Uaithne generator Java

Designed by: **Juan Luis Paz Rojas**

Foreword

Improving the productivity of the development team has been one of the great motivations behind the design of the architecture of Uaithne, and it alone represents a great improvement over the classical n-tier architecture. After implementing the new architecture there was still room to improve productivity, automatically generating code that is responsible for handling the verbosity required by Java, and above all, that generates the access code to the database with its respective SQL queries that handles most operations.

Uaithne is accompanied, optionally, with a code generator that allows to generate (and regenerate) the operations and entities from a small definition, as well as the logic of access to database required by them, using MyBatis as database access framework. In this way, only those activities that really require human work are left to the programmer due to the decisions that must be made by him.

The Uaithne architecture and the code generator working together have proved in a multitude of projects of different sizes to be a very useful tool for the programming of the backend of applications, and during this time, achieving a high productivity of the development team.

This document is the reference documentation of Uaithne code generator. It is assumed that previously the reader already knows the architecture of Uaithne; if not, it is recommended to first read the document that explains the architecture of Uaithne for Java.

juanluispaz@gmail.com
https://github.com/juanluispaz/uaithne-generator-java
Version: 0.6.0 - March 30, 2018

Table of contents

Uaithne generator Java

Designed by: **Juan Luis Paz Rojas**

Code generator

Uaithne includes a code generator that allows reducing the amount of code needed to build the operations and their executors, and it is also possible to generate the implementation of the executors to access the database with their respective SQL queries.

The generator admits that the name of the modules, entities and operations begin with the character _ that is ignored when generating the code; in such a way that the name of the generated elements do not include the symbol _ (only when it is the first character of the name). The use of the symbol _ is useful to distinguish between the input of the generator and the output of this, so it does not lend itself to confusion when using the autocomplete tool of the IDE.

Modules

To facilitate the maintenance of the code, Uaithne groups the operations and optionally the entities into modules. By having this logical grouping of operations, all the highly related pieces are kept together. For example, you can create a module per table of the database containing all the operations that operate on that table and the entities that represent it.

The modules are created by adding the `@OperationModule` annotation to a class, and this class acts as a container for all the operations and entities that make up the module.

```
@OperationModule
public class _events {

}
```

The class annotated with `@OperationModule` acts as a container for the elements that make up the module, so that inside, it is only expected to find constructions that the Uaithne generator is capable of processing; in no case should there be fields for these classes or methods in this class or in any of the classes it contains.

The name of the module is used as the package name of the generated operations, so it is recommended that its name follows the recommendations for naming packages; for this type of packages among others are: first letter in lowercase and name in plural.

Entities

The entities represent the tables in the database and are created by adding the `@Entity` annotation to a class that only contains the fields that comprise it. For convenience, it is allowed to declare entities within modules although only the operations belong to the module.

```
@OperationModule
public class _events {

    @Entity
    class _Event {
        @Id
        Integer id;
        String title;
        Date start;
        Date end;
        @Optional
        String description;
        Integer calendarId;
    }
}
```

For `_Event`, the following class that represents the entity is generated:

```
public class Event implements Serializable {

    private Integer id;
    private String title;
    private Date start;
    private Date end;
    private String description;
    private Integer calendarId;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public Event() { }

    public Event(String title, Date start, Date end, Integer calendarId) {
        this.title = title;
        this.start = start;
        this.end = end;
        this.calendarId = calendarId;
    }

    public Event(Integer id, String title, Date start, Date end, Integer calendarId) {
        this.id = id;
        this.title = title;
        this.start = start;
        this.end = end;
        this.calendarId = calendarId;
    }
}
```

The fields of an entity can be annotated with `@Id` to indicate those fields identify the object. The fields that have the annotation `@Id` are used in the `equals` and `hashCode` methods; if there is no field with the `@Id` annotation, all the fields of the entity are used in the implementation of `equals` and `hashCode`.

Fields that are optional, that is, whose value support `null`, must be marked with the annotation `@Optional`, so that the generator can use this information among other places in the generated constructors (only the obligatory fields are included in the constructors). Fields marked with the annotation `@ExcludeFromConstructor` are not included in the constructor even if they are mandatory.

In the generated entity, the `toString` method is implemented including the name of the entity and the name and value of each of the fields that make up the entity, unless the field is marked with the annotation `@ExcludeFromToString` in which case it is not included in the implementation of `toString` method.

The entities can extend other entities, in such a way that the generator takes into account the fields of each of the entities and when generating them, it respects the hierarchy with which they have been made. It can also extend from another class that is not an entity (in whose case the generator only takes into account that the entity extends that class) or implements interfaces that do not require implementing any method (such as `Serializable`).

If the entity is defined within a module, a series of operations can be generated: (the annotation `@Entity` allows to indicate which should be generated, or it can be activated by default in the configuration of Uaithne)

- **`InsertEvent`**: Inserts an Event object in the data source and returns the id of the inserted record.
- **`UpdateEvent`**: Updates an Event object in the data source and returns the number of updated records.
- **`DeleteEventById`**: Removes an Event record from the data source. This operation receives the record id and returns the number of deleted records.
- **`SelectEventById`**: Retrieves an Event record from the data source. This operation receives the record id and returns the Event object with that id or `null` if it cannot be found.
- **`SaveEvent`**: Inserts or updates an Event object in the data source. If the id of the Event object that receives in the operation is `null`, inserts a new record; if not, updates the record with that identifier. This operation returns the identifier of the inserted record.
- **`MergeEvent`**: Updates the fields of an Event record in the data source whose values are other than `null`. This operation updates only the fields that have value in the record whose id matches the one supplied in the object and returns the number of updated records.
- **`JustInsertEvent`**: Inserts an Event object in the data source and returns the number of records inserted. This operation does not try to recover the id of the inserted record.
- **`JustSaveEvent`**: Inserts or updates an Event object in the data source. If the id of the Event object that receives the operation is `null`, inserts a new record; if not, updates the record with that id. This operation does not try to recover the id of the inserted record, so it returns the number of records affected.

In order to generate these operations, it is necessary that it be contained within a module and that it has a single field marked with the `@Id` annotation defined by it or by any entity from which it

extends directly or indirectly.

In order to generate the `Save` or `JustSave` operations, it is necessary that the data type of the field marked with the `@Id` allows `null`.

Entity views

In many circumstances, when consulting the database, you do not want to extract an entire record, instead you want to extract some of that information, or even that information can be combined with information from another table; in those cases an entity can be created that only contains the desired information. This type of entity will be called an "entity view" and its construction is identical to that of an entity with the difference that it does not represent an entire record stored in a table.

Entity views are created by adding the `@EntityView` annotation to a class that only contains the fields that comprise it. For convenience, it is allowed to declare entity views within modules although only the operations belong to the module.

```
@OperationModule
public class _events {

    [@Entity _Event ...]

    @EntityView
    class _EventView {
        @Id
        Integer id;
        String title;
    }
}
```

The only difference between an entity and an entity view during the code generation process is the entity views don’t generate the entity operations (even if their generation has been marked by default).

Related entities

When creating an entity or entity view, it is possible to specify a related entity to which it is being defined. When a related entity is specified (this being an entity or an entity view regardless of the case), it causes the generator to use the information from the related entity to complement the entity or entity view except the base class that it extends or the interfaces that it implements. In this way the information specified in the related entity does not have to be repeated. In the case of the fields, the information of these is complemented with the information of the field of the same name in the related entity, except if it is mandatory or optional that can only be specified in the entity or entity view in which it is defined.

```
@OperationModule
public class _events {

    // [@Entity _Event ...]
```

```
    @EntityView(related = _Event.class)
    class _EventView {
        Integer id;
        String title;
    }

}
```

When `_Event` is indicated as the related entity, the entity view `_EventView` does not have to mark again the `id` field as an identifier (among many other things that at this point have not yet been specified in the entity), since that information is extracted from the field of the same name in the entity `_Event`.

Additional information for the entity definition

If desired, it is possible to alter some aspects of the generated code to get it treated differently through a series of complementary annotations.

Information that can be added to the entity definition:

- **`abstract`**: If the entity or entity view is marked as abstract, it causes the generated entity to be abstract.
- **`extends`**: If a base class is added (using `extends`) to the entity, it causes the generated entity to extend from that base class. If the base class is another entity, the generator will use the information provided in the base class to generate the code.
- **`implements`**: If interfaces are implemented (using `implements`) in the entity, the generated entity implements the indicated interfaces.
- **`@Doc`**: If this annotation is added to an entity, it causes the text indicated in it to be written as documentation of the generated entity. This annotation receives an array of string, where each item of the array corresponds to a line of the documentation.
- **`@Deprecated`**: If this annotation is added to the entity, it causes the generated entity to be marked as deprecated.
- Any annotation that is outside the Uaithne annotation package (`org.uaithne.annotations.*`) will be generated in the resulting entity.

Information that can be added to the field definitions:

- **`@Id`**: Indicates that the field is part of the identifier of the record. It is possible to have several fields that make up the identifier of the record; but, if there is only one, the operations that use the id or return the id can be generated.

  The fields marked as identifiers are what are used to implement the `equals` and `hashCode` methods, so that two instances will be equal if all their fields marked with `@Id` are equal. If the entity does not have fields marked as identifiers, all the fields that make up the entity will be used to generate the `equals` and `hashCode` methods.

  If you have an entity with several fields that make up the identifier, it is recommended to mark it as an entity view and directly handle the operations that insert, update, or delete the record.

- **`@Optional`**: Indicates that the field is optional, so it supports null. If a field does not have

the `@Optional` annotation, it is assumed to be mandatory, and consequently it does not support null.

- **`@ExcludeFromConstructor`**: Indicates that the field should not be included in the constructor of the generated entity class. Fields marked with `@Optional` are automatically excluded from the constructor of the class of the generated entity.
- **`@ExcludeFromObject`**: Indicates that this field should not be included in the class of the generated entity. Its existence has other purposes and in no case should be part of the Java code generated for the entity.
- **`@ExcludeFromToString`**: Indicates that this field should not be included in the implementation of the `toString` method of the generated class.
- **`@DefaultValue`**: It indicates which is the default value to be assigned to the field during the class initialization of the generated entity. The value indicated with this annotation is translated into an assignment of the value to the field before the constructor is executed.

  Example:

  ```
  @EntityView
  class _Foo {
      @DefaultValue("10")
      Integer i;
  }
  ```

  Generate:

  ```
  public class Foo implements Serializable {

      private Integer i = 10;

      [getters & setters ...]
      [toString, equals & hashCode methods ...]

      public Foo() { }

      public Foo(Integer i) {
          this.i = i;
      }
  }
  ```

  The `@DefaultValue` annotation receives a string that will be interpreted according to the data type of the field. If the type is **boolean or Boolean** the text must be `true` or `false`. If it is a **numeric type** (`byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, `float`, `Float`, `double`, `Double`) the text must be a number. If it is **`BigInteger` or `BigDecimal`**, the text is the one that is going to be used to initialize the new instance of these classes. If it is **`char` or `Character`** the text must be a character that will be appropriately quoted. If it is **`String`**, the text must be a sequence of characters that will be properly quoted. If it is **any other type**, the text will be treated as the Java code required to perform the initialization.

- **`@Doc`**: If this annotation is added to the field, it causes the text indicated in it to be written as documentation of this in the generated entity. This annotation receives an array of string, where each item of the array corresponds to a line of the documentation.
- **`@Deprecated`**: If this annotation is added to the field, it causes the field in the generated entity to be marked as deprecated.

- **transient**: If the non-serializable field is marked, this field is marked as non-serializable in the generated entity.
- Any annotation that is outside the Uaithne annotation package (`org.uaithne.annotations.*`) will be generated in the field of the resulting entity.

Operations

The operations are created by adding the annotation `@Operation` (or any other more specific annotation that indicates the type of operation) to a class that contains the fields that will contain the necessary information to be able to carry out its execution. The operation returns a result whose type is specified in the result parameter of the annotation (in those operations whose result type is not implicit). The operations must always be contained within a module.

Example:

```
@OperationModule
public class _synchronization {

    @Operation(result = Void.class)
    class _SyncCalendar {
        Integer calendarId;
        // [Any other parameter required for doing the synchronization ...]
    }

}
```

Note: If you want to have an operation that does not return a value, the `Void` class must be used as the result type, whose only valid value is null.

Limitation: It is not possible to specify the result of operations that include generic arguments.

Different kinds of operations

Uaithne supports different kinds of operations, which indicate to the generator the semantics of this, which is especially useful when generating the necessary code to access the database. The different kinds of operations supported are:

General operations:

- **`@Operation`**: This operation is the most general of all, lacking any semantics, whose implementation must always be provided by the programmer

Query operations:

- **`@SelectOne`**: This operation returns the entity whose fields match the criteria indicated in the operation.
- **`@SelectMany`**: This operation returns a list of entities whose fields match the criteria indicated in the operation.
- **`@SelectPage`**: This operation returns a list of entities as a paged view whose fields match the criteria indicated in the operation.
- **`@SelectCount`**: This operation returns the number of entities whose fields match the criteria indicated in the operation.

- **`@SelectEntityById`**: This operation returns the entity whose id is the one indicated in the operation.

Entity modification operations:

- **`@InsertEntity`**: This operation inserts a record with the values indicated in the entity contained by the operation. This operation returns, as specified, the inserted record id (by default) or the number of inserted records.
- **`@UpdateEntity`**: This operation updates a record with the values indicated in the entity contained by the operation. The record to be updated is the one that matches the id indicated in the entity. This operation returns the number of updated records.
- **`@DeleteEntityById`**: This operation removes a record whose id is the one indicated in the operation. This operation returns the number of deleted records.
- **`@SaveEntity`**: This operation inserts or updates a record with the values indicated in the entity contained by the operation. The record to be updated is the one that matches with the id indicated in the entity; if the id is null, a new record is inserted. This operation returns, as specified, the record id inserted or updated (by default) or the number of records inserted or updated.
- **`@MergeEntity`**: This operation updates a record with the values indicated in the entity contained by the operation whose values are different from null. The record to be updated is the one that matches with the id indicated in the entity. This operation returns the number of updated records.

Data modification operations not restricted to entities:

- **`@Insert`**: This operation inserts a record with the values indicated in the operation. This operation returns, as specified, the inserted record id (by default) or the number of inserted records.
- **`@Update`**: This operation updates a record with the values indicated in the operation. This operation returns the number of updated records.
- **`@Delete`**: This operation removes the records with the values indicated by the operation. This operation returns the number of deleted records.

@Operation

This is the most general operation of all, and it has no associated semantics. So, it is an operation that executes "something" and returns "some" result. The fields of the operation will contain the information necessary for it to be carried out.

Parameters:

- **`result`**: indicates the result data type of the operation.

Example: For the next operation definition

```
@Operation(result = Void.class)
class _SyncCalendar {
    Integer calendarId;
    [Any other parameter required for doing the synchronization ...]
}
```

The next class that represents the operation is generated:

```
public class SyncCalendar implements Operation<Void> {

    private Integer calendarId;
    [Any other parameter required for doing the synchronization ...]

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public SyncCalendar() { }

    public SyncCalendar(Integer calendarId [, Any other parameter...]) {
        this.calendarId = calendarId;
        [Any other parameter initialization ...]

    }

}
```

@SelectOne

This operation retrieves an entity (or entity view) from a data source, typically a database. The fields of the operation will contain the necessary information to recover the desired entity from the data source.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`limit`**: if set to `true`, it indicates to the backend (typically the database access implementation) that although it can potentially return more than one record, if it occurs, the operation returns the first record found instead of giving an error.
- **`related`**: when the result of the operation is not an entity, you must indicate in this field the entity to which the operation refers.

