Uaithne generator Java

Diseñado por: **Juan Luis Paz Rojas**

Prólogo

Mejorar la productividad del equipo de desarrollo ha sido una de las grandes motivaciones detrás del diseño de de la arquitectura de Uaithne, y sí sola representa una gran mejoría respecto a la clásica arquitectura n capas. Tras implementar la nueva arquitectura aún quedaba margen para mejorar la productividad, generando código automáticamente que se encargue de manejar la verbosidad requerida por Java, y sobre todo, que genere el código de acceso a la base de datos con sus respectivas queries SQL que maneje la mayoría de operaciones.

Uaithne viene acompañado, opcionalmente, con un generador de código que permite generar (y regenerar) las operaciones y entidades a partir de una pequeña definición, así como la lógica de acceso a base de datos requeridas por estas, usando para ello MyBatis como framework de acceso a la base de datos. De esta forma, solo se deja al programador aquellas actividades en las que realmente se requiera trabajo humano debido a las decisiones que se deben tomar.

La arquitectura de Uaithne y el generador de código trabajando juntos han demostrado en multitud de proyectos de diversos tamaños ser una herramienta de gran utilidad para la programación del backend de aplicaciones consiguiendo durante ello una alta productividad del equipo de desarrollo.

Este documento es la documentación de referencia de generador de código de Uaithne, se asume que previamente el lector ya conoce la arquitectura de Uaithne, de no ser así, se recomienda leer primero el documento que explica la arquitectura de Uaithne para Java.

juanluispaz@gmail.com
https://github.com/juanluispaz/uaithne-generator-java
Versión: 0.6.0 - 30 de marzo de 2018

Índice

Uaithne generator Java

Diseñado por: **Juan Luis Paz Rojas**

Generador de código

Uaithne incluye un generador de código que permite reducir la cantidad de código necesario para construir las operaciones y sus ejecutores, también es posible generar la implementación de los ejecutores para acceder a la base de datos con sus respectivas queries SQL.

El generador admite que el nombre de los módulos, entidades y operaciones empiece por el carácter _ que es ignorado al generar el código, de forma tal que el nombre de los elementos generados no incluyen el símbolo _ (solo cuando sea el primer carácter del nombre). El uso del símbolo _ es útil para distinguir entre la entrada del generador y la salida de este, de esta forma no se presta a confusiones al utilizar la herramienta de autocompletado del IDE.

Módulos

Para facilitar el mantenimiento el código, Uaithne agrupa las operaciones y opcionalmente las entidades en módulos; al tener esta agrupación lógica de operaciones se mantienen todas las piezas altamente relacionadas juntas. Por ejemplo, se puede crear un módulo por tabla de la base de datos conteniendo todas las operaciones de que operan sobre esa tabla y las entidades que la representan.

Los módulos se crean al añadir la anotación `@OperationModule` a una clase, esta clase actúa como contenedor de todas las operaciones y entidades que conforman el módulo.

```
@OperationModule
public class _events {

}
```

La clase anotada con `@OperationModule` actúa como contenedor de los elementos que conforman el módulo, por lo que dentro solo se esperan encontrar construcciones que el generador de Uaithne sea capaz de procesar, en ningún caso debe existir campos para esta clases o métodos en esta clase o en cualquiera de las clases que contenga.

El nombre del módulo se utiliza como nombre de paquete de las operaciones generadas, por lo que se recomienda que su nombre siga las recomendaciones para nombrar paquetes, que para este tipo de paquetes entre otras son: primera letra en minúscula y nombre en plural.

