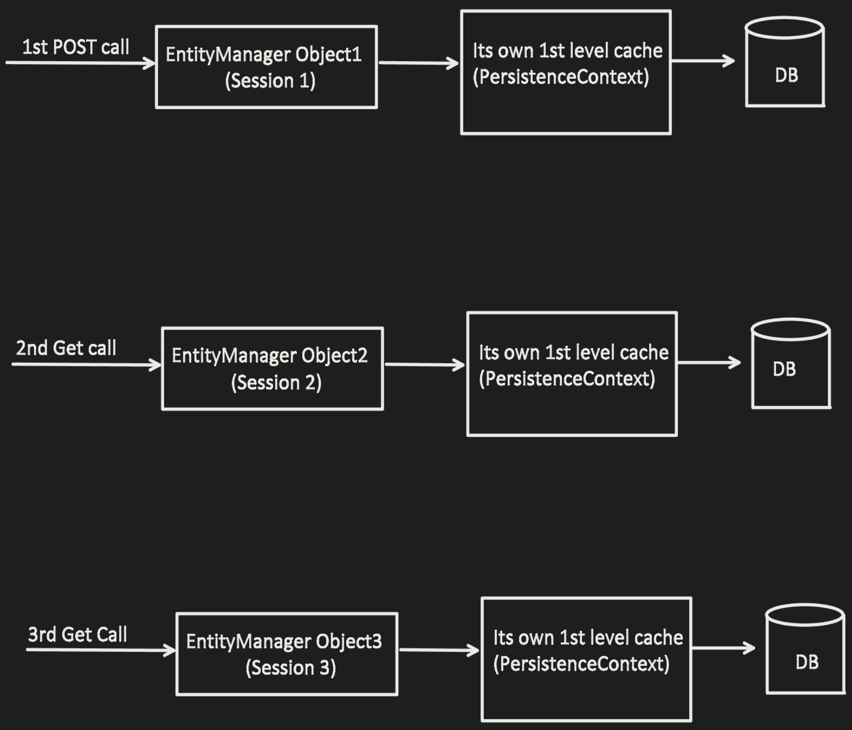
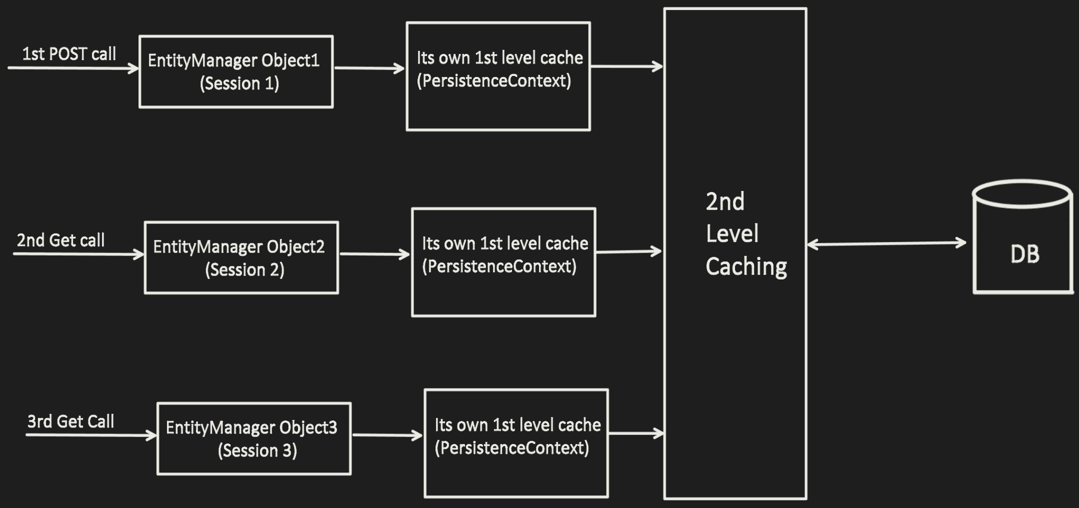
From previous video, First level caching, we already know that, for each HTTP REQUEST, different EntityManager Object (session) is created and its have its own Persistence context (1st level cache)


1st POST call
EntityManager Object1 (Session 1)
Its own 1st level cache (PersistenceContext)
DB
2nd Get call
EntityManager Object2 (Session 2)
Its own 1st level cache (PersistenceContext)
DB
3rd Get Call
EntityManager Object3 (Session 3)
Its own 1st level cache (PersistenceContext)
DB

