid, Neville grows into a
courageous and capable wizard, playing a key role in the final battle against Voldemort.', 'Gryffindor', 'Neville Longbottom',
'https://ik.imagekit.io/hpapi/neville.jpg',8);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( 'Harrys loyal best friend, Ron is known for
his humor, courage, and strategic thinking in chess and battles', 'Gryffindor', 'Ron Weasley',
'https://ik.imagekit.io/hpapi/ron.jpg',null);
INSERT INTO fictional_character ( bio, house, name, imageurl,wandid) VALUES( 'The Keeper of Keys and Grounds at
Hogwarts, Hagrid is known for his love of magical creatures and his loyalty to Dumbledore', 'Gryffindor', 'Rubeus Hagrid',
'https://ik.imagekit.io/hpapi/hagrid.png',6);
```

2. Let create CharacterController class for the REST API endpoints.

3. Update the class for spring specific annotations.

```
package com.training.SpringBootRESTRepo.rest;

import com.training.SpringBootRESTRepo.entity.FictionalCharacter;
import com.training.SpringBootRESTRepo.service.CharacterService;
import jakarta.persistence.EntityExistsException;
import jakarta.persistence.EntityNotFoundException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.List;
import java.util.Map;


@RestController
@RequestMapping("/api/characters")
@CrossOrigin(origins = "*",methods = {RequestMethod.DELETE, RequestMethod.GET, RequestMethod.PUT,
RequestMethod.POST})
public class CharacterController {

    Logger logger = LoggerFactory.getLogger(CharacterController.class);
    @Autowired
    private CharacterService characterService;

    @GetMapping
    public List<FictionalCharacter> getAllCharacters(@RequestParam(defaultValue = "all") String house){
        logger.info("GET All Characters for house {}", house);
       if(house.equals("all")) {
            List<FictionalCharacter> ob = characterService.getAllCharacters();
            System.out.println(ob.size());
            return ob;
       }else{
            List<FictionalCharacter> ob = characterService.getCharacterByHouse(house);
            System.out.println(ob.size());
            return ob;
       }
    }
    @GetMapping("/id/{id}")
    public ResponseEntity<Object> getCharacterById(@PathVariable int id){
        try {
            FictionalCharacter ob = characterService.getCharacterById(id);
            return ResponseEntity.ok(ob);
        }catch (EntityNotFoundException e){
            Map<String, String> errorMap = new HashMap<>();
            errorMap.put("error", e.getMessage());
            return ResponseEntity.badRequest().body(errorMap);
        }
    }
    @GetMapping("/name/{name}")
    public ResponseEntity<Object> getCharacterByName(@PathVariable String name){
        try {
            FictionalCharacter ob = characterService.getCharacterByName(name);
            return ResponseEntity.ok(ob);
        }catch (EntityNotFoundException e){
```

```
            Map<String, String> errorMap = new HashMap<>();
            errorMap.put("error", e.getMessage());
            return ResponseEntity.badRequest().body(errorMap);
        }
    }

}
```

4. Please test the above API using below endpoints and GET request:
   http://localhost:8081/api/characters
   http://localhost:8081/api/characters/id/1
   http://localhost:8081/api/characters/name/Hermione Granger

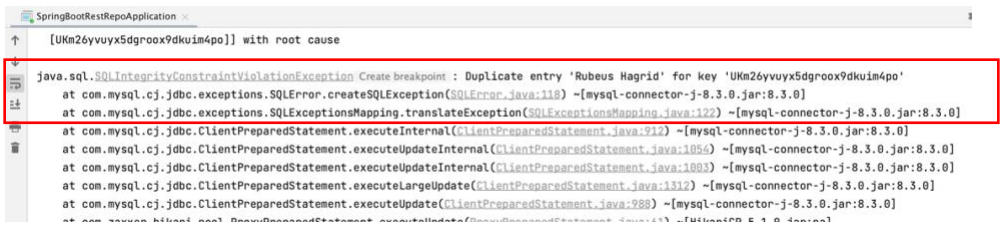
Step 10: Insert Data:

1. Add below methods in CharacterService and CharatcerController class respectively.

