FPT SOFTWARE QUY NHON AI Valley

SPRING BOOT Persistence

KienNCT3 - Mentor HaiNV21

Content:

1) Quick Guide on Loading Initial Data with Spring Boot.

2) Spring Boot with Multiple SQL Import Files.

3) Show Hibernate/JPA SQL Statements from Spring Boot.

4) Spring Boot With H2 Database.

5) Configure and Use Multiple Data Sources in Spring Boot.

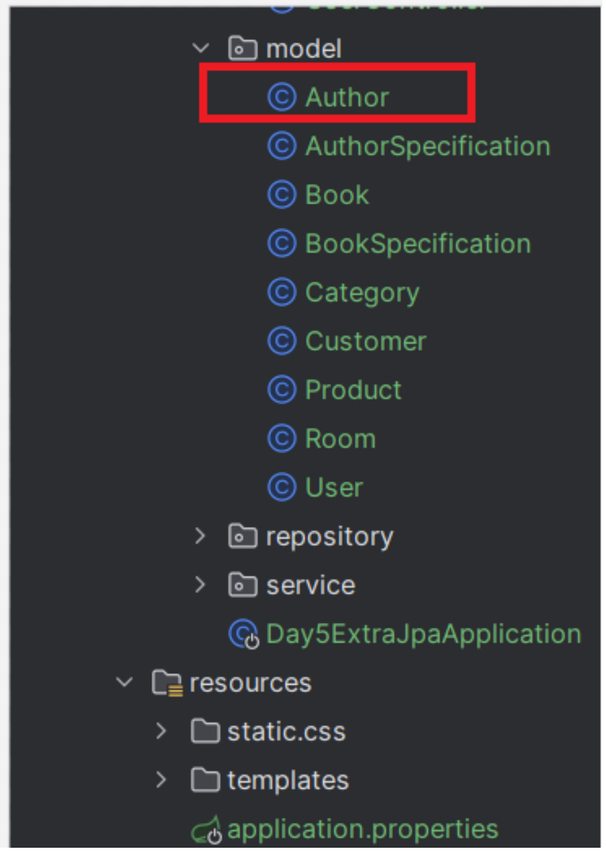
What is “Persistence” in context of Spring Boot

- In Spring Boot, "persistence" is a concept that refers to an application's ability to save and retrieve data from a data source such as a database. This includes designing and implementing classes and methods to perform CRUD operations on data.

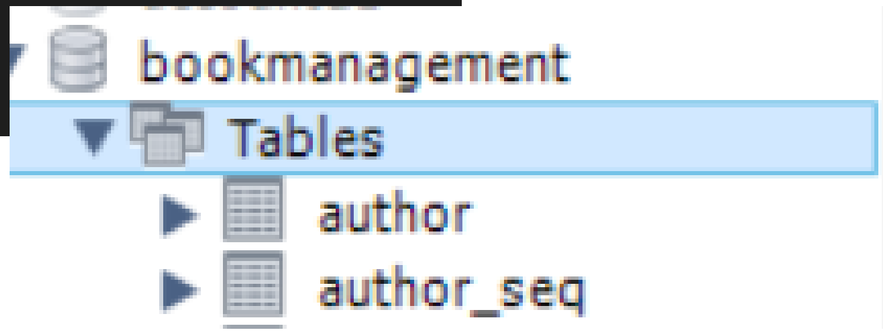
- In the context of Spring Boot, "persistence" typically involves using ORM technologies such as Hibernate or JPA to interact with the database. ORM helps us work with data in Java applications without paying much attention to how the data is stored in the database.

Quick Guide on Loading Initial Data with Spring Boot

- Spring Boot makes it really easy to manage our database changes. If we leave the default configuration, it'll search for entities in our packages and create the respective tables automatically.




```
spring.datasource.url=jdbc:mysql://localhost:3306/bookmanagement
spring.datasource.username=root
spring.datasource.password=trungkien

#JPA config
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```



Result in database

Quick Guide on Loading Initial Data with Spring Boot

- Spring Boot makes it really easy to manage our database changes. If we leave the default configuration, it'll search for entities in our packages and create the respective tables automatically.

- But we'll sometimes need more fine-grained control over the database alterations. And that's when we can use the **data.sql** and **schema.sql** files in Spring.

Quick Guide on Loading Initial Data with Spring Boot

- Quick guide:

Step 1: Create file data sql like: data.sql, schemal.sql,...

Step 2: Put it in folder: main/resources

Step 3: Config in configuration file .yml or .properties

Step 4: Spring Boot will automatically run it file when start up.

Quick Guide on Loading Initial Data with Spring Boot

Case 1: Use file data.sql

First we have entity **User** and file **data.sql:**

```java
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
}
```

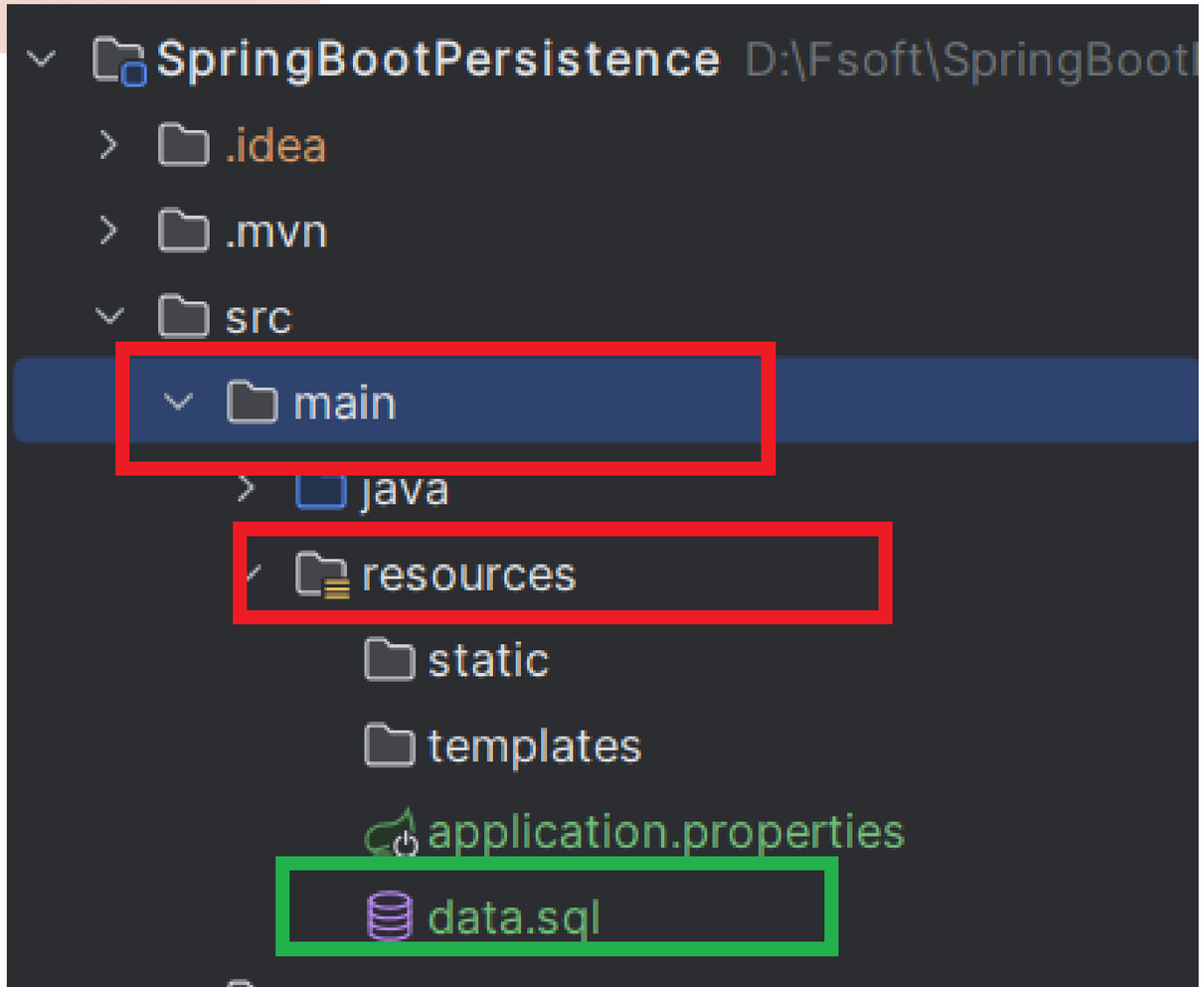
data

Limit to

```sql
insert into user(id, name)
values
    (1, 'ChienNN21'),
    (2, 'AnhNV9'),
    (3, 'KienNCT3'),
    (4, 'HaiNV21');
```

Quick Guide on Loading Initial Data with Spring Boot

Then we put file data.sql in folder main/resources



After that, we config to load initial data in file .properties and run application:

```
#JPA config
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
```

- spring.jpa.defer-datasource-initialization=true: This ensures that the databases used by JPA are ready before any other SQL actions are performed.

- spring.sql.init.mode=always: SQL files will always be executed every time the application is started. This ensures that the original data in the database is updated or re-initialized every time the application starts.

Quick Guide on Loading Initial Data with Spring Boot

Result in database:

```
1 •  SELECT * FROM bookmanagement.user;
```

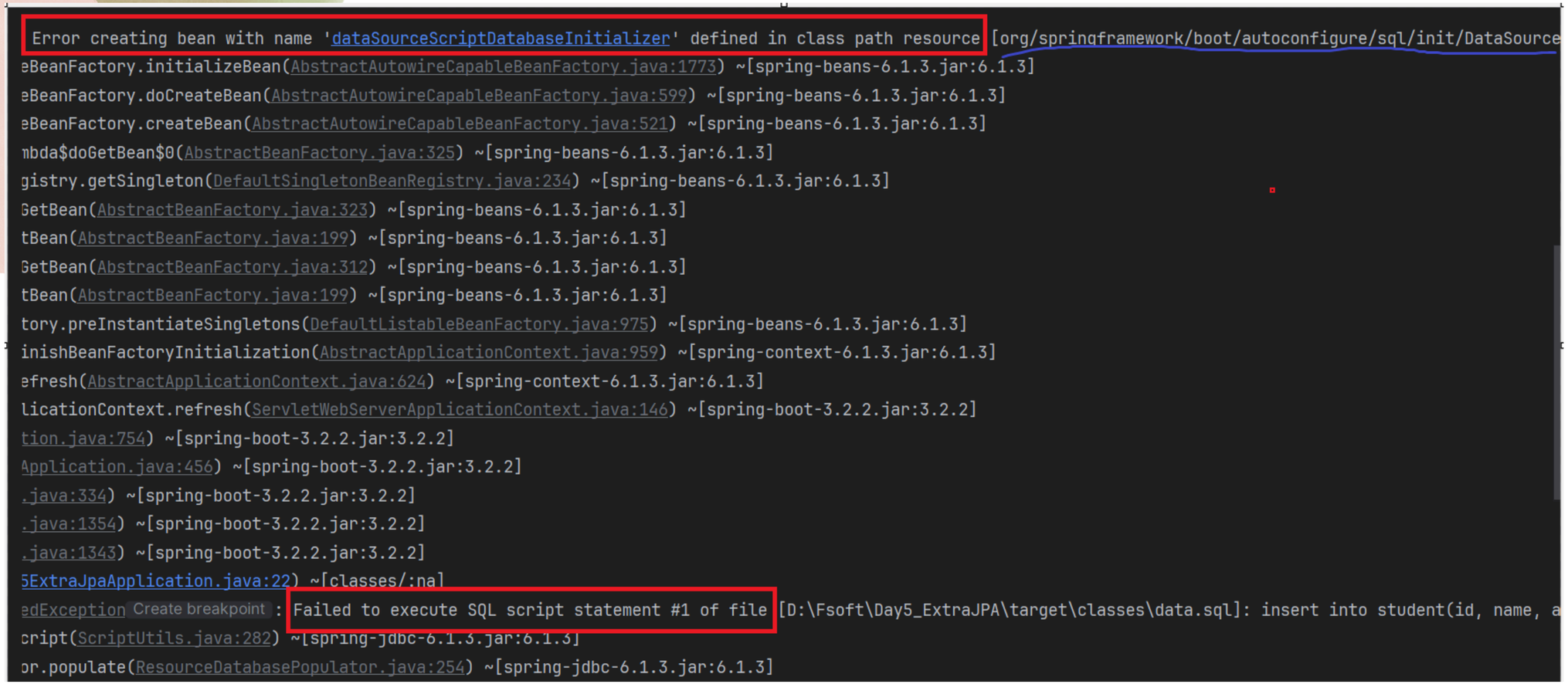
Result Grid | Filter Rows: | Edit:

id	name
4	Nguyen Van A
5	Nguyen Van D
6	Nguyen Van E
7	Nguyen Van F
9	Nguyen Van B
NULL	NULL

data.sql

```sql
insert into user(id, name)
values
    (4, 'Nguyen Van A'),
    (5, 'Nguyen Van D'),
    (6, 'Nguyen Van E'),
    (7, 'Nguyen Van F'),
    (9, 'Nguyen Van B');
```

After load successfully, we can set spring.sql.init.mode = never to not load continue, if we not set, then the error dataSource will be occured:



Solution:

```
spring.sql.init.mode=never
```

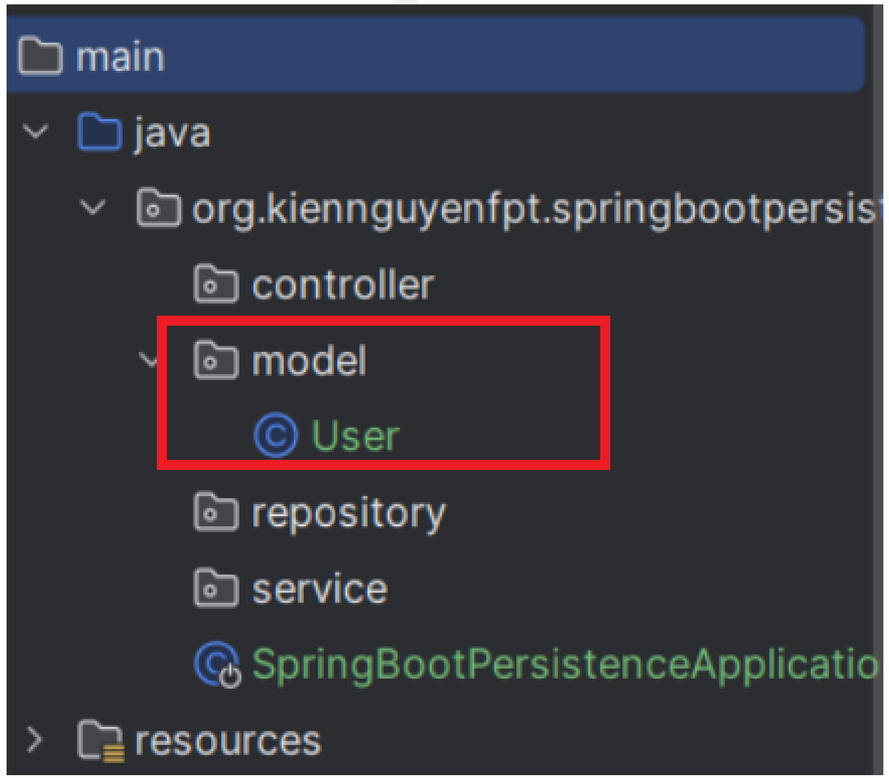
At this time, database structure will be based on this statement:

```
spring.jpa.hibernate.ddl-auto=
```

Quick Guide on Loading Initial Data with Spring Boot

Case 2: Use file schema.sql

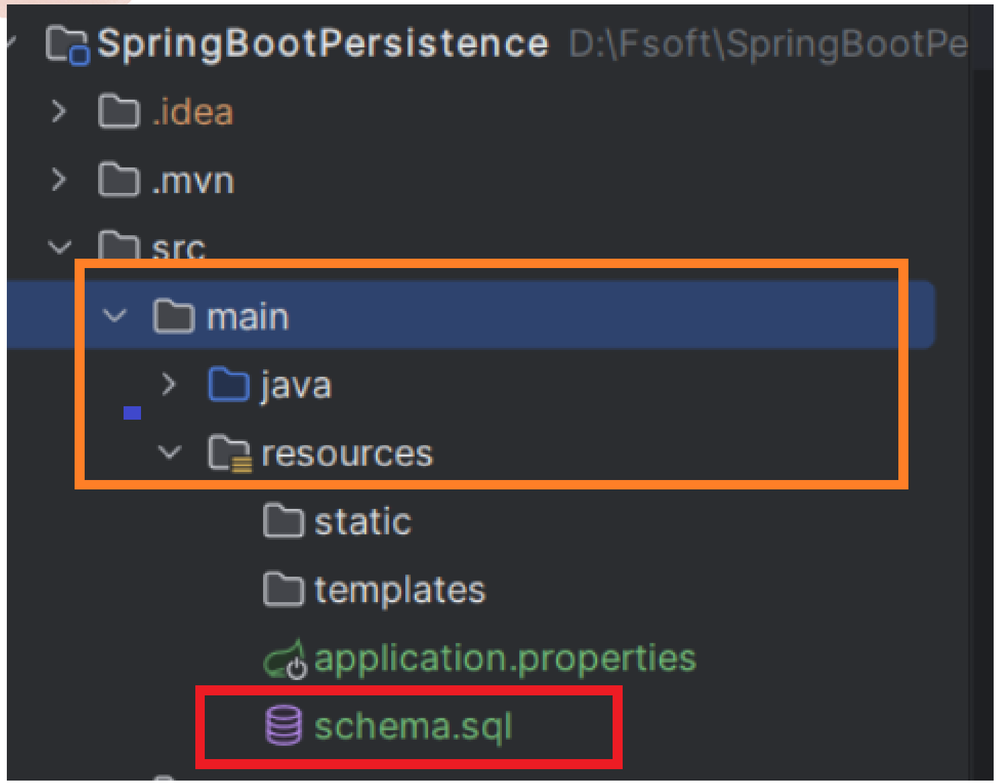
Suppose that we don't have Customer entity in Java app, and our purpose is to load initial data from Customer table



Quick Guide on Loading Initial Data with Spring Boot

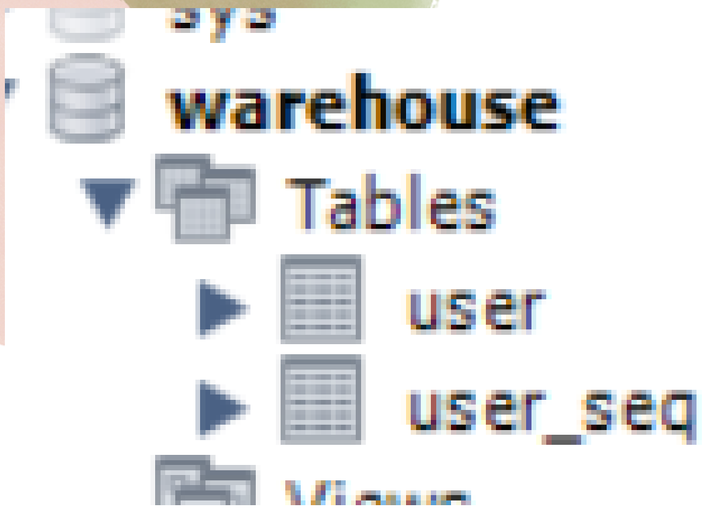
Case 2: Use file schema.sql

We can create file schema.sql to create and insert data to Customer table //insert statement to customer

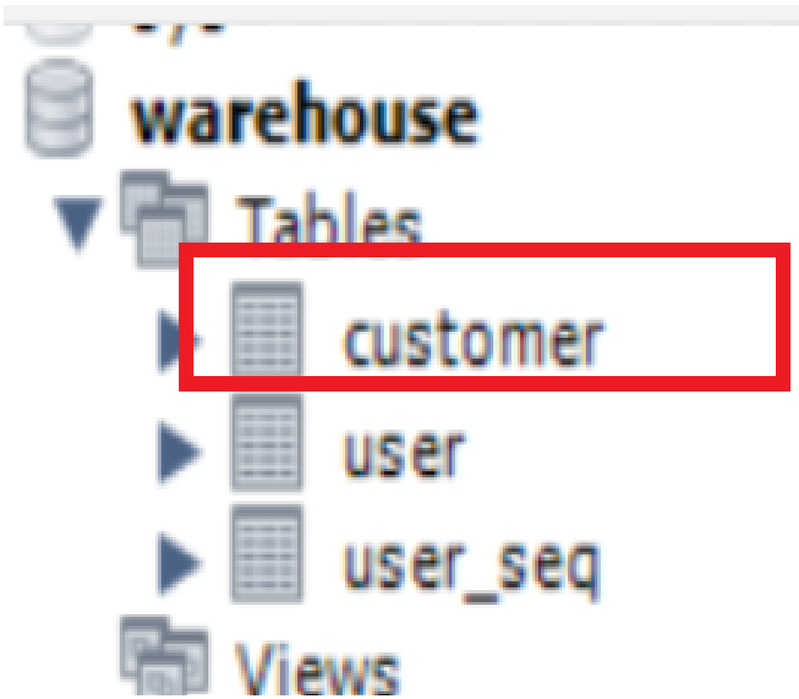


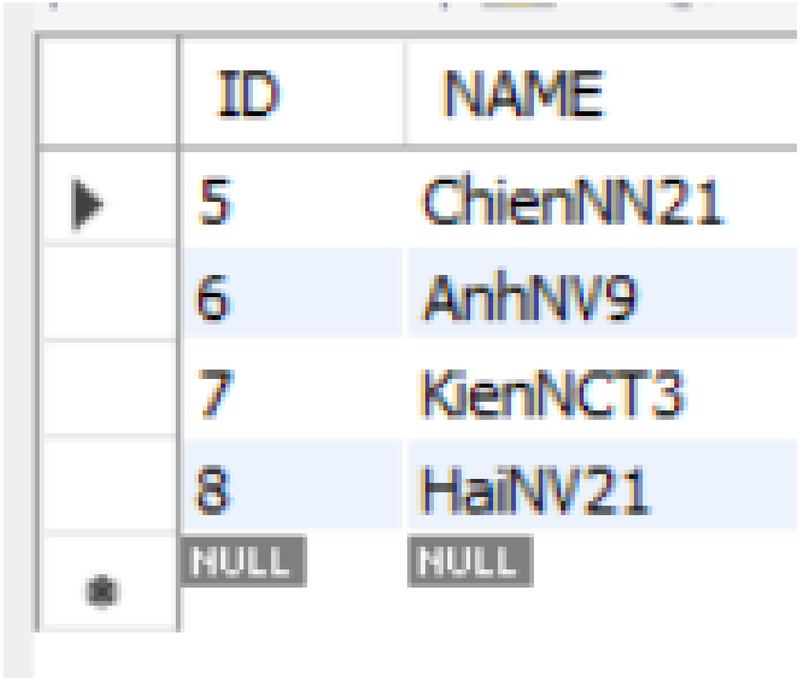

