#27 : OneToOne Mapping (Unidirectional and Bidirectional)

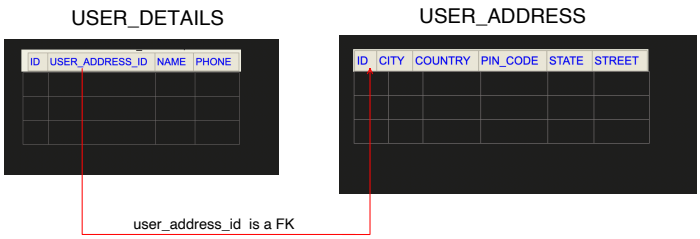
Sunday, 29 December 2024 12:26 PM

@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() {
    }
}
```

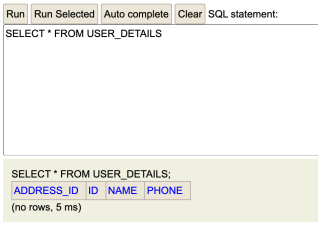


- By default hibernate choose:
  - the Foreign Key (FK) name as : <field_name_id>
    that's why for "userAddress" it created the FK name as : user_address_id
  - Chooses the Primary Key (PK) of other table.

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

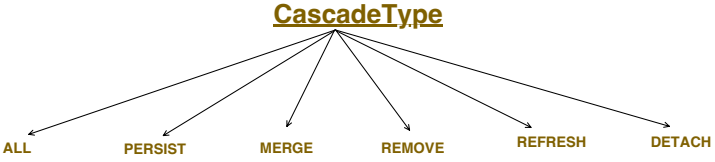
(no rows, 1 ms)

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.

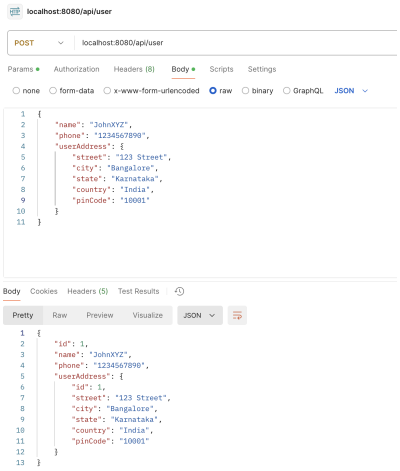
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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
}
```

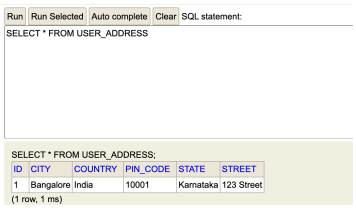
```java
@Entity
@Table(name = "user_address")
public class UserAddress {

   @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.

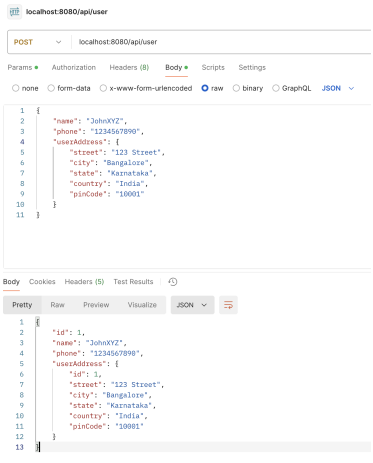
Lets, first see, what happen if we use only **PERSIT** Cascade type

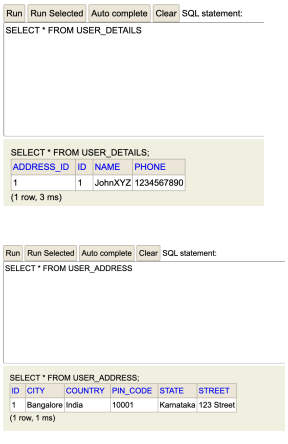
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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;
    }
}
```

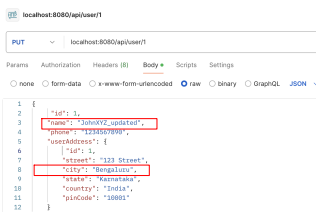
We are using the same "save" method to update, internally it check, if its new entity or not, by looking for ID field present in entity object. Since we are passing ID in Request body, it will try to update instead of insert.