```
public FictionalCharacter addNewCharacter(FictionalCharacter character){
        if(characterRepo.existsById(character.getId()))
            throw new EntityExistsException("Cannot add "+character.getId()+" already exists");
        return characterRepo.save(character);
    }
}

@PostMapping()
public ResponseEntity<Object> addCharacter(@RequestBody FictionalCharacter character){
    try{
        FictionalCharacter ob = characterService.addNewCharacter(character);
        return ResponseEntity.ok(ob);
    }catch (EntityExistsException e){
        Map<String, String> errorMap = new HashMap<>();
        errorMap.put("error", e.getMessage());
        return ResponseEntity.badRequest().body(errorMap);
    }
}
```

2. Test sample data and making a POST request should give an **Internal Server Error** if an entry for name already exists:

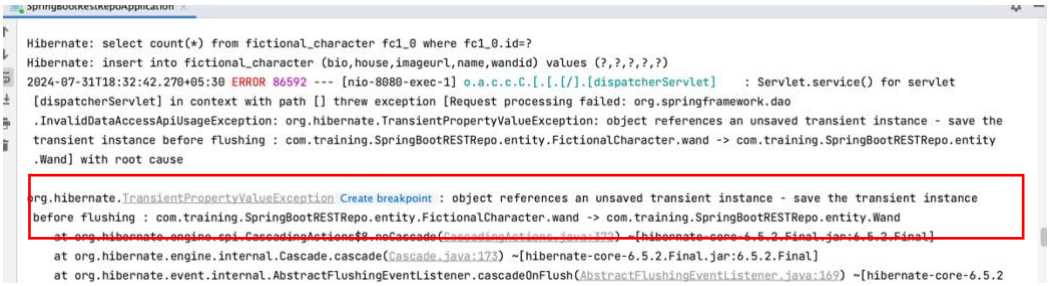
```
{
"name": "Rubeus Hagrid",
"house": "Gryffindor",
"wand": {
"wood": "oak",
"core": "",
"length": 16
},
"bio":"half-giant who is the gamekeeper and Keeper of Keys and Grounds of Hogwarts, the primary setting for the first six novels",
"imageurl": "https://ik.imagekit.io/hpapi/hagrid.png"
}
```



3. Trying below sample data also gives an "**Internal Server Error", since fictional character referencing an unsaved wand reference**: Watch the insert query has wandid as a column.

```
{
"name": "Bellatrix Lestrange",
"house": "Slytherin",
"wand": {
"wood": "walnut",
"core": "dragon heartstring",
"length": 12.75
},
"bio": "She evolved from an unnamed peripheral character in Harry Potter and the Goblet of Fire into a major antagonist in subsequent novels. ",
```

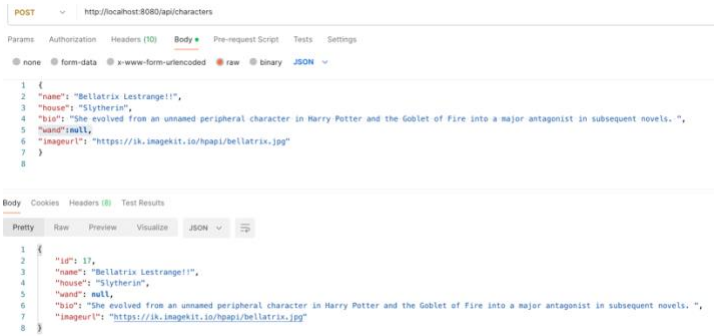
"imageurl": "https://ik.imagekit.io/hpapi/bellatrix.jpg"
}



4. If you remove the wand reference or pass it null in the JSON data, it works fine as follows:

{
"name": "Bellatrix Lestrange",
"house": "Slytherin",
"bio": "She evolved from an unnamed peripheral character in Harry Potter and the Goblet of Fire into a major antagonist in subsequent novels. ",
"imageurl": "https://ik.imagekit.io/hpapi/bellatrix.jpg"
}

{
"name": "Bellatrix Lestrange",
"house": "Slytherin",
"bio": "She evolved from an unnamed peripheral character in Harry Potter and the Goblet of Fire into a major antagonist in subsequent novels. ",
"wand":null,
"imageurl": "https://ik.imagekit.io/hpapi/bellatrix.jpg"
}



5. More sample data to try just in case:
{
"name": "Argus Filch",
"gender": "male",
"house": "",
"wand": {
"wood": "",
"core": "",
"length": null
},
"image": "https://ik.imagekit.io/hpapi/filch.jpg",
"bio":""
}

6. Now when we pass character data , we also want to save wand data. There are 2 ways that can be achieved.

   a. First saving the wand instance and then saving character instance. Update below method in CharacterService class:

```
public FictionalCharacter addNewCharacter(FictionalCharacter character){
    if(characterRepo.existsById(character.getId()))
       throw new EntityExistsException("cannot insert "+character.getId()+" already exists");
    if(character.getWand() != null) {
       wandRepo.save(character.getWand());
     }
    return characterRepo.save(character);
  }
```

Try POST request with below data, it should work:

```
{
"name": "Bellatrix Lestrange",
"house": "Slytherin",
"wand": {
"wood": "walnut",
"core": "dragon heartstring",
"length": 12.75
},
"bio": "She evolved from an unnamed peripheral character in Harry Potter and the Goblet of Fire into a major antagonist in subsequent novels. ",
"imageurl": "https://ik.imagekit.io/hpapi/bellatrix.jpg"
}
```

b. Alternatively can use cascade. Update OneToOne annotation for wand property in FictionalCharacter class.

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "wandid")
private Wand wand;
```

Comment out the if block highlighted below:

```
public FictionalCharacter addNewCharacter(FictionalCharacter character){
    if(characterRepo.existsById(character.getId()))
       throw new EntityExistsException("cannot insert "+character.getId()+" already exists");
    /*
            if(character.getWand() != null) {
                wandRepo.save(character.getWand());
             }
    */
    return characterRepo.save(character);
  }
```

Now try making POST request and you should see entry for both tables in database:

```
{
"name": "Bellatrix Lestrange",
"house": "Slytherin",
"wand": {
"wood": "walnut",
"core": "dragon heartstring",
"length": 12.75
},
"bio": "She evolved from an unnamed peripheral character in Harry Potter and the Goblet of Fire into a major antagonist in subsequent novels. ",
"imageurl": "https://ik.imagekit.io/hpapi/bellatrix.jpg"
}
```

Step 11: Update Data:

1. Add below methods in CharacterService and CharatcerController class respectively.

```
    public FictionalCharacter updateCharacter (FictionalCharacter character){
        if(!characterRepo.existsById(character.getId()))
           throw new EntityNotFoundException("cannot update "+character.getId()+" does not exists");
        /*
                if(character.getWand() != null) {
                    wandRepo.save(character.getWand());
                 }
        */
        return characterRepo.save(character);
      }
  @PutMapping
  public ResponseEntity<Object> updateCharacter(@RequestBody FictionalCharacter character){
    try {
      FictionalCharacter ob = characterService.updateCharacter(character);
      return ResponseEntity.ok(ob);
    }catch (EntityNotFoundException e){
      Map<String, String> errorMap = new HashMap<>();
      errorMap.put("error", e.getMessage());
      return ResponseEntity.badRequest().body(errorMap);
    }
}
```

2. PUT also works the same as POST.

   a. If Cascade is not used then updating fictional character will not update the wand unless you uncomment the part highlighted above.

