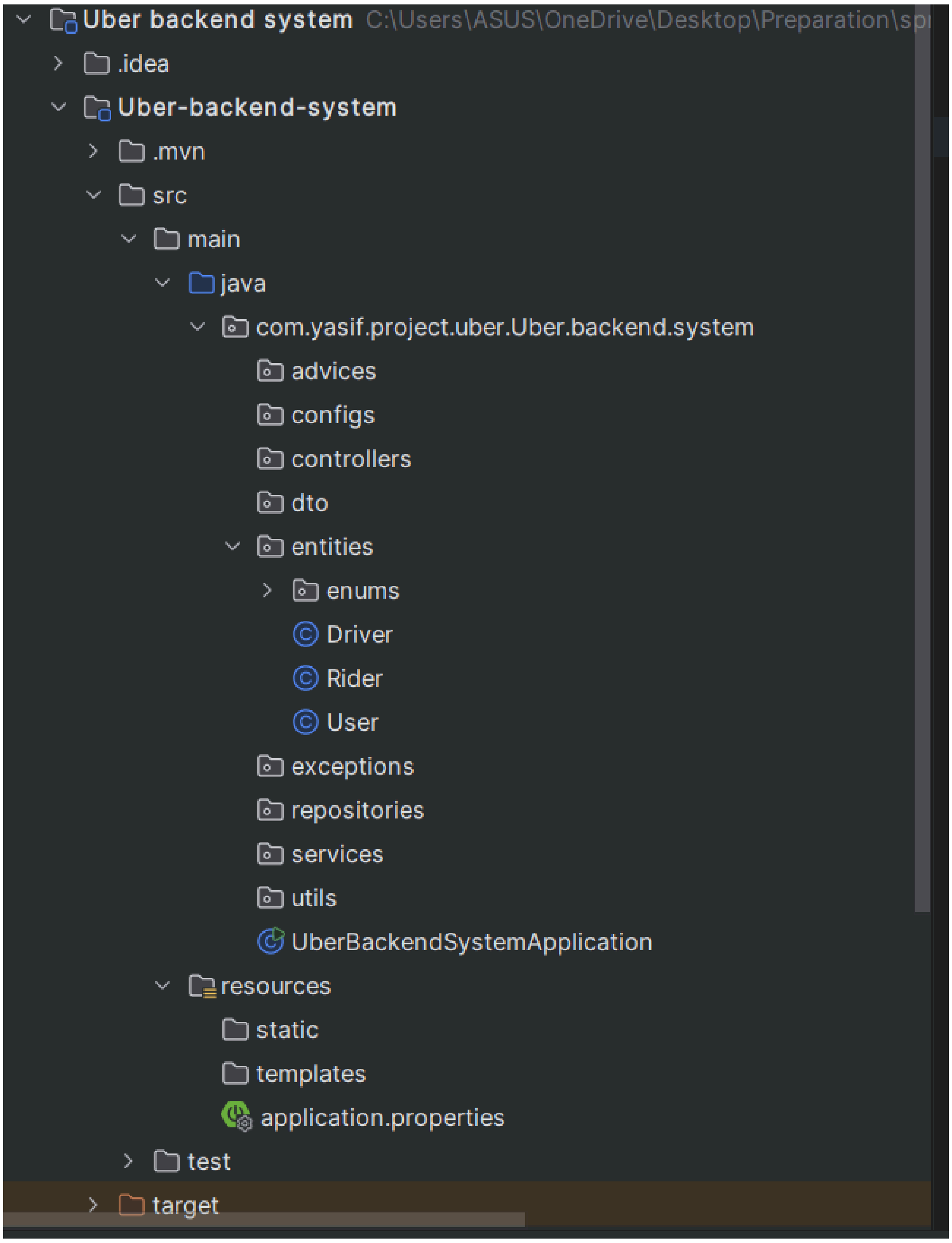
Project Uber.

Here is the initial foundation of my project along with the entities I defined today. I'm starting by solidifying the overall project structure, which currently looks like this.



```
Uber backend system  C:\Users\ASUS\OneDrive\Desktop\Preparation\sp
  .idea
  Uber-backend-system
    .mvn
    src
      main
        java
          com.yasif.project.uber.Uber.backend.system
            advices
            configs
            controllers
            dto
            entities
              enums
              Driver
              Rider
              User
            exceptions
            repositories
            services
            utils
            UberBackendSystemApplication
        resources
          static
          templates
          application.properties
      test
    target
```

Explanation of the Project Structure

The folder hierarchy reflects a clean, production-grade Spring Boot backend architecture. At a macro level, the structure showcases a clear separation of concerns, ensuring scalability, maintainability, and enterprise-class extensibility.

Here’s the breakdown of each functional cluster:

1. advices

This package is dedicated to global exception handling and cross-cutting error-management workflows. Typically holds @ControllerAdvice classes to ensure consistent API error responses.

2. configs

A centralized configuration hub used for Spring beans, security configurations, CORS policies, and other infrastructure-level settings.

3. controllers

This layer orchestrates inbound traffic. It hosts REST controllers that expose endpoints to clients and delegate business workflows to the service tier.

4. dto

A dedicated space for Data Transfer Objects—ensuring clean API contracts by decoupling internal entity representations from external responses.

5. entities

This is the core domain model of the system.

- **enums**
  Houses enumerations such as Driver, Rider, and User. These likely represent role types or user categories that define behavior and authorization boundaries across the system.
- **(Other entity classes)**
  Will represent your core business objects like User, Ride, DriverProfile, Vehicle, Rating, etc.

6. exceptions

Contains custom exception classes supporting granular error signaling. This empowers the API to surface business-specific failures cleanly.

7. repositories

This is the data access layer powered by Spring Data JPA. Interfaces here abstract DB operations and provide CRUD capabilities out of the box.

8. services

The business logic nucleus of the application. Services drive workflows, orchestrate domain operations, and ensure adherence to business rules.

9. utils

A utility layer to consolidate helper classes, reusable functions, and shared logic.

10. resources

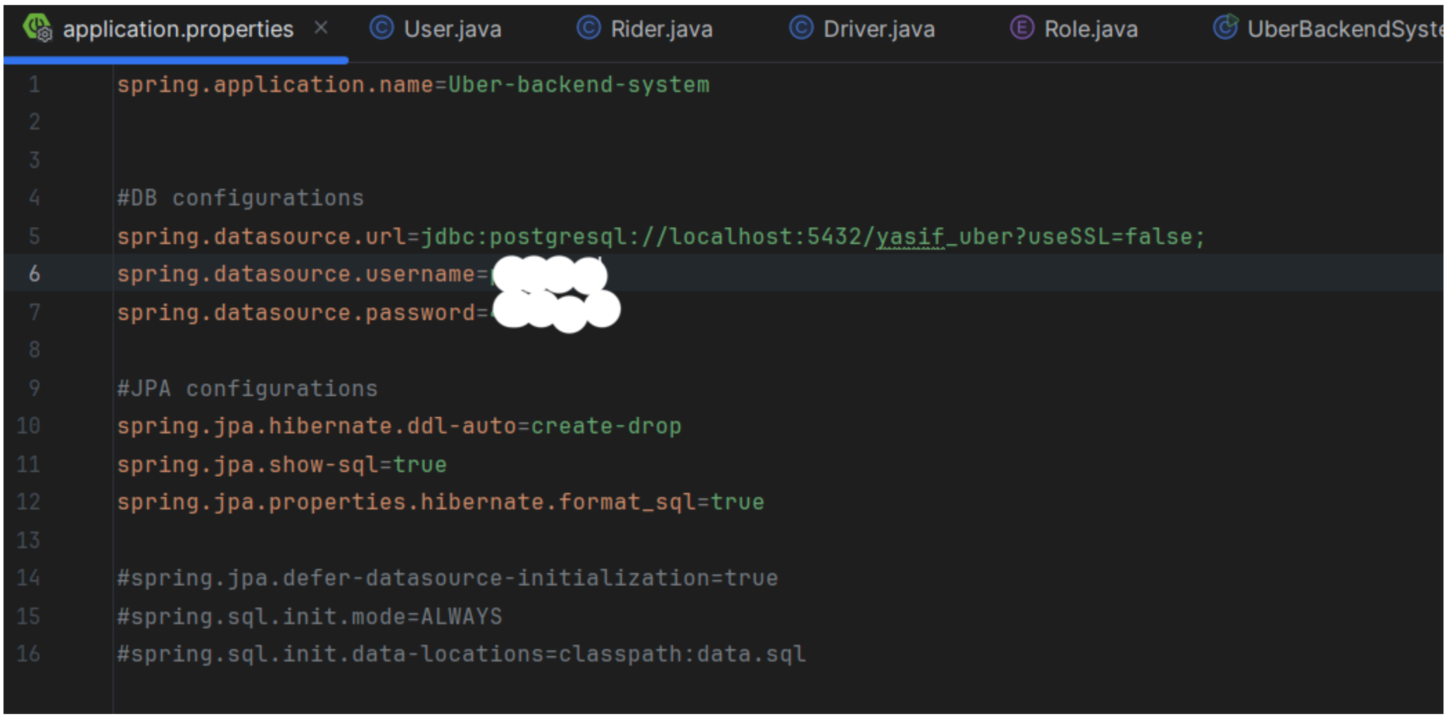
Standard Spring Boot resources directory:

- **static** – Static assets (images, JS, CSS, etc.)
- **templates** – Thymeleaf/HTML templates (if using MVC)
- **application.properties** – Centralized configuration file for the application lifecycle.

