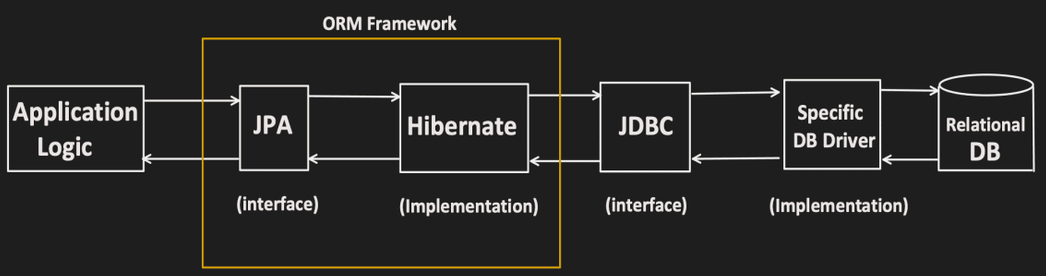


Before going to JPA, lets recall JDBC

JDBC (Java Database Connectivity) provides an Interface to:

- Make connection with DB
- Query DB
- and process the result

Actual implementation is provided by Specific DB Drivers.

For example:

MySQL
- Driver : Connector/J
- Class : *com.mysql.cj.jdbc.Driver*

PostgreSQL
- Driver : PostgreSQL JDBC Driver
- Class : *org.postgresql.Driver*

H2 (in-memory)
- Driver : H2 Database Engine
- Class : *org.h2.Driver*

Using JDBC without Springboot

```java
public class DatabaseConnection {

    public Connection getConnection() {
        try {
            // H2 Driver loading
            Class.forName( className: "org.h2.Driver");

            // Establish connection with DB
            return DriverManager.getConnection( url: "jdbc:h2:mem:userDB",  user: "sa",  password: "");

        }
        catch (ClassNotFoundException | SQLException e) {
            //handle  exception
        }

        return null;
    }
}
```

DB Name

```java
public class UserDAO {

    public void createUserTable() {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            Statement statementQuery = connection.createStatement();
            String sql = "CREATE TABLE users(user_id INT AUTO_INCREMENT PRIMARY KEY, user_name VARCHAR(100), age INT)";
            statementQuery.executeUpdate(sql);
        }
        catch (SQLException e) { /** handle exception **/ }
        finally { /** close statementQuery and db connection **/ }
    }

    public void createUser(String userName, int userAge) {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            String sqlQuery = "INSERT INTO users(user_name, age) VALUES (?, ?)";
            PreparedStatement preparedQuery = connection.prepareStatement(sqlQuery);
            preparedQuery.setString( parameterIndex: 1, userName);
            preparedQuery.setInt( parameterIndex: 2, userAge);
            preparedQuery.executeUpdate();
        }
        catch (SQLException e) { /** handle exception **/ }
        finally { /** close preparedQuery and db connection **/ }
    }

    public void readUsers() {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            String sqlQuery = "SELECT * FROM users";
            PreparedStatement preparedQuery = connection.prepareStatement(sqlQuery);
            ResultSet output = preparedQuery.executeQuery();
            while (output.next()) {
                String userDetails = output.getInt( columnLabel: "user_id") +
                        ":" + output.getString( columnLabel: "user_name") +
                        ":" + output.getInt( columnLabel: "age");
                System.out.println(userDetails);
            }
        }
        catch (SQLException e) { /** handle exception **/ }
        finally { /** close preparedQuery and db connection **/ }
    }
}
```

If we see above example:

- Connection
- Statement
- PreparedStatement
- ResultSet etc.

All are interfaces which JDBC provide and each specific driver provide the implementation for it.

But there are so much of BOILERCODE present like:

- Driver class loading
- DB Connection Making
- Exception Handling
- Closing of the DB connection and other objects like Statement etc.
- Manual handling of DB Connection Pool

  Etc..

Using JDBC with Springboot

pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Springboot provides *JdbcTemplate* class, which helps to remove all the boiler code.

```java
@Component
public class UserService {

    @Autowired
    UserRepository userRepository;

    public void createTable() {
        userRepository.createTable();
    }

    public void insertUser(String userName, int age) {
        userRepository.insertUser(userName, age);
    }

    public List<User> getUsers() {
        List<User> users = userRepository.getUsers();
        for(User user : users) {
            System.out.println(user.userId + ":" + user.getUserName() + ":" + user.getAge());
        }
        return users;
    }
}
```

```java
@Repository
public class UserRepository {

    @Autowired
    JdbcTemplate jdbcTemplate;

    public void createTable() {
        jdbcTemplate.execute( sql: "CREATE TABLE users (user_id INT AUTO_INCREMENT PRIMARY KEY, " +
                "user_name VARCHAR(100), age INT)");
    }

    public void insertUser(String name, int age) {
        String insertQuery = "INSERT INTO users (user_name, age) VALUES (?, ?)";
        jdbcTemplate.update(insertQuery, name, age);
    }

    public List<User> getUsers() {
        String selectQuery = "SELECT * FROM users";
        return jdbcTemplate.query(selectQuery, (rs, rowNum) -> {
            User user = new User();
            user.setUserId(rs.getInt( columnLabel: "user_id"));
            user.setUserName(rs.getString( columnLabel: "user_name"));
            user.setAge(rs.getInt( columnLabel: "age"));
            return user;
        });
    }
}
```

application.properties

```
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
```

```java
public class User {

    int userId;
    String userName;
    int age;

  //getters and setters
}
```

Driver class loading -
JdbcTemplate load it at the time of application startup in DriverManager class.

DB Connection Making -
jdbcTemplate takes care of it, whenever we execute any query.

Exception Handling -
in Plain JDBC, we get very abstracted '*SQLException*' but in jdbcTemplate, we get granular error like DuplicateKeyException, QueryTimeoutException etc.. (defined in org.springframework.dao package).

Closing of the DB connection and other resources -
when we invoke update or query method, after success or failure of the operation, jdbcTemplate takes care of either closing or return the connection to Pool itself.

- Manual handling of DB Connection Pool -
Springboot provides default jdbc connection pool i.e. *'HikariCP'* with Min and Max pool size of 10.
And we can change the configuration in *'application.properties'*

```
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
```

We can also configure different jdbc connection pool if we want like below:

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }
}
```

JdbcTemplate frequently used methods

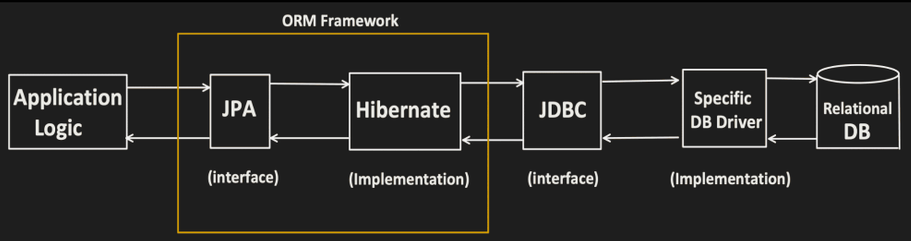
Method Name	Use For	Sample

update(String sql, Object... args)	Insert Update Delete	String insertQuery = "INSERT INTO users (user_name, age) VALUES (?, ?)";
		int rowsAffected = jdbcTemplate.update(insertQuery , "X", 27);
		
		String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?";
		int rowsAffected = jdbcTemplate.update(updateQuery, 29, 1);

update(String sql, PreparedStatementSetter pss)	Insert Update Delete	String insertQuery= "INSERT INTO users (user_name, age) VALUES (?, ?)";
		jdbcTemplate.update(insertQuery, (PreparedStatement ps) -> {
		ps.setString(1, "X");
		ps.setInt(2, 25); });
		
		String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?";
		jdbcTemplate.update(updateQuery, (PreparedStatement ps) -> {
		ps.setString(1, 29);
		ps.setInt(2, 1);
		});

query(String sql, RowMapper<T> rowMapper)	Get multiple Rows	List<User> users = jdbcTemplate.query("SELECT * FROM users", (rs, rowNum) -> {
		User user = new User();
		user.setUserId(rs.getInt("user_id"));
		user.setUserName(rs.getString("user_name"));
		user.setAge(rs.getInt("age"));
		return user; });

queryForList(String sql, Class<T> elementType)	Get Single Column of Multiple Rows	List<String> userNames = jdbcTemplate.queryForList("SELECT user_name FROM users", String.class);
queryForObject(String sql, Object[] args, Class<T> requiredType)	Get single Row	User user = jdbcTemplate.queryForObject("SELECT * FROM users WHERE user_id = ?", new Object[]{1}, User.class);
queryForObject(String sql, Class<T> requiredType)	Get Single Value	int userCount = jdbcTemplate.queryForObject("SELECT COUNT(*) FROM users", Integer.class);



ORM (Object-Relational Mapping)

- Act as a bridge between Java Object and Database tables.
- Unlike JDBC, where we have to work with SQL, with this, we can interact with database using Java Objects.

Lets first see, 1 happy flow first, before deep diving into JPA

pom.xml

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

application.properties

```
#database connection properties
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @GetMapping(path = "/test-jpa")
    public List<UserDetails> getUser() {
        UserDetails userDetails = new UserDetails( name: "xyx",
                email: "xyz@conceptandcoding.com");
        userDetailsService.saveUser(userDetails);
        return userDetailsService.getAllUsers();
    }
}
```

```
@Service
public class UserDetailsService {

    @Autowired
    private UserDetailsRepository userDetailsRepository;

    public void saveUser(UserDetails user) {
        userDetailsRepository.save(user);
    }

    public List<UserDetails> getAllUsers() {
        return userDetailsRepository.findAll();
    }
}
```

```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {
}
```

```
@Entity
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters

}
```

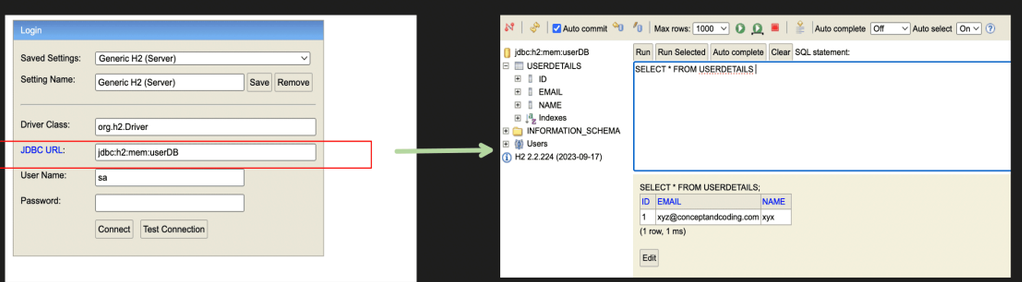
Output:


```
[
    {
        "id": 1,
        "name": "xyx",
        "email": "xyz@conceptandcoding.com",
        "age": 28
    }
]
```


To enable the console, we can add, below two properties in "application.properties"

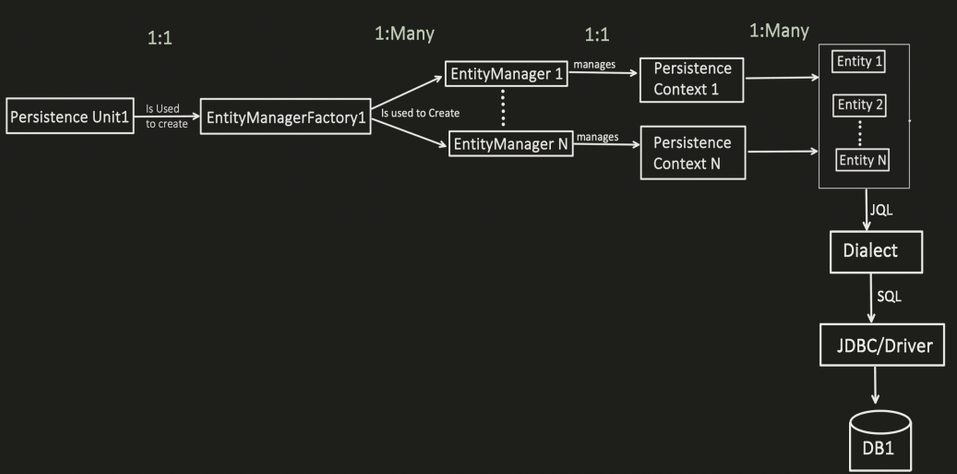
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

http://localhost:8080/h2-console



Looks very simple, but what's happening inside?

JPA Architecture/Components involved:



pom.xml

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

1. Persistence Unit

- Logical grouping of Entity classes which share same configurations.
- Configuration details like:
  - Database connection properties
  - JPA Provider (hibernate etc.) etc.

Persistence.xml

```xml
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">

    <persistence-unit name="persistentUnitNameHere" transaction-type="RESOURCE_LOCAL">

        <!-- Logical grouping of Entity Classes which all shares same configurations -->
        <class>com.conceptandcoding.entity.SampleEntity1</class>
        <class>com.conceptandcoding.entity.SampleEntity2</class>

        <!-- which JPA provider we want to use, hibernate or OpenJPA or EclipseLink etc... -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>


        <!-- DB Connection Properties -->
        <properties>

            <!-- specific provider dialect -->
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect" />

            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:userDB" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <property name="javax.persistence.jdbc.password" value="" />
            <!-- more properties here -->
        </properties>

    </persistence-unit>

</persistence>
```

application.properties

```
#database connection properties
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Define packages to scan
# (its optional, spring boot looks for all @Entity, but still if we want specific package lookup
spring.jpa.packages-to-scan=com.conceptandcoding.entity

#define provider
#(its optional, spring boot automatically picks based on provider)
spring.jpa.properties.javax.persistence.provider=org.hibernate.jpa.HibernatePersistenceProvider
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

#transaction type, Default is RESOURCE_LOCAL only
spring.jpa.properties.javax.persistence.transactionType=RESOURCE_LOCAL
```

2. EntityManagerFactory

- Using Persistence Unit configuration, EntityManagerFactory object get created during application startup.
- If any property is not provided, default one is picked and set.
- 1 EntityManagerFactory for 1 Persistence Unit.
- This class act as a Factory to create an object of EntityManager.

LocalContainerEntityManagerFactoryBean.java

```
@Override
protected EntityManagerFactory createNativeEntityManagerFactory() throws PersistenceException {
    Assert.state( expression: this.persistenceUnitInfo != null,  message: "PersistenceUnitInfo not initialized");

    PersistenceProvider provider = getPersistenceProvider();
    if (provider == null) {
        String providerClassName = this.persistenceUnitInfo.getPersistenceProviderClassName();
        if (providerClassName == null) {
            throw new IllegalArgumentException(
                    "No PersistenceProvider specified in EntityManagerFactory configuration, " +
                            "and chosen PersistenceUnitInfo does not specify a provider class name either");
        }
        Class<?> providerClass = ClassUtils.resolveClassName(providerClassName, getBeanClassLoader());
        provider = (PersistenceProvider) BeanUtils.instantiateClass(providerClass);
    }

    if (logger.isDebugEnabled()) {
        logger.debug("Building JPA container EntityManagerFactory for persistence unit '" +
                this.persistenceUnitInfo.getPersistenceUnitName() + "'");
    }
    EntityManagerFactory emf =
            provider.createContainerEntityManagerFactory(this.persistenceUnitInfo, getJpaPropertyMap());
    postProcessEntityManagerFactory(emf, this.persistenceUnitInfo);

    return emf;
}
```

What if we want to manually create and object of EntityManagerFactory?

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        adapter.setGenerateDdl(true);
        adapter.setDatabasePlatform("org.hibernate.dialect.H2Dialect");
        return adapter;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(
            DataSource dataSource,
            JpaVendorAdapter jpaVendorAdapter) {
        LocalContainerEntityManagerFactoryBean emf1 = new LocalContainerEntityManagerFactoryBean();
        emf1.setDataSource(dataSource);
        emf1.setJpaVendorAdapter(jpaVendorAdapter);
        emf1.setPackagesToScan("com.conceptandcoding.learningspringboot.jpa");
        emf1.setPersistenceUnitName("uniqueFactoryName"); //  unique name for our EntityManagerFactory
        return emf1;
    }
}
```

3. Transaction Manager association with EntityManagerFactory

During persistence unit, we have specified the value for "transaction-type" value either:

- RESOURCE_LOCAL (default)
- JTA (Java Transaction API)

Transaction Manager could be of 2 type:

- Manager, which managing transaction For 1 DB.

- Manager, which managing transaction which can span across multiple DB. That's also possible using JTA.

During application startup, after EntityManagerFactory object created, based on RESOURCE_LOCAL or JTA, transaction manager object get created.

UseCase1: Transaction Manager for managing txn for 1 DB

application.properties

```
#database connection properties
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Define packages to scan
# (its optional, spring boot looks for all @Entity, but still if we want specific package lookup only)
spring.jpa.packages-to-scan=com.conceptandcoding.entity

#define provider
#(its optional, spring boot automatically picks based on provider)
spring.jpa.properties.javax.persistence.provider=org.hibernate.jpa.HibernatePersistenceProvider
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

#transaction type, Default is RESOURCE_LOCAL only
spring.jpa.properties.javax.persistence.transactionType=RESOURCE_LOCAL
```

JpaTransactionManager.java
(implementation of PlatformTransactionManager)

```
@Override
public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
    if (getEntityManagerFactory() == null) {
        if (!(beanFactory instanceof ListableBeanFactory lbf)) {
            throw new IllegalStateException("Cannot retrieve EntityManagerFactory by persistence unit name " +
                    "in a non-listable BeanFactory: " + beanFactory);
        }
        setEntityManagerFactory(EntityManagerFactoryUtils.findEntityManagerFactory(lbf, getPersistenceUnitName()));
    }
}
```

```
beanType = {Class@10098} "class org.springframework.orm.jpa.JpaTransactionManager" ... Navigate
  cachedConstructor = null
  name = "org.springframework.orm.jpa.JpaTransactionManager"
  module = {Module@13639} "unnamed module @71623278"
  classLoader = {ClassLoaders$AppClassLoader@10105}
  classData = null
  packageName = "org.springframework.orm.jpa"
  componentType = null
```

We can create it manually too:

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        adapter.setGenerateDdl(true);
        adapter.setDatabasePlatform("org.hibernate.dialect.H2Dialect");
        return adapter;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(
            DataSource dataSource,
            JpaVendorAdapter jpaVendorAdapter) {
        LocalContainerEntityManagerFactoryBean emf1 = new LocalContainerEntityManagerFactoryBean();
        emf1.setDataSource(dataSource);
        emf1.setJpaVendorAdapter(jpaVendorAdapter);
        emf1.setPackagesToScan("com.conceptandcoding.learningspringboot.jpa");
        emf1.setPersistenceUnitName("uniqueFactoryName"); //  unique name for our EntityManagerFactory
        return emf1;
    }

    @Bean
    public JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }
}
```

UseCase2: Transaction Manager for creating txn which can span across multiple DB

I will create a separate video, in which I will explain, how we can handle txn which can span across multiple database.
As it required some time for explaining JTA related nuisance and keyworks like Atomikos, XADataSource and how it uses 2PC to orchestrate txn etc.. which is out of the context for todays topic.

But at this point, we can understand that, TRANSACTION MANAGER is created based on what "transaction-type" we provided in the persistence unit (application.properties) file.

---------- Above all steps happened during application startup, below steps happen when API gets invoked ----------

4. EntityManager and Persistence Context

EntityManager :

- Its an interface in JPA that provides methods to perform CRUD operations on entities.
  - **persist()** (for saving)
  - **merge()** (for updating)
  - **find()** (for fetching)
  - **remove()** (for deleting)
  - **createQuery()** (for executing JPQL queries)

- EntityManager interface methods are implemented by JPA Vendors like Hibernate etc.
- EntityManagerFactory helps to create an Object of EntityManager.

PersistenceContext:

- Consider its a first level cache.
- For each EntityManager, PersistenceContext object is created, which hold list of Entities its working on.
- Also manage the life cycle of entity.

Entity Manager Insert/Update/Delete operations are Transaction bounded. Means, it first check if 'Transaction' is open, if not, it will throw exception.
But not all (READ operations are not Transaction bounded)

```java
@Service
public class UserDetailsService {

    @Autowired
    private UserDetailsRepository userDetailsRepository;

    public void saveUser(UserDetails user) {
        userDetailsRepository.save(user);
    }

    public List<UserDetails> getAllUsers() {
        return userDetailsRepository.findAll();
    }
}
```

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {
}
```

Internally JpaRepository all Insert, Update, Delete methods are annotated with @Transactional, so even if we do not write, spring framework takes care of it.

```java
@Transactional
public <S extends T> S save(S entity) {
    Assert.notNull(entity, message: "Entity must not be null");
    if (this.entityInformation.isNew(entity)) {
        this.entityManager.persist(entity);
        return entity;
    } else {
        return this.entityManager.merge(entity);
    }
}
```

What if, I try to directly call EntityManager persist method, instead of spring framework.

```java
@Service
public class UserDetailsService {

    @PersistenceContext
    EntityManager entityManager;

    public void saveUser(UserDetails user) {
        entityManager.persist(user);
    }
}
```

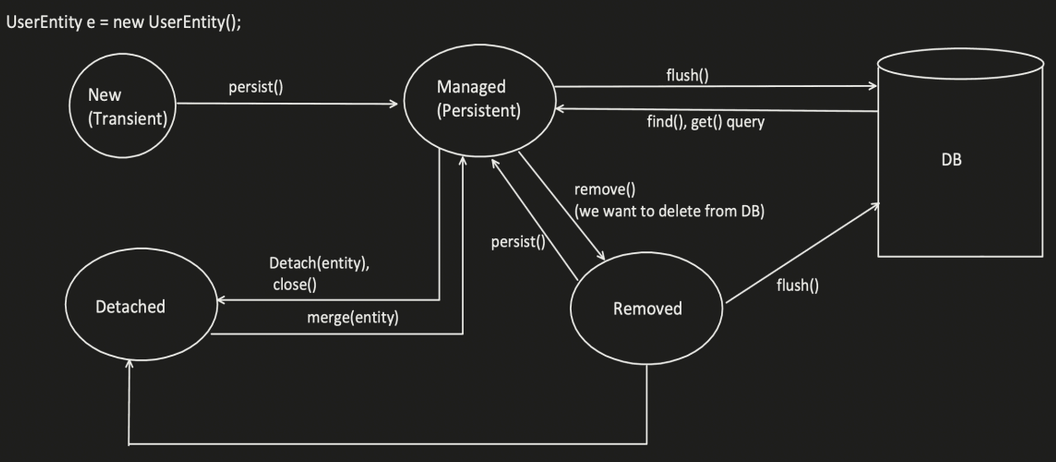
```
jakarta.persistence.TransactionRequiredException Create breakpoint : No EntityManager with actual transaction available for current thread
    at org.springframework.orm.jpa.SharedEntityManagerCreator$SharedEntityManagerInvocationHandler.invoke(SharedEntityManagerCreator.ja
    at jdk.proxy2/jdk.proxy2.$Proxy108.persist(Unknown Source) ~[na:na]
    at com.conceptandcoding.learningspringboot.jpa.service.UserDetailsService.saveUser(UserDetailsService.java:23) ~[classes/:na]
    at com.conceptandcoding.learningspringboot.UserController.getUser(UserController.java:22) ~[classes/:na] <4 internal lines>
```

If I add @Transactional, it works fine now.

```java
@Service
public class UserDetailsService {

    @PersistenceContext
    EntityManager entityManager;

    @Transactional
    public void saveUser(UserDetails user) {
        entityManager.persist(user);
    }
}
```

```json
[
    {
        "id": 1,
        "name": "xyx",
        "email": "xyz@conceptandcoding.com"
    }
]
```

Life cycle of Entity in persitenceContext:



JPA - PART3 (First-Level Cache)

application.properties

```
#database connection
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=


# Enable the H2 console and set the path
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

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

```
@Entity
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}
```

Controller class

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @GetMapping(path = "/test-jpa")
    public UserDetails getUser() {
        UserDetails userDetails = new UserDetails( name: "xyx",
                email: "xyz@conceptandcoding.com");
        userDetailsService.saveUser(userDetails);
        UserDetails output1 =  userDetailsService.getUser( primaryKey: 1L);
        return output1;
    }

    @GetMapping(path = "/read-jpa")
    public UserDetails getUser2() {
        UserDetails output1 =  userDetailsService.getUser( primaryKey: 1L);
        return output1;
    }
}
```

```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {
}
```

- Internally using EntityManager API only, but it provides some predefined methods like findAll, deleteAll etc.
- Also provides pagination and sorting capability...
- Also @Transactional support on operations which needs it on operation like Insert, modify, delete.
- Start and close EntityManager resource, so no need to manage it.