Example: For the next operation definition

```
@SelectOne(result = _Calendar.class)
class _SelectCalendarByTitle {
    String title;
}
```

The next class that represents the operation is generated:

```
public class SelectCalendarByTitle implements Operation<Calendar> {

    private String title;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public SelectCalendarByTitle() { }

    public SelectCalendarByTitle(String title) {
```

```
        this.title = title;
    }
}
```

@SelectMany

This operation retrieves a list of entities (or entity views) from a data source, typically a database. The fields of the operation will contain the necessary information to recover the desired entities from the data source.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`distinct`**: if set to `true`, it indicates to the backend (typically the database access implementation) that although it can potentially return repeated records, in case of occurrence, it must be sure to return unique records (dropping the ones that were repeated).
- **`related`**: when the result of the operation is not an entity, you must indicate in this field the entity to which the operation refers.

Example: For the next operation definition

```
@SelectMany(result = _Event.class)
class _SelectMomentEvents {
    Integer calendarId;
    Date moment;
}
```

The next class that represents the operation is generated:

```
public class SelectMomentEvents implements Operation<List<Event>> {

    private Integer calendarId;
    private Date moment;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public SelectMomentEvents() { }

    public SelectMomentEvents(Integer calendarId, Date moment) {
        this.calendarId = calendarId;
        this.moment = moment;
    }
}
```

@SelectPage

This operation retrieves a paged list of entities (or entity views) from a data source, typically a database. The fields of the operation will contain the necessary information to recover the desired entities from the data source.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`distinct`**: if set to `true`, it indicates to the backend (typically the database access implementation) that although it can potentially return repeated records, in case of occurrence, it must be sure to return unique records (dropping the ones that were repeated).
- **`related`**: when the result of the operation is not an entity, you must indicate in this field the entity to which the operation refers.

Example: For the next operation definition

```
@SelectPage(result = _Event.class)
class _SelectCalendarEvents {
    Integer calendarId;
}
```

The next class that represents the operation is generated:

```
public class SelectCalendarEvents implements DataPageRequest, Operation<DataPage<Event>> {

    private Integer calendarId;
    private BigInteger limit;
    private BigInteger offset;
    private BigInteger dataCount;
    private boolean onlyDataCount;

    [getters & setters ...]

    @Override
    public BigInteger getMaxRowNumber() {
        if (limit == null) {
            return null;
        }
        if (offset == null) {
            return limit;
        }
        return limit.add(offset);

    }

    [toString, equals & hashCode methods ...]

    public SelectCalendarEvents() { }

    public SelectCalendarEvents(Integer calendarId) {
        this.calendarId = calendarId;
    }

    public GetCalendarEvents(Integer calendarId, BigInteger limit) {
        this.calendarId = calendarId;
        this.limit = limit;
    }

    public SelectCalendarEvents(Integer calendarId, BigInteger limit, BigInteger offset) {
```

```
        this.calendarId = calendarId;
        this.limit = limit;
        this.offset = offset;
    }

    public SelectCalendarEvents(Integer calendarId, BigInteger limit, BigInteger offset, BigInteger dataCount) {
        this.calendarId = calendarId;
        this.limit = limit;
        this.offset = offset;
        this.dataCount = dataCount;
    }

    public SelectCalendarEvents(Integer calendarId, boolean onlyDataCount) {
        this.calendarId = calendarId;
        this.onlyDataCount = onlyDataCount;
    }
}
```

The `@SelectPage` operations implement the `DataPageRequest` interface; this interface defines a series of properties that any `@SelectPage` operation must have to control the paging of the data. The `DataPageRequest` interface that is defined as:

```
public interface DataPageRequest extends Serializable {

    public BigInteger getLimit();
    public void setLimit(BigInteger limit);

    public BigInteger getOffset();
    public void setOffset(BigInteger offset);

    public BigInteger getMaxRowNumber();

    public BigInteger getDataCount();
    public void setDataCount(BigInteger dataCount);

    public boolean isOnlyDataCount();
    public void setOnlyDataCount(boolean onlyDataCount);
}
```

In addition to the fields indicated for the operation, some other more are automatically added:

- **`limit`**: indicates the number of records to be obtained in a data page, that is, it is the maximum size of the page. If this field is null, it indicates that it has no limit, so the maximum number of possible records must be brought back.
- **`offset`**: indicates the number of records to be ignored prior to the data page that you wish to consult. **Example**: If you have the size of the page as 20 records, and you want to obtain the third page, the value of this field should be (3 - 1) * 20, which is 40 records before being ignored, since you will get the records 41 to 60 (if you consider that the indexes start at one). If this field is null, it indicates that no record will be ignored.
- **`maxRowNumber`**: this is a read-only property that is calculated as the sum of the limit and the offset and represents the maximum index of the record to be consulted. **Example**: If you have the size of the page is 30 records, and you will ignore the first 90 records (fourth page is desired) the value of this property would be 30 + 90, which is 120, so it is will get records

91 to 120 (considering that the indexes start at zero).
- **`dataCount`**: indicates the total number of records. If you specify the value of this field in the operation, it causes that the operation not to have to consult the total number of records, instead it is assumed that the value is the one indicated here, and this is useful to avoid the additional query that counts the records.
- **`onlyDataCount`**: if `true` is specified in this field, it causes the operation to only check the total amount of the record, without bringing the data of any of the pages.

The result of an `@SelectPage` operation is a `DataPage` object that contains the list of records on the page as well as information on the total number of records, page size, and number of records previously ignored. The `DataPage` class is defined as:

```
public class DataPage<RESULT> implements Serializable {

    private BigInteger limit;
    private BigInteger offset;
    private BigInteger dataCount;
    private List<RESULT> data;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public DataPage() { }

    public DataPage(BigInteger limit, BigInteger offset, BigInteger dataCount, List<RESULT> data) {
        this.limit = limit;
        this.offset = offset;
        this.dataCount = dataCount;
        this.data = data;
    }
}
```

The DataPage class contains the following fields:

- **`limit`**: contains the `limit` used in the operation that returned this object and represents the maximum size of the page.
- **`offset`**: contains the `offset` used in the operation that returned this object and represents the number of previous records that have been ignored.
- **`dataCount`**: contains the total number of records; if the operation specified one value, the value of this field is the one indicated in the operation; but, if not, the value is the count of the total number of existing records without paging.
- **`data`**: contains the list of the records that belong to the requested page.

@SelectCount

This operation retrieves the number of entities (or entity views) from a data source, typically a database. The fields of the operation will contain the necessary information to determine which entities are to be counted in the data source.

Parameters:

- **`result`**: indicates which type of data is the result of the operation; by default it is

`BigInteger`; although, it would be perfectly valid to indicate any other type of numerical data, such as `Integer` or `Long`. The result of this operation is always a number with the number of records found.
- **`related`**: you must indicate in this field the entity to which the operation refers.

Example: For the next operation definition

```
@SelectCount(related = _Event.class)
class _CountEventsInCalendar {

    Integer calendarId;

}
```

The next class that represents the operation is generated:

```
public class CountEventsInCalendar implements Operation<BigInteger> {

    private Integer calendarId;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public CountEventsInCalendar() { }

    public CountEventsInCalendar(Integer calendarId) {
        this.calendarId = calendarId;
    }

}
```

@SelectEntityById

This operation retrieves an entity (or entity view) given its identifier from a data source, typically a database. Operations of this type assume that they receive the identifier of the entity (which type is the type of the field that carries the annotation `@Id` in the entity) and do not admit additional fields.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`limit`**: if set to `true`, it indicates to the backend (typically the database access implementation) that although it can potentially return more than one record, if it occurs, the operation returns the first record found instead of giving an error.

Requirements:

- This operation requires that the resulting entity (or entity view) have a single identifier field declared.

Example: For the next operation definition

```
@SelectEntityById(result = _Event.class)
class _SelectEventById {
    // No fields allowed here
```

```
}
```

The next class that represents the operation is generated:

```
public class SelectEventById implements SelectByIdOperation<Integer, Event> {

    private Integer id;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public SelectEventById() { }

    public SelectEventById(Integer id) {
        this.id = id;
    }
}
```

Fields of the generated operation:

- **`id`**: identifier of the entity to be recovered (which is the type of field that carries the `@Id` annotation in the entity).
- No additional field is allowed. `@SelectEntityById` operations must be defined without any field within it. The `id` field will automatically be added.

The generated operation implements the `SelectByIdOperation` interface; this interface extends from `Operation` and defines the `id` property. The definition of the `SelectByIdOperation` interface is:

```
public interface SelectByIdOperation<IDTYPE, RESULT> extends Operation<RESULT> {

    public IDTYPE getId();
    public void setId(IDTYPE id);
}
```

@InsertEntity

This operation inserts an entity (or entity view) into a data source, typically a database. Operations of this type assume that they receive the entity and do not support additional fields. These operations return the identifier of the inserted record (by default) or the number of records inserted (if the value of the `returnLastInsertedId` property of the annotation is set to `false`) .

Parameters:

- **`value`**: indicates what type of entity to be inserted. **Note**: in Java, when the property of the annotation is called value and it is the only property to be used, it is possible to omit the name of the property and specify the value directly.
- **`returnLastInsertedId`**: if set to `false`, it indicates that the operation must return the number of inserted records (which type is `Integer`), otherwise (by default), the operation returns the id of the inserted record (which type is the type of the field that carries the annotation `@Id` in the entity).

Requirements:

- This operation requires that the inserted entity (or entity view) have a single identifier field declared, unless the `returnLastInsertedId` field of the annotation has been set to false, in which case this requirement is not required.

Example: For the next operation definition

```
@InsertEntity(_Event.class)
class _InsertEvent {
    // No fields allowed here
}
```

The next class that represents the operation is generated:

```
public class InsertEvent implements InsertValueOperation<Event, Integer> {

    private Event value;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public InsertEvent() { }

    public InsertEvent(Event value) {
        this.value = value;
    }
}
```

Fields of the generated operation:

- **`value`**: entity to be inserted.
- No additional field is allowed. `@InsertEntity` operations must be defined without any field within it. The `value` field will automatically be added.

The generated operation implements the `InsertValueOperation` interface; this interface extends from `Operation` and defines the `value` property. The definition of the `InsertValueOperation` interface is:

```
public interface InsertValueOperation<VALUE, RESULT> extends Operation<RESULT> {
    public VALUE getValue();
    public void setValue(VALUE value);
}
```

Example: If in the definition of the operation was indicated, you want the result to be the number of records inserted

```
@InsertEntity(value = _Event.class, returnLastInsertedId = false)
class _InsertEvent {
    // No fields allowed here
}
```

The next class that represents the operation is generated:

```
public class InsertEvent implements JustInsertValueOperation<Event, Integer> {
```

```
    private Event value;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public InsertEvent() { }

    public InsertEvent(Event value) {
        this.value = value;
    }
}
```

Fields of the generated operation:

- **`value`**: entity to be inserted.
- No additional field is allowed, the `@InsertEntity` operations must be defined without any field within it. The `value` field will automatically be added.

The generated operation implements the `JustInsertValueOperation` interface; this interface extends from `Operation` and defines the `value` property. The definition of the `JustInsertValueOperation` interface is:

```
public interface JustInsertValueOperation<VALUE, RESULT> extends Operation<RESULT> {
    public VALUE getValue();
    public void setValue(VALUE value);
}
```

@UpdateEntity

This operation updates an entity (or entity view) in a data source, typically a database. Operations of this type assume that they receive the entity and do not admit additional fields. These operations return the number of updated records (which type is `Integer`). The fields to update are all the fields of the entity except the identifier that is used to determine which record is the one to be updated.

Parameters:

- **`value`**: indicates the type of entity to be updated. The id of the indicated entity will correspond to the id of the record to be updated. **Note**: in Java, when the property of the annotation is called value and it is the only property to be used, it is possible to omit the name of the property and specify the value directly.

Requirements:

- This operation requires that the entity (or entity view) to be updated has a single identifier field declared.

Example: For the next operation definition

```
@UpdateEntity(_Event.class)
class _UpdateEvent {
    // No fields allowed here
}
```

The next class that represents the operation is generated:

```
public class UpdateEvent implements UpdateValueOperation<Event, Integer> {

    private Event value;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public UpdateEvent() { }

    public UpdateEvent(Event value) {
        this.value = value;
    }
}
```

Fields of the generated operation:

- **`value`**: entity to be updated.
- No additional field is allowed. `@UpdateEntity` operations must be defined without any field within it. The `value` field will automatically be added.

The generated operation implements the `UpdateValueOperation` interface; this interface extends from `Operation` and defines the `value` property. The definition of the `UpdateValueOperation` interface is:

```
public interface UpdateValueOperation<VALUE, RESULT> extends Operation<RESULT> {

    public VALUE getValue();
    public void setValue(VALUE value);
}
```

@DeleteEntityById

This operation removes an entity (or entity view) given its identifier from a data source, typically a database. Operations of this type assume that they receive the identifier of the entity (which type is the type of the field that carries the annotation `@Id` in the entity) and do not admit additional fields. These operations return the number of records eliminated (which type is `Integer`).

Parameters:

- **`related`**: indicates the type of the entity to be eliminated.

Requirements:

- This operation requires that the entity (or entity view) to be eliminated has a single identifier field declared.

Example: For the next operation definition

```
@DeleteEntityById(related = _Event.class)
class _DeleteEventById {
    // No fields allowed here
}
```

The next class that represents the operation is generated:

```
public class DeleteEventById implements SelectByIdOperation<Integer, Event> {
    private Integer id;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public DeleteEventById() { }

    public DeleteEventById(Integer id) {
        this.id = id;
    }
}
```

Fields of the generated operation:

- **`id`**: identifier of the entity to be eliminated (which type is the type of field that carries the `@Id` annotation in the entity).
- No additional field is allowed. `@DeleteEntityById` operations must be defined without any field within it. The `id` field will automatically be added.

The generated operation implements the `DeleteByIdOperation` interface; this interface extends from `Operation` and defines the `id` property. The definition of the `DeleteByIdOperation` interface is:

```
public interface DeleteByIdOperation<IDTYPE, RESULT> extends Operation<RESULT> {
    public IDTYPE getId();
    public void setId(IDTYPE id);
}
```

@SaveEntity

This operation inserts or updates an entity (or entity view) in a data source, typically a database. Operations of this type assume that they receive the entity and do not admit additional fields. These operations return the identifier of the inserted or updated record (by default) or the number of records inserted or updated (if the value of the property `returnLastInsertedId` of the annotation is set to `false`). The fields to update are all the fields of the entity except the identifier that is used to determine which record is the one to be updated; in case of being null the id is inserted a new record.

Parameters:

- **`value`**: indicates the type of entity to be inserted or updated. **Note**: in Java, when the property of the annotation is called value and it is the only property to be used, it is possible to omit the name of the property and specify the value directly.indica cual es el tipo de la entidad a ser insertada o actualizada.
- **`returnLastInsertedId`**: if set to false it indicates that the operation must return the number of inserted or updated records (which type is `Integer`), otherwise (by default), returns the id of the inserted or updated record (which type is the type of field that carries the `@Id` annotation in the entity).

Requirements:

- This operation requires that the entity (or entity view) to be inserted or updated has a single identifier field declared.
- This operation requires that the data type of the identifier field admit null; so, it can not be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`; instead of them you can use `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` or any other object.

Example: For the next operation definition

```
@SaveEntity(_Event.class)
class _SaveEvent {
    // No fields allowed here
}
```

The next class that represents the operation is generated:

```
public class SaveEvent implements InsertValueOperation<Event, Integer> {

    private Event value;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public SaveEvent() { }

    public SaveEvent(Event value) {
        this.value = value;
    }

}
```

Fields of the generated operation:

- **`value`**: entity to be inserted or updated.
- No additional field is allowed. `@SaveEntity` operations must be defined without any field within it. The `value` field will automatically be added.

The generated operation implements the `SaveValueOperation` interface; this interface extends from `Operation` and defines the `value` property. The definition of the `SaveValueOperation` interface is:

```
public interface SaveValueOperation<VALUE, RESULT> extends Operation<RESULT> {

    public VALUE getValue();
    public void setValue(VALUE value);
}
```

Example: If in the definition of the operation you indicate you want as result the number of inserted or updated records

```
@SaveEntity(value = _Event.class, returnLastInsertedId = false)
class _SaveEvent {
    // No fields allowed here
```

```
}
```

The next class that represents the operation is generated:

```
public class SaveEvent implements JustSaveValueOperation<Event, Integer> {

    private Event value;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public SaveEvent() { }

    public SaveEvent(Event value) {
        this.value = value;
    }
}
```

Fields of the generated operation:

- **`value`**: entity to be inserted or updated.
- No additional field is allowed. `@SaveEntity` operations must be defined without any field within it. The `value` field will automatically be added.

