Spring Data in depth

2023-24

© G. Malnati, 2021-24

Spring Data in nutshell

- A general framework for persistence
  - Suitable for both SQL and NoSQL databases
- Based on the `@Repository` concept
  - An adapter that takes domain objects (`@Entity`, `@Document`, …) and makes specific calls to the persistence layer, implementing CRUD operations
- Repositories are defined as interfaces the implementation of which is automatically generated by Spring Data
  - This avoids copying and pasting the same code over and over, possibly introducing subtle errors

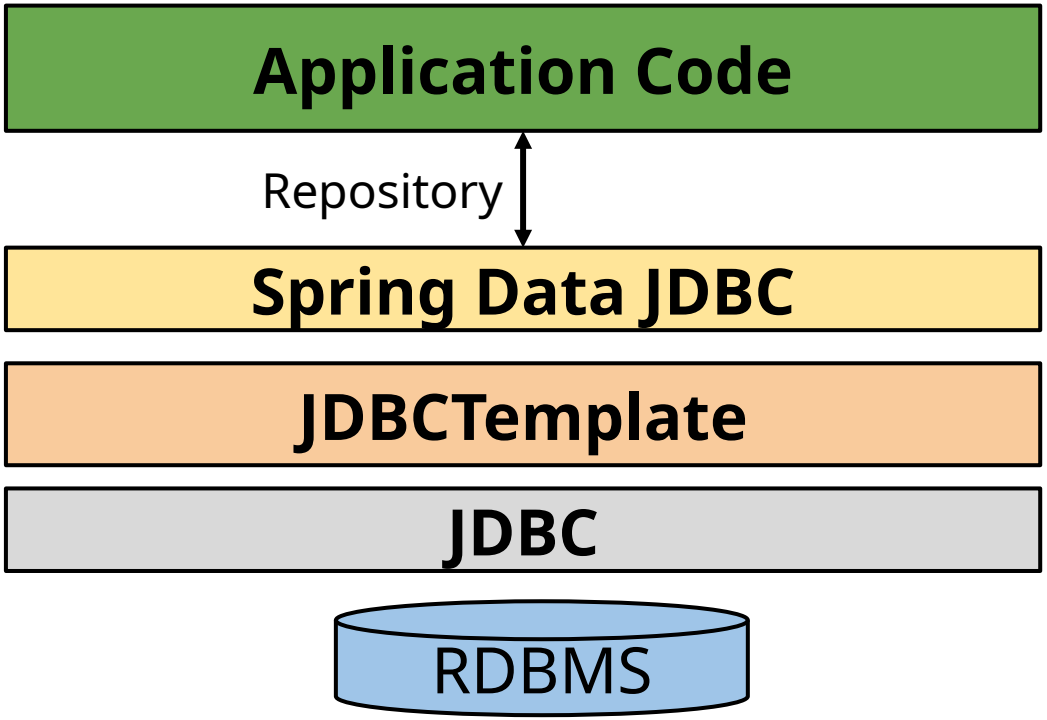
© G. Malnati, 2021-24

Defining a data persistence layer

- Spring Data assumes the presence of an underlying data persistence layer, to which actual operations are delegated
- In the case of RDBMS, there are three alternatives
  - Spring Data JDBC
  - Spring Data JPA
  - Spring Data R2DBC

© G. Malnati, 2021-24

Spring Data JDBC



© G. Malnati, 2021-24

Spring Data JDBC

- Spring Data JDBC is an object-relational mapping framework for relational databases that aims to avoid most of the complexity of other ORM frameworks
  - This is the simplest approach to persist data on a relational DBMS
- When an entity is loaded, the corresponding SQL statement is run
  - No lazy loading, no caching
- When an entity is explicitly saved, it is persisted to the DBMS
  - If no explicit save is performed, data changes are lost
  - No dirty tracking, no sessions
- Entities are mapped to tables using simple rules
  - Annotations provide limited support to customize this behaviour

© G. Malnati, 2021-24

Managing database schema

- Spring Data JDBC offers no support for automatically deriving database schema from entity definition
  - All tables need to be created explicitly beforehand
  - No verification of conformity of the the entity structure is performed, either
- Entities are classes containing a field annotated with **`@Id`**
  - **`@Table`** and **`@Column`** annotations can provide extra details about mapping names between the to realms
- Aggregate roots are recognized by the existence of a corresponding **`@Repository`** interface
  - These can extends standard repositories interfaces and contain custom methods, that must be prefixed with **`@Query`** annotations encapsulating plain SQL statements

© G. Malnati, 2021-24

Persisting entities

- Entities are persisted via the Repository `save(…)` method
  - If the aggregate is new, this results in an insert for the aggregate root, followed by insert statements for all directly or indirectly referenced entities
  - If the aggregate root is not new, all referenced entities get deleted, the aggregate root gets updated, and all referenced entities get deleted and inserted again
- Relationships between different aggregates are manipulated explicitly, using external keys
  - No support is provided for automatic implementation of this kind of relationships

© G. Malnati, 2021-24

Retrieving entities

- Repositories offer several methods to fetch data from the underlying database
  - The Spring Data object mapping module is responsible for creating instances of domain objects and map fetched data onto their properties
  - This impacts creating instances via constructors and populating their properties
- Spring Data JDBC fully supports Kotlin data classes and immutability

© G. Malnati, 2021-24

Entity creation and mapping

- Whenever Spring Data JDBC need to create an entity form data read in the DBMS, the following process will be followed
  - If there's a no-argument constructor, it will be used. Other constructors will be ignored
  - If there's a single constructor taking arguments, it will be used
  - If there are multiple constructors taking arguments, the one to be used by Spring Data will have to be annotated with `@PersistenceConstructor`

© G. Malnati, 2021-24

Entity creation and mapping

- Once an instance of the entity has been created, Spring Data populates all remaining persistent properties of that class
  - All properties that have not been initialized by the constructor are set
  - If a property is immutable, but the class offers a corresponding `with…` method, this will be used to derive a new instance containing the corresponding property
  - If not set by the constructor, the ID property is always populated first

© G. Malnati, 2021-24

Supported property types

- All primitive types, enums and strings
- Date and time (including `LocalDateTime` and derived)
- References to other entities
  - They are considered one-to-one relationships
  - The table of the referenced entity must have a column named as the referencing entity
- `Set<some_entity>`
  - This is considered a one-to-many relationship
  - The table of the referenced entity must have a column named as the referencing entity (as above)

© G. Malnati, 2021-24

Supported property types

- **`Map<simple_type, some_entity>`** is considered a qualified one-to-many relationship
  - The table of the referenced entity is expected to have two additional columns: one named the same as the table of the referencing entity for the foreign key and one with the same name and an additional _key suffix for the map key
- **`List<some_entity>`** is interpreted as a **`Map<Integer, some_entity>`**
- No support is available for many-to-one or many-to-many relationships
  - These are, by definition, different aggregates and references to them are encoded as simple id values (foreign keys)

© G. Malnati, 2021-24

Low-level mechanisms

- Internally, Spring Data JDBC repositories are implemented by proxying/extending the class `SimpleJdbcRepository<Entity, ID>`
  - Instances of this class gets a reference to a `JdbcTemplate` instance
- `JdbcTemplate` is the engine that powers all the JDBC workflows, simplifying DB access and helping avoiding common errors
  - It executes core JDBC operations (either single or batched) leaving application code to provide SQL and extract results
  - It can be used independently of Spring Data JDBC, if no repository support is needed