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public void saveUser(UserDetails user) {
        userDetailsRepository.save(user);
    }

    public UserDetails getUser(Long primaryKey) {
        return userDetailsRepository.findById(primaryKey).get();
    }
}
```

Creating EntityManager

SimpleJpaRepository.java

```
@Transactional
public <S extends T> S save(S entity) {
    Assert.notNull(entity,  message: "Entity must not be null");
    if (this.entityInformation.isNew(entity)) {
        this.entityManager.persist(entity);
        return entity;
    } else {
        return this.entityManager.merge(entity);
    }
}
```

'SessionImpl' is the Hibernate implementation for EntityManager APIs, creates PersistenceContext f or each EntityManager and is doing first-level caching in the map

StatefulPersistenceContext.java

```
@Override
public EntityHolder claimEntityHolderIfPossible(
        EntityKey key,
        Object entity,
        JdbcValuesSourceProcessingState processingState,
        EntityInitializer initializer) {
    EntityHolderImpl holder = EntityHolderImpl.forEntity(key, key.getPersister(), entity);
    final EntityHolderImpl oldHolder = getOrInitializeEntitiesByKey().putIfAbsent(
            key,
            holder
    );

    if (oldHolder != null) {
        if (entity != null) {
            assert oldHolder.entity == null || oldHolder.entity == entity;
            oldHolder.entity = entity;
        }
        // Skip setting a new entity initializer if there already is one owner
        // Also skip if an entity exists which is different from the effective optional object.
        // The effective optional object is the current object to be refreshed,
        // which always needs re-initialization, even if already initialized
        if (oldHolder.entityInitializer != null
                || oldHolder.entity != null && oldHolder.state != EntityHolderState.ENHANCED_PROXY && (
                processingState.getProcessingOptions().getEffectiveOptionalObject() == null
                        || oldHolder.entity != processingState.getProcessingOptions().getEffectiveOptionalObject())
        ) {
            return oldHolder;
        }
        holder = oldHolder;
    }
    assert holder.entityInitializer == null || holder.entityInitializer == initializer;
    holder.entityInitializer = initializer;
    processingState.registerLoadingEntityHolder(holder);
    return holder;
}
```

> key = {EntityKey@11006} "EntityKey[com.conceptandcoding.learningspringboot.jpa.entity.UserDetails#1]"
> entity = null

StatefulPersistenceContext.java

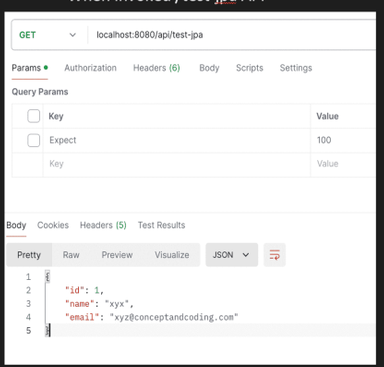
```
@Override
public void addEntity(EntityKey key, Object entity) {
    EntityHolderImpl holder = EntityHolderImpl.forEntity(key, key.getPersister(), entity);
    final EntityHolderImpl oldHolder = getOrInitializeEntitiesByKey().putIfAbsent(
            key,
            holder
    );
    if (oldHolder != null) {
        assert oldHolder.entity == null || oldHolder.entity == entity;
        oldHolder.entity = entity;
        holder = oldHolder;
    }
    holder.state = EntityHolderState.INITIALIZED;
    final BatchFetchQueue fetchQueue = this.batchFetchQueue;
    if (fetchQueue != null) {
        fetchQueue.removeBatchLoadableEntityKey(key);
    }
}
```

> key = {EntityKey@10767} "EntityKey[com.conceptandcoding.learningspringboot.jpa.entity.UserDetails#1]"
> entity = {UserDetails@10768}
> id = {Long@14243} 1
> name = "xyx"
> email = "xyz@conceptandcoding.com"

During application startup (JPA creates fresh DB and table):

```
2024-11-11T15:00:46.273+05:30  INFO 9537 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' t
Hibernate:
    drop table if exists user_details cascade
Hibernate:
    create table user_details (
        id bigint generated by default as identity,
        email varchar(255),
        name varchar(255),
        primary key (id)
    )
2024-11-11T15:00:46.285+05:30  INFO 9537 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-11-11T15:00:46.374+05:30  WARN 9537 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may b
2024-11-11T15:00:46.555+05:30  INFO 9537 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2024-11-11T15:00:46.560+05:30  INFO 9537 --- [           main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.542 seconds (process running for 1.7)
```

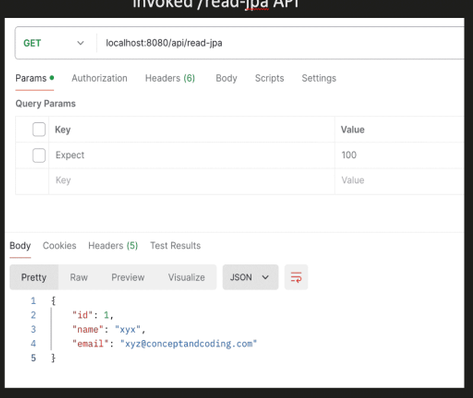
When invoked /test-jpa API




```
Hibernate:
    drop table if exists user_details cascade
Hibernate:
    create table user_details (
        id bigint generated by default as identity,
        email varchar(255),
        name varchar(255),
        primary key (id)
    )
2024-11-11T15:02:28.409+05:30  INFO 9564 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit '
2024-11-11T15:02:28.490+05:30  WARN 9564 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, d
2024-11-11T15:02:28.647+05:30  INFO 9564 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2024-11-11T15:02:28.653+05:30  INFO 9564 --- [           main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.404 seconds (process runn
2024-11-11T15:02:32.097+05:30  INFO 9564 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-11-11T15:02:32.097+05:30  INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2024-11-11T15:02:32.098+05:30  INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
```

Only insert call, even though we are doing first insert and then read

invoked /read-jpa API



```
2024-11-11T15:02:28.409+05:30  INFO 9564 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit '
2024-11-11T15:02:28.490+05:30  WARN 9564 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, d
2024-11-11T15:02:28.647+05:30  INFO 9564 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2024-11-11T15:02:28.653+05:30  INFO 9564 --- [           main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.404 seconds (process run
2024-11-11T15:02:32.097+05:30  INFO 9564 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-11-11T15:02:32.097+05:30  INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2024-11-11T15:02:32.098+05:30  INFO 9564 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
Hibernate:
    select
        ud1_0.id,
        ud1_0.email,
        ud1_0.name
    from
        user_details ud1_0
    where
        ud1_0.id=?
```

So, PersistenceContext scope is associated with EntityManager:

```java
@Service
public class UserDetailsService {


    @Autowired
    EntityManagerFactory entityManagerFactory;

    public UserDetails saveUser(UserDetails user) {

        EntityManager entityManager = entityManagerFactory.createEntityManager(); //session1 created
        entityManager.getTransaction().begin(); //transaction created
        entityManager.persist(user);
        entityManager.find(UserDetails.class, primaryKey: 1L);
        UserDetails output =  entityManager.find(UserDetails.class, primaryKey: 1L);
        System.out.println("i am able to find the data, name is:" + output.getName());
        entityManager.getTransaction().commit(); //transaction committed
        entityManager.close(); // session1 closed


        EntityManager entityManager2 = entityManagerFactory.createEntityManager(); //session2 created
        entityManager2.getTransaction().begin(); //transaction created
        entityManager2.find(UserDetails.class, primaryKey: 1L);
        UserDetails output2 =  entityManager2.find(UserDetails.class, primaryKey: 1L);
        System.out.println("Session2: i am able to find the data, name is:" + output.getName());
        entityManager2.getTransaction().commit(); //transaction committed
        entityManager2.close(); // session1 closed

        return output2;
    }
}
```

Output:

```
2024-11-11T16:58:58.995+05:30  INFO 10877 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available
Hibernate:
    drop table if exists user_details cascade
Hibernate:
    create table user_details (
        id bigint generated by default as identity,
        email varchar(255),
        name varchar(255),
        primary key (id)
    )
2024-11-11T16:58:59.006+05:30  INFO 10877 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory
2024-11-11T16:58:59.025+05:30  WARN 10877 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by
2024-11-11T16:58:59.236+05:30  INFO 10877 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) wi
2024-11-11T16:58:59.241+05:30  INFO 10877 --- [           main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.3
2024-11-11T16:59:01.873+05:30  INFO 10877 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet
2024-11-11T16:59:01.873+05:30  INFO 10877 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServl
2024-11-11T16:59:01.873+05:30  INFO 10877 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
i am able to find the data, name is:xyx
Hibernate:
    select
        ud1_0.id,
        ud1_0.email,
        ud1_0.name
    from
        user_details ud1_0
    where
        ud1_0.id=?
Session2: i am able to find the data, name is:xyx
```

And EntityManager is created for each HTTP Request, so different methods within same HTTP Request share the EntityManager

DispatcherServelt Code snapshot:

```
if (!mappedHandler.applyPreHandle(processedRequest, response)) {
    return;
}

// Actually invoke the handler.
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

if (asyncManager.isConcurrentHandlingStarted()) {
    return;
}
```

```
@Override
public void preHandle(WebRequest request) throws DataAccessException {
    String key = getParticipateAttributeName();
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
    if (asyncManager.hasConcurrentResult() && applyEntityManagerBindingInterceptor(asyncManager, key)) {
        return;
    }

    EntityManagerFactory emf = obtainEntityManagerFactory();
    if (TransactionSynchronizationManager.hasResource(emf)) {
        // Do not modify the EntityManager: just mark the request accordingly.
        Integer count = (Integer) request.getAttribute(key, WebRequest.SCOPE_REQUEST);
        int newCount = (count != null ? count + 1 : 1);
        request.setAttribute(getParticipateAttributeName(), newCount, WebRequest.SCOPE_REQUEST);
    }
    else {
        logger.debug("Opening JPA EntityManager in OpenEntityManagerInViewInterceptor");
        try {
            EntityManager em = createEntityManager();
            EntityManagerHolder emHolder = new EntityManagerHolder(em);
            TransactionSynchronizationManager.bindResource(emf, emHolder);

            AsyncRequestInterceptor interceptor = new AsyncRequestInterceptor(emf, emHolder);
            asyncManager.registerCallableInterceptor(key, interceptor);
            asyncManager.registerDeferredResultInterceptor(key, interceptor);
        }
        catch (PersistenceException ex) {
            throw new DataAccessResourceFailureException("Could not create JPA EntityManager", ex);
        }
    }
}
```

```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @GetMapping(path = "/test-jpa")
    public UserDetails getUser() {
        UserDetails userDetails1 = new UserDetails( name: "xyx",
                email: "xyz@conceptandcoding.com");

        userDetailsService.saveUser(userDetails1);
        return userDetailsService.findByID( primaryKey: 1L);
    }

}
```

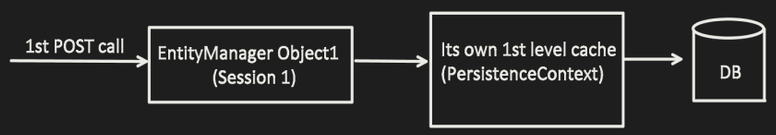
```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public void saveUser(UserDetails user) {

        userDetailsRepository.save(user);
    }

    public UserDetails findByID(Long primaryKey) {
        return  userDetailsRepository.findById(primaryKey).get();
    }
}
```

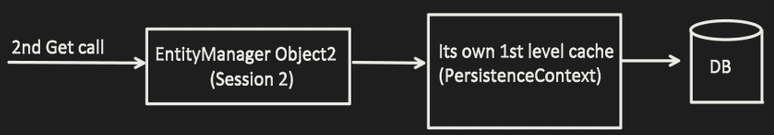
Internally both uses the same Entity Manager, therefore when findByID invoked, Cache Hit will happen

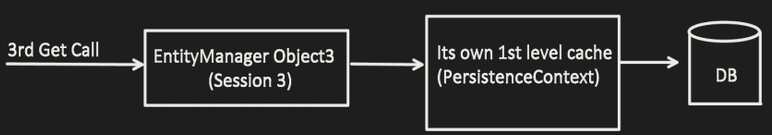
Output:

```
2024-11-11T18:25:19.099+05:30  INFO 11152 --- [          main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH
Hibernate:
    drop table if exists user_details cascade
Hibernate:
    create table user_details (
        id bigint generated by default as identity,
        email varchar(255),
        name varchar(255),
        primary key (id)
    )
2024-11-11T18:25:19.180+05:30  INFO 11152 --- [          main] j.LocalContainerEntityManagerFactoryBean : Init
2024-11-11T18:25:19.185+05:30  WARN 11152 --- [          main] JpaBaseConfiguration$JpaWebConfiguration : spri
2024-11-11T18:25:19.343+05:30  INFO 11152 --- [          main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomc
2024-11-11T18:25:19.347+05:30  INFO 11152 --- [          main] c.c.\.SpringbootApplication              : Star
2024-11-11T18:25:24.721+05:30  INFO 11152 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Init
2024-11-11T18:25:24.722+05:30  INFO 11152 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Init
2024-11-11T18:25:24.722+05:30  INFO 11152 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Comp
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
```

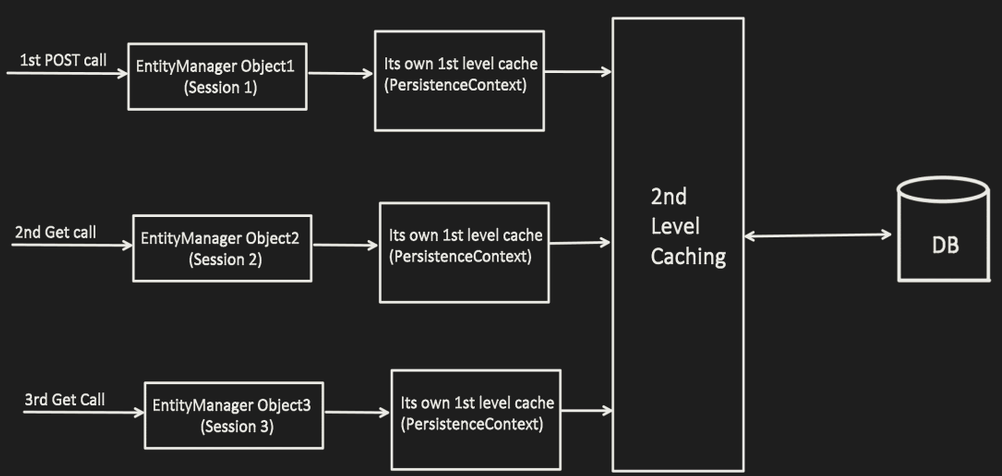
From previous video, First level caching, we already know that, for each HTTP REQUEST, different EntityManager Object (session) is created and its have its own Persistence context (1st level cache)







Now, in Second Level caching or L2 caching, We will achieve something like this:



Lets first see, one happy flow, and see what all it takes to enable the 2nd level caching

pom.xml

```xml
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.10.8</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jcache</artifactId>
    <version>6.5.2.Final</version>
</dependency>
<dependency>
    <groupId>javax.cache</groupId>
    <artifactId>cache-api</artifactId>
    <version>1.1.1</version>
</dependency>
```

application.properties

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory
spring.jpa.properties.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider
logging.level.org.hibernate.cache.spi=DEBUG
```

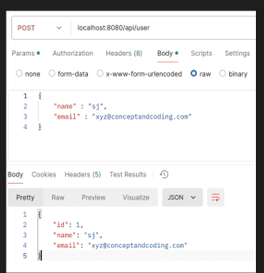
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails getUser2() {
        return userDetailsService.findByID( primaryKey: 1L);
    }
}
```

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {

        return userDetailsRepository.save(user);
    }

    public UserDetails findByID(Long primaryKey) {
        return  userDetailsRepository.findById(primaryKey).get();
    }
}
```

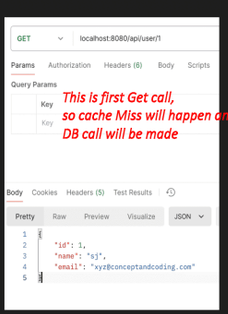
```java
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
        region = "userDetailsCache")
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}
```

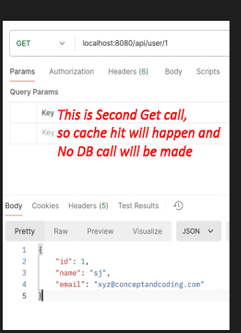
```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {
}
```

1. During Insert, data is directly inserted into DB, no Cache insertion or validation happens



2. Get:
   During Get, JPA will check, if data is present in cache? If Yes, its cache hit and return else its cache miss and it will fetch from DB and put into Cache.





```
2024-11-25T20:43:25.760+05:30  INFO 7872 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
2024-11-25T20:43:27.706+05:30 DEBUG 7872 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDe
2024-11-25T20:43:27.707+05:30 DEBUG 7872 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Cache miss : region = `userDetailsCache`,
Hibernate:
    select
        ud1_0.id,
        ud1_0.email,
        ud1_0.name
    from
        user_details ud1_0
    where
        ud1_0.id=?
2024-11-25T20:43:27.729+05:30 DEBUG 7872 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Caching data from load [region=`userDeta
2024-11-25T20:43:28.292+05:30 DEBUG 7872 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDe
2024-11-25T20:43:28.293+05:30 DEBUG 7872 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Checking readability of read-write cache
2024-11-25T20:43:28.295+05:30 DEBUG 7872 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Cache hit : region = `userDetailsCache`,
'
```

1. Why in pom.xml, 3 dependencies required?

```
<dependency>
<groupId>org.ehcache</groupId>
<artifactId>ehcache</artifactId>
<version>3.10.8</version>
</dependency>
```

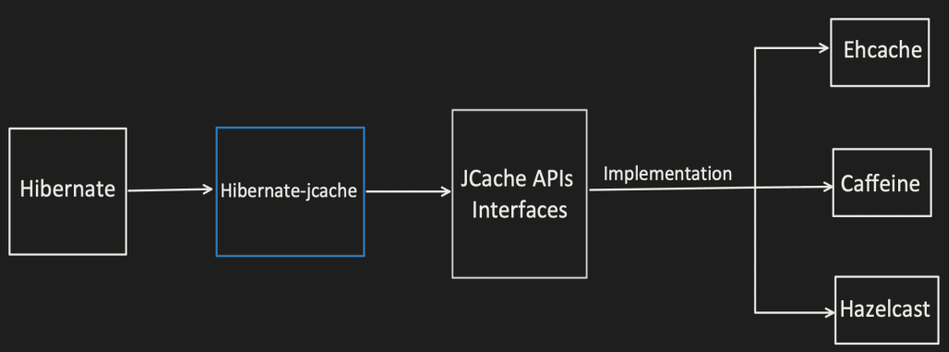
→ Provides the core implementation of Second level caching

```
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-jcache</artifactId>
<version>6.5.2.Final</version>
</dependency>
```

→ Hibernate specific Caching logic comes with this. Like we use Annotations over entity ***@Cache***, we used ***CacheConcurrencyStrategy***, so specific logic need to be executed, and this library help us with that.

```
<dependency>
<groupId>javax.cache</groupId>
<artifactId>cache-api</artifactId>
<version>1.1.1</version>
</dependency>
```

→ Provides the interface for Jcache, hibernate interact with these APIs.
Helps to achieve Loose coupling. We can change from Ehcache to some other Jcache compliant caching provider without changing code.



