#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.IDENTI
    private Long id;

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

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

USER_DETAILS

ID	USER_ADDRESS_ID	NAME	PHONE

USER_ADDRESS

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET

user_address_id is a FK

- 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 {
```

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

```java
    @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
    @Id
    @GeneratedValue(strategy =
    private Long id;

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

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

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

```
SELECT * FROM USER_DETAILS
```

```
SELECT * FROM USER_DETAILS;
```

ADDRESS_ID	ID	NAME	PHONE

(no rows, 5 ms)

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
}
```

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

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)

```
SELECT * FROM USER_ADDRESS
```

SELECT * FROM USER_ADDRESS;

CITY	COUNTRY	PIN_CODE	STATE	STREET

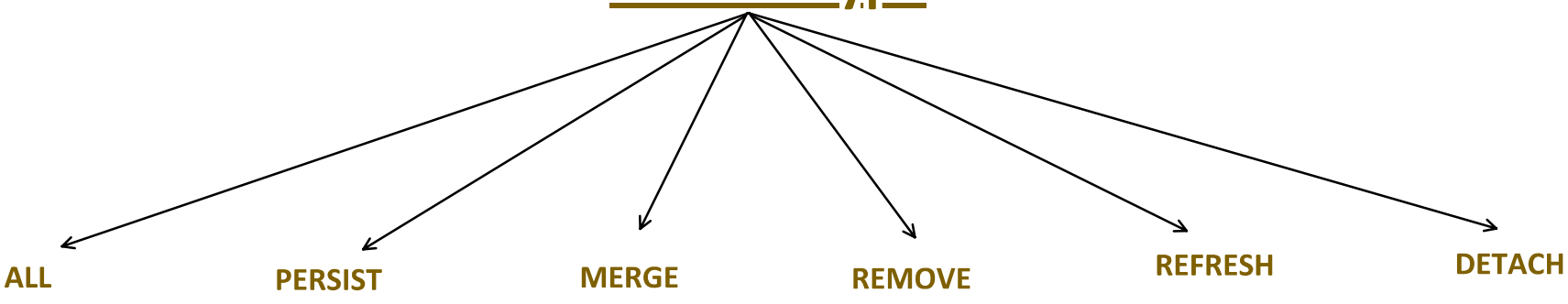
(no rows, 1 ms)

SELECT * FROM INFORMATION_SCHEMA.INDEX_COLUMNS;

INDEX_CATALOG	INDEX_SCHEMA	INDEX_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	ORDERING_SPECI
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

```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;
```

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

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

    private String street;
```

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

    //getters and setters
}
```

```
    private String city;
    private String state;
    private String country;
    private String pinCode;

    //getters and setters
}
```

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

    @Autowired
    UserDetailsService userDetailsService;

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

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

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

```
@Repos
public
}
```

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 GraphQL JSON

```
{
    "name": "JohnXYZ",
    "phone": "1234567890",
    "userAddress": {
        "street": "123 Street",
        "city": "Bangalore",
        "state": "Karnataka",
        "country": "India",
        "pinCode": "10001"
    }
}
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

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

Run Run Selected Auto complete Clear SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ	1234567890

(1 row, 1 ms)

Run Run Selected Auto complete Clear SQL statement:

```
SELECT * FROM USER_ADDRESS
```

SELECT * FROM USER_ADDRESS;

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

(1 row, 1 ms)

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

```
@RestController
```

```
@Service
public class UserDetailsService {
```

```java
@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
    @Autowired
    UserDetailsRepository userDeta

    public UserDetails saveUser(Us
        return userDetailsReposit
    }

    public UserDetails updateUser
        Optional<UserDetails> exis
        if(existingUser.isPresent
            return userDetailsRepo
        }
        return null;
    }
}
```

1st INSERT OPERATION

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 GraphQL JSON

```json
{
    "name": "JohnXYZ",
    "phone": "1234567890",
    "userAddress": {
        "street": "123 Street",
        "city": "Bangalore",
        "state": "Karnataka",
        "country": "India",
        "pinCode": "10001"
    }
}
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

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

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

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ	1234567890

(1 row, 3 ms)

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

```
SELECT * FROM USER_ADDRESS
```

SELECT * FROM USER_ADDRESS;

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

(1 row, 1 ms)

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

localhost:8080/api/user/1

PUT localhost:8080/api/user/1

Params Authorization Headers (8) Body • Scripts Settings

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

```json
{
    "id": 1,
    "name": "JohnXYZ_updated",
    "phone": "1234567890",
    "userAddress": {
        "id": 1,
        "street": "123 Street",
        "city": "Bengaluru",
        "state": "Karnataka",
        "country": "India",
        "pinCode": "10001"
    }
}
```

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

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ_updated	1234567890

(1 row, 1 ms)

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

