Advanced data manipulation

Agenda

- Entities in Jmix
- Entity annotations and attributes
- Traits
- Soft Delete
- JPA Callbacks and entity events
- Data Stores
- DB Versioning and Liquibase
- DataManager and data security
- EntityManager
- Transactions manager
- Data loading
- Entitites and query cache
- Optimistic and pessimistic locks

Entity types

- **JPA** entities - Java objects stored in a database using Java Persistence API
- **DTOs** - plain Java objects, not associated to any persistence layer
- **Key-Value** entity - dynamic entity without pre-defined attributes

JPA Entity

```
@JmixEntity
@Table(name = "SAMPLE_CUSTOMER")
@Entity(name = "sample_Customer")
public class Customer {

    @JmixGeneratedValue
    @Id
    @Column(name = "ID", nullable = false)
    private UUID id;

    @InstanceName
    @NotNull
    @Column(name = "NAME", nullable = false)
    private String name;

    @Email
    @Column(name = "EMAIL", unique = true)
    private String email;

// other getters and setters
```

Jmix Annotations

- **`@JmixEntity`** - mandatory annotation that shows that this entity is managed by Jmix

- **`@JmixId`** - entity identifier for DTO

- **`@JmixGeneratedValue`** - shows that the value should be generated upon entity creation

```
@JmixEntity
@Table(name = "SAMPLE_CUSTOMER")
@Entity(name = "sample_Customer")
public class Customer {
  // ...
}

@Store(name = "inmem")
@JmixEntity(name = "sample_Metric")
public class Metric {

    @JmixProperty(mandatory = true)
    @JmixId
    @JmixGeneratedValue
    private UUID id;
    //...
}
```

Jmix Annotations

- **`@JmixProperty`** - annotation indicates that an object field or method is an Jmix entity attribute
- **`@InstanceName`** - user-readable text to represent entity in the UI.
- **`@DependsOnProperties`** - specifies properties to generate the value.

```
@InstanceName
@DependsOnProperties({"latitude", "longitude"})
public String getDisplayName(Messages messages) {
    return messages.formatMessage(
          getClass(), "GeoPointEntity.instanceName", this.latitude,
this.longitude);
}
```

Jmix Annotations

- **@PropertyDatatype** - specifies property Jmix datatype
- **@Store** - links the entity with an addional DataStore
- **@Composition** - shows that the link between entities is a composition

```
@PropertyDatatype("year
")
@Column(name = "YEAR_")
private Integer
productionYear;
```

```
@Composition
@OneToMany(mappedBy =
"order")
private List<OrderLine>
lines;
```

Jmix Annotations

- **`@PostConstruct`** - marks a method that should be executed after entity initialization

```
@PostConstruct
void init(TimeSource timeSource) {
      setDate(timeSource.now().toLocalDate());
}
```

- **`@SystemLevel`** - shows that this is an internal entity/attribute and should not be displayed in the UI by default
- **`@DbView`** - shows that this entity is mapped to a database view

Jmix Annotations

`@DdlGeneration` – defines whether we should generate DDL for this entity

Generation mode is defined in the enum **DbScriptGenerationMode**:

- `CREATE_AND_DROP (default)` – scripts to generate DB and drop non-existent objects
- `CREATE_ONLY` – recreate DB, update scripts without delete column statements
- `DISABLED` – do not generate scripts

Entity Attributes

Based on fields:

```
@Column(name = "FIRST_NAME")
protected String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String
firstName) {
    this.firstName = firstName;
}
```

Based on methods:

```
@JmixProperty
@DependsOnProperties({"firstName",
"lastName"})
public String getFullName() {
    return this.firstName + " " +
this.lastName;
}
```

Entity Attributes

By default, **@JmixEntity(annotatedPropertiesOnly = false)** attributes are:

- JPA entity: all fields apart from **@javax.persistence.Transient;**
- DTO: all fields
- JPA and DTO: all properties and methods with **@JmixProperty**

Otherwise (**@JmixEntity(annotatedPropertiesOnly = true)**)

- JPA and DTO: fields and methods annotated with **@JmixProperty only**

Supported datatypes

- `java.lang.String`
- `java.lang.Character`
- `java.lang.Boolean`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Double`
- `java.math.BigDecimal`
- `java.util.Date`
- `java.time.LocalDate`
- `java.time.LocalTime`
- `java.time.LocalDateTime`
- `java.time.OffsetTime`
- `java.time.OffsetDateTime`
- `java.sql.Date`
- `java.sql.Time`
- `java.util.UUID`
- `java.net.URI`
- `byte[]`
- `Enumeration`
- `Entity or entities collection`

Attribute types

- **Enum** - adds enumeration that implements EnumClass
- **Association / Composition** - adds entity or entities collection depending on association type
- **Embedded** – adds a reference to an embedable entity
- **Datatype** - adds a simple data type

Attribute types. Enum

Enumeration in Jmix - is Java enum type which implements EnumClass and has an id field of **Integer** or **String** type

- We can rename and reorder enum constants safely
- If no enum value found, entity will be loaded with null value

Attribute types. Enum

```
public enum CustomerGrade implements
EnumClass<String> {

        BRONZE("B"),  GOLD("G"), PLATINUM("P");

        private String id;

        CustomerGrade(String value) {
                this.id = value;
        }

        public String getId() {
                return id;
        }

        @Nullable
        public static CustomerGrade
fromId(String id) {
                // ...
        }
}
```

```
@Column(name = "GRADE")
private String grade;

public CustomerGrade getGrade() {
        return grade == null ? null :
CustomerGrade.fromId(grade);
}

public void setGrade(CustomerGrade grade) {
        this.grade = grade == null ? null :
grade.getId();
}
```

Attribute types. Association

- Jmix supports all Association types
  - 1:1
  - 1:M
  - M:1
  - M:M

- M : M causes join table generation

Attribute types. Composition

Composition supports **only**

- 1 : M
- 1 : 1

Also, Jmix supports multi-level composition

```
@Composition
@OneToMany(mappedBy = "film")
private List<Country> countries;
```

Attribute types. One-to-One

Motivation:

- Separate data by its update frequency
- Caching is needed
- Security settings are required

Attribute types. One-to-One

In Jmix UI we can edit One-to-One association by:

- EntityPicker

```
<entityPicker id="capitalField" property="capital">
```

- Other fields with dot notation to specify properties

```
<textField id="capitalNameField" property="capital.name"/>
```

Attribute types. Embedded

```
@JmixEntity
@Embeddable
public class Address {

@Column(name = "STREET")
private String street;

@Column(name = "NUMBER")
private Integer apartmentNumber;
}
```

```
@EmbeddedParameters(nullAllowed = false)
@Embedded
@AttributeOverrides({
        @AttributeOverride(name = "street",
                                            column = @Column(name =
"ADDRESS_STREET")),
        @AttributeOverride(name = "apartmentNumber",
                                            column = @Column(name =
"ADDRESS_NUMBER"))
})
private Address address;
```

Custom datatypes

`Datatype` – interface to convert attribute values from and to strings

- BigDecimalDatatype
- BooleanDatatype
- LongDatatype
- DateDatatype, etc.