2. Lets understand application.properties and Region:

This tell hibernate to use hibernate-jcache class to manage caching, we can also provide here direct ehcache factory class, means bypassing Jcache interface

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory
spring.jpa.properties.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider
logging.level.org.hibernate.cache.spi=DEBUG
```

Region:

Helps in logical grouping of cached data.
For each Region (or say group), we can apply different caching strategy like

- Eviction policy
- TTL
- Cache size
- Concurrency strategy etc.

Which helps in achieving granular level management of cached data (either Entity, Collection or Query results)

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
        region = "userDetailsCache")
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}
```

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
        region = "orderDetailsCache")
public class OrderDetails {

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

    private String productName;
    private int quantity;
    private double price;

    // Getters and Setters
}
```

ehcache.xml

(file within "src/main/resources/" path)

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://www.ehcache.org/ehcache.xsd">

    <cache alias="userDetailsCache"
           maxElementsInMemory="100"
           timeToLiveSeconds="60"
           evictionStrategy="LIFO" />

    <cache alias="orderDetailsCache"
           maxElementsInMemory="1000"
           timeToLiveSeconds="200"
           evictionStrategy="FIFO" />
</ehcache>
```

3. Different *CacheConcurrencyStrategy*

```java
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
        region = "userDetailsCache")
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}
```

S.No.	Strategy
1.	READ_ONLY
2.	READ_WRITE
3.	NONSTRICT_READ_WRITE
4.	TRANSACTIONAL

1. READ_ONLY

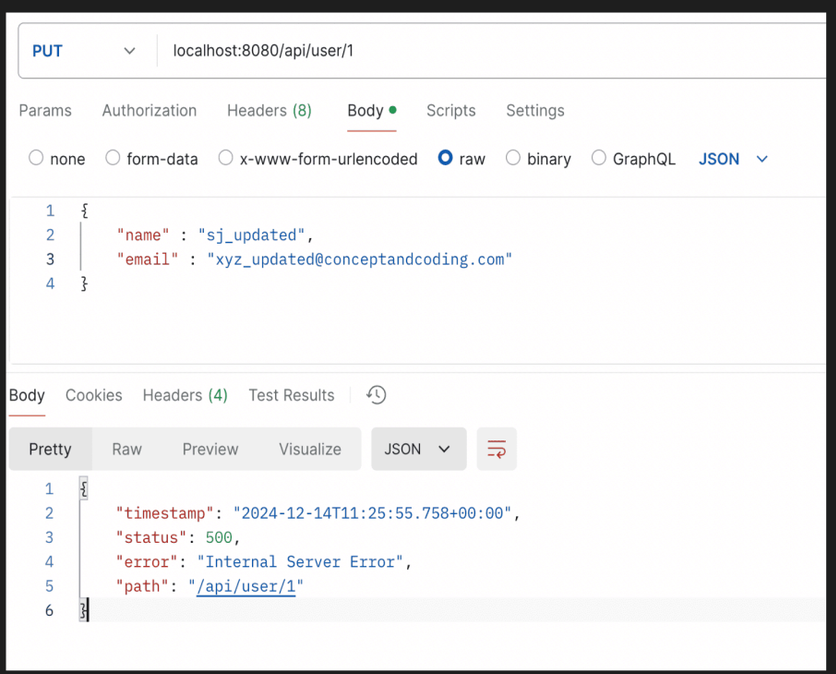
- Good for Static Data
- Which do not require any updates
- If try to update just entity, exception will come

```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @PutMapping(path = "/user/{id}")
    public UserDetails updateUser(@PathVariable Long id, @RequestBody UserDetails userDetails) {
        return userDetailsService.updateUser(id, userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails getUser2() {
        return userDetailsService.findByID( primaryKey: 1L);
    }
}
```

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {

        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        UserDetails existingUser = userDetailsRepository.findById(id).get();
        existingUser.setName(user.getName());
        existingUser.setEmail(user.getEmail());
        return userDetailsRepository.save(existingUser);
    }

    public UserDetails findByID(Long primaryKey) {
        return  userDetailsRepository.findById(primaryKey).get();
    }
}
```

```java
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY,
        region = "userDetailsCache")
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}
```

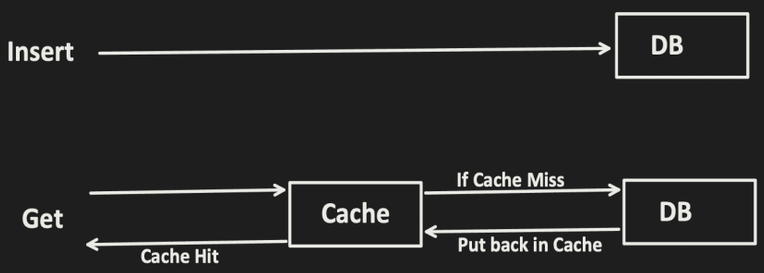
```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {
}
```

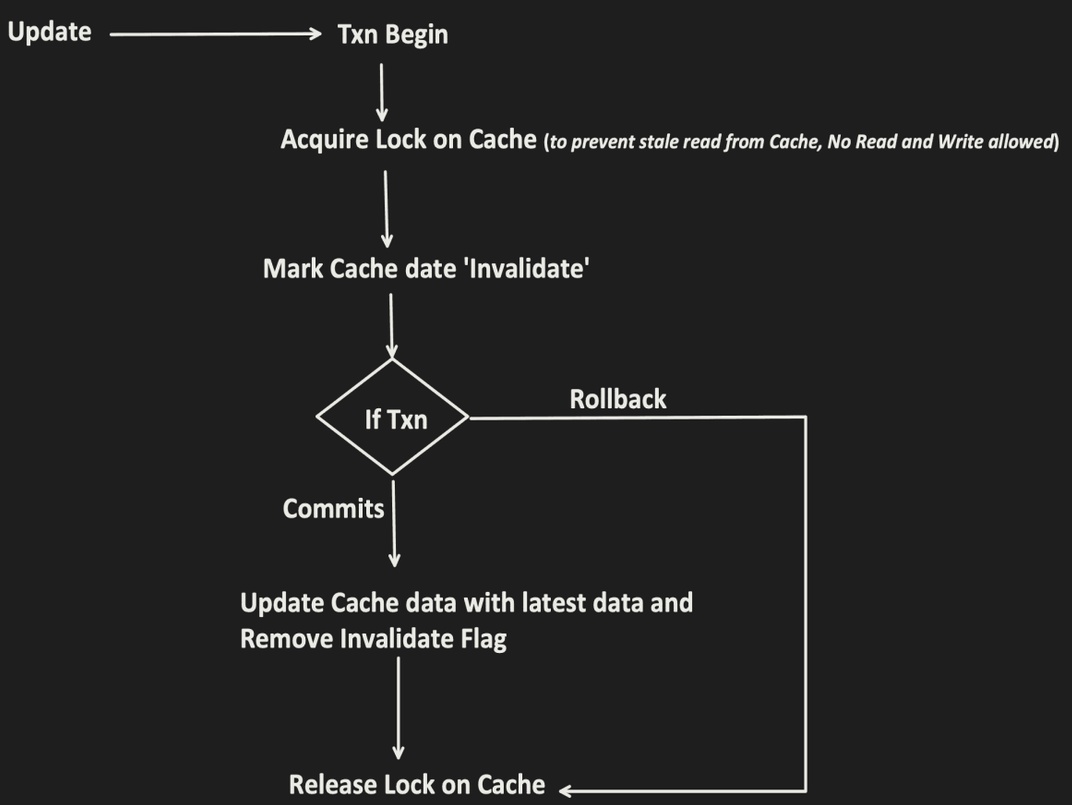


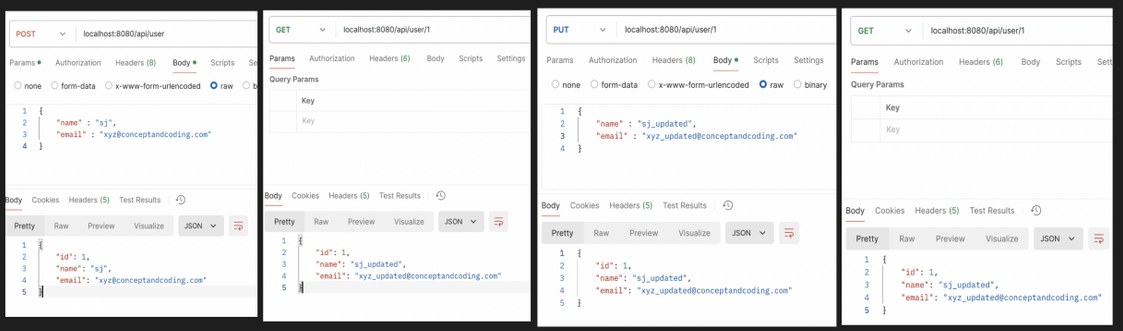

```
java.lang.UnsupportedOperationException Create breakpoint : Can't update readonly object
    at org.hibernate.cache.spi.support.EntityReadOnlyAccess.update(EntityReadOnlyAccess.java:71) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
    at org.hibernate.action.internal.EntityUpdateAction.updateCache(EntityUpdateAction.java:329) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
    at org.hibernate.action.internal.EntityUpdateAction.updateCacheItem(EntityUpdateAction.java:220) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
```

2. READ_WRITE

- During Read, it put Shared Lock, other Read can also acquire Shared Lock. But no Write operation.
- During Update, it put Exclusive Lock, other Read and Write operation not allowed.








```
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
```

Insert Operation (directly inserted into DB)

```
2024-12-14T20:16:29.039+05:30 DEBUG 33070 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDetailsCache` (AccessType[read-write])] by key [cc
2024-12-14T20:16:29.040+05:30 DEBUG 33070 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Cache miss : region = `userDetailsCache`, key = `com.conceptandcoding.learningspringboot
Hibernate:
    select
        ud1_0.id,
        ud1_0.email,
        ud1_0.name
    from
        user_details ud1_0
    where
        ud1_0.id=?
```

Get Operation (Cache Miss, read from DB and inserted into Cache)

```
2024-12-14T20:16:29.058+05:30 DEBUG 33070 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Caching data from load [region=`userDetailsCache` (AccessType[read-write])] : key[com.cc
2024-12-14T20:17:27.407+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDetailsCache` (AccessType[read-write])] by key [cc
2024-12-14T20:17:27.409+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Checking readability of read-write cache item [timestamp=`7103232364662784`, version=`nu
2024-12-14T20:17:27.409+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Cache hit : region = `userDetailsCache`, key = `com.conceptandcoding.learningspringboot.
2024-12-14T20:17:27.416+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Locking cache item [region=`userDetailsCache` (AccessType[read-write])] : `com.conceptar
Hibernate:
    update
        user_details
    set
        email=?,
        name=?
    where
        id=?
```

Update Operation (Cache Lock and update both DB and Cache on Success)

```
2024-12-14T20:17:50.476+05:30 DEBUG 33070 --- [nio-8080-exec-5] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDetailsCache` (AccessType[read-write])] by key [cc
2024-12-14T20:17:50.477+05:30 DEBUG 33070 --- [nio-8080-exec-5] o.h.c.s.support.AbstractReadWriteAccess  : Checking readability of read-write cache item [timestamp=`7103232603873280`, version=`nu
2024-12-14T20:17:50.478+05:30 DEBUG 33070 --- [nio-8080-exec-5] o.h.c.s.support.AbstractReadWriteAccess  : Cache hit : region = `userDetailsCache`, key = `com.conceptandcoding.learningspringboot.
```

Get Operation (Cache hit, No DB hit)

3. NONSTRICT_READ_WRITE

- During Read, No Lock is acquired at all.
- During Update, after txn commit successful, Cache is mark Invalidated and not updated with Fresh data.
- Good for Heavy Read application.
- So if Update and Read happens in parallel, its a chance that read operation get the stale data.

4. TRANSACTIONAL

- Acquire READ lock and Also WRITE lock.
- Updates the cache too, after txn commit successfully.
- Any other READ operation during cache lock, goes directly to DB.
- Any other WRITE operation during cache lock, waits in queue.

spring.jpa.hibernate.ddl-auto configuration tells hibernate regarding how to create and manage the DB Schema.

S.No.	Values	Create Schema	Update Schema	Delete Schema	Details
1.	none	no	no	no	Do nothing. Good for ***Production***
2.	update	yes	yes	no	Does update but without deleting any existing data or schema. Good for ***Development*** environment
3.	validate	no	no	no	During application startup, does matching between entities and DB Schema. If mismatch found, throws exception.
4.	create	yes	yes	yes	Drops and re-create the schema during application startup.
5.	create-drop	Yes	yes	yes	Creates the schema during startup and drops the schema when the application shutdown. *(generally by-default for in memory databases like H2)*

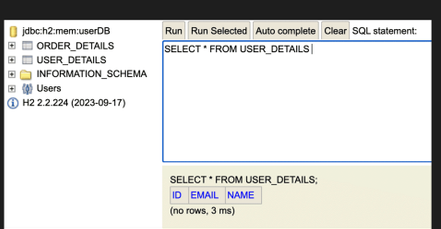
Mapping Classes to Tables

@Tables Annotation

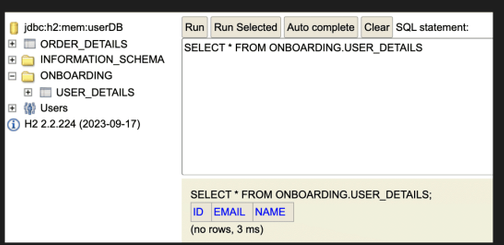
- Its an Optional field, if not defined, hibernate will generate table name based on entity name.
- Generally it follows CamelCase to UPPER_SNAKE_CASE means: UserDetails -> USER_DETAILS

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Table{
String name() default "";
String schema() default"";
UniqueConstraint[] uniqueConstraints() default{};
Index[] indexes() default{};
}
```

```
@Table(name= "USER_DETAILS")
@Entity
public class UserDetails {

//fields and their getters and setters

}
```




```
@Table(name= "USER_DETAILS", schema= "ONBOARDING")
@Entity
public class UserDetails {

//fields and their getters and setters

}
```



```java
@Table(name= "USER_DETAILS",
        schema= "ONBOARDING",
         uniqueConstraints={
              @UniqueConstraint(columnNames="phone"), //single column unique constraint
              @UniqueConstraint(columnNames={"name","email"}) //composite unique constraint
       })
@Entity
public class UserDetails {

       @Id
       private Long id;
       private String name;
       private String email;
       private String phone;

       //Constructors
       public UserDetails(){
       }

       //Getters and setters
}
```

SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE;

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME
USERDB	ONBOARDING	USER_DETAILS	ID	USERDB	ONBOARDING	CONSTRAINT_3
USERDB	ONBOARDING	USER_DETAILS	PHONE	USERDB	ONBOARDING	UKHWI9ICWRBOMPOWJO1L8MQQU49
USERDB	ONBOARDING	USER_DETAILS	EMAIL	USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM
USERDB	ONBOARDING	USER_DETAILS	NAME	USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM

```java
@Table(name= "USER_DETAILS",
        schema= "ONBOARDING",
         uniqueConstraints={
              @UniqueConstraint(columnNames="phone"), //single column unique constraint
              @UniqueConstraint(columnNames={"name","email"}) //composite unique constraint
         },
         indexes={
              @Index(name="index_phone", columnList="phone"), //index on single column
              @Index(name="index_name_email", columnList="name, email") //index on composite column
       })
@Entity
public class UserDetails {

       @Id
       private Long id;
       private String name;
       private String email;
       private String phone;

       //Constructors
       public UserDetails(){
       }

       //Getters and setters
}
```

SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS

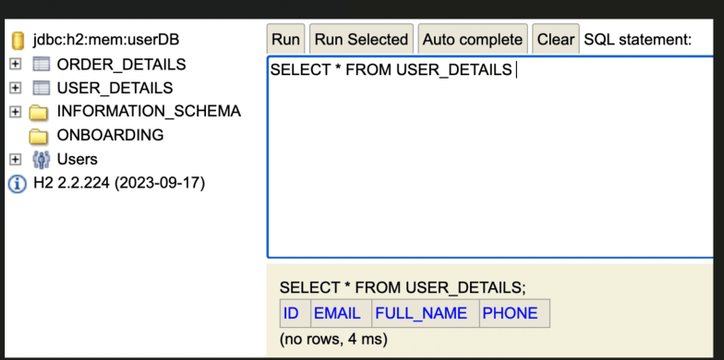
SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS;

INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME
USERDB	ONBOARDING	PRIMARY_KEY_3	USERDB	ONBOARDING	USER_DETAILS	ID
USERDB	ONBOARDING	INDEX_PHONE	USERDB	ONBOARDING	USER_DETAILS	PHONE
USERDB	ONBOARDING	INDEX_NAME_EMAIL	USERDB	ONBOARDING	USER_DETAILS	NAME
USERDB	ONBOARDING	INDEX_NAME_EMAIL	USERDB	ONBOARDING	USER_DETAILS	EMAIL
USERDB	ONBOARDING	UKHWI9ICWRBOMPOWJO1L8MQQU49_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	PHONE
USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	NAME
USERDB	ONBOARDING	UKDRYA4RWKEMVJ1HS8D8P7N59FM_INDEX_3	USERDB	ONBOARDING	USER_DETAILS	EMAIL

@Column Annotation

- Its an Optional field, if not defined, JPA will add it with default values.

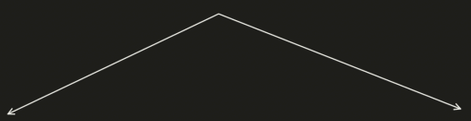
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    private Long id;
    @Column(name = "full_name", unique = true, nullable = false, length = 255)
    private String name;
    private String email;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}
```



@Id Annotation and @GeneratedValue Annotation

Primary Key: must be unique, not null and used to uniquely identify each record.
- **@Id** annotation is used to mark the field as primary key.
- Each entity can have only 1 primary key.
- Only 1 field can be annotated with @Id.

Composite Primary key: combination of two or more columns to form a primary key.



Using **@Embeddable** and **@EmbeddedId** annotation.

Using **@IdClass** and **@Id** annotation

Rules to follow for both the approach:

- Must be a public class.
- Must Implement the Serializable interface.
- Must have no-arg constructor
- Must override the equals() and hashCode() methods

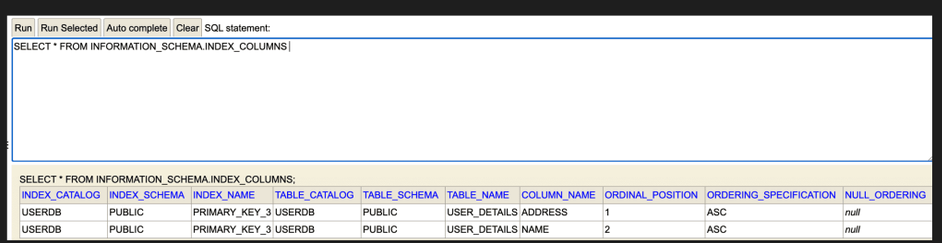
Using **@IdClass** and **@Id** annotation

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    private String name;
    private String address;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters

}
```

I want these 2 columns to be defined as Composite Key

```java
@Table(name = "user_details")
@IdClass(UserDetailsCK.class)
@Entity
public class UserDetails {

    @Id
    private String name;
    @Id
    private String address;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}
```

```java
public class UserDetailsCK implements Serializable {

    private String name;
    private String address;

    public UserDetailsCK() {
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof UserDetailsCK)) {
            return false;
        }
        UserDetailsCK userCK = (UserDetailsCK) obj;
        return this.name.equals(userCK.name) && this.address.equals(userCK.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, address);
    }
}
```



Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS
```

SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS;

INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	ORDERING_SPECIFICATION	NULL_ORDERING
USERDB	PUBLIC	PRIMARY_KEY_3	USERDB	PUBLIC	USER_DETAILS	ADDRESS	1	ASC	*null*
USERDB	PUBLIC	PRIMARY_KEY_3	USERDB	PUBLIC	USER_DETAILS	NAME	2	ASC	*null*

Why we need to override equals and hashCode methods?

- From previous video, we know that JPA internally maintains FIRST LEVEL CACHING.
- Also, we can implement SECOND LEVEL CACHING.
- And these caching uses HashMap and which relies on key (generally primary key becomes the key). So proper equals() and hashCode() method is required.

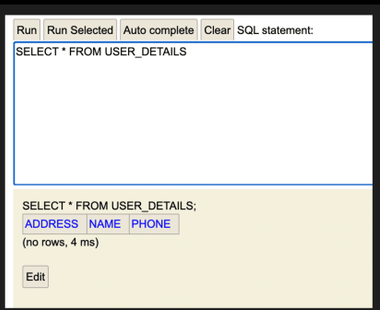
Whey we need to implement Serializable interface?

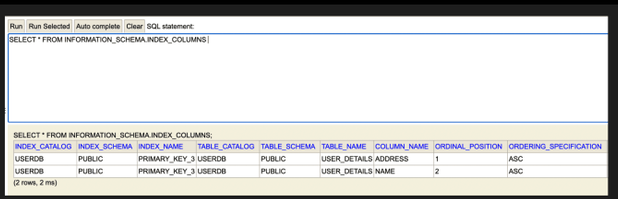
- Unlike Single primary key like String, Long etc. Composite key are custom classes.
- So, JPA need to make sure that those class should be properly serializable (in case if required like transfer over the network in case of distributed caching).

Using **@Embeddable** and **@EmbeddedId** annotation.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @EmbeddedId
    UserDetailsCK userDetailsCK;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    // Getters and setters
}
```

```java
@Embeddable
public class UserDetailsCK implements Serializable {

    private String name;
    private String address;

    public UserDetailsCK() {
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof UserDetailsCK)) {
            return false;
        }
        UserDetailsCK userCK = (UserDetailsCK) obj;
        return this.name.equals(userCK.name) && this.address.equals(userCK.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, address);
    }
}
```