The generated operation implements the `JustSaveValueOperation` interface; this interface extends from `Operation` and defines the `value` property. The definition of the `JustSaveValueOperation` interface is:

```
public interface JustSaveValueOperation<VALUE, RESULT> extends Operation<RESULT> {

    public VALUE getValue();
    public void setValue(VALUE value);
}
```

@MergeEntity

This operation updates an entity (or entity view) in a data source, typically a database. Operations of this type assume that they receive the entity and do not admit additional fields. These operations return the number of updated records (which type is `Integer`). The fields to update are all the fields of the entity whose value is different from null except the identifier that is used to determine which record is the one to be updated.

Parameters:

- **`value`**: indicates the type of entity to be updated. The id of the indicated entity will correspond to the id of the record to be updated. **Note**: in Java, when the property of the annotation is called value and it is the only property to be used, it is possible to omit the name of the property and specify the value directly.

Requirements:

- This operation requires that the entity (or entity view) to be eliminated has a single identifier field declared.

Example: For the next operation definition

```
@MergeEntity(_Event.class)
class _MergeEvent {
    // No fields allowed here
}
```

The next class that represents the operation is generated:

```
public class MergeEvent implements MergeValueOperation<Event, Integer> {

    private Event value;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public MergeEvent() { }

    public MergeEvent(Event value) {
        this.value = value;
    }

}
```

Fields of the generated operation:

- **`value`**: entity to be updated.
- No additional field is allowed. `@MergeEntity` operations must be defined without any field within it. The `value` field will automatically be added.

The generated operation implements the `MergeValueOperation` interface, this interface extends from `Operation` and defines the `value` property. The definition of the `MergeValueOperation` interface is:

```
public interface MergeValueOperation<VALUE, RESULT> extends Operation<RESULT> {

    public VALUE getValue();
    public void setValue(VALUE value);
}
```

@Insert

This operation inserts a record into a data source, typically a database; and returns the identifier of the inserted record (by default) or the number of records inserted (if the value of the `returnLastInsertedId` property of the annotation is set to `false`). The values to insert in the registry are those specified as fields of the operation. This operation is much more flexible than `@InsertEntity` because it allows you to specify the fields to insert in the new record.

Parameters:

- **`related`**: indicates which is the related entity to the record to be inserted.
- **`returnLastInsertedId`**: if set to `false` it indicates that the operation must return the number of records inserted (which type is `Integer`), otherwise (by default), returns the id of the inserted record (which type is the type of the field that carries the annotation `@Id` in the

entity).

Requirements:

- This operation requires that the entity (or entity view) associated with the entry have a single identifier field declared, unless the `returnLastInsertedId` field of the annotation has been set to false, in which case this requirement is not required.

Example: For the next operation definition

```
@Insert(related = _Calendar.class)
class _InsertCalendarByTitle {
    String title;
}
```

The next class that represents the operation is generated:

```
public class InsertCalendarByTitle implements Operation<Integer> {
    private String title;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public InsertCalendarByTitle() { }

    public InsertCalendarByTitle(String title) {
        this.title = title;
    }
}
```

Example: If in the definition of the operation was indicated you want as result the number of inserted records

```
@Insert(related = _Calendar.class, returnLastInsertedId = false)
class _InsertCalendarByTitle {
    String title;
}
```

The next class that represents the operation is generated:

```
public class InsertCalendarByTitle implements Operation<Integer> {

    private String title;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public InsertCalendarByTitle() { }

    public InsertCalendarByTitle(String title) {
        this.title = title;
    }

}
```

@Update

This operation updates a record in a data source, typically a database; and returns the number of updated records (which is of the Integer type). The fields to update are all the fields of the operation that have the annotation `@SetValue`; if in this annotation the property `ignoreWhenNull` is set to true, only the field is updated when its value in the operation is different from null. All those fields of the operation that do not have the `@SetValue` annotation are used to determine which record is the one to be updated. This operation is much more flexible than `@UpdateEntity` because it allows you to specify the fields you want to use to search the records to be updated (you can have several, which do not necessarily have to be the registry's identifier); it also allows you to specify which fields are going to be updated in the record.

Parameters:

- **`related`**: indicates which is the entity related to the record to be updated.

The `@Update` operation updates all the records that match the fields supplied in the operation, except those fields that have the `@SetValue` annotation; these fields are the ones that will be updated with the values supplied in the operation.

`@SetValue parameters`:

- **ignoreWhenNull**: indicates whether this field should be skipped when its value is null. That is, if it is `false`, the field is updated with the value supplied in the operation, even when it is null, in which case the new value will be null; but if `true`, the field is updated with the supplied value as long as it is not null, but if it is null, the value is not updated. Setting `ignoreWhenNull` to `true` is useful to get certain fields updated optionally. The default value is `false`.

  Note: Setting `ignoreWhenNull` to `true` requires that the field additionally have the `@Optional` annotation indicating that the field is optional, and therefore can support null.

Example: For an operation that allows updating the description of an event given its title and the calendar id to which it belongs

```
@Update(related = _Event.class)
class _UpdateEventDescrptionByCalendarIdAndTitle {
    Integer calendarId;
    String title;
    @SetValue
    String description;
}
```

The next class that represents the operation is generated:

```
public class UpdateEventDescrptionByCalendarIdAndTitle implements Operation<Integer> {

    private Integer calendarId;
    private String title;
    private String description;
```

```
    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public UpdateEventDescrptionByCalendarIdAndTitle() { }

    public UpdateEventDescrptionByCalendarIdAndTitle(Integer calendarId, String title, String description) {
        this.calendarId = calendarId;
        this.title = title;
        this.description = description;
    }
}
```

Example: For an operation that allows updating the title of a calendar and optionally its description given its identifier

```
@Update(related = _Calendar.class)
class _UpdateCanlendarTitleAndDescription {
    Integer id;
    @SetValue
    String title;
    @Optional
    @SetValue(ignoreWhenNull = true)
    String description;
}
```

The next class that represents the operation is generated:

```
public class UpdateCanlendarTitleAndDescription implements Operation<Integer> {

    private Integer id;
    private String title;
    private String description;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public UpdateCanlendarTitleAndDescription() { }

    public UpdateCanlendarTitleAndDescription(Integer id, String title) {
        this.id = id;
        this.title = title;
    }
}
```

@Delete

This operation removes a record from a data source, typically a database and returns the number of deleted records (which type is `Integer`). All fields of the operation are used to determine which or which records to delete. This operation is much more flexible than `@DeleteEntityById` since it allows you to specify the fields by which are searched the records to be deleted (there may be several, which do not necessarily have to be the identifier of the record).

Parameters:

- **`related`**: indicates which is the entity related with the record to be deleted.

Example: For the next operation definition

```
@Delete(related = _Event.class)
class _DeleteEventByCalendarIdAndTitle {

    Integer calendarId;
    String title;

}
```

The next class that represents the operation is generated:

```
public class DeleteEventByCalendarIdAndTitle implements Operation<Integer> {

    private Integer calendarId;
    private String title;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]

    public DeleteEventByCalendarIdAndTitle() { }

    public DeleteEventByCalendarIdAndTitle(Integer calendarId, String title) {
        this.calendarId = calendarId;
        this.title = title;
    }
}
```

Additional information for an operation definition

If desired, it is possible to alter some aspects of the generated code to get it treated differently through a series of complementary annotations

Información que se pueden añadir sobre la operación:

- **`abstract`**: if the operation is marked as abstract, it causes the generated operation to be abstract.
- **`extends`**: adding a base class (using `extends`) to the operation causes the generated operation to extend from that base class.
- **`implements`**: if interfaces are implemented (using `implements`) in the entity, the generated operation implements the indicated interfaces.
- **`@Doc`**: if this annotation is added to the entity, it causes the text indicated in it to be written as documentation of the generated operation. This annotation receives an array of string, where each item of the array corresponds to a line of the documentation.
- **`@Deprecated`**: if this annotation is added to the operation, it causes the generated entity to be marked as deprecated.
- **`@Manually`**: if this annotation is added, it is indicated to the backend (typically the database) that an implementation for this operation should not be generated, since the implementation will be provided manually by the programmer.
- Any annotation that is outside the Uaithne annotation package

(`org.uaithne.annotations.*`) will be generated in the resulting operation.
- If you want an operation to have no result, you must use the `Void` class as the result type, whose only valid value is null.

Information that can be added over the fields:

- **`@Optional`**: indicates that the field is optional, so it supports null. If a field does not have the `@Optional` annotation it is assumed to be mandatory, and consequently it does not support null.
- **`@ExcludeFromConstructor`**: indicates that the field should not be included in the constructor of the generated operation class. The fields marked with `@Optional` are automatically excluded from the constructor of the generated operation class.
- **`@ExcludeFromObject`**: indicates that this field should not be included in the class of the operation generated; its existence has other purposes and in no case should be part of the Java code generated for the operation.
- **`@ExcludeFromToString`**: indicates that this field should not be included in the implementation of the `toString` method of the generated class.
- **`@DefaultValue`**: indicates which is the default value to be assigned to the field during the initialization of the generated operation class. The value indicated with this annotation must be understood as being translated into an assignment of the value to the field before the constructor is executed.

  Example:

```
@Operation(result = Void.class)
class _Foo {
    @DefaultValue("10")
    Integer i;
}
```

  Generate:

```
public class Foo implements Operation<Void> {

    private Integer i = 10;

    [getters & setters ...]
    [toString, equals & hashCode methods ...]
    [execute, executePostOperation & getExecutorSelector methods ...]

    public Foo() { }

    public Foo(Integer i) {
        this.i = i;
    }
}
```

  The `@DefaultValue` annotation receives a string that will be interpreted according to the data type of the field. If the type is **boolean or Boolean** the text must be `true` or `false`. If it is a **numeric type** (`byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, `float`, `Float`, `double`, `Double`) the text must be a number. If it is **`BigInteger` or `BigDecimal`** the text is the one that is going to be used to initialize the new instance of these classes. If it is **`char`**

or `Character` the text must be a character that will be appropriately quoted. If it is **`String`** the text must be a sequence of characters that will be properly quoted. If it is **any other type** the text will be treated as the Java code required to perform the initialization.

- **`@Doc`**: If this annotation is added to the field, it causes the text indicated in it to be written as documentation of this in the generated entity. This annotation receives an array of string, where each item of the array corresponds to a line of the documentation.
- **`@Deprecated`**:If this annotation is added to the field, it causes the field in the generated entity to be marked as deprecated.
- **`@Manually`**: if this annotation is added, it is indicated to the backend (typically the database) that no logic associated with this field should be generated, since the implementation will be provided manually by the programmer. If this field should not be used at all by the data access layer since it obtains its value in higher layers, the `onlyProgrammatically` property of this annotation must be set to `true`.
- **transient**: If the non-serializable field is marked, this field is marked as non-serializable in the generated entity.
- Any annotation that is outside the Uaithne annotation packet (`org.uaithne.annotations.*`) will be generated in the field of the resulting operation.

Operations' characteristics

- All operations generated directly or indirectly implement the `Operation` interface.
- All the generated operations reimplement the `toString` method that returns the name of the operation and the value of each of the fields, except those fields that have the annotation `@ExcludeFromToString` or `@ExcludeFromObject`.
- All generated operations reimplement the `equals` and `hashCode` method; in the new implementation of these methods it is considered that two operations are equal if their type and the value of each of their fields are equal; those fields with the annotation `@ExcludeFromObject` are excluded or that are marked as `transient`.
- All generated operations have a constructor with all mandatory fields; that is, those that do not have the `@Optional` annotation; those fields that have the annotation `@ExcludeFromObject` are excluded from the constructor. The fields in the constructor appear in the same order in which they are declared in the generator input.
- All the operations generated implement each of the methods required by the framework for its operation, that is, it implements each of the methods of the `Operation` interface (only in some Uaithne configurations).
- All the generated operations are instantiable unless they are defined as abstract in the generator input; consequently, to use the operations it is enough to create a new instance of it (it is not necessary to create classes that extend from the operation).

Operations generated for an entity

If the entity is defined within a module, a series of operations can be generated by default (the annotation `@Entity` allows to indicate which ones should be generated, it can also be activated by default in the Uaithne configuration). In the name of the operation the string *`«Entity»`* is replaced by the name of the entity. If there is an operation of the same name, the operation of the entity with matching name is not generated.

- **`Insert`*`«Entity»`***: Insert an entity in the data source, return the id of the inserted record.

This operation is equivalent to the one generated with `@InsertEntity`.

- **`Update«Entity»`**: Update an entity in the data source, return the number of records affected. This operation is equivalent to the one generated with `@UpdateEntity`.
- **`Delete«Entity»ById`**: Deletes an entity from the data source given its id, returns the number of deleted records. This operation is equivalent to the one generated with `@DeleteEntity`.
- **`Select«Entity»ById`**: Retrieves an entity from the data source given its the id and returns the entity or null if it can not find it. This operation is equivalent to the one generated with `@SelectEntityById`.
- **`Save«Entity»`**: Insert or update an entity in the data source; if the identifier of the entity that receives the operation is null inserts a new record, if it does not update the record with that identifier. This operation returns the identifier of the inserted record. This operation is equivalent to the one generated with `@SaveEntity`.
- **`Merge«Entity»`**: Updates the fields of an entity in the data source whose values are different from null. This operation updates the record of the entity whose identifier matches the one indicated within it. This operation only updates the fields that have value in the entity; that is, those whose value is different from null; the fields with null values will be ignored. This operation is equivalent to the one generated with `@MergeEntity`.
- **`JustInsert«Entity»`**: Insert an entity in the data source; it return the number of records inserted without trying to recover the id of the inserted record. This operation is equivalent to the one generated with `@InsertEntity`, setting the value of the `returnLastInsertedId` property of the annotation to `false`.
- **`JustSave«Entity»`**: Insert or update an entity in the data source according to whether the identifier of the entity that receives the operation is null inserts a new record, if it does not update the record with that identifier. This operation returns the number of records affected without trying to recover the id of the inserted record. This operation is equivalent to the one generated with `@SaveEntity`, setting the value of the `returnLastInsertedId` property of the annotation to `false`.

In order to generate the operations for an entity, it is necessary that it be contained within a module and that it have a single field marked with the `@Id` annotation defined by it or by any entity from which it extends directly or indirectly.

In order to generate the `Save` or `JustSave` operations it is necessary that the data type of the field marked with the `@Id` annotation is a data type that can be assigned null (so it can not be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, instead of them you can use `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` or any other object).

Query generator

By using the code generator it is possible to tell Uaithne to generate the executors implementations to access to the database (including the SQL queries), thus reducing even more the work required by the programmer to have the system backend worked.

To generate the base access layer, the implementation of the executor is generated with the necessary Java code to invoke the database through the use of MyBatis, and, as well, an XML file of MyBatis is generated with the SQL queries to be executed for each operation.

Supported databases:

- Oracle (10, 11, 12)
- Sql Server (2005, 2008, 2012, 2014, 2016, 2017)
- PostgreSql (8.4, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 10.0)
- Derby (10.6, 10.7, 10.8, 10.9, 10.10, 10.11, 10.12, 10.13, 10.14)
- MySql (5.5, 5.6, 5.7)

To activate the generation of the database access layer, simply add the annotation `@MyBatisMapper` on the module definition class.

```
@OperationModule
@MyBatisMapper
public class _events {

}
```

The inclusion of the `@MyBatisMapper` annotation causes that two files are generated for each of the databases included in the Uaithne configuration; the first one with the implementation of MyBatis access in Java; and the second one with the MyBatis mapping XML that includes the SQL queries.