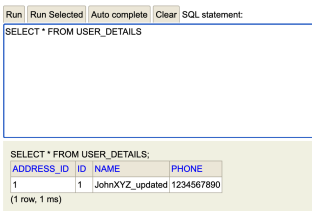
1st INSERT OPERATION

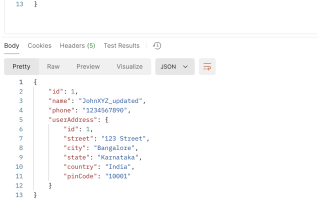


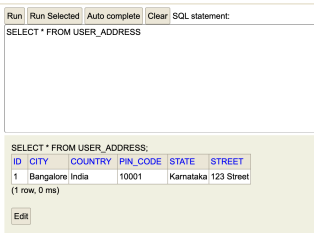


2nd UPDATE OPERATION:
(Changes made in both UserDetails and UserAddress, but only UserDetails changes reflected)





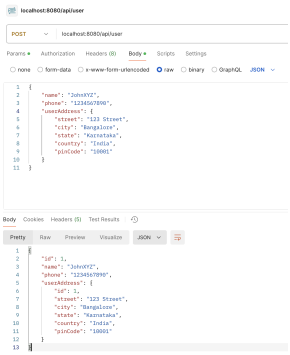


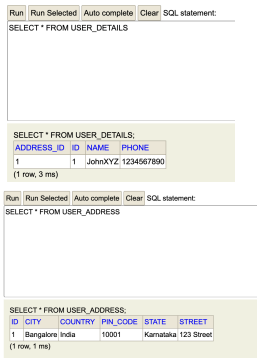


Now If I want to both INSERT and then UPDATE association capability, means I need both PERSIT + MERGE Cascade Type

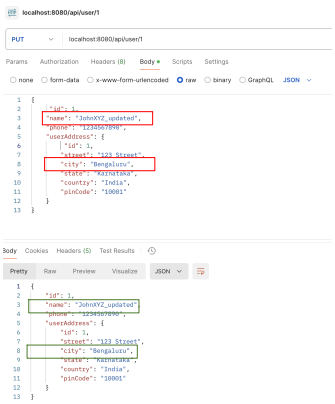
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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
}
```

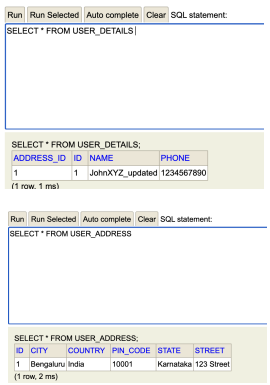
1st INSERT OPERATION , Similar like above.





2nd UPDATE OPERATION





CascadeType.REMOVE

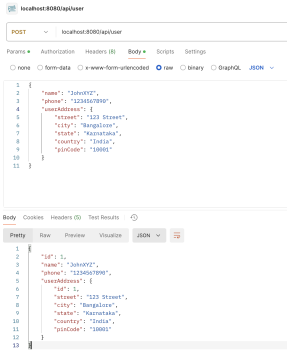
- Deleting the User entity automatically delete its associated UserAddress entity data.