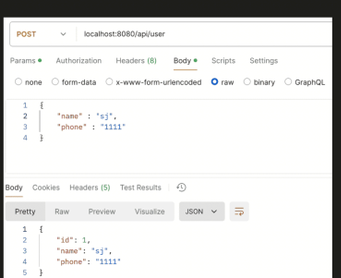
@GeneratedValue Annotation

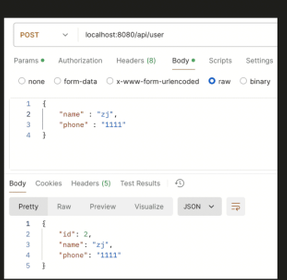
- Now we know, how to define Primary key.
- But we can also define its generation strategy too. By default, primary key columns are not autofill.
- It works with @Id annotation (only for single primary key not for composite one)

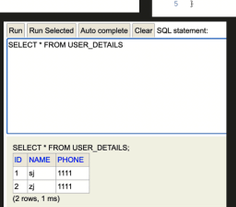
1. GenerationType.IDENTITY

- Each insert, generates a new identifier (auto-increment field)

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

   //getters and setters
}
```





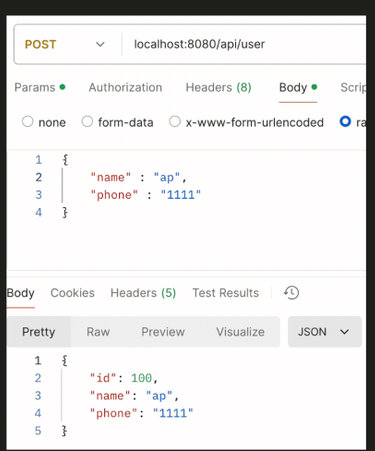


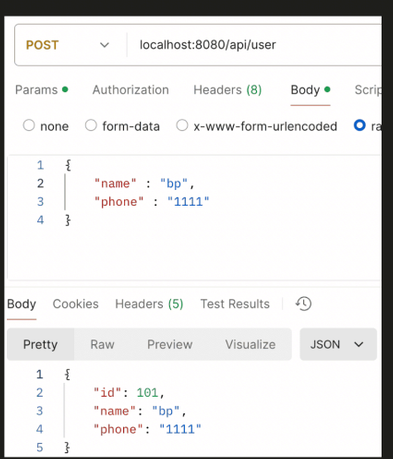
2. GenerationType.SEQUENCE

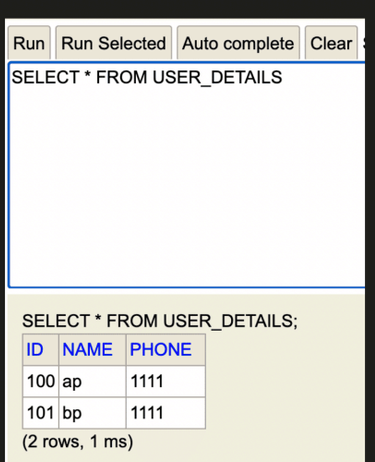
- Used to generate Unique numbers.
- Speed up the efficiency when we cache sequence values.
- More control than IDENTITY.

>> CREATE SEQUENCE user_seq INCREMENT BY 25 START WITH 100 MAXVALUE 9999;

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "unique_user_seq")
    @SequenceGenerator(name = "unique_user_seq", sequenceName = "db_seq_name", initialValue = 100, allocationSize = 5)
    private Long id;
    private String name;
    private String phone;

    // Constructors
    public UserDetails() {
    }

    public UserDetails(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    //getters and setters
}
```







- ***name***: it's a unique name, internal for JPA, we can use it in different entity.
- ***sequenceName***: it's a name which is created in DB. Or if you have manually created the sequence in DB, then use that name here.
- ***initialValue***: sequence no starts from.
- ***allocationSize***: cache data, hibernate prefetch this much ids before hand, so that it will not query DB again and again.

We have given allocationSize = 5, so after 5 calls only, next db call will be made for sequence

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***1st call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***2nd call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***3rd call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***4th call***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```
→ ***5th call***

```
Hibernate:
    select
        next value for db_seq_name
```
→ ***After 5th call, hibernate will fetch another 5 values.***

```
Hibernate:
    insert
    into
        user_details
        (name, phone, id)
    values
        (?, ?, ?)
```

Advantage of SEQUENCE over IDENTITY:

1. Custom logic (start point, increment etc.)
2. Sequence generation logic is independent of table, so multiple tables an use it.f
3. Range of IDs can be cached, so we can avoid hitting database each time a new id is required.
   (during IDENTITY, while INSERTION internally DB is auto generating the next ID , which require additional DB call)
4. Better portability, means IDENTITY is very DB specific while SEQUENCE can provide more consistent behavior across multiple DBs.

3. GenerationType.TABLE

- @TableGenerator annotation is used but its very less efficient.
- Because:
  - Separate Table is created, just for managing unique IDs.
  - Each time id is required, SELECT-UPDATE query is executed.
  - Complex concurrency handling, when multiple operations happening in parallel, it requires LOCK/UNLOCK functionality. Which can lead to performance bottleneck.
    In SEQUENCE type, its handle internally by DB using atomic counter, so its much more efficient.

#27 : OneToOne Mapping (Unidirectional and Bidirectional)

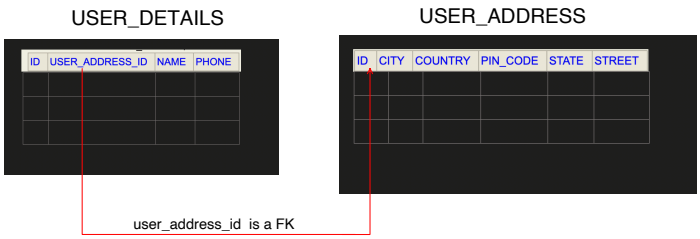
Sunday, 29 December 2024 12:26 PM

@OneToOne Unidirectional

- One Entity(A) references only one instance of another Entity(B).
- But reference exist only in one Direction i.e. from Parent(A) to Child(B).

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```java
@Entity
@Table(name = "user_address")
public class UserAddress {

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

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    // Constructors
    public UserAddress() {
    }
}
```

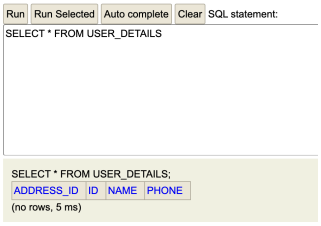


- By default hibernate choose:
  - the Foreign Key (FK) name as : <field_name_id>
    that's why for "userAddress" it created the FK name as : user_address_id
  - Chooses the Primary Key (PK) of other table.

But If we need more control over it, we can use **@JoinColumn** annotation.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```java
@Entity
@Table(name = "user_address")
public class UserAddress {

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

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    // Constructors
    public UserAddress() {
    }
}
```



What about Composite Key, how to reference it?

- We need to use ***@JoinColumns*** and need to map all columns.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumns({
            @JoinColumn(name = "address_street", referencedColumnName = "street"),
            @JoinColumn(name = "address_pin_code", referencedColumnName = "pinCode")
    })
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```java
@Entity
@Table(name = "user_address")
public class UserAddress {

    @EmbeddedId
    private UserAddressCK id;

    private String city;
    private String state;
    private String country;


   //getters and setters
}
```

```java
@Embeddable
public class UserAddressCK {

    private String street;
    private String pinCode;

   //getters and setters
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

ID	ADDRESS_PIN_CODE	ADDRESS_STREET	NAME	PHONE

(no rows, 2 ms)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_ADDRESS
```

SELECT * FROM USER_ADDRESS;

CITY	COUNTRY	PIN_CODE	STATE	STREET

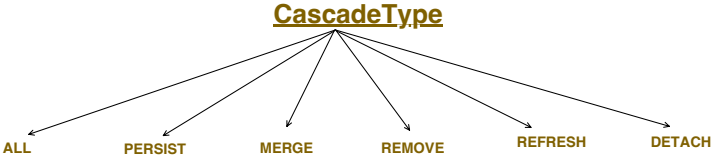
(no rows, 1 ms)

SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS;

INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	ORDERING_SPECIFICATION
USERDB	PUBLIC	PRIMARY_KEY_3	USERDB	PUBLIC	USER_DETAILS	ID	1	ASC
USERDB	PUBLIC	CONSTRAINT_INDEX_3	USERDB	PUBLIC	USER_DETAILS	ADDRESS_PIN_CODE	1	ASC
USERDB	PUBLIC	CONSTRAINT_INDEX_3	USERDB	PUBLIC	USER_DETAILS	ADDRESS_STREET	2	ASC
USERDB	PUBLIC	PRIMARY_KEY_9	USERDB	PUBLIC	USER_ADDRESS	PIN_CODE	1	ASC
USERDB	PUBLIC	PRIMARY_KEY_9	USERDB	PUBLIC	USER_ADDRESS	STREET	2	ASC

Now before we proceed with further Mappings, there is one more important thing i.e.

CascadeType



- Without CascadeType, any operation on Parent do not affect Child entity.
- Managing Child entities explicitly can be error-prone.

CascadeType.PERSIST

- Persisting/Inserting the User entity automatically persists its associated UserAddress entity data.

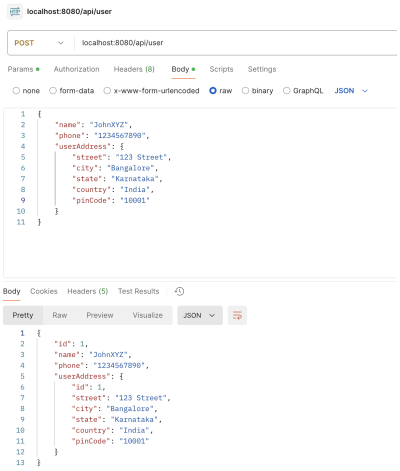
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

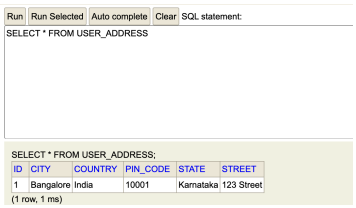
```java
@Entity
@Table(name = "user_address")
public class UserAddress {

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

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    //getters and setters
}
```

```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }
}
```

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }
}
```

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {
}
```





CascadeType.MERGE

- Updating the User entity automatically updates its associated UserAddress entity data.

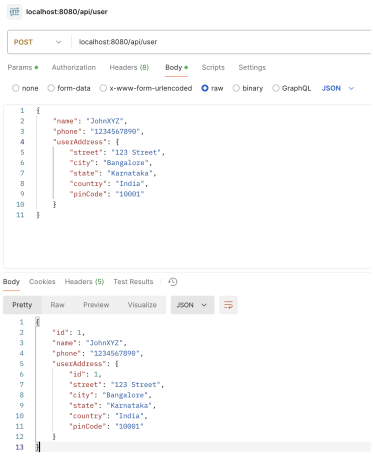
Lets, first see, what happen if we use only **PERSIT** Cascade type

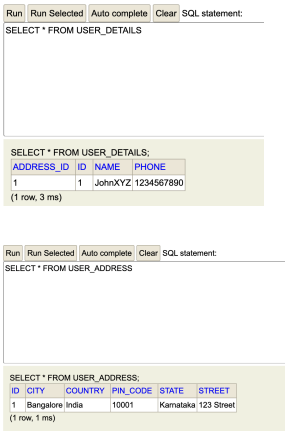
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @PutMapping(path = "/user/{id}")
    public UserDetails updateUser(@PathVariable Long id, @RequestBody UserDetails userDetails) {
        return userDetailsService.updateUser(id, userDetails);
    }
}
```

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsRepository.findById(id);
        if(existingUser.isPresent()) {
            return userDetailsRepository.save(user);
        }
        return null;
    }
}
```

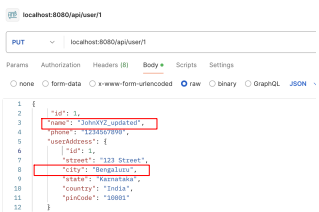
We are using the same "save" method to update, internally it check, if its new entity or not, by looking for ID field present in entity object. Since we are passing ID in Request body, it will try to update instead of insert.

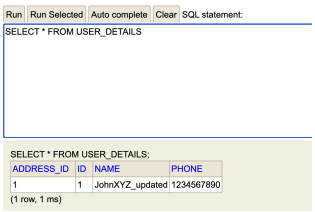
1st INSERT OPERATION

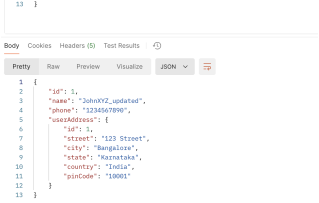


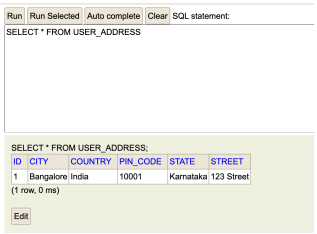


2nd UPDATE OPERATION:
(Changes made in both UserDetails and UserAddress, but only UserDetails changes reflected)





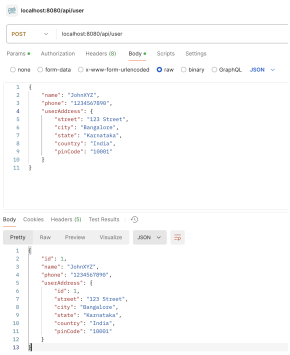


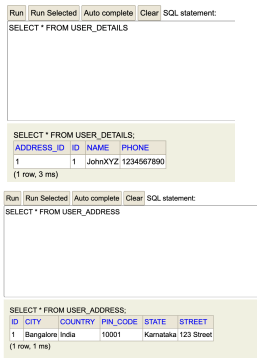


Now If I want to both INSERT and then UPDATE association capability, means I need both PERSIT + MERGE Cascade Type

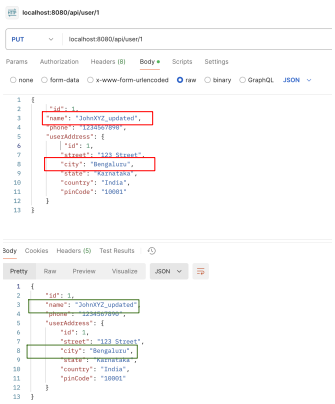
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

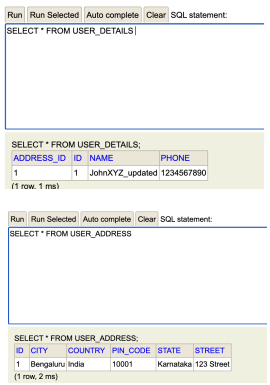
1st INSERT OPERATION , Similar like above.





2nd UPDATE OPERATION





CascadeType.REMOVE

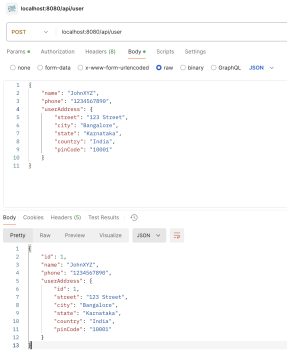
- Deleting the User entity automatically delete its associated UserAddress entity data.

```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @PutMapping(path = "/user/{id}")
    public UserDetails updateUser(@PathVariable Long id, @RequestBody UserDetails userDetails) {
        return userDetailsService.updateUser(id, userDetails);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userDetailsService.deleteUser(id);
        return ResponseEntity.noContent().build();
    }
}
```

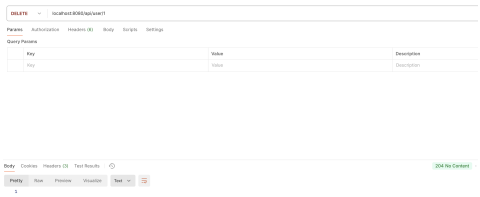
```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsRepository.findById(id);
        if(existingUser.isPresent()) {
            return userDetailsRepository.save(user);
        }
        return null;
    }

    public void deleteUser(Long userId) {
        userDetailsRepository.deleteById(userId);
    }
}
```

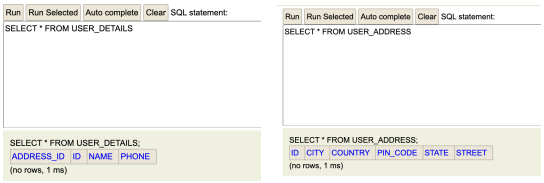
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

1st INSERT OPERATION , Similar like above.



2nd DELETE OPERATION





CascadeType.REFRESH and CascadeType.DETACH

- Both are Less frequently used.

CascaseType.REFRESH

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsRepository.findById(id);
        if(existingUser.isPresent()) {
            return userDetailsRepository.save(user);
        }
        return null;
    }

    public void deleteUser(Long userId) {
        userDetailsRepository.deleteById(userId);
    }
}
```

Internally JPA code, has access to EntityManager object, which maintains persistenceContext and thus handles FIRST LEVEL CACHING

```java
@Transactional
public <S extends T> S save(S entity) {
    Assert.notNull(entity, message: "Entity must not be null");
    if (this.entityInformation.isNew(entity)) {
        this.entityManager.persist(entity);
        return entity;
    } else {
        return this.entityManager.merge(entity);
    }
}
```

EntityManager.java

```java
/**
 * Refresh the state of the instance from the database,
 * overwriting changes made to the entity, if any.
 * @param entity  entity instance
 * @throws IllegalArgumentException if the instance is not
 *         an entity or the entity is not managed
 * @throws TransactionRequiredException if there is no
 *         transaction when invoked on a container-managed
 *         entity manager of type <code>PersistenceContextType.TRANSACTION</code>
 * @throws EntityNotFoundException if the entity no longer
 *         exists in the database
 */
public void refresh(Object entity);
```

- Sometime, we want to BYPASS 'First Level Caching'
- So, EntityManager has one method called "refresh".
- What it does is, for that entity directly read the value from DB instead of First Level Cache

So, When we use the CascadeType.REFRESH, we tell JPA to not only read Parent Entity from DB but also its associated Child Entities too.

So in this case, whenever on USERDETAILS entity, entityManager will internally invoke 'refresh() method', this Cascade type make sure that, USERADDRESS should also be read from DB not from First Level Cache.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.REFRESH})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

CascaseType.DETACH

- Similarly like persist, remove, refresh method. EntityManager also has 'detach' method.
- Which purpose is to remove the given entity from the PERSISTENCE CONTEXT.
- Means JPA is not managing its lifecycle now.

```java
/**
 * Remove the given entity from the persistence context, causing
 * a managed entity to become detached.  Unflushed changes made
 * to the entity if any (including removal of the entity),
 * will not be synchronized to the database.  Entities which
 * previously referenced the detached entity will continue to
 * reference it.
 * @param entity  entity instance
 * @throws IllegalArgumentException if the instance is not an
 *         entity
 * @since 2.0
 */
public void detach(Object entity);
```

So, When we use the CascadeType.DETACH, we tell JPA to not only detach Parent Entity from persistence context but also its associated Child Entities too.

So in this case, whenever on USERDETAILS entity, entityManager will internally invoke 'detach() method', this Cascade type make sure that, USERADDRESS should also be detached/removed from persistence context.

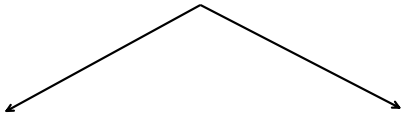
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = {CascadeType.DETACH})
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

CascaseType.ALL

- Means we want all the different Cascade types capabilities like PERSIST, MERGE, REMOVE, REFRESH & DETACH.

Okay, we saw INSERT, UPDATE, REMOVE operation, but what about GET?

Does child entities always get loaded with Parent Entity?

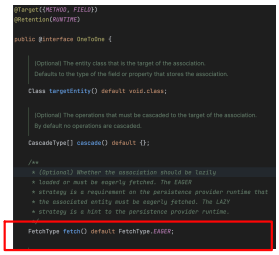


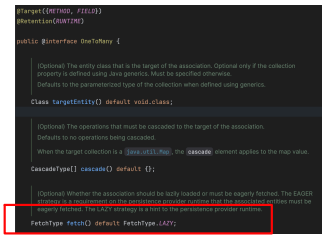
Eager Loading

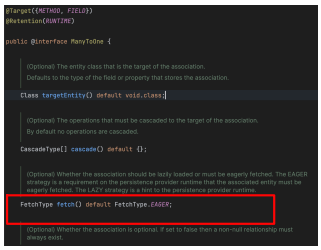
Lazy Loading

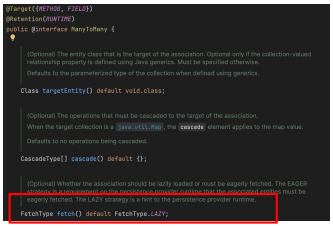
- It means, associated entity is loaded immediately along with the parent entity.
- Default for @OneToOne and @ManyToOne

- It means, associated entity is NOT loaded immediately.
- Only loaded when explicitly accessed like when we call userDetail.getUserAddress().
- Default for @OneToMany, @ManyToMany.









- JPA, do an assumption that, since there is only 1 child entity present, so possibly it might be required while accessing parent entity.

- But here, there can be many child entities present for a parent entity, so returning all the rows of the child entity might impact performance, so by-default it uses LAZY technique.