11. UberBackendSystemApplication

The main entry point annotated with @SpringBootApplication. Orchestrates auto-configuration, component scanning, and bootstrapping of the entire service.

Next, I have my application.properties file, which is used to define the database configurations and Hibernate settings.



```
spring.application.name=Uber-backend-system


#DB configurations
spring.datasource.url=jdbc:postgresql://localhost:5432/yasif_uber?useSSL=false;
spring.datasource.username=[redacted]
spring.datasource.password=[redacted]

#JPA configurations
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

#spring.jpa.defer-datasource-initialization=true
#spring.sql.init.mode=ALWAYS
#spring.sql.init.data-locations=classpath:data.sql
```

Executive-Level Explanation of application.properties

Your application.properties file is essentially the **central configuration hub** for your Spring Boot service. It operationalizes key database touchpoints and governs ORM behavior. Let's break it down with a structured, industry-standard interpretation:

🔧 1. Application Metadata

```
spring.application.name=Uber-backend-system
```

This parameter establishes a **service identifier** within the broader application landscape. It's leveraged by logging frameworks, monitoring dashboards, and distributed systems tooling.

🗄️ 2. Database Configuration Layer (PostgreSQL)

spring.datasource.url=jdbc:postgresql://localhost:5432/yasif_uber?useSSL=false

spring.datasource.username=

spring.datasource.password=

What this represents:

- **spring.datasource.url** defines the network endpoint for your PostgreSQL instance.
- **username and password** set up authentication for the DataSource object Spring Boot provisions under the hood.
- This configuration ensures the application maintains a persistent, secure, and optimized connection to your yasif_uber database.

🧩 3. JPA + Hibernate Operational Controls

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Breakdown:

✓ **ddl-auto=create-drop**
This instructs Hibernate to **create all tables on startup** and **drop them on shutdown**. Ideal for rapid prototyping or Day-0 development, but not recommended in production.

✓ **show-sql=true**
Enables SQL visibility in the console — highly valuable for early-cycle debugging and query validation.

✓ **format_sql=true**
Beautifies SQL output, making it easier to inspect relational behavior.

📝 4. Optional Initialization Controls (Commented Out)

```
#spring.jpa.defer-datasource-initialization=true
#spring.sql.init.mode=ALWAYS
#spring.sql.init.data-locations=classpath:data.sql
```

If activated, these properties automate schema bootstrapping and data seeding from a data.sql file — an enterprise-grade pattern used for consistent environment replication (e.g., dev, UAT, CI pipelines).

🎯 Overall Summary

Your configuration establishes a **clean, modular, and scalable** baseline for your Uber backend system. You've set up:

- A clearly named microservice
- A fully operational PostgreSQL integration
- Hibernate tuned for development visibility
- Ready-to-enable SQL initialization tooling

This structure aligns well with industry best practices for early-stage backend architecture.

1. spring.jpa.defer-datasource-initialization=true

This property ensures that **Spring Boot delays running data initialization scripts until after Hibernate has completed schema generation**.

Why this matters

In your Uber backend system, you are using:

- spring.jpa.hibernate.ddl-auto=create-drop
- PostgreSQL
- Entity-based schema generation

When Hibernate starts, it **drops and re-creates your tables**.
If your SQL initialization scripts (like data.sql) run *before* Hibernate creates the schema, they will fail because the tables don't exist yet.

By enabling this flag, Spring Boot orchestrates the startup pipeline like this:

1. Hibernate creates the database schema.
2. **After that**, Spring runs schema.sql, data.sql, or any SQL initialization script.

Net business value

It avoids startup failures and ensures that your seed data (test riders, test drivers, admin users, etc.) loads consistently.

2. spring.sql.init.mode=ALWAYS

This property controls **when SQL initialization scripts should run**.

Values

Mode	Meaning
NEVER	Never run SQL scripts.
EMBEDDED	Run only for in-memory DBs (H2, HSQL, Derby).
ALWAYS	Run for every environment (PostgreSQL, MySQL, production, dev, etc.).

What ALWAYS does

If you set:

spring.sql.init.mode=ALWAYS

Then Spring Boot will **always execute** your SQL scripts—schema.sql and data.sql—regardless of which database you're connected to.

Why it's valuable for your project

TheUber backend system may require initial:

- Roles (ADMIN/DRIVER/RIDER)
- Default admin user
- Test riders or drivers
- Basic lookup data

Setting this to ALWAYS ensures your App initializes the data every time you run the project **without manually inserting records**.

3. spring.sql.init.data-locations=classpath:data.sql

This tells Spring Boot **where to find your SQL initialization file**.

How it works

If your file is placed under:

src/main/resources/data.sql

then its classpath location is simply:

classpath:data.sql

Spring Boot will load that file and execute all SQL statements inside it at startup.

Typical content in data.sql for your project

For example, your Uber app might load:

INSERT INTO role(name) VALUES ('ADMIN');

INSERT INTO role(name) VALUES ('DRIVER');

INSERT INTO role(name) VALUES ('RIDER');

INSERT INTO users (name, email, role_id) VALUES ('Admin User', 'admin@gmail.com', 1);

This ensures your application has **foundational data available instantly**.

How all three work together

When used together:

spring.jpa.defer-datasource-initialization=true

spring.sql.init.mode=ALWAYS

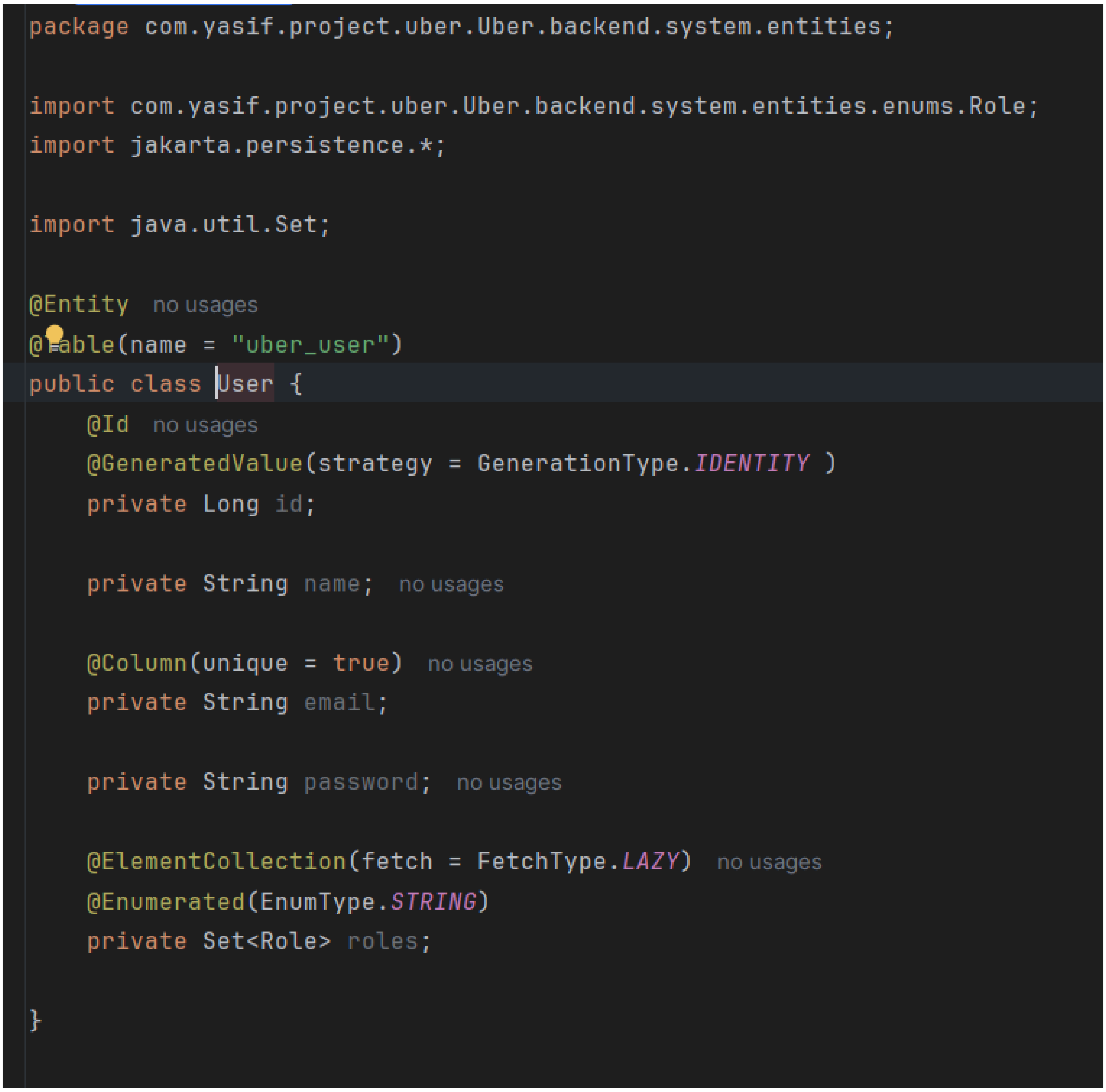
spring.sql.init.data-locations=classpath:data.sql