Queries by default for each operation

The central idea behind the generation of queries is that for each operation a query is generated by default, which through annotations, the programmer can indicate to the generator the changes required to achieve the desired query.

@Operation

For this operation no code is generated; its implementation must be provided by the programmer. This same effect can be achieved by adding the `@Manually` annotation on any other operation.

@SelectOne

In this operation all the fields of the resulting entity make up the `select` clause of the query and all the fields of the operation make up the `where` clause joined with the `and` operator. If the operation has the parameter `limit` to `true`, the code is added additional necessary for the query to return only the first record found.

Example: For the next operation definition

```
@SelectOne(result = _Calendar.class)
class _SelectCalendarByTitle {
    String title;
}
```

The next query is generated for PostgreSQL:

```
select
    id,
    title,
    description
from
    Calendar
where
    title = #{title,jdbcType=VARCHAR}
```

But if the parameter `limit` in the operation is setted to `true`:

```
@SelectOne(result = _Calendar.class, limit = true)
class _SelectCalendarByTitle {
    String title;
}
```

It generate the next query for PostgreSQL:

```
select
    id,
    title,
    description
from
    Calendar
where
    title = #{title,jdbcType=VARCHAR}
fetch next 1 rows only
```

@SelectMany

In this operation all the fields of the resulting entity make up the `select` clause of the query and all the fields of the operation make up the `where` clause joined with the `and` operator. If the operation has the parameter `distinct` to `true`, the `distinct` modifier is added to the `select` clause.

Example: For the next operation definition

```
@SelectMany(result = _Event.class)
class _SelectMomentEvents {
    Integer calendarId;
    Date moment;
}
```

The next query is generated for PostgreSQL:

```
select
    id,
```

```
    title,
    start,
    end,
    description,
    calendarId
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
    and moment = #{moment,jdbcType=TIMESTAMP}
```

But if the parameter `distinct` in the operation is setted to `true`:

```
@SelectMany(result = _Event.class, distinct = true)
class _SelectMomentEvents {
    Integer calendarId;
    Date moment;
}
```

It generate the next query for PostgreSQL:

```
select distinct
    id,
    title,
    start,
    end,
    description,
    calendarId
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
    and moment = #{moment,jdbcType=TIMESTAMP}
```

@SelectPage

In this operation, two different queries are used: one to count the number of records and another to obtain the data of the page. To count the number of records, a `select count(*)` is done and all the fields of the operation are included to make up the `where` clause joined between them with the `and` operator. The query to obtain the data of the page contains all the fields of the resulting entity in the `select` clause and all the fields of the operation make up the `where` clause joined between them with the `and` operator; the additional code necessary is added so that the query only returns the data of the requested page. If the operation has the `distinct` parameter to true, the `distinct` modifier is added to the `select` clause of both queries.

Example: For the next operation definition

```
@SelectPage(result = _Event.class)
class _SelectCalendarEvents {
    Integer calendarId;
}
```

The next query that counts the number of records is generated for PostgreSQL:

```
select
    count(*)
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
```

The next query that brings back the data of the requested page is also generated:

```
select
    id,
    title,
    start,
    end,
    description,
    calendarId
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
<if test='offset != null or limit != null'>
    <if test='offset != null'>offset #{offset,jdbcType=NUMERIC}</if>
    <if test='limit != null'>limit #{limit,jdbcType=NUMERIC}</if>
</if>
```

But if the parameter `distinct` in the operation is setted to `true`:

```
@SelectPage(result = _Event.class, distinct = true)
class _SelectCalendarEvents {
    Integer calendarId;
}
```

The next query that counts the number of records is generated for PostgreSQL:

```
select count(*) from (
select distinct
    id,
    title,
    start,
    end,
    description,
    calendarId
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
)
```

The next query that brings back the data of the requested page is also generated:

```
select distinct
    id,
    title,
    start,
    end,
```

```
    description,
    calendarId
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
<if test='offset != null or limit != null'>
    <if test='offset != null'>offset #{offset,jdbcType=NUMERIC}</if>
    <if test='limit != null'>limit #{limit,jdbcType=NUMERIC}</if>
</if>
```

@SelectCount

In this operation, a `select count(*)` is made that includes all the fields of the operation in the `where` clause of the query joined between them with the `and` operator.

Example: For the next operation definition

```
@SelectCount(related = _Event.class)
class _CountEventsInCalendar {
    Integer calendarId;
}
```

The next query is generated for PostgreSQL:

```
select
    count(*)
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
```

@SelectEntityById

In this operation all the fields of the resulting entity make up the `select` clause of the query and the id field (which is automatically added to the operation) is included in the `where` clause. If the operation has the `limit` parameter to `true`, the additional necessary code is added to the query to return only the first record found.

Example: For the next operation definition

```
@SelectEntityById(result = _Event.class)
class _SelectEventById {
    // No fields allowed here
}
```

The next query is generated for PostgreSQL:

```
select
    id,
    title,
    start,
    end,
    description,
    calendarId
```

```
from
    Event
where
    id = #{id,jdbcType=INTEGER}
```

But if the parameter `limit` in the operation is setted to `true`:

```
@SelectEntityById(result = _Event.class, limit = true)
class _SelectEventById {
    // No fields allowed here
}
```

It generate the next query for PostgreSQL:

```
select
    id,
    title,
    start,
    end,
    description,
    calendarId
from
    Event
where
    id = #{id,jdbcType=INTEGER}
fetch next 1 rows only
```

@InsertEntity

In this operation all the fields of the entity to be inserted contained in the `value` field (which is automatically added to the operation) are specified in the `insert into` clause with their respective values; except the fields marked with the annotation `@Id` which presence depends on the generation strategy of the id.

Example: For the next operation definition (where the id is automatically generated)

```
@InsertEntity(_Event.class)
class _InsertEvent {
    // No fields allowed here
}
```

The next query is generated for PostgreSQL:

```
insert into
    Event
(
    title,
    start,
    end,
    description,
    calendarId
) values (
    #{title,jdbcType=VARCHAR},
    #{start,jdbcType=TIMESTAMP},
    #{end,jdbcType=TIMESTAMP},
```

```
    #{description,jdbcType=VARCHAR},
    #{calendarId,jdbcType=INTEGER}
)
```

Note: returning the id of the inserted record may require the execution of a second query depending on the database manager.

@UpdateEntity

In this operation all the fields of the entity to be updated contained in the `value` field (which is automatically added to the operation) are translated into fields updates in the table except the fields marked with the `@Id` annotation that are used in the `where` clause joined between them with the `and` operator.

Example: For the next operation definition

```
@UpdateEntity(_Event.class)
class _UpdateEvent {
    // No fields allowed here
}
```

The next query is generated for PostgreSQL:

```
update
    Event
set
    title = #{title,jdbcType=VARCHAR},
    start = #{start,jdbcType=TIMESTAMP},
    end = #{end,jdbcType=TIMESTAMP},
    description = #{description,jdbcType=VARCHAR},
    calendarId = #{calendarId,jdbcType=INTEGER}
where
    id = #{id,jdbcType=INTEGER}
```

@DeleteEntityById

In this operation, a delete is made where the `where` clause includes the `id` field (which is automatically added to the operation).

Example: For the next operation definition

```
@DeleteEntityById(related = _Event.class)
class _DeleteEventById {
    // No fields allowed here
}
```

The next query is generated for PostgreSQL:

```
delete from
    Event
where
    id = #{id,jdbcType=INTEGER}
```

@SaveEntity

In this operation two different queries are used: one to insert a record and another to update it. The record is inserted if the id (the field annotated with `@Id`) is null in entity contained in the `value` field (that is automatically added to the operation) or updated if it is not null (so the record is updated with the supplied id).

For the insertion query, all the fields of the entity are included in the `insert` clause with their respective values, except the fields marked with the annotation `@Id` that their presence depends on the generation strategy of the id.

For the update query, all fields of the entity are translated field into field updates in the table, except for the fields marked with the `@Id` annotation that are used in the `where` clause joined with the `and` operator.

Example: For the next operation definition

```
@SaveEntity(_Event.class)
class _SaveEvent {
    // No fields allowed here
}
```

The next insert query is generated for PostgreSQL:

```
insert into
    Event
(
    title,
    start,
    end,
    description,
    calendarId
) values (
    #{title,jdbcType=VARCHAR},
    #{start,jdbcType=TIMESTAMP},
    #{end,jdbcType=TIMESTAMP},
    #{description,jdbcType=VARCHAR},
    #{calendarId,jdbcType=INTEGER}
)
```

The next update query is also generated:

```
update
    Event
set
    title = #{title,jdbcType=VARCHAR},
    start = #{start,jdbcType=TIMESTAMP},
    end = #{end,jdbcType=TIMESTAMP},
    description = #{description,jdbcType=VARCHAR},
    calendarId = #{calendarId,jdbcType=INTEGER}
where
    id = #{id,jdbcType=INTEGER}
```

Note: returning the id of the inserted record may require the execution of a second query

depending on the database manager.

@MergeEntity

In this operation, all the fields of the entity to be updated contained in the `value` field (which is automatically added to the operation) are translated into field updates in the table, as long as their value is different from null, except for the fields marked with the `@Id` annotation that are used in the `where` clause joined with the `and` operator.

Example: For the next operation definition

```
@MergeEntity(_Event.class)
class _MergeEvent {
    // No fields allowed here
}
```

The next query is generated for PostgreSQL:

```
update
    Event
<set>
    <if test='title != null'>title = #{title,jdbcType=VARCHAR},</if>
    <if test='start != null'>start = #{start,jdbcType=TIMESTAMP},</if>
    <if test='end != null'>end = #{end,jdbcType=TIMESTAMP},</if>
    <if test='description != null'>description = #{description,jdbcType=VARCHAR},</if>
    <if test='calendarId != null'>calendarId = #{calendarId,jdbcType=INTEGER}</if>
</set>
where
    id = #{id,jdbcType=INTEGER}
```

@Insert

In this operation all the fields of the operation are specified in the `insert` clause with their respective values.

Example: For the next operation definition (where the id is automatically generated)

```
@Insert(related = _Calendar.class)
class _InsertCalendarByTitle {
    String title;
}
```

The next query is generated for PostgreSQL:

```
insert into
    Calendar
(
    title
) values (
    #{title,jdbcType=VARCHAR}
)
```

Note: returning the id of the inserted record may require the execution of a second query depending on the database manager.

@Update

In this operation, all the fields marked with the `@SetValue` annotation are translated into field updates in the table; the fields that do not have that annotation are used in the `where` clause joined with the `and` operator. If in the `@SetValue` annotation the parameter `ignoreWhenNull` is set to `true`, only the field is updated if it has a value, if its value is null it is ignored.

Example: For the next operation definition

```
@Update(related = _Calendar.class)
class _UpdateCanlendarTitleAndDescription {
    Integer id;
    @SetValue
    String title;
    @Optional
    @SetValue(ignoreWhenNull = true)
    String description;
}
```

The next insert query is generated for PostgreSQL:

```
update
    Calendar
<set>
    title = #{title,jdbcType=VARCHAR},
    <if test='description != null'>description = #{description,jdbcType=VARCHAR}</if>
</set>
where
    id = #{id,jdbcType=INTEGER}
```

@Delete

In this operation, a `delete` is made where the `where` clause contains all the fields of the operation joined with the `and` operator.

Example: For the next operation definition

```
@Delete(related = _Event.class)
class _DeleteEventByCalendarIdAndTitle {

    Integer calendarId;
    String title;

}
```

The next insert query is generated for PostgreSQL:

```
delete from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
    and title = #{title,jdbcType=VARCHAR}
```

Customization of generated queries

Uaithne allows you to alter the queries generated using a series of complementary annotations that change the default behavior of the generator.

Basic annotations:

- **`@MappedName`**: It allows to indicate the name of the field or table in the database.
- **`@Manually`**: Tells the queries generator to ignore the operation or a field.
- **`@ValueWhenNull`**: Tells the queries generator to use the value indicated in this annotation when the field value is null.
- **`@ForceValue`**: Tells the queries generator to use the given value instead of the one supplied by the field.

Annotations that only affect the insertion:

- **`@HasDefaultValueWhenInsert`**: It indicates to the queries generator that the field has a default value in the database; so if the given value of the field is null, the default value of the database must be used when insert.

Annotations that allow you to adapt `where` clause:

- **`@Optional`**: It indicates to the generator that the value of this field is optional; so it must be treated appropriately. For example, it should only be taken into account in the `where` if the field value is not null.
- **`@Comparator`**: It allows to indicate to the queries generator which is the rule of comparison of a field with the value in the database. By default, if this annotation is not used it is compared for equality; but if the data type of the field is a list it checks that the value of the column is in the list.
- **`@CustomComparator`**: It allows to specify to the queries generator a complex comparison rule to be used to compare the value of the field with one in the database.

Annotation to sort the data:

- **`@OrderBy`**: Indicates to the generator that this field is an ordering clause of the resulting records by the query.

@MappedName

This annotation allows you to specify the name in the database of an entity or field.

In an entity Uaithne uses the name of the entity as the table name; but, if this annotation is added over an entity (or entity view), the value of this annotation will be used as the name of the table in the database.

In a field, either from an entity (or entity view) or from an operation, Uaithne uses the name of the field as the name of the column in the database by default; but, if this annotation is added over the field the value of this annotation will be used as the name of the column in the database.

Example: For the next entity definition

```
@Entity
@MappedName("table_calendar")
class _Calendar {
    @Id
    Integer id;
    @MappedName("calendar_title")
    String title;
    @Optional
    String description;
}
```

The next query is generated for the operation of `@SelectEntityById` in PostgreSQL:

```
select
    id,
    calendar_title as "title",
    description
from
    table_calendar
where
    id = #{id,jdbcType=INTEGER}
```

Tip: the value supplied in the `@MappedName` annotation represents an SQL fragment; so it is possible, for example, for an entity to provide a JOIN value or for a field to be a SQL fragment that calculates in value; as long as it makes sense for operations.

Example: For the next entity definition

```
@EntityView
@MappedName("calendar c join event e on c.id = e.calendar_id")
class _CalendarEvent {
    @MappedName("c.id")
    Integer calendarId;
    @MappedName("c.title")
    String calendatTitle;
    @Id
    @MappedName("e.id")
    Integer eventId;
    @MappedName("e.title")
    String eventTitle;
}
```

The next query is generated for the operation of `@SelectEntityById` in PostgreSQL:

```
select
    c.id as "calendarId",
    c.title as "calendatTitle",
    e.id as "eventId",
    e.title as "eventTitle"
from
    calendar c join event e on c.id = e.calendar_id
where
    e.id = #{id,jdbcType=INTEGER}
```

@Manually

This annotation allows to indicate to the generator of access to the database that the programmer is the one who will take care of the annotated element; so, it must be ignored during the generation.

If it is placed on an operation, it tells Uaithne to not generate the access to the database for this operation; so, the implementation in java of this operation must be provided by the programmer.

If it is placed on a field, either from an entity (or entity view) or from an operation, it causes the field in question to be ignored in the generated query (as if it did not exist).

Parameters:

- **onlyProgrammatically**: It indicates to the generator that this field will never be used in an SQL query (it only has utility in the experimental generation of stored procedures). Default value: `false`.

Tip: It is possible to mark a field as `@Manually` and then use it in a SQL query fragment that is supplied by the programmer; thus allowing to make more complex queries than initially supported by Uaithne.

@ValueWhenNull

This annotation allows to indicate to the generator to use the value supplied in this annotation in the query when the field (of an entity or of an operation) value on which it is applied is null.

Note: This annotation requires that the field over which it is applied supports null, so it must also have the annotation `@Optional`.

Example: For the next operation definition

```
@SelectOne(result = _Calendar.class)
class _SelectCalendarByTitle {
    @Optional
    @ValueWhenNull("'default calendar'")
    String title;
}
```

The next query is generated for PostgreSQL:

```
select
    id,
    title,
    description
from
    Calendar
where
    title = <if test='title != null'> #{title,jdbcType=VARCHAR} </if> <if test='title
== null'> 'default calendar' </if>
```

Tip: the value supplied in the annotation `@ValueWhenNull` represents an SQL fragment; so, it is possible, for example, that the SQL fragment is the call to an SQL function that returns the value to

finally be used.