We can also control this default behavior:

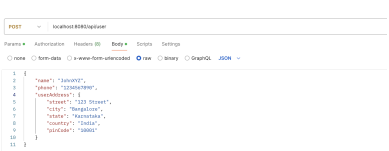
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails fetchUser(@PathVariable Long id) {
        return userDetailsService.findByID(id);
    }
}
```

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

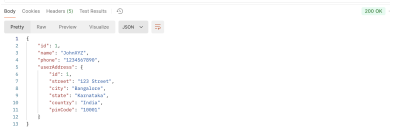
```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails findByID(Long primaryKey) {
        return  userDetailsRepository.findById(primaryKey).get();
    }
}
```

Lets try testing this code:

1st Insert Operation: (Success)



2nd Get Operation: (Failure)



GET operation failed because of loading UserAddress LAZILY, Since during JSON creation at the time of response, UserAddress data is missing thus library like JAKSON don't know how to serialize it and thus it failed while constructing the response.

```
com.fasterxml.jackson.databind.exc.InvalidDefinitionException Create breakpoint : No serializer found for class org.hibernate.proxy.pojo.bytebuddy.ByteBuddyInterceptor
    at com.fasterxml.jackson.databind.exc.InvalidDefinitionException.from(InvalidDefinitionException.java:77) ~[jackson-databind-2.17.2.jar:2.17.2]
    at com.fasterxml.jackson.databind.SerializerProvider.reportBadDefinition(SerializerProvider.java:1330) ~[jackson-databind-2.17.2.jar:2.17.2]
    at com.fasterxml.jackson.databind.DatabindContext.reportBadDefinition(DatabindContext.java:414) ~[jackson-databind-2.17.2.jar:2.17.2]
```

Hey, then how come INSERT operation, we don't see this serialization issue?

Its because during INSERT, we are inserting the data in DB and also its inserted into persistence Context (first level cache), so UserAddress data is already present in-memory. So when response is returned as part of same INSERT operation, it do not make any DB call, just fetched from the persistence context.

So, for GET operation, how to solve it:

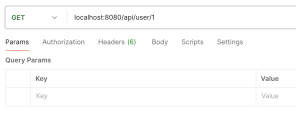
- Use **@JsonIgnore**
- This will remove the UserAddress field totally for both Lazy and Eager loading.

- Using **DTO (Data Transfer Object)**
- Much clean and recommended approach.
- Instead of sending Entity directly, first response will be mapped to our DTO object.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    @JsonIgnore
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

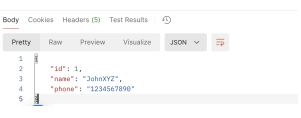
```java
public class UserDetailsDTO {

    private Long id;
    private String name;
    private String phone;
    private String address;

    // Constructor to populate from UserDetails entity
    public UserDetailsDTO(UserDetails userDetails) {
        this.id = userDetails.getId();
        this.name = userDetails.getName();
        this.phone = userDetails.getPhone();
        System.out.println("going to query user address here now");
        this.address = userDetails.getUserAddress() != null ?
                userDetails.getUserAddress().getStreet() : null;
    }

    //getters and setters
}
```

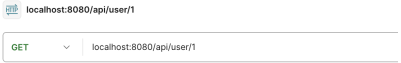
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    public UserDetailsDTO toDTO() {
        return new UserDetailsDTO(this);
    }

    //getters and setters
}
```

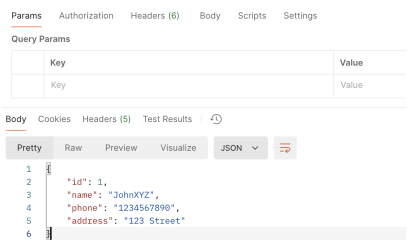


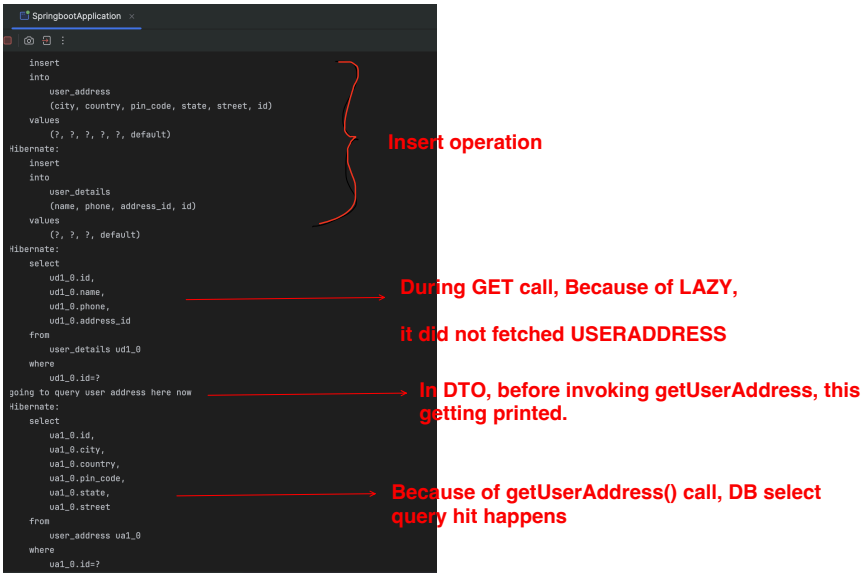
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetailsDTO fetchUser(@PathVariable Long id) {
        return userDetailsService.findByID(id).toDTO();
    }
}
```

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails findByID(Long primaryKey) {
        return  userDetailsRepository.findById(primaryKey).get();
    }
}
```



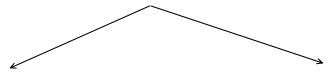






@OneToOne bidirectional

- Both entities hold reference to each other, means:
  - UserDetails has a reference to UserAddress.
  - UserAddress also has a reference back to UserDetails (only in Object, not in DB table)



Owner side

Inverse side

- Holds the Foreign Key relationship in a table.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

- No Foreign key is created in table.
- Only holds ***Object*** reference of owing entity.

```java
@Entity
@Table(name = "user_address")
public class UserAddress {

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

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    @OneToOne(mappedBy = "userAddress", fetch = FetchType.EAGER)
    private UserDetails userDetails;

    //getters and setters
}
```

Table looks exactly same as OneToOne Unidirectional only

SELECT * FROM USER_DETAILS;

ADDRESS_ID	ID	NAME	PHONE

(no rows, 5 ms)

SELECT * FROM USER_ADDRESS;

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET

(no rows, 4 ms)

address_id is a FK

But we now have capability to go backward from UserAddress to UserDetails.

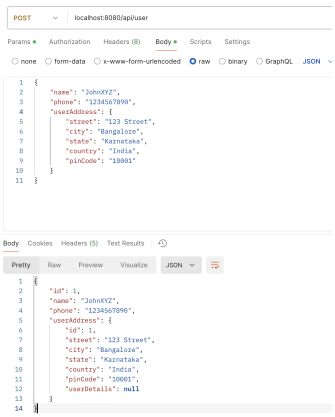
Created a controller and service class to query UserAddress entity

```java
@RestController
@RequestMapping(value = "/api/")
public class UserAddressController {

    @Autowired
    UserAddressService userDetailsService;

    @GetMapping("/user-address/{id}")
    public UserAddress fetchUser(@PathVariable Long id) {
        return userDetailsService.findByID(id);
    }
}
```

```java
@Service
public class UserAddressService {

    @Autowired
    UserAddressRepository userAddressRepository;

    public UserAddress findByID(Long primaryKey) {
        return  userAddressRepository.findById(primaryKey).get();
    }
}
```

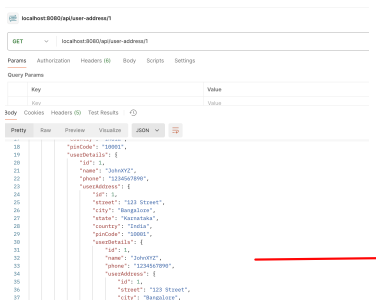
Let's observe the behavior now,

1st: Insert Operation like before, using UserDetail controller API:




```
2024-12-31T16:45:04.749+05:30  INFO 39163 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
Hibernate:
    insert
    into
        user_address
        (city, country, pin_code, state, street, id)
    values
        (?, ?, ?, ?, ?, default)
Hibernate:
    insert
    into
        user_details
        (name, phone, address_id, id)
    values
        (?, ?, ?, default)
```

2nd: Get call of UserAddress Controller API:



INFINITE RECURSION

```
{
  "state": "Karnataka",
  "country": "India",
  "pinCode": "10001",
  "userDetails": {
    "id": 1,
    "name": "JohnXYZ",
    "phone": "1234567890",
    "userAddress": {
      "id": 1,
      "street": "123 Street",
```

```
select
    ua1_0.id,
    ua1_0.city,
    ua1_0.country,
    ua1_0.pin_code,
    ua1_0.state,
    ua1_0.street,
    ud1_0.id,
    ud1_0.name,
    ud1_0.phone
from
    user_address ua1_0
left join
    user_details ud1_0
        on ua1_0.id=ud1_0.address_id
where
    ua1_0.id=?
2024-12-31T16:47:23.303+05:30  WARN 39163 --- [nio-8080-exec-4] .w.s.m.s.DefaultHandlerExceptionResolver : Ignoring exception, response committed already: org.springframework.http.conv
2024-12-31T16:47:23.303+05:30  WARN 39163 --- [nio-8080-exec-4] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.http.converter.HttpMessageNotWritableException:
```

Successfully executed JOIN query from UserAddress to UserDetail table

But exception comes during Response building

Why because :

During response construction:

1. Jackson starts serializing UserAddress after UserAddress is serialized.
2. It encounters UserDetails within UserAddress and starts serializing it.
   After UserDetails is serialized.
3. Inside UserDetails , it encounters UserAddress again and serialization keeps going on in loop.

Infinite Recursion issue in bidirectional mapping can be solved via:

@JsonManagedReference and @JsonBackReference

@JsonManagedReference:

- Should be Used only in Owing entity.
- Tells explicitly Jackson to go ahead and serialize the child entity.

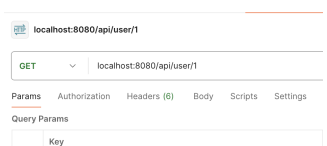
@JsonBackReference:

- Should only be Used with Inverse/Child entity.
- Tells explicitly Jackson to not serialize the parent entity.

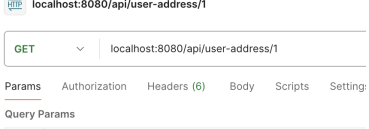
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    @JsonManagedReference
    private UserAddress userAddress;


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

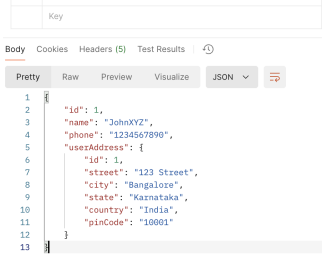
```java
@Entity
@Table(name = "user_address")
public class UserAddress {

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

   private String street;
   private String city;
   private String state;
   private String country;
   private String pinCode;

   @OneToOne(mappedBy = "userAddress", fetch = FetchType.EAGER)
   @JsonBackReference
   private UserDetails userDetails;

   //getters and setters
}
```

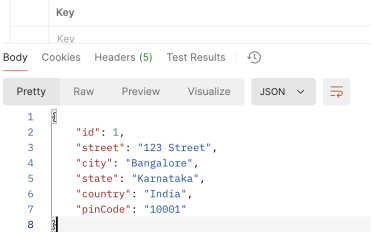
GET API call on Parent Entity "UserDetail"



GET API call on Child Entity "UserAddress"







Is there a way that, I can load the associated entity from both side, but still avoid infinite recursion:

@JsonIdentityInfo

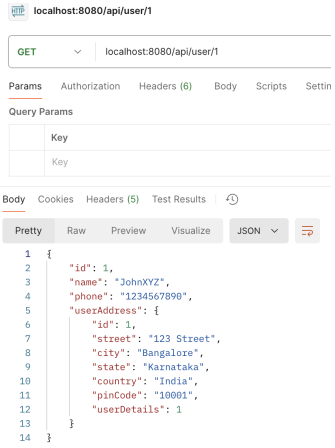
- During serialization , Jackson gives the unique ID to the entity (based on property field).
- Though which Jackson can know, if that particular id entity is already serialized before, then it skip the serialization.

```java
@Table(name = "user_details")
@Entity
@JsonIdentityInfo(
        generator = ObjectIdGenerators.PropertyGenerator.class,
        property = "id"
)
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private UserAddress userAddress;

    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

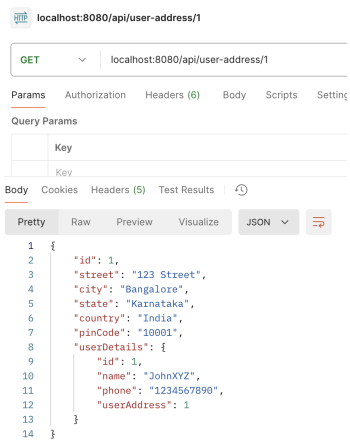
Any unique field from entity we can give ,generally we give PK

```java
@Entity
@Table(name = "user_address")
@JsonIdentityInfo(
        generator = ObjectIdGenerators.PropertyGenerator.class,
        property = "id"
)
public class UserAddress {

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

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    @OneToOne(mappedBy = "userAddress", fetch = FetchType.EAGER)
    private UserDetails userDetails;

    //getters and setters

}
```

GET API call on Parent Entity "UserDetail"



GET API call on Child Entity "UserAddress"



#28 JPA-Part7

Saturday, 18 January 2025 11:38 AM

One-to-Many Unidirectional

- One entity associated with multiple records in another entity like: User can have many Orders.
- Reference exist only in 1 direction i.e. from Parent to Child.

- Since its 1:Many, means 1 parent have multiple child and we can not store multiple child ids in 1 parent row, so it creates a ***NEW TABLE*** and stores the mapping.
- By default its ***Lazy*** loading, means when query parents, child rows are not fetched.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    private List<OrderDetails> orderDetails = new ArrayList<>();


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```java
@Table(name = "order_details")
@Entity
public class OrderDetails {

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

    private String productName;

    //getters and setters
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

```
SELECT * FROM USER_DETAILS;
```

ID	NAME	PHONE

(no rows, 1 ms)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM ORDER_DETAILS
```

```
SELECT * FROM ORDER_DETAILS;
```

ID	PRODUCT_NAME

(no rows, 4 ms)

Edit

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS_ORDER_DETAILS
```

```
SELECT * FROM USER_DETAILS_ORDER_DETAILS;
```

ORDER_DETAILS_ID	USER_DETAILS_ID

(no rows, 1 ms)

What if, we don't want to create new table:

- We can use ***@JoinColumn***, this also tells JPA that we want to store the FK in Child table instead of creating a new table.

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```java
@Table(name = "order_details")
@Entity
public class OrderDetails {

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

    private String productName;

    //getters and setters
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

USER_ID	NAME	PHONE

(no rows, 1 ms)

Run | Run Selected | Auto complete | Clear | SQL statement:

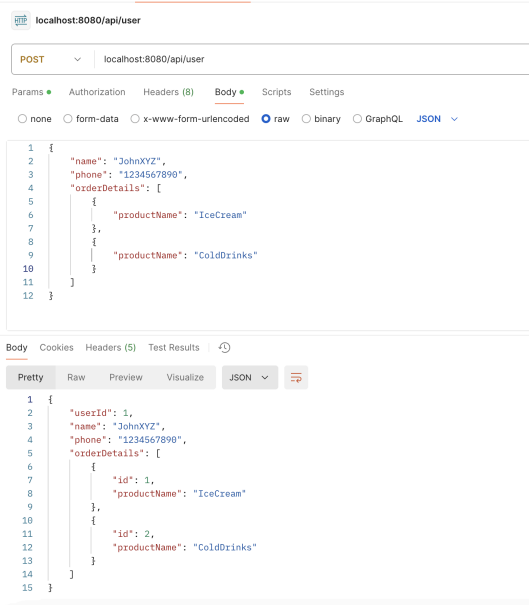
```
SELECT * FROM ORDER_DETAILS
```

SELECT * FROM ORDER_DETAILS;

ID	USER_ID_FK	PRODUCT_NAME

(no rows, 1 ms)

1. INSERT CALL



Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM ORDER_DETAILS
```

SELECT * FROM USER_DETAILS;

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

(1 row, 1 ms)

SELECT * FROM ORDER_DETAILS;

ID	USER_ID_FK	PRODUCT_NAME
1	1	IceCream
2	1	ColdDrinks

(2 rows, 1 ms)

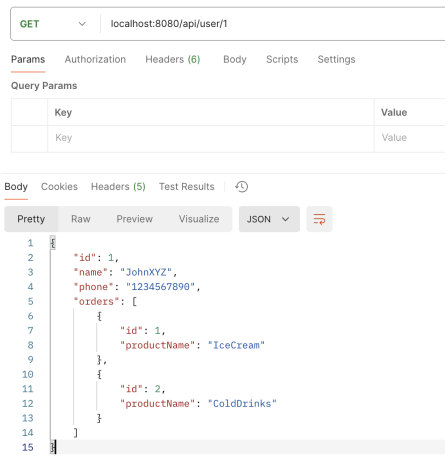
2. GET CALL (LAZY)

```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetailsDTO fetchUser(@PathVariable Long id) {
        UserDetails output =  userDetailsService.findByID(id);
        System.out.println("going to map UserDetails to UserDTO");
        UserDetailsDTO userDTO = output.mapUserDetailsToUserDTO();
        return userDTO;
    }
}
```

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();

    public UserDetailsDTO mapUserDetailsToUserDTO() {

        return new UserDetailsDTO(this);
    }

    //getters and setters

}
```

```java
public class UserDetailsDTO {

    private Long id;
    private String name;
    private String phone;
    private List<OrderDetails> orders;

    // Constructor to populate from UserDetails entity
    public UserDetailsDTO(UserDetails userDetails) {
        this.id = userDetails.getUserId();
        this.name = userDetails.getName();
        this.phone = userDetails.getPhone();
        System.out.println("going to query order table here now");
        this.orders = userDetails.getOrderDetails();
    }

    //getters and setters
}
```

Output:




```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_id=?
going to map UserDetails to UserDTO
going to query order table here now
Hibernate:
    select
        od1_0.user_id_fk,
        od1_0.id,
        od1_0.product_name
    from
        order_details od1_0
    where
        od1_0.user_id_fk=?
```

EAGER fetch:

```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails fetchUser(@PathVariable Long id) {
        return userDetailsService.findByID(id);
    }
}
```

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();


    // Constructors
    public UserDetails() {
    }

    //getters and setters

}
```

Output:

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.name,
        ud1_0.phone,
        od1_0.user_id_fk,
        od1_0.id,
        od1_0.product_name
    from
        user_details ud1_0
    left join
        order_details od1_0
            on ud1_0.user_id=od1_0.user_id_fk
    where
        ud1_0.user_id=?
```

Different Cascade Types:

Cascade Type	Impact
CascadeType.PERSIST	Saving User(Parent) also saves related Orders (Child).
CascadeType.MERGE	Updating User also updates Orders.
CascadeType.REMOVE	Deleting User also deletes Orders.
CascadeType.ALL	Includes all cascade operations.

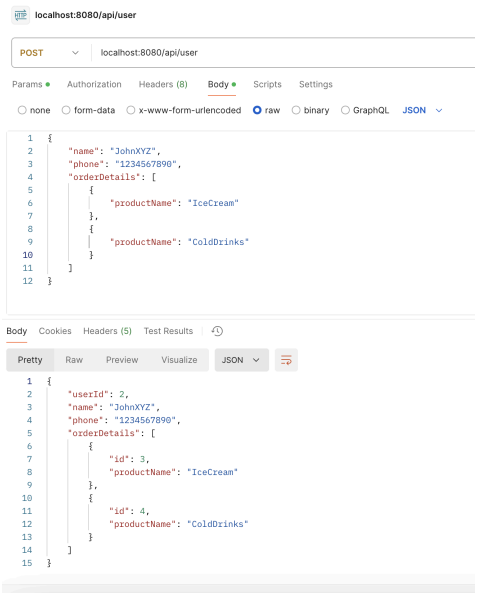
Orphan Removal:

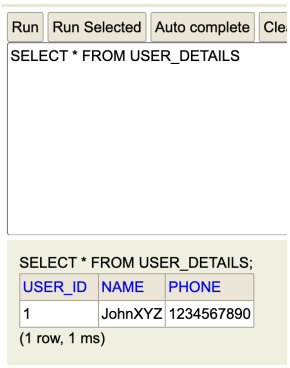
Automatically removes child entry when child removed from Parent collection.

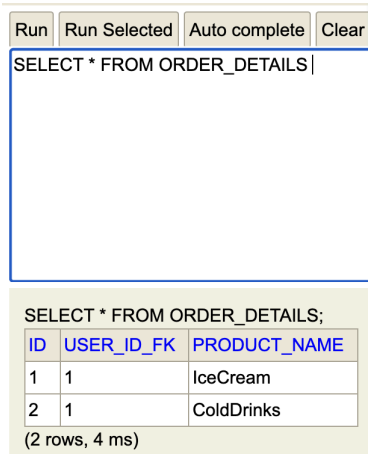
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

    @PostMapping(path = "/user")
    public UserDetails insertUser(@RequestBody UserDetails userDetails) {
        return userDetailsService.saveUser(userDetails);
    }

    @GetMapping("/user/{id}")
    public UserDetails testOrphan(@PathVariable Long id) {
        UserDetails output =  userDetailsService.findByID(id);
        output.getOrderDetails().remove( index: 0);
       userDetailsService.saveUser(output);
       return output;

    }
}
```

```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = false)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

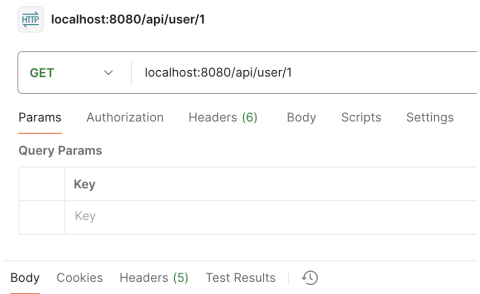
1st: INSERT

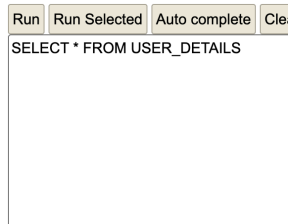


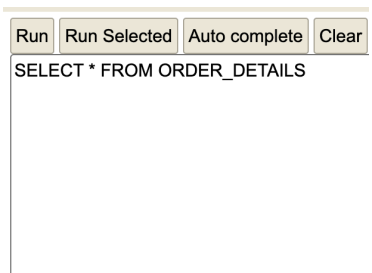




2nd: Testing Orphan Removal (With OrphanRemoval = false)







Pretty Raw Preview Visualize JSON


