

[@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).

```
@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
}
```

```
@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() {
    }
}
```

USER_DETAILS

ID	USER_ADDRESS_ID	NAME	PHONE

user_address_id is a FK

USER_ADDRESS

ID	CITY	COUNTRY	PIN_CODE	STATE	STREET

- 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.

```
@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
}
```

```
@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() {
    }
}
```

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.

```

@Entity
@Table(name = "user_details")
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
}

```

```

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

    @EmbeddedId
    private UserAddressCK id;

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

    //getters and setters
}

```

```

@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

Run | Run Selected | Auto complete | Clear | SQL statement:
SELECT * FROM USER_ADDRESS

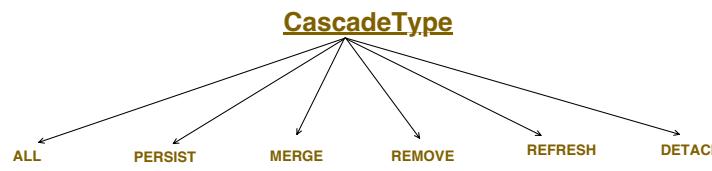
SELECT * FROM USER_DETAILS;
ID ADDRESS_PIN_CODE ADDRESS_STREET NAME PHONE
(no rows, 2 ms)

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.



- 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.

```

@Entity
@Table(name = "user_details")
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
}

```

```

@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
}

```

```

@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);
    }
}

```

```

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

```

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

```

1 {
2   "name": "JohnXYZ",
3   "phone": "1234567890",
4   "userAddress": {
5     "street": "123 Street",
6     "city": "Bangalore",
7     "state": "Karnataka",
8     "country": "India",
9     "pinCode": "10001"
10   }
11 }
12
13 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

localhost:8080/api/user

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS

1	1	JohnXYZ	1234567890
---	---	---------	------------

(1 row, 1 ms)

localhost:8080/api/user

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_ADDRESS

1	1	1	JohnXYZ	1234567890
---	---	---	---------	------------

(1 row, 1 ms)

localhost:8080/api/user

Run Selected Auto complete Clear SQL statement:

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 **PERSIST** Cascade type

```
@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);
    }
}
```

```
@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

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

```

1 {
2   "name": "JohnXYZ",
3   "phone": "1234567890",
4   "userAddress": {
5     "street": "123 Street",
6     "city": "Bangalore",
7     "state": "Karnataka",
8     "country": "India",
9     "pinCode": "10001"
10   }
11 }
12
13 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

localhost:8080/api/user

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS

1	1	JohnXYZ	1234567890
---	---	---------	------------

(1 row, 3 ms)

localhost:8080/api/user

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_ADDRESS

1	1	1	JohnXYZ	1234567890
---	---	---	---------	------------

(1 row, 3 ms)

localhost:8080/api/user

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_ADDRESS;

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

(1 row, 3 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

```

1 {
2   "id": 1,
3   "name": "JohnXYZ",
4   "phone": "1234567898",
5   "userAddress": {
6     "id": 1,
7     "street": "123 Street",
8     "city": "Bangalore",
9     "state": "Karnataka",
10    "country": "India",
11    "pinCode": "10001"
12  }
13 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

localhost:8080/api/user/1

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS

1	1	JohnXYZ	1234567890
---	---	---------	------------

(1 row, 1 ms)

localhost:8080/api/user/1

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_ADDRESS;

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

(1 row, 1 ms)

localhost:8080/api/user/1

Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS;

1	1	JohnXYZ	1234567890
---	---	---------	------------

(1 row, 1 ms)

```

13 }

Run Run Selected Auto complete Clear SQL statement:
SELECT * FROM USER_ADDRESS

Pretty Raw Preview Visualizer JSON ▾

1 {
2   "id": 1,
3   "name": "JohnXYZ_updated",
4   "phone": "1234567890",
5   "userAddress": [
6     {
7       "street": "123 Street",
8       "city": "Bangalore",
9       "state": "Karnataka",
10      "country": "India",
11      "pinCode": "10001"
12    }
13 }

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 PERSIST + MERGE Cascade Type

```

@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.