```sql
create table CUSTOMER(
  ID int not null AUTO_INCREMENT,
  NAME varchar(100) not null,
  PRIMARY KEY ( ID )
);

insert into CUSTOMER(id, name)
values
    (5, 'ChienNN21'),
    (6, 'AnhNV9'),
    (7, 'KienNCT3'),
    (8, 'HaiNV21');
```

Result in database:



Before(not containing customer table)





	ID	NAME
▶	5	ChienNN21
	6	AnhNV9
	7	KienNCT3
	8	HaiNV21
*	NULL	NULL

After

Case 3: Use both file schema.sql and data.sql

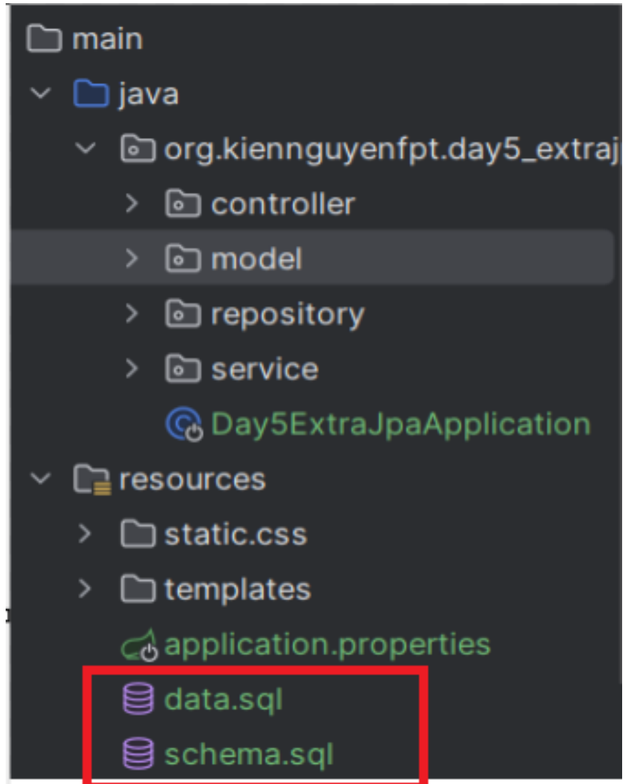
```java
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long Id;
    private String name;
}
```

data.sql

```sql
insert into student(id, name)
values
    (10, 'Nguyen Van A'),
    (11, 'Nguyen Van D'),
    (12, 'Nguyen Van E'),
    (13, 'Nguyen Van F'),
    (14, 'Nguyen Van B');
```

```sql
create table IF NOT EXISTS student
(
    id bigint not null auto_increment primary key ,
    name varchar(255) not null,
    age integer
);
```

In “schema.sql”, we add “age” field that does not have in Student entity



	id	name	age
▶	10	Nguyen Van A	NULL
	11	Nguyen Van D	NULL
	12	Nguyen Van E	NULL
	13	Nguyen Van F	NULL
	14	Nguyen Van B	NULL
*	NULL	NULL	

Result in database

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=none
```

schema.sql is read for any additional schema changes, and data.sql is executed to populate the database.

Testing using @Sql

This annotation is used to specify specific SQL files that we want to execute during application database initialization.

import_students.sql

```sql
delete from student;
insert into student(id, name)
values
    (20, 'KienNCT3'),
    (21, 'AnhVN9');
```

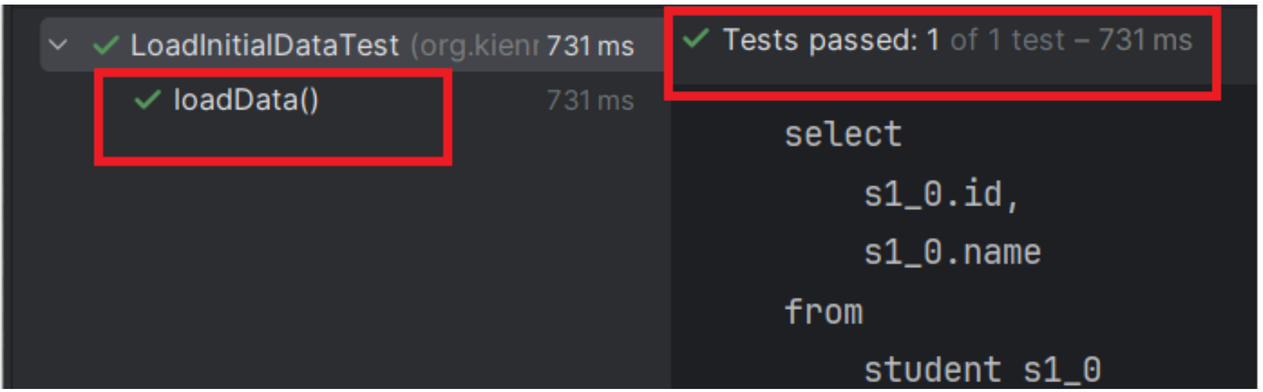
students_schema.sql

```sql
create table IF NOT EXISTS student
(
    id bigint not null auto_increment primary key,
    name varchar(255) not null,
    age integer
);
```

Assume that we have two file .sql

Testing using @Sql

```java
@SpringBootTest
@Sql({"/students_schema.sql", "/import_students.sql"})
public class LoadInitialDataTest {
    2 usages
    private final StudentRepository rep;
    new *
    @Autowired
    public LoadInitialDataTest(StudentRepository rep) {
        this.rep = rep;
    }
    new *
    @Test
    public void loadData() {
        assertEquals( expected: 2, rep.findAll().size());
    }
}
```



Check result in database:

```sql
SELECT * FROM bookmanagement.student;
```

Result Grid | Filter Rows: | Edit:

id	name	age
20	KienNCT3	NULL
21	AnhVN9	NULL
NULL	NULL	NULL

Spring Boot with Multiple SQL Import files

- Spring Boot allows us to import data into our database – mainly to prepare data for integration tests.

- There are two possibilities. We can use import.sql (Hibernate support) or data.sql (Spring JDBC support) files to load data.

- However, sometimes we want to split one big SQL file into a few smaller ones, for better readability or to share some files with an init data between modules.

Spring Boot with Multiple SQL Import files

students_schema.sql

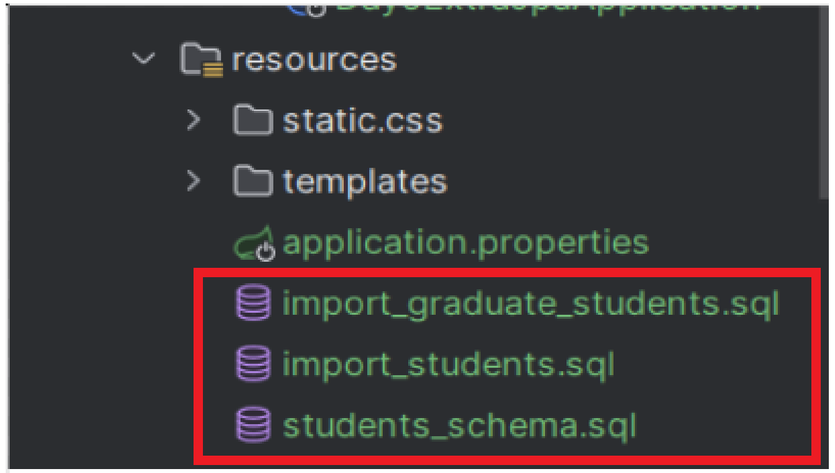
```sql
create table IF NOT EXISTS student
(
    id bigint not null auto_increment primary key,
    name varchar(255) not null,
    age integer
);
```

import_graduate_students.sql

```sql
insert into student(id, name, age)
values
    (30, 'Nguyen Van A', 30),
    (31, 'Nguyen Van B', 31);
```

import_students.sql

```sql
insert into student(id, name, age)
values
    (2, 'KienNCT3', 20),
    (21, 'AnhVN9', 19);
```



resources folder

Spring Boot with Multiple SQL Import files

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.sql.init.data-locations=classpath:students_schema.sql,classpath:import_graduate_students.sql,classpath:import_students.sql

spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
```

```
1 •  SELECT * FROM bookmanagement.student;
```

Result Grid | Filter Rows: | Edit:

id	age	name
2	20	KienNCT3
21	19	AnhVN9
30	30	Nguyen Van A
31	31	Nguyen Van B

- spring.sql.init.data-locations: specify the locations of files containing SQL data that Spring Boot will use to initialize data in the database when the application is started.
//fixing: import_*_students

Show Hibernate/JPA SQL Statements from Spring Boot

1. Config in file application.properties

```
#JPA config
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
```

spring.jpa.show-sql=true: When set to true, Spring Boot will show SQL statements generated by JPA during application execution. This is very useful during development and debugging when we wanna test the SQL queries that our application is generating.

Show Hibernate/JPA SQL Statements from Spring Boot

1. Config in file application.properties

```
#JPA config
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
```

localhost:8080/book?id=159


```
"content": [
    {
        "id": 159,
        "name": "FPT Bo 1",
        "authorID": {
            "id": 162,
            "name": "Author 10",
            "age": 54,
            "address": "Address 10",
            "phone": "Phone 10"
        },
        "categoryID": {
            "id": 3,
            "name": null
        }
    }
```


Result in console:

```
Hibernate: select b1_0.id,b1_0.author_id,b1_0.category_id,b1_0.name from book b1_0 join author ai1_0 on ai1_0.id=b1_0.author_id where b1_0.id=? limit ?,?
Hibernate: select b1_0.id,b1_0.author_id,b1_0.category_id,b1_0.name from book b1_0 join author ai1_0 on ai1_0.id=b1_0.author_id where b1_0.id=? limit ?,?
Hibernate: select a1_0.id,a1_0.address,a1_0.age,a1_0.name,a1_0.phone from author a1_0 where a1_0.id=?
Hibernate: select c1_0.id,c1_0.name from category c1_0 where c1_0.id=?
```

2.1 Config in file **application.properties**

To beautify pretty print the SQL, we can add:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Room {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long bookID;
    @ManyToOne
    private User userID;
    @ManyToOne
    Product productId;
    private int price;
}
```

Result in console:

```
Hibernate:
    create table room (
        bookid bigint not null,
        checkin datetime(6),
        price integer not null,
        product_id_id bigint,
        userid_id bigint,
        primary key (bookid)
    ) engine=InnoDB
```

Show Hibernate/JPA SQL Statements from Spring Boot

2.2 Configuring loggers

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

- The first line logs the SQL queries.
- The second statement logs the prepared statement parameters.