```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();
    }
}
```

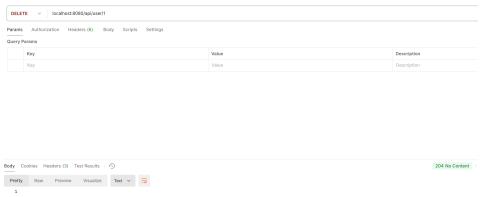
```java
@Service
public class UserDetailsService {

    @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);
    }
}
```

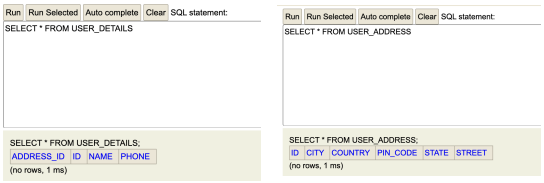
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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.

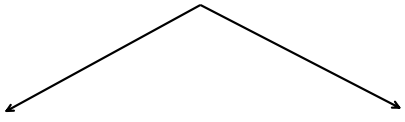
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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?

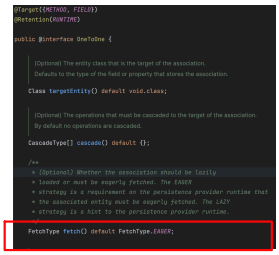


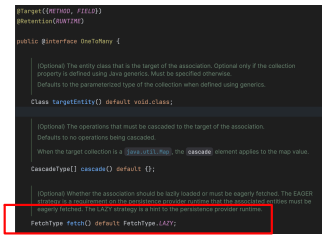
Eager Loading

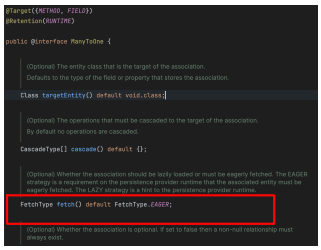
Lazy Loading

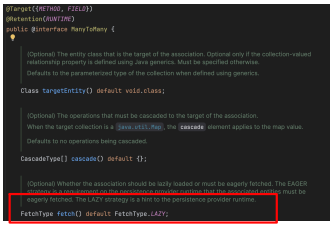
- It means, associated entity is loaded immediately along with the parent entity.
- Default for @OneToOne and @ManyToOne

- 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:

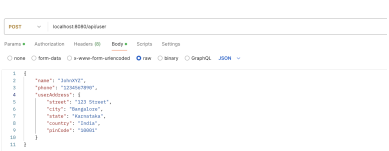
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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
}
```

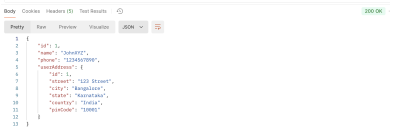
```java
@Service
public class UserDetailsService {

    @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:

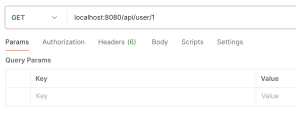
- Use **@JsonIgnore**
- This will remove the UserAddress field totally for both Lazy and Eager loading.

- 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
}
```

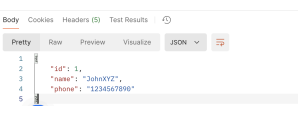
```java
public class UserDetailsDTO {

    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
}
```

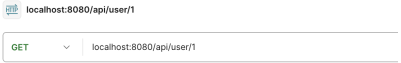
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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
}
```

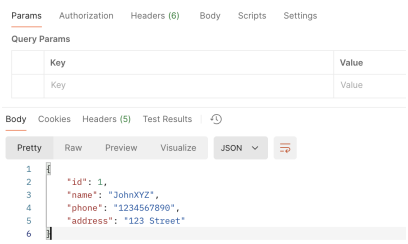


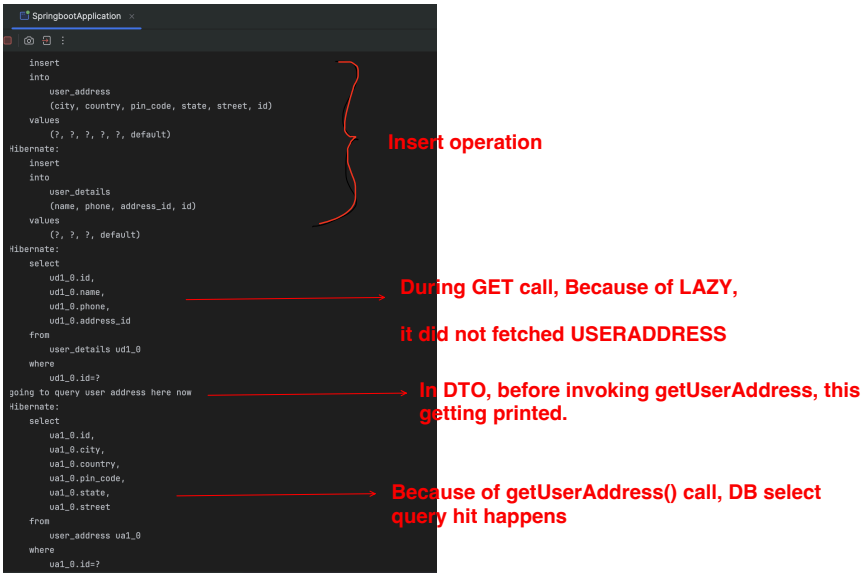
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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();
    }
}
```