Entidades

Las entidades representan las tablas en la base de datos y se crean al añadir la anotación `@Entity` a una clase que solo contiene los campos que la conforman. Por comodidad se permite declarar entidades dentro de módulos aunque solo las operaciones pertenecen al módulo.

```
@OperationModule
public class _events {

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

Para `_Event` se genera la siguiente clase que representa la entidad:

```
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;
    }
}
```

A los campos de una entidad se le pueden añadir la anotación `@Id` para indicar que ese campo (o esos campos) identifican al objeto, los campos que tengan la anotación `@Id` se utilizan en los métodos `equals` y `hashCode`, si no existe ningún campo con la anotación `@Id` se utilizan todos los campos de la entidad en la implementación de `equals` y `hashCode`.

Los campos que son opcionales, es decir, cuyo valor admiten nulo, deben ser marcados con la anotación `@Optional`, de forma tal que el generador pueda utilizar esta información entre otro lugares en los constructores generados (solo los campos obligatorios son incluidos en los constructores). Los campos marcados con la anotación `@ExcludeFromConstructor` no son incluidos en el constructor aunque sean obligatorios.

En la entidad generada se implementa el método `toString` que incluye el nombre de la entidad así como el nombre y valor de cada uno de los campos que conforman la entidad, al menos que el campo esté marcado con la anotación `@ExcludeFromToString` en cuyo caso no es incluido en la implementación de método `toString`.

Las entidades pueden extender otras entidades, de forma tal que el generador tiene en cuenta los campos de cada una de las entidades y al generarla respeta la jerarquía con la que se han hecho, también puede extender de otra clase que no sea una entidad (en cuyo caso el generador solo tiene en cuenta que la entidad extiende esa clase) o implementar interfaces que no requieran implementar ningún método (como `Serializable`).

Si la entidad está definida dentro de un módulo una serie de operaciones: (la anotación `@Entity` permite indicar cuales se deben generar, también se puede activar que se generen por defecto en la configuración de Uaithne)

- **`InsertEvent`**: Inserta un objeto Event en la fuente de datos, retorna el Id del registro insertado.
- **`UpdateEvent`**: Actualiza un objeto Event en la fuente de datos, retorna la cantidad de registros actualizados.
- **`DeleteEventById`**: Elimina un registro Event de la fuente de datos, esta operación recibe el id del registro y retorna la cantidad de registros eliminados.
- **`SelectEventById`**: Recupera un registro Event de la fuente de datos, esta operación recibe el id del registro y retorna el objeto Event con ese id o nulo si no lo encuentra.
- **`SaveEvent`**: Inserta o actualiza un objeto Event en la fuente de datos, si el identificador del objeto Event que recibe la operación es nulo inserta un nuevo registro, si no actualiza el registro con ese identificador, retorna el identificador del registro insertado.
- **`MergeEvent`**: Actualiza los campos de un registro Event en la fuente de datos cuyos valores sean distintos de nulo. Esta operación actualiza solo los campos que posean valor en el registro cuyo id coincida con el suministrado en el objeto y retorna la cantidad de registros actualizados.
- **`JustInsertEvent`**: Inserta un objeto Event en la fuente de datos, retorna la cantidad de registros insertados. Esta operación no trata de recuperar el id del registro insertado.
- **`JustSaveEvent`**: Inserta o actualiza un objeto Event en la fuente de datos, si el identificador del objeto Event que recibe la operación es nulo inserta un nuevo registro, si no actualiza el registro con ese identificador. Esta operación no trata de recuperar el id del registro insertado, por lo que retorna la cantidad de registros afectados.

Para poder generar estas operaciones es necesario que esté contenida dentro de un módulo y que tenga un único campo marcado con la anotación `@Id` definido por esta o por cualquier entidad

de la que extienda directa o indirectamente.

Para poder generar las operaciones `Save` o `JustSave` es necesario que el tipo de datos del campo marcado con la anotación `@Id` sea un tipo de dato al que se le pueda asignar nulo.

Vista de entidades

En muchas circunstancias, al consultar la base de datos, no se desea extraer un registro entero, sino que en su lugar se desea extraer parte de esa información, o incluso, esa información puede estar combinada con información de otra tabla; en esos casos se puede crear una entidad que solo contenga la información deseada, a este tipo de entidades se les denominará “vista de entidad” y su construcción es idéntica a la de una entidad con la diferencia que no representa de manera íntegra a un registro almacenado en una tabla.

Las vistas de entidades se crean al añadir la anotación `@EntityView` a una clase que solo contiene los campos que la conforman. Por comodidad se permite declarar vista de entidades dentro de módulos aunque solo las operaciones pertenecen al módulo.