Show Hibernate/JPA SQL Statements from Spring Boot

2.2 Configuring loggers

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

```
2024-03-11T10:21:27.651+07:00 DEBUG 27984 --- [nio-8080-exec-1] org.hibernate.SQL                        : select a1_0.id,a1_0.address,a1_0.age,a1_0.name,a1_0
2024-03-11T10:21:27.654+07:00 DEBUG 27984 --- [nio-8080-exec-1] org.hibernate.SQL                        : select a1_0.id,a1_0.address,a1_0.age,a1_0.name,a1_0
2024-03-11T10:21:27.668+07:00 DEBUG 27984 --- [nio-8080-exec-1] org.hibernate.SQL                        : select count(b1_0.id) from book b1_0 join author ai
```

```
] org.hibernate.SQL                       : select a1_0.id,a1_0.address,a1_0.age,a1_0.name,a1_0.phone from author a1_0 where a1_0.id=?
] org.hibernate.SQL                       : select a1_0.id,a1_0.address,a1_0.age,a1_0.name,a1_0.phone from author a1_0 where a1_0.id=?
] org.hibernate.SQL                       : select count(b1_0.id) from book b1_0 join author ai1_0 on ai1_0.id=b1_0.author_id where ai1_0.age between ? and ?
```

Show Hibernate/JPA SQL Statements from Spring Boot

2.2 Configuring loggers //type of logging level

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
spring.jpa.properties.hibernate.format_sql=true
```

localhost:8080/books?authorAgeBetween=40,80


```
"content": [
    {
        "id": 284,
        "name": "FPT Bo 126",
        "authorID": {
            "id": 154,
            "name": "Author 2",
            "age": 48,
            "address": "Address 2",
            "phone": "Phone 2"
        },
        "categoryID": {
            "id": 1,
            "name": "Comic"
        }
    },
    {
        "id": 184,
        "name": "FPT Bo 26",
        "authorID": {
            "id": 155,
            "name": "Author 3",
            "age": 60,
            "address": "Address 3",
            "phone": "Phone 3"
        },
        "categoryID": {
            "id": 1,
            "name": "Comic"
        }
    },
```


Result on localhost:

```
2024-03-11T10:27:37.909+07:00 DEBUG 15840 --- [nio-8080-exec-1] org.hibernate.SQL
    select
        count(b1_0.id)
    from
        book b1_0
    join
        author ai1_0
            on ai1_0.id=b1_0.author_id
    where
        ai1_0.age between ? and ?
```

Show Hibernate/JPA SQL Statements from Spring Boot

2.2 Configuring loggers

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
spring.jpa.properties.hibernate.format_sql=true
```

Another option is to use Logging JdbcTemplate Queries

```
logging.level.org.springframework.jdbc.core.JdbcTemplate=DEBUG
logging.level.org.springframework.jdbc.core.StatementCreatorUtils=TRACE
```

Spring Boot with H2 database

Definition of H2 database:

- Lightweight and easy-to-use relational database, written in Java. The main feature of H2 is that it can operate in embedded mode or server mode.

- Run on many different platforms, including Java SE (Standard Edition), Java EE (Enterprise Edition), and even Android.

Spring Boot with H2 database

Configuration:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

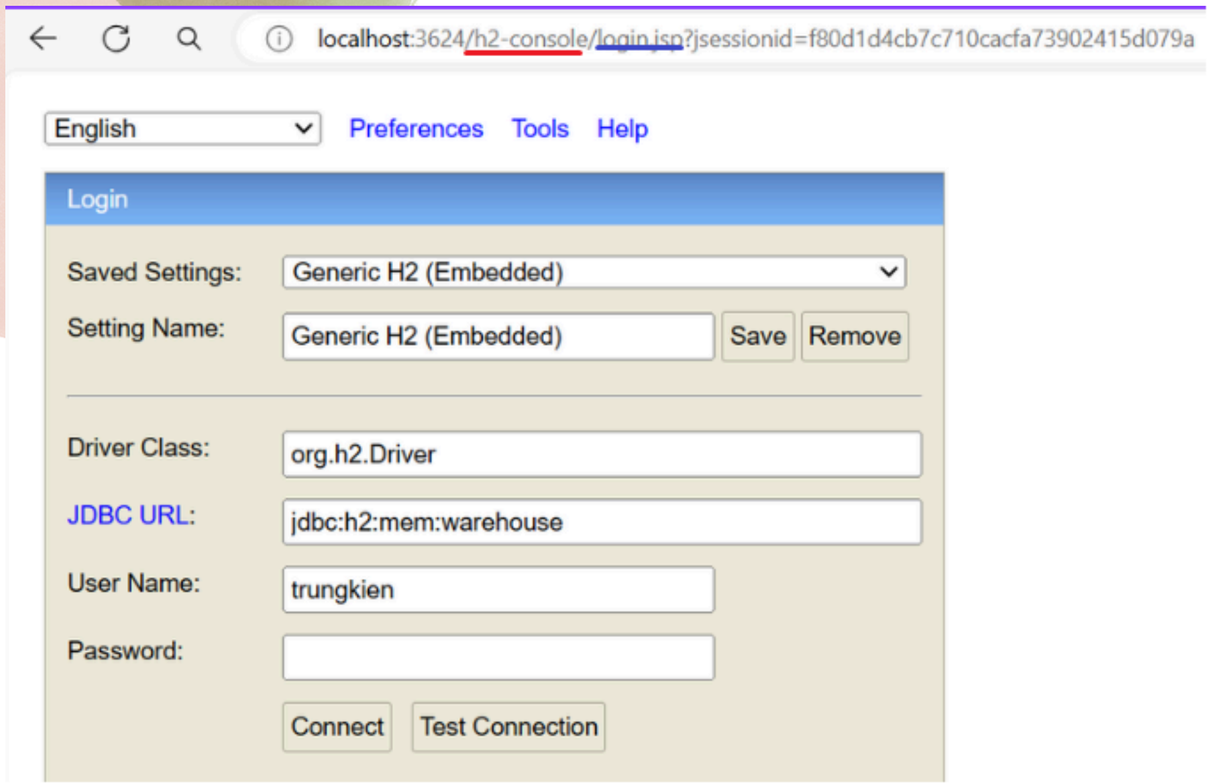
Spring Boot with H2 database

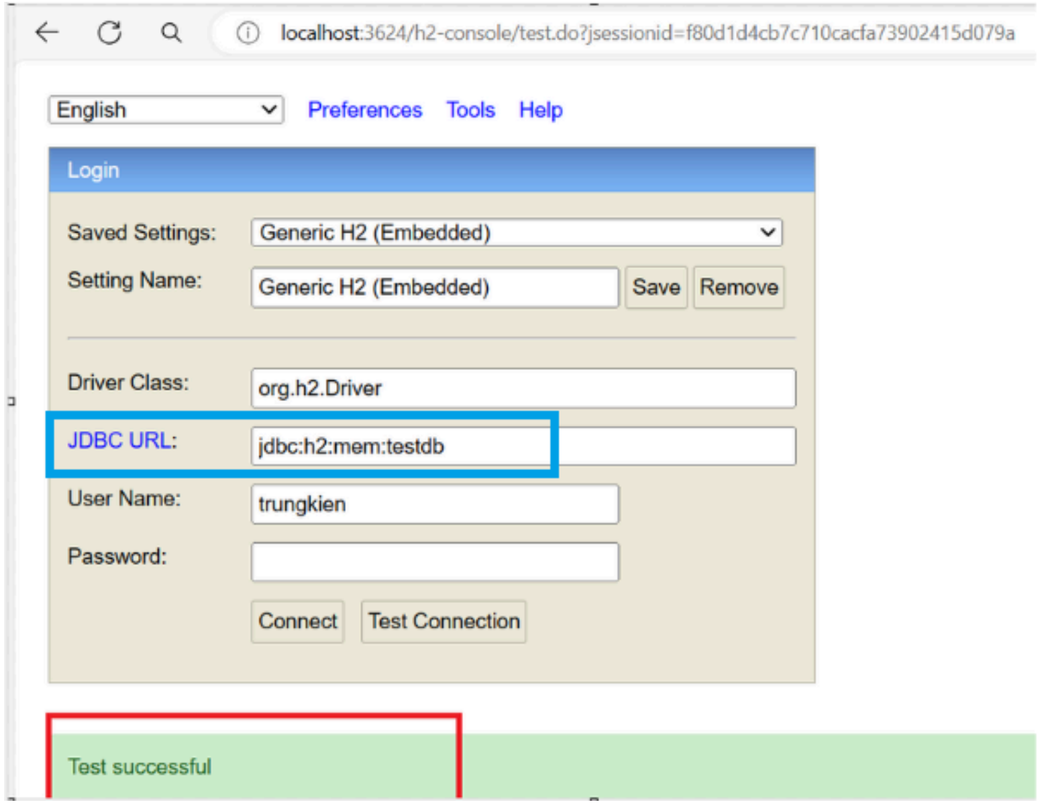
Database configuration:

```
#H2 db
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=trungkien
spring.datasource.password=hellofsoft
spring.datasource.driver-class-name=org.h2.Driver
spring.h2.console.enabled=true
```