The execution pipeline becomes:

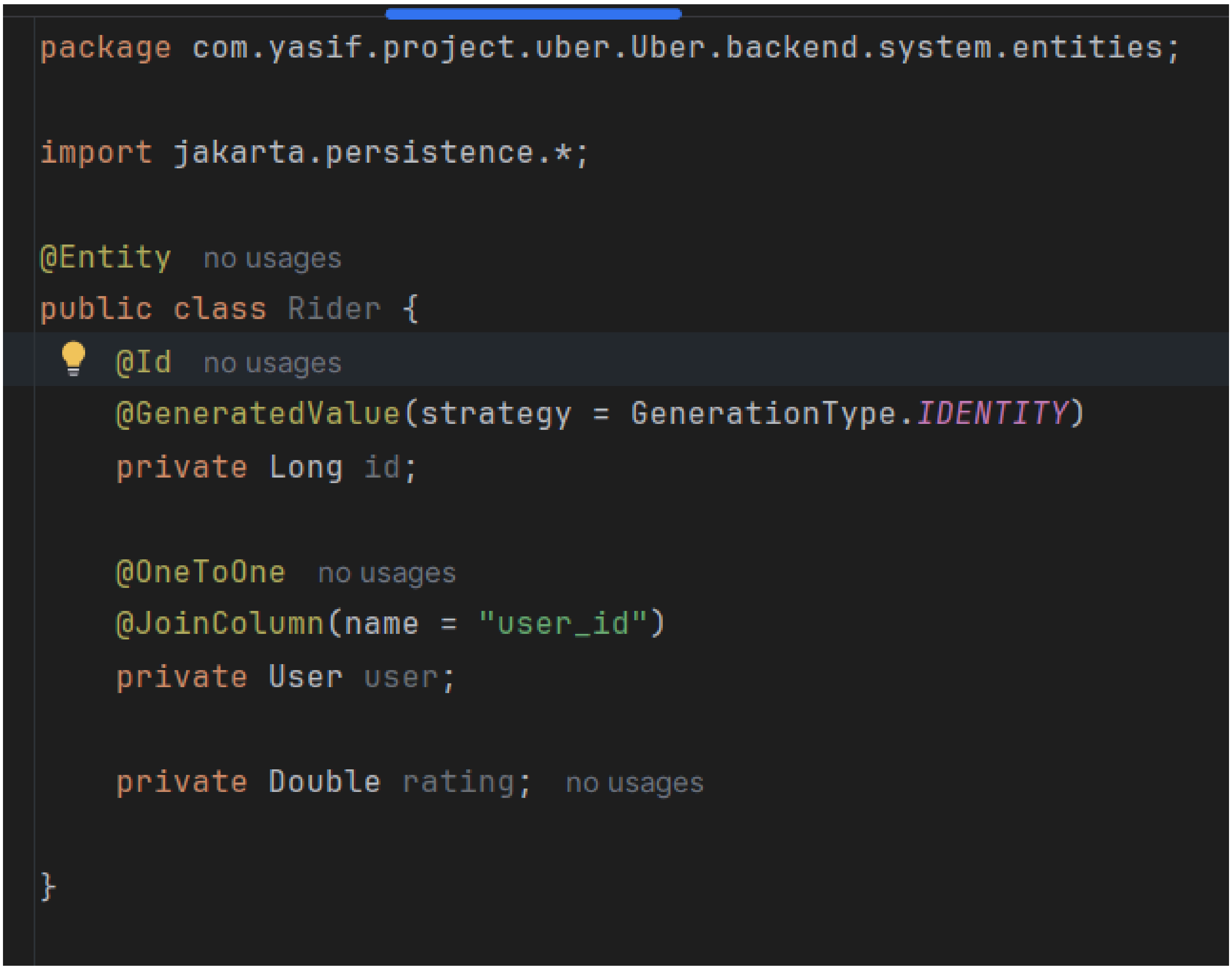
1. Hibernate creates all tables from your entities.
2. Spring reads your SQL file from /resources/data.sql.
3. SQL inserts run successfully because the tables are already created.

This is the ideal setup for projects where schemas evolve frequently during development cycles.

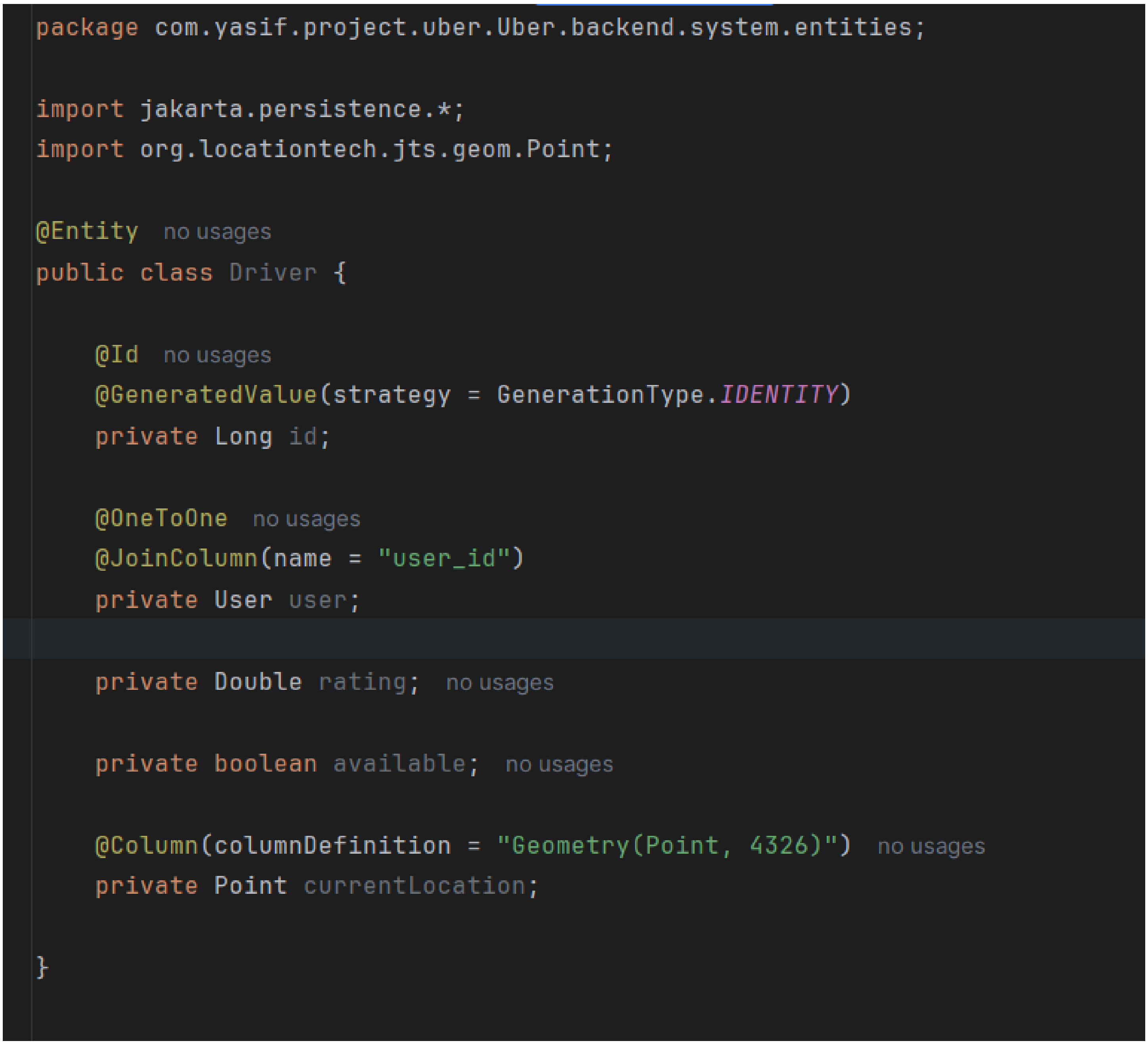
Today I created three entities: Rider, Driver, and User. See the entity definitions below.



```
package com.yasif.project.uber.Uber.backend.system.entities;

import com.yasif.project.uber.Uber.backend.system.entities.enums.Role;
import jakarta.persistence.*;

import java.util.Set;

@Entity  no usages
@Table(name = "uber_user")
public class User {
    @Id  no usages
    @GeneratedValue(strategy = GenerationType.IDENTITY )
    private Long id;

    private String name;  no usages

    @Column(unique = true)  no usages
    private String email;

    private String password;  no usages

    @ElementCollection(fetch = FetchType.LAZY)  no usages
    @Enumerated(EnumType.STRING)
    private Set<Role> roles;

}
```



```java
package com.yasif.project.uber.Uber.backend.system.entities;

import jakarta.persistence.*;

@Entity  no usages
public class Rider {
    @Id  no usages
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne  no usages
    @JoinColumn(name = "user_id")
    private User user;

    private Double rating;  no usages

}
```



```java
package com.yasif.project.uber.Uber.backend.system.entities;

import jakarta.persistence.*;
import org.locationtech.jts.geom.Point;

@Entity  no usages
public class Driver {

    @Id  no usages
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne  no usages
    @JoinColumn(name = "user_id")
    private User user;

    private Double rating;  no usages

    private boolean available;  no usages

    @Column(columnDefinition = "Geometry(Point, 4326)")  no usages
    private Point currentLocation;

}
```

🔍 User Entity — The Foundational Identity Layer

The User class functions as the **primary identity object** within the system. It anchors core authentication and authorization workflows and acts as the parent entity for role-specific profiles such as **Rider** and **Driver**.