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


1st POST call
EntityManager Object1 (Session 1)
Its own 1st level cache (PersistenceContext)
2nd Get call
EntityManager Object2 (Session 2)
Its own 1st level cache (PersistenceContext)
3rd Get Call
EntityManager Object3 (Session 3)
Its own 1st level cache (PersistenceContext)
2nd Level Caching
DB

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

pom.xml

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

application.properties

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

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

    @Autowired
    UserDetailsService userDetailsService;

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

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

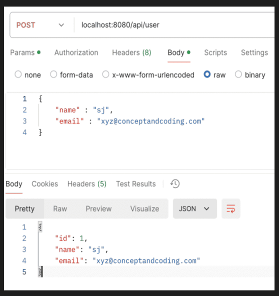
```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {

        return userDetailsRepository.save(user);
    }

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

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

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

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

    // Getters and setters
}
```

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

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

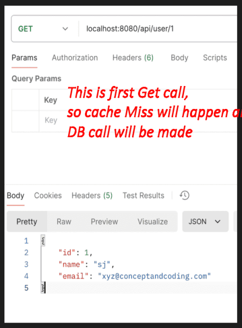
2. Get:
   During Get, JPA will check, if data is present in cache? If Yes, its cache hit and return else its cache miss and it will fetch from DB and put into Cache.



```
localhost:8080/api/user
```

```json
{
    "name" : "sj",
    "email" : "xyz@conceptandcoding.com"
}
```

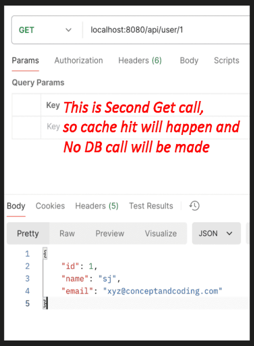
```json
{
    "id": 1,
    "name": "sj",
    "email": "xyz@conceptandcoding.com"
}
```



This is first Get call, so cache Miss will happen and DB call will be made

```
localhost:8080/api/user/1
```

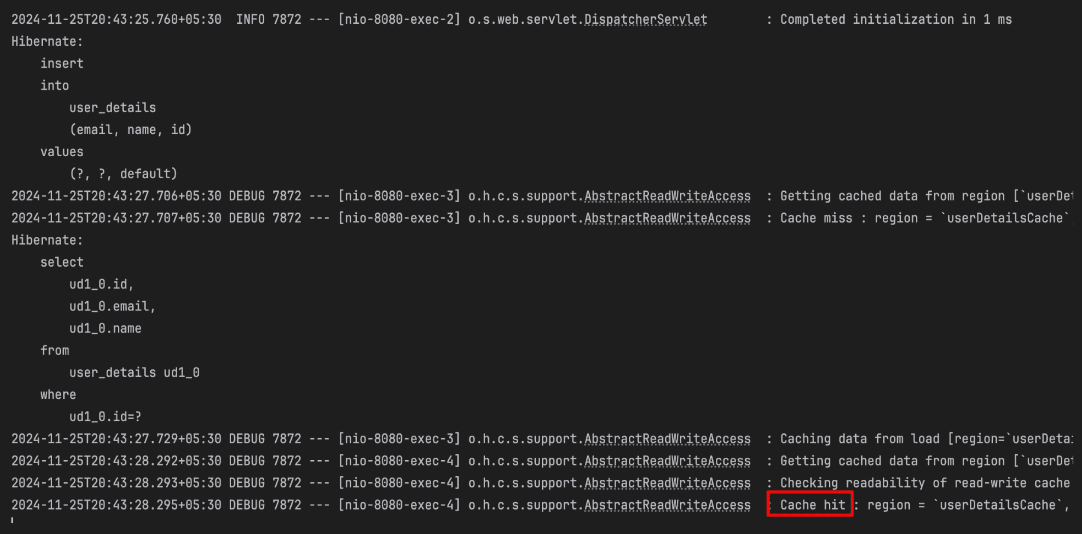
```json
{
    "id": 1,
    "name": "sj",
    "email": "xyz@conceptandcoding.com"
}
```



This is Second Get call, so cache hit will happen and No DB call will be made

```
localhost:8080/api/user/1
```