© G. Malnati, 2021-24

Support for DDD in Spring Data JDBC

- Spring Data JDBC allows to use the persistence layer as requested by DDD requirements
  - Aggregate roots are entities that contain references to their dependent entities, which can form a 1-1 or 1-N relationship
  - In the former case, the dependent entity is modelled as a simple property of the aggregate root, in the latter one it is modelled as a collection (preferably a **`Set`**) of them
- Spring Data JDBC knows a class is an aggregate root when it contains a repository for that class
  - Since the aggregate entities are connected through object references, Spring Data JDBC also knows what the aggregates are and can transfer data to the database as aggregates
  - In order to insert or update data, the entire aggregate needs to be saved

© G. Malnati, 2021-24

Support for DDD in Spring Data JDBC

- Whenever an aggregate root object is saved or deleted, it has the opportunity to publish a domain event
  - In order to support such a process, two method-level annotations are provided
- A method annotated with **`@DomainEvents`** can return either a single event instance or a collection of events
  - It must not take any arguments
  - Its implementation is in charge of returning the events that need to be published, as a consequence of the last evolution of the object
- After all events have been published, if there is a method annotated with **`@AfterDomainEventPublication`**, it will be invoked
  - This can be used to clean the list of events to be published

© G. Malnati, 2021-24

Support for DDD in Spring Data JDBC

```
data class Warehouse(@Id val id: String, val location: String) {
  @MappedCollection
  private val inventoryItems = mutableSetOf<InventoryItems>()
  fun addInventoryItem(inventoryItem: InventoryItem) {
      _domainEvents.add(InventoryItemAdded(this, inventoryItem))
      inventoryItems.add(inventoryItem)
  }

  private val _domainEvents = mutableListOf<Any>()
  @DomainEvents
  fun domainEvents(): List<Any> = _domainEvents

  @AfterDomainEventPublication
  fun cleanup() {
    _domainEvents.clear()
  }
}

data class InventoryItem (@Id val id: String, val name: String, val count: Int)
```

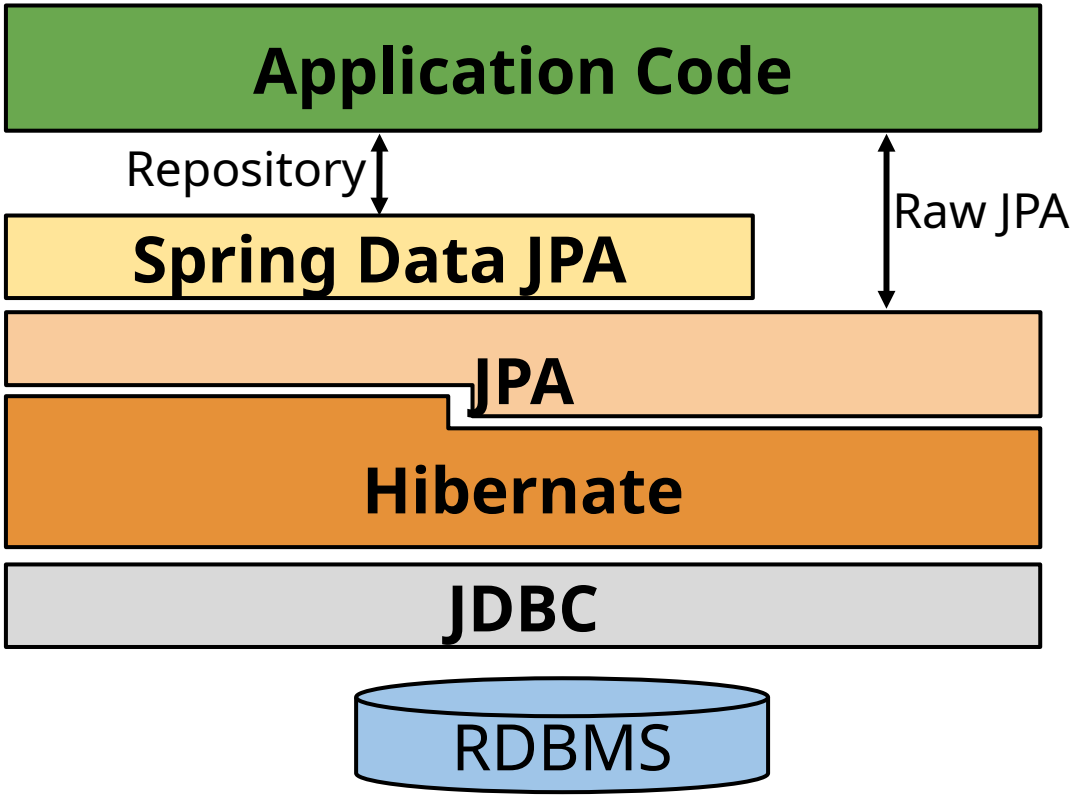

© G. Malnati, 2021-24

Spring Data JPA

- While extremely powerful, the concept of entity supported by JPA is based on different assumptions from those used by DDD
  - JPA entities have less restrictions than in DDD
  - There is no explicit support for implementing aggregates and aggregate roots
- Spring Data wraps the JPA specifications in a more general framework which can make the implementation of DDD abstractions easier to be developed
  - Interface **`Persitable`**, that denotes something that has an ID and should be persisted in a DBMS, and class **`AbstractAggregateRoot`**, that provides method for dispatching domain events whenever the entity is saved

© G. Malnati, 2021-24

Spring Data JPA



© G. Malnati, 2021-24

JPA - Jakarta (Java) Persistence API

- Specification defines a standard for management of persistence and object to relational mapping in Java environments
  - Originally released in 2006, it underwent several updates: current version is 3.1 (March 2022)
- An ORM (Object-to-Relational-Mapping) is a software layer that (automatically) converts tables rows into object instances and vice-versa
  - Keeping the two representations in sync
  - Reference ORM implementation is provided by JBoss Hibernate (v6.2)