```
SELECT * FROM USER_ADDRESS
```

SELECT * FROM USER_ADDRESS;

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

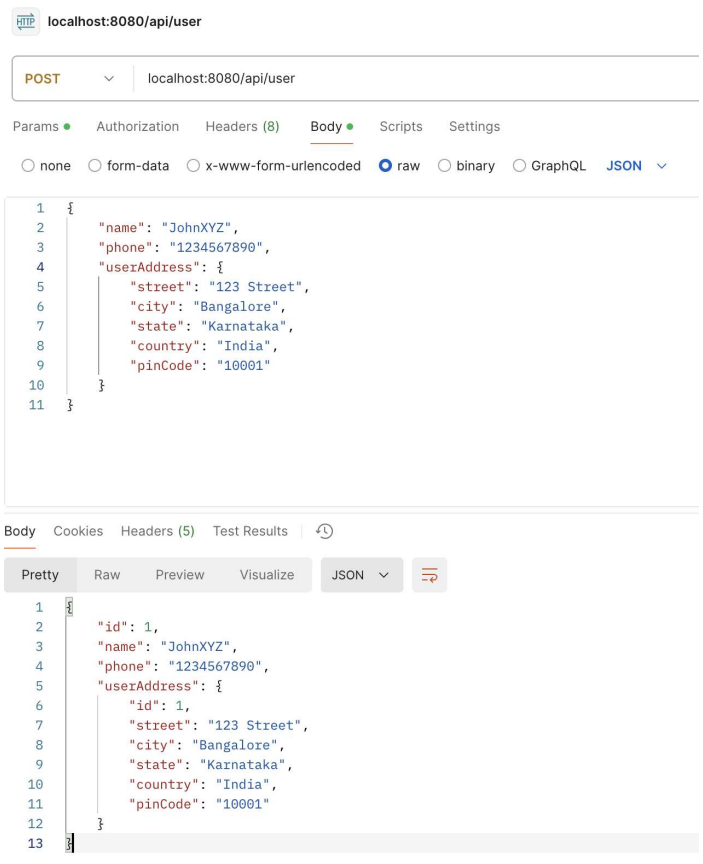
(1 row, 0 ms)

Edit

Now If I want to both INSERT and then UPDATE association capability, means I need both PERSIT + MERG
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.



Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS

SELECT * FROM USER_DETAILS;

ADDRESS_ID	ID	NAME	PHONE
1	1	JohnXYZ	1234567890

(1 row, 3 ms)

Run Run Selected Auto complete Clear SQL statement:

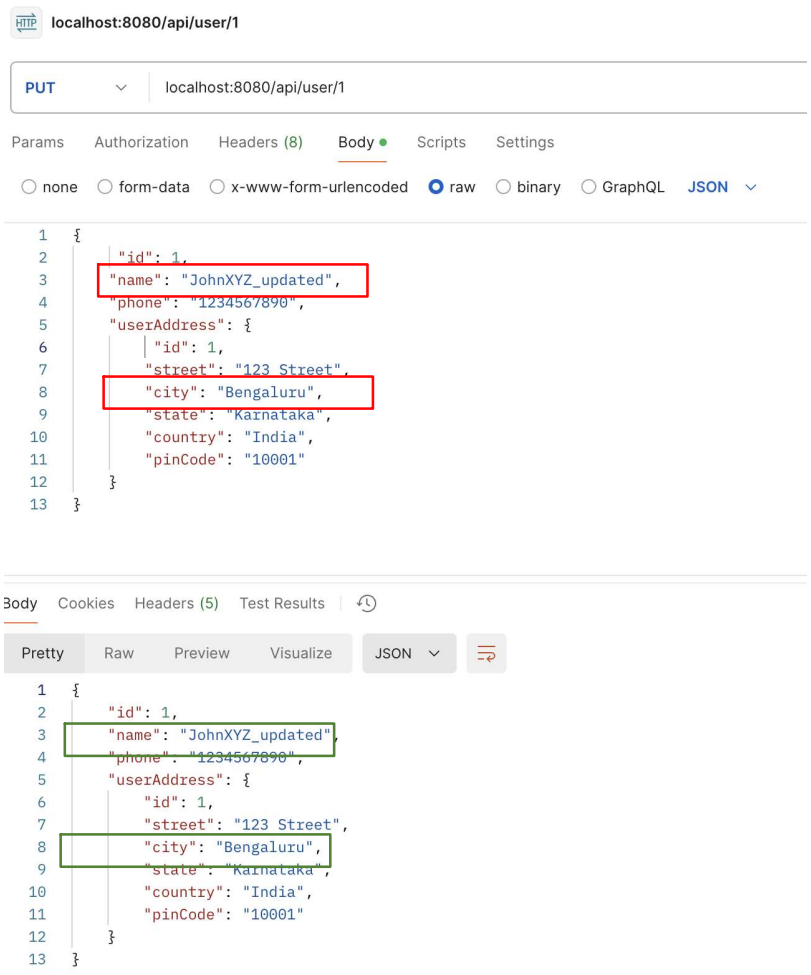
SELECT * FROM USER_ADDRESS

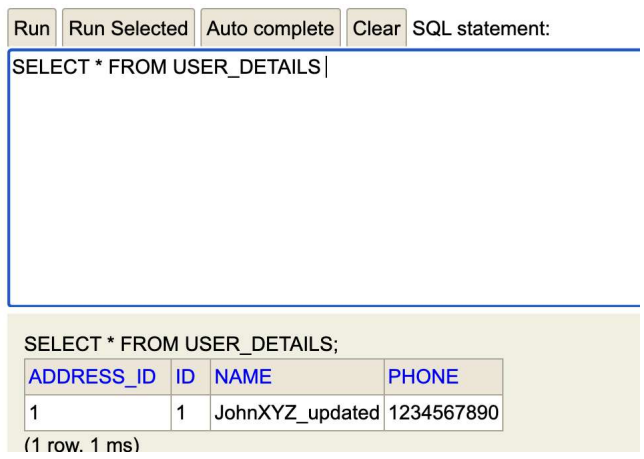
SELECT * FROM USER_ADDRESS;

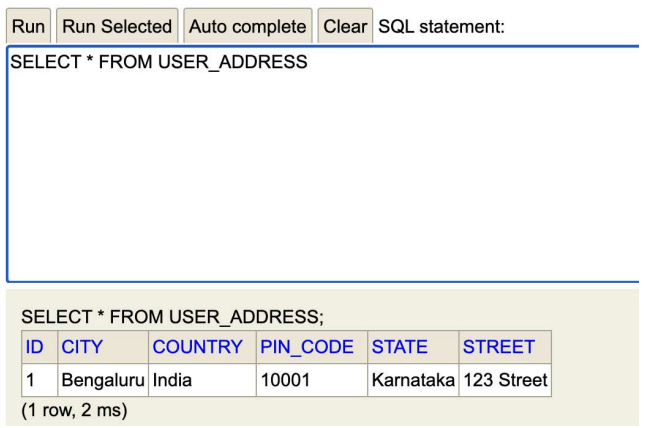
ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	Bangalore	India	10001	Karnataka	123 Street

(1 row, 1 ms)

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();
    }
}
```