Key Components

@Entity & @Table(name = "uber_user")
Establishes a dedicated table to ensure clean separation of user attributes.

1. **Primary Key (id)**
   a. Utilizes IDENTITY strategy for auto-increment behavior.
2. **Unique Email Constraint**

```java
@Column(unique = true)
private String email;
```

Ensures a globally unique identifier for login workflows.

3. **Role Management**

```java
@ElementCollection(fetch = FetchType.LAZY)
@Enumerated(EnumType.STRING)
private Set<Role> roles;
```

Enables multi-role provisioning. A single user can be both a rider and a driver if the business workflow requires.

Business Positioning

This entity lays the groundwork for future-ready capabilities like **RBAC**, **MFA**, and **audit trails**, which are downstream enablers for enterprise-grade security.

🚗 Rider Entity — Customer Profile Layer

The Rider entity extends the User footprint by capturing attributes associated with a customer booking a ride.

Key Components

One-to-One Relationship with User

```java
@OneToOne
@JoinColumn(name = "user_id")
private User user;
```

Each Rider is powered by exactly one User record. This enforces a consolidated identity structure.

Rating Field

```
private Double rating;
```

Captures the rider's marketplace reputation and can inform future recommendation models or fraud-prevention algorithms.

Business Positioning

This profile represents the demand side of the ecosystem. Embedding it as a dedicated entity ensures clean separation when scaling transactions, trip history, or loyalty features.

🚕 Driver Entity — Supply-Side Operational Profile

The Driver class operationalizes the supply-side persona in your system.

Key Components

One-to-One Relationship with User
Similar to the Rider entity, the Driver has a strict 1:1 linkage to a User identity.

Availability Flag

```
private boolean available;
```

Enables operational logic for ride allocation engines.

```
@Column(columnDefinition = "Geometry(Point, 4326)")
private Point currentLocation;
```

Uses a **PostGIS geometry type** to store precise latitude/longitude coordinates.
This is essential for real-time driver discovery, surge computations, and routing algorithms.

Business Positioning

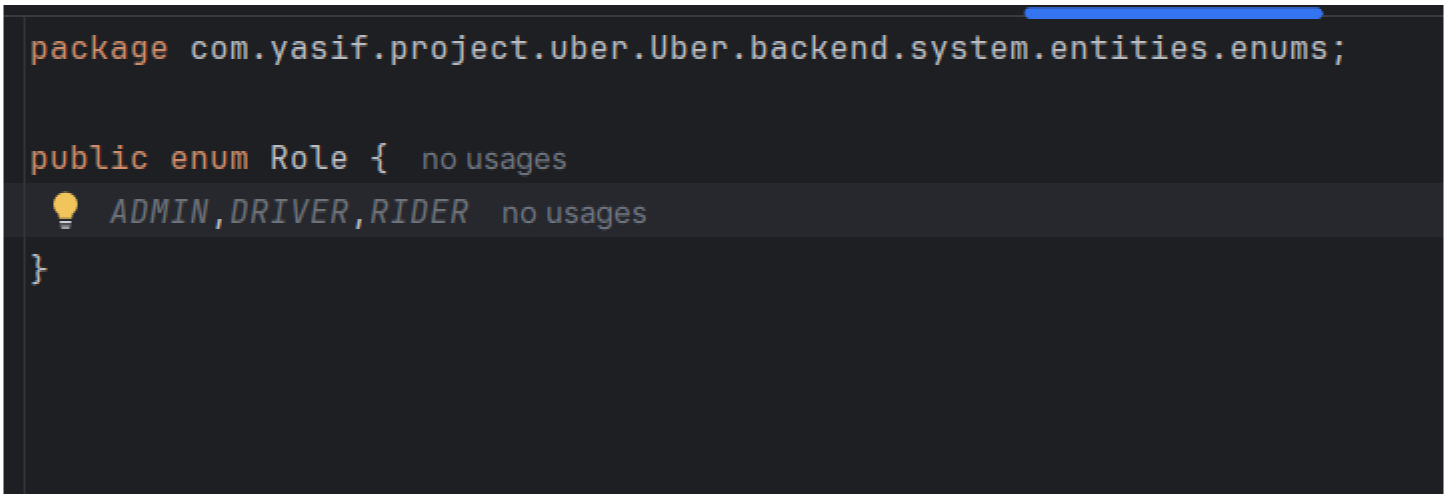
This class is foundational to enabling dynamic dispatch, proximity-based driver matching, and future integrations like route optimization engines.

🔧 Summary — Architectural Perspective

Entity	Role	Relationship	Key Attributes
User	Base identity	Parent	Name, email, password, roles
Rider	Customer	1:1 User	Rating
Driver	Service provider	1:1 User	Rating, availability, geo-location

The domain model is structured in a scalable and enterprise-aligned fashion. You've created a clean separation of concerns, enabling the system to evolve into a robust ride-hailing platform as your project matures.

For now, I have created an enum called Role with the values ADMIN, DRIVER, and RIDER.



```
package com.yasif.project.uber.Uber.backend.system.entities.enums;

public enum Role {  no usages
    ADMIN,DRIVER,RIDER  no usages
}
```

Sure! Here's a clear explanation of your enum:

```
public enum Role {
    ADMIN, DRIVER, RIDER
}
```

What it is:

- **enum** in Java is a special type that represents a fixed set of constants.
- Role is the name of the enum. It defines a type that can only have one of the specified values: ADMIN, DRIVER, or RIDER.

How it works:

- Each value (ADMIN, DRIVER, RIDER) is a constant of type Role.
- You can use it in your code to represent user roles clearly, instead of using plain strings or integers. This helps avoid mistakes like typos.

Example usage:

```java
Role userRole = Role.ADMIN;

if (userRole == Role.DRIVER) {
    System.out.println("This user is a driver.");
} else if (userRole == Role.RIDER) {
    System.out.println("This user is a rider.");
}
```

Benefits:

1. **Type safety** – The compiler ensures you only use the predefined constants.
2. **Readability** – Makes your code easier to understand.
3. **Maintainability** – If you want to add a new role, you just add it to the enum.

So basically, this enum allows you to handle user roles in a clean and structured way in your Uber backend system.

Some new thing i learned

@Embedded and @Embeddable annotations

The @Embedded annotation in **JPA** is used to include the fields of one class (called an **embeddable class**) into another entity **without creating a separate table**.

Key Points:

- It is used **inside an entity**.
- The class being embedded should be annotated with @Embeddable.
- The fields of the embeddable class are **mapped as columns** in the table of the owning entity.

Example:

```java
@Embeddable
public class Address {
    private String street;
    private String city;
    private String zipCode;
}

@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Embedded
    private Address address; // fields of Address will be columns in User table
}
```

What happens:

- The User table will have columns: id, name, street, city, zipCode.
- No separate table is created for Address.

Use case: When you want to reuse a group of fields (like Address, ContactInfo) across multiple entities without creating a separate table for each.

@Embeddable — What it means

@Embeddable is used on a class to declare that this class is **not an entity by itself**, but its fields can be **embedded into another entity's table**.

- It does **not** create a separate table.
- It acts like a reusable value object.
- Typical use cases: **Address**, **Location**, **Audit fields**, **Coordinates**, etc.