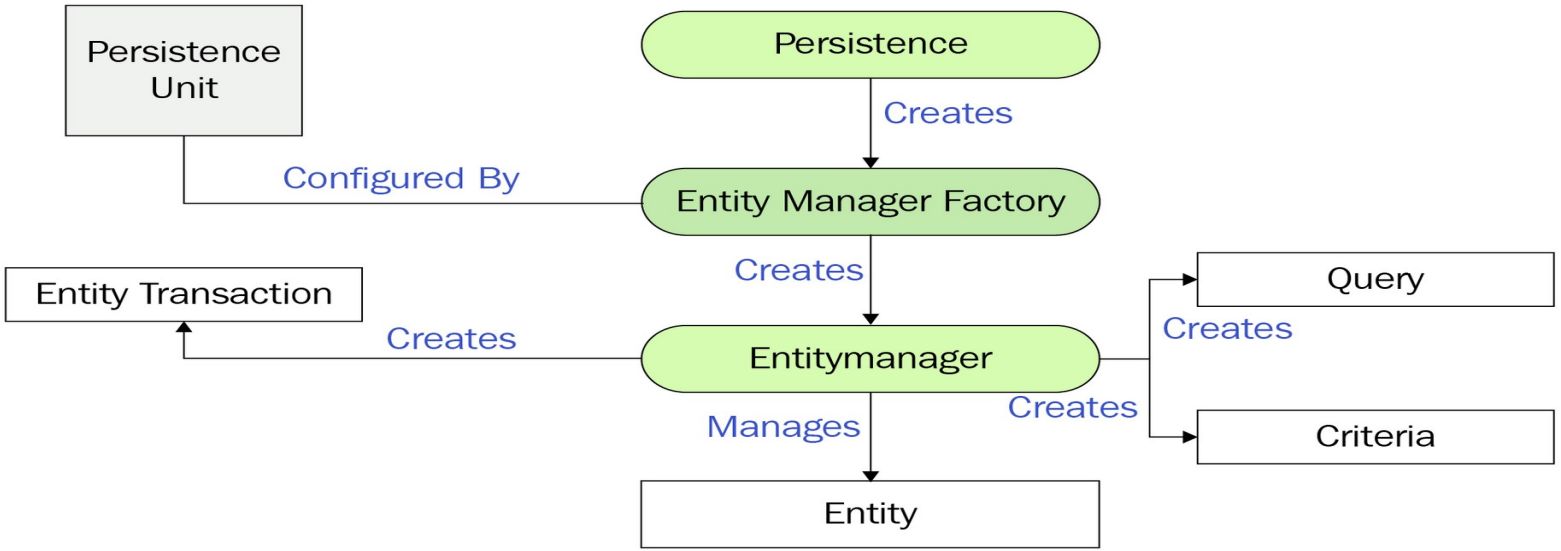
© G. Malnati, 2021-24

JPA – Jakarta (Java) Persistence API

- A generic architecture that provides developers with all the required operations and techniques for mapping objects from and into a relational database
  - Specifies a set of behaviours in terms of interfaces
  - Requires a concrete implementation, providing the actual classes supporting the interfaces
- Hibernate is a concrete implementation of JPA
  - But it offers also a set of additional features, based on a proprietary API
  - These may matter a lot when high performances are a must in the application development

© G. Malnati, 2021-24

JPA architecture



© G. Malnati, 2021-24

JPA interfaces and concepts

- **`PersistenceProvider`**
  - A concrete software module conforming to the JPA specification (i.e., Hibernate, by default)
- **`EntityManagerFactory`**
  - A singleton object responsible of creating Entity Managers and configuring them to communicate with a specific database, as described by the persistence unit
- **`EntityManager`**
  - An object that encapsulates a connection to a database, offering the actual methods for persisting and retrieving the entities mapped on that DBMS

© G. Malnati, 2021-24

JPA interfaces and concepts

- ***Entities***
  - Classes that represent a domain object in the application
  - Each entity is represented in the DBMS with a table
  - A specific instance of an entity corresponds to a record in that table
- ***Entity transaction***
  - A database transaction that can be either committed or rolled back according to the application state
  - Any operations (query, insert, update, delete) should be performed within the boundaries of an entity transaction

© G. Malnati, 2021-24

JPA interfaces and concepts

- **`Query`**
  - An object that encapsulate a custom query
  - JPA provides a custom query language (named JPQL) that can used to perform queries with object-oriented concepts
- ***Persistence unit***
  - Defines a persistence context, describing connection information to the DBMS, involved entities, and other useful configuration information

© G. Malnati, 2021-24

Schema ownership

- The duality in data representation (Entities vs tables) allows each of the two sides to push changes to the opposite one
  - While JPA allows for many different solutions, in real applications the schema does belong to the DBMS
  - Thus, the database schema should not be updated as a consequence of (possibly involuntary) changes in the code base, but always checked against the entities structure to verify any inconsistencies
  - This is achieved by setting the `spring.jpa.hibernate.ddl-auto=validate` property
- The DBMS schema must be part of the source code
  - Any variation of it should be included in the Version Control System repository and committed

© G. Malnati, 2021-24

Entity configuration

- Each class defining an entity is annotated with `@Entity`
  - By default, it is matched to a table having a corresponding (lower-case) name in the DBMS
  - This may be changed, using annotation `@Table(name="table_name")`
- The properties of an entity are mapped to table columns, based on their type and annotations
  - Attribute `@Column(…)` can be used to provide custom details on the column naming
  - As well as to express extra constraints, like nullability and uniqueness

© G. Malnati, 2021-24

Entity constraints in Hibernate

- An entity must be annotated with the `@Entity` annotation
- It must have a public or protected no-arg constructor
  - Other constructors are allowed
- It should not be final, nor its method/properties should be final
- It may extend a non-entity class
  - If this is labelled with `@MappedSuperclass`, its properties become part of the current entity
- It must provide an `@Id`-labelled property
- It must provide useful implementations for `equals(…)` and `hashCode()`

© G. Malnati, 2021-24

Entity constraints in Hibernate

- The need of having the entity class open, makes it difficult to use data classes
  - As the official Spring guide for Kotlin says (https://github.com/spring-guides/tut-spring-boot-kotlin#persistence-with-jpa):

> Here we don’t use data classes with val properties because JPA is not designed to work with immutable classes or the methods generated automatically by data classes. If you are using other Spring Data flavor, most of them are designed to support such constructs so you should use classes like data class User(val login: String, …) when using Spring Data MongoDB, Spring Data JDBC, etc.

Entity constraints in Hibernate

- The **`jpa`** Gradle plugin provides a no-args constructor for all classes labelled with **`@Entity`**, **`@Embeddable`**, or **`@MappedSuperclass`**
  - This allows Hibernate to instantiate classes and use Kotlin non-nullable properties with JPA
- The allopen Gradle plugin helps meet the non final constraints on methods and classes
  - This allows Hibernate to generate proxies for lazy-loading the entity

```
plugins { //other plugins…
  id("plugin.jpa") version "1.9.22"
  id("plugin.allopen") version "1.9.22"
}
```

Entity constraints in Hibernate

- In order to improve performances, Hibernate can lazy load instances via runtime proxies
  - Synthesized classes that extends the original entity
  - The first time any of their method/properties is invoked, the actual entity is fetched, thus avoiding round-trips with the DBMS if their content is never accessed
- This happens when an entity is connected to other entities via relationships
  - The relationship annotation may contain a fetch attribute, specifying the policy to be adopted

```
@ManyToOne(fetch = FetchType.LAZY)
var department: Department? = null
```


© G. Malnati, 2021-24

Identifying entities

- Each entity must have one property labelled with **`@Id`**, representing the primary key
  - The corresponding column in the DBMS must be both unique and non null
- The primary key can have a meaning in the domain
  - This is called a **natural ID**
  - Or it can be generated synthetically: this is called a **surrogate ID**
- The **`@Id`** property can be temporarily null, if it is a surrogate
  - As long as the object has not yet been stored
  - And provided that the property is annotated with **`@GeneratedValue(strategy = …)`**

© G. Malnati, 2021-24

Identifying entities

- Natural keys may be composed of multiple columns
  - Either the corresponding columns are all labelled with `@Id`
  - Or a single property labelled with `@EmbeddableId` is introduced
  - In the latter case, the property should refer to a class marked with `@Embeddable`, listing all the columns that are part of the natural key
- Compound keys might incur in performance penalties
  - Since they require multi-column joins
- When working in Kotlin, always prefer a surrogate key to a natural one