   b. If Cascade is used then updating fictional character will also update the wand if below object is passed:

```
{
   "id": 1,
   "name": "Albus Dumbledore!!",
   "house": "Gryffindor",
   "wand": {
      "id": 5,
      "wood": "Elder!! wood",
      "core": "Thestral tail hair",
      "length": "15 inches"
   },
   "bio": "The wise and powerful headmaster of Hogwarts, Dumbledore is known for his wisdom, kindness, and crucial role in the fight against Voldemort.",
   "imageurl": "https://ik.imagekit.io/hpapi/albus.jpg"
}
```

   Alternatively if only fictional character is passed then only it will be saved.

```
{
   "id": 1,
   "name": "Albus Dumbledore!!",
   "house": "Gryffindor",
   "bio": "The wise and powerful headmaster of Hogwarts, Dumbledore is known for his wisdom, kindness, and crucial role in the fight against Voldemort.",
   "imageurl": "https://ik.imagekit.io/hpapi/albus.jpg"
}
```

Step 12: Delete Data:

1. Add below in CharacterService and CharacterController respectively:

   Below code works when cascade is not used.

```
public boolean deleteCharacter(int id){
   if(!characterRepo.existsById(id))
      throw new EntityNotFoundException("cannot delete "+id+" does not exist");
   FictionalCharacter character = characterRepo.findById(id).get();
   Wand ob = character.getWand();
   characterRepo.delete(character);
   if(ob != null)
      wandRepo.delete(ob);
   return true;
}
```

   Below code works when cascade is used.

```
public boolean deleteCharacter(int id){
   if(!characterRepo.existsById(id))
      throw new EntityNotFoundException("cannot delete "+id+" does not exist");
   FictionalCharacter character = characterRepo.findById(id).get();
   characterRepo.delete(character);
   return true;
}
```

```
@DeleteMapping("/{id}")
  public ResponseEntity<Object> deleteCharacter(@PathVariable int id){
     Map<String, String> map = new HashMap<>();
     try {
        characterService.deleteCharacter(id);
        map.put("message", "Delete successful");
        return ResponseEntity.ok(map);
     }catch (EntityNotFoundException e){
        map.put("error", e.getMessage());
        return ResponseEntity.badRequest().body(map);
     }

  }
```

2. Add below code for deleting wand object in both service and controller:

```
public boolean deleteWand(int id) {
   if(!wandRepo.existsById(id))
      throw new EntityNotFoundException("cannot delete "+id+" does not exist");
   Wand wand= wandRepo.findById(id).get();
   wandRepo.delete(wand);
   return true;
}

@DeleteMapping("/wand/{id}")
public ResponseEntity<Object> deleteWand(@PathVariable int id){
   Map<String, String> map = new HashMap<>();
   try {
      characterService.deleteWand(id);
      map.put("message", "Delete successful");
      return ResponseEntity.ok(map);
   }catch (EntityNotFoundException  e){
      map.put("error", e.getMessage());
   }catch (Exception e){
      if(e.getCause() != null && e.getCause().getCause() instanceof SQLIntegrityConstraintViolationException) {
         SQLIntegrityConstraintViolationException sql_violation_exception = (SQLIntegrityConstraintViolationException)
e.getCause().getCause() ;
         map.put("error", "SQLIntegrityConstraintViolationException has accured. Foreign key cpnstraint" );
      } else {
         System.out.println(e.getMessage());
      }
   }
   return ResponseEntity.badRequest().body(map);
}
```

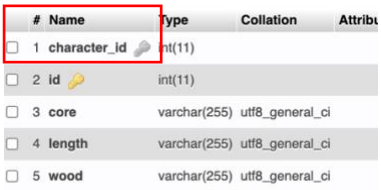
If you try to delete wand id that is referenced in fictional_character table, you will get foreign key constraint violation.