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



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

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

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

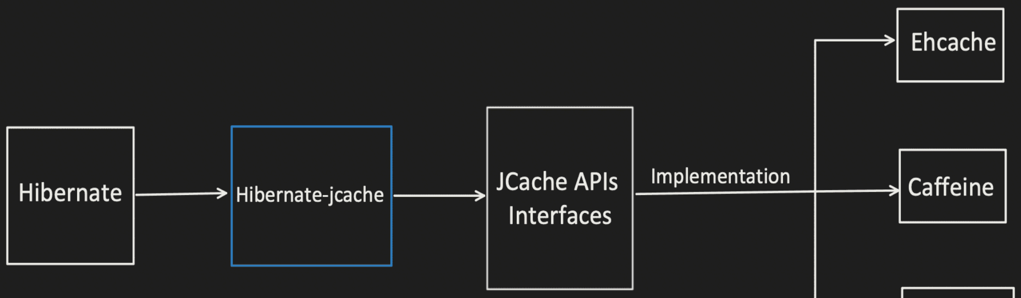
→ Provides the core implementation of Second level caching

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

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

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

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



2. Lets understand application.properties and Region:

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

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

Region:

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

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

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

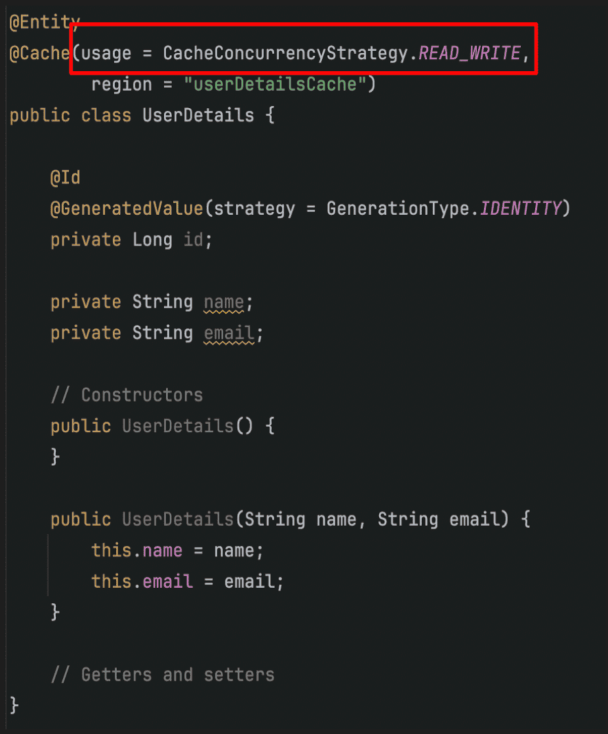
```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
        region = "userDetailsCache")
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

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

    // Getters and setters
}
```

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

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

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

    // Getters and Setters
}
```

ehcache.xml
(file within "src/main/resources/" path)

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

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

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

3. Different ***CacheConcurrencyStrategy***

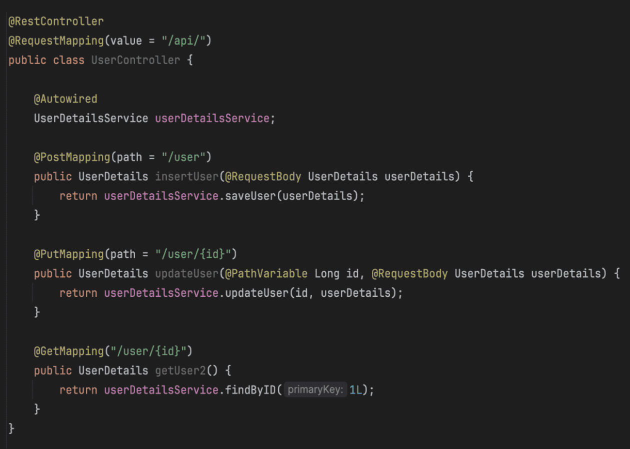


```java
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE,
        region = "userDetailsCache")
public class UserDetails {