```
{
    "userId": 1,
    "name": "JohnXYZ",
    "phone": "1234567890",
    "orderDetails": [
        {
            "id": 2,
            "productName": "ColdDrinks"
        }
    ]
}
```


SELECT * FROM USER_DETAILS;

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

(1 row, 1 ms)

SELECT * FROM ORDER_DETAILS;

ID	USER_ID_FK	PRODUCT_NAME
1	*null*	IceCream
2	1	ColdDrinks

(2 rows, 0 ms)

Console Output: Persistence Context knows all the info, which is present in memory

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_id=?
Hibernate:
    select
        od1_0.user_id_fk,
        od1_0.id,
        od1_0.product_name
    from
        order_details od1_0
    where
        od1_0.user_id_fk=?
Hibernate:
    update
        order_details
    set
        user_id_fk=null
    where
        user_id_fk=?
        and id=?
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "user_id_fk", referencedColumnName = "userId")
    private List<OrderDetails> orderDetails = new ArrayList<>();


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

localhost:8080/api/user/1

Run Run Selected Auto complete Cle

GET localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

Key
Key

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "userId": 1,
    "name": "JohnXYZ",
    "phone": "1234567890",
    "orderDetails": [
        {
            "id": 2,
            "productName": "ColdDrinks"
        }
    ]
}
```


SELECT * FROM USER_DETAILS

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS;

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

(1 row, 1 ms)

SELECT * FROM ORDER_DETAILS

SELECT * FROM ORDER_DETAILS;

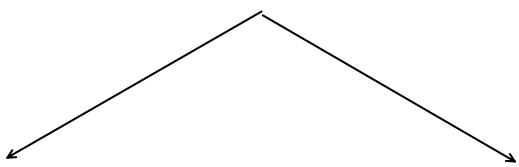
ID	USER_ID_FK	PRODUCT_NAME
2	1	ColdDrinks

(1 row, 1 ms)

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_id=?
Hibernate:
    select
        od1_0.user_id_fk,
        od1_0.id,
        od1_0.product_name
    from
        order_details od1_0
    where
        od1_0.user_id_fk=?
Hibernate:
    update
        order_details
    set
        user_id_fk=null
    where
        user_id_fk=?
        and id=?
Hibernate:
    delete
    from
        order_details
    where
        id=?
```

One-to-ManyBidirectional

- Parent reference to child.
- Each Child reference to Parent.



Inverse Side

Owning Side

- No Foreign key is created in table.
- Only holds ***Object*** reference of owing entity.

- Holds the Foreign Key relationship in a table.

```java
@Table(name = "user_details")
@Entity
@JsonIdentityInfo(
        generator = ObjectIdGenerators.PropertyGenerator.class,
        property = "userId")
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;
    @OneToMany(mappedBy = "userDetails", cascade = CascadeType.ALL)
    private List<OrderDetails> orderDetails = new ArrayList<>();

    public Long getUserId() {
        return userId;
    }
    public void setUserId(Long userId) {
        this.userId = userId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
    public List<OrderDetails> getOrderDetails() {
        return orderDetails;
    }
    public void setOrderDetails(List<OrderDetails> orderDetails) {
        this.orderDetails = orderDetails;
        for(OrderDetails order: orderDetails) {
            order.setUserDetails(this);
        }
    }
}
```

```java
@Table(name = "order_details")
@Entity
@JsonIdentityInfo(
        generator = ObjectIdGenerators.PropertyGenerator.class,
        property = "id")
public class OrderDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String productName;
    @ManyToOne
    @JoinColumn(name = "user_id_owing_fk", referencedColumnName = "userID")
    private UserDetails userDetails;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
    public UserDetails getUserDetails() {
        return userDetails;
    }
    public void setUserDetails(UserDetails userDetails) {
        this.userDetails = userDetails;
    }
}
```

localhost:8080/api/user

POST localhost:8080/api/user

Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary

```
{
    "name": "JohnXYZ",
    "phone": "1234567890",
    "orderDetails": [
        {
            "productName": "IceCream"
        },
        {
            "productName": "ColdDrinks"
        }
    ]
}
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "userId": 1,
    "name": "JohnXYZ",
    "phone": "1234567890",
    "orderDetails": [
        {
            "id": 1,
            "productName": "IceCream",
            "userDetails": 1
        },
        {
            "id": 2,
            "productName": "ColdDrinks",
            "userDetails": 1
        }
    ]
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

USER_ID	NAME	PHONE
1	JohnXYZ	1234567890

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM ORDER_DETAILS
```

SELECT * FROM ORDER_DETAILS;

ID	USER_ID_OWING_FK	PRODUCT_NAME
1	1	IceCream
2	1	ColdDrinks

(2 rows, 1 ms)

localhost:8080/api/user/1

GET localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

Key
Key

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "userId": 1,
```

```
    "name": "JohnXYZ",
    "phone": "1234567890",
    "orderDetails": [
        {
            "id": 1,
            "productName": "IceCream",
            "userDetails": 1
        },
        {
            "id": 2,
            "productName": "ColdDrinks",
            "userDetails": 1
        }
    ]
}
```


Many-to-One:Unidirectional

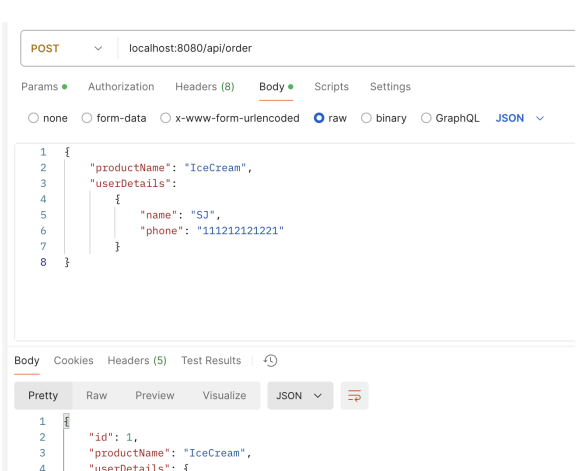
- In this, we generally talk from the perceptive of Child like : Many Orders can be placed by 1 User.
- So still, User is considered as Parent and Orders as Child.
- Parent don't have reference to child.
- Each child has reference to parent.

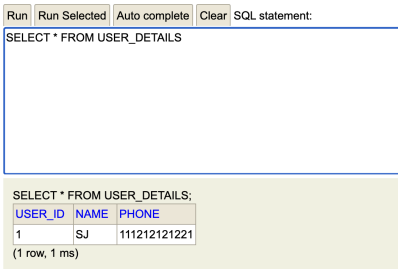
```java
@RestController
@RequestMapping(value = "/api/")
public class OrderController {

    @Autowired
    OrderService orderService;

    @GetMapping("/order/{id}")
    public OrderDetails fetchUser(@PathVariable Long id) {
        return orderService.findByID(id);
    }

    @PostMapping(path = "/order")
    public OrderDetails insertOrder(@RequestBody OrderDetails orderDetails) {
        return orderService.saveOrder(orderDetails);
    }
}
```

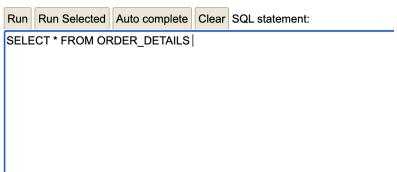
```java
@Service
public class OrderService {

    @Autowired
    OrderDetailsRepository orderDetailsRepository;

    public OrderDetails findByID(Long primaryKey) {
        return  orderDetailsRepository.findById(primaryKey).get();
    }

    public OrderDetails saveOrder(OrderDetails order) {
        return orderDetailsRepository.save(order);
    }
}
```

```java
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;
    private String name;
    private String phone;

  //getters and setters
}
```

```java
@Table(name = "order_details")
@Entity
public class OrderDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String productName;
    @ManyToOne (cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id_owing_fk", referencedColumnName = "userID")
    private UserDetails userDetails;

    //getter and setter
}
```







```
    "userId": 1,
    "name": "SJ",
    "phone": "111212121221"
}
```

SELECT * FROM ORDER_DETAILS;

ID	USER_ID_OWING_FK	PRODUCT_NAME
1	1	IceCream

(1 row, 1 ms)

Try it out:
Many-to-One: Bidirectional would be same as OneToMany Bidirectional

Many-to-Many: Unidirectional

- Reference from One way only

```java
@Table(name = "order_details")
@Entity
public class OrderDetails {

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


    @ManyToMany (cascade = CascadeType.ALL)
    @JoinTable(
            name = "order_product", // new Join table name
            joinColumns = @JoinColumn(name = "order_id"), // Foreign key for Order
            inverseJoinColumns = @JoinColumn(name = "product_id") // Foreign key for Product
    )
    private List<ProductDetails> productDetails = new ArrayList<>();

    public Long getOrderNo() {
        return orderNo;
    }

    public void setOrderNo(Long orderNo) {
        this.orderNo = orderNo;
    }

    public List<ProductDetails> getProductDetails() {
        return productDetails;
    }

    public void setProductDetails(List<ProductDetails> productDetails) {
        this.productDetails = productDetails;
    }
}
```

```java
@Table(name = "product_details")
@Entity
public class ProductDetails {

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

    private String name;
    private double price;

   //getters and setters
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM ORDER_DETAILS
```

SELECT * FROM ORDER_DETAILS;

ORDER_NO

(no rows, 1 ms)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM PRODUCT_DETAILS
```

SELECT * FROM PRODUCT_DETAILS;

PRICE	PRODUCT_ID	NAME

(no rows, 0 ms)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM ORDER_PRODUCT
```

SELECT * FROM ORDER_PRODUCT;

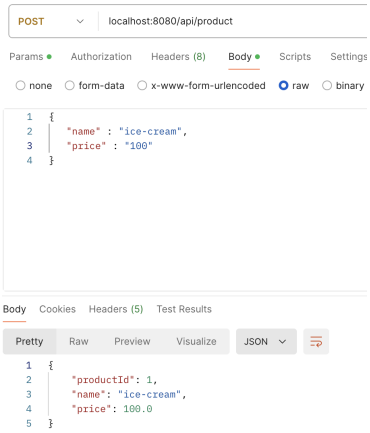
ORDER_ID	PRODUCT_ID

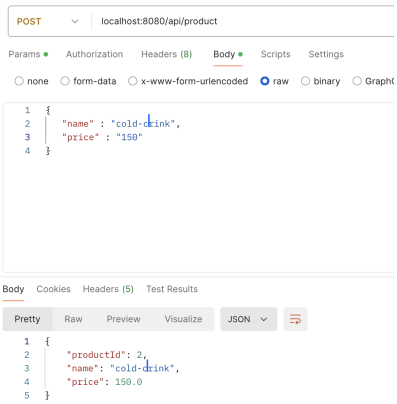
(no rows, 1 ms)

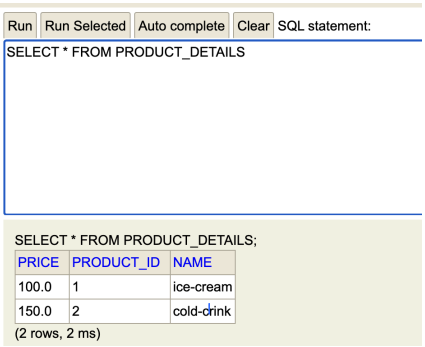
```java
@RestController
@RequestMapping(value = "/api/")
public class ProductController {

    @Autowired
    ProductService productService;

    @PostMapping(path = "/product")
    public ProductDetails insertUser(@RequestBody ProductDetails product) {
        return productService.saveProduct(product);
    }
}
```

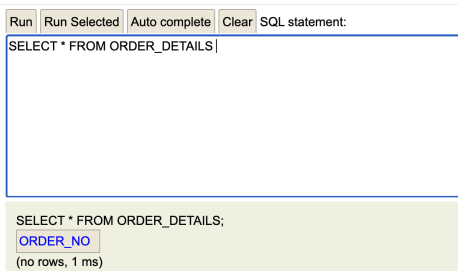
```java
@RestController
@RequestMapping(value = "/api/")
public class OrderController {

    @Autowired
    OrderService orderService;

    @Autowired
    ProductService productService;

    @GetMapping("/order/{id}")
    public OrderDetails fetchUser(@PathVariable Long id) {
        return orderService.findByID(id);
    }

    @PostMapping(path = "/order")
    public OrderDetails insertOrder(@RequestBody OrderDetails orderDetail) {

        List<ProductDetails> managedProducts = orderDetail.getProductDetails().stream()
                .map(product -> productService.findByID(product.getProductId()))
                .collect(Collectors.toList());

        orderDetail.setProductDetails(managedProducts);
        return orderService.saveOrder(orderDetail);
    }

}
```

1st : CREATE PRODUCTS

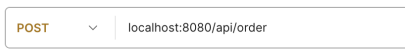


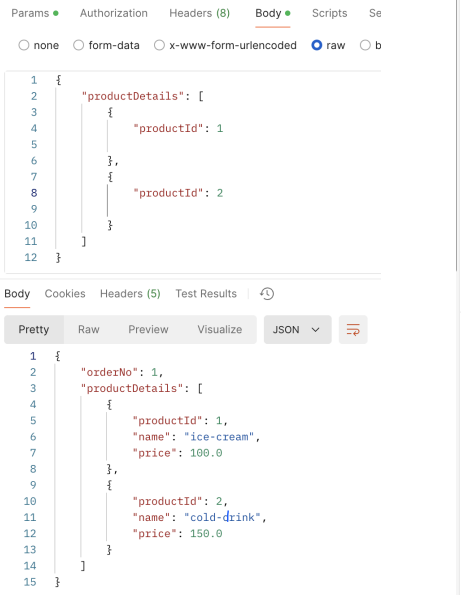


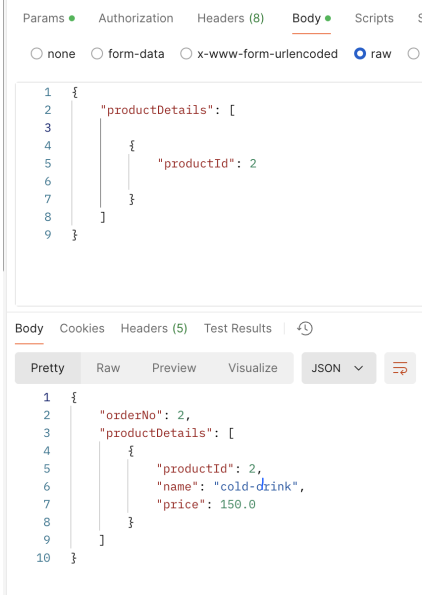


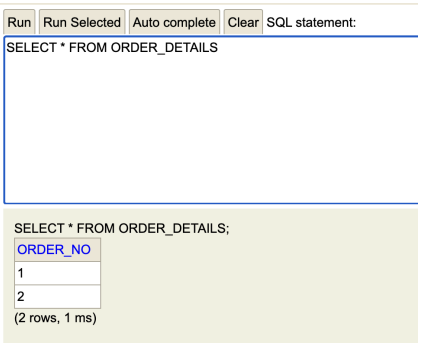


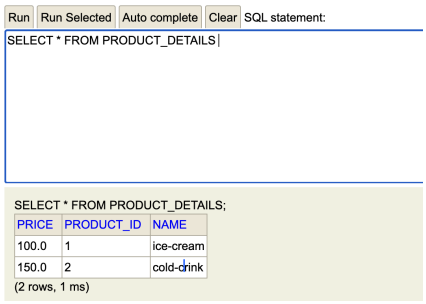
2nd : CREATE Multiple Orders

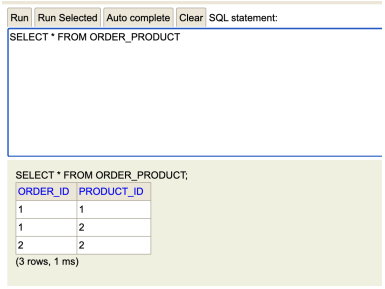












Many-to-Many: Bidirectional

. Since its Many to Many, anyone can be Owing and inverse side.

```java
@Table(name = "order_details")
@Entity
public class OrderDetails {

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

    @ManyToMany (cascade = CascadeType.ALL)
    @JoinTable(
            name = "order_product", // new Join table name
            joinColumns = @JoinColumn(name = "order_id"), // Foreign key for Order
            inverseJoinColumns = @JoinColumn(name = "product_id") // Foreign key for Product
    )
    private List<ProductDetails> productDetails = new ArrayList<>();

    public Long getOrderNo() {
        return orderNo;
    }

    public void setOrderNo(Long orderNo) {
        this.orderNo = orderNo;
    }

    public List<ProductDetails> getProductDetails() {
        return productDetails;
    }

    public void setProductDetails(List<ProductDetails> productDetails) {
        this.productDetails = productDetails;
    }
}
```

```java
@Table(name = "product_details")
@Entity
public class ProductDetails {

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

    private String name;
    private double price;

    @ManyToMany(mappedBy = "productDetails")
    @JsonIgnore
    List<OrderDetails> orders = new ArrayList<>();

    //GETTERS AND SETTERS
}
```

Till now, our *Repository interface* looks like this

```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {


}
```

And in service class, we used to invoke methods which are available in JPA framework

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails findByID(Long primaryKey) {
        return userDetailsRepository.findById(primaryKey).get();
    }

}
```

Then, we have something called: **Derived Query**

- Automatically generates queries from the methods.
- Need to follow a specific naming convention.
- Derived query used for GET/REMOVE operations but not for INSERT/UPDATE
    - Insert and Update operations is supported though "save()"

PartTree.java

```
private static final String QUERY_PATTERN = "find|read|get|query|search|stream";
private static final String COUNT_PATTERN = "count";
private static final String EXISTS_PATTERN = "exists";
private static final String DELETE_PATTERN = "delete|remove";
private static final Pattern PREFIX_TEMPLATE = Pattern.compile( //
        "^(" + QUERY_PATTERN + "|" + COUNT_PATTERN + "|" + EXISTS_PATTERN + "|" + DELETE_PATTERN + ")((\\p{Lu}.*?))??By");
```

"^(findIreadIgetIqueryIsearchIstreamIcountIexistsIdeleteIremove)((\\p{Lu}.*?))??By"

Method name should start with either One of these values : find or read or get etc..

0 or More characters

Uppercase Letter (ex: A,B,C etc..)

'By' at the end of the String

```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {


    List<UserDetails> findUserDetailsByName(String userName);
}
```

```
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;
```

```
  //getters and setters
}
```

Query in which it get translates too:

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
```

Different Use cases:

And:

```
List<UserDetails> findUserDetailsByNameAndPhone(String userName, String phone);
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
        and ud1_0.phone=?
```

Or:

```
List<UserDetails> findUserDetailsByNameAndPhoneOrUserId(String userName, String phone, Long id);
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
        and ud1_0.phone=?
        or ud1_0.user_id=?
```

Comparison:

Part.java

```
BETWEEN(2, "IsBetween", "Between"),
IS_NOT_NULL(0, "IsNotNull", "NotNull"),
IS_NULL(0, "IsNull", "Null"),
LESS_THAN("IsLessThan", "LessThan"),
LESS_THAN_EQUAL("IsLessThanEqual", "LessThanEqual"),
GREATER_THAN("IsGreaterThan", "GreaterThan"),
GREATER_THAN_EQUAL("IsGreaterThanEqual", "GreaterThanEqual"),
BEFORE("IsBefore", "Before"),
AFTER("IsAfter", "After"),
NOT_LIKE("IsNotLike", "NotLike"),
LIKE("IsLike", "Like"),
STARTING_WITH("IsStartingWith", "StartingWith", "StartsWith"),
ENDING_WITH("IsEndingWith", "EndingWith", "EndsWith"),
IS_NOT_EMPTY(0, "IsNotEmpty", "NotEmpty"),
IS_EMPTY(0, "IsEmpty", "Empty"),
NOT_CONTAINING("IsNotContaining", "NotContaining", "NotContains"),
CONTAINING("IsContaining", "Containing", "Contains"),
NOT_IN("IsNotIn", "NotIn"),
IN("IsIn", "In"),
NEAR("IsNear", "Near"),
WITHIN("IsWithin", "Within"),
REGEX("MatchesRegex", "Matches", "Regex"),
EXISTS(0, "Exists"),
TRUE(0, "IsTrue", "True"),
FALSE(0, "IsFalse", "False"),
NEGATING_SIMPLE_PROPERTY("IsNot", "Not"),
SIMPLE_PROPERTY("Is", "Equals");
```

```
List<UserDetails> findUserDetailsByNameIsIn(List<String> userName);
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name in (?)
```

```
List<UserDetails> findUserDetailsByNameLike(String userName);
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name like ? escape '\'
```

Delete:

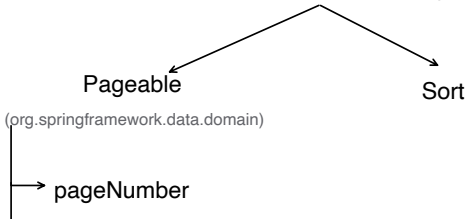
- Need to add @Transactional annotation.

```
@Transactional
void deleteByName(String userName);
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
Hibernate:
    delete
    from
        user_details
    where
        user_id=?
Hibernate:
    delete
    from
        user_details
    where
        user_id=?
Hibernate:
    delete
    from
        user_details
    where
        user_id=?
```

Paginations and Sorting in Derived Query:

. JPA provides 2 interfaces to support Pagination and Sorting i.e.