```
@OperationModule
public class _events {

    [@Entity _Event ...]

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

La única diferencia entre una entidad y una vista de entidad durante el proceso de generación de código, es que para la vista de entidades no se generan operaciones de entidades (incluso si se ha marcado su generación por defecto).

Entidades afines

Al crear una entidad o vista de entidad es posible especificar una entidad afín a la que se está definiendo, cuando se especifica una entidad afín (pudiendo ser esta una entidad o una vista de entidad sin importar el caso) provoca que el generador utilice la información de la entidad afín para complementar la entidad o la vista de entidad, excepto la clase base que extienda o las interfaces que implemente, de forma tal que no se tenga que repetir la información especificada en la entidad afín; en el caso de los campos, la información de estos es complementada con la información del campo de igual nombre en la entidad afín, excepto si es obligatorio u opcional que solo se puede especificar en la entidad o vista de entidad en la que se define.

```
@OperationModule
public class _events {

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

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

}
```

A indicar que `_Event` es la entidad afín la vista de entidad `_EventView` no tiene que volver a marcar el campo `id` como identificador (entre muchas otras cosas que en este punto aún no se han especificado en la entidad), ya que esa información se extrae del campo de igual nombre de la entidad `_Event`.

Información complementaria para la definición de la entidad

Si se desea, es posible alterar algunos aspectos del código generado para conseguir que sea tratado de manera diferente mediante una serie de anotaciones complementarias.

Información que se pueden añadir sobre la entidad:

- **`abstract`**: si se marca la entidad o vista de entidad como abstracta provoca que la entidad generada sea abstracta.
- **`extends`**: si se añade una clase base (usando `extends`) a la entidad provoca que la entidad generada extienda de esa clase base, si la clase base es otra entidad, el generador usará la información provista en la clase base para generar el código.
- **`implements`**: si se implementan interfaces (usando `implements`) en la entidad provoca que la entidad generada implemente las interfaces indicadas.
- **`@Doc`**: si se añade esta anotación a la entidad provoca que el texto indicado en ella sea escrito como documentación de la entidad generada. Esta anotación recibe un array de string, en donde cada item del array corresponde a una línea de la documentación.
- **`@Deprecated`**: si se añade esta anotación a la entidad provoca que la entidad generada sea marcada como obsoleta.
- Cualquier anotación que esté fuera del paquete de anotaciones de Uaithne (`org.uaithne.annotations.*`) será generado en la entidad resultante.

Información que se puede añadir sobre los campos:

- **`@Id`**: indica que el campo forma parte del identificador del registro, es posible tener varios campos que conformen el identificador del registro, pero si se desea que Uaithne genere las operaciones que usan el id o retornan el id es necesario que solo tengan un único identificador.

  Los campos marcados como identificadores son lo que se usan para implementar los métodos `equals` y `hashCode`, de forma tal que dos instancias serán iguales si todos sus campos marcados con `@Id` son iguales. Si la entidad no tiene campos marcados como identificadores todos los campos que conforman la entidad se utilizarán para generar los métodos `equals` y `hashCode`.

  Si se tiene tiene una entidad con varios campos que conforman el identificador se recomienda marcarla como una vista de entidad y manejar directamente las operaciones

que insertan, actualizan o eliminan el registro.

- **`@Optional`**: indica que el campo es opcional, por lo que admite nulo. Si un campo no posee la anotación `@Optional` se asume que es obligatorio, y en consecuencia no admite nulo.
- **`@ExcludeFromConstructor`**: indica que el campo no debe ser incluido en el constructor de la clase de la entidad generada. Los campos marcados con `@Optional` son automáticamente excluidos del constructor de la clase de la entidad generada.
- **`@ExcludeFromObject`**: indica que este campo no debe ser incluido en la clase de la entidad generada, su existencia tiene otros propósitos y en ningún caso debe formar parte del código Java generado para la entidad.
- **`@ExcludeFromToString`**: indica que este campo no debe ser incluido en la implementación del método `toString` de la clase generada.
- **`@DefaultValue`**: indica cual es el valor por defecto a ser asignado al campo durante la inicialización de la clase de la entidad generada. El valor indicado con esta anotación se traduce en una asignación del valor al campo antes que se ejecute el constructor.