Example:

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String zipCode;
}
```

This class cannot stand alone in the database. It must be used inside an entity.

@Embedded — What it means

@Embedded is used inside an entity to indicate that the embeddable class’s fields should be **inserted as columns** in the same table.

Example:

```java
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Embedded
    private Address address;
}
```

Result:
 The user table will contain:

- id
- name
- street
- city
- zip_code

All coming from the embedded Address.

In simple terms

- **@Embeddable** → Defines a reusable component class.
- **@Embedded** → Inserts that component's fields into the entity's table.

@GeneratedValue(strategy = GenerationType.IDENTITY) — What it does

This annotation tells JPA **how the primary key should be generated** when a new record is inserted into the database.

It's placed on your @Id field.

Identity Strategy (GenerationType.IDENTITY)

This strategy delegates ID generation to the **database itself**.
Most commonly used with **MySQL**, **PostgreSQL**, and other SQL databases that support **auto-increment** columns.

When you insert a new row:

- You **don't** supply the ID.
- The database automatically increments and assigns the next value.
- JPA fetches that generated value back into your entity.

Typical Example

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

Operational Behavior

When you save a new entity:

```
User user = new User();
user.setName("Yasif");
userRepository.save(user);
```

The SQL behind the scenes looks like:

```
INSERT INTO user (name) VALUES ('Yasif');
```

Then the database returns the generated ID, for example:

```
id = 101
```

JPA automatically updates the entity instance with that value.

Why enterprises commonly use IDENTITY

- **Database-driven auto-increment** simplifies the lifecycle.
- **Predictable and robust** for high-write workloads.
- **No need for sequence tables or extra queries**.

Pros

- Straightforward and compatible with most relational systems.
- Zero additional configuration.
- Ensures a stable and incremental primary key.

Cons

- JPA cannot batch insert efficiently because IDs are generated per row.
- If you need optimized batch processing, SEQUENCE strategy is better.

Quick Comparison for Context

Strategy	Generates ID	Best for
IDENTITY	DB auto-increment	MySQL, quick setups
SEQUENCE	Database sequence	PostgreSQL, Oracle, batching
AUTO	JPA picks strategy	Simple projects
TABLE	Uses separate table	Rarely used today

@GeneratedValue — All Attributes Explained

```
@GeneratedValue(
    strategy = GenerationType.IDENTITY,
    generator = "customGeneratorName"
)
```

The annotation has **two attributes**:

1. strategy

Defines **how the primary key will be generated**.

Available Strategies:

a) GenerationType.IDENTITY

- Database auto-increment.
- DB assigns the ID.
- Common for MySQL/PostgreSQL.

b) GenerationType.SEQUENCE

Uses a database sequence object.

Requires defining a sequence:

```
@SequenceGenerator(name = "seq_user", sequenceName = "user_seq")
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_user")
```

High-performance for batch inserts.

c) GenerationType.TABLE

- Uses a separate table to store counters.
- Very slow, rarely used today.

d) GenerationType.AUTO

- Hibernate picks the strategy based on the database.
- Good for simple or portable apps.

2. generator

This links @GeneratedValue to a custom generator defined using:

- @SequenceGenerator
- @TableGenerator
- or custom Hibernate generators

Example with sequence:

```
@SequenceGenerator(
    name = "user_seq_gen",
    sequenceName = "user_sequence",
    allocationSize = 1
)
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user_seq_gen")
```

Example with table generator:

```
@TableGenerator(
    name = "user_table_gen",
    table = "id_generator",
    pkColumnName = "gen_name",
    valueColumnName = "gen_value",
    pkColumnValue = "user_id",
    allocationSize = 1
)
@GeneratedValue(strategy = GenerationType.TABLE, generator = "user_table_gen")
```

Supporting Generator Annotations

These are not attributes of @GeneratedValue, but they work alongside it:

@SequenceGenerator Attributes

Used with SEQUENCE strategy.

Attribute	Meaning
name	Identifier for the generator
sequenceName	DB sequence name
allocationSize	How many IDs fetched at a time
initialValue	Starting sequence value
catalog, schema	Optional qualifiers

@TableGenerator Attributes

Used with TABLE strategy.

Attribute	Meaning
name	Generator name
table	Table storing ID values
pkColumnName	Column for generator key
valueColumnName	Column for current value
pkColumnValue	Key identifier
allocationSize	Increment size
initialValue	Initial counter

Summary View

Annotation	Purpose
@GeneratedValue	Specifies strategy & links generator
@SequenceGenerator	Defines sequence configuration
@TableGenerator	Defines table-based ID generation

Absolutely — here's a structured, enterprise-grade explanation that breaks down **each annotation**, their **runtime behavior**, and **why they're used together**, all in a clean and digestible format.

1. @ElementCollection — What It Represents

Definition:

@ElementCollection is used when you want to store a **collection of simple values** (Strings, Integers, Enums, Embeddables) **without creating a separate Entity class**.

It creates a **secondary table** automatically to store these values.

Use Cases:

- List of roles
- List of skills
- List of tags
- List of addresses (if using embeddables)

Example Structure:

```
@ElementCollection
private List<String> tags;
```

Expected DB outcome: JPA creates a new table:

```
user_tags (user_id, tags)
```

2. fetch = FetchType.LAZY — Why It Matters

This defines **when the collection should be loaded** from the database.

FetchType.LAZY

- The collection is **NOT loaded** when the entity loads.
- It gets loaded **only when accessed**:

```
user.getRoles().size();
```

This optimizes performance and reduces unnecessary SQL overhead.

Why Lazy Fetch Is Industry-Standard:

- Collection tables often grow large; loading them eagerly is expensive.
- LAZY improves scalability for large datasets.

3. @Enumerated(EnumType.STRING) — What It Does

This annotation tells JPA **how to store an enum value**.

EnumType.STRING

- Stores the **name of the enum** (e.g., "ADMIN", "DRIVER", "RIDER").
- Human-readable.
- Safe if enum order changes.

Example:

```
@Enumerated(EnumType.STRING)
private Role role;
```

Database column will store:

ADMIN

Not 0 or 1.

4. Putting Them Together

When you apply both:

```
@ElementCollection(fetch = FetchType.LAZY)
@Enumerated(EnumType.STRING)
private Set<Role> roles;
```

You are telling JPA:

Meaning:

- roles is a **collection**, not a separate entity table.
- JPA should create an **auto-managed join table** to store user roles.
- Each role is stored as a **string enum value**.
- The collection loads only **when needed** (LAZY).

Resulting Table (Auto-Generated):

```
user_roles
----------------------
user_id | role
1       | ADMIN
1       | RIDER
2       | DRIVER
```

5. Why This Pattern Is Enterprise-Friendly

This approach is widely used in production systems for role management because:

- No need to maintain a separate Role entity.
- Clean mapping for enum-based access control.
- LAZY fetch improves performance at scale.
- Enum values stored as strings prevent data corruption if enums reorder.

6. When to Use @CollectionTable

You can customize the table name:

```
@ElementCollection(fetch = FetchType.LAZY)
@CollectionTable(
    name = "user_roles",
    joinColumns = @JoinColumn(name = "user_id")
)
@Enumerated(EnumType.STRING)
private Set<Role> roles;
```

This gives full control over table naming conventions.

7. Quick Summary for Your Uber Backend System

Annotation	Responsibility
@ElementCollection	Creates a collection table for simple values
fetch = FetchType.LAZY	Loads values only when accessed
@Enumerated(EnumType.STRING)	Stores enum values as readable strings

This is a perfect pattern for **user roles**, **driver preferences**, **ride categories**, etc.

1. FetchType (inside @ElementCollection)

FetchType.LAZY

- The collection is **not loaded** when the parent entity loads.
- It loads **only when you access it**.
- Reduces unnecessary database hits.
- Best practice for collections because they can grow large.

FetchType.EAGER

- The collection loads **immediately** with the parent entity.
- Leads to extra joins and bigger queries.
- Not recommended for large-scale systems.

2. @ElementCollection

- Used when the field is a **collection of simple types** (String, Integer, Enum, Embeddable).
- Automatically creates a **separate table** to store those values.
- The new table links back to the main entity with a foreign key.
- No separate entity class is needed.

Example:

```
@ElementCollection
private Set<Role> roles;
```

3. EnumType (inside @Enumerated)

EnumType.STRING

- Stores the **name** of the enum value in the database.
- Example stored value: "ADMIN"
- human-readable and safe even if enum order changes.

EnumType.ORDINAL

- Stores the **index/position** of the enum.
- Example stored value: 0, 1, 2
- Not safe — changing enum order breaks data.

@OneToOne — What This Mapping Means

This annotation signals that **one record in this entity** is linked to **exactly one record** in another entity.

In your case:

```
@OneToOne
@JoinColumn(name = "user_id")
private User user;
```

This means:

- Each row in the current table is tied to **one and only one** User.
- The relationship is stored using a **foreign key column** called user_id.

It’s a point-to-point association — typically used for profile info, documents, ratings, etc., where the dependent data belongs to only one parent user.

How Many Types of Mappings Exist?

JPA provides four core relationship mappings:

1. **@OneToOne** — one entity linked to one entity
2. **@OneToMany** — one entity linked to many
3. **@ManyToOne** — many entities linked to one
4. **@ManyToMany** — many linked to many

These cover all relational modeling patterns.

@JoinColumn — All Attributes Explained

@JoinColumn controls how the foreign key column is created and managed.

Here are the key attributes you can use:

Attribute	Purpose
name	Name of the foreign key column (user_id)
referencedColumnName	Specifies the primary key column in the target table (default: id)
nullable	Whether the FK column can be NULL
unique	Ensures a unique FK (often used in 1-1 relationships)
insertable	Whether JPA can insert data into this column
updatable	Whether JPA can update this column
columnDefinition	Custom SQL definition for the column
table	If the column resides in a different table (rarely needed)

Real-World Interpretation

- @OneToOne is the business contract.
- @JoinColumn defines the physical foreign key infrastructure.

Together, they establish a tightly coupled, single-responsibility relationship that is common in identity, profile, and metadata models within backend service ecosystems.

Smart choice — Hibernate Spatial brings GIS support into your ORM layer. Below is a compact, production-ready explainer plus concrete Maven snippets and practical gotchas.

What **Hibernate Spatial** is

Hibernate Spatial is an extension of Hibernate ORM that adds first-class support for spatial / GIS types (Point, LineString, Polygon, etc.), spatial SQL functions and dialect integration (e.g., PostGIS). It lets you map org.locationtech.jts geometry types on entity fields and perform spatial queries through JPA/HQL.

Which Maven artifacts you typically need

1. **Hibernate Spatial** (match the Hibernate ORM major version you're using)

```
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-spatial</artifactId>
  <version>6.6.37.Final</version> <!-- or the hibernate-orm version you use -->
</dependency>
```

(artifact coordinates vary by release series; the module lives under org.hibernate.orm for modern versions).

1. **JTS (LocationTech)** — geometry model & utilities (used by Hibernate Spatial):

```
<dependency>
  <groupId>org.locationtech.jts</groupId>
  <artifactId>jts-core</artifactId>
  <version>1.18.2</version>
</dependency>
```

JTS is the canonical Java geometry library.

1. **PostGIS JDBC (if using PostgreSQL/PostGIS)** — driver that understands geometric columns:

```
<dependency>
  <groupId>net.postgis</groupId>
  <artifactId>postgis-jdbc</artifactId>
  <version>2.5.1</version>
</dependency>
```

Use the PostGIS JDBC driver appropriate to your PostGIS/Postgres version.

1. **Optional:** Jackson JTS module (for JSON serialization of geometry), or other helpers:

```xml
<!-- jackson-datatype-jts or similar -->
<dependency>
  <groupId>org.n52.jackson</groupId>
  <artifactId>jackson-datatype-jts</artifactId>
  <version>1.2.9</version>
</dependency>
```

(Useful for REST endpoints returning geometry values.)

Quick example — snippet for pom.xml

<!-- Hibernate ORM + Spatial (choose version that matches your hibernate-core) -->

```xml
<!-- Hibernate ORM + Spatial (choose version that matches your hibernate-core) -->
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-spatial</artifactId>
  <version>6.6.37.Final</version>
</dependency>

<!-- JTS geometry library -->
<dependency>
  <groupId>org.locationtech.jts</groupId>
  <artifactId>jts-core</artifactId>
  <version>1.18.2</version>
</dependency>

<!-- PostGIS JDBC driver (if using PostGIS) -->
<dependency>
  <groupId>net.postgis</groupId>
  <artifactId>postgis-jdbc</artifactId>
  <version>2.5.1</version>
</dependency>
```

Minimal configuration notes

Dialect / platform: with PostGIS you typically set a spatial-aware dialect (older guidance: org.hibernate.spatial.dialect.postgis.PostgisDialect); newer Hibernate versions integrate

spatial types more transparently — still ensure hibernate-spatial is on the classpath.
Example Spring Boot property:

```
spring.jpa.properties.hibernate.dialect=org.hibernate.spatial.dialect.postgis.PostgisDialect
```

(Check your Hibernate version — dialect classes/names evolved between major releases.)

Entity mapping: map JTS types directly:

```
import org.locationtech.jts.geom.Point;