@ForceValue

This annotation tells to the generator to use the value supplied in this annotation regardless of the value of the field (of an entity or operation).

Note: This annotation is similar to `@ValueWhenNull` only that the value will always be applied, regardless of whether the value of the field is null or not. It does not require the field to have the `@Optional` annotation.

Example: For the next operation definition

```
@SelectOne(result = _Calendar.class)
class _SelectCalendarByTitle {
    @ForceValue("'default calendar'")
    String title;
}
```

The next query is generated for PostgreSQL:

```
select
   id,
   title,
   description
from
   Calendar
where
   title = 'default calendar'
```

Tip: the value supplied in the @ForceValue annotation represents an SQL fragment; so, it is possible, for example, that the SQL fragment is the call to an SQL function that returns the value to be used finally.

@HasDefaultValueWhenInsert

This annotation indicates that the field of an entity (or entity view) has a default value in the database; so, when inserting the record, if the value of this field is null, the default value of the database is used.

Example: For the next entity definition

```
@Entity
class _Calendar {
    @Id
    Integer id;
    String title;
    @HasDefaultValueWhenInsert
    String description;
}
```

The next query is generated for the `@InsertEntity` operation in PostgreSQL:

```
insert into
   Calendar
(
   title,
   description
) values (
   #{title,jdbcType=VARCHAR},
   <if test='description != null'> #{description,jdbcType=VARCHAR} </if> <if test='description == null'>
default </if>
)
```

@Optional

This annotation tells Uaithne that the value of a field is optional; so, it admits null. When this annotation is used in the fields of an operation, it indicates that this field should be ignored when it is null. It happens when the field conditions the result of the operation, that is, when this field is part of the `where` or the `order by` of the query.

Example: For the next operation definition

```
@SelectMany(result = _Event.class)
class _SelectEventStaringOnDate {
    Date start;
    @Optional
    Integer calendarId;
}
```

The next query is generated for PostgreSQL:

```
select
    id,
    title,
    start,
    end,
    description,
    calendarId
from
    Event
<where>
    start = #{start,jdbcType=TIMESTAMP}
    <if test='calendarId != null'>and calendarId = #{calendarId,jdbcType=INTEGER}</if>
</where>
```

@Comparator

This annotation allows you to specify to Uaithne how to compare the value of the field of an operation with the value of the column in the database when the field is used as part of the `where` clause of a query. By default, if this annotation is not provided, the comparison is made by equality, but if the data type of the field is a list, check that the value of the column is in the list.

Example: For the next operation definition

```
@SelectMany(result = _Event.class)
class _SelectEventStaringAfterDate {
    @Comparator(Comparators.LARGER_AS)
    Date start;
    Integer calendarId;
}
```

The next query is generated for PostgreSQL:

```
select
    id,
    title,
    start,
    end,
    description,
    calendarId
from
    Event
where
    start &gt;= #{start,jdbcType=TIMESTAMP}
    and calendarId = #{calendarId,jdbcType=INTEGER}
```

Possible values:

- **`EQUAL`**: The value of the column must be equal to the value of the field. SQL:

  ```
  [[column]] = [[value]]
  ```

- **`NOT_EQUAL`**: The value of the column must not be equal to the value of the field. SQL:

  ```
  [[column]] <> [[value]]
  ```

- **`EQUAL_INSENSITIVE`**: The value of the column must be equal to the value of the field ignoring the case sensitivity. SQL:

  ```
  lower([[column]]) = lower([[value]])
  ```

- **`NOT_EQUAL_INSENSITIVE`**: The value of the column must not be equal to the value of the field ignoring the case sensitivity. SQL:

  ```
  lower([[column]]) <> lower([[value]])
  ```

- **`SMALLER`**: The value of the column must be less than the value of the field. SQL:

  ```
  [[column]] < [[value]]
  ```

- **`LARGER`**: The value of the column must be greater than the value of the field. SQL:

  ```
  [[column]] > [[value]]
  ```

- **`SMALL_AS`**: The value of the column must be less than or equal to the value of the field. SQL:

```
[[column]] <= [[value]]
```

- **LARGER_AS**: The value of the column must be greater than or equal to the value of the field. SQL:

```
[[column]] >= [[value]]
```

- **IN**: The value of the column must be contained in the list provided as the value of the field. SQL:

```
[[column]] in ([[value1]],[[value2]],...)
```

- **NOT_IN**: The value of the column must not be contained in the list provided as the value of the field. SQL:

```
[[column]] not in ([[value1]],[[value2]],...)
```

- **LIKE**: The value of the column must be similar to the given pattern as the value of the field. SQL:

```
[[column]] like [[value]]
```

- **NOT_LIKE**: The value of the column should not be similar to the given pattern as the value of the field. SQL:

```
[[column]] not like [[value]]
```

- **LIKE_INSENSITIVE**: The value of the column must be similar to the given pattern as the value of the field ignoring the case sensitivity. SQL:

```
lower([[column]]) like lower([[value]])
```

- **NOT_LIKE_INSENSITIVE**: The value of the column should not be similar to the given pattern as a value of the field ignoring the case sensitivity. SQL:

```
lower([[column]]) not like lower([[value]])
```

- **START_WITH**: The value of the column must start with the given pattern as the value of the field. SQL:

```
[[column]] like ([[value]] || '%')
```

- **NOT_START_WITH**: The value of the column must not start with the given pattern as the value of the field. SQL:

```
[[column]] not like ([[value]] || '%')
```

- **END_WITH**: The value of the column must end with the given pattern as the value of the field. SQL:

```
[[column]] like ('%' || [[value]])
```

- **NOT_END_WITH**: The value of the column must not end with the given pattern as the value of the field. SQL:

```
[[column]] not like ('%' || [[value]])
```

- **START_WITH_INSENSITIVE**: The value of the column must start with the given pattern as the value of the field ignoring case sensitivity. SQL:

```
lower([[column]]) like (lower([[value]]) || '%')
```

- **NOT_START_WITH_INSENSITIVE**: The value of the column should not start with the given pattern as the value of the field ignoring case sensitivity. SQL:

lower(**[[column]]**) not like (lower(**[[value]]**) || '%')

- **END_WITH_INSENSITIVE**: The value of the column must end with the given pattern as the value of the field ignoring case sensitivity. SQL:

```
lower([[column]]) like ('%' || lower([[value]]))
```

- **NOT_END_WITH_INSENSITIVE**: The value of the column must not end with the given pattern as the value of the field ignoring case sensitivity. SQL:

```
lower([[column]]) not like ('%' || lower([[value]]))
```

- **CONTAINS**: The value of the column must contain the given pattern as the value of the field. SQL:

```
[[column]] like ('%' || [[value]] || '%')
```

- **NOT_CONTAINS**: The value of the column must not contain the given pattern as the value of the field. SQL:

```
[[column]] not like ('%' || [[value]] || '%')
```

- **CONTAINS_INSENSITIVE**: The value of the column must contain the given pattern as the value of the field ignoring case sensitivity. SQL:

```
lower([[column]]) like ('%' || lower([[value]]) || '%')
```

- **NOT_CONTAINS_INSENSITIVE**: The value of the column must not contain the given pattern as the value of the field ignoring case sensitivity. SQL:

```
lower([[column]]) not like ('%' || lower([[value]]) || '%')
```

Note: In Sql Server || is replaced by + as a concatenation operator, and in PostgreSql `ilike` is used instead of the combination of `lower` and `like`.

@CustomComparator

This annotation allows you to specify to Uaithne how you should compare the value of the field of an operation with the value of the column in the database when the field is used as part of the `where` clause of the query. The difference with `@Comparator` is that `@CustomComparator` allows you to specify the comparison rule instead of using a predefined one.

The comparison pattern corresponds to an SQL fragment with some embedded sequences that allow to include certain information of the column or the field in the final SQL code; this sequence

must be contained between `[[` and `]]`, for example, `[[value]]` is replaced by the value of the field in the generated SQL.

Example: For the next operation definition

```
@SelectMany(result = _Event.class)
class _SelectMomentEvents {
    Integer calendarId;
    @CustomComparator("start >= [[value]] and end <= [[value]]")
    Date moment;
}
```

The next query is generated for PostgreSQL:

```
select
    id,
    title,
    start,
    end,
    description,
    calendarId
from
    Event
where
    calendarId = #{calendarId,jdbcType=INTEGER}
    and start &gt;= #{moment,jdbcType=TIMESTAMP} and end &lt;=
#{moment,jdbcType=TIMESTAMP}
```

Possible embedded sequences:

- **column**
- COLUMN

  Name of the column in the database that has been specified with `@MappedName`; if it does not have, the name of the field in the Java object is used.

- **name**
- NAME

  Name of the field in the Java object.

- **value**
- VALUE

  Value of the field in the Java object.

  This sequence takes the value of the annotation `@ForceValue`; if the field does not have this annotation, it takes the value of the annotation `@ValueWhenNull`; and, if it does not have the last one, it takes the value of the field. It is equivalent in the latter case to:

  ```
  #{[[name]][[jdbcType]][[typeHandler]]}
  ```

- **jdbcType**
- JDBC_TYPE

  JDBC data type of the field.

  The value of this sequence always starts with `,jdbcType=` followed by the name of the JDBC type of the field that is read from the annotation `@JdbcType`; if the field have no one, the best correspondence is selected for the data type of the field.

- **typeHandler**
- TYPE_HANDLER

  MyBatis TypeHandler, if one was specified for the field using the `@MyBatisTypeHandler` annotation

  The value of this sequence always starts with `,typeHandler=` followed by the full name of the TypeHandler class specified in the field with the annotation `@MyBatisTypeHandler`. If the field don't have this annotation, an empty string is used.

- **valueNullable**
- VALUE_NULLABE

  Value of the field in the Java object ignoring the `@ValueWhenNull` annotation

- **valueWhenNull**
- VALUE_WHEN_NULL

  Value of the `@ValueWhenNull` annotation

- **unforcedValue**
- UNFORCED_VALUE

  Value of the field in the Java object ignoring the `@ForceValue` annotation

- **unforcedNullbaleValue**
- UNFORCED_NULLABLE_VALUE

  Value of the field in the Java object ignoring the `@ForceValue` and `@ValueWhenNull` annotations

- **forcedValue**
- FORCED_VALUE

  Value of the `@ForceValue` annotation

@OrderBy

This annotation marks one or more fields of an operation as part of the order by clause of a select. It can be used in the `@SelectOne`, `@SelectMany`, and `@SelectPage` operations.

A field marked with the annotation `@OrderBy` must be of type string and its value must be the content of the order by clause of the query, but with the difference that the name of the fields of the resulting entity should be used instead of the name of the columns in the database; Uaithne automatically translates the `order by` before executing it in the database to its corresponding

SQL, preventing in the process any risk of SQL Injections and at the same time hiding the details of the database to the upper layers.

Example: For the the next entity and operation definition

```
@Entity
@MappedName("table_event")
class _Event {
    @Id
    @MappedName("column_id")
    Integer id;
    @MappedName("column_title")
    String title;
    @MappedName("column_start")
    Date start;
    @MappedName("column_end")
    Date end;
    @Optional
    @MappedName("column_description")
    String description;
    @MappedName("column_calendar_id")
    Integer calendarId;
}

@SelectMany(result = _Event.class)
class _SelectAllEvents {
    Integer calendarId;
    @OrderBy
    String orderResultBy;
}
```

The next query is generated for PostgreSQL:

```
select
    column_id as "id",
    column_title as "title",
    column_start as "start",
    column_end as "end",
    column_description as "description",
    column_calendar_id as "calendarId"
from
    table_event
where
    column_calendar_id = #{calendarId,jdbcType=INTEGER}
order by ${orderResultBy}
```

The `orderResultBy` property of the operation can receive as a comma-separated string with the list of the name of the fields of the resulting `Event` entity, and each of them can have the `asc` or `desc` modifier; this string is is case-insensitive. The field value is translated into its corresponding SQL version before MyBatis is called, so `${orderResultBy}` will contain the SQL version of the SQL clause supplied to the operation.

Example of valid values for the `orderResultBy` field:

- `title` which is translated to SQL as `column_title`
- `title desc` which is translated to SQL as `column_title desc`
- `title asc` which is translated to SQL as `column_title asc`
- `start desc, title` which is translated to SQL as `column_start desc, column_title`

Customizing the generation of the record identifier

Uaithne supports multiple strategies for generating the identifier of a record in the database; starting from a base strategy specified in the Uaithne configuration, that can be to use self-generated fields or to use sequences with predefined name pattern. With the help of several annotations, you can control, case by case, how to handle the generation of the register identifier.

Limitation: Uaithne only supports a single identifier field per entity in order to automatically generate the queries and operations that receive or return the identifier. To avoid this limitation, it is possible to use several strategies:

- In **tables with** composite keys, but where a field that makes up the primary key in the database can act as an **alternate key**: in these cases it is sufficient to use the alternate key as an identifier in Uaithne, thus having a unique identifier for practical purposes. This strategy is useful when, by design, you want to use composite keys in the database but the table has its own unique identifier.
- In the tables with composite keys, **where it is not possible to apply the previous strategy**, you should avoid using operations that use the registry identifier automatically; that is, you can not use the operations that make Insert, Update, Delete, Save, Merge of entities; instead, you have to use the Insert, Update and Delete operations not restricted to entities; that is, those that are built with the annotations `@Insert`, `@Update` and `@Delete`.

Annotations that allow controlling the generation of the identifier:

- **`@Id`**: This annotation indicates which one is the identifier field. If the autogenerated parameter is set to `false`, it indicates that this identifier is not self-generated, therefore its value must be supplied when the record is inserted.
- **`@IdSequenceName`**: This annotation indicates to the generator that the sequence name specified ad value of this annotation must be used for the generation of the identifier.
- **`@IdQueries`**: This annotation indicates that the generated queries must use the provided queries to generate the identifier.

Record identifier generation strategies supported by Uaithne:

- **Manually**: In which the identifier must be provided manually before inserting the record in the database as a value of the field in the entity. To activate this mode, the `autogenerated` property of the `@Id` annotation must be set to `false`.
- **Auto-incremental columns in the database**: In which the identifier is automatically generated by the data type of the column in the database. This is the default mode in the database with support for auto-incremental data types (it can be changed in the Uaithne configuration). Note: it is possible to emulate this behavior in the database that does not support it using triggers.
- **Secuencias en la base de datos**: En el cual el identificador es generado mediante el uso explícito de una secuencia, por lo que al hacer un insert se debe incluir en la query la invocación a la secuencia para que retorne el siguiente valor. Para activar este modo hay

que añadir la anotación `@IdSecuenceName` al campo especificándole el nombre de la secuencia. En las base de datos sin soporte a tipos de datos autoincrementales (se puede cambiar en la configuración de Uaithne) este el el modo por defecto, y se usa un patrón para la generación del nombre de la secuencia en aquellos campos sin la anotación `@IdSecuenceName` que se puede especificar en la configuración de Uaithne.

- **Sequences in the database**: In which the identifier is generated through the explicit use of a sequence, so when making an insert the query the next value of the sequence must be retrieved. To activate this mode, you must add the `@IdSecuenceName` annotation to the field specifying the name of the sequence. In the databases without support for auto-incremental data types (it can be changed in the Uaithne configuration) this is the default mode, and a pattern is used to generate the name of the sequence in those fields without the `@IdSecuenceName` annotation that can be specified in the Uaithne configuration.
- **Identificadores generados mediante queries**: En el cual el programador especifica manualmente, mediante el uso de la anotación `@IdQueries`, el fragmento sql necesario para generar el identificador del registro a ser insertado (especificándolo en la propiedad `selectNextValue`), y la query necesaria para recuperar el identificador del registro insertado (especificándola en la propiedad `selectCurrentValue`).
- **Identifiers generated by queries**: In which the programmer specifies manually, through the use of the `@IdQueries` annotation. In this annotation, you must provide the fragment sql necessary to generate the identifier of the record to be inserted (specifying it in the property `selectNextValue`), and the necessary query to recover the identifier of the inserted record (specifying it in the `selectCurrentValue` property).