→ pageSize (no of records per page)

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    List<UserDetails> findUserDetailsByNameStartingWith(String userName, Pageable page);

}
```

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public List<UserDetails> findByNameDerived(String name) {
        Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 5); // Page 0, 5 records per page
        return userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    }
}
```

If we need more info about Pages, then we can use "Page" as return type

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    Page<UserDetails> findUserDetailsByNameStartingWith(String userName, Pageable page);

}
```

```java
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 5); // Page 0, 5 records per page
    Page<UserDetails> userDetailsPage = userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    List<UserDetails> userDetailsList = userDetailsPage.getContent();
    System.out.println("total pages: " + userDetailsPage.getTotalPages());
    System.out.println("is first page: " + userDetailsPage.isFirst());
    System.out.println("is last page: " + userDetailsPage.isLast());
    return userDetailsList;
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

USER_ID	PHONE	USER_NAME
1	12312	A
2	12312	AB
3	12312	ABC
4	12312	ABCD
5	12312	ABCDE
6	12312	ABCDEF

(6 rows, 3 ms)

GET localhost:8080/api/user/byname_derived/A

Params Authorization Headers (6) Body Scripts Settings

Query Params

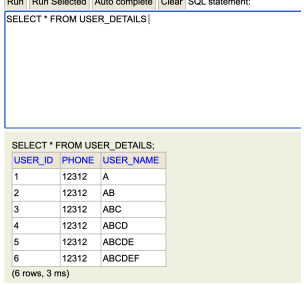
Key	Value
Key	Value

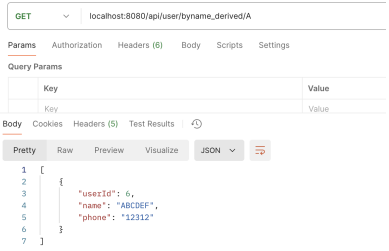
Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "userId": 1,
        "name": "A",
        "phone": "12312"
    },
    {
        "userId": 2,
        "name": "AB",
        "phone": "12312"
    },
    {
        "userId": 3,
        "name": "ABC",
        "phone": "12312"
    },
    {
        "userId": 4,
        "name": "ABCD",
        "phone": "12312"
    },
    {
        "userId": 5,
        "name": "ABCDE",
        "phone": "12312"
    }
]
```


```java
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 1,  pageSize: 5); // Page 0, 5 records per page
    Page<UserDetails> userDetailsPage = userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    List<UserDetails> userDetailsList = userDetailsPage.getContent();
    System.out.println("total pages: " + userDetailsPage.getTotalPages());
    System.out.println("is first page: " + userDetailsPage.isFirst());
    System.out.println("is last page: " + userDetailsPage.isLast());
    return userDetailsList;
}
```

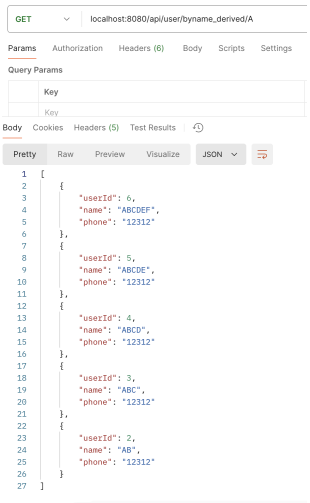





```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name like ? escape '\'
    offset
        ? rows
    fetch
        first ? rows only
total pages: 2
is first page: false
is last page: true
```

Paginations with Sorting:

```java
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 5, Sort.by( ...properties: "name").descending()); // Page 0, 5 records per page
    Page<UserDetails> userDetailsPage = userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    List<UserDetails> userDetailsList = userDetailsPage.getContent();
    System.out.println("total pages: " + userDetailsPage.getTotalPages());
    System.out.println("is first page: " + userDetailsPage.isFirst());
    System.out.println("is last page: " + userDetailsPage.isLast());
    return userDetailsList;
}
```



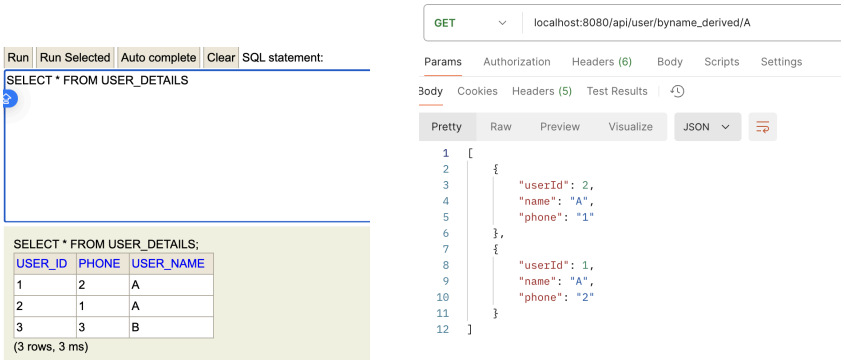
Only Sorting:

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    List<UserDetails> findUserDetailsByNameStartingWith(String userName, Sort sort);

}
```

```java
public List<UserDetails> findByNameDerived(String name) {
    return userDetailsRepository.findUserDetailsByNameStartingWith(name, Sort.by( ...properties: "name").descending());
}
```

- Sort.by accepts multiple fields.
- When multiple fields provided, sorting applied in order.
- first it sort by first field and if there are duplicates then second field is used and so on.

```
public List<UserDetails> findByNameDerived(String name) {
    return userDetailsRepository.findUserDetailsByNameStartingWith(name, Sort.by( ...properties: "name", "phone").ascending());
}
```



- If we need different sorting order for different fields

```
public List<UserDetails> findByNameDerived(String name) {

    Sort sort = Sort.by(
            Sort.Order.asc( property: "name"),
            Sort.Order.desc( property: "phone")
    );
    return userDetailsRepository.findUserDetailsByNameStartingWith(name, sort);
}
```

Queries which are little complex and can't be handled via Derived Query, we can use:

JPQL:

- Java Persistence Query Language.
- Similar to SQL but works on ***Entity Object*** instead of direct database.
  - Its database independent

- Works with Entity name and fields and not with table column names.

Syntax:

There is no strict rule for Return type:
- you can return List or
-Single object
But, if say there are more than one rows, but in return type, we return Single Object, then JPQL will throw an exception

JPQL query with JOIN

- OneToOne

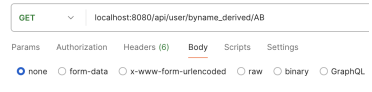
```java
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_address")
    private UserAddress userAddress;

   //getters and setters
}
```

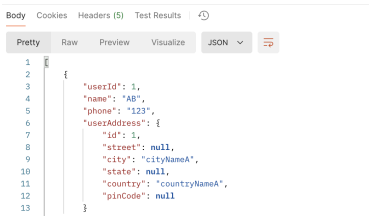
```java
@Entity
@Table(name = "user_address")
public class UserAddress {

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

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    //getters and setters
}
```

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query("SELECT ud FROM UserDetails ud JOIN ud.userAddress ad WHERE ud.name = :userFirstName")
    List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);

}
```

We don't specifically need to put "On" here, JPA will automatically do that

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ud1_0.user_address
    from
        user_details ud1_0
    join
        user_address ua1_0
            on ua1_0.id=ud1_0.user_address
    where
        ud1_0.user_name=?
```





```
14  }
15  }
```

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query("SELECT ud.name, ad.country FROM UserDetails ud JOIN ud.userAddress ad WHERE ud.name = :userFirstName")
    List<Object[]> findUserDetailsWithAddress(@Param("userFirstName") String userName);

}
```

```java
public class UserDTO {

    String userName;
    String country;

    // Constructor to populate from UserDetails entity
    public UserDTO(String userName, String country) {
       this.userName = userName;
       this.country = country;
    }

    //getters and setters
}
```

```java
public   List<UserDTO> findByNameDerived(String name) {
    List<Object[]> dbOutput =  userDetailsRepository.findUserDetailsWithAddress(name);
     List<UserDTO> output = new ArrayList<>();
     for(Object[] val : dbOutput) {
         String userName = (String) val[0];
         String country = (String) val[1];
         UserDTO dto = new UserDTO(userName, country);
         output.add(dto);
     }
     return output;

}
```

If we don’t, want Object[] to be used, we can also return direct custom DTO

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query("SELECT new com.conceptandcoding.learningspringboot.jpa.DTO.UserDTO(ud.name, ad.country) FROM UserDetails ud JOIN ud.userAddress ad WHERE ud.name = :userFirstName")
    List<UserDTO> findUserDetailsWithAddress(@Param("userFirstName") String userName);

}
```

- OneToMany

```java
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id") //fk in user address table
    private List<UserAddress> userAddressList = new ArrayList<>();

   //getters and setters
}
```

```java
@Entity
@Table(name = "user_address")
public class UserAddress {

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

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    //getters and setters
}
```

```java
@Query("SELECT ud FROM UserDetails ud JOIN ud.userAddressList ad WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);
```

N+1 Problem and its Solution:

Problem :

Say, 1 User can have Many Addresses.
And our Query is such that, it can fetch more than 1 Users. Then this problem can occurs.

So, say we have 'N' Users. Then below queries will be hit by JPA:

- 1 query to fetch all the USERS.
- For each User it will fetch ADDRESSES, so for N users, it will fetch N times.

So total number of query hit : N+1.

So we need to find the way, so that only 1 QUERY it hit instead of N+1.

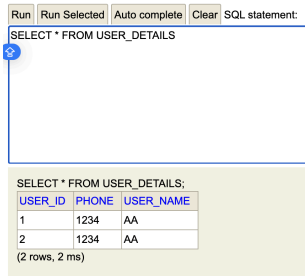
Before going for the solution for this problem, One question might be coming to our mind:

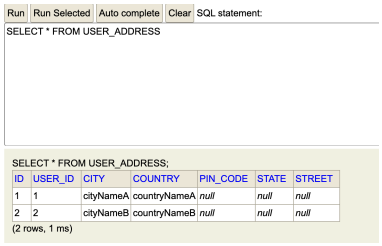
What if, we use EAGER initialization, then can we avoid this issue?

NO because EAGER initialization do not work, when our query tries to fetch multiple PARENT rows and that also have multiple CHILD.

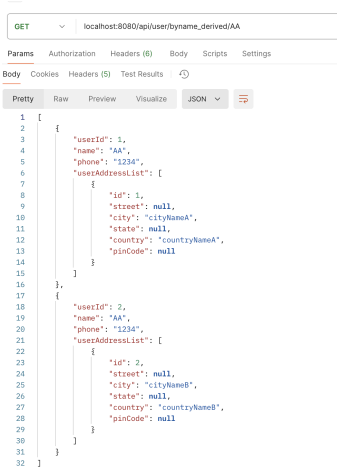
In previous video, we tested EAGER with ***"findByID(id)"*** method, in which it make sure that, our query is fetching only 1 PARENT and that can have many CHILD, that's fine. In that JPA internally draft a JOIN query.

But when Multiple parent with Multiple child get involved, EAGER do not work in just 1 query, it first fetches all the parent and then for each parent, it fetch all its child.






```
@Query("SELECT ud FROM UserDetails ud JOIN ud.userAddressList ad WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);
```




```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    join
        user_address ua1_0
            on ud1_0.user_id=ua1_0.user_id
    where
        ud1_0.user_name=?
Hibernate:
    select
        ua1_0.user_id,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_address ua1_0
    where
        ua1_0.user_id=?
Hibernate:
    select
        ua1_0.user_id,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_address ua1_0
    where
        ua1_0.user_id=?
```

1 query to fetch all users with Name "AA".
So it will return 2 users.

For each user
Its fetching all its addresses.

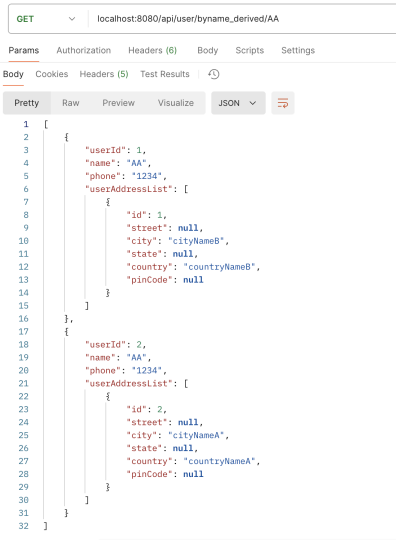
So for 2 users,
2 select query on child table

So, how to solve this, N+1 problem?

Solution1: using ***JOIN FETCH***
(JPQL)

```
@Query("SELECT ud FROM UserDetails ud JOIN FETCH ud.userAddressList ad WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ual1_0.user_id,
        ual1_0.id,
        ual1_0.city,
        ual1_0.country,
        ual1_0.pin_code,
        ual1_0.state,
        ual1_0.street
    from
        user_details ud1_0
    join
        user_address ual1_0
            on ud1_0.user_id=ual1_0.user_id
    where
        ud1_0.user_name=?
```



Solution2: using ***@BatchSize(size=10)***

- It wont make only 1 query, but it will reduce it, as it will divide it into batches

```
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @BatchSize(size = 10)
    @JoinColumn(name = "user_id") //fk in user address table
    private List<UserAddress> userAddressList;

   //getters and setters
}
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    join
        user_address ual1_0
            on ud1_0.user_id=ual1_0.user_id
    where
        ud1_0.user_name=?
Hibernate:
    select
        ual1_0.user_id,
        ual1_0.id,
        ual1_0.city,
        ual1_0.country,
        ual1_0.pin_code,
        ual1_0.state,
        ual1_0.street
    from
        user_address ual1_0
    where
        ual1_0.user_id in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

Solution3: using ***@EntityGraph(attributePaths="userAddressList")***

- Used over method (helpful in derived methods)
- Tell JPA to fetch all the entries of UserAddress along with user details.

```
@EntityGraph(attributePaths = "userAddressList")
List<UserDetails> findUsersBy();
```

How to join Many tables?

Its almost same as SQL only

Say, we have
Table A has one to many relationship with Table B
Table B has one to many relationship with Table C

```
@Query("SELECT a FROM A a  JOIN a.bList b  JOIN b.cList c WHERE c.someProperty =
:someValue")
List<A> findAWithBAndC(@Param("someValue") String someValue);
```

@Modifying Annotation

- when @Query annotation used, by-default JPA expects ***SELECT*** query.
- If we try to use "DELETE" or "INSERT" or "UPDATE" query with @Query, JPA will throw error, that:

```
query.IllegalSelectQueryException Create breakpoint : Expecting a SELECT Query [org.hibernate.query.sqm.tree.select.SqmSelectStatement],
ernate.query.sqm.internal.SqmUtil.verifyIsSelectStatement(SqmUtil.java:109) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
ernate.query.sqm.internal.QuerySqmImpl.verifySelect(QuerySqmImpl.java:494) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
```

- @Modifying annotation, is to tell JPA that, expect either "DELETE" or "INSERT" or "UPDATE" query with @Query
- Since we are trying to update the DB, we also need to use @Transactional annotation.

```
@Modifying
@Transactional
@Query("DELETE FROM UserDetails ud WHERE ud.name = :userFirstName")
void deleteByUserName(@Param("userFirstName") String userName);
```

Understanding Usage of Flush and Clear:

- **As we know, Flush just pushed the persistence context changes to DB but hold the value in persistence context.**
- **Clear, purge the persistence context, and required fresh DB call**

```
@Modifying
@Query("DELETE FROM UserDetails ud WHERE ud.name = :userFirstName")
void deleteByUserName(@Param("userFirstName") String userName);
```

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    @Transactional
    public void deleteByUserName(String name) {
        userDetailsRepository.findById(1L).get();
        userDetailsRepository.deleteByUserName(name);
        Optional<UserDetails> output = userDetailsRepository.findById(1L);
        System.out.println("output present: " + output.isPresent());

    }
}
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_details ud1_0
    left join
        user_address ua1_0
```

```
            on ua1_0.id=ud1_0.user_address_id
    where
        ud1_0.user_id=?
Hibernate:
    delete
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
output present: true
```

Now using, Flush and Clear

```
@Modifying(flushAutomatically = true, clearAutomatically = true)
@Query("DELETE FROM UserDetails ud WHERE ud.name = :userFirstName")
void deleteByUserName(@Param("userFirstName") String userName);
```

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    @Transactional
    public void deleteByUserName(String name) {
        userDetailsRepository.findById(1L).get();
        userDetailsRepository.deleteByUserName(name);
        Optional<UserDetails> output = userDetailsRepository.findById(1L);
        System.out.println("output present: " + output.isPresent());

    }
}
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_details ud1_0
    left join
        user_address ua1_0
            on ua1_0.id=ud1_0.user_address_id
    where
        ud1_0.user_id=?
Hibernate:
    delete
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_details ud1_0
    left join
        user_address ua1_0
            on ua1_0.id=ud1_0.user_address_id
    where
        ud1_0.user_id=?
output present: false
```

Pagination and Sorting in JPQL

Same like discussed in derived query method

```
@Query("SELECT ud FROM UserDetails ud WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetails(@Param("userFirstName") String userName, Pageable pageable);
```

```
public List<UserDetails> findByUserName(String name) {
    Pageable page = PageRequest.of( pageNumber: 1, pageSize: 5);
    return userDetailsRepository.findUserDetails(name, page);
}
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ud1_0.user_address_id
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
    offset
        ? rows
    fetch
        first ? rows only
```

@NamedQuery Annotation

- We can name our Query, so that we can reuse it.

```
@Table(name = "user_details")
@Entity
@NamedQuery(name = "findByUserName",
        query = "SELECT u FROM UserDetails u WHERE u.name = :userFirstName")
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    private UserAddress userAddress;

    //getters and setters
}
```

```
@Query(name = "findByUserName")
List<UserDetails> findUserDetails(@Param("userFirstName") String userName, Pageable pageable);
```

Native Query:

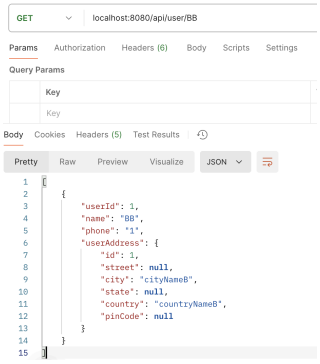
- Plain SQL queries.
- Directly interact with Database, thus if in future DB changes, code changes also required.
- No caching, lazy loading or entity life cycle management happens.

When to use over JPQL:

- More complex queries, including database specific features like JSONB, LATERAL JOIN
- Need to fetch non-entity results or joins table without any entity relationship.
- Query efficiency like Bulk operations.

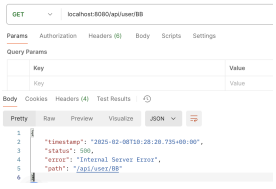
```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query(value = "SELECT * FROM user_details WHERE user_name = :userFirstName", nativeQuery = true)
    List<UserDetails> getUserDetailsByNameNativeQuery(@Param("userFirstName") String userName);

}
```

When all the fields (*) of the table are returned by Native Query, JPA internally does the mapping between DB column name and Entity fields.



But, when Native Query returned partial fields, then JPA don't map it to Entity by default.