© G. Malnati, 2021-24

Natural keys

- If using natural keys, it is necessary to specify that they are mandatory and immutable

```
@Entity
class Author (

  @Id
  @Column(updatable = false, nullable = false)
  val id: String,

  //… other properties

) {
  // methods
}
```

Generating surrogate keys

- If an `@Id` property is annotated with `@GeneratedValue`, the implementation is in charge of assigning it a value, the first time it is stored
  - You can specify the generation strategy
- `GenerationType.AUTO`
  - Lets the persistence provider choose the generation strategy (default behaviour)
- `GenerationType.IDENTITY`
  - Relies on an auto-incremented value bound to the the column
  - Forces Hibernate to perform insertion immediately, possibly preventing some forms of optimizations (batch inserts)

© G. Malnati, 2021-24

Generating surrogate keys

- **`GenerationType.SEQUENCE`**
  - Relies on a database sequence to generate unique values
  - Such a query is not bound to a specific table and the returned value can be incremented even if no element is inserted

```
CREATE SEQUENCE sequence_1 start with 1 increment by 1;

CREATE TABLE students ( ID number(10), NAME char(20) );

INSERT into students VALUES(sequence_1.nextval,'John');
INSERT into students VALUES(sequence_1.nextval,'Mary');

 ______________________
| ID  |      NAME      |
------------------------
|  1  |     John       |
|  2  |     Mary       |
 ----------------------
```


© G. Malnati, 2021-24

Generating surrogate keys

```
@Entity
@Table(name = "students")
class Student (
  val name: String,
  // other properties…

  // last one is id with default value
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
                  generator = "student_generator")
  @SequenceGenerator(name="student_generator",
    sequenceName = "sequence_1",
    initialValue = 1,
    allocationSize = 1)
  @Column(updatable = false, nullable = false)
  var id : Long? = null,
)
```


© G. Malnati, 2021-24

Generating surrogate keys

- **`GenerationType.TABLE`**
  - Selects the key from a table in the DBMS
  - Obsolete strategy that introduces a large overhead when saving records
  - Almost all modern DBMS do support some better technique
- If the key type is not a numeric one (Int, Long, BigInteger) but UUID, hibernate generates the key value at application level, providing a pseudo-random value
  - Key property must be labelled with `@GeneratedValue(generator = "uuid2")` and `@GenericGenerator(name = "uuid2", strategy = "uuid2")`
  - Largely improves multi-master DBMS performances
  - Creates inefficiency in indices, since UUIDs form a sparse set

Providing surrogate keys

```
@Entity
@Table(name = "students")
class Student (
    val name: String
    //other properties

    @Id
    @Column(columnDefinition = "BINARY(16)",
            updatable = false, nullable = false)
    @GeneratedValue(generator = "uuid2")
    @GenericGenerator(name="uuid2", strategy="uuid2")
    var id : UUID? = null,

) {
  //other methods
}
```


© G. Malnati, 2021-24

Implementing equals and hashCode

- When an entity has ***xToMany*** relationships with an other entity, it is convenient to declare the corresponding property as a `(Mutable)Set<TargetEntity>`
  - Unfortunately, Hibernate guarantees equivalence of persistent identity (database row) and Java identity only inside a particular session scope
  - So, as soon as instances retrieved in different sessions are mixed, a proper implementation of `equals(…)` and `hashCode()` must be provided to have meaningful semantics for sets
- Although the `@Id` field looks as a valid solution for implementing equality test and hashCode, problems may arise
  - Especially if the id is a `@GeneratedValue`
  - While an entity has not yet been persisted, its is is null and comparison should be based on address equality

© G. Malnati, 2021-24

Implementing equals and hashCode

- Usually, equality should obey to the following constraints, in priority order
  - If the other object is null, return false
  - It the other object is identical to this, return true
  - If the other object has a different class, return false
  - If the current id is null, return false
  - Otherwise return the result of comparing the two ids
- The existence of proxies may make the comparison more difficult
  - We need to extract the actual class of the other element, via `ProxyUtils.getUserClass(other)`

© G. Malnati, 2021-24

Implementing equals and hashCode

- As per Java (and Kotlin) specifications, equals and hashCode should be coherent
  - If two objects are equal, they should return the same value from **`hashCode()`**
  - The converse does not need to be true
- The value returned by hashCode should not change if the content of the object does not change
  - But with generated ids, this may be a problem: when an object is persisted, it gets its own id (that was previously null), but it does not change from the perspective of the application domain
- The problem may be solved having **`hashCode()`** returning a constant value
  - This impact on the performance of hash tables (and hash sets) whose retrieve complexity moves from O(1) to O(N)

© G. Malnati, 2021-24

Creating a general solution

- The many constraints that an entity must satisfy can be approached by designing a generic abstract class that encapsulate the `@Id` property
  - It will be labelled with `@MappedSuperclass`
  - It is responsible to implement `equals(…)` and `hashCode()` (and, possibly, `toString()`)
- And having the actual entities derive from such a class
  - Thus benefiting from the shared implementation of the helper methods, preventing a lot of boilerplate code
- Note that, in order to get access to reflection (used in the following example for implementing the toString() method), the following dependency need to be added to the project
  - `implementation(kotlin("reflect"))`

© G. Malnati, 2021-24

Creating a general solution

```
@MappedSuperclass
abstract class EntityBase<T: Serializable> {
    companion object {
        private const val serialVersionUID = -43869754L
    }

    @Id
    @GeneratedValue
    private var id:T?  = null

    fun getId(): T? = id

    override fun toString(): String {
        return "@Entity ${this.javaClass.name}(id=$id)"
    }
    //continues in next slide…
```


© G. Malnati, 2021-24

Creating a general solution

```
    //continues from previous slide…

    override fun equals(other: Any?): Boolean {
       if (other == null) return false
       if (other === this) return true
       if (javaClass != ProxyUtils.getUserClass(other))
           return false
       other as EntityBase<*>
       return if (null == id) false
              else this.id == other.id
    }

    override fun hashCode(): Int {
        return 31 //any value will do
    }
}
```


© G. Malnati, 2021-24

Creating a general solution

- The EntityBase class is generic
  - It allows subclasses to specify the type of the primary key to be used
- The class is labelled with **`@MappedSuperclass`**
  - Thus, its only property (id) becomes part of the subclasses
  - Method **`getId()`** allows subclasses to access the value of the id, but not to change it
- Entities are declared in the following way

```
@Entity class Course(
    val name: String
): EntityBase<Long>() { }
```

Mapping object properties

- Per each property in the Entity class, the following items can be specified via the `@Column` annotation
  - The name of the DBMS column onto which the value will be mapped, (via value)
  - The SQL type to choose (via columnDefinition)
  - Further constraints (insertable, updatable, nullable, unique, length, precision, scale)