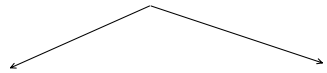






@OneToOne bidirectional

- 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.

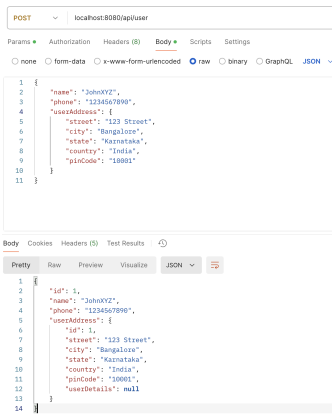
Created a controller and service class to query UserAddress entity

```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();
    }
}
```

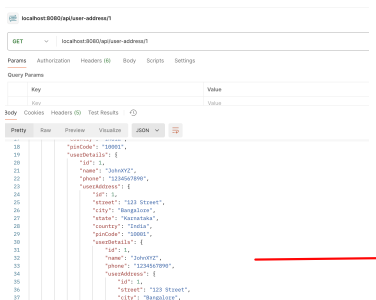
Let's observe the behavior now,

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.

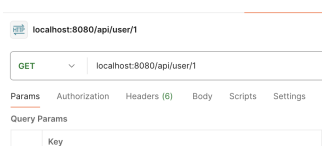
@JsonBackReference:

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

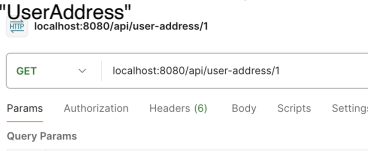
```java
@Table(name = "user_details")
@Entity
public class UserDetails {

    @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
}
```

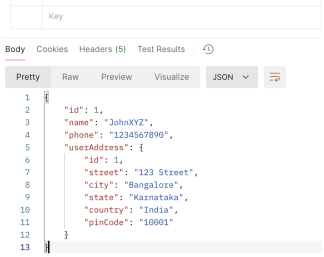
```java
@Entity
@Table(name = "user_address")
public class UserAddress {

  @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
}
```

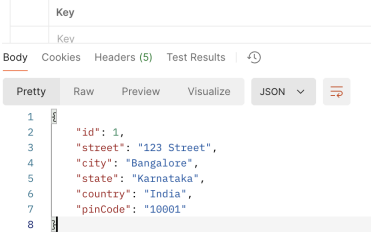
GET API call on Parent Entity "UserDetail"



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

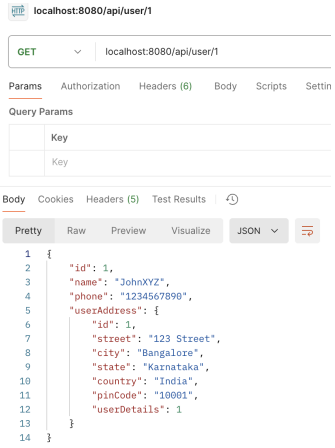
- During serialization , Jackson gives the unique ID to the entity (based on property field).
- Though which Jackson can know, if that particular id entity is already serialized before, then it skip the serialization.

```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"