@Id

This annotation allows you to specify to Uaithne that the field of an entity (or entity view) to which it applies is the identifier of the record.

Parameters:

- **autogenerated**: Indicates if the field is autogenerated. If it is set to `false` (by default its value is `true`) it indicates to Uaithne that the identifier of the record will be the value that the field has when it is going to be inserted, so the value must be included in the insert query and it must be supplied before the insert is executed.

@IdSecuenceName

This annotation allows to specify to Uaithne to use the provided sequence name as a parameter to generate the identifier of the record.

Note:

This annotation can be placed on the field or on the entity. By placing it on the entity, it modifies the behavior of the field annotated with `@Id` even if it is in a base class.

@IdQueries

This annotation allows Uaithne to specify the SQL code to generate the next identifier value at the time of insertion and the SQL query to retrieve the identifier of the inserted record.

Parameters:

- **selectNextValue**: Indicates the SQL fragment to be embedded in the insert query to generate the next identifier value of the record to be inserted.
- **selectCurrentValue**: Indicates the SQL query that must be executed to recover the value of the newly inserted record.

Note:

This annotation can be placed on the field or on the entity. By placing it on the entity, it modifies the behavior of the field annotated with `@Id` even if it is in a base class.

Customizing the MyBatis code

Uaithne uses MyBatis to access the database, and wherever the generator allows specifying a fragment or SQL query, it is possible to embed MyBatis instructions (not allowed in the experimental generation of stored procedures).

Names of the columns:

When doing a `select` query, the transformation of the result in Java objects is done by the name of the column, so it is necessary that the name of the column in the result of the query is the same as the name of the property in the object that receives the value; if the names do not match, it is necessary to include the `as` clause to rename the field in the query.

Special characters:

- **`>`**: Uaithne automatically transforms the character > to `>` so that the queries written as Uaithne entries are valid in the MyBatis XML.
- **`<`**: Uaithne automatically transforms the character < to `<` so that the queries written as Uaithne entries are valid in the MyBatis XML.
- **`{[`**: If you want to write the character < as part of an XML fragment, you must replace it with `{[`. Example: To write the XML tag `<set>` in the XML generated from MyBatis it is necessary to write `{[set]}` in the input of Uaithne.
- **`]}`**: If you want to write the character > as part of an XML fragment, you must replace it with `]}`. Example: To write the XML tag `</set>` in the XML generated from MyBatis it is necessary to write `{[/set]}` in the input of Uaithne.

Annotations that allow to control the MyBatis code:

- **@JdbcType**: Allows you to specify the JDBC data type of the field.
- **@MyBatisTypeHandler**: It allows specifying for a field which is the TypeHandler in charge of transforming the value of the java object into the value that the database receives. Limitation: This annotation does not affect the process of converting the result of the database into value to receive the Java object.
- **@MyBatisCustomSqlStatementId**: It allows specifying the `statementId` to be executed by the operation that receives the annotation. Using this annotation causes Uaithne not to generate the query in the MyBatis XML and causes the Java code to invoke the supplied `statementId` (instead of the one that would be generated if you don't use this annotation).

@JdbcType

It allows to specify to Uaithne the JDBC data type that should be used for the field on which the annotation is applied in the MyBatis XML; if it is no specified, one is assigned automatically depending on the type (it can be changed in the configuration of Uaithne), even when it is the type itself, or when it is a generic argument of the `List` or `ArrayList` types or array type:

- **Varchar**: For the type `java.lang.String`
- **Timestamp**: For the types `java.util.Date`, `java.sql.Timestamp`
- **Date**: For the type `java.sql.Date`
- **Time**: For the type `java.sql.Time`
- **Boolean**: For the types `boolean`, `java.lang.Boolean`
- **Tinyint**: For the types `byte`, `java.lang.Byte`
- **Smallint**: For the types `short`, `java.lang.Short`
- **Integer**: For the types `int`, `java.lang.Integer`
- **Bigint**: For the types `long`, `java.lang.Long`
- **Real**: For the types `float`, `java.lang.Float`
- **Double**: For the types `double`, `java.lang.Double`
- **Numeric**: For the types `java.math.BigDecimal`, `java.math.BigInteger`
- **Char**: For the types `char`, `java.lang.Character`

@MyBatisTypeHandler

Allows to specify to Uaithne to use for the field over which the annotation is applied the TypeHandler specified as a parameter in the MyBatis XML.

Limitation:

This annotation only affects the conversion of the value from Java to Sql but not the opposite, if you want it to affect also in the opposite direction you must specify a global TypeHandler in the configuration of MyBatis.

@MyBatisCustomSqlStatementId

It allows to specify to Uaithne that for the operation on which this annotation is applied, the supplied MyBatis `statementId` must be used; this causes that an entry in the MyBatis XML with the query is not generated for this operation; instead of it, the supplied `statementId` is used. The implementation of that `statementId` must be included in the configuration of MyBatis manually adding another XML file that contains it.

Parameters:

- **`value`**: `StatementId` to be used by the operation.
- **`countSatementId`**: For the `@SelectPage` operations, this parameter allows specifying the `statementId` that returns the number of existing records; for this operation, the `value` parameter must have the `statementId` that returns the content of the selected page.
- **`saveInsertStatementId`**: For the `@Save` operations, this parameter allows specifying the `statementId` that inserts the record; for this operation, the `value` parameter must have the `statementId` that updates the record.

Accessing fields from SQL fragments

Uaithne allows access to the value of any field in any fragment or SQL query, thus increasing the flexibility of the generated query.

The access pattern to a field must be a sequence of characters contained between `{{` and `}}`; for example, `{{field}}` is replaced by the value of the field in the generated SQL. This sequence of characters can be composed of up to three parts (at least one must have one), each part must be separated by the character `:` with format as follows:

> {{*field*}}
>
> {{*field*:*rule*}}
>
> {{*field*:*rule*:*arg*}}

Where:

- **`field`**: corresponds to the name of the field at the input to Uaithne.
- **`rule`**: indicates the rule to be used to generate the SQL code reference to the field. If no rule is specified the `value` rule is assumed.
- **`arg`**: allows to specify for some rules an argument to be used by it to generate the SQL code to access the field.

Example:

If you want to access the value of the `title` field, you need to write `{{title}}` in the SQL that receives Uaithne; that sequence is transformed to `#{title,jdbcType=VARCHAR}` in the MyBatis XML. You can also write `{{title:value}}` to get the same result.

Basic access rules:

- **`condition`**
- `CONDITION`

  Generates the comparison rule of the field as defined in the operation, that is, it uses comparison by equality or `IN`, but if the field contains the annotation `@Comparator` or `@CustomComparator`, the specified rule is generated.

- **`custom`**
- `CUSTOM`

  This rule allows access to the same behavior as the annotation `@CustomComparator`; it receives the comparison pattern as an argument. Example:

  ```
  {{moment:custom:start >= [[value]] and end <= [[value]]}}
  ```

  generates the code:

  ```
  start &gt;= #{moment,jdbcType=TIMESTAMP} and end &lt;=
  #{moment,jdbcType=TIMESTAMP}
  ```

- **ifNull**
- IF_NULL

  This rule generates the opening of an `if` that verifies if the value is null. Example:

  ```
  {{field:ifNull}}
  ```

  generates the code:

  ```
  <if test='field == null'>
  ```

- **ifNotNull**
- IF_NOT_NULL

  This rule generates the opening of an `if` that verifies if the value is not null. Example:

  ```
  {{field:ifNotNull}}
  ```

  generates the code:

  ```
  <if test='field != null'>
  ```

- **endIf**
- END_IF

  This rule generates the closing of an `if`. Example:

  ```
  {{field:endIf}}
  ```

  generates the code:

  ```
  </if>
  ```

Rules for access to embedded comparison sequences:

You can also specify as an access rule any of the embedded sequences available in the `@CustomComparator` annotation without using the `[[` or `]]` characters. These are:

- **column**
- COLUMN

- **name**
- NAME

- **value**

- VALUE

- **jdbcType**
- JDBC_TYPE

- **typeHandler**
- TYPE_HANDLER

- **valueNullable**

- VALUE_NULLABE
- **valueWhenNull**
- VALUE_WHEN_NULL

- **unforcedValue**
- UNFORCED_VALUE

- **unforcedNullbaleValue**
- UNFORCED_NULLABLE_VALUE

- **forcedValue**
- FORCED_VALUE

Rules of access to predefined comparators:

You can also specify as an access rule any comparator allowed in the `@Comparator` annotation or some of its aliases, generating the comparison rule for the field with the column in the database.

- `=`
- `==`
- `equal`
- **`EQUAL`**

  Equivalent to `EQUAL` comparator of the annotation `@Comparator`.

- `!=`
- `<>`
- `notEqual`
- **`NOT_EQUAL`**

  Equivalent to `NOT_EQUAL` comparator of the annotation `@Comparator`.

- `i=`
- `i==`
- `I=`
- `I==`
- `iequal`
- `IEQUAL`
- `equalInsensitive`
- **`EQUAL_INSENSITIVE`**

  Equivalent to `EQUAL_INSENSITIVE` comparator of the annotation `@Comparator`.

- `i!=`
- `i<>`
- `I!=`
- `I<>`
- `inotEqual`
- `INOT_EQUAL`
- `notEqualInsensitive`
- **`NOT_EQUAL_INSENSITIVE`**

Equivalent to `NOT_EQUAL_INSENSITIVE` comparator of the annotation `@Comparator`.

- `<`
- `smaller`
- **`SMALLER`**

Equivalent to `SMALLER` comparator of the annotation `@Comparator`.

- `>`
- `larger`
- **`LARGER`**

Equivalent to `LARGER` comparator of the annotation `@Comparator`.

- `<=`
- `smallAs`
- **`SMALL_AS`**

Equivalent to `SMALL_AS` comparator of the annotation `@Comparator`.

- `>=`
- `largerAs`
- **`LARGER_AS`**

Equivalent to `LARGER_AS` comparator of the annotation `@Comparator`.

- `in`
- **`IN`**

Equivalent to `IN` comparator of the annotation `@Comparator`.

- `notIn`
- **`NOT_IN`**

Equivalent to `NOT_IN` comparator of the annotation `@Comparator`.

- `like`
- **`LIKE`**

Equivalent to `LIKE` comparator of the annotation `@Comparator`.

- `notLike`
- **`NOT_LIKE`**

Equivalent to `NOT_LIKE` comparator of the annotation `@Comparator`.

- `ilike`
- `ILIKE`
- `likeInsensitive`
- **`LIKE_INSENSITIVE`**

Equivalent to `LIKE_INSENSITIVE` comparator of the annotation `@Comparator`.

- `notIlike`

- NOT_ILIKE
- notLikeInsensitive
- **NOT_LIKE_INSENSITIVE**

  Equivalent to NOT_LIKE_INSENSITIVE comparator of the annotation @Comparator.

- startWith
- **START_WITH**

  Equivalent to START_WITH comparator of the annotation @Comparator.

- notStartWith
- **NOT_START_WITH**

  Equivalent to NOT_START_WITH comparator of the annotation @Comparator.

- endWith
- **END_WITH**

  Equivalent to END_WITH comparator of the annotation @Comparator.

- notEndWith
- NOT_END_WITH

  Equivalent to NOT_END_WITH comparator of the annotation @Comparator.

- istartWith
- ISTART_WITH
- startWithInsensitive
- **START_WITH_INSENSITIVE**

  Equivalent to START_WITH_INSENSITIVE comparator of the annotation @Comparator.

- notIstartWith
- NOT_ISTART_WITH
- notStartWithInsensitive
- **NOT_START_WITH_INSENSITIVE**

  Equivalent to NOT_START_WITH_INSENSITIVE comparator of the annotation @Comparator.

- iendWith
- IEND_WITH
- endWithInsensitive
- **END_WITH_INSENSITIVE**

  Equivalent to END_WITH_INSENSITIVE comparator of the annotation @Comparator.

- notIendWith
- NOT_IEND_WITH
- notEndWithInsensitive
- **NOT_END_WITH_INSENSITIVE**

Equivalent to `NOT_END_WITH_INSENSITIVE` comparator of the annotation `@Comparator`.

- `contains`
- **`CONTAINS`**

  Equivalent to `CONTAINS` comparator of the annotation `@Comparator`.

- `notContains`
- **`NOT_CONTAINS`**

  Equivalent to `NOT_CONTAINS` comparator of the annotation `@Comparator`.

- `icontains`
- `ICONTAINS`
- `containsInsensitive`
- **`CONTAINS_INSENSITIVE`**

  Equivalent to `CONTAINS_INSENSITIVE` comparator of the annotation `@Comparator`.

- `notIcontains`
- `NOT_ICONTAINS`
- `notContainsInsensitive`
- **`NOT_CONTAINS_INSENSITIVE`**

  Equivalent to `NOT_CONTAINS_INSENSITIVE` comparator of the annotation `@Comparator`.

Customizing the clauses of the queries with @CustomSqlQuery

By using the `@CustomSqlQuery` annotation Uaithne allows to partially or totally modify the generated query of the operation on which it is applied, or to add SQL fragments in the desired clause. The name of the annotation parameter corresponds to the name of the clause to be replaced, but there is the possibility of adding code before or after the clause by using the parameters whose name starts with `before` and `after` followed by the name of the clause.

The supported clauses are:

- **query**: which replaces all the generated query by the one supplied.
- **select**: which replaces the content of the `select` clause of a select query.
- **from**: which replaces the from clause of the query, either in a select, insert, update, or delete query. This point refers to the name of the table on which the action is performed.
- **where**: which replaces the content (or adds one if it does not have) the content of the `where` clause of a select, update or delete query.
- **groupBy**: add the `group by` clause of a select query. In this SQL fragment you can include the `having` subclause if the `group by` requires one.
- **orderBy**: that replaces the content (or adds one if it does not have) the `order by` clause of a select query.
- **insertInto**: which replaces the content of the `into` clause of an insert query. This point refers to the elements that are placed inside the first pair of parentheses when you write `insert into table (...) values (...)`
- **insertValues**: which replaces the contents of the `values` clause of an insert query. This point refers to the elements that are placed inside the second pair of parentheses when you

write `insert into table (...) values (...)`
- **updateSet**: which replaces the content of the `set` clause of an update query. This point refers to the list of fields to be updated and their respective values

Other parameters of `@CustomSqlQuery`:

- **tableAlias**: it allows to add an alias to the table in the query; which causes that you add the alias in the from and in every place where the name of the column of a field is prefixed with the alias followed by a period.
- **excludeEntityFields**: allows you to specify a list with the name of the fields that you want to be ignored during the generation of the query.
- **isProcedureInvocation**: allows specifying if it is an invocation to a stored procedure, so it must be marked as `CALLABLE` in the MyBatis XML. If it is not given any value the type is automatically detected by the operation type function (`TRUE` for the special stored procedure operations, `FALSE` for the rest).

To show how each parameter alters the result of the generated query, the following queries will be used (the JDBC data type is omitted for easy reading):

```
select id, title from calendar where id = #{id} order by ${orderBy}

insert into calendar(title, description) values (#{title}, #{description})

update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

In the modified query it will be placed between `[` and `]` in the content of the query that is replaced by the value of the parameter, and if `[...]` is used it is because the value is added at that point. The changes made in the query are highlighted in red. The queries crossed out is because it does not apply.

query

Replace the entire query with the supplied value.

```
[select id, title from calendar where id = #{id} order by ${orderBy}]

[insert into calendar(title, description) values (#{title}, #{description})]

[update calendar set title = #{title} where id = #{id}]

[delete from calendar where id = #{id}]
```

beforeQuery

Add the supplied value to the beginning of the query.

```
[...] select id, title from calendar where id = #{id} order by ${orderBy}

[...] insert into calendar(title, description) values (#{title}, #{description})
```

```
[...] update calendar set title = #{title} where id = #{id}

[...] delete from calendar where id = #{id}
```

afterQuery

Add the supplied value to the end of the query.