© G. Malnati, 2021-24

Mapping object properties

- Property type is used to understand how the property will map onto the table
  - Elementary types (**`Boolean`**, **`Int`** and its derivatives, **`Float`**, **`Double`**, **`BigInteger`**, **`BigDecimal`**, **`String`**, **`Enums`**, **`Date`**, **`Time`**, **`Timestamp`** and derivatives) are directly mapped to their column and managed as value types
  - Types marked with the **`@Embeddable`** annotation are exploded in their elementary properties and added to the current table
  - Types marked with **`@Entity`** or that extend **`Collection<T>`** (where class **`T`** is marked with **`@Entity`**) denotes relationships: this must be marked with some special annotation to express their cardinality: **`@OneToOne`**, **`@ManyToOne`**, **`@OneToMany`**, **`@ManyToMany`**

© G. Malnati, 2021-24

Date and time mapping

- In a typical relational database, there are three basic data types for columns:
  - Date, time, timestamp
- All of these three data types are represented as instances of java.util.Date object
  - `@Temporal` annotation is used to disambiguate it
  - `TemporalType.DATE`: represents date-only values
  - `TemporalType.TIME`: represents time-only values
  - `TemporalType.TIMESTAMP`: represents date with time values
- Internally, a `Date` only stores a long number
  - It represents the number of milliseconds from the Epoch (1970-01-01 00:0:00.000 UTC)
  - In order to represent it in local date and time, a time zone (offset) is needed: this must be stored in a different field

© G. Malnati, 2021-24

Relationships

- In a DBMS, relationships are implemented by referencing the primary key, via foreign keys
  - Given the schema and a record, it is possible to reach all other records related to it
- Navigation from entity to entity may happen in three ways:
  - By directly selecting the corresponding row, if the given record contains a foreign key (1-to-1, many-to-1)
  - By selecting all rows of the target table that refer to the primary key of the given record (1-to-many)
  - By performing a inner join on a secondary table, in order to select those rows that are related to the primary key of the given record (many-to-many)

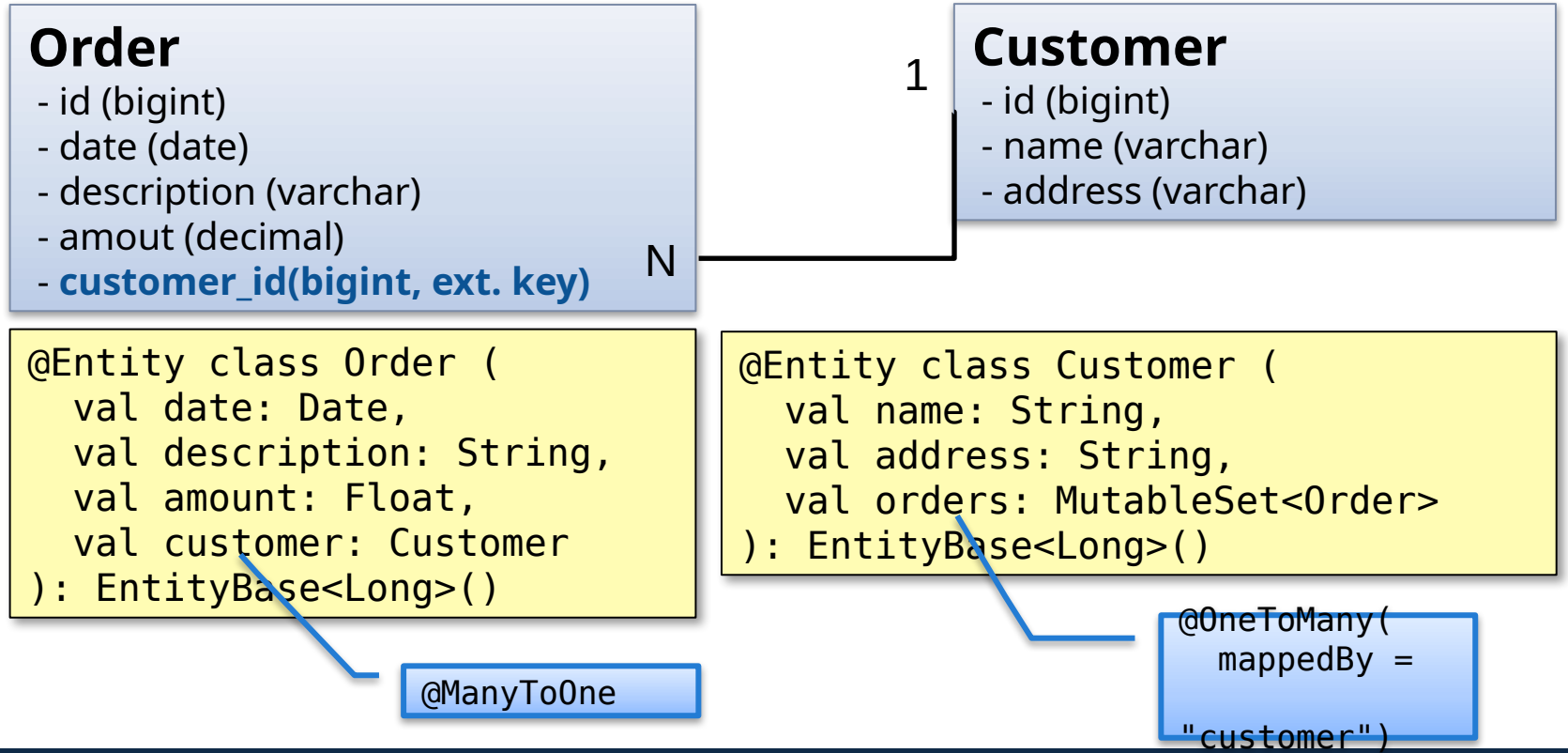
© G. Malnati, 2021-24

Relationships

- In the application code, relationship are modelled via pointers
  - These are intrinsically unidirectional
- When the application logic requires navigating a relationship along both directions, it must be defined in both entities
  - Possibly referring to an intermediate join table
- Whenever the relationship changes as a consequence of the application code execution, changes must be propagated on the opposite site
  - If this is mapped as bi-directional

© G. Malnati, 2021-24

Relationships



© G. Malnati, 2021-24

Relationships

```
@Entity class Customer (
  val name: String,
  val address: String,

  @OneToMany(mappedBy="customer")
  val orders = mutableSetOf<Order>(),
) : EntityBase<Long>

{

  fun addOrder(o: Order) {
      o.customer = this;  // maps the other side
                          // of the relationship
      orders.add(o);
  }

}
```


© G. Malnati, 2021-24

@OneToOne

- Each instance of the first entity is related to (exactly/at most) one instance of the second entity
  - **Employee** ←→ **EmployeeDetails**

```
@Entity
class Employee (
  @OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)
  var employeeDetails: EmployeeDetails?,
): EntityBase<Long> ()
```

```
@Entity
class EmployeeDetails (
  @OneToOne
  var employee: Employee,
  @Id var id: Long? =null
){}
```

```
@Entity
class EmployeeDetails (
  @OneToOne
  @MapsId
  val employee: Employee;
) {}
```

@OneToMany

- Each instance refers to a collection of entities
  - Stored in a (Mutable)Set
  - Referenced entities must have a working implementation of equals and hashCode

```
@Entity
class Department (val name: String ): EntityBase<Long>() {

    @OneToMany(mappedBy = "department")
    val employees = mutableSetOf<Employee>()

    fun addEmployee(e: Employee) {
        e.department = this; //map the reverse side of the relationship
        employees.add(e)
    }
}
```