This attribute is used to access to H2 console

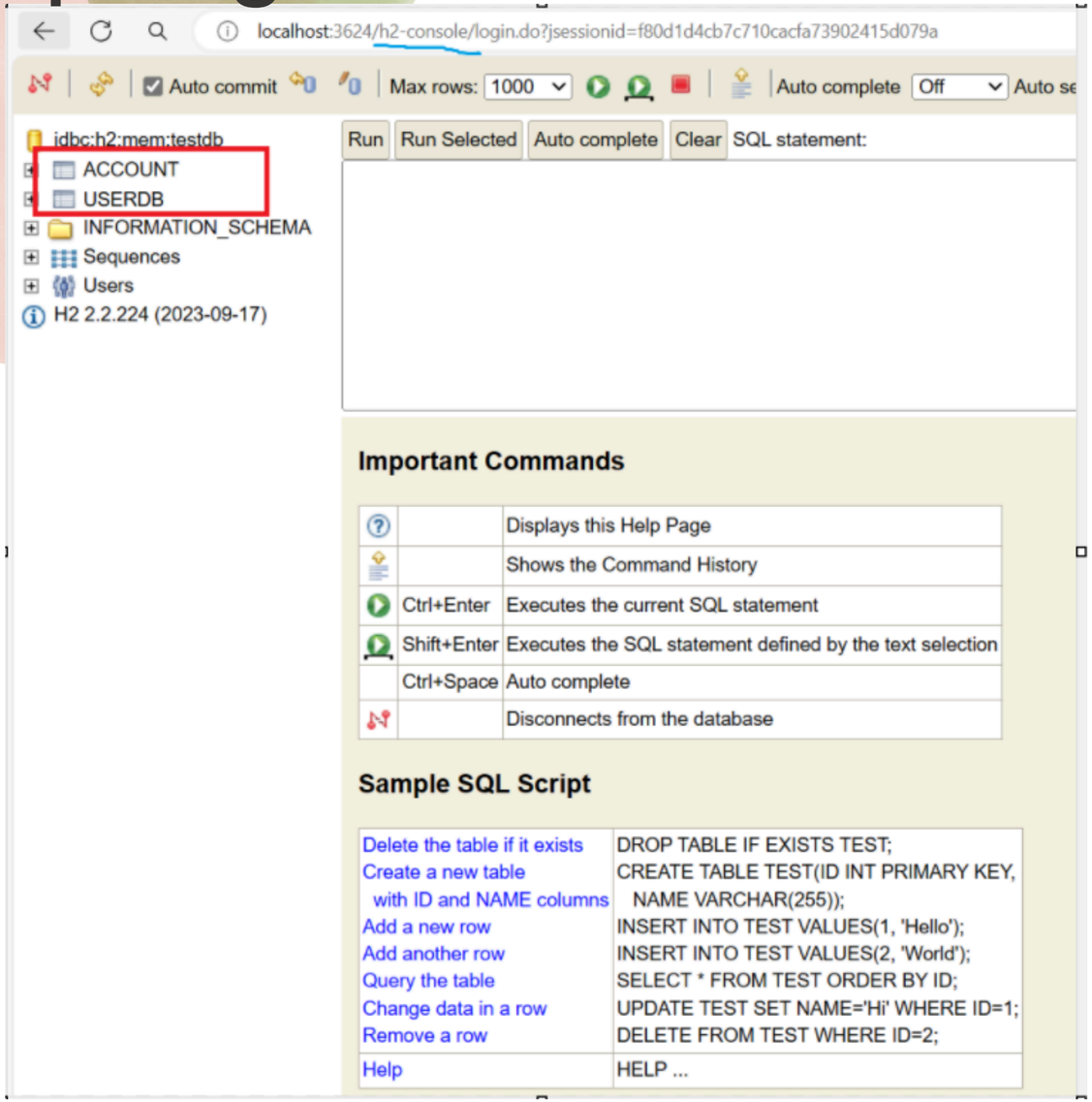
Spring Boot with H2 database






```
Added connection conn0: url=jdbc:h2:mem:testdb user=TRUNGKIEN
```

Spring Boot with H2 database

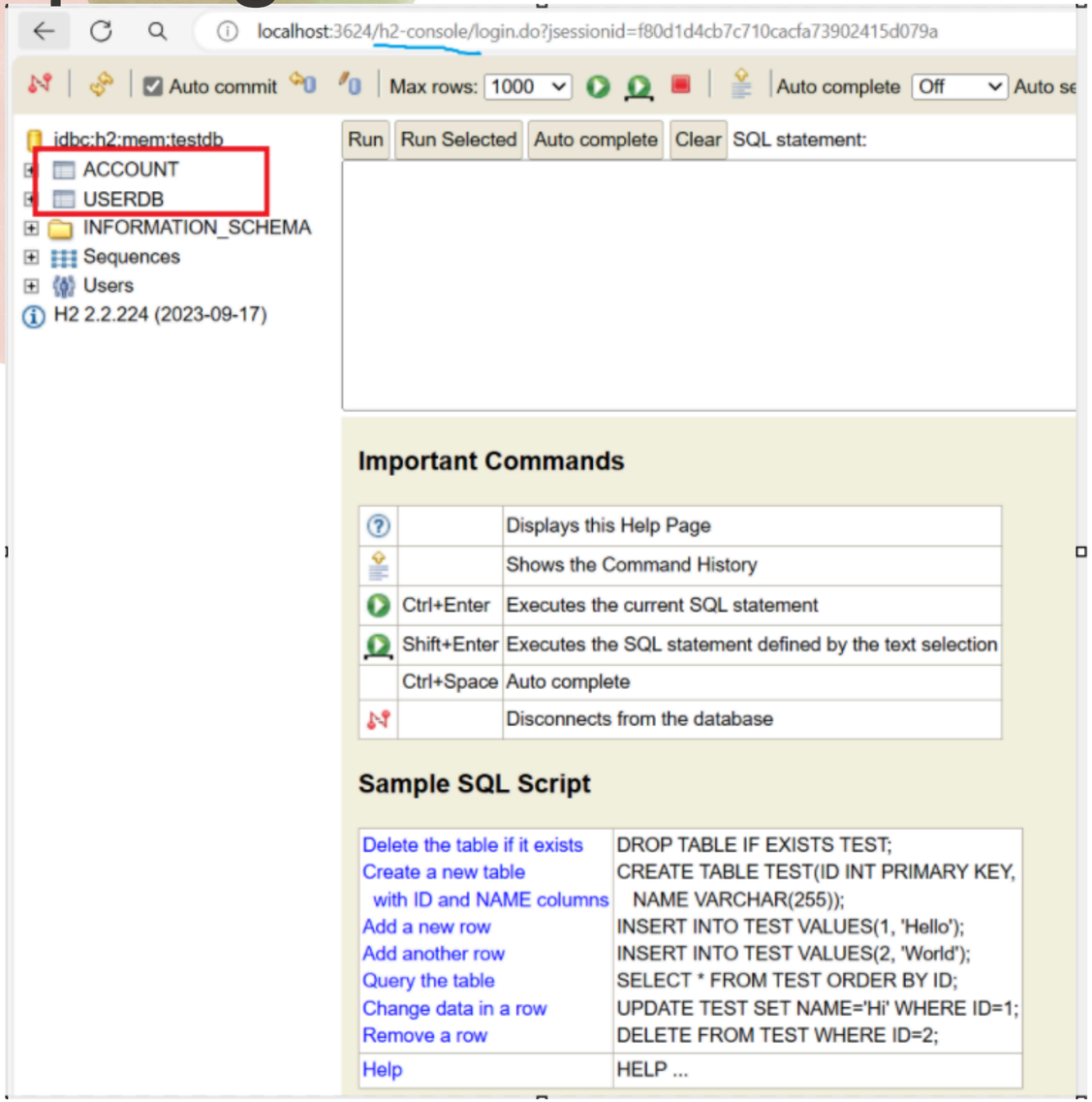


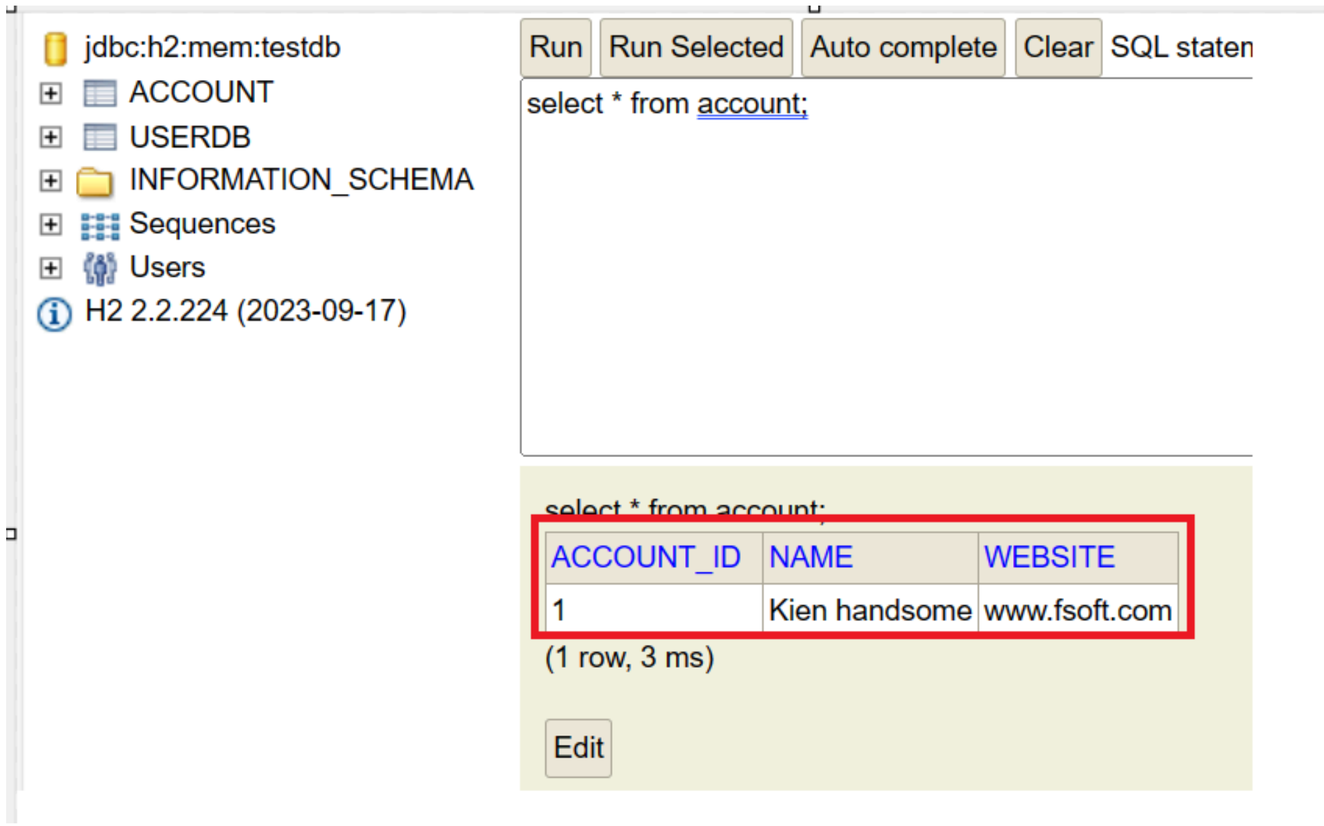

```java
@PostConstruct
private void generateAccounts() {
    Account account = new Account();
    account.setName("Kien handsome");
    account.setWebsite("www.fsoft.com");
    rep.save(account);
```

```
Hibernate:
    insert
    into
        account
        (name, website, account_id)
    values
        (?, ?, default)
```