```
select id, title from calendar where id = #{id} order by ${orderBy} [...]

insert into calendar(title, description) values (#{title}, #{description}) [...]

update calendar set title = #{title} where id = #{id} [...]

delete from calendar where id = #{id} [...]
```

select

Replaces the select of the query with the supplied value.

```
select [id, title] from calendar where id = #{id} order by ${orderBy}

~~insert into calendar(title, description) values (#{title}, #{description})~~

~~update calendar set title = #{title} where id = #{id}~~

~~delete from calendar where id = #{id}~~
```

beforeSelectExpression

Add the supplied value at the beginning of the select.

```
select [...] id, title from calendar where id = #{id} order by ${orderBy}

~~insert into calendar(title, description) values (#{title}, #{description})~~

~~update calendar set title = #{title} where id = #{id}~~

~~delete from calendar where id = #{id}~~
```

afterSelectExpression

Add the supplied value to the end of the select.

```
select id, title [...] from calendar where id = #{id} order by ${orderBy}

~~insert into calendar(title, description) values (#{title}, #{description})~~

~~update calendar set title = #{title} where id = #{id}~~

~~delete from calendar where id = #{id}~~
```

from

Replace the from of the query with the supplied value.

```
select id, title from [calendar] where id = #{id} order by ${orderBy}

insert into [calendar](title, description) values (#{title}, #{description})

update [calendar] set title = #{title} where id = #{id}

delete from [calendar] where id = #{id}
```

beforeFromExpression

Add the value supplied to the beginning of the from.

```
select id, title from [...] calendar where id = #{id} order by ${orderBy}

insert into [...] calendar(title, description) values (#{title}, #{description})

update [...] calendar set title = #{title} where id = #{id}

delete from [...] calendar where id = #{id}
```

afterFromExpression

Add the value supplied to the end of the from.

```
select id, title from calendar [...] where id = #{id} order by ${orderBy}

insert into calendar [...](title, description) values (#{title}, #{description})

update calendar [...] set title = #{title} where id = #{id}

delete from calendar [...] where id = #{id}
```

where

Replace the where of the query with the supplied value.

```
select id, title from calendar where [id = #{id}] order by ${orderBy}

~~insert into calendar(title, description) values (#{title}, #{description})~~

update calendar set title = #{title} where [id = #{id}]

delete from calendar where [id = #{id}]
```

beforeWhereExpression

Add the supplied value to the beginning of the where.

```
select id, title from calendar where [...] id = #{id} order by ${orderBy}
```

~~`insert into calendar(title, description) values (#{title}, #{description})`~~

```
update calendar set title = #{title} where [...] id = #{id}
```

```
delete from calendar where [...] id = #{id}
```

afterWhereExpression

Add the value supplied to the end of the where.

```
select id, title from calendar where id = #{id} [...] order by ${orderBy}
```

~~`insert into calendar(title, description) values (#{title}, #{description})`~~

```
update calendar set title = #{title} where id = #{id} [...]
```

```
delete from calendar where id = #{id} [...]
```

groupBy

Add the value supplied as group by to the query.

```
select id, title from calendar where id = #{id} group by [...] order by
${orderBy}
```

~~`insert into calendar(title, description) values (#{title}, #{description})`~~

~~`update calendar set title = #{title} where id = #{id}`~~

~~`delete from calendar where id = #{id}`~~

beforeGroupByExpression

Add the value supplied to the start of the group by, if you do not have a group by add one.

```
select id, title from calendar where id = #{id} group by [...] order by
${orderBy}
```

~~`insert into calendar(title, description) values (#{title}, #{description})`~~

~~`update calendar set title = #{title} where id = #{id}`~~

~~`delete from calendar where id = #{id}`~~

afterGroupByExpression

Add the value supplied at the end of the group by, if you do not have a group by add one.

```
select id, title from calendar where id = #{id} group by [...] order by
${orderBy}
```

~~`insert into calendar(title, description) values (#{title}, #{description})`~~

```
update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

orderBy

Replace the order by of the query with the supplied value, if you do not have an order by add one.

```
select id, title from calendar where id = #{id} order by [${orderBy}]

insert into calendar(title, description) values (#{title}, #{description})

update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

beforeOrderByExpression

Add the value supplied at the beginning of order by, if you do not have an order by add one.

```
select id, title from calendar where id = #{id} order by [...] ${orderBy}

insert into calendar(title, description) values (#{title}, #{description})

update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

afterOrderByExpression

Add the value supplied at the end of the order by, if you do not have an order by add one.

```
select id, title from calendar where id = #{id} order by ${orderBy} [...]

insert into calendar(title, description) values (#{title}, #{description})

update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

insertInto

Replace the into of the insert query with the supplied value.

```
select id, title from calendar where id = #{id} order by ${orderBy}

insert into calendar([title, description]) values (#{title}, #{description})

update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

beforeInsertIntoExpression

Add the supplied value at the beginning of the into in the insert.

```
select id, title from calendar where id = #{id} order by ${orderBy}
```

```
insert into calendar([...] title, description) values (#{title}, #{description})
update calendar set title = #{title} where id = #{id}
delete from calendar where id = #{id}
```

afterInsertIntoExpression

Add the value supplied at the end of the into in the insert.

```
select id, title from calendar where id = #{id} order by ${orderBy}
insert into calendar(title, description [...]) values (#{title}, #{description})
update calendar set title = #{title} where id = #{id}
delete from calendar where id = #{id}
```

insertValues

Replace the values of the insert query with the supplied value.

```
select id, title from calendar where id = #{id} order by ${orderBy}
insert into calendar(title, description) values ([#{title}, #{description}])
update calendar set title = #{title} where id = #{id}
delete from calendar where id = #{id}
```

beforeInsertValuesExpression

Add the value supplied at the beginning of the values in the insert.

```
select id, title from calendar where id = #{id} order by ${orderBy}
insert into calendar(title, description) values ([...] #{title}, #{description})
update calendar set title = #{title} where id = #{id}
delete from calendar where id = #{id}
```

afterInsertValuesExpression

Add the value supplied to the end of the values in the insert.

```
select id, title from calendar where id = #{id} order by ${orderBy}
insert into calendar(title, description) values (#{title}, #{description} [...])
update calendar set title = #{title} where id = #{id}
delete from calendar where id = #{id}
```

updateSet

Replaces the set of the update query with the supplied value.

```
select id, title from calendar where id = #{id} order by ${orderBy}

insert into calendar(title, description) values (#{title}, #{description})

update calendar set [title = #{title}] where id = #{id}

delete from calendar where id = #{id}
```

beforeUpdateSetExpression

Add the supplied value to the start of the set in the update.

```
select id, title from calendar where id = #{id} order by ${orderBy}

insert into calendar(title, description) values (#{title}, #{description})

update calendar set [...] title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

afterUpdateSetExpression

Add the supplied value to the end of the set in the update.

```
select id, title from calendar where id = #{id} order by ${orderBy}

insert into calendar(title, description) values (#{title}, #{description})

update calendar set title = #{title} [...] where id = #{id}

delete from calendar where id = #{id}
```

tableAlias

Add the supplied value as alias of the name of the table specified in the entity, and prefix access to the columns managed by the entity with this value followed by a period.

```
select [t.].id, [t].title from calendar [t] where [t].id = #{id} order by
${orderBy}

insert into calendar [t](title, description) values (#{title}, #{description})

update calendar [t] set title = #{title} where [t].id = #{id}

delete from calendar [t] where [t].id = #{id}
```

excludeEntityFields

It ignores during the generation of the query the fields of name coming from the entity supplied as a list to this annotation. If the title name field is ignored in the following example, the result would be

(it shows crossed out what would not be included in the query):

```
select id, title from calendar where id = #{id} order by ${orderBy}

insert into calendar(title, description) values (#{title}, #{description})

update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

isProcedureInvocation

It allows to indicate to the generator that the query is a call to a stored procedure and therefore the statementType in the MyBatis XML must be marked as `CALLABLE`. If its value is `TRUE` it indicates that it must be marked as `CALLABLE`, if it is `FALSE` it indicates that it should not be marked as `CALLABLE`, if no value is specified or `UNSPECIFIED` is specified indicates that it is detected automatically according to the type of operation (`TRUE` for operations special stored procedure, `FALSE` for the rest). **Note**: this parameter does not alter the query in any way.

```
select id, title from calendar where id = #{id} order by ${orderBy}

insert into calendar(title, description) values (#{title}, #{description})

update calendar set title = #{title} where id = #{id}

delete from calendar where id = #{id}
```

Accessing the context of the application from SQL

It is possible to specify in the Uaithne configuration the entity that represents the execution context of the application; and, access it from any fragment or SQL query that Uaithne or MyBatis receives.

The context of the application is an entity that contains values specific to the current execution, such as the user's login. This information is complementary to the operation but is not part of it and it is independent of the action that you are executing. The context is normally used to store information related to the use, which.

To access the context of the application you just need to write `_app.` followed by the name of the field in the context entity using the same syntax to access the value of a field in an entity or operation, that is, using `{{` and `}}` in Uaithne or `#{` and `}` in MyBatis.

Ejemplo: For the next definition of the entity that has been configured as the entity that represents the context

```
@Entity
public class _AppContext {
    String login;
}
```

To access the value of the login property in the context of execution of the application, you must write:

Using the Uaithne access syntax to fields:

```
{{_app.login}}
```

Using the MyBatis access syntax to fields:

```
#{_app.login,jdbcType=VARCHAR}
```

The use of the execution context of the application can be done from any SQL fragment or query, and even from any manually written MyBatis XML, and should always be read only, it is usually quite useful if it is used in the `@DefaultValueWhenNull` annotations or `@ForceValue`.

Invoking simple stored procedures

The invocation of simple stored procedures, which only have input parameters, but do not have output parameters are created using the operations already described above that best define the logic of the stored procedure and using the `@CustomSqlQuery` annotation to indicate the invocation query of the procedure stored or annotation `@MyBatisCustomSqlStatementId` to indicate the MyBatis `statementId` that contains the invocation query of the stored procedure.

Operations typically used to invoke stored procedures are:

- **`@Insert`**: It allows to invoke stored procedures that make changes in the database, and its main action may be, among other things, to insert one or more records in one or several tables.
- **`@Update`**: It allows invoking stored procedures that make changes in the database, and its main action may be, among other things, to update one or more records in one or several tables.
- **`@Delete`**: It allows invoking stored procedures that make changes in the database, and its main action may be, among other things, to eliminate one or more records in one or several tables.

Using the `@CustomSqlQuery` annotation:

- **`query`**: where the invocation query of the stored procedure is placed, which usually looks like (depending on the database manager, the syntax is different):
    - **Most database if `isProcedureInvocation` is marked as `TRUE`**:

      ```
      call nombreProcedimentoAlmacenado({{parametro1}}, {{parametro2}})
      ```

    - **Oracle**:

      ```
      call nombreProcedimentoAlmacenado({{parametro1}}, {{parametro2}})
      ```

    - **Sql Server**:

      ```
      exec nombreProcedimentoAlmacenado {{parametro1}}, {{parametro2}}
      ```

    - **Postgre SQL**:

      ```
      select nombreProcedimentoAlmacenado({{parametro1}}, {{parametro2}})
      ```

    - **Derby**:

```
call nombreProcedimentoAlmacenado({{parametro1}}, {{parametro2}})
```

- **MySql**:

```
call nombreProcedimentoAlmacenado({{parametro1}}, {{parametro2}})
```

- **`isProcedureInvocation`**: Optional. If set to `TRUE`, indicates that the generated query must be invoked using a JDBC CallableStatement. When generating the code with MyBatis if this property is set to `TRUE`, the `CALLABLE` type is placed in the generated query as a `statementType`. The default value is `FALSE` except for complex queries to access the database where `TRUE`.

@Insert

It allows invoking stored procedures that make changes in the database, its main action may be, among other things, inserting one or more records in one or several tables and it must be used in conjunction with the `@CustomSqlQuery` annotation where the query is indicated with the invocation of the stored procedure or the `@MyBatisCustomSqlStatementId` annotation to indicate the MyBatis `statementId` that contains the invocation query of the stored procedure.

Example: For the next operation definition

```
@Insert(related = _Event.class)
@CustomSqlQuery(
        isProcedureInvocation = Ternary.TRUE,
        query = "select InsertHolidays({{calendarId}})"
)
class _InsertHolidays {
    Integer calendarId;
}
```

The query is generated for PostgreSQL, indicating to MyBatis that `statementType` is of `CALLABLE` type:

```
select InsertHolidays(#{calendarId,jdbcType=INTEGER})
```

@Update

It allows invoking stored procedures that make changes in the database, its main action may be, among other things, updating one or more records in one or more tables and it must be used in conjunction with the `@CustomSqlQuery` annotation where the query is indicated with the invocation of the stored procedure or the `@MyBatisCustomSqlStatementId` annotation to indicate the MyBatis `statementId` that contains the invocation query of the stored procedure.

Example: For the next operation definition

```
@Update(related = _Event.class)
@CustomSqlQuery(
        isProcedureInvocation = Ternary.TRUE,
        query = "select UpdateHolidays({{calendarId}})"
)
class _UpdateHolidays {
    Integer calendarId;
}
```

The query is generated for PostgreSQL, indicating to MyBatis that `statementType` is of `CALLABLE` type:

```
select UpdateHolidays(#{calendarId,jdbcType=INTEGER})
```

@Delete

It allows to invoke stored procedures that make changes in the database, its main action being, among other things, to eliminate one or several records in one or more tables and it must be used in conjunction with the annotation `@CustomSqlQuery` where the query is indicated with the invocation of the stored procedure or the `@MyBatisCustomSqlStatementId` annotation to indicate the MyBatis `statementId` that contains the invocation query of the stored procedure.

Example: For the next operation definition

```
@Delete(related = _Event.class)
@CustomSqlQuery(
        isProcedureInvocation = Ternary.TRUE,
        query = "select DeleteHolidays({{calendarId}})"
)
class _DeleteHolidays {
    Integer calendarId;
}
```

The query is generated for PostgreSQL, indicating to MyBatis that `statementType` is of `CALLABLE` type:

```
select DeleteHolidays(#{calendarId,jdbcType=INTEGER})
```

Complex stored procedures invocation

Uaithne offers four types of special operations to invoke stored procedures that have output parameters, where the output parameters can be used as a result of the operation.

Operations to invoke stored procedures with output parameters:

- **`ComplexSelectCall`**: It allows to invoke a stored procedure that performs a query in the database without making changes to it.
- **`ComplexInsertCall`**: It allows to invoke a stored procedure that makes changes to the database, which could be an insert (it is not mandatory).
- **`ComplexUpdateCall`**: It allows to invoke a stored procedure that makes changes to the database, which could be an update (it is not mandatory).
- **`ComplexDeleteCall`**: It allows to invoke a stored procedure that makes changes to the database, which could be a delete (it is not mandatory).

Important: This type of operations does not generate the query automatically so it must be provided using `@CustomSqlQuery(query="...")` indicating the query or with `@MyBatisCustomSqlStatementId` indicating the MyBatis `statementId` that contains in a non-generated file the query to be executed.

The object that MyBatis receives in this type of operations is different from the other operations

offered by Uaithne, it is these operations that create a special object that contains the properties:

- **`operation`**: contains the object with the information of the operation to be executed.
- **`result`**: contains or will contain the result of the operation. If the result of the operation is an entity (or entity view) this property is pre-initialized (it can be changed using the `initResult` property available in the operation annotation) by calling the default constructor.

The fact that the operation receives this container object causes that when writing the query it is necessary to explicitly specify the origin of the information when writing the query, prefixing the name of the field with operation. or result according to the case.

Example: How to access a field from MyBatis

```
#{operation.title,jdbcType=VARCHAR}

#{operation.description,jdbcType=VARCHAR,mode=IN}

#{result.id,jdbcType=INTEGER,mode=OUT}
```

To support the input and output parameters, Uaithne adds to the container object the properties with the same name and data type that exists in the operation and in the result, but only if the result is an entity (or entity view). Any subproperty that complies with the criteria that are contained within a property (only if it is an entity) contained within the operation is also added (if there are multiple matches, it remains with the first one, according to the order that the programmer added them).