Custom datatypes

```
# Date/time formats
dateFormat = dd/MM/yyyy
dateTimeFormat = dd/MM/yyyy HH:mm
offsetDateTimeFormat = dd/MM/yyyy HH:mm Z
timeFormat = HH:mm
offsetTimeFormat = HH:mm Z

# Number formats
integerFormat = #,##0
doubleFormat = #,##0.###
decimalFormat = #,##0.##
```

```
# Number separators
numberDecimalSeparator = .
numberGroupingSeparator = ,

# Booleans
trueString = True
falseString = False
```

Custom datatypes

```
@DatatypeDef(
                id = "year",
                javaClass = Integer.class
)
@Ddl("int")
public class YearDatatype implements Datatype<Integer> {

       @Override
       public String format(@Nullable Object value) { /* ... */ }

       @Override
       public String format(@Nullable Object value, Locale locale) { /* ... */ }

       @Nullable
       @Override
       public Integer parse(@Nullable String value) throws ParseException { /*
... */ }

       @Nullable
       @Override
       public Integer parse(@Nullable String value, Locale locale) throws
ParseException { /* ... */ }
}
```

Traits

HasUUID - provides client-generated GUID and an entity ID

```
// ...
public class OrderInfo {

      @JmixGeneratedValue
      @Column(name = "ID",
nullable = false)
      @Id
      private UUID id;

      // …
}
```

```
// ...
public class Card {

      @Column(name = "ID", nullable =
false)
      @Id
      private Long id;

      @JmixGeneratedValue
      @Column(name = "UUID")
      private UUID uuid;

      // …
}
```

Traits

Versioned - provides optimistic locking using JPA

```
@Column(name = "VERSION", nullable = false)
@Version
private Integer version;
```

Traits

Audit of creation and modification

```
@CreatedBy
@Column(name = "CREATED_BY")
private String createdBy;

@CreatedDate
@Temporal(TemporalType.DATE)
@Column(name = "CREATED_DATE")
private Date createdDate;
```

```
@LastModifiedBy
@Column(name = "LAST_MODIFIED_BY")
private String lastModifiedBy;

@LastModifiedDate
@Temporal(TemporalType.DATE)
@Column(name = "LAST_MODIFIED_DATE")
private Date lastModifiedDate;
```

Traits

Soft Delete - provides soft deletion of entity instances

```
@DeletedBy
@Column(name = "DELETED_BY")
private String deletedBy;

@DeletedDate
@Temporal(TemporalType.DATE)
@Column(name = "DELETED_DATE")
private Date deletedDate;
```

Soft Deletion

- Filtered in
  - JPQL queries
  - Associations
  - Collection attributes(To many)
- Not filtered in
  - Reverse attributes (M:1)

Soft Deletion

- **`@OnDelete`** - what to do if current entity is soft deleted
- **`@OnDeleteInverse`** - what to do if a referenced entity is soft deleted

Deletion policies:

- **`DeletePolicy.DENY`**
- **`DeletePolicy.CASCADE`**
- **`DeletePolicy.UNLINK`**

Soft Deletion

To disable **Soft Delete** use **hint**:

- **`PersistenceHints.SOFT_DELETION`** = false.

```
public Customer loadDeletedCustomer(Id<Customer> customerId) {
       return
dataManager.load(customerId).hint(PersistenceHints.SOFT_DELETION,
false).one();
}


public void hardDeleteCustomer(Customer customer) {
       dataManager.save(
              new SaveContext()
                     .removing(customer)
                     .setHint(PersistenceHints.SOFT_DELETION, false)
       );
}
```

DTO Entity

```java
@JmixEntity
public class OperationResult {

    private String result;

    private Integer errorCode;

    private String errorMessage;

    // other getters and setters

}
```

DTO Entity

Attribute annotations

```
@JmixEntity(name =
"sample_OperationResult",  annotatedPropertiesOnly = true)
public class OperationResult {

       @JmixProperty(mandatory = "true")
       private String result;

       @JmixProperty
        private String errorMessage;

        private Integer errorCode;

        // getters and setters
}
```

Key-Value Entity

- **Loading KeyValueEntity**

```
 List<KeyValueEntity> entities = dataManager.loadValues(
          "select e.customer, sum(e.amount) from sample_Order e group by
e.customer")
        .properties("customer", "total")
        .list();
```

- **Reading KeyValueEntity**

```
for (KeyValueEntity entity : entities) {
    Customer customer = entity.getValue("customer");
    BigDecimal totalAmount = entity.getValue("total");
    // ...
}
```

Bean validation

Jmix users JSR-380 annotations and **Hibernate Validator** library

Bean Validation pros:

- Validation logic is in the data model
- Custom annotations allowed
- You can put constraints not only on fields and classes but also on methods and method parameters.

Bean validation. Annotations

Can be applied to:

- Entities
- POJOs
- Fields and getters
- Service methods

Numeric:

- `Min, Max, Positive, PositiveOrZero, Negative, NegativeOrZero`

Dates:

- `Past, PastOrPresent, Future, FutureOrPresent`

Collections, strings:

- `Size, NotEmpty`

Other:

- `NotNull, NotBlank, Email, Pattern`

Bean validation. Groups

- RestApiChecks
- UiComponentChecks
- UiCrossFieldChecks
- javax.validation.groups.Default

Metadata

Main API entry point is `Metadata` bean. Gives access to information about entities. Main classes are:

- MetaClass
- MetaProperty
- MetaPropertyPath

Entity states

- **New** – newly created, not saved
- **Managed** – loaded from the DB or just saved
- **Detached** - loaded from the DB and detached from context

EntityStates Bean

Provides an information about an entity state:

- `isNew()`
- `isManaged()`
- `isDetached()`
- `isLoaded()`

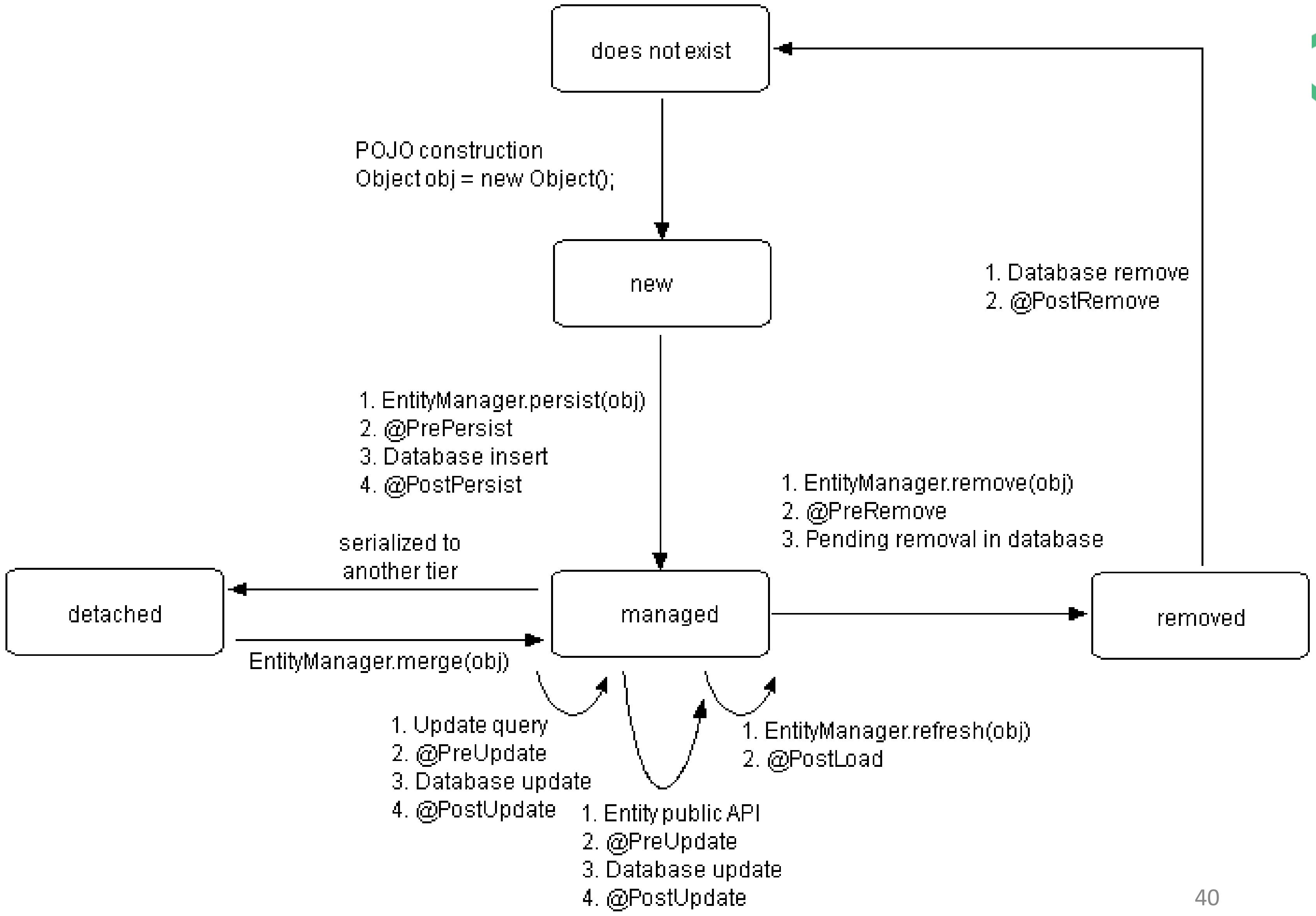
JPA callbacks

Pre

- @PrePersist
- @PreUpdate
- @PreRemove

Post

- @PostPersist
- @PostUpdate
- @PostRemove
- @PostLoad



Jmix entities lifecycle events

When using DataManager, it publishes the following Spring application events:

- EntityChangedEvent
- EntitySavingEvent
- EntityLoadingEvent

Jmix entities lifecycle events

EntityChangedEvent - contains changes information: operation type, changed entity ID, etc.

```
@Component
public class CustomerEventListener {

@EventListener
public void  onCustomerChangedBeforeCommit(EntityChangedEvent<Customer>
event) {
                // ...
       }
}
```

Jmix entities lifecycle events

Handling **EntityChangedEvent** after commit

```
@Component
public class CustomerEventListener {

        @TransactionalEventListener
        void
onCustomerChangedAfterCommit(EntityChangedEvent<Customer> event) {
                // ...
        }
}
```

To load or save data, a transaction required. In **DataManager**, we can use **setJoinTransaction(false)** on it's **SaveContext()**;

Jmix entities lifecycle events

EntitySavingEvent и EntityLoadingEvent

```
@Component
public class EntityEventListener {

        @EventListener
        void onOrderSaving(EntitySavingEvent<Order> event) {
                if (event.isNewEntity()) {
                        Order order = event.getEntity();
                        order.setNumber(generateOrderNumber());
                }
        }

        @EventListener
        void onCustomerLoading(EntityLoadingEvent<Customer> event) {
                // ...
        }
}
```

Data Store

- **DataStore** is an abstraction over any data store:
  - RDBMS
  - NoSQL
  - File
  - etc.
- Contains a minimal set of methods:
  - load()
  - loadList()
  - loadValues()
  - getCount()
  - save()

Data Store

Connection parameters specified in properties

```
main.datasource.url =
jdbc:hsqldb:file:.jmix/hsqldb/sample
main.datasource.username = sa
main.datasource.password =
```

```
@Bean
@Primary
@ConfigurationProperties("main.datasource")
DataSourceProperties dataSourceProperties() {
        return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("main.datasource.hikari")
DataSource dataSource(DataSourceProperties dataSourceProperties) {
        return dataSourceProperties.initializeDataSourceBuilder().build();
}
```