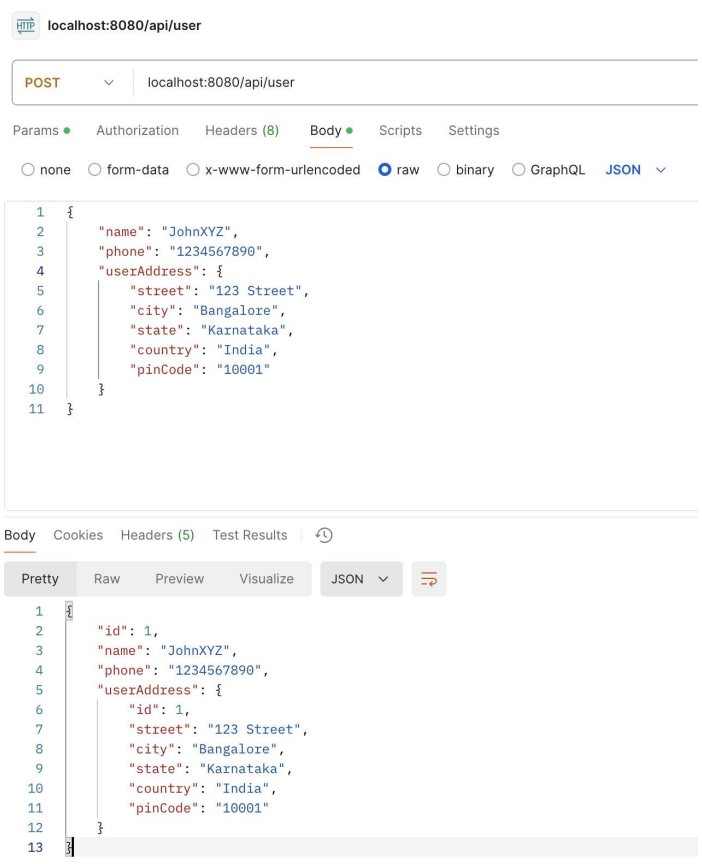
```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepos

    public UserDetails saveUser(UserDetail
        return userDetailsRepository.save(
    }

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

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

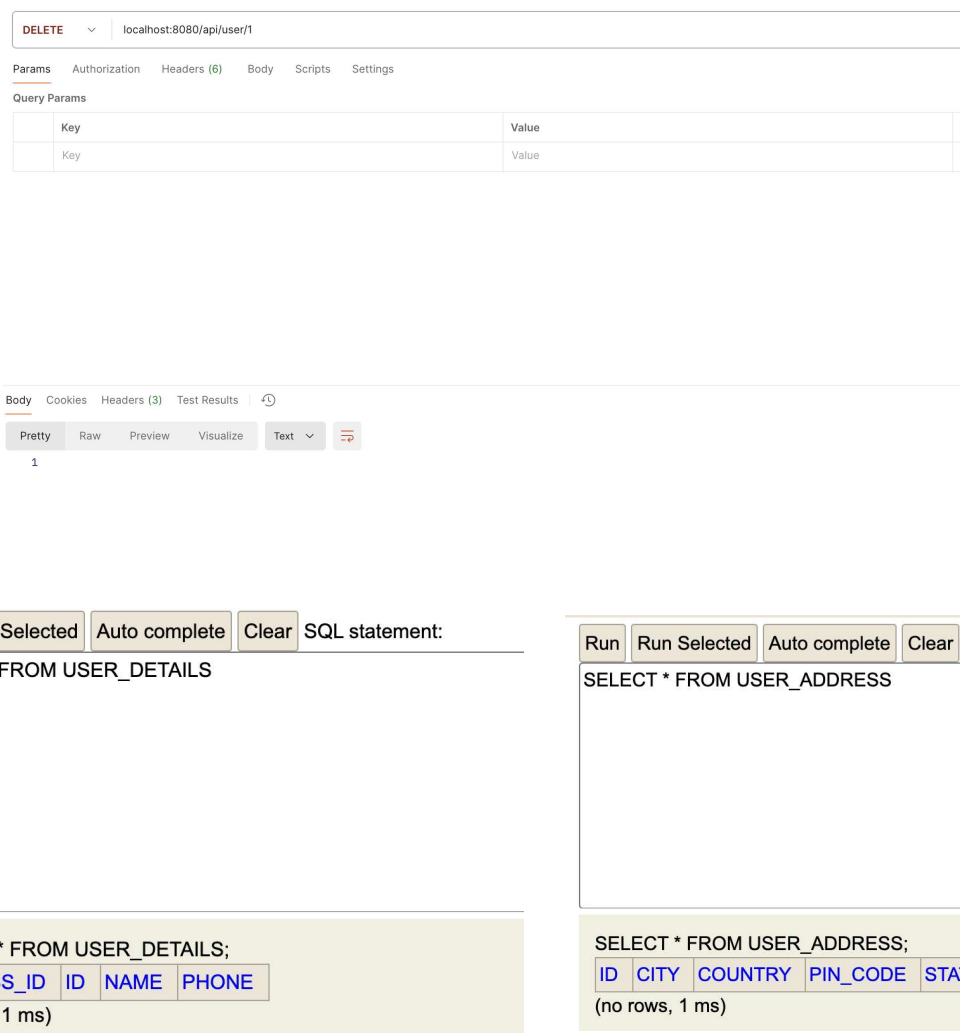
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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