© G. Malnati, 2021-24

@ManyToOne

- The current entity is related to (at most / exactly) one other entity
  - A join column is added to the table, whose name defaults to the **<field_name>_<other_id_field>**
  - This can be overridden via the **@JoinColumn** annotation

```
@Entity
class Employee (
    var name : String,

    @ManyToOne
    var department: Department? = null

): EntityBase<Long>()
```


© G. Malnati, 2021-24

@ManyToMany

- These relationships require a join table
  - Where pairs of related keys are stored

```
@Entity
class Student( val name:String): EntityBase<Long>() {
    @ManyToMany
    @JoinTable(name="student_course",
       joinColumns = [JoinColumn(name="student_id")],
       inverseJoinColumns = [JoinColumn(name="course_id")]
    )
    val courses: MutableSet<Course> = mutableSetOf()

    fun addCourse(c: Course) {
        courses.add(c)
        c.students.add(this)
    }
}
```


© G. Malnati, 2021-24

@ManyToMany

- Both sides may offer convenience functions to manipulate the relationship
  - Owning side references the table, the owned one refers to the other side via the **mappedBy** attribute

```
@Entity
class Course(val name:String): EntityBase<Long>() {

    @ManyToMany(mappedBy = "courses")
    val students: MutableSet<Student> = mutableSetOf()

    fun addStudent(s:Student) {
        students.add(s)
        s.courses.add(this)
    }
}
```


© G. Malnati, 2021-24

Updating relationships

- A special situation arises when one entity is created, updated, or destroyed
  - Depending on application logic, this might require creating/updating, destroying other entities as well
- This is called **cascading** and can be controlled by the **`cascade`** attribute of the various relationship annotations
  - It express the **transitivity constraints** that the relationship enforces
  - Its value is an array of the following values
- **`CascadeType.PERSIST`**
  - Propagates the persist operation to the other side of the relationship
- **`CascadeType.MERGE`**
  - When a merge is executed (copying the properties of the object onto the corresponding record with the same id), the operation is executed also on the other entity

© G. Malnati, 2021-24

Updating relationships

- **`CascadeType.REMOVE`**
  - Removes the referred to entity when the current one is removed
- **`CascadeType.DETACH`**
  - When the current entity is detached by the persistence context, the related entity is detached as well
- **`CascadeType.REFRESH`**
  - When the object is re-read from the DBMS, the other i, too
- **`CascadeType.ALL`**
  - All of the above operation are propagated to the other side of the relationship

© G. Malnati, 2021-24

JPA operations

- In JPA, the **`EntityManager`** is the class responsible of keeping in sync entities and the database
  - This class is obtained from the **`EntityManagerFactory`**, via the **`createEntityManager()`** method
- The **`EntityManager`** will do its job of keeping entities in sync since its creation until it gets closed
  - In this timeframe, several DBMS transactions can be performed
- The current transaction is referred to by the **`transaction`** property of the **`EntityManager`**
  - It offers the **`begin()`**, **`commit()`** and **`rollback()`** methods to delimit the set of operations that must be executed with ACID properties

© G. Malnati, 2021-24

JPA entities states

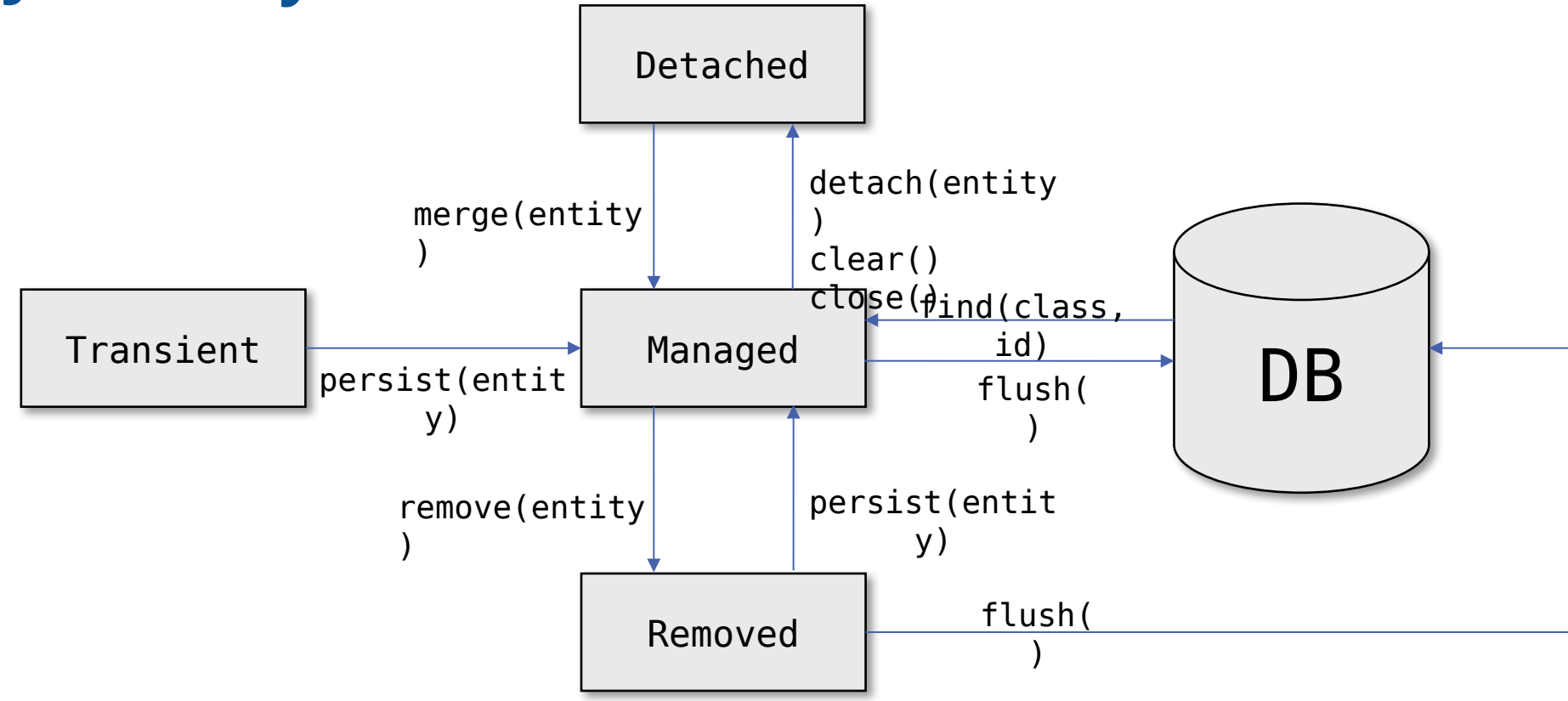
- Every entity has a state at each point, during runtime :
  - **`Managed`** – A managed entity is one that is synchronized with the database: any changes in its properties will be reflected in the database, as long as it is in this state
  - **`Detached`** – A detached entity is one that is not synchronized with the database: this happens when the **`EntityManager`** is closed, or cleared (to discard changes)
  - **`Removed`** – If a remove operation is performed, the mapped database record doesn't get removed immediately: when the next **`flush()`** operation will be issued, the SQL statement to delete it will be executed
  - **`Transient`** – When an object is created, the JPA provider has no notion of its existence, so no action can be taken to align it to the database