Example: How to access an entry and exit field from MyBatis

If you have an operation that has a property (or an entity with a property) called `start` of type `java.util.Date`, a property of the same name is created in the container object, which allows access to the value of this from the operation, but when the value is setted it is written in the result. You do not need to use any prefix to use it:

```
#{start,javaType=java.util.Date,jdbcType=TIMESTAMP,mode=INOUT}
```

The input and output properties allow access from MyBatis to the operation and result as a single property, reading the value of the operation, but when the value is modified, the new value is stored in the result (without modifying the operation).

Limitation:

When using these input and output properties it may be necessary to specify the Java data type (using `javaType=`), because if not, MyBatis does not take it into account, so the data type is the default, which does not necessarily coincides with the result, causing a dark error at run time that indicates that the object cannot be cast or that a method with that data type cannot be invoked.

@ComplexSelectCall

It allows creating an operation to invoke a stored procedure that has output parameters as if it were a select, that is, without collateral effect in the database. **Note**: this operation invokes the `selectOne` method of MyBatis, and this method can cause the transaction not to commit in the database, leaving the changes made by the stored procedure pending commit if it makes any

changes.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`initResult`**: indicates whether the result must be initialized (calling the constructor without arguments) before invoking the stored procedure, thus allowing its properties to be used in the invocation of the stored procedure.

  If it is **`TRUE`**, the result is initialized by calling the constructor without arguments.

  If it is **`FALSE`**, the result is not initialized, so its value will be null, 0 or false depending on its data type and none of its properties can be accessed in the invocation of the stored procedure.

  If **`UNSPECIFIED`** (by default) the result is initialized if it is an entity (or entity view), otherwise it is not initialized.

Example: For the next entity and operation definition

```
@EntityView
class ComplexProcedureResult {
    String paramInOut;
    String paramOut;
}

@ComplexSelectCall(result = ComplexProcedureResult.class)
@CustomSqlQuery(query = {
    "select myComplexSelectProcedureCall(",
    "    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }",
    "    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}",
    "    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }",
    ")"
})
class ComplexSelectProcedureCall {
    String paramIn;
    String paramInOut;
}
```

The query is generated for PostgreSQL, indicating to MyBatis that `statementType` is of `CALLABLE` type:

```
select myComplexSelectProcedureCall(
    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }
    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}
    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }
)
```

Important:

- The **input parameters** are present in the operation and in the query they must be written by prefixing the name of the field with "`operation.`"
- The **output parameters** are present in the resulting entity and in the query must be written by prefixing the name of the field with "`result.`"
- The **input and output parameters** are present in the operation and in the resulting entity,

and in the query they must be written without specifying any prefix.

@ComplexInsertCall

It allows creating an operation to invoke a stored procedure that has output parameters as if it were an `insert`, that is, it has a collateral effect in the database. **Note**: this operation invokes the insert method of MyBatis, but JDBC does not distinguish between insert, update or delete.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`initResult`**: indicates whether the result must be initialized (calling the constructor without arguments) before invoking the stored procedure, thus allowing its properties to be used in the invocation of the stored procedure.

  If it is **`TRUE`**, the result is initialized by calling the constructor without arguments.

  If it is **`FALSE`**, the result is not initialized, so its value will be null, 0 or false depending on its data type and none of its properties can be accessed in the invocation of the stored procedure.

  If **`UNSPECIFIED`** (by default) the result is initialized if it is an entity (or entity view), otherwise it is not initialized.

Example: For the next entity and operation definition

```
@EntityView
class ComplexProcedureResult {
    String paramInOut;
    String paramOut;
}


@ComplexInsertCall(result = ComplexProcedureResult.class)
@CustomSqlQuery(query = {
    "select myComplexInsertProcedureCall(",
    "    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }",
    "    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}",
    "    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }",
    ")"
})
class ComplexInsertProcedureCall {
    String paramIn;
    String paramInOut;
}
```

The query is generated for PostgreSQL, indicating to MyBatis that `statementType` is of `CALLABLE` type:

```
select myComplexInsertProcedureCall(
    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }
    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}
    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }
)
```

Important:

- The **input parameters** are present in the operation and in the query they must be written by prefixing the name of the field with "`operation.`"
- The **output parameters** are present in the resulting entity and in the query must be written by prefixing the name of the field with "`result.`"
- The **input and output parameters** are present in the operation and in the resulting entity, and in the query they must be written without specifying any prefix.

@ComplexUpdateCall

It allows creating an operation to invoke a stored procedure that has output parameters as if it were an `update`, that is, it has a collateral effect in the database. **Note**: this operation invokes the insert method of MyBatis, but JDBC does not distinguish between insert, update or delete.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`initResult`**: indicates whether the result must be initialized (calling the constructor without arguments) before invoking the stored procedure, thus allowing its properties to be used in the invocation of the stored procedure.

  If it is **`TRUE`**, the result is initialized by calling the constructor without arguments.

  If it is **`FALSE`**, the result is not initialized, so its value will be null, 0 or false depending on its data type and none of its properties can be accessed in the invocation of the stored procedure.

  If **`UNSPECIFIED`** (by default) the result is initialized if it is an entity (or entity view), otherwise it is not initialized.

Example: For the next entity and operation definition

```
@EntityView
class ComplexProcedureResult {
    String paramInOut;
    String paramOut;
}

@ComplexUpdateCall(result = ComplexProcedureResult.class)
@CustomSqlQuery(query = {
    "select myComplexUpdateProcedureCall(",
    "    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }",
    "    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}",
    "    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }",
    ")"
})
class ComplexUpdateProcedureCall {
    String paramIn;
    String paramInOut;
}
```

The query is generated for PostgreSQL, indicating to MyBatis that `statementType` is of `CALLABLE` type:

```
select myComplexUpdateProcedureCall(
    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }
    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}
    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }
)
```

Important:

- The **input parameters** are present in the operation and in the query they must be written by prefixing the name of the field with "`operation.`"
- The **output parameters** are present in the resulting entity and in the query must be written by prefixing the name of the field with "`result.`"
- The **input and output parameters** are present in the operation and in the resulting entity, and in the query they must be written without specifying any prefix.

@ComplexDeleteCall

It allows creating an operation to invoke a stored procedure that has output parameters as if it were an `delete`, that is, it has a collateral effect in the database. **Note**: this operation invokes the insert method of MyBatis, but JDBC does not distinguish between insert, update or delete.

Parameters:

- **`result`**: indicates the result data type of the operation.
- **`initResult`**: indicates whether the result must be initialized (calling the constructor without arguments) before invoking the stored procedure, thus allowing its properties to be used in the invocation of the stored procedure.

  If it is **`TRUE`**, the result is initialized by calling the constructor without arguments.

  If it is **`FALSE`**, the result is not initialized, so its value will be null, 0 or false depending on its data type and none of its properties can be accessed in the invocation of the stored procedure.

  If **`UNSPECIFIED`** (by default) the result is initialized if it is an entity (or entity view), otherwise it is not initialized.

Example: For the next entity and operation definition

```
@EntityView
class ComplexProcedureResult {
    String paramInOut;
    String paramOut;
}

@ComplexDeleteCall(result = ComplexProcedureResult.class)
@CustomSqlQuery(query = {
    "select myComplexDeleteProcedureCall(",
    "    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }",
    "    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}",
    "    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }",
    ")"
})
class ComplexDeleteProcedureCall {
    String paramIn;
    String paramInOut;
```

```
}
```

The query is generated for PostgreSQL, indicating to MyBatis that `statementType` is of `CALLABLE` type:

```
select myComplexDeleteProcedureCall(
    #{operation.paramIn,jdbcType=VARCHAR,mode=IN   }
    #{paramInOut       ,jdbcType=VARCHAR,mode=INOUT}
    #{result.paramOut  ,jdbcType=VARCHAR,mode=OUT  }
)
```

Important:

- The **input parameters** are present in the operation and in the query they must be written by prefixing the name of the field with "`operation.`"
- The **output parameters** are present in the resulting entity and in the query must be written by prefixing the name of the field with "`result.`"
- The **input and output parameters** are present in the operation and in the resulting entity, and in the query they must be written without specifying any prefix.

Miscellaneous

Related vs Extends

You can do extends of entities to add more fields, with related (present as a property in the `@Entity` annotations and `@EntityView`) you can make a subset of it.

In operations or in entities with `related` entities, the fields maintain the annotations established in the source entities, as long as they are called the same, except the `@Optional` annotation.

Annotations to control fields with a special semantics

@InsertDate

Indicates that the field represents the date of insertion of the record, so when generating the insert query this field is always taken into account and its value is replaced by the current data date and time (the value is not read from the field to make the insert). Its data type must be `java.util.Date`, `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp`.

If you create `@Insert` operations that reference an entity with this field, this field is automatically added to the insert (even if it is not present in the operation).

This annotation is usually combined with the annotation `@ExcludeFromObject` so that it is taken into account during the generation of the query but is completely ignored in Java objects.

@InsertUser

Indicates that the field represents the user who ordered the insertion of the record, so when generating the insert query this field is always taken into account. Its data type must be String.

To give value to this field it is usually used with the annotation `@ValueWhenNull` or `@ForceValue` with which the current user of the execution context of the application is read.

If you create `@Insert` operations that reference an entity with this field, this field is automatically added to the insert (even if it is not present in the operation).

This annotation is usually combined with the annotation `@ExcludeFromObject` so that it is taken into account during the generation of the query but is completely ignored in Java objects.

@UpdateDate

Indicates that the field represents the update date of the record, so when generating the update query this field is always taken into account and its value is replaced by the current data date and time (the value is not read from the field to perform the update). Its data type must be `java.util.Date`, `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp`.

If you create `@Update` operations that reference an entity with this field, this field is automatically added in the update (even if it is not present in the operation).

This annotation is usually combined with the annotation `@ExcludeFromObject` so that it is taken

into account during the generation of the query but is completely ignored in Java objects.

Probably a field with this annotation also requires the use of the `@Optional` annotation to indicate that it may not have value in the database if it has not been updated yet, unless the field is also marked with the annotation `@InsertDate`.

@UpdateUser

Indicates that the field represents the user who ordered the update of the record, so when generating the update query this field is always taken into account. Its data type must be String.

To give value to this field it is usually used with the annotation `@ValueWhenNull` or `@ForceValue` with which the current user of the execution context of the application is read.

If you create `@Update` operations that reference an entity with this field, this field is automatically added in the update (even if it is not present in the operation).

This annotation is usually combined with the annotation `@ExcludeFromObject` so that it is taken into account during the generation of the query but is completely ignored in Java objects.

Probably a field with this annotation also requires the use of the `@Optional` annotation to indicate that it may not have value in the database if it has not been updated yet, unless the field is also marked with the annotation `@InsertUser`.

@DeleteDate

Indicates that the field represents the date of deletion of the record, so when generating the update query (to perform the logical deletion) this field is always taken into account and its value is replaced by the current date and time of the data (the value of the field to perform the update is not read). Its data type must be `java.util.Date`, `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp`.

The use of this annotation requires that the entity possess a field with the `@DeletionMark` annotation to activate the logical deletion.

If you create `@Delete` operations that refer to an entity with this field, in the update (to perform the logical deletion) this field is automatically added (even if it is not present in the operation).

In those entities with logical deletion where the erase mark is the erase date, this annotation is accompanied by the annotations `@DeletetionMark` and `@Optional`.

This annotation is usually combined with the annotation `@ExcludeFromObject` so that it is taken into account during the generation of the query but is completely ignored in Java objects.

Probably a field with this annotation also requires the use of the `@Optional` annotation to indicate that it may not have value in the database if it has not been deleted yet, unless the field is also marked with the annotation `@InsertDate` or `@UpdateDate` or both.

@DeleteUser

Indicates that the field represents the user that ordered the deletion of the record, so when generating the update query (to perform the logical deletion) this field is always taken into account.

Its data type must be `String`.

The use of this annotation requires that the entity possess a field with the `@DeletionMark` annotation to activate the logical deletion.

To give value to this field it is usually used with the annotation `@ValueWhenNull` or `@ForceValue` with which the current user of the execution is read from the context of the application.

If you create `@Delete` operations that refer to an entity with this field, in the update (to perform the logical deletion) this field is automatically added (even if it is not present in the operation).

In those entities with logical deletion where the deletion mark is the deletion user, this annotation is accompanied by the `@DeletetionMark` and `@Optional` annotations.

This annotation is usually combined with the annotation `@ExcludeFromObject` so that it is taken into account during the generation of the query but is completely ignored in Java objects.

Probably a field with this annotation also requires the use of the `@Optional` annotation to indicate that it may not have value in the database if it has not been deleted yet, unless the field is also marked with the annotation `@InsertUser` or `@UpdateUser` or both.

@DeletionMark

Indicates that the field represents the logical deletion mark of the record.

The use of a field with this annotation causes Uaithne to generate a logical deletion update (by updating the value of this field) instead of a physical deletion delete; the use of this annotation also causes it to be automatically included in the where of the update and select a clause that ensures that the record is not deleted, and in the insert is included indicating that the record has not been deleted.

The data types supported are Boolean or any other type of data annotated with `@Optional`.

To handle the value of this annotation in the case that its data type is not boolean, it is usually combined with the annotation `@DeleteDate` or `@DeleteUser`.

This annotation is usually combined with the annotation `@ExcludeFromObject` so that it is taken into account during the generation of the query but is completely ignored in Java objects.

@IgnoreLogicalDeletion

In operations with this annotation the generated query must be performed ignoring any field annotated with `@DeletionMark`; so if it is an update or a select it is not included in the where the check that the record is not deleted, but if it is a deletion operation, the physical deletion is carried out using a delete instead of a logical deletion.

@Version

Indicates that the field represents the mark that indicates the version of a record. Its operation is still to be defined, for now it is only used to indicate in the entity of its purpose, but in the generated queries it is always ignored.

Other annotations to better control the queries

@PageQueries

It allows to specify separately the query of an `@SelectPage` operation allowing to indicate the query that retrieves the data of the page and the query that allows to count the number of existing records.

Parameters:

- **`selectCount`:** Query to check the amount of existing records.
- **`selectPage`:** Query to be executed to retrieve the data from the data page.

~~@EntityQueries~~

It should probably be marked as deprecated. It allows to specify the queries of the operations that are generated automatically for an entity. It is better to use in this case the operations of entities in combination with `@CustomSqlQuery` that allows to achieve better results.

Parameters:

- **`selectById`:** Query to be executed to recover the record given its identifier.
- **`insert`:** Insertion query of a new record.
- **`lastInsertedId`:** Query that allows you to recover the identifier of the record that was just inserted.
- **`update`:** Update query of a record in the database.
- **`deleteById`:** Query to be executed to delete a record given its identifier.
- **`merge`:** Query to be executed to update a record in the database, but only updates those columns whose fields in the entity are not null.

~~@Query~~

Probably it should be marked as deprecated. It allows to specify the query of an operation. It is better to use the annotation `@CustomSqlQuery` instead.

Java Beans Validations

- You can use the Java Beans Validations annotations on the fields of the entities or operations, these are generated in the resulting classes.
- It takes a lot of advantage that the fields maintain the annotations established to the entities when they are used in the operations or related entities. The only requirement is to have the same name. This avoid to to repeat the information of the Java Beans Validations annotation. It is enough to place them in the entities and in any other field in the operations or entities with related that is not in the original entity.

Uaithne configuration

- Uaithne configurations: `@UaithneConfiguration`
- Generation of common classes using `@SharedLibrary`
- Generation of common MyBatis classes using `@SharedMyBatisLibrary`
- MyBatis configuration: it is necessary to use `ApplicationParameterDriver` and

`RetainIdPlugin` to allow everything works correctly

Ω end Ω

Designed by: **Juan Luis Paz Rojas** - https://github.com/juanluispaz