```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query(value = "SELECT user_name, phone FROM user_details WHERE user_name = :userFirstName", nativeQuery = true)
    List<UserDetails> getUserDetailsByNameNativeQuery(@Param("userFirstName") String userName);

}
```




```
org.h2.jdbc.JdbcSQLSyntaxErrorException Create breakpoint : Column "user_id" not found [42122-224]
    at org.h2.message.DbException.getJdbcSQLException(DbException.java:514) ~[h2-2.2.224.jar:2.2.224]
    at org.h2.message.DbException.getJdbcSQLException(DbException.java:489) ~[h2-2.2.224.jar:2.2.224]
    at org.h2.message.DbException.get(DbException.java:223) ~[h2-2.2.224.jar:2.2.224]
    at org.h2.message.DbException.get(DbException.java:199) ~[h2-2.2.224.jar:2.2.224]
    at org.h2.jdbc.JdbcResultSet.getColumnIndex(JdbcResultSet.java:3518) ~[h2-2.2.224.jar:2.2.224]
    at org.h2.jdbc.JdbcResultSet.findColumn(JdbcResultSet.java:178) ~[h2-2.2.224.jar:2.2.224]
```

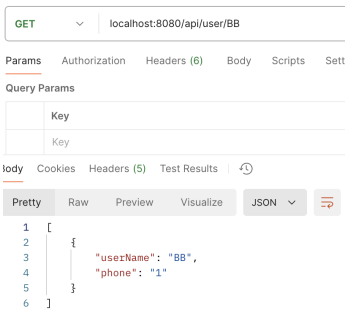
We need to manually tell JPA, how to do the mapping.

1st: Using **@SqlResultSetMapping and @NamedNativeQuery** Annotation

```java
@Table(name = "user_details")
@Entity
@NamedNativeQuery(
        name = "UserDetails.getUserDetailsByName",
        query = "SELECT user_name, phone FROM user_details WHERE user_name = :userFirstName",
        resultSetMapping = "UserDTOMapping"
)
@SqlResultSetMapping(
        name = "UserDTOMapping",
        classes = @ConstructorResult(
                targetClass = UserDTO.class,
                columns = {
                        @ColumnResult(name = "user_name", type = String.class),
                        @ColumnResult(name = "phone", type = String.class)}
        )
)
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    private UserAddress userAddress;

   //getters and setters
}
```

```java
public class UserDTO {

    String userName;
    String phone;

    public UserDTO(String userName, String phone) {
        this.userName = userName;
        this.phone = phone;
    }

    //getters and setters
}
```

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query(name = "UserDetails.getUserDetailsByName", nativeQuery = true)
    List<UserDTO> getUserDetailsByNameNativeQuery(@Param("userFirstName") String userName);

}
```



2nd: With Manual mapping

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query(value = "SELECT user_name, phone FROM user_details WHERE user_name = :userFirstName", nativeQuery = true)
    List<Object[]> getUserDetailsByNameNativeQuery(@Param("userFirstName") String userName);

}
```

```java
public List<UserDTO> getUserDetailsByNameNativeQuery(String name) {
    List<Object[]> results = userDetailsRepository.getUserDetailsByNameNativeQuery(name);
    return results.stream() Stream<Object[]>
            .map(obj -> new UserDTO((String) obj[0], (String) obj[1])) Stream<UserDTO>
            .collect(Collectors.toList());
}
```

Dynamic Native Query:

```java
@Service
public class UserDetailsService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<UserDTO> getUserDetailsByNameNativeQuery(String userName) {
        StringBuilder queryBuilder = new StringBuilder("SELECT ud.user_name AS user_name, ud.phone AS phone, ua.city AS city ");
        queryBuilder.append("FROM user_details ud ");
        queryBuilder.append("JOIN user_address ua ON ud.user_address_id = ua.id ");
        queryBuilder.append("WHERE 1=1 ");

        List<Object> parameters = new ArrayList<>();

        // Dynamically add conditions
        if (userName != null && !userName.isEmpty()) {
            queryBuilder.append("AND ud.user_name = ? ");
            parameters.add(userName);
        }

        // Create the native query
        Query nativeQuery = entityManager.createNativeQuery(queryBuilder.toString());

        // Set the parameters for the query
```

SELECT ud.user_name AS user_name, ud.phone AS phone, ua.city AS city FROM user_details ud JOIN user_address ua ON ud.user_address_id = ua.id WHERE 1=1 AND ud.user_name = ?

```
        for (int i = 0; i < parameters.size(); i++) {
            nativeQuery.setParameter( position: i + 1, parameters.get(i));
        }

        // Execute and get results
        List<Object[]> result = nativeQuery.getResultList();

        // Map the result to UserDTO
        return UserDTO.mapResultToDTO(result);

    }
}
```

Pagination and Sorting in Native SQL:

1st way:

```
public List<UserDTO> getUserDetailsByNameNativeQuery(String userName) {
    StringBuilder queryBuilder = new StringBuilder("SELECT ud.user_name AS user_name, ud.phone AS phone, ua.city AS city ");
    queryBuilder.append("FROM user_details ud ");
    queryBuilder.append("JOIN user_address ua ON ud.user_address_id = ua.id ");
    queryBuilder.append("WHERE 1=1 ");

    List<Object> parameters = new ArrayList<>();

    // Dynamically add conditions
    if (userName != null && !userName.isEmpty()) {
        queryBuilder.append("AND ud.user_name = ? ");
        parameters.add(userName);
    }

    //sorting
    queryBuilder.append("ORDER BY ").append("ud.user_name").append(" DESC");

    //pagination
    int size = 5;
    int page = 0;
    queryBuilder.append(" LIMIT ? OFFSET ? ");
    parameters.add(size);
    parameters.add(page * size);

    // Create the native query
    Query nativeQuery = entityManager.createNativeQuery(queryBuilder.toString());

    // Set the parameters for the query
    for (int i = 0; i < parameters.size(); i++) {
        nativeQuery.setParameter( position: i + 1, parameters.get(i));
    }

    // Execute and get results
    List<Object[]> result = nativeQuery.getResultList();

    // Map the result to UserDTO
    return UserDTO.mapResultToDTO(result);
}
```

2nd way:

```
public List<UserDetails> getUserDetailsByNameNativeQuery(String name) {

    Pageable pageableObj = PageRequest.of( pageNumber: 0,  pageSize: 5, Sort.by( ...properties: "phone").descending());
    return userDetailsRepository.getUserDetailsByNameNativeQuery(name, pageableObj);

}
```

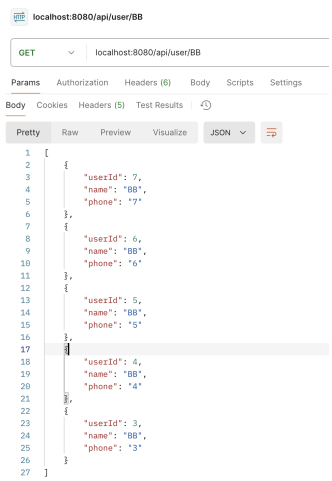
```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query(value = "SELECT * FROM user_details ud WHERE ud.user_name = :userName",
            nativeQuery = true)
    List<UserDetails> getUserDetailsByNameNativeQuery(@Param("userName") String userName, Pageable pageable);

}
```

```
Hibernate:
    SELECT
        *
    FROM
        user_details ud
    WHERE
        ud.user_name = ?
    order by
        ud.phone desc
    fetch
        first ? rows only
```

SELECT * FROM USER_DETAILS;

USER_ID	PHONE	USER_NAME
1	1	BB
2	2	BB
3	3	BB
4	4	BB
5	5	BB
6	6	BB
7	7	BB

(7 rows, 3 ms)

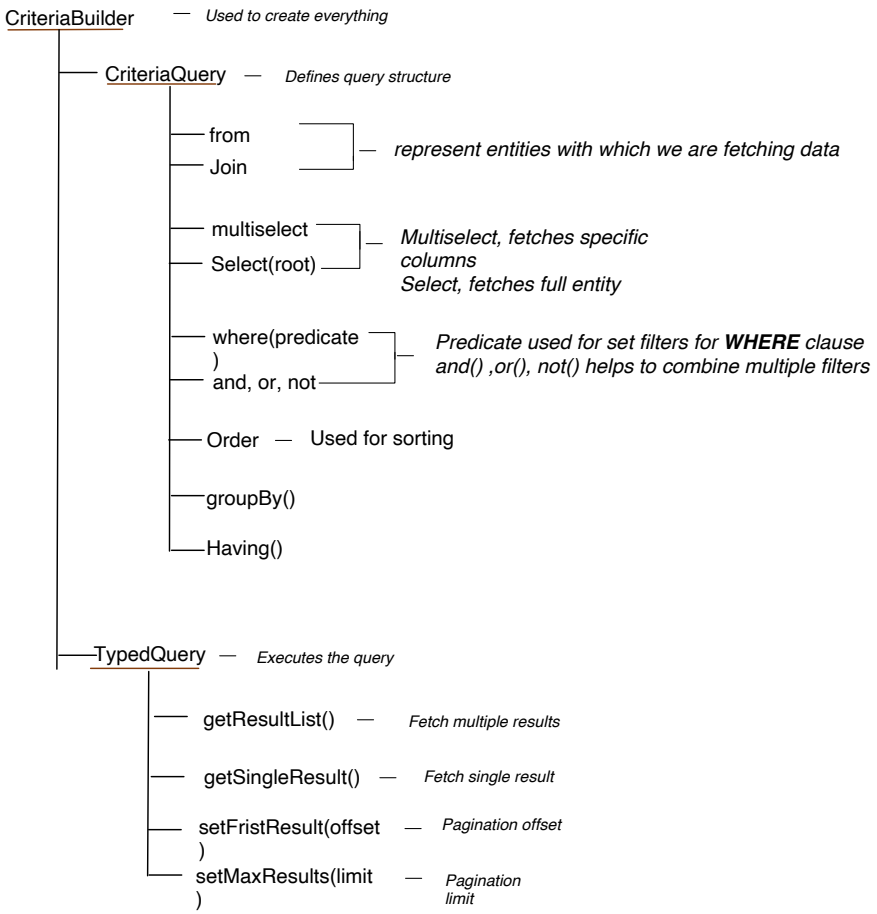


Criteria API:

Native SQL queries support dynamic query building, but they are database-dependent and don't leverage JPA abstraction.

That's why **JPA Criteria API** exists, it allows you to build dynamic, type-safe queries without writing raw SQL.

Lets understand the Hierarchy



Controller class:

```
@GetMapping("/user/{phone}")
public List<UserDetails> getUserDetailsByPhoneCriteriaAPI(@PathVariable Long phone) {
    return userDetailsService.getUserDetailsByPhoneCriteriaAPI(phone);
}
```

Service class:

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    @PersistenceContext
    private EntityManager entityManager;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public List<UserDetails> getUserDetailsByPhoneCriteriaAPI(Long phoneNo) {

        CriteriaBuilder cb = entityManager.getCriteriaBuilder();

        CriteriaQuery<UserDetails> crQuery = cb.createQuery(UserDetails.class); //what my each row would look like, so in this case each row would be UserDetails

        Root<UserDetails> user = crQuery.from(UserDetails.class); // from clause

        crQuery.select(user); //select *

        Predicate predicate = cb.equal(user.get("phone"), phoneNo); // where clause
        crQuery.where(predicate);
```

```
        TypedQuery<UserDetails> query = entityManager.createQuery(crQuery);
        List<UserDetails> output = query.getResultList();

        return output;
    }
}
```

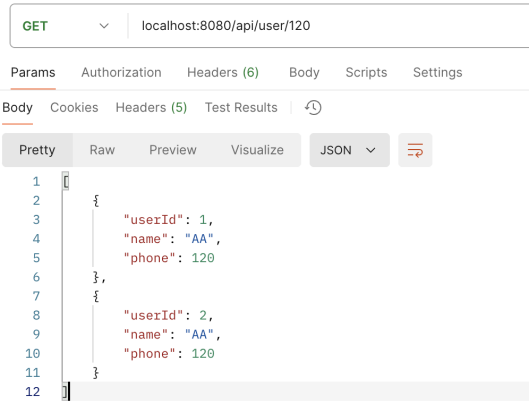
Entity class:

```
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private Long phone;

   //getters and setters
}
```

SELECT * FROM USER_DETAILS;

PHONE	USER_ID	USER_NAME
120	1	AA
120	2	AA
110	3	AA

(3 rows, 3 ms)




```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.phone=?
```

Comparison operator:

Root<UserDetails>user=crQuery.from(UserDetails.class); //from clause

Method	Description	Equivalent SQL query
cb.equal(user.get("phone"),123);	Check for equality	Where phone = 123
cb.notEqual(user.get("phone"),123);	Check for in-equality	Where phone <> 123
cb.gt(user.get("phone"),123);	Greater than	Where phone > 123
cb.ge(user.get("phone"),123);	Greater than or equal	Where phone >= 123
cb.lt(user.get("phone"),123);	Less than	Where phone < 123
cb.le(user.get("phone"),123);	Less than or equal	Where phone <= 123

Logical operator:

Method	Description	Equivalent SQL query
cb.and(predicate1, predicate2);	Combining two conditions using and	Where conditiion1 AND condition2
cb.or(predicate1, predicate2);	Combining two conditions using or	Where conditiion1 OR condition2
cb.not(predicate1,);	Negate the condition	WHERE NOT conditiion1

```
Predicate predicate1 = cb.equal(user.get("phone"), phoneNo); // where clause
Predicate predicate2 = cb.notEqual(user.get("name"), y: "AA"); // where clause
Predicate finalPredicate = cb.and(predicate1, predicate2);

crQuery.where(finalPredicate);
```

Strings Operations:

Method	Description	Equivalent SQL query
cb.like(user.get("name"), "S%");	Name starts with J	Where name LIKE 'S%'
cb.notLike(user.get("name"), "S%");	Name do not start with J	Where name NOT LIKE 'S%'

Collection Operations:

Method	Description	Equivalent SQL query
cb.in(user.get("phone")).value(11).value(7); 	Check if phone no is in the list	Where phone IN (11, 7)
cb.not(user.get("phone").in(11,7));	Check if phone no is not in the list	Where phone NOT IN (11, 7)

Select Multiple fields:

```
public List<UserDTO> getUserDetailsByPhoneCriteriaAPI(Long phoneNo) {

    CriteriaBuilder cb = entityManager.getCriteriaBuilder();

    CriteriaQuery<Object[]> crQuery = cb.createQuery(Object[].class); //what my each row would look like

    Root<UserDetails> user = crQuery.from(UserDetails.class); // from clause

    crQuery.multiselect(user.get("name"), user.get("phone")); //select multiple fields

    Predicate predicate1 = cb.equal(user.get("phone"), phoneNo); // where clause
    crQuery.where(predicate1);

    TypedQuery<Object[]> query = entityManager.createQuery(crQuery);
    List<Object[]> results = query.getResultList();

    // Processing results
    List<UserDTO> output = new ArrayList<>();
    for (Object[] row : results) {

        String name = (String) row[0];
        Long phone = (Long) row[1];
        UserDTO result = new UserDTO(name, phone);
        output.add(result);
    }
    return output;
}
```

Join

```
public List<UserDTO> getUserDetailsByPhoneCriteriaAPI(Long phoneNo) {

    CriteriaBuilder cb = entityManager.getCriteriaBuilder();

    CriteriaQuery<Object[]> crQuery = cb.createQuery(Object[].class); //what my each row would look like

    Root<UserDetails> user = crQuery.from(UserDetails.class); // from clause

    Join<UserDetails, UserAddress> address =  user.join( attributeName: "userAddress", JoinType.INNER);

    crQuery.multiselect(user.get("name"), address.get("city")); //select all the files of both the table

    Predicate predicate1 = cb.equal(user.get("phone"), phoneNo); // where clause
    crQuery.where(predicate1);

    TypedQuery<Object[]> query = entityManager.createQuery(crQuery);
    List<Object[]> results = query.getResultList();

    // Processing results
    List<UserDTO> output = new ArrayList<>();
    for (Object[] row : results) {

        String name = (String) row[0];
        String city = (String) row[1];
        UserDTO result = new UserDTO(name, city);
        output.add(result);
    }
    return output;
}
```

Pagination and Sorting

```java
public List<UserDetails> getUserDetailsByPhoneCriteriaAPI(Long phoneNo) {

    CriteriaBuilder cb = entityManager.getCriteriaBuilder();

    CriteriaQuery<UserDetails> crQuery = cb.createQuery(UserDetails.class); //what my each row would look like

    Root<UserDetails> user = crQuery.from(UserDetails.class); // from clause

    crQuery.select(user); //all columns of UserDetails table

    Predicate predicate1 = cb.equal(user.get("phone"), phoneNo); // where clause
    crQuery.where(predicate1);

    // Sorting
    crQuery.orderBy(cb.desc(user.get("name"))); // ORDER BY name DESC


    TypedQuery<UserDetails> query = entityManager.createQuery(crQuery);
    query.setFirstResult(0); //kind of page number or offset
    query.setMaxResults(5); // page size

    List<UserDetails> results = query.getResultList();
    return results;
}
```

SELECT * FROM USER_DETAILS;

PHONE	USER_ID	USER_NAME
1	1	A
1	2	B
1	3	C
1	4	D
1	5	E
1	6	F
1	7	G
1	8	H

GET localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "userId": 8,
        "name": "H",
        "phone": 1
    },
    {
        "userId": 7,
        "name": "G",
        "phone": 1
    },
    {
        "userId": 6,
        "name": "F",
        "phone": 1
    },
    {
        "userId": 5,
        "name": "E",
        "phone": 1
    },
    {
        "userId": 4,
        "name": "D",
        "phone": 1
    }
]
```


Specification API

1st problem it solves is: ***CODE DUPLICITY***

- In Criteria API its possible that, same filter (predicate) used at multiple methods and class, so there is always a challenge of Code Duplicity.

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    @PersistenceContext
    private EntityManager entityManager;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public List<UserDetails> getUserDetailsByPhoneCriteriaAPI(Long phoneNo) {

        CriteriaBuilder cb = entityManager.getCriteriaBuilder();

        CriteriaQuery<UserDetails> crQuery = cb.createQuery(UserDetails.class); //what my each row would look like, so in this case each row would be UserDetails

        Root<UserDetails> user = crQuery.from(UserDetails.class); // from clause

        crQuery.select(user); //select *

        Predicate predicate = cb.equal(user.get("phone"), phoneNo); // where clause
        crQuery.where(predicate);

        TypedQuery<UserDetails> query = entityManager.createQuery(crQuery);
        List<UserDetails> output = query.getResultList();

        return output;
```

Possible that, same Predicate is used in multiple methods or even different class too

Through Specification API, we can solve this:

Specification Interface support following methods

Method	Description	Equivalent SQL query
toPredicate()	Abstract method, for which we need to provide implementation	Where clause
and()	specf1.and(spec2)	Where cond1 AND cond2
or()	specf1.or(spec2)	Where cond1 OR cond2
not()	Specification.not(spec1)	Where NOT condition1

```
public class UserSpecification {

    public static Specification<UserDetails> equalsPhone(Long phoneNo) {

        return (root, query, cb) -> {
            return cb.equal(root.get("phone"), phoneNo);
        };
    }
}
```

Service Class

```
public List<UserDetails> getUserDetailsByPhoneSpecificationAPI(Long phoneNo) {

    CriteriaBuilder cb = entityManager.getCriteriaBuilder();

    CriteriaQuery<UserDetails> crQuery = cb.createQuery(UserDetails.class); //what my each row would look like

    Root<UserDetails> userRoot = crQuery.from(UserDetails.class); // from clause

    crQuery.select(userRoot); //all columns of UserDetails table

    Specification<UserDetails> specification = UserSpecification.equalsPhone(phoneNo);
    Predicate predicate = specification.toPredicate(userRoot, crQuery, cb);
    crQuery.where(predicate);


    TypedQuery<UserDetails> query = entityManager.createQuery(crQuery);
    query.setFirstResult(0); //kind of page number or offset
    query.setMaxResults(5); // page size

    List<UserDetails> results = query.getResultList();
    return results;
}
```

2nd problem it solves is: ***CODE BOILERPLATE***

Even though we have taken out the predicate logic / filtering logic out, still there are so many boiler code present here

All, we need to tell JPA that:

- From Which table we have to fetch the data, including joins
- What all columns
- Filtering in where clause

that's it, JPA should take care of everything like object creation, query building and execution.

JpaSpecificationExecutor

→ Optional<T> findOne (Specification<T> spec);

→ List<T> findAll (Specification<T> spec);

→ Page<T> findAll (Specification<T> spec, Pageable pageable);

→ Boolean exists (Specification<T> spec);

JpaSpecificationExecutor Framework code

```
@Override
public Page<T> findAll(@Nullable Specification<T> spec, Pageable pageable) {

    TypedQuery<T> query = getQuery(spec, pageable);
    return pageable.isUnpaged() ? new PageImpl<>(query.getResultList())
            : readPage(query, getDomainClass(), pageable, spec);
}
```

```
protected <S extends T> TypedQuery<S> getQuery(@Nullable Specification<S> spec, Class<S> domainClass, Sort sort) {

    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<S> query = builder.createQuery(domainClass);

    Root<S> root = applySpecificationToCriteria(spec, domainClass, query);
    query.select(root);

    if (sort.isSorted()) {
        query.orderBy(toOrders(sort, root, builder));
    }

    return applyRepositoryMethodMetadata(entityManager.createQuery(query));
}
```

```
private <S, U extends T> Root<U> applySpecificationToCriteria(@Nullable Specification<U> spec, Class<U> domainClass,
        CriteriaQuery<S> query) {

    Assert.notNull(domainClass, message: "Domain class must not be null");
    Assert.notNull(query, message: "CriteriaQuery must not be null");

    Root<U> root = query.from(domainClass);

    if (spec == null) {
        return root;
    }

    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    Predicate predicate = spec.toPredicate(root, query, builder);

    if (predicate != null) {
        query.where(predicate);
    }

    return root;
}
```

```
public class UserSpecification {

    public static Specification<UserDetails> equalsPhone(Long phoneNo) {

        return (root, query, cb) -> {
            return cb.equal(root.get("phone"), phoneNo);
        };
    }

    public static Specification<UserDetails> likeName(String name) {

        return (root, query, cb) -> {
            return cb.like(root.get("name"), pattern: "%" + name + "%");
        };
    }

    public static Specification<UserDetails> joinAddress() {
        return (root, query, cb) -> {
            Join<UserDetails, UserAddress> address = root.join( attributeName: "userAddress", JoinType.INNER);
            return null;
        };
    }
}
```

```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> , JpaSpecificationExecutor<UserDetails> {

}
```

```
public List<UserDetails> getUserDetailsByPhoneSpecificationAPI() {

    Specification<UserDetails> result = Specification.where(UserSpecification.joinAddress())
            .and(UserSpecification.equalsPhone( phoneNo: 123l))
            .and(UserSpecification.likeName("AA"));

    return userDetailsRepository.findAll(result);
}
```

Compares to Criteria API, Specification API is more clean and has reusable code

```java
public List<UserDetails> getUserDetailsByPhoneCriteriaAPI(Long phoneNo) {

    CriteriaBuilder cb = entityManager.getCriteriaBuilder();

    CriteriaQuery<UserDetails> crQuery = cb.createQuery(UserDetails.class); //what my each row would look like

    Root<UserDetails> user = crQuery.from(UserDetails.class); // from clause
    Join<UserDetails, UserAddress> address = user.join( attributeName: "userAddress", JoinType.INNER);

    crQuery.select(user); //all columns of UserDetails table

    Predicate predicate1 = cb.equal(user.get("phone"),  y: 123); // where clause
    Predicate predicate2 = cb.equal(user.get("name"),  y: "% AA %"); // where clause
    crQuery.where(cb.and(predicate1, predicate2));

    TypedQuery<UserDetails> query = entityManager.createQuery(crQuery);
    return query.getResultList();
}
```