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

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

    // Getters and setters
}
```

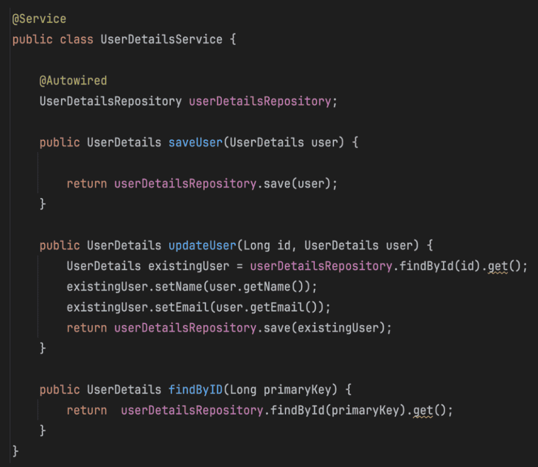
S.No.	Strategy
1.	READ_ONLY
2.	READ_WRITE
3.	NONSTRICT_READ_WRITE
4.	TRANSACTIONAL

1. READ_ONLY

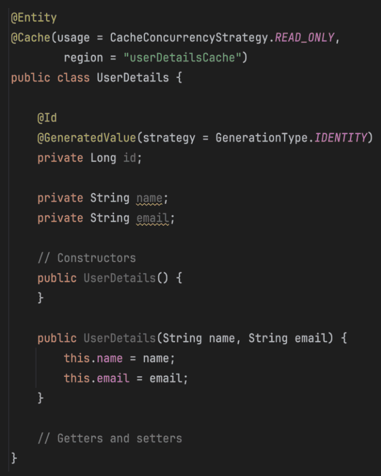
- Good for Static Data
- Which do not require any updates
- If try to update just entity, exception will come



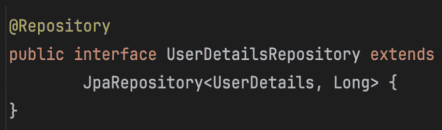
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserDetailsService userDetailsService;

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

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

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



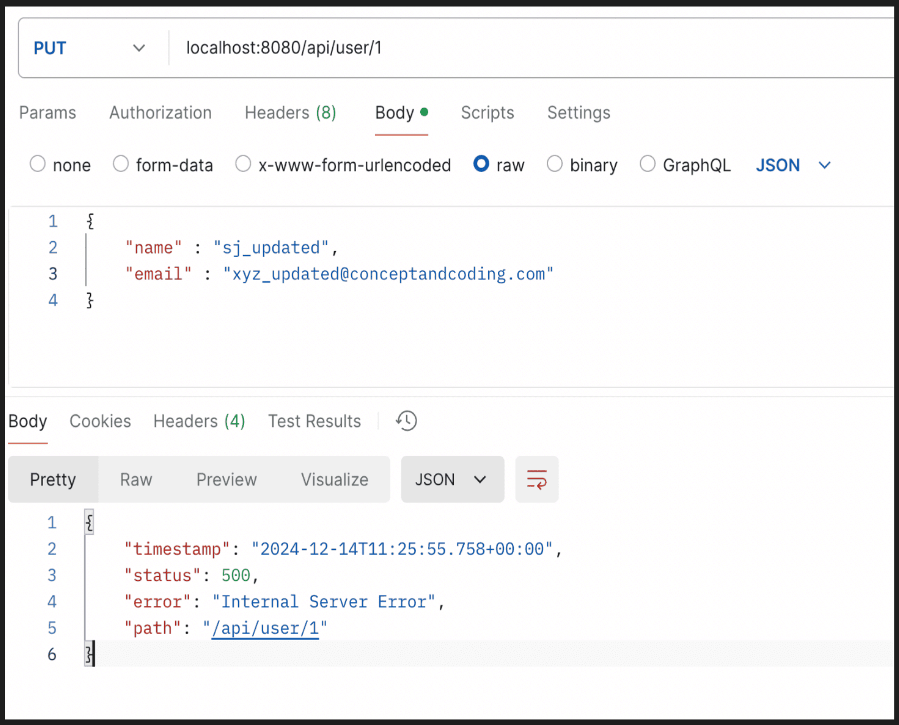
```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {

        return userDetailsRepository.save(user);
    }

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

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



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

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

    private String name;
    private String email;

    // Constructors
    public UserDetails() {
    }

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

    // Getters and setters
}
```



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



PUT localhost:8080/api/user/1

Params Authorization Headers (8) Body ● Scripts Settings

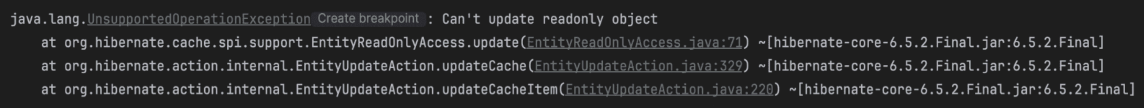
none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```json
{
    "name" : "sj_updated",
    "email" : "xyz_updated@conceptandcoding.com"
}
```