```

localhost:8080/api/user
POST - localhost:8080/api/user

Params Authorization Headers (0) Body Scripts Settings
None Form-data x-www-form-urlencoded Raw Binary GraphQL JSON ▾

1 {
2   "name": "JohnXYZ",
3   "phone": "1234567890",
4   "userAddress": [
5     {
6       "street": "123 Street",
7       "city": "Bangalore",
8       "state": "Karnataka",
9       "country": "India",
10      "pinCode": "10001"
11    }
12 }

Body Cookies Headers (0) Test Results
Pretty Raw Preview Visualizer JSON ▾

1 {
2   "id": 1,
3   "name": "JohnXYZ",
4   "phone": "1234567890",
5   "userAddress": [
6     {
7       "street": "123 Street",
8       "city": "Bangalore",
9       "state": "Karnataka",
10      "country": "India",
11      "pinCode": "10001"
12    }
13 }

Run Run Selected Auto complete Clear SQL statement:
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
ID CITY COUNTRY PIN_CODE STATE STREET
1 Bangalore India 10001 Karnataka 123 Street
(1 row, 1 ms)

Body Cookies Headers (0) Test Results
Pretty Raw Preview Visualizer JSON ▾

1 {
2   "id": 1,
3   "name": "JohnXYZ_updated",
4   "phone": "1234567890",
5   "userAddress": [
6     {
7       "street": "123 Street",
8       "city": "Bangalore",
9       "state": "Karnataka",
10      "country": "India",
11      "pinCode": "10001"
12    }
13 }

Body Cookies Headers (0) Test Results
Pretty Raw Preview Visualizer JSON ▾

1 {
2   "id": 1,
3   "name": "JohnXYZ_updated",
4   "phone": "1234567890",
5   "userAddress": [
6     {
7       "street": "123 Street",
8       "city": "Bangalore",
9       "state": "Karnataka",
10      "country": "India",
11      "pinCode": "10001"
12    }
13 }

SELECT * FROM USER_ADDRESS;
ID CITY COUNTRY PIN_CODE STATE STREET
1 Bangalore India 10001 Karnataka 123 Street
(1 row, 2 ms)

```

2nd UPDATE OPERATION

```

localhost:8080/api/user/1
PUT - localhost:8080/api/user/1

Params Authorization Headers (0) Body Scripts Settings
None Form-data x-www-form-urlencoded Raw Binary GraphQL JSON ▾

1 {
2   "id": 1,
3   "name": "JohnXYZ_updated",
4   "phone": "1234567890",
5   "userAddress": [
6     {
7       "street": "123 Street",
8       "city": "Karnataka",
9       "state": "India",
10      "country": "India",
11      "pinCode": "10001"
12    }
13 }

Body Cookies Headers (0) Test Results
Pretty Raw Preview Visualizer JSON ▾

1 {
2   "id": 1,
3   "name": "JohnXYZ_updated",
4   "phone": "1234567890",
5   "userAddress": [
6     {
7       "street": "123 Street",
8       "city": "Bangalore",
9       "state": "Karnataka",
10      "country": "India",
11      "pinCode": "10001"
12    }
13 }

Run Run Selected Auto complete Clear SQL statement:
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
ID CITY COUNTRY PIN_CODE STATE STREET
1 Bengaluru India 10001 Karnataka 123 Street
(1 row, 2 ms)

```

CascadeType.REMOVE

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

```

@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();
    }
}

```

```

@Service
public class UserDetailsService {

    @Autowired
    UserDetailsServiceRepository userDetailsService;

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

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsServiceRepository.findById(id);
        if(existingUser.isPresent()) {
            existingUser.get().setCity(user.getCity());
            existingUser.get().setCountry(user.getCountry());
            existingUser.get().setPinCode(user.getPinCode());
            existingUser.get().setStreet(user.getStreet());
            existingUser.get().setName(user.getName());
            existingUser.get().setPhone(user.getPhone());
            return userDetailsServiceRepository.save(existingUser.get());
        }
        return null;
    }

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

```

```

@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

POST /api/user

Body

```

{
    "name": "JohnDoe",
    "phone": "1234567890",
    "userAddress": {
        "id": 1,
        "street": "123 Main Street",
        "city": "Bangalore",
        "state": "Karnataka",
        "country": "India",
        "pinCode": "560001"
    }
}

```

DELETE /api/user/1

Body

CascadeType.REFRESH and CascadeType.DETACH

- Both are Less frequently used.