```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;
```

Internally JPA code, has access to Ent
persistenceContext and thus handles

```
@Transactional
```

```
    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);
    }
}
```

```
    Assert.notNull(entity, message: "E
    if (this.entityInformation.isNew(e
        this.entityManager.persist(ent
        return entity;
    } else {
        return this.entityManager.merg
    }
}
```

EntityManage

```
    /**
     * Refresh the state of the instance from
     * overwriting changes made to the entity,
     * @param entity  entity instance
     * @throws IllegalArgumentException if the
     *         an entity or the entity is not
     * @throws TransactionRequiredException i
     *         transaction when invoked on a
     *         entity manager of type <code>Pe
     * @throws EntityNotFoundException if the
     *         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 associate Entities too.

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

```
@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.

```
/**
 * 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 associated Child Entities too.

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

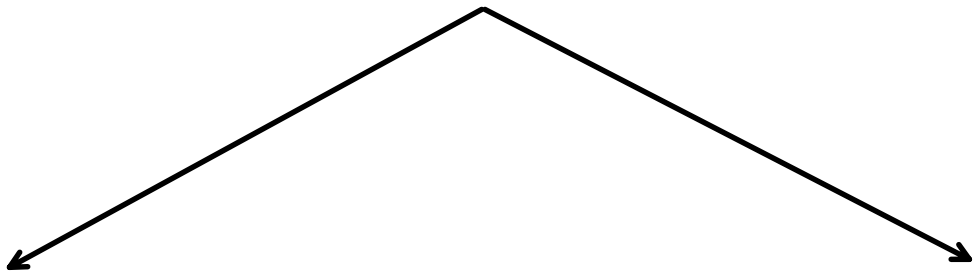
```
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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

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

Lazy Loading

- It means, associated entity is NOT loac
- Only loaded when explicitly accessed l userDetail.getUserAddress().

- Default for @OneToMany, @ManyToM

```java
@Target({METHOD, FIELD})
@Retention(RUNTIME)

public @interface OneToOne {

    (Optional) The entity class that is the target of the association.
    Defaults to the type of the field or property that stores the association.
    Class targetEntity() default void.class;

    (Optional) The operations that must be cascaded to the target of the association.
    By default no operations are cascaded.
    CascadeType[] cascade() default {};

    /**
     * (Optional) Whether the association should be lazily
     * loaded or must be eagerly fetched. The EAGER
     * strategy is a requirement on the persistence provider runtime that
     * the associated entity must be eagerly fetched. The LAZY
     * strategy is a hint to the persistence provider runtime.
     */
    FetchType fetch() default FetchType.EAGER;
```

```java
@Target({METHOD, FIELD})
@Retention(RUNTIME)

public @interface OneToMany {

    (Optional) The entity class that is the target of the association. Optional only if the collection
    property is defined using Java generics. Must be specified otherwise.
    Defaults to the parameterized type of the collection when defined using generics.
    Class targetEntity() default void.class;

    (Optional) The operations that must be cascaded to the target of the association.
    Defaults to no operations being cascaded.
    When the target collection is a java.util.Map, the cascade element applies to the map value.
    CascadeType[] cascade() default {};

    (Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER
    strategy is a requirement on the persistence provider runtime that the associated entities must be
    eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.
    FetchType fetch() default FetchType.LAZY;
```

```java
@Target({METHOD, FIELD})
@Retention(RUNTIME)

public @interface ManyToOne {

    (Optional) The entity class that is the target of the association.
    Defaults to the type of the field or property that stores the association.
    Class targetEntity() default void.class;

    (Optional) The operations that must be cascaded to the target of the association.
    By default no operations are cascaded.
    CascadeType[] cascade() default {};

    (Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER
    strategy is a requirement on the persistence provider runtime that the associated entity must be
    eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.
    FetchType fetch() default FetchType.EAGER;

    (Optional) Whether the association is optional. If set to false then a non-null relationship must
    always exist.
```

```java
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ManyToMany {

    (Optional) The entity class that is the target of the association. Optional only if the collection-valued
    relationship property is defined using Java generics. Must be specified otherwise.
    Defaults to the parameterized type of the collection when defined using generics.
    Class targetEntity() default void.class;

    (Optional) The operations that must be cascaded to the target of the association.
    When the target collection is a java.util.Map, the cascade element applies to the map value.
    Defaults to no operations being cascaded.
    CascadeType[] cascade() default {};