@Column(columnDefinition = "geometry(Point,4326)")
private Point location;
```

Hibernate Spatial will handle SQL generation and type conversions.

Coordinate reference system (SRID): include SRID (e.g., 4326) in column definition for correct storage/queries.

Version & compatibility guidance (practical risks)

- **Version alignment is critical.** Hibernate Spatial must be compatible with your Hibernate ORM version — pick the hibernate-spatial release that matches your hibernate-core. Mismatches cause ClassNotFound / MappingException.
- **DB driver vs DB server:** ensure the PostGIS JDBC version matches the PostGIS/Postgres server you run.
- **Testing:** validate DDL generation (geometry column types) and run a few spatial queries (ST_Distance, ST_Within, etc.) during integration tests.

Common troubleshooting pointers

- type "geometry" does not exist → database lacks PostGIS extension or the column definition is wrong (enable CREATE EXTENSION postgis;).
- ClassNotFoundException for spatial classes → missing hibernate-spatial or mismatched versions.

Install and setup postgres and postgis

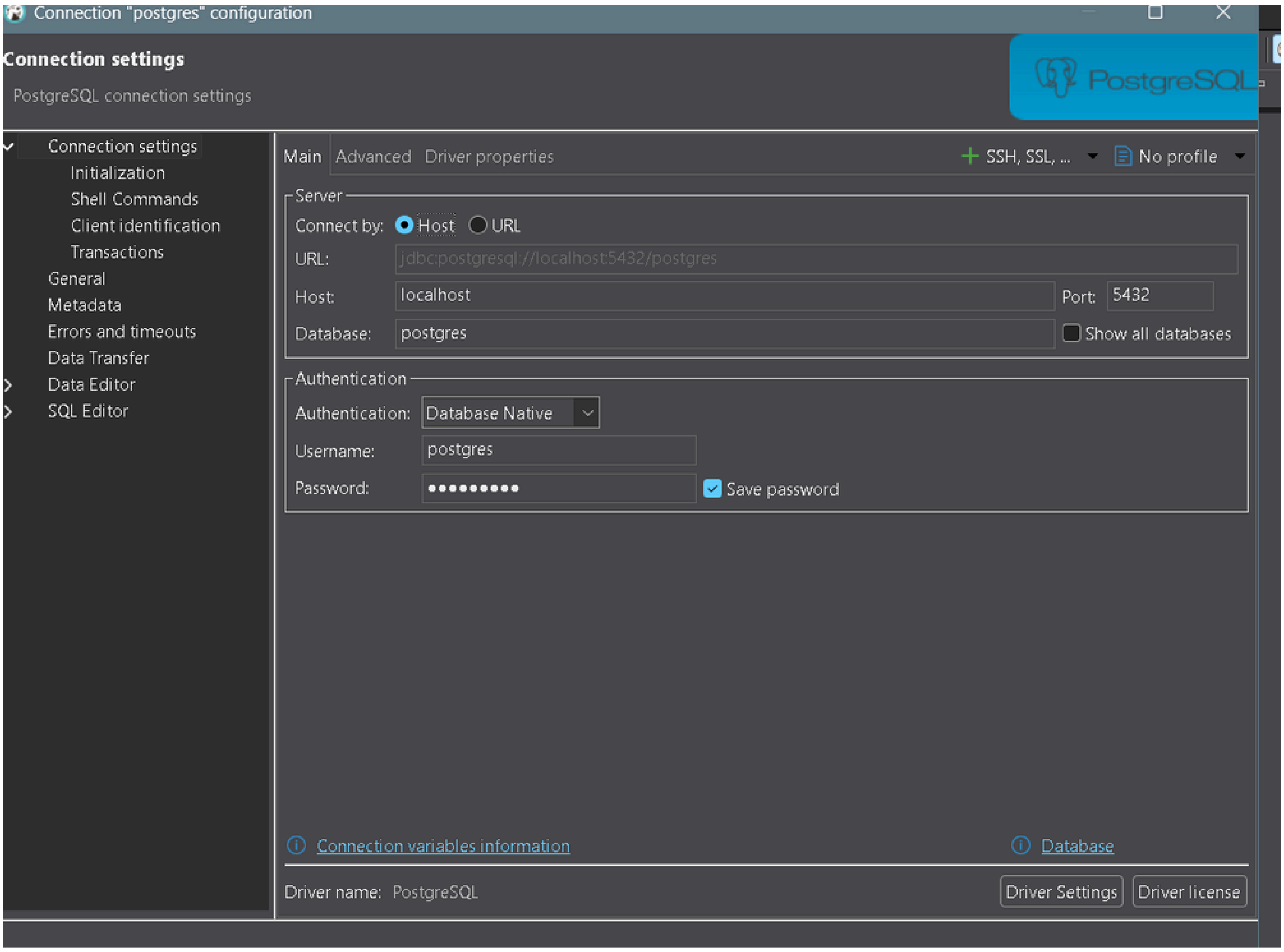
Windows 1. Installation process here 2.

Follow this video: Getting Started with PostGIS in QGIS on Windows

or this artical:https://www.bostongis.com/PrinterFriendly.aspx?content_name=postgis_tut01

after the installation install DBeaver: Download

now open DBeaver and setup new connection and provide username and password which you defined when installing postgres and postgis.


Connection "postgres" configuration

Connection settings

PostgreSQL connection settings

Server

Connect by: [✓ Host] URL

URL: jdbc:postgresql://localhost:5432/postgres

Host: localhost Port: 5432

Database: postgres ☐ Show all databases

Authentication

Authentication: Database Native

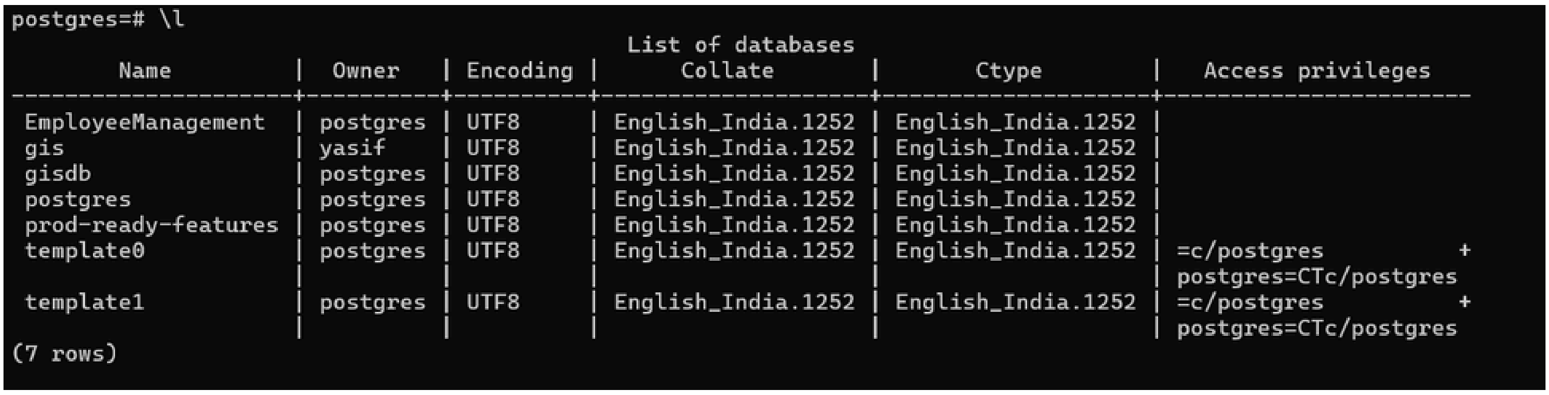
Username: postgres

Password: ●●●●●●●●●● ☑ Save password

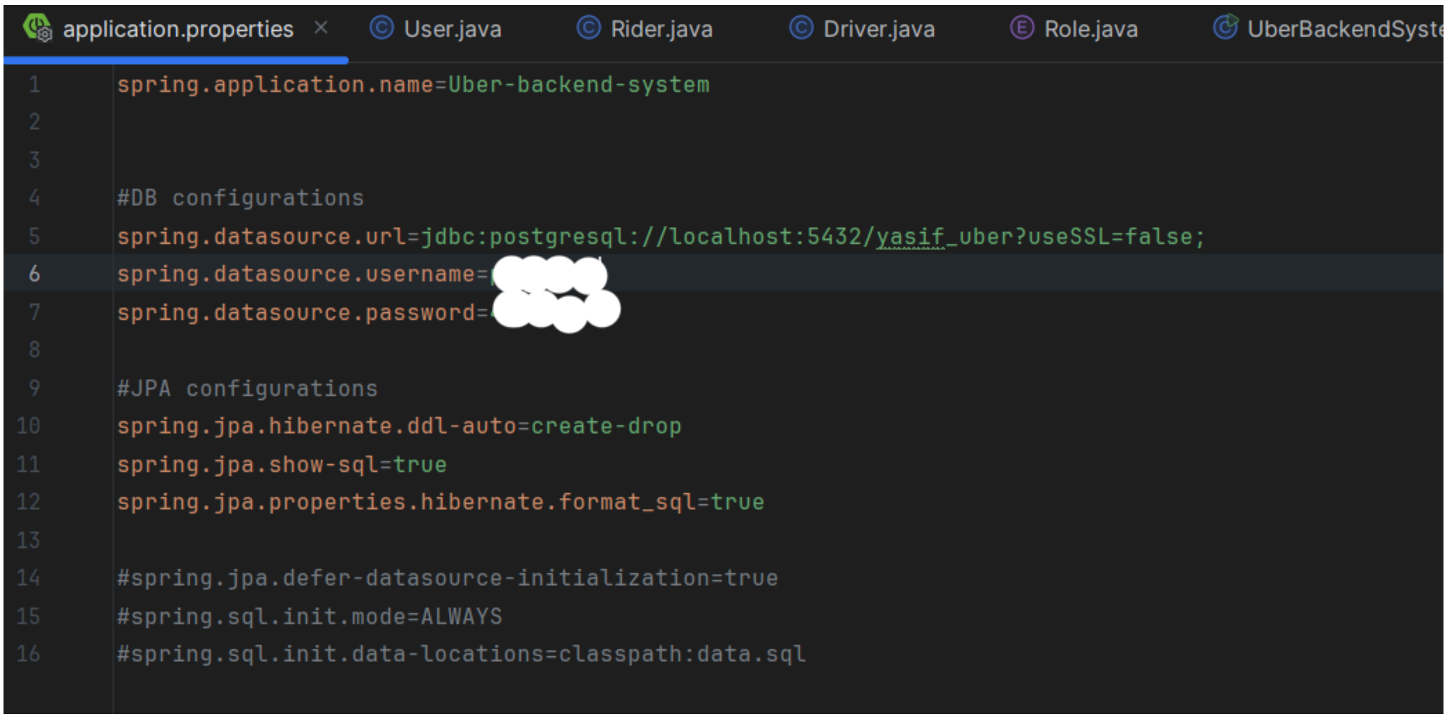
Driver name: PostgreSQL

after make new connection make one Database of any name you want in the psql command

create database anyname;

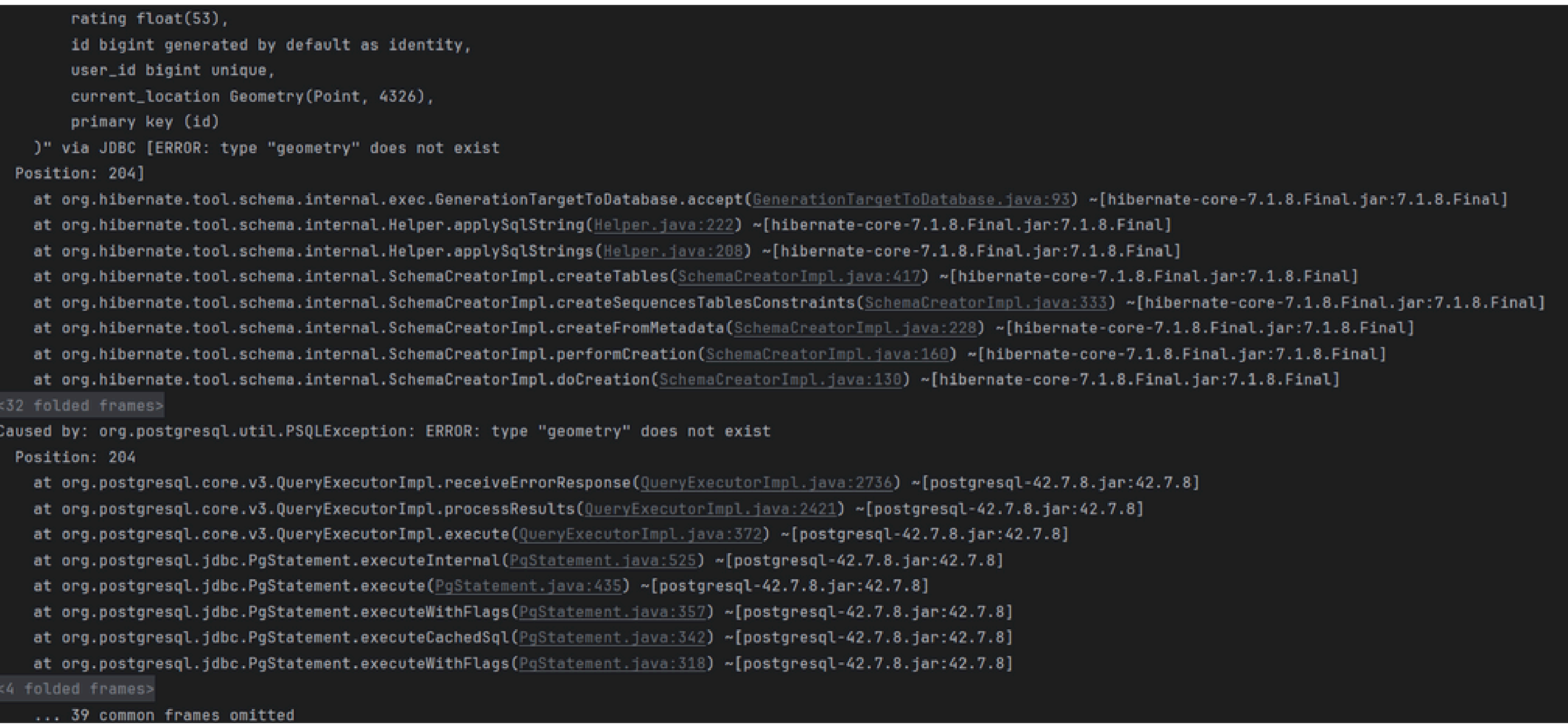