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON

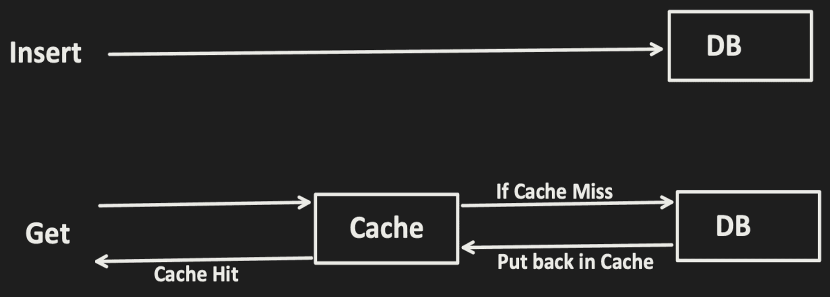
```json
{
    "timestamp": "2024-12-14T11:25:55.758+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/api/user/1"
}
```

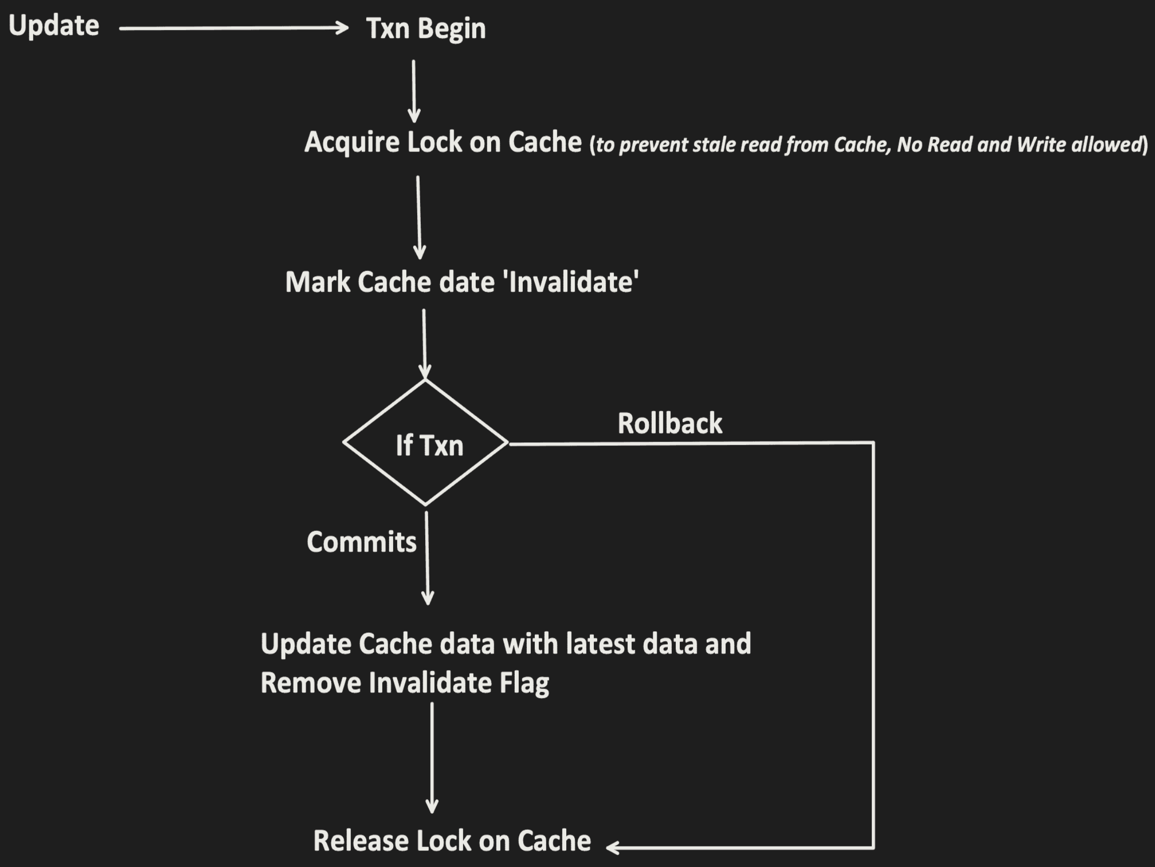


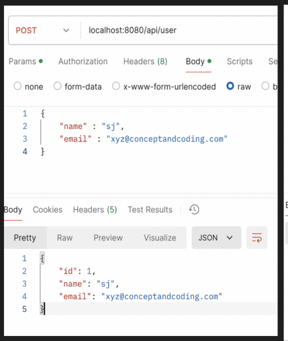
```
java.lang.UnsupportedOperationException Create breakpoint : Can't update readonly object
    at org.hibernate.cache.spi.support.EntityReadOnlyAccess.update(EntityReadOnlyAccess.java:71) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
    at org.hibernate.action.internal.EntityUpdateAction.updateCache(EntityUpdateAction.java:329)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

2. READ_WRITE

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