    (Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER
    strategy is a requirement on the persistence provider runtime that the associated entities must be
    eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.
    FetchType fetch() default FetchType.LAZY;
```

- 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 chil entities present for a parent ent so returning all the rows of the c entity might impact performanc by-default it uses LAZY techniqu

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
    private Long id;
    private String name;
    private String phone;
```

```
}
```

```
    @OneToOne(cascade = Casc
    @JoinColumn(name = "addr
    private UserAddress user


    // Constructors
    public UserDetails() {
    }

    //getters and setters
}
```

```
@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)

POST localhost:8080/api/user

Params • Authorization Headers (8) Body • Scripts Settings

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

```
{
    "name": "JohnXYZ",
    "phone": "1234567890",
    "userAddress": {
        "street": "123 Street",
        "city": "Bangalore",
        "state": "Karnataka",
        "country": "India",
        "pinCode": "10001"
    }
}
```

Body Cookies Headers (5) Test Results 200 OK

Pretty Raw Preview Visualize JSON

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

2nd Get Operation: (Failu

GET localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

	Key	Value
	Key	Value

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "timestamp": "2024-12-31T08:44:42.478+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/api/user/1"
}
```

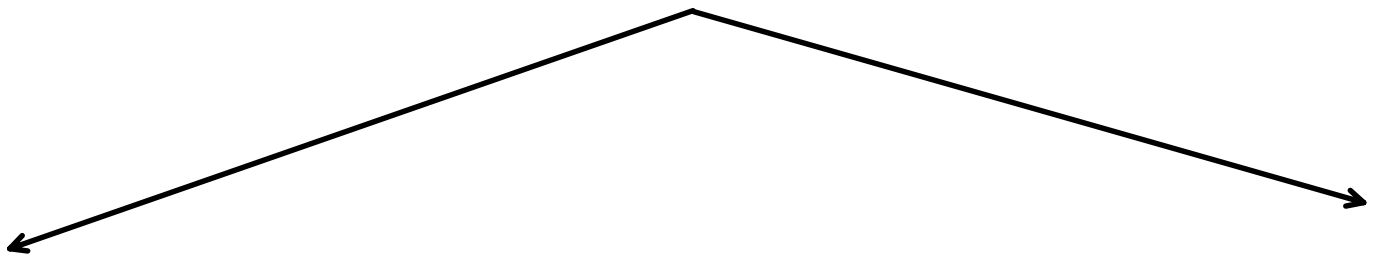
GET operation failed because of loading UserAddress LAZILY, Since during JS creation at the time of response, UserAddress data is missing thus library li JAKSON don't know how to serialize it and thus it failed while constructing t response.

```
com.fasterxml.jackson.databind.exc.InvalidDefinitionException Create breakpoint : No serializer found for class org.hibernate.p
    at com.fasterxml.jackson.databind.exc.InvalidDefinitionException.from(InvalidDefinitionException.java:77) ~[jackson-dat
    at com.fasterxml.jackson.databind.SerializerProvider.reportBadDefinition(SerializerProvider.java:1330) ~[jackson-databi
    at com.fasterxml.jackson.databind.DatabindContext.reportBadDefinition(DatabindContext.java:414) ~[jackson-databind-2.17
```

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 pers
Context (first level cache), so UserAddress data is already present in-memory. So when r
returned as part of same INSERT operation, it do not make any DB call, just fetched from
persistence context.

So, for GET operation, how to solve it:

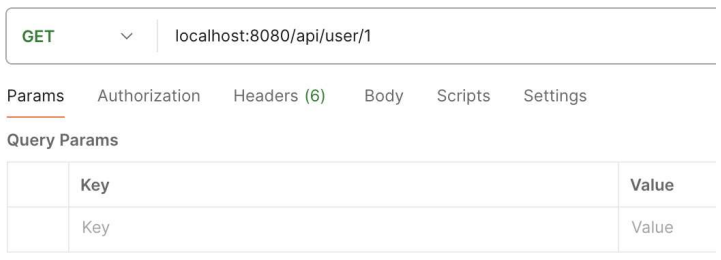


- Use **@JsonIgnore**
- This will remove the UserAddress field totally for both Lazy and Eager loading.

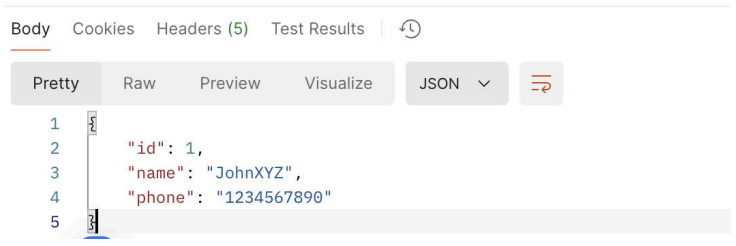
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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
}
```

- Using **DTO (Data Transfer Obje**
- Much clean and recommended
- Instead of sending Entity direct mapped to our DTO object.

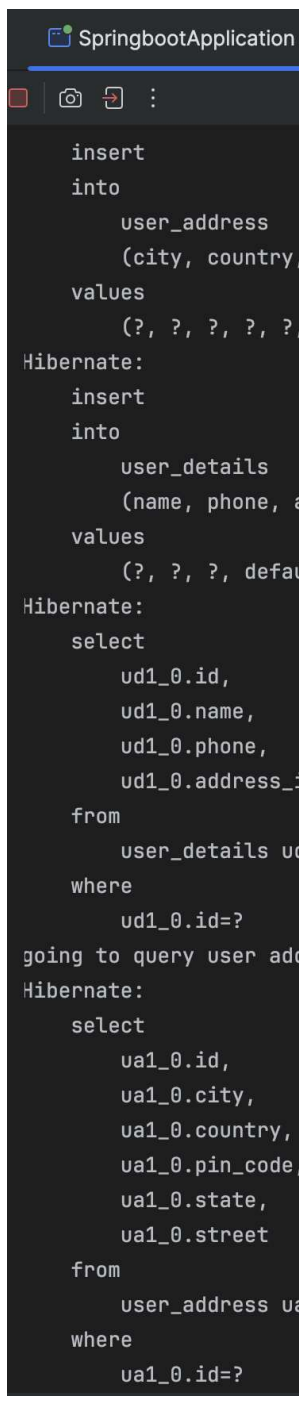
```java
public class UserDetailsDTO {