Step 13: Bidirectional Mapping

1. What if a character loses the wand? 😱 I need to be able to get character information from wand id to know it belongs to whom. So Wand class should be associated with character class in some way
2. For this reason we use bidirectional mapping. Update Wand class to have a reference of FictionalCharacter type as follows:

```
@Entity
@Table(name = "wand")
public class Wand {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;
   private String wood;
   private String core;
   private String length;

   @OneToOne
   private FictionalCharacter character;
// other part remain the same

// add getter and setter for this character reference
}
```

3. Now rerun the main method, see the table wand structrure. It adds character_id as a FK reference in wand table.



#	Name	Type	Collation	Attribu
1	character_id	int(11)		
2	id	int(11)		
3	core	varchar(255)	utf8_general_ci	
4	length	varchar(255)	utf8_general_ci	
5	wood	varchar(255)	utf8_general_ci	

4. This is not the expected behaviour. To tell Spring boot JPA not to create a FK reference, add mappedBy attribute in Wand class as follows

```
@OneToOne(mappedBy=”wand”)
  private FictionalCharacter character;
```

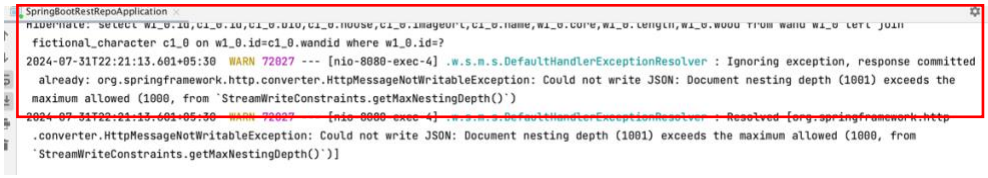
MappedBy signals hibernate that the key for the relationship is on the other side.

This means that although you link 2 tables together, only 1 of those tables has a foreign key constraint to the other one. MappedBy allows you to still link from the table not containing the constraint to the other table.

5. Now update service and controller classes respectively as follows:

```
public FictionalCharacter getCharacterByWandId(int wandid){
   if(!wandRepo.existsById(wandid))
      throw new EntityNotFoundException("cannot fetch as wand with id "+wandid+" does not exist");
   Wand wand = wandRepo.findById(wandid).get();
   return wand.getCharacter();
}

@GetMapping("/wand/{id}")
public ResponseEntity<Object> getCharacterWithWandId(@PathVariable int id){
   Map<String, Object> map = new HashMap<>();
   try {
      map.put(AppConstants.STATUS, Status.SUCCESS);
      FictionalCharacter character= characterService.getCharacterByWandId(id);
      map.put("character",character);
      return ResponseEntity.ok(map);
   }catch (EntityNotFoundException e){
      map.put("error", e.getMessage());
      return ResponseEntity.badRequest().body(map);
   }

}
```

6. Now rerun the application and make a GET request on below URL:
   http://localhost:8080/api/characters /wand/1
   It will throw below error. Reason is when wand.getCharacter() method is called, it returns an object of FictionalCharacter. But FictionalCharacter class internally refers wand and hence called getWand() function and this chain call continues getting StackPverflowError.



We use the @JsonIgnore annotation to specify a method or field that should be ignored during serialization and deserialization processes. This marker annotation belongs to the Jackson library.

We often apply this annotation to exclude fields that may not be relevant or could contain sensitive information. We use it on a field or a method to mark a property we'd like to ignore.

7. To overcome this issue mark the character reference in Wand class and vice-a-versa as follows:

```
@OneToOne(mappedBy="wand")
@JsonIgnoreProperties("wand")
 private FictionalCharacter character; // add this in Wand class

@OneToOne
@JsonIgnoreProperties("character") // add this in FictionalCharacter class
 private Wand wand;
```

8. Now rerun the application and making GET request again should fetch data appropriately.

Step 14: One To Many [Self STUDY]

1. Look at below classes in the bitbucket code:
   Customer , Invoice entities.
2. Customer and Invoice service classes
3. Customer and Invoice controller classes

Step 15: 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.