Additional Stores

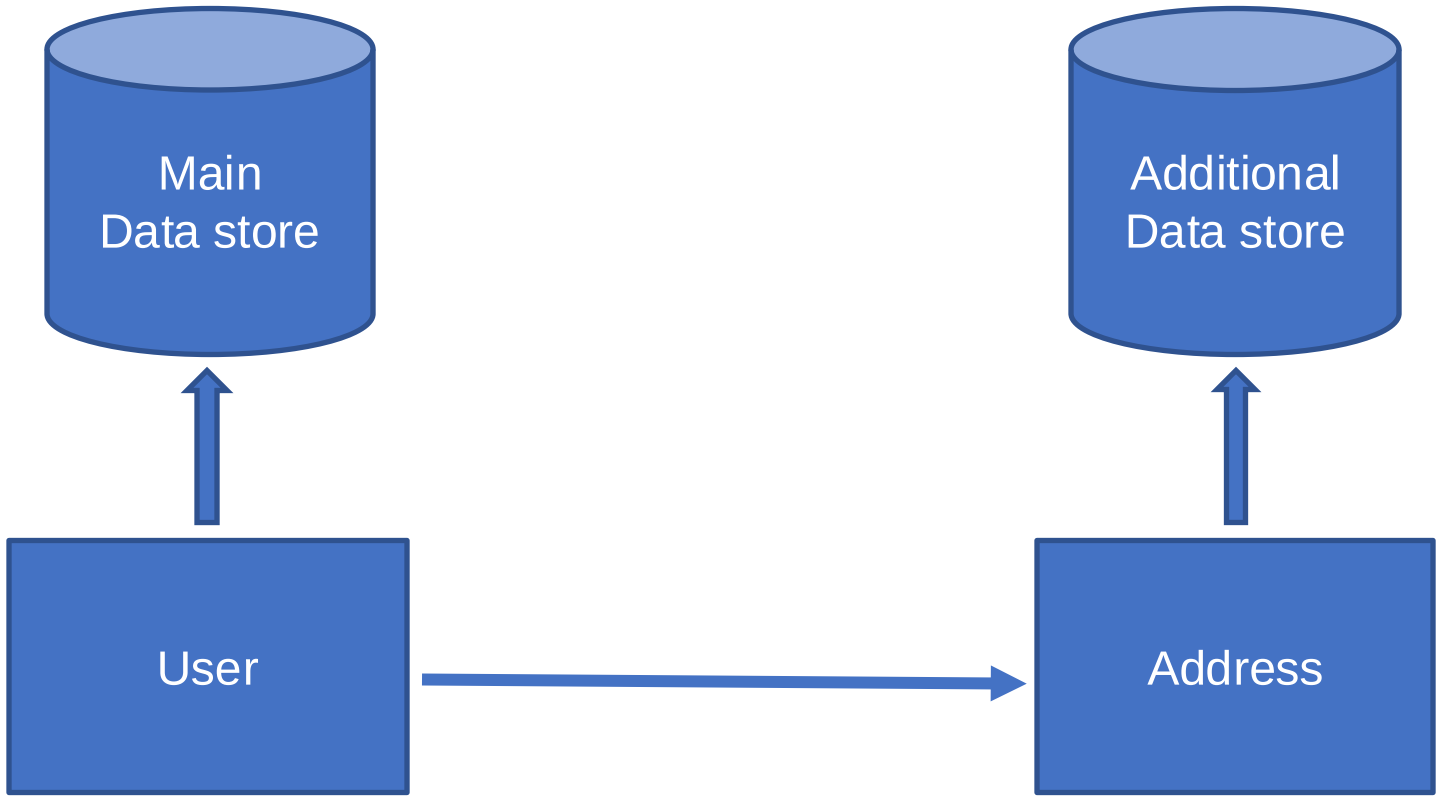
To work with several DBs, we need additional datastores

```
jmix.core.additional-stores = locations,inmem
locations.datasource.url =
jdbc:hsqldb:file:.jmix/hsqldb/locations
locations.datasource.username = sa
locations.datasource.password =
```

Beans to work with additional data stores:

- `DataSourceProperties`
- `DataSource`
- `LocalContainerEntityManagerFactoryBean`
- `JpaTransactionManager`
- `SpringLiquibase`

Entities in different datastores



Entities in different datastores

```
@SystemLevel
@Column(name = "ADDRESS_ID")
private UUID addressId;

@Transient
@JmixProperty
@DependsOnProperties("addressId")
private Address address;

// getter / setters
```

DB versioning. Liquibase

Liquibase is a library with tools and plugins:

- Maven
- Gradle
- CLI

In Jmix we use Jmix Studio to work with Liquibase

DB versioning. Liquibase

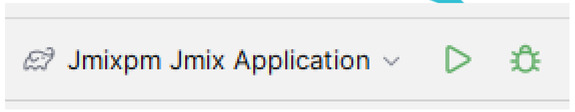
Main unit - changeSet in changelog files