```
postgres=# \l
                                          List of databases
         Name          |  Owner   | Encoding |      Collate       |       Ctype        |   Access privileges
-----------------------+----------+----------+--------------------+--------------------+-----------------------
 EmployeeManagement    | postgres | UTF8     | English_India.1252 | English_India.1252 |
 gis                   | yasif    | UTF8     | English_India.1252 | English_India.1252 |
 gisdb                 | postgres | UTF8     | English_India.1252 | English_India.1252 |
 postgres              | postgres | UTF8     | English_India.1252 | English_India.1252 |
 prod-ready-features   | postgres | UTF8     | English_India.1252 | English_India.1252 |
 template0             | postgres | UTF8     | English_India.1252 | English_India.1252 | =c/postgres          +
                       |          |          |                    |                    | postgres=CTc/postgres
 template1             | postgres | UTF8     | English_India.1252 | English_India.1252 | =c/postgres          +
                       |          |          |                    |                    | postgres=CTc/postgres
(7 rows)
```

now after creating database and database name its username and password in the properties like this



```
spring.application.name=Uber-backend-system


#DB configurations
spring.datasource.url=jdbc:postgresql://localhost:5432/yasif_uber?useSSL=false;
spring.datasource.username=[redacted]
spring.datasource.password=[redacted]

#JPA configurations
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

#spring.jpa.defer-datasource-initialization=true
#spring.sql.init.mode=ALWAYS
#spring.sql.init.data-locations=classpath:data.sql
```

Now if you run your application you will get error like this which says



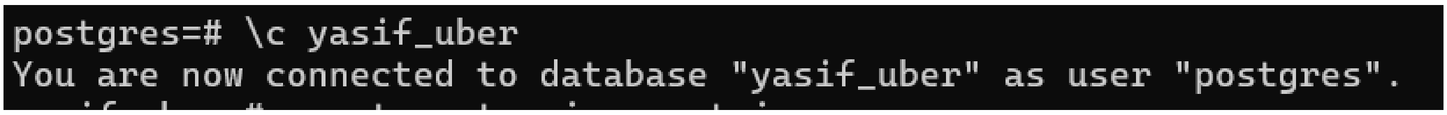
```
        rating float(53),
        id bigint generated by default as identity,
        user_id bigint unique,
        current_location Geometry(Point, 4326),
        primary key (id)
    )" via JDBC [ERROR: type "geometry" does not exist
  Position: 204]
    at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase.java:93) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
    at org.hibernate.tool.schema.internal.Helper.applySqlString(Helper.java:222) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
    at org.hibernate.tool.schema.internal.Helper.applySqlStrings(Helper.java:208) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.createTables(SchemaCreatorImpl.java:417) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.createSequencesTablesConstraints(SchemaCreatorImpl.java:333) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.createFromMetadata(SchemaCreatorImpl.java:228) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.performCreation(SchemaCreatorImpl.java:160) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.doCreation(SchemaCreatorImpl.java:130) ~[hibernate-core-7.1.8.Final.jar:7.1.8.Final]
<32 folded frames>
Caused by: org.postgresql.util.PSQLException: ERROR: type "geometry" does not exist
  Position: 204
    at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2736) ~[postgresql-42.7.8.jar:42.7.8]
    at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2421) ~[postgresql-42.7.8.jar:42.7.8]
    at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:372) ~[postgresql-42.7.8.jar:42.7.8]
    at org.postgresql.jdbc.PgStatement.executeInternal(PgStatement.java:525) ~[postgresql-42.7.8.jar:42.7.8]
    at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:435) ~[postgresql-42.7.8.jar:42.7.8]
    at org.postgresql.jdbc.PgStatement.executeWithFlags(PgStatement.java:357) ~[postgresql-42.7.8.jar:42.7.8]
    at org.postgresql.jdbc.PgStatement.executeCachedSql(PgStatement.java:342) ~[postgresql-42.7.8.jar:42.7.8]
    at org.postgresql.jdbc.PgStatement.executeWithFlags(PgStatement.java:318) ~[postgresql-42.7.8.jar:42.7.8]
<4 folded frames>
    ... 39 common frames omitted
```

geometry does not exist.

why this error comes because you do not have postgis extension in the database. remember every database has there own postgis extension for each database. now here after creating database in the psql shell. After that select that database by using

This error occurs because the PostGIS extension is not enabled in your database. Remember, every PostgreSQL database needs its own PostGIS extension.

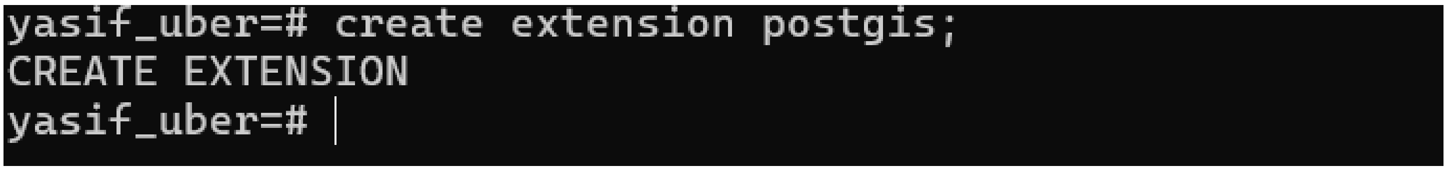
“\c DB_name”



```
postgres=# \c yasif_uber
You are now connected to database "yasif_uber" as user "postgres".
```

now create postgis extension

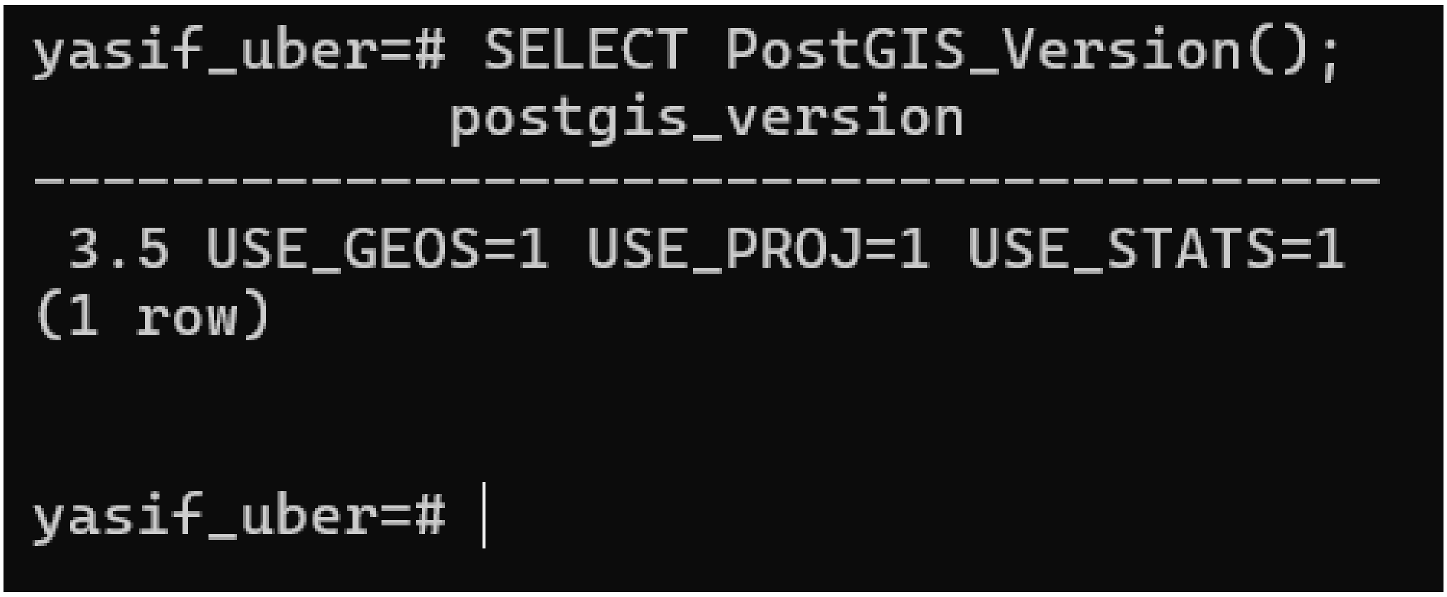
create extension postgis;



```
yasif_uber=# create extension postgis;
CREATE EXTENSION
yasif_uber=#
```

if you want to verify just run this

select PostGIS_Version();



```
yasif_uber=# SELECT PostGIS_Version();
            postgis_version
---------------------------------------
 3.5 USE_GEOS=1 USE_PROJ=1 USE_STATS=1
(1 row)


yasif_uber=#
```

Now the error will gone.

Stay tuned