The interface after connect successfully

Spring Boot with H2 database





The interface after connect successfully

Configure and use multiple data sources in Spring Boot

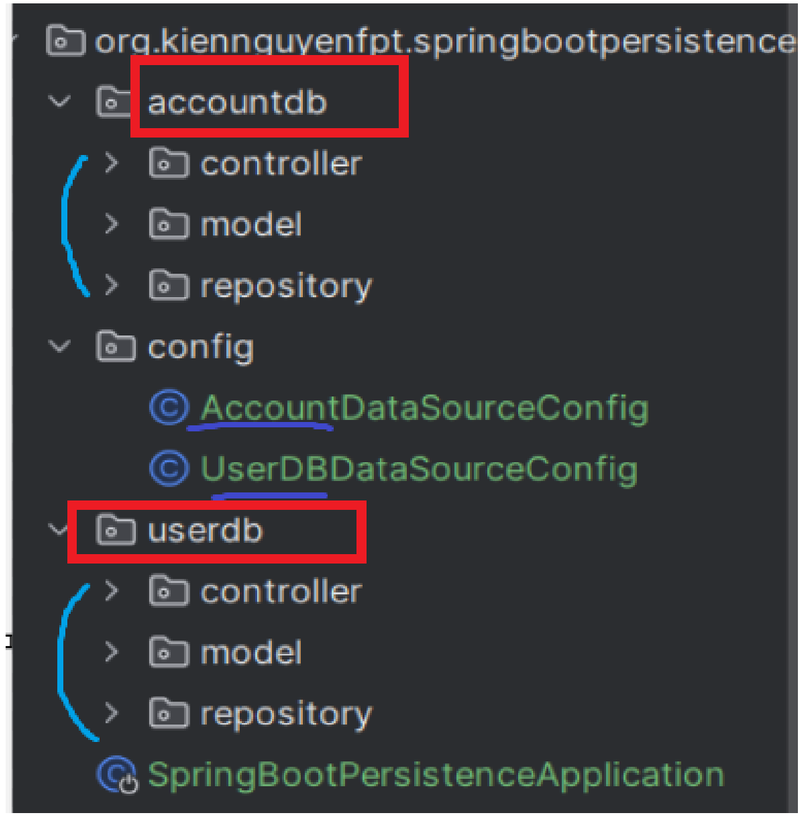
- In Spring Boot application, we usually store data in a single database.

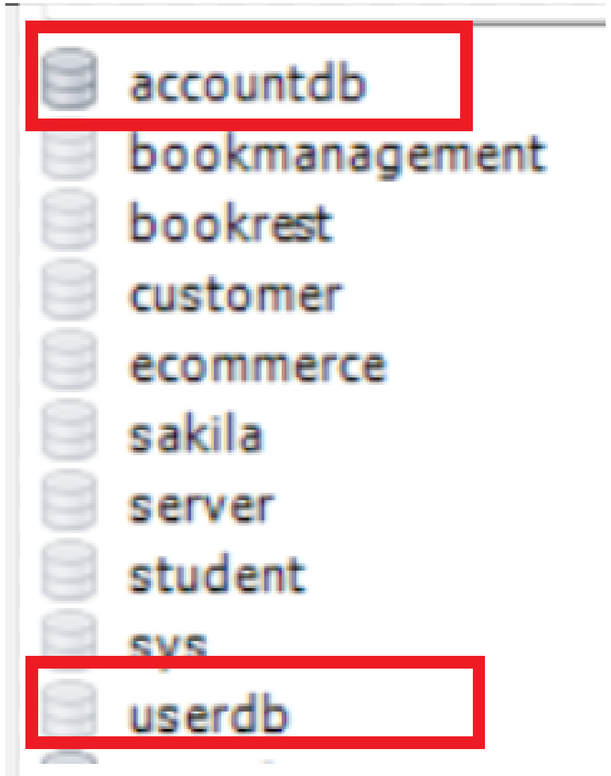
- Sometimes multiple databases need to be accessed. We need configure and use multiple data sources with Spring Boot.

Configure and use multiple data sources in Spring Boot

Step by step to do:

- Create 2 database and data sources.
- Create 2 packages for different entities and repository.
- Create config files for each databases.
- Execute and test

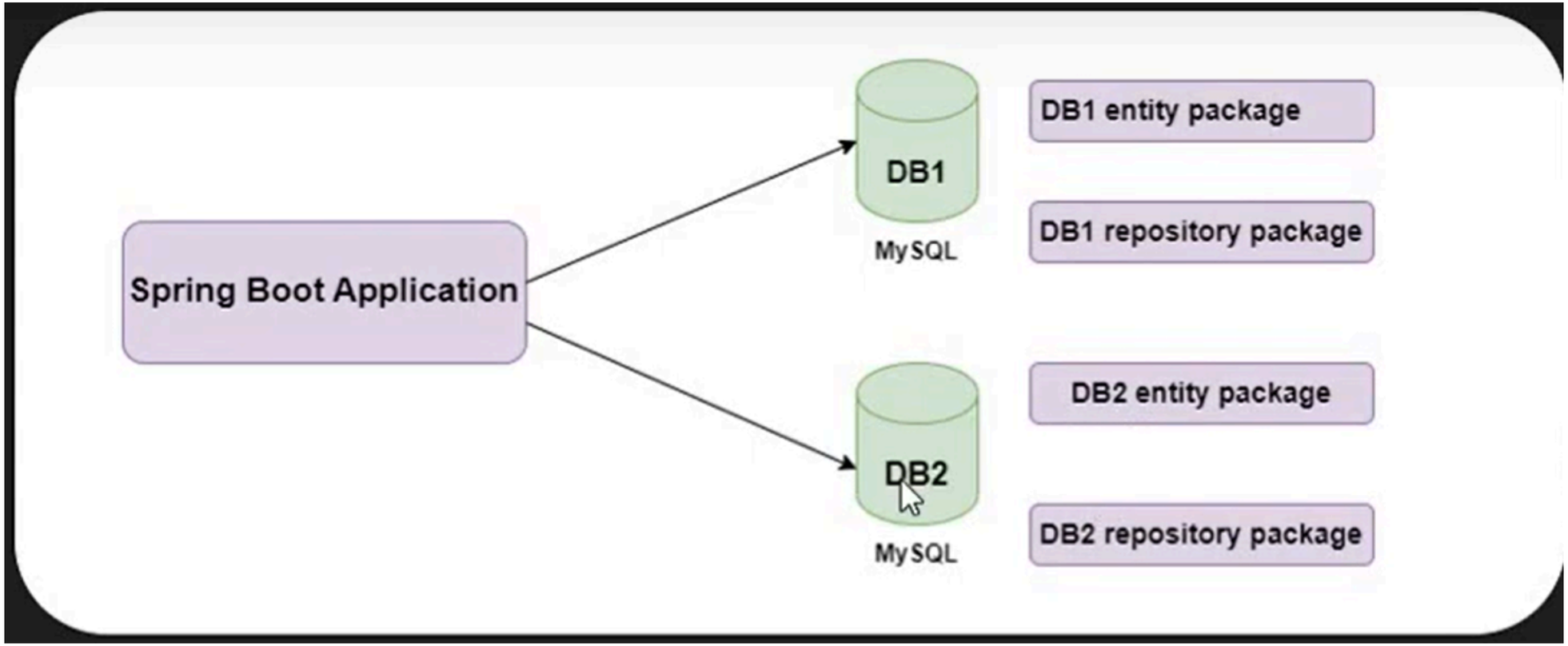




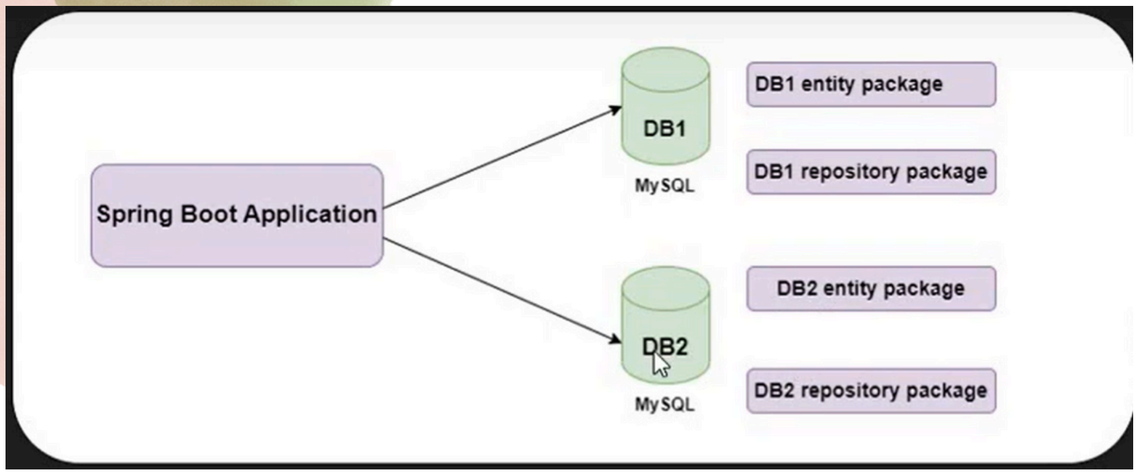
Configure and use multiple data sources in Spring Boot

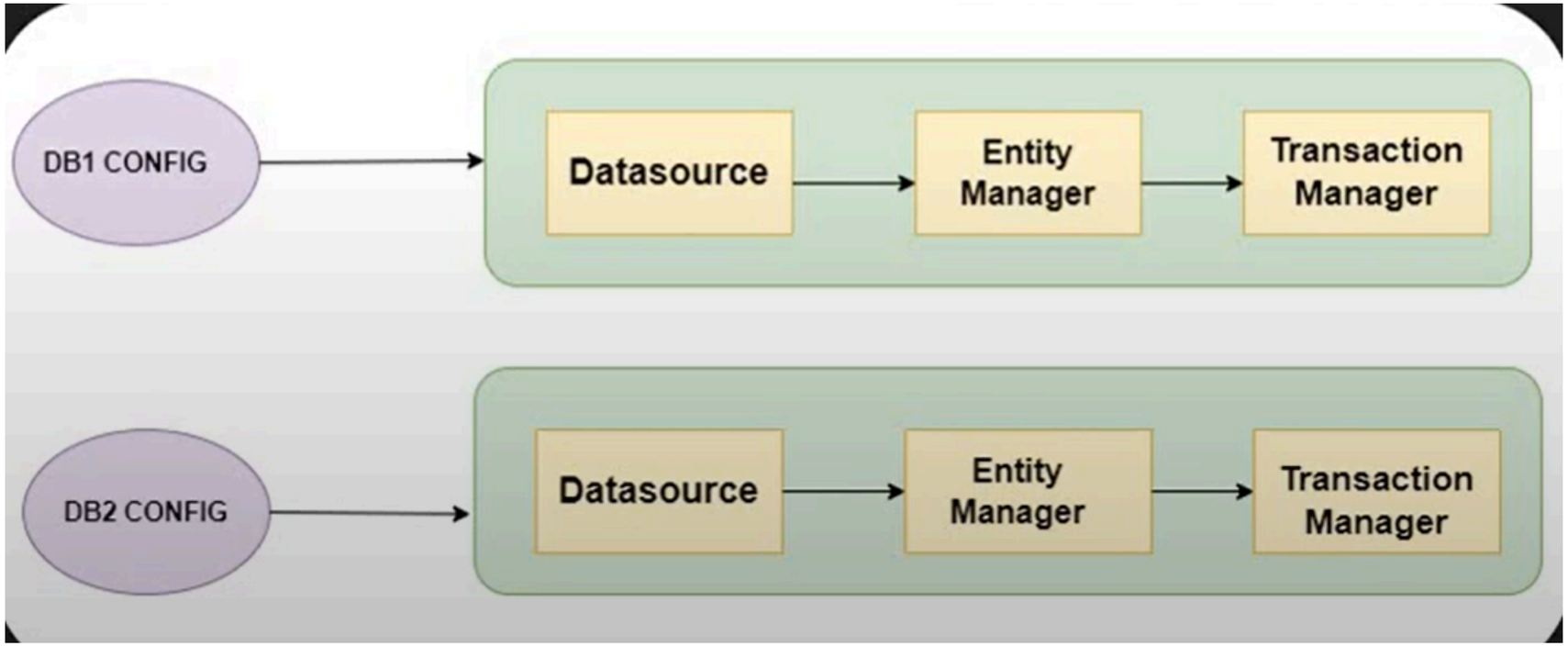
Step by step to do:

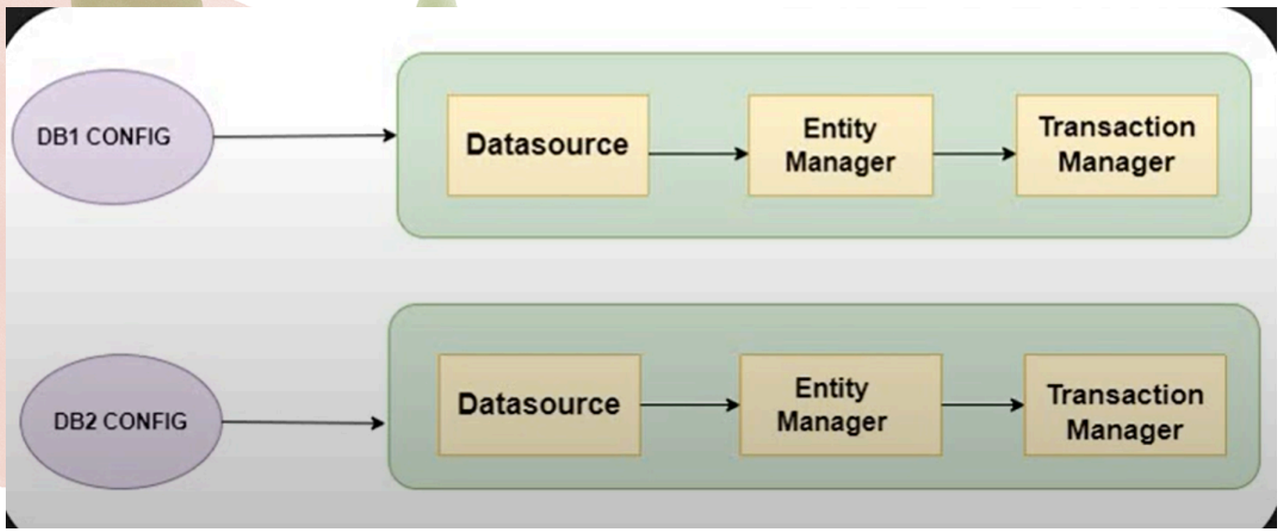
- Create 2 database and data sources
- Create 2 packages for different entities and repository
- Create config files for each datasources
- Execute and test



Configure and use multiple data sources in Spring Boot

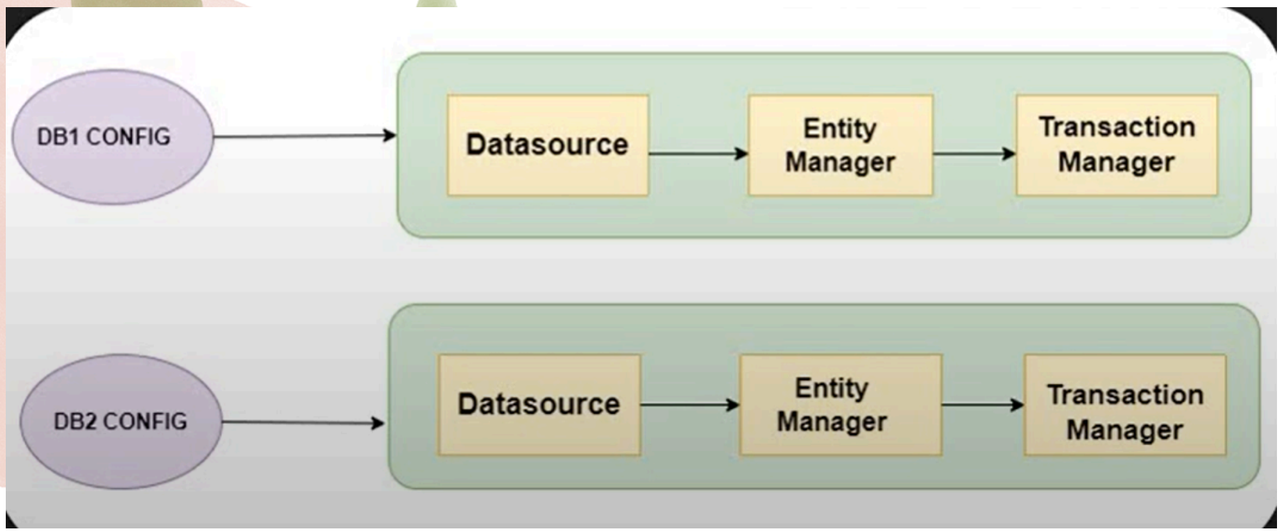






- DataSource is an interface in Java for managing connections to databases. It is often configured through properties such as database URL, username, password...

- EntityManager is an interface in JPA to manage entities and perform operations to the database. Each **EntityManager** is associated with a specific **DataSource** and is used to interact with the database via JPA.



- TransactionManager: It is an interface in Spring Framework to manage transactions in the application. Each **TransactionManager** is also associated with a specific **DataSource.** It ensures that transactions are executed successfully or rolled back correctly.

First, we have 2 entity that belongs to 2 different datasources

```java
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long accountId;
    private String name;
    private String website;
}
```

This entity belongs to accountdb database

```java
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class UserDB {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long userId;
    private String name;
    private String email;
    private int age;
}
```

This entity belongs to userdb database

Configure and use multiple data sources in Spring Boot

We config in file .properties to use 2 data sources in MySQL

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

#account datasource
spring.accountdb.datasource.jdbc-url=jdbc:mysql://localhost:3306/accountdb?createDatabaseIfNotExist=true
spring.accountdb.datasource.username=root
spring.accountdb.datasource.password=trungkien

# userdb datasource
spring.userdb.datasource.jdbc-url=jdbc:mysql://localhost:3306/userdb?createDatabaseIfNotExist=true
spring.userdb.datasource.username=root
spring.userdb.datasource.password=trungkien
```

Then we next to config the config file for each database

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
        entityManagerFactoryRef = "accountEntityManagerFactory",
        transactionManagerRef = "accountTransactionManager",
        basePackages = {"org.kiennguyenfpt.springbootpersistence.accountdb.repository"}
)
```

Set up configurations for **accountdb**

@EnableTransactionManagement: used to enable transaction management in Spring applications

@EnableJpaRepositories: used to enable the use of JPA repositories. It provides the necessary information for Spring to create implementations of JPA-based repositories, specifically:

+ **entityManagerFactoryRef**: Reference to the EntityManagerFactory bean, used to create the EntityManager. In this case, **accountEntityManagerFactory** is used for accountdb repositories.

+ **transactionManagerRef**: Reference to the TransactionManager bean, used to manage transactions.

+ **basePackages**: Specify the package containing the JPA repositories. Spring will search for repositories in these packages for management.
//without basePackeages....

Configure and use multiple data sources in Spring Boot

```
@Bean(name="accountDatasource")
@Primary
@ConfigurationProperties(prefix = "spring.accountdb.datasource")
public DataSource accountDataSource() {
    return DataSourceBuilder.create().build();
}
```

We creates a Spring-managed DataSource bean, named "accountDatasource", using configurations from the configuration file prefixed with "spring.accountdb.datasource"

```
return DataSourceBuilder.create().build();
```

Use DataSourceBuilder to create a new DataSource, then use build() method to construct and return DataSource. The DataSource properties will be configured based on the values in the configuration file specified by the "spring.accountdb.datasource" prefix.

Configure and use multiple data sources in Spring Boot

```
@Primary
@Bean(name="accountEntityManagerFactory")
public LocalContainerEntityManagerFactoryBean accountEntityManagerFactory(
        EntityManagerFactoryBuilder builder, @Qualifier("accountDatasource")DataSource dataSource) {
    return builder
            .dataSource(dataSource)
            .packages( ...packagesToScan: "org.kiennguyenfpt.springbootpersistence.accountdb.model")
            .build();
}
```

- We create an EntityManagerFactory bean to manage the EntityManagers for the accountdb

- EntityManagerFactoryBuilder: This object contains the configuration and information needed to create the EntityManagerFactory.

- To sum up, we creates an EntityManagerFactory bean with name "accountEntityManagerFactory", which uses a DataSource bean with name "accountDatasource" to create an EntityManager for the accountdb .
//change model to “entity”

Configure and use multiple data sources in Spring Boot

```
@Bean(name="accountTransactionManager")
public PlatformTransactionManager accountTransactionManager(
        @Qualifier("accountEntityManagerFactory") EntityManagerFactory entityManagerFactory) {
    return new JpaTransactionManager(entityManagerFactory);
}
```

We create a PlatformTransactionManager bean, named "accountTransactionManager", and use the previously created EntityManagerFactory to configure the JpaTransactionManager to manage transactions for the application's database

Here is full configuration for accountdb database

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
        entityManagerFactoryRef = "accountEntityManagerFactory",
        transactionManagerRef = "accountTransactionManager",
        basePackages = {"org.kiennguyenfpt.springbootpersistence.accountdb.repository"}
)
```

```
public class AccountDataSourceConfig {
    new *
    @Bean(name="accountDatasource")
    @Primary
    @ConfigurationProperties(prefix = "spring.accountdb.datasource")
    public DataSource accountDataSource() {
        return DataSourceBuilder.create().build();
    }
    new *
    @Primary
    @Bean(name="accountEntityManagerFactory")
    public LocalContainerEntityManagerFactoryBean accountEntityManagerFactory(
            EntityManagerFactoryBuilder builder, @Qualifier("accountDatasource")DataSource dataSource) {
        return builder
                .dataSource(dataSource)
                .packages( ...packagesToScan: "org.kiennguyenfpt.springbootpersistence.accountdb.model")
                .build();
    }
    new *
    @Bean(name="accountTransactionManager")
    public PlatformTransactionManager accountTransactionManager(
            @Qualifier("accountEntityManagerFactory") EntityManagerFactory entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}
```

Similar for configuration of userdb database