JPA entities states

- Changes in the state of an entity occur when methods of the **`EntityManager`** are invoked
  - **`find(…)`**, **`persist(…)`**, **`merge(…)`** cause entities to enter the **`Managed`** state
  - **`detach(…)`**, **`clear()`**, **`close()`** cause entities to enter the **`Detached`** state
  - **`remove(…)`** cause an entity to enter the **`Removed`** state
- Actual changes in the DBMS only occur when the **`flush()`** method is invoked

© G. Malnati, 2021-24

JPA entity states



© G. Malnati, 2021-24

Persisting entities

- The **`persist(…)`** method is used to insert a new record in the database
  - ```
    val m = Movie("Title")
    em.persist(m)
    ```
  - When this is executed, a row will be inserted in the movie table, with all the values that have been initialized in the instance
  - The entity m will enter the **`Managed`** state: any change to its properties will take place within the transaction boundary

© G. Malnati, 2021-24
    ```

Retrieving entities

- The `find(…)` method is used to retrieve an entity from the database, given its primary key
  - `val m = em.find(Movie::class.java, 12L);`
  - The returned instance is a managed one
  - If any change is applied to it, it will take effect in the DBMS within the bounds of the current transaction
- The `getReference(…)` method is used to create a proxy to the database record
  - `val m = em.getReference(Movie::class.java,12L);`
  - No actual data transfer will take place with the DBMS until a read operation is performed on the proxy
  - If the proxy's properties are updated, this will take effect as soon as the transaction commits

Merging entities

- The `merge(…)` method is used to update an entity in the database, starting from an unmanaged instance
  - It takes an unmanaged entity, finds the matching one in the database using its primary key, then updates the values in the database using the given one and returns a managed instance
  - The original entity stays unmanaged: any further change to it, will not propagate to the DBMS

© G. Malnati, 2021-24

Deleting entities

- The `remove(…)` method is used to delete an entity from the database
  - The argument must be a managed entity or a proxy
  - If only the primary key is available, it is convenient to get a proxy to the database record and remove that

```
val movie = em.getReference(Movie::class.java, 12L)
em.remove(movie)
```


© G. Malnati, 2021-24

Querying entities

- JPA allows writing queries using a DSL named JPQL – Java Persistence Query Language
  - This is used to perform complex queries on database entities and to perform bulk update operations
- It looks like SQL, borrowing the same syntax
  - But its expressions refer to the object properties and not the table and column names
- The `createQuery(...)` method is used to create a `Query` object

```
val query = entityManager.createQuery(
      "SELECT m FROM Movie m",
       Movie::class.java
)
query.resultList.forEach{ m -> println(m) }
```


© G. Malnati, 2021-24

Query Language syntax

- The **`FROM`** clause defines from which entities the data get selected
  - An alias (variable name) can be added to reference that specific entity inside the rest of the query
  - `val q = em.createQuery("SELECT m, d FROM Movie m JOIN m.director d")`
  - If more than one entities is selected, the returned result will be a list of **`Array<Any>`**, where the array has as many elements as the number of selected entities

© G. Malnati, 2021-24

Query Language syntax

- The **`WHERE`** clause can be added to restrict the set of returned entities
  - JPQL supports a set of basic operators to define comparison expressions, which can be combined with **`AND`**, **`OR`**, **`NOT`** to create more complex expressions
  - Supported operators include **`=`**, **`<>`**, **`>`**, **`=>`**, **`<`**, **`<=`**, **`BETWEEN … AND …`**, **`LIKE`**, **`IS NULL`**, **`IS NOT NULL`**, **`IN (…)`**, **`IS EMPTY`**, **`size(..)`**, **`MEMBER OF`**
- A set of functions can be used in the **`SELECT`** and **`WHERE`** clause to return derived values
  - Like in SQL, grouping and ordering can be specified as well

```
https://thorben-janssen.com/jpql/
```


© G. Malnati, 2021-24

Optimizing memory usage

- All managed entities are referred to by the `EntityManager`
  - They require some more memory to represent their state
- If, inside a transaction, too many persistent objects are created, memory can saturate
  - When a massive number of entities need to be created, it is necessary to adopt special care, in order to reduce the transaction span

© G. Malnati, 2021-24

Massive insertions

```
fun massiveInsert() {
  val em: EntityManager = …

  em.transaction.begin();
  for (i in 1..1_000_000) {
      val point = Point(i, 1_000_000-i)
      em.persist(point)
      if ((i % 10_000) == 0) {
          em.transaction.commit()
          em.clear()
          em.transaction.begin()
      }
  }
  em.transaction.commit();
}
```


© G. Malnati, 2021-24

Paging results

- The number of records retrieved by a query can be limited,
  - Setting properties maxResults and firstResult
  - This allows paging

```
val query=em.createQuery(
             "SELECT e FROM SomeEntity e",
             SomeEntity::class.java)
query.maxResults = noOfRecordsPerPage
query.firstResult = (pageIndex * noOfRecordsPerPage)