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

    // Constructor to populate from U
    public UserDetailsDTO(UserDetails
        this.id = userDetails.getId()
        this.name = userDetails.getNa
        this.phone = userDetails.getP
        System.out.println("going to
        this.address = userDetails.ge
                userDetails.getUserAd
    }

    //getters and setters
}
```




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

    @Autowired
    UserDetailsService userDetailsService
```

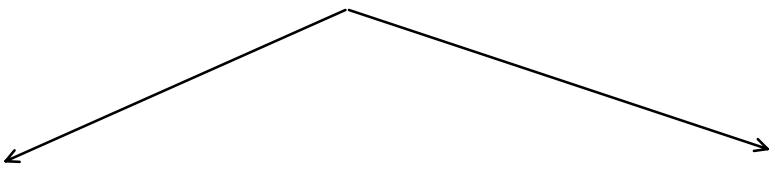



```
    @PostMapping(path = "/user")
    public UserDetails insertUser(@Reques
        return userDetailsService.saveUse
    }

    @GetMapping("/user/{id}")
    public UserDetailsDTO fetchUser(@Path
        return userDetailsService.findByI
    }
}
```



@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 ta



Owner side

- Holds the Foreign Key relationship in a table.

Inverse side

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

```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.IDENT
   private Long id;

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

    @OneToOne(mappedBy = "userAddress", fetch = Fe
    private UserDetails userDetails;

    //getters and setters
}
```

Table looks exactly same as OneToOne Unidirectional only

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

```
SELECT * FROM USER_DETAILS
```

```
SELECT * FROM USER_DETAILS;
```

ADDRESS_ID	ID	NAME	PHONE

(no rows, 5 ms)

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

```
SELECT * FROM USER_ADDRESS
```

```
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.

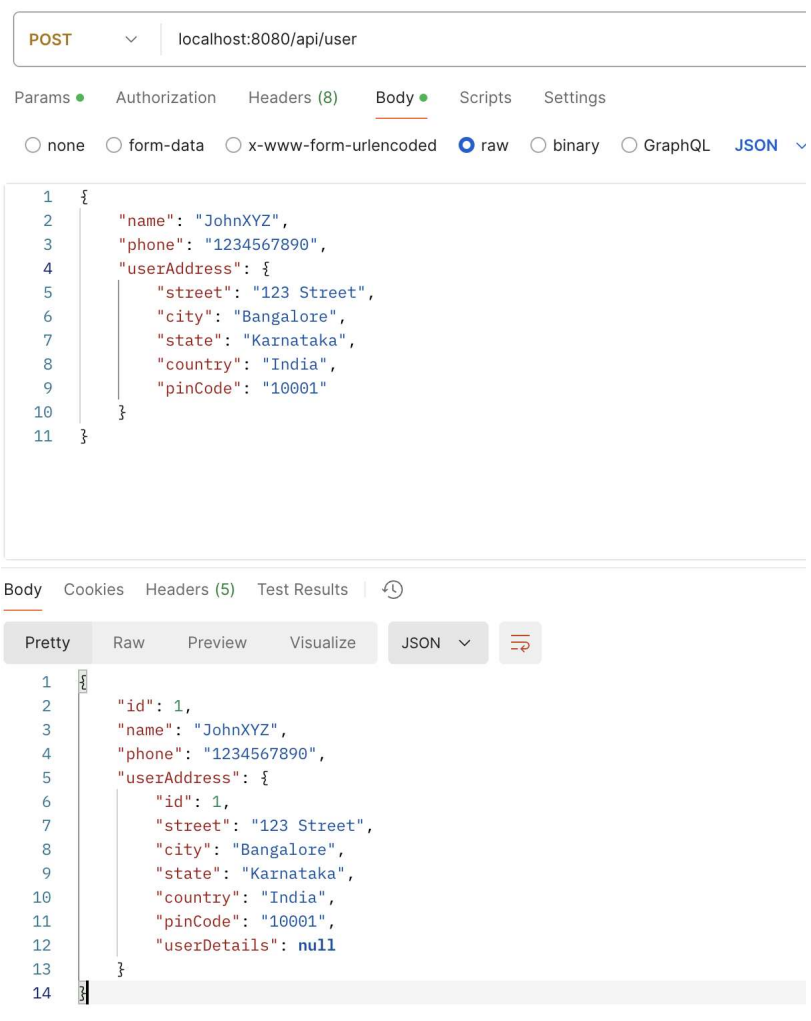
Created a controller and service class to query UserAddress entity