```java
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
        entityManagerFactoryRef = "userEntityManagerFactory",
        transactionManagerRef = "userTransactionManager",
        basePackages = {"org.kiennguyenfpt.springbootpersistence.userdb.repository"}
)

public class UserDBDataSourceConfig {
    new *
    @Bean(name="userDatasource")
    @ConfigurationProperties(prefix = "spring.userdb.datasource")
    public DataSource userDataSource() { return DataSourceBuilder.create().build(); }
    new *
    @Bean(name="userEntityManagerFactory")
    public LocalContainerEntityManagerFactoryBean userEntityManagerFactory(
            EntityManagerFactoryBuilder builder, @Qualifier("userDatasource")DataSource dataSource) {
        return builder
                .dataSource(dataSource)
                .packages( ...packagesToScan: "org.kiennguyenfpt.springbootpersistence.userdb.model")
                .build();
    }
    new *
    @Bean(name="userTransactionManager")
    public PlatformTransactionManager userTransactionManager(
            @Qualifier("userEntityManagerFactory") EntityManagerFactory entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
```

- The **userdb** configuration setup is also identical to the **accountdb** configuration.
- Datasource properties, packages and names must be changed.

Execute and test API:

```java
@PostConstruct
private void generateAccounts() {
    for (int i = 1; i <= 50; i++) {
        Account account = new Account();
        account.setName("Account " + i);
        account.setWebsite("www.example" + i + ".com");
        rep.save(account);
    }
}
```

We generate data for accountdb

```java
@PostConstruct
void init() {
    List<UserDB> users = new ArrayList<>();
    for (int i = 1; i <= 10; i++) {
        UserDB user = new UserDB();
        user.setName("User " + i);
        user.setEmail("user" + i + "@example.com");
        user.setAge(20 + i);
        users.add(user);
    }
    rep.saveAll(users);
}
```

We generate data for userdb

Execute and test API:

GET http://localhost:3624/accounts

Params Authorization Headers (6) Body Pre-request Scrip

Query Params

Key Valu

ody Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "accountId": 1,
        "name": "Account 1",
        "website": "www.example1.com"
    },
    {
        "accountId": 2,
        "name": "Account 2",
        "website": "www.example2.com"
    },
    {
        "accountId": 3,
        "name": "Account 3",
        "website": "www.example3.com"
    },
    {
        "accountId": 4,
        "name": "Account 4",
        "website": "www.example4.com"
    },
    {
        "accountId": 5,
        "name": "Account 5",
```


GET http://localhost:3624/accounts

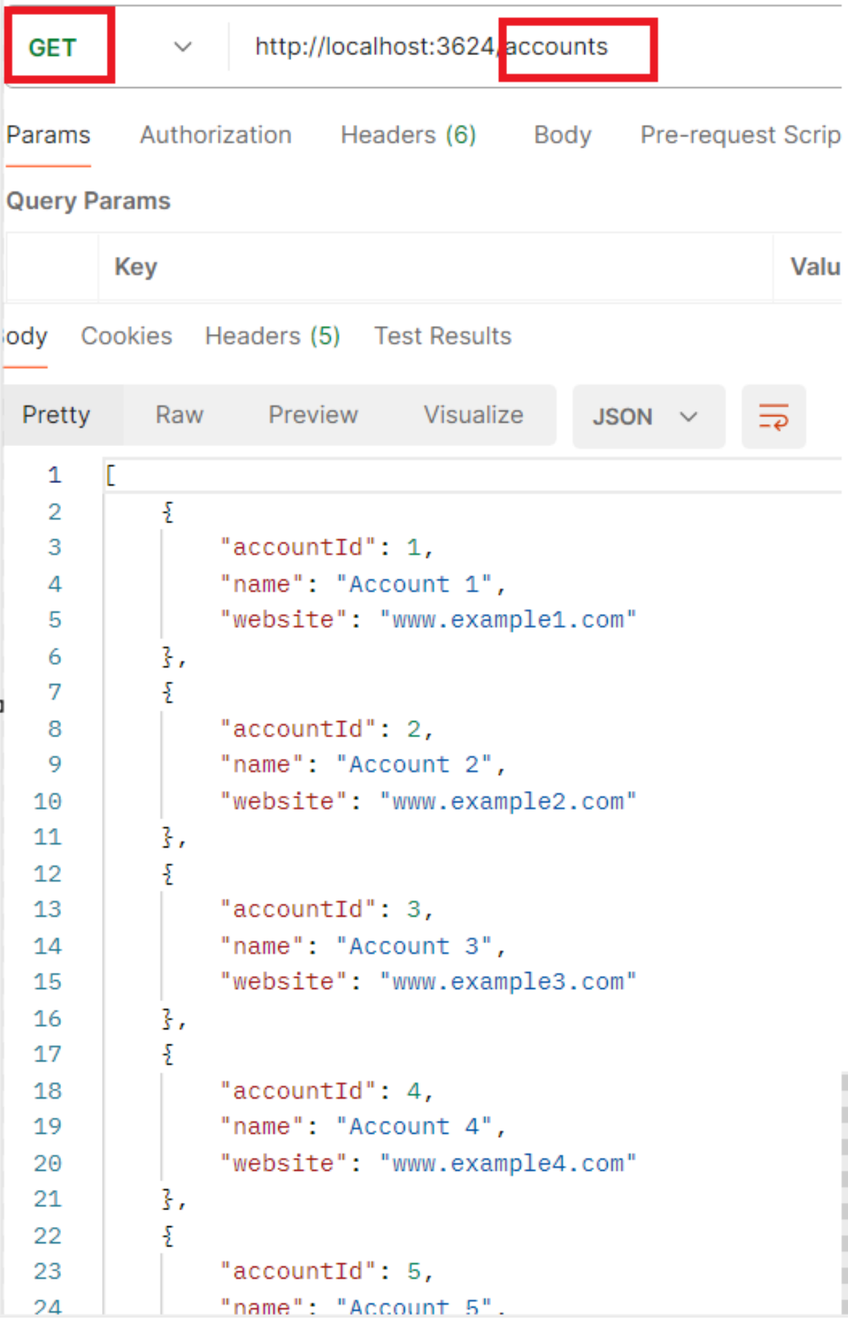
Params Authorization Headers (6) Body Pre

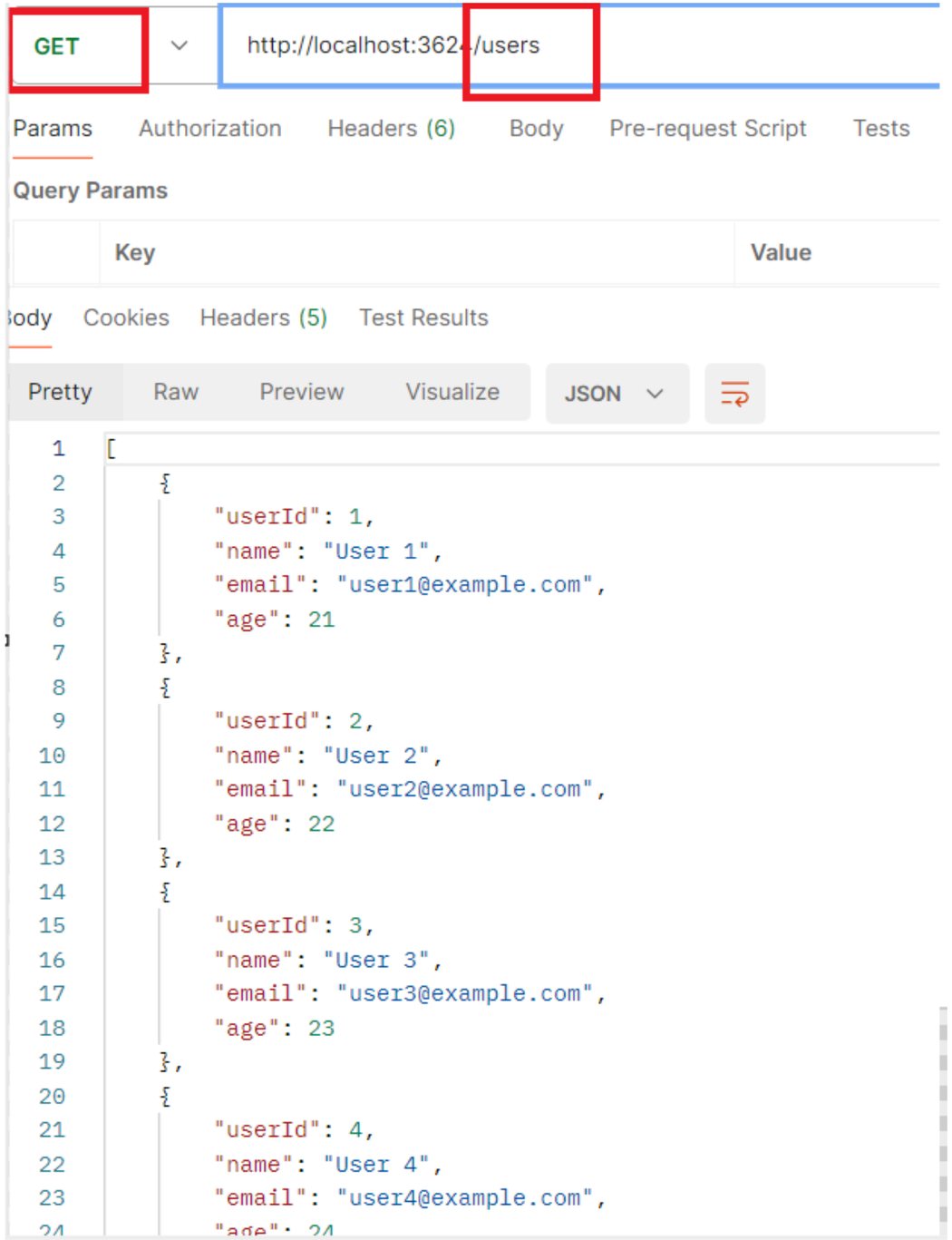
Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
    {
        "accountId": 45,
        "name": "Account 45",
        "website": "www.example45.com"
    },
    {
        "accountId": 46,
        "name": "Account 46",
        "website": "www.example46.com"
    },
    {
        "accountId": 47,
        "name": "Account 47",
        "website": "www.example47.com"
    },
    {
        "accountId": 48,
        "name": "Account 48",
        "website": "www.example48.com"
    },
    {
        "accountId": 49,
        "name": "Account 49",
        "website": "www.example49.com"
    },
    {
        "accountId": 50,
        "name": "Account 50",
        "website": "www.example50.com"
```

Execute and test API:





Execute and test API: //extremely failed

GET http://localhost:3624/users/3

Params Authorization Headers (6) Body Pre-re

Query Params

	Key
	Key

ody Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "userId": 3,
    "name": "User 3",
    "email": "user3@example.com",
    "age": 23
}
```


GET http://localhost:3624/accounts/8

Params Authorization Headers (6) Body Pre-request

Query Params

	Key
	Key

3ody Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "accountId": 8,
    "name": "Account 8",
    "website": "www.example8.com"
}
```

FPT SOFTWARE QUY NHON

Greatful Thank You for your listening!

KienNCT3 - Mentor HaiNV21