CascadeType.REFRESH

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

```

@Service
public class UserDetailsService {

    @Autowired
    UserDetailsServiceRepository userDetailsService;

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

    public UserDetails updateUser(Long id, UserDetails user) {
        Optional<UserDetails> existingUser = userDetailsServiceRepository.findById(id);
        if(existingUser.isPresent()) {
            existingUser.get().setCity(user.getCity());
            existingUser.get().setCountry(user.getCountry());
            existingUser.get().setPinCode(user.getPinCode());
            existingUser.get().setStreet(user.getStreet());
            existingUser.get().setName(user.getName());
            existingUser.get().setPhone(user.getPhone());
            return userDetailsServiceRepository.save(existingUser.get());
        }
        return null;
    }

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

```

EntityManager.java

```

    /**
     * Refreshes the state of the instance from the database.
     * Reversing changes made to the entity, if any.
     * @param entity entity instance
     * @throws TransactionRequiredException if the instance is not managed
     * @throws TransactionalRequiredException if there is no transaction when managed in a container-managed
     * @throws EntityNotRefreshedException if the entity is not 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.

```
@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
}
```

CascadeType.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 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.

```
@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() {
    }

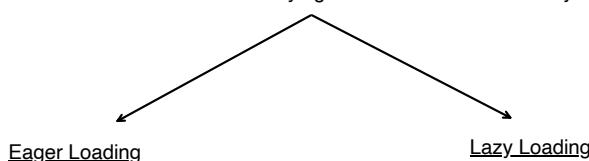
    //getters and setters
}
```

CascadeType.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?



- 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.

```
@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 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;
}
```

```
@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<? extends Entity> 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 [JavaList, 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;
}
```

```
@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 eagerly 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.
}
```

```
@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<? extends Entity> targetEntity() default void.class;

    // (Optional) The operations that must be cascaded to the target of the association.
    // When the target collection is a [JavaList, 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;
}
```

- 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:

```
@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);
    }
}
```

```
@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
}
```

```
@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:8990/api/user

Params	Authorization	Headers	Body	Script	Settings
<input checked="" type="radio"/> form-data	<input type="radio"/> www-form-urlencoded	<input type="radio"/> raw	<input type="radio"/> binary	<input type="radio"/> GraphQL	<input type="radio"/> JSON
<pre>1 { 2 "name": "Avinash", 3 "phone": "9876543210", 4 "address": "123 Main Street", 5 "city": "New York", 6 "state": "New York", 7 "zipCode": "10001", 8 "pincode": "300001" 9 } 10 11</pre>					

2nd Get Operation: (Failure)

GET localhost:8990/api/user

Params	Authorization	Headers	Body	Script	Settings					
<input type="radio"/> Query Params										
<table border="1"> <thead> <tr> <th>Key</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>Key</td> <td>Value</td> <td>Description</td> </tr> </tbody> </table>	Key	Value	Description	Key	Value	Description				
Key	Value	Description								
Key	Value	Description								

```

Body Cookies Headers (5) Test Results 200 OK
Pretty Raw Preview Visualize JSON
1 {
2     "id": 1,
3     "name": "JohnXZ",
4     "address": "123 Street",
5     "userAddress": {
6         "street": "123 Street",
7         "city": "New York",
8         "state": "New York",
9         "zip": "10001"
10    }
11 }

```

Body Cookies Headers (4) Test Results 500 Internal Server Error

```

Pretty Raw Preview Visualize JSON
1 {
2     "timestamp": "2024-12-31T08:44:42.479+00:00",
3     "status": 500,
4     "error": "Internal Server Error",
5     "path": "/api/user/1"
6 }

```

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.

```

@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
}

```

```

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
}

```

```

@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
}

```

GET localhost:8080/api/user/1

Params	Authorization	Headers (8)	Body	Scripts	Settings
Query Params					

```

@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();
    }
}

```

```

@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsService;

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

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

```

Body Cookies Headers (5) Test Results 200 OK

```

Pretty Raw Preview Visualize JSON
1 {
2     "id": 1,
3     "name": "JohnXZ",
4     "phone": "1234567890"
5 }

```

localhost:8080/api/user/1

GET localhost:8080/api/user/1

Params Authorization Headers (6) Body Scripts Settings

Query Params

Key	Value
Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1
2   "id": 1,
3   "name": "JohnXYZ",
4   "phone": "1234567890",
5   "address": "123 Street"
6

```

Insert operation:

```

insert
into
    user_address
    (city, country, pin_code, state, street, id)
values
    (?, ?, ?, ?, ?, default)

```

During GET call, Because of LAZY, it did not fetch USERADDRESS:

```

select
    ud1_0.id,
    ud1_0.name,
    ud1_0.phone,
    ud1_0.address_id
from
    user_details ud1_0
where
    ud1_0.id?

```

In DTO, before invoking getUserAddress, this getting printed:

```

select
    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.id?

```

Because of getUserAddress() call, DB select query hit happens:

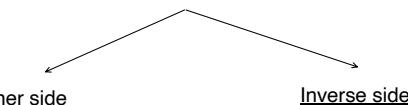
```

select
    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.id?

```

@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)



- Holds the Foreign Key relationship in a table.
- No Foreign key is created in table.
- Only holds **Object** reference of owing entity.