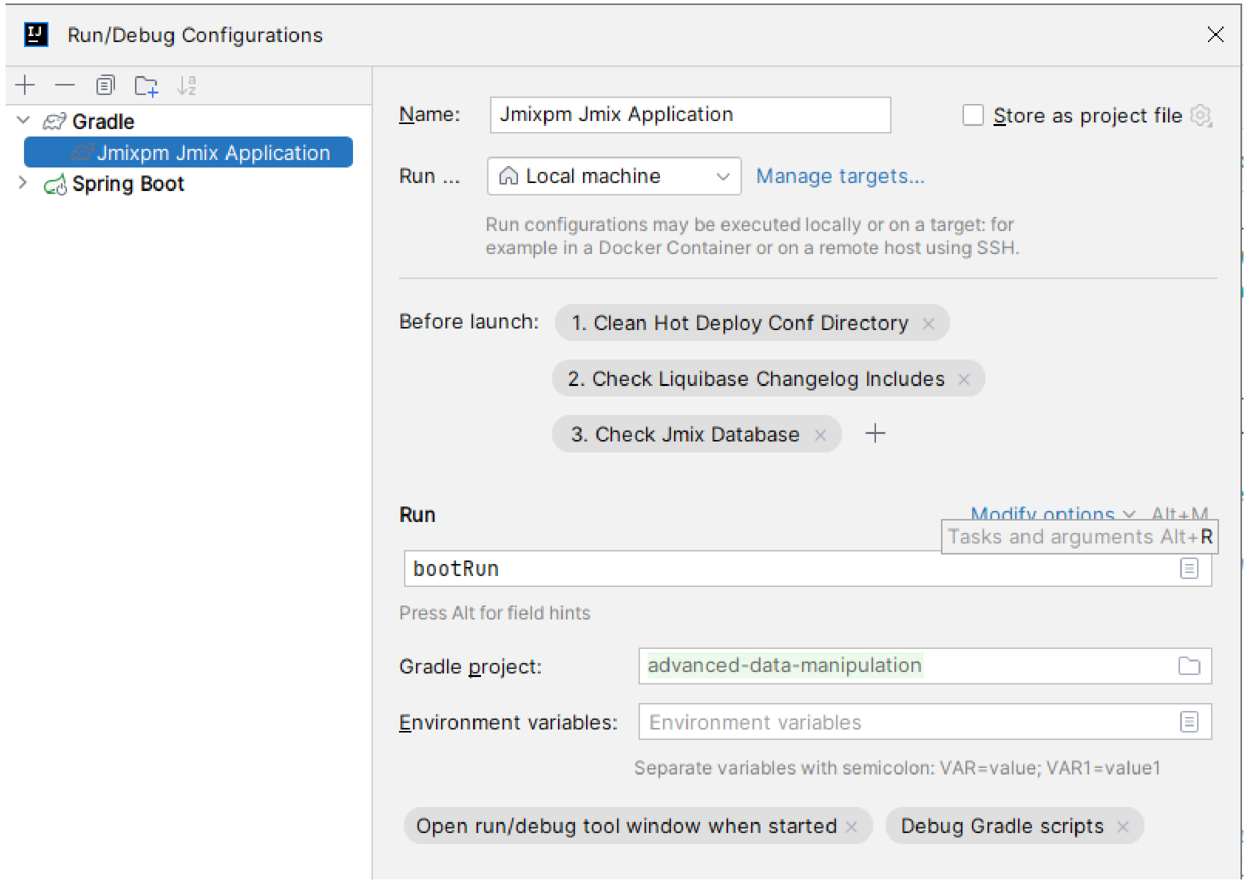
```
<changeSet id="1" author="demo">
      <createTable tableName="USER_">
           <column name="ID" type="${uuid.type}">
                 <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="USERNAME" type="varchar(255)">
                 <constraints nullable="false"/>
            </column>
            <column name="PASSWORD" type="varchar(255)"/>
      </createTable>
</changeSet>
```

```
<changeSet id="3" author="demo">
      <insert tableName="USER_"
                dbms="postgresql, mssql, hsqldb">
           <column name="ID"
                        value="60885987-1b61-4247-94c7-dff348347f93"/>
           <column name="USERNAME" value="admin"/>
           <column name="PASSWORD" value="{noop}admin"/>
      </insert>
</changeSet>
```

DB versioning. Liquibase

Jmix Studio adds its own operations to Run/Debug configuration before launch.




Run/Debug Configurations

Gradle
Jmixpm Jmix Application
Spring Boot

Name: Jmixpm Jmix Application
Store as project file
Run ... Local machine
Manage targets...
Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

Before launch:
1. Clean Hot Deploy Conf Directory
2. Check Liquibase Changelog Includes
3. Check Jmix Database

Run
Modify options Alt+M
Tasks and arguments Alt+R
bootRun
Press Alt for field hints

Gradle project: advanced-data-manipulation
Environment variables: Environment variables
Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started
Debug Gradle scripts

DB versioning. Liquibase

- Root changelog file:

```
src/main/resources/<base_package>/liquibase
```

- Required property to specify the path to the root changelog file:

```
<data-store-name>.liquibase.change-log=com/company/myapp/liquibase/changelog.xml
```

- Example of root changelog file content:

```
<include file="/io/jmix/data/liquibase/changelog.xml"/>
<include file="/io/jmix/flowuidata/liquibase/changelog.xml"/>
<include file="/io/jmix/securitydata/liquibase/changelog.xml"/>

<includeAll path="/com/company/myapp/liquibase/changelog"/>
```

DB versioning. Liquibase

Changelog structure

```
├── liquibase/
│   ├── changelog/
│   │   ├── 010-init-user.xml
│   │   └── 2020/
│   │       ├── 11/
│   │       │   ├── 12-010-fe2b82e6.xml
│   │       │   └── 27-010-fe2b82e6.xml
│   │       └── 12/
│   │           └── 17-010-fe2b82e6.xml
│   ├── changelog.xml
│   ├── locations-changelog/
│   │   └── 2020/
│   │       └── 11/
│   │           ├── 25-010-fe2b82e6.xml
│   │           └── 28-010-fe2b82e6.xml
│   └── locations-changelog.xml
```

DB versioning. Liquibase

Liquibase internal tables:

- `databasechangelog`
- `databasechangeloglock`

DB versioning. Liquibase

We use **main** prefix instead of **spring** for the main data store:

- main.liquibase.contexts
- main.liquibase.enabled

Property **change-log**, should not be changed!

We can use prefixes for tables to ignore them:

- main.datasource.studio.liquibase.exclude-prefixes = abc_,foo,bar

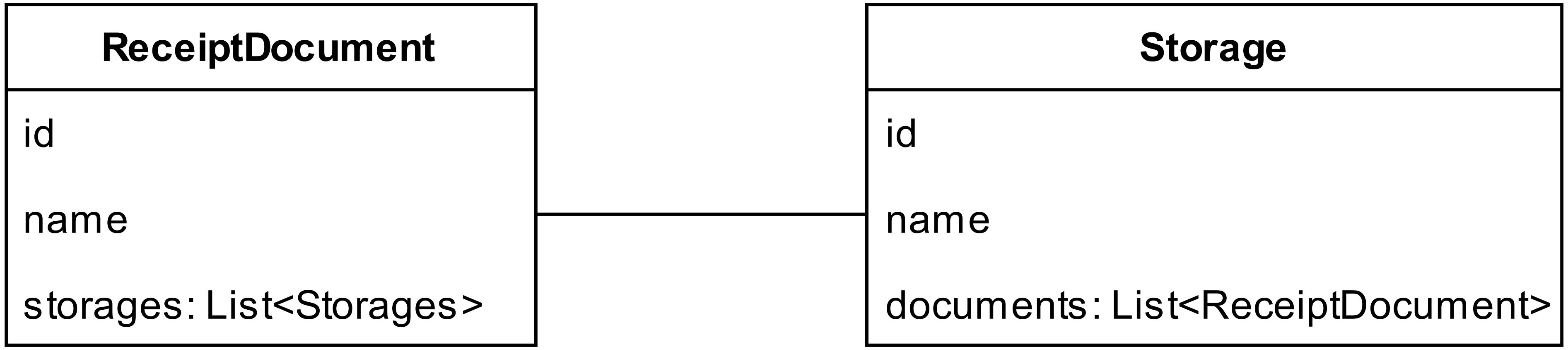
Data model extension

Jmix allows you to extend the functionality of framework add-ons with an application or add-on that is lower in the hierarchy.