val l=query.resultList
```


© G. Malnati, 2021-24

JPA Repositories

- When using SpringData JPA, repositories instances are implemented as proxy of class **`SimpleJpaRepository`**
  - It encapsulates a reference to an **`EntityManager`** that will track objects of the given type
  - A repository offers a more sophisticated interface than the **`plain EntityManager`**
- Saving and updating entities is achieved via the **`save()`** method
  - This will invoke either **`em.persist(…)`** or **`em.merge(…)`** depending if the entity is new or not
  - State detection is based on **`version`** and **`id`** property value

© G. Malnati, 2021-24

Querying repositories

- Repositories offer a bunch of ready-made query methods, covering general cases
  - More can be added, specifying the query manually, via the `@Query` annotation, or having it being derived from the method name
- `@Query` annotations, actually, can also be used for introducing custom update and delete methods
  - The extra annotation `@Modifying` need to be added

© G. Malnati, 2021-24

Transactions

- By default, provided CRUD methods on repository instances are transactional
  - For read operations, the transaction configuration **`readOnly`** flag is set to true, thus informing the **`transactionManager`** that no change will take place
- Custom methods should be explicitly marked with **`@Transactional`**
  - Possibly specifying the attribute **`readOnly=true`**, if it is a read operation
- If the transaction need to span across several methods or repositories, a **`@Service`** method labelled with **`@Transactional`** must be introduced

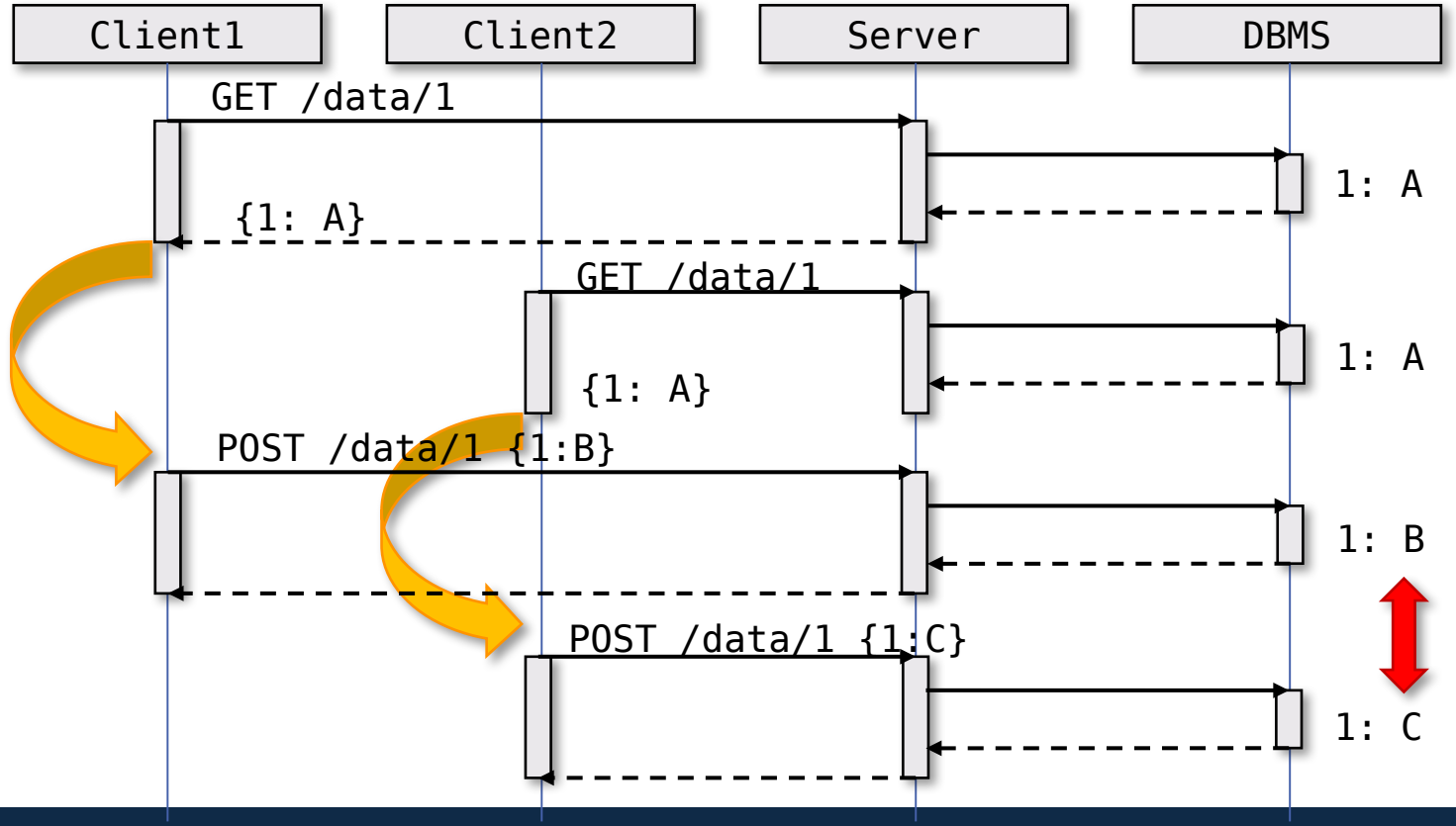
© G. Malnati, 2021-24

Locking

- If the application has concurrent writers to the same objects, then a locking strategy is critical in order to prevent data corruption
  - Locking assumes a co-operative approach to maintain data integrity
  - It may involve application level changes, and ensuring other applications accessing the database also do so correctly for the locking policy being used
- Setting the transaction isolation strategy may not be enough for a typical web application

© G. Malnati, 2021-24

Locking



© G. Malnati, 2021-24

Locking

- JPA offers two locking strategy to manage concurrency
  - Optimistic locking and pessimistic locking
- Optimistic locking relies on entities having a numerical or timestamp field labelled with `@Version`
  - Its value will be automatically managed by the repository implementation
  - When a read operation is performed, the field is populated from the database
- When a write operation is performed, an extra check is added, in the transaction commit code, to verify if the version has changed
  - In case of mismatch, an `OptimisticLockException` will be thrown
  - Otherwise the transaction commits and increments the version

© G. Malnati, 2021-24

Locking

- In case of pessimistic locking, JPA creates a transaction that obtains a lock on the data until the transaction is completed
  - This prevents other transactions from making any updates to the entity until the lock is released
  - But introduces potentially long delays in concurrent transactions, thus lowering the overall scalability of the system
- If a lock cannot be obtained (because another lock is in place), different exceptions can be thrown
  - `PessimisticLockException` causes a transaction-level rollback
  - `LockTimeoutException` causes a statement-level rollback
  - `PersistenceException` causes a transaction-level rollback
- Pessimistic locks will be automatically released when the transaction commits or roll-backs

© G. Malnati, 2021-24

Locking

- JPA provides three pessimistic locking modes
  - **`PESSIMISTIC_READ`** obtains a long-term read lock on the data to prevent the data from being updated or deleted: other transactions may read the data during the lock, but will not be able to modify or delete the data
  - **`PESSIMISTIC_FORCE_INCREMENT`** obtains a long-term read lock on the data to prevent the data from being updated or deleted; it also increments the value of **`@Version`** property
  - **`PESSIMISTIC_WRITE`** obtains a long-term read lock on the data to prevent the data from being read, updated or deleted

© G. Malnati, 2021-24

Locking

- To specify the lock mode to be used, the `@Lock(LockModeType)` annotation is added on repository query methods

```
interface UserRepository: Repository<User, Long> {

  // Plain query method
  @Lock(LockModeType.READ)
  fun findByLastname(lastname:String): List<User>

  // Redeclaration of a CRUD method with locking
  @Lock(LockModeType.READ)
  fun findAll(): List<User>;
}
```

Resolving concurrency conflicts

- In order to deal with concurrency conflicts a proper compromise must be achieved
  - Between overall data-coherence and system usability
- If optimistic locking is used, versions must be propagated to clients along with data
  - The client will send it back to the server that will check the version and commit the transaction
  - In case of **`OptimisticLockException`**, the conflict should be reported back to the user, who is the only one who can safely decide what to do

`https://en.wikibooks.org/wiki/Java_Persistence/Locking`

Auditing

- JPA supports auditing by inserting extra properties and annotations on entities
  - If a temporal property is labelled with **`@CreatedDate`** or **`@LastModifiedDate`**, its value will be automatically populated whenever the corresponding event takes place
  - If the Spring Security infrastructure is in place and a security context storing the currently authenticated user is available, properties annotated with **`@CreatedBy`** and **`@LastModifiedBy`** will be populated with the corresponding data fetched from the **`Principal`**

© G. Malnati, 2021-24

Spring Data JDBC vs Spring Data JPA

- Entity classes are not marked with **`@Entity`**
  - Relationships are not annotated with **`@OneToOne`**, **`@OneToMany`**, **`@ManyToOne`**, **`@ManyToMany`**
- Spring Data JDBC recognizes aggregates roots from repositories
  - All classes that are targeted by a **`@Repository`** interface are considered roots
  - Since the aggregate entities are connected through object references, Spring Data JDBC also knows what the aggregates are and can transfer them to the database
- No automatic derivation of the DBMS schema is performed
  - Classes must be derived from existing tables, not the other way round

© G. Malnati, 2021-24