```

@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
}

@Table(name = "user_address")
@Entity
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.

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_ADDRESS

```
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 entity.

Created a controller and service class to query UserAddress entity

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

    @Autowired
    UserAddressService userDetailsService;

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

```
@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:

```
POST      localhost:8080/api/user
Params   Authorization Headers (0) Body Scripts Settings
none   form-data x-www-form-urlencoded raw binary GraphQL JSON
1
2     "name": "John Doe",
3     "phone": "1234567890",
4     "userAddress": [
5         {
6             "street": "123 Street",
7             "city": "Mumbai",
8             "state": "Karnataka",
9             "country": "India",
10            "pinCode": "10001"
11        }
12
13
14 ]
```

```
1
2     {
3         "id": 1,
4         "name": "John Doe",
5         "phone": "1234567890",
6         "userAddress": [
7             {
8                 "id": 1,
9                 "street": "123 Street",
10                "city": "Mumbai",
11                "state": "Karnataka",
12                "country": "India",
13                "pincode": "10001",
14                "userDetails": null
15            }
16        ]
17     }
```

```
2024-12-31T16:45:04.749+05:30 INFO 39123 --- [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:

```
GET      localhost:8080/api/user-address/1
Params   Authorization Headers (0) Body Scripts Settings
Query Params
Key Value
Body Cookies Headers (0) Test Results
Pretty Raw Preview Visualize JSON
1
2     {
3         "id": 1,
4         "name": "John Doe",
5         "userAddress": [
6             {
7                 "id": 1,
8                 "street": "123 Street",
9                 "city": "Mumbai",
10                "state": "Karnataka",
11                "country": "India",
12                "pincode": "10001",
13                "userDetails": null
14            }
15        ]
16     }
```

INFINITE RECURSION

```

38     "state": "Karnataka",
39     "city": "Mysore",
40     "pinCode": "570001",
41     "address": "123 Street",
42     "lat": 12.9154,
43     "long": 77.6129,
44     "id": 1234567890,
45     "userAddress": [
46       {
47         "street": "123 Street",
48       }
49     ]
50   }
51 }

52 }
```

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 @JsonBackReference
and

@JsonManagedReference:

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

@JsonBackReferenc

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

```

@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
}

@Table(name = "user_address")
@Entity
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;

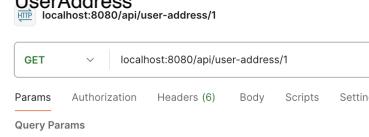
    //getters and setters
}

```

GET API call on Parent Entity "UserDetail"



GET API call on Child Entity "UserAddress"



```

1 {
2   "id": 1,
3   "name": "JohnXYZ",
4   "phone": "1234567890",
5   "userAddress": {
6     "id": 1,
7     "street": "123 Street",
8     "city": "Bangalore",
9     "state": "Karnataka",
10    "country": "India",
11    "pinCode": "10001"
12  }
13 }

1 {
2   "id": 1,
3   "street": "123 Street",
4   "city": "Bangalore",
5   "state": "Karnataka",
6   "country": "India",
7   "pinCode": "10001"
8 }

```

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.

```

@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
}

@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      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  ⚙️

1 {
2   "id": 1,
3   "name": "JohnXYZ",
4   "phone": "1234567890",
5   "userAddress": {
6     "id": 1,
7     "street": "123 Street",
8     "city": "Bangalore",
9     "state": "Karnataka",
10    "country": "India",
11    "pinCode": "10001",
12  }
13 }

14

```

GET API call on Child Entity "UserAddress"

```

GET      localhost:8080/api/user-address/1

Params   Authorization   Headers (6)   Body   Scripts   Settings
Query Params
Key
Key

Body   Cookies   Headers (5)   Test Results   ⚙️
Pretty  Raw  Preview  Visualize  JSON  ⚙️

1 {
2   "id": 1,
3   "street": "123 Street",
4   "city": "Bangalore",
5   "state": "Karnataka",
6   "country": "India",
7   "pinCode": "10001",
8   "userDetails": {
9     "id": 1,
10    "name": "JohnXYZ",
11    "phone": "1234567890",
12    "userAddress": 1
13  }
14 }

15

```