Update
Txn Begin
Acquire Lock on Cache (to prevent stale read from Cache, No Read and Write allowed)
Mark Cache date 'Invalidate'
If Txn
Rollback
Commits
Update Cache data with latest data and Remove Invalidate Flag
Release Lock on Cache



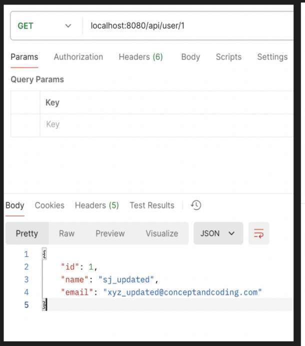
POST localhost:8080/api/user

Body: raw

```
{
    "name" : "sj",
    "email" : "xyz@conceptandcoding.com"
}
```

Response:

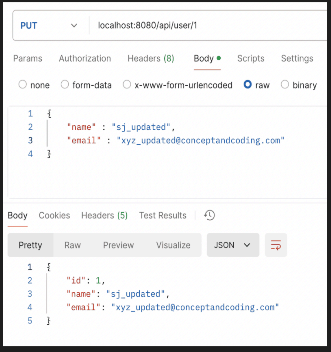
```
{
    "id": 1,
    "name": "sj",
    "email": "xyz@conceptandcoding.com"
}
```



GET localhost:8080/api/user/1

Response:

```
{
    "id": 1,
    "name": "sj_updated",
    "email": "xyz_updated@conceptandcoding.com"
}
```



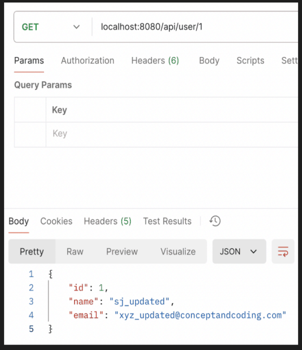
PUT localhost:8080/api/user/1

Body: raw