To extend data model entities, Jmix provides its own extension mechanism:

- **@ReplaceEntity**

Data model extension



DataLoadCoordinator facet

Facet is designed for triggering data loaders and for declarative linking of data loaders to data containers, visual components, and screen events.

```
<dataLoadCoordinator>
    <refresh loader="citiesDl">
        <onViewEvent type="Init"/>
        <onComponentValueChanged component="nameField"
                                 likeClause="CASE_INSENSITIVE"
                                 param="name"/>
    </refresh>
</dataLoadCoordinator>


<dataLoadCoordinator auto="true"/>
```

Data Security

Jmix Data Security consists of:

- User roles
- Access management
  - Entities operations
  - Entities attributes
  - Entities instances

Data Security and DataManager

DataManager by default **follows** the configured entity permissions for users.

- **load(), loadList(), loadValue(), loadValues(), getCount()** – read security policies
- **save(), remove()** – create, update and delete policies

- **unconstrained()** – returns DataManager that do not follow security policies

EntityManager

```
@PersistenceContext
private EntityManager entityManager;

@Transactional
public Customer createCustomer() {
       Customer customer =
metadata.create(Customer.class);
       customer.setName("Bob");
       entityManager.persist(customer);
       return customer;
}
```

```
@PersistenceContext(unitName = "db1")
private EntityManager entityManagerForDb1;

@Transactional("db1TransactionManager")
public Foo createFoo() {
       Foo foo = metadata.create(Foo.class);
       foo.setName("foo1");
       entityManagerForDb1.persist(foo);
       return foo;
}
```

EntityManager. FetchPlan

```
@PersistenceContext
private EntityManager entityManager;
@Autowired
private FetchPlans fetchPlans;

@Transactional
public Order findOrder(UUID orderId) {
       FetchPlan fetchPlan = fetchPlans.builder(Order.class)
                    .add("customer")
                    .build();
       Map<String, Object> properties =
PersistenceHints.builder()
                     .withFetchPlan(fetchPlan)
                     .build();
       return entityManager.find(Order.class, orderId,
properties);
}
```

EntityManager. FetchPlan

```
@Transactional
public Order loadGraphOfPartialEntities(UUID orderId) {
       FetchPlan fetchPlan = fetchPlans.builder(Order.class)
                     .addAll("number", "date", "customer.name")
                     .partial()
                     .build();

       Map<String, Object> properties = PersistenceHints.builder()
                     .withFetchPlan(fetchPlan)
                     .build();

       return entityManager.find(Order.class, orderId, properties);
}
```

EntityManager. Soft Delete

To disable soft delete, set PersistenceHints.SOFT_DELETION to false.

```
@Transactional
public void hardDelete(Product product) {
       entityManager.setProperty(PersistenceHints.SOFT_DELETION
, false);
       entityManager.remove(product);
}
```

EntityManager. Limitations

- Does not generate **EntitySavingEvent** and **EntityLoadingEvent**
- Lazy attributes fetch do not work
- Do not support references from **additional** stores
- Security policies are not applied

Transactions management. Declarative approach

- @org.springframework.transaction.annotation.Transactional

```
@Transactional
public void makeDiscountsForAll() {
       List<Order> orders = dataManager.load(Order.class)
                     .query("select o from Order o where o.customer is not null")
                     .list();
       for (Order order : orders) {
              BigDecimal newTotal = orderService.calculateDiscount(order);
              order.setAmount(newTotal);
              dataManager.save(order);
              Customer customer =
customerService.updateCustomerGrade(order.getCustomer());
              dataManager.save(customer);
       }
}
```

Transactions management. Declarative approach

- @org.springframework.transaction.annotation.Transactional

```
@Transactional(transactionManager="ordersTransactionManager")
public void makeDiscountsForAll() {
       List<Order> orders = dataManager.load(Order.class)
                    .query("select o from Order o where o.customer is not null")
                    .list();
       for (Order order : orders) {
              BigDecimal newTotal = orderService.calculateDiscount(order);
              order.setAmount(newTotal);
              dataManager.save(order);
              Customer customer =
customerService.updateCustomerGrade(order.getCustomer());
              dataManager.save(customer);
       }
}
```

Transactions management. Programmatic approach

- org.springframework.transaction.support.TransactionTemplate

```
@Bean
@Primary
TransactionTemplate transactionTemplate(PlatformTransactionManager
transactionManager) {
        return new TransactionTemplate(transactionManager);
}
```

```
@Bean
TransactionTemplate
db1TransactionTemplate(@Qualifier("db1TransactionManager")

  PlatformTransactionManager transactionManager) {
        return new TransactionTemplate(transactionManager);
}
```

Transactions management. Programmatic approach

```java
@Autowired
private TransactionTemplate transactionTemplate;

public UUID createOrderAndReturnId() {
    return transactionTemplate.execute(status -> {
        Customer customer = dataManager.create(Customer.class);
        customer.setName("Alice");
        customer = dataManager.save(customer);

        Order order = dataManager.create(Order.class);
        order.setCustomer(customer);

        order = dataManager.save(order);
        return order.getId();
    });
}
```

Transactions management. Programmatic approach

```java
@Autowired
private TransactionTemplate transactionTemplate;

public void createOrder() {
    transactionTemplate.executeWithoutResult(status -> {
        Customer customer = dataManager.create(Customer.class);
        customer.setName("Alice");
        customer = dataManager.save(customer);

        Order order = dataManager.create(Order.class);
        order.setCustomer(customer);

        dataManager.save(order);
    });
}
```

Fetching Data

There are two strategies:

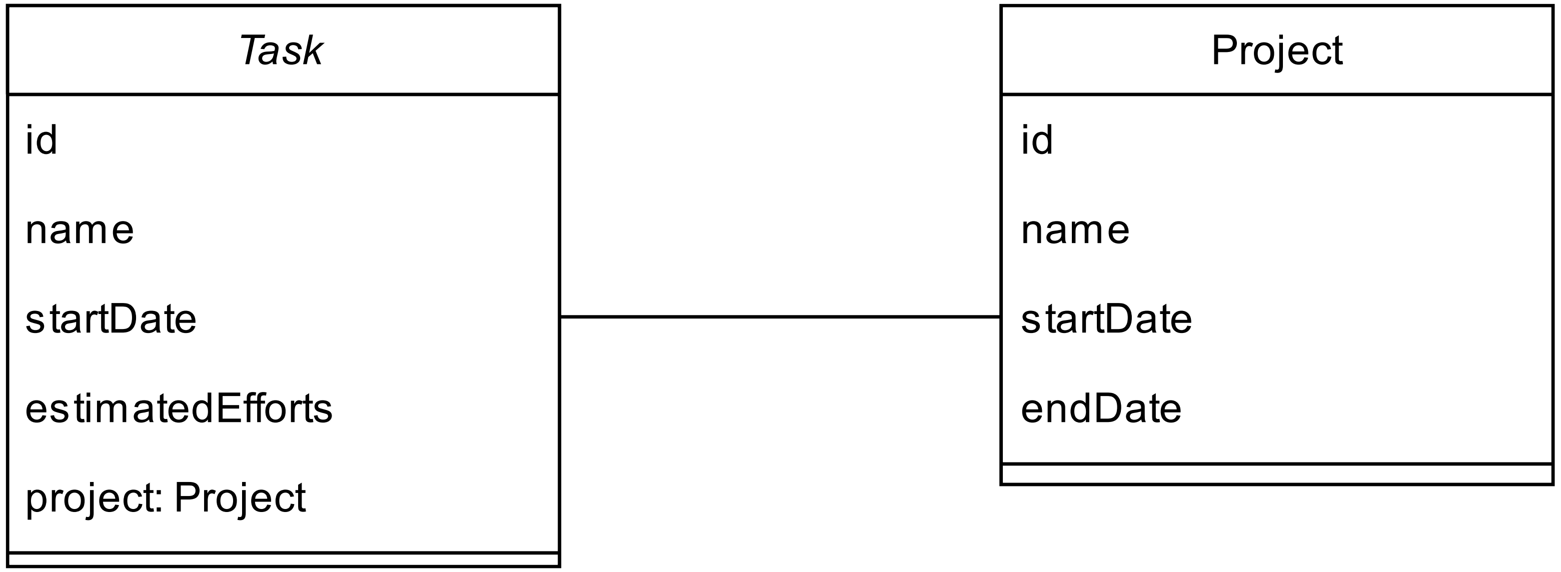
- **EAGER** - the related entity is loaded from the database together with the root entity
- **LAZY** - the related entity is transparently loaded from the database when the reference property is accessed

Fetching Data. Lazy Loading

```
String getCustomerName(Id<Order> orderId) {
       Order order = dataManager.load(orderId).one();
       return order.getCustomer().getName();
}

List<String> getProductNames(Id<Order> orderId) {
       Order order = dataManager.load(orderId).one();
       return order.getLines().stream()
                    .map(orderLine ->
orderLine.getProduct().getName())
                    .collect(Collectors.toList());
}
```

Fetching Data. FetchPlan



```
<collection id="projectTasksDc"
            class="com.company.jmixpm.entity.Task">
    <fetchPlan extends="_base">
        <property name="project" fetchPlan="_base"/>
    </fetchPlan>
</collection>
```

Fetching Data. Partial Load

For the entity below, the following data will be fetched:

- All attributes required for InstanceName
- Assignee attribute.

```
<instance id="projectTaskDc"
              class="com.company.jmixpm.entity.ProjectTask">
      <fetchPlan extends="_instance_name">
           <property name="assignee" fetchPlan="_base"/>
      </fetchPlan>
      <loader/>
</instance>
```

Fetching Data. Built-in FetchPlans

- **`_local`** - includes all local attributes
- **`_instance_name`** - includes attributes required for **InstanceName**.
- **`_base`** - combination of **`_local`** and **`_instance_name`**`. Used by default`

Fetching Data. Creating FetchPlan

Creating FetchPlan programmatically

```
@Autowired
private FetchPlans fetchPlans;

private List<Order> loadOrders() {
     FetchPlan fetchPlan = fetchPlans.builder(Order.class)
                    .addFetchPlan(FetchPlan.BASE)
                    .add("customer")
                    .build();
     return
dataManager.load(Order.class).all().fetchPlan(fetchPlan).lis
t();
}
```

Fetching Data. Creating FetchPlan

Creating FetchPlan for using with **FetchPlanRepository:**

1. Create a fetch-plans.xml file
2. Define path to the file in application properties:

```
jmix.core.fetch-plans-config=dataaccess/ex1/fetch-plans.xml
```

```xml
<fetchPlans xmlns="http://jmix.io/schema/core/fetch-plans">
    <fetchPlan class="dataaccess.ex1.entity.Order"
               name="full"
               extends="_base">
        <property name="customer" fetchPlan="_instance_name"/>
        <property name="lines">
        <property name="product" fetchPlan="_instance_name"/>
        <property name="quantity"/>
        </property>
    </fetchPlan>
</fetchPlans>
```

Jmix Data Repositories

Repository creation:

```
public interface CustomerRepository extends JmixDataRepository<Customer, UUID> { }
```

In application support:

```
@SpringBootApplication
@EnableJmixDataRepositories
public class DemoApplication implements AppShellConfigurator {}
```

Jmix Data Repositories | Features

- `Load` methods can accept a fethc plan
- `create()` method instantiates a new entity.
- `getById()` method with the non-optional result loads an entity by id and throws the exception if the entity is not found.
- `getDataManager()` method returns DataManager.
- `save()` method persists the provided entity and returns saved instance, loaded with the specified fetch plan.

Jmix Data Repositories | Examples

From the method name:

```
List<Customer> findByEmailContainingIgnoreCase(String part);
```

With query and parameter:

```
@Query("select c from sample_Customer c where c.email like :email")
List<Customer> findCustomersByEmail(@Param("email") String part);
```

Pageable:

```
Page<Customer> findByEmailContainingIgnoreCase(String part, Pageable pageable);
```

Jmix Data Repositories | Examples

With fetch plan passed in method:

```
List<Customer> findByEmailContainingIgnoreCase(String part, FetchPlan plan);
```

With shared fetch plan:

```
@FetchPlan("customer-minimal")
List<Customer> findByEmail(String email);
```

Jmix Data Repositories | Data access

- `@ApplyConstraints` – default value is true – data access constraints are checked by default

- `@ApplyConstraints(false)` can be added to the method or entire repository. The **UnconstrainedDataManager** is going to be used for annotated methods or for all methods in annotated repository.