```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
    }
}
```

Let's observe the behavior now,

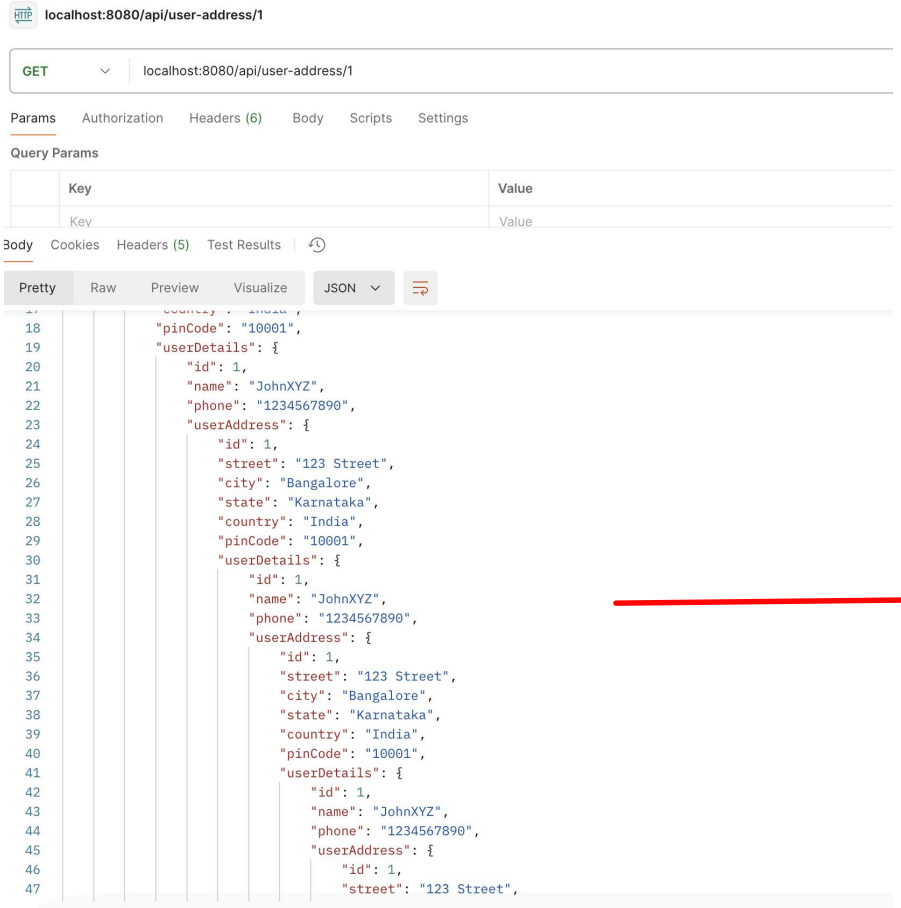
1st: Insert Operation like before, using UserDetail controller API:




```
2024-12-31T16:45:04.749+05:30  INFO 39163 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 m
```

```
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 REC

```
    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.spring
2024-12-31T16:47:23.303+05:30  WARN 39163 --- [nio-8080-exec-4] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.http.converter.HttpMessageNo
```

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.

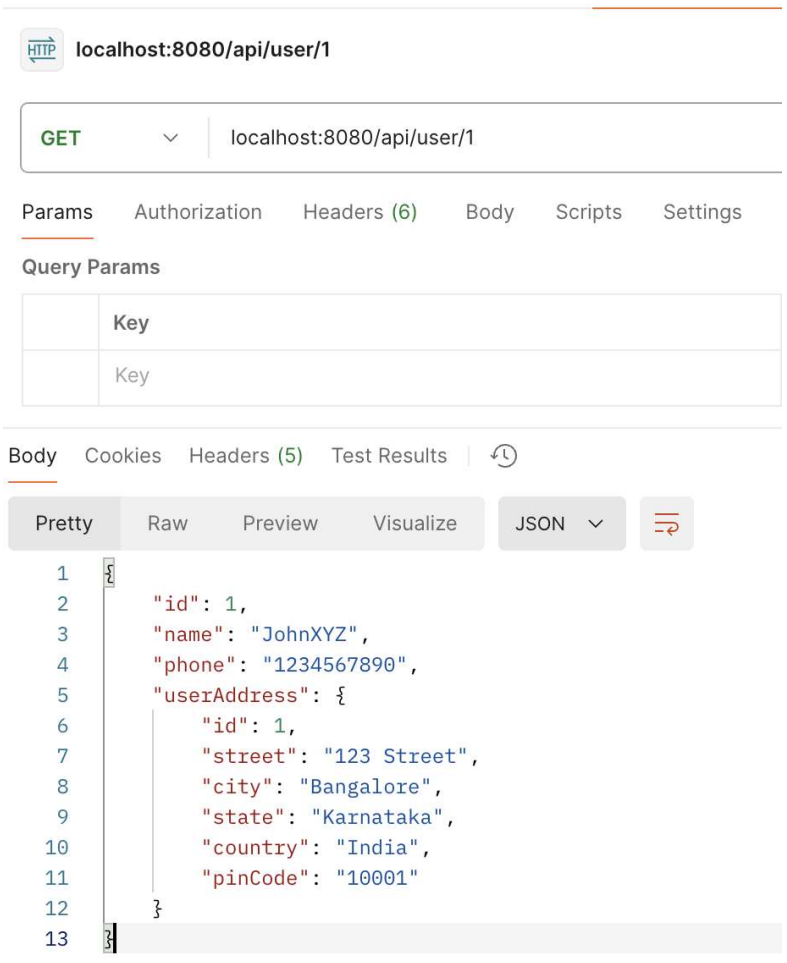
@JsonBackReference:

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

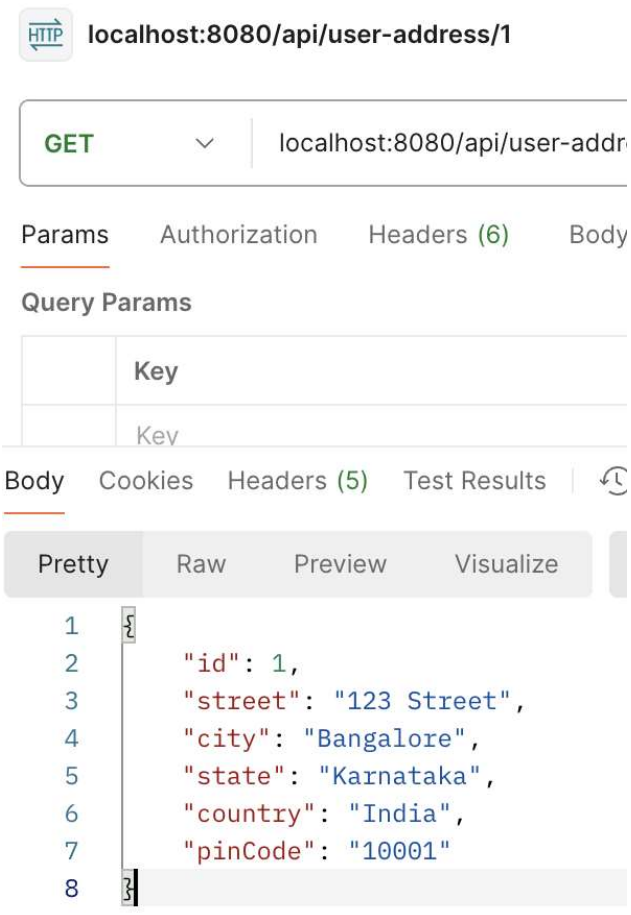
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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.
    @JsonBackReference
    private UserDetails userDetails;

    //getters and setters
}
```

GET API call on Parent Entity "UserDetail"



GET API call on Child Entity "Us



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

@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, th 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() {
    }
```

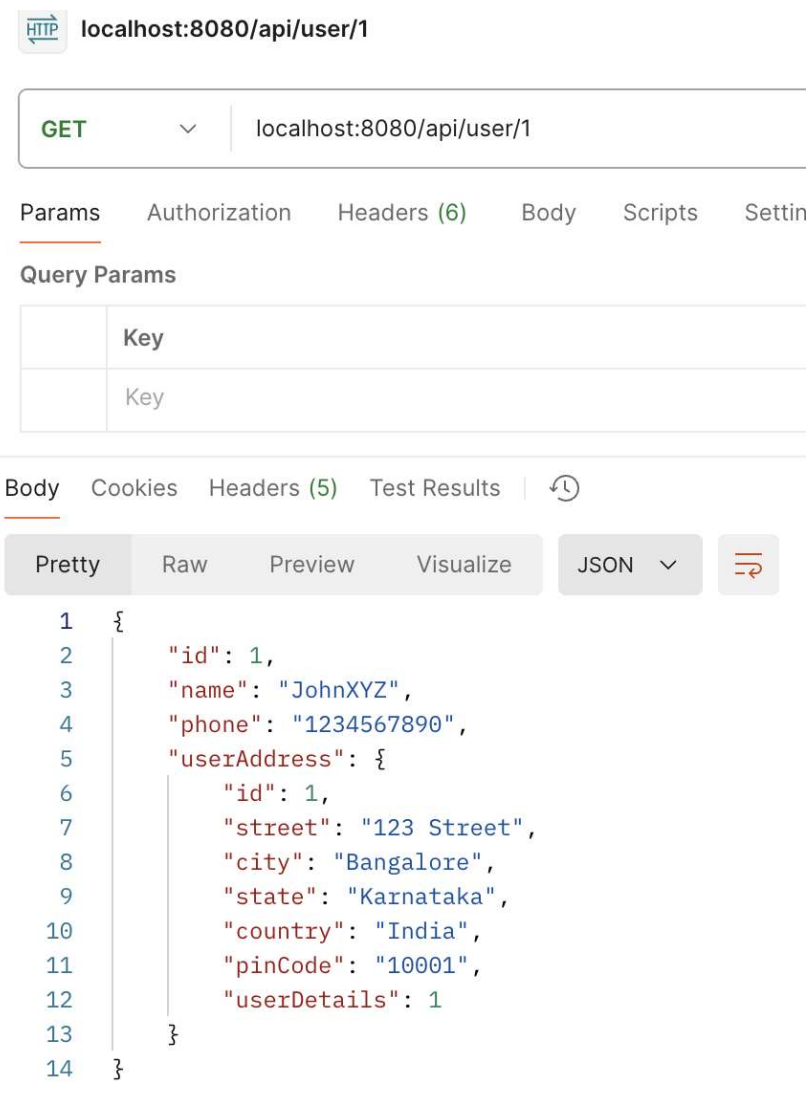
Any unique field from entity we can give, generally we give PK

```java
@Entity
@Table(name = "user_address")
@JsonIdentityInfo(
        generator = ObjectIdGenerators.PropertyGenerator.clas
        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.EA
    private UserDetails userDetails;
```

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

```
    //getters and setters

}
```

GET API call on Parent Entity "UserDetail"



GET API call on Child Entity "UserA