```
{
    "name" : "sj_updated",
    "email" : "xyz_updated@conceptandcoding.com"
}
```

Response:

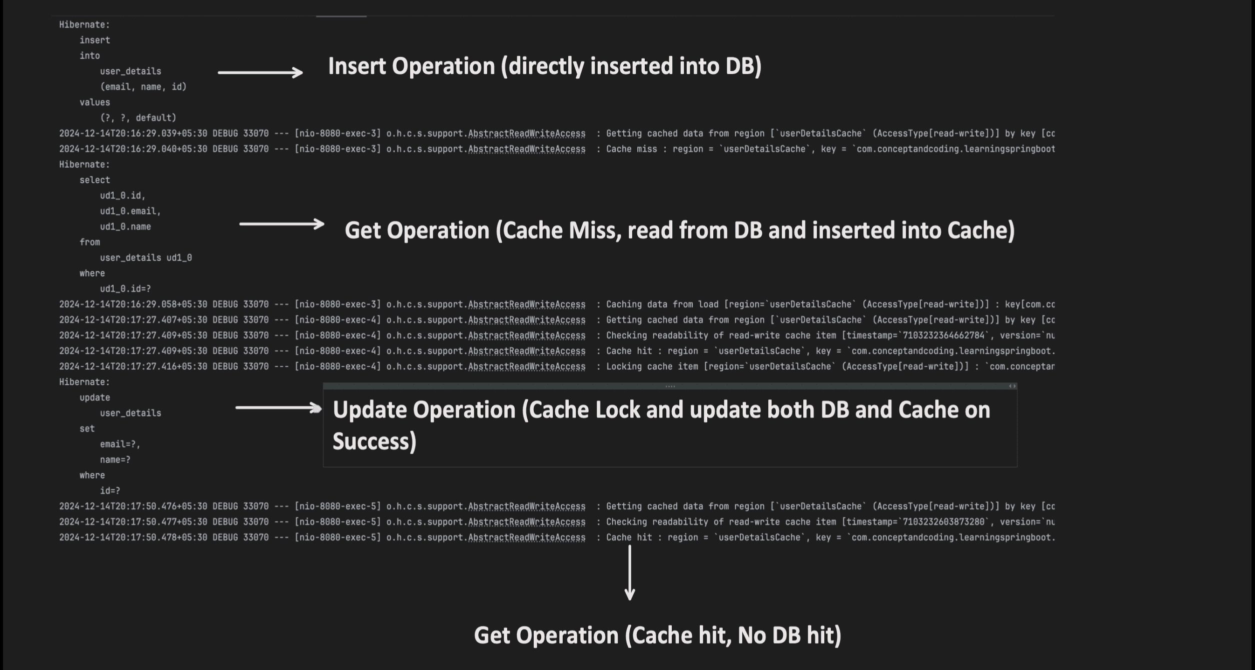
```
{
    "id": 1,
    "name": "sj_updated",
    "email": "xyz_updated@conceptandcoding.com"
}
```



GET localhost:8080/api/user/1

Response:

```
{
    "id": 1,
    "name": "sj_updated",
    "email": "xyz_updated@conceptandcoding.com"
}
```



```
Hibernate:
    insert
    into
        user_details
        (email, name, id)
    values
        (?, ?, default)
2024-12-14T20:16:29.039+05:30 DEBUG 33070 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDetailsCache` (AccessType[read-write])] by key [cc
2024-12-14T20:16:29.040+05:30 DEBUG 33070 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Cache miss : region = `userDetailsCache`, key = `com.conceptandcoding.learningspringboot
Hibernate:
    select
        ud1_0.id,
        ud1_0.email,
        ud1_0.name
    from
        user_details ud1_0
    where
        ud1_0.id=?
2024-12-14T20:16:29.058+05:30 DEBUG 33070 --- [nio-8080-exec-3] o.h.c.s.support.AbstractReadWriteAccess  : Caching data from load [region=`userDetailsCache` (AccessType[read-write])] : key[com.cc
2024-12-14T20:17:27.407+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDetailsCache` (AccessType[read-write])] by key [cc
2024-12-14T20:17:27.409+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Checking readability of read-write cache item [timestamp=`7103232364662784`, version=`nu
2024-12-14T20:17:27.409+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Cache hit : region = `userDetailsCache`, key = `com.conceptandcoding.learningspringboot.
2024-12-14T20:17:27.416+05:30 DEBUG 33070 --- [nio-8080-exec-4] o.h.c.s.support.AbstractReadWriteAccess  : Locking cache item [region=`userDetailsCache` (AccessType[read-write])] : `com.conceptar
Hibernate:
    update
        user_details
    set
        email=?,
        name=?
    where
        id=?
2024-12-14T20:17:50.476+05:30 DEBUG 33070 --- [nio-8080-exec-5] o.h.c.s.support.AbstractReadWriteAccess  : Getting cached data from region [`userDetailsCache` (AccessType[read-write])] by key [cc
2024-12-14T20:17:50.477+05:30 DEBUG 33070 --- [nio-8080-exec-5] o.h.c.s.support.AbstractReadWriteAccess  : Checking readability of read-write cache item [timestamp=`7103232603873280`, version=`nu
2024-12-14T20:17:50.478+05:30 DEBUG 33070 --- [nio-8080-exec-5] o.h.c.s.support.AbstractReadWriteAccess  : Cache hit : region = `userDetailsCache`, key = `com.conceptandcoding.learningspringboot.
```

Insert Operation (directly inserted into DB)
Get Operation (Cache Miss, read from DB and inserted into Cache)
Update Operation (Cache Lock and update both DB and Cache on Success)
Get Operation (Cache hit, No DB hit)

3. NONSTRICT_READ_WRITE

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

4. TRANSACTIONAL

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