NOTE: Of course, if the repository is annotated as `@ApplyConstraints(false)`, access in methods annotated as `@ApplyConstraints` is going to be checked

JPQL extensions

- Session and user attributes (session_, current_user_)
- Case-insensitive search
- Functions
- Macros
- Time constants

JPQL extensions. Macros

- @between `select c from Customer where @between(c.createTs, now, now+1, day)`
- @today `select d from Doc where @today(d.createTs)`
- @dateEquals `select d from Doc where @dateEquals(d.createTs, :param)`
- @dateBefore `select d from sales_Doc where @dateBefore(d.createTs, now+1))`
- @dateAfter `select d from Doc where @dateAfter(d.createTs, now-1)`
- @enum `where d.type = @enum(com.company.demo.entity.DocType.INVOICE)`

JPQL extensions. Time constants

- `FIRST_DAY_OF_CURRENT_YEAR`
- `FIRST_DAY_OF_CURRENT_MONTH,`
- `FIRST_DAY_OF_CURRENT_WEEK,`
- `START_OF_CURRENT_DAY,`
- `START_OF_YESTERDAY,`
- `END_OF_CURRENT_HOUR,`
- `END_OF_CURRENT_MINUTE`
- `etc.`

```
select e from sample_Doc e where e.localDate >= FIRST_DAY_OF_CURRENT_YEAR
```

Entities cache

- Jmix uses cache for **fetching by ID** only
- Data fetch by other attributes still uses database
- Accociatiated entities use caching too

```
eclipselink.cache.shared.sales_Order=true
eclipselink.cache.size.sales_Order=500
```

Query cache

Query cache stores entities IDs returned by JPQL queries

Query cache can be enabled by:

- Hints in Query and EntityManager: `setHint(PersistenceHints.CACHEABLE, true)`
- setCacheable() method in LoadContext.Query interface when working with DataManager (also in ByQuery(), ByCondition()).
- setCacheable() method in CollectionLoader or in cacheable XML attribute in UI

Optimistic locking

Used in low-concurrent systems. Uses versioning approach

```
@Column(name = "VERSION", nullable =
false)
@Version
private Integer version;
```

Pessimistic locking

If optimistic lock causes **too many rollbacks**

```
@PessimisticLock
@JmixEntity
@Table(name = "ORDER_")
@Entity(name = "demo_Order")
public class Order {
```

Pessimistic locking

Pessimistic Locking add-on is **required** to add the locking functionality

Quartz add-on **required** to manage timeout automatically

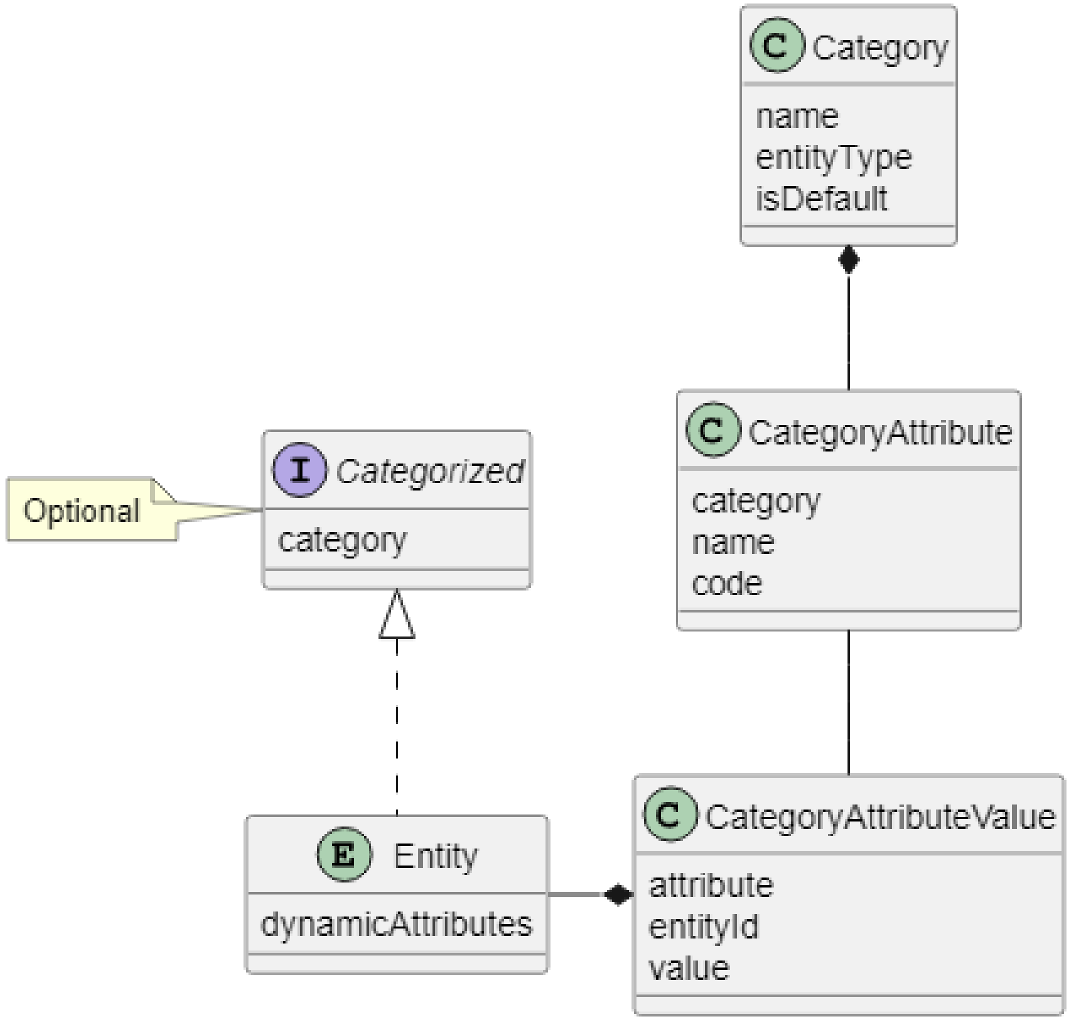
- `jmix.pslock.expiration-cron=0/5 * * * * ?`
- `jmix.pslock.use-default-quartz-configuration=false`

Optional theme: Dynamic attributes

Allows you extending data model without schema change

- Stored in the main data store
- Loaded automatically by Jmix

Dynamic attributes



Dynamic attributes

Supported by **DataManager**

- **setHint(DynAttrQueryHints.LOAD_DYN_ATTR, true)** of LoadContext()
- **hint(DynAttrQueryHints.LOAD_DYN_ATTR, true)** of the fluent API.

EntityValues allows us to get dynamic attributes:

- **EntityValues#getValue(task, "+task-notes-description")**;

Q&A