  Ejemplo:

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

  Genera:

  ```
  public class Foo implements Serializable {

      private Integer i = 10;

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

      public Foo() { }

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

  La anotación `@DefaultValue` recibe un string que será interpretado en función del tipo de dato del campo, si el tipo es **`boolean` o `Boolean`** el texto debe ser `true` o `false`, si es un tipo **numérico** (`byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, `float`, `Float`, `double`, `Double`) el texto debe ser un número, si se trata de **`BigInteger` o `BigDecimal`** el texto es el que se va a usar para inicializar la nueva instancia de estas clases, si es **`char` o `Character`** el texto debe ser un carácter que será apropiadamente encomillado, si se trata de **`String`** el texto debe ser una secuencia de caracteres que será apropiadamente encomillado, si se trata de **cualquier otro tipo** el texto se tratará como el código Java requerido para realizar la inicialización.

- **`@Doc`**: si se añade esta anotación al campo provoca que el texto indicado en ella sea

escrito como documentación de este en la entidad generada. Esta anotación recibe un array de string, en donde cada item del array corresponde a una línea de la documentación.
- **`@Deprecated`**: si se añade esta anotación al campo provoca que en la entidad generada este sea marcado como obsoleto.
- **transient**: se se marca el campo no serializable provoca que en la entidad generada este campo sea marcado como no serializable.
- Cualquier anotación que esté fuera del paquete de anotaciones de Uaithne (`org.uaithne.annotations.*`) será generado en el campo de la entidad resultante.

Operaciones

Las operaciones se crean al añadir la anotación `@Operation` (o cualquier otra anotación más específica que indica el tipo de operación) a una clase que contiene los campos que contendrán la información necesaria para poder llevar a cabo su ejecución, la operación retorna un resultado cuyo tipo se especifica en el parámetro `result` de la anotación (en aquellas operaciones cuyo tipo de resultado no es implícito). Las operaciones siempre deben estar contenidas dentro de un módulo.

Ejemplo:

```
@OperationModule
public class _synchronization {

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

}
```

Nota: Si se desea tener una operación que no retorna valor se debe usar como tipo de resultado la clase `Void`, cuyo único valor válido es nulo.

Limitación: no es posible especificar resultado de operaciones que incluyan argumentos genéricos.

Tipos de operaciones

Uaithne soporta distintos tipos de operaciones, que indican al generador la semántica de esta, lo cual es especialmente útil cuando se genera el código necesario para acceder a la base de datos. Los diferentes tipos de operaciones soportados son:

Operaciones generalistas:

- **`@Operation`**: Esta operación es la más general de todas, carente de toda semántica, cuya implementación debe ser siempre provista por el programador.

Operaciones de consulta:

- **`@SelectOne`**: Esta operación retorna la entidad cuyos campos coincidan con los criterios

indicados en la operación.
- **`@SelectMany`**: Esta operación retorna un listado de entidades cuyos campos coincidan con los criterios indicados en la operación.
- **`@SelectPage`**: Esta operación retorna un listado de entidades a modo de vista paginada cuyos campos coincidan con los criterios indicados en la operación.
- **`@SelectCount`**: Esta operación retorna el número de entidades cuyos campos coincidan con los criterios indicados en la operación.
- **`@SelectEntityById`**: Esta operación retorna la entidad cuyo id sea el indicado en la operación.

Operaciones de modificación de entidades:

- **`@InsertEntity`**: Esta operación inserta un registro con los valores indicados en la entidad contenida por la operación. Retorna, según se especifique, el id del registro insertado (por defecto) o la cantidad de registros insertados.
- **`@UpdateEntity`**: Esta operación actualiza un registro con los valores indicados en la entidad contenida por la operación, el registro a actualizar es el que coincida con el id indicado en la entidad. Retorna la cantidad de registros actualizados.
- **`@DeleteEntityById`**: Esta operación elimina un registro cuyo id sea el indicado en la operación. Retorna la cantidad de registros eliminados.
- **`@SaveEntity`**: Esta operación inserta o actualiza un registro con los valores indicados en la entidad contenida por la operación, el registro a actualizar es el que coincida con el id indicado en la entidad, de no tener se inserta. Retorna, según se especifique, el id del registro insertado o actualizado (por defecto) o la cantidad de registros insertados o actualizados.
- **`@MergeEntity`**: Esta operación actualiza un registro con los valores indicados en la entidad contenida por la operación cuyos valores sean distinto de nulo, el registro a actualizar es el que coincida con el id indicado en la entidad. Retorna la cantidad de registros actualizados.

Operaciones de modificación de datos no restringidas a entidades:

- **`@Insert`**: Esta operación inserta un registro con los valores indicados en la operación. Retorna, según se especifique, el id del registro insertado (por defecto) o la cantidad de registros insertados.
- **`@Update`**: Esta operación actualiza un registro con los valores indicados en la operación. Retorna la cantidad de registros actualizados.
- **`@Delete`**: Esta operación elimina los registro con los valores indicados por la la operación. Retorna la cantidad de registros eliminados.

@Operation

Esta es la operación más general de todas, y no tiene ninguna semántica asociada, por lo que se trata de una operación que ejecuta “algo” y retorna “algún” resultado; los campos de la operación contendrán la información necesaria para que esta se lleve a cabo.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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

Esta operación recupera una entidad (o vista de entidad) de una fuente de datos, típicamente una base de datos; los campos de la operación contendrán la información necesaria para recuperar la entidad deseada de la fuente de datos.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación.
- **`limit`**: si se establece en `true` indica al backend (típicamente la implementación de acceso a la base de datos) que aunque potencialmente pueda retornar más de un registro, en caso de ocurrir, la operación retorna el primer registro encontrado en lugar de dar un error.
- **`related`**: si el resultado de la operación no es una entidad, en este campo se debe indicar la entidad a la cual la operación hace referencia.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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

Esta operación recupera un listado de entidades (o vistas de entidades) de una fuente de datos, típicamente una base de datos; los campos de la operación contendrán la información necesaria para recuperar las entidades deseadas de la fuente de datos.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación.
- **`distinct`**: si se establece en `true` indica al backend (típicamente la implementación de acceso a la base de datos) que aunque potencialmente pueda retornar registros repetidos, en caso de ocurrir, debe asegurarse de retornar registros únicos (eliminado los repetidos).
- **`related`**: si el resultado de la operación no es una entidad, en este campo se debe indicar la entidad a la cual la operación hace referencia.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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

Esta operación recupera un listado paginado de entidades (o vistas de entidades) de una fuente de datos, típicamente una base de datos; los campos de la operación contendrán la información necesaria para recuperar las entidades deseadas de la fuente de datos.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación.
- **`distinct`**: si se establece en `true` indica al backend (típicamente la implementación de acceso a la base de datos) que aunque potencialmente pueda retornar registros repetidos, en caso de ocurrir, debe asegurarse de retornar registros únicos (eliminado los repetidos).
- **`related`**: si el resultado de la operación no es una entidad, en este campo se debe indicar la entidad a la cual la operación hace referencia.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Las operaciones `@SelectPage` implementan la interfaz `DataPageRequest`, esta interfaz define una serie de propiedades que debe tener cualquier operación `@SelectPage` que permiten controlar el paginado de los datos. La interfaz `DataPageRequest` que se define como:

```
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);
}
```

Adicionalmente a los campos que se indiquen para la operación se añaden automáticamente algunos más:

- **`limit`**: indica la cantidad de registros a ser obtenidos en una página de datos, es decir, es el tamaño máximo de la página. Si este campo es nulo indica que no tiene límite, por lo que se debe traer la máxima cantidad de registros posibles.
- **`offset`**: indica la cantidad de registros a ser ignorados previos a la página de datos que se desea consultar. **Ejemplo**: Si se tiene que el tamaño de la página es de 20 registros, y se desea obtener la tercera página, el valor de este campo debería ser (3 - 1) * 20, que son 40

registros previos a ser ignorados, ya que se va a obtener los registros 41 al 60 (considerando que los índices empiezan en cero). Si este campo es nulo indica que no se va a ignorar ningún registro.

- **`maxRowNumber`**: esta es una propiedad de solo lectura que se calcula como la suma del limit y el offset y representa el índice máximo del registro a ser consultado. **Ejemplo**: Si se tiene que el tamaño de la página es de 30 registros, y se va a ignorar los primeros 90 registros (se desea la cuarta página) el valor de esta propiedad sería 30 + 90, que son 120, por lo que se va a obtener los registros 91 al 120 (considerando que los índices empiezan en cero).
- **`dataCount`**: indica la cantidad total de registros, si se especifica el valor de este campo en la operación provoca que la operación no tenga que consultar la cantidad total de registros, en su lugar se asume que el valor es el aquí indicado, esto es útil para evitar la consulta adicional que cuenta los registros.
- **`onlyDataCount`**: si se especifica `true` en este campo provoca que la operación únicamente consulte la cantidad total de registro, sin traerse los datos de ninguna de las páginas.

El resultado de una operación `@SelectPage` es un objeto de tipo `DataPage` que contiene la lista de registros de la página así como la información de la cantidad total de registros, tamaño de la página y cantidad de registros ignorados previamente. La clase `DataPage` se define como:

```
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;
    }
}
```

La clase `DataPage` contiene los campos:

- **`limit`**: contiene el `limit` usado en la operación que retornó este objeto y representa el tamaño máximo de la página.
- **`offset`**: contiene el `offset` usado en la operación que retornó este objeto y representa la cantidad de registros previos que han sido ignorados.
- **`dataCount`**: contiene la cantidad total de registros, si la operación especificó uno el valor de este campo es el indicado en el operación, sino el valor es el conteo de la cantidad total de registros existente.

- **`data`**: contiene el listado de los registros que pertenecen a la página solicitada.

@SelectCount

Esta operación recupera la cantidad de entidades (o vistas de entidades) de una fuente de datos, típicamente una base de datos; los campos de la operación contendrán la información necesaria para determinar cuáles son las entidades a ser contadas en la fuente de datos.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación, por defecto es `BigInteger`, aunque sería perfectamente válido indicar cualquier otro tipo de dato numérico, tal como `Integer` o `Long`. El resultado de esta operación siempre es un número ya que retorna la cantidad de registros.
- **`related`**: este campo se debe indicar la entidad a la cual la operación hace referencia.

Ejemplo: Para la definición de la operación

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

    Integer calendarId;

}
```

Se genera la clase que representa la operación:

```
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

Esta operación recupera una entidad (o vista de entidad) dado su identificador de una fuente de datos, típicamente una base de datos; las operaciones de este tipo asumen que reciben el identificador de la entidad (que es del tipo del campo que lleva la anotación `@Id` en la entidad) y no admiten campos adicionales.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación.
- **`limit`**: si se establece en `true` indica al backend (típicamente la implementación de acceso a la base de datos) que aunque potencialmente pueda retornar más de un registro,

en caso de ocurrir, la operación retorna el primer registro encontrado en lugar de dar un error.

Requisitos:

- Esta operación requiere que la entidad (o vista de entidad) resultante tenga declarado un único campo identificador.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Campos de la operación generada:

- **`id`**: identificador de la entidad que se desea recuperar (que es del tipo del campo que lleva la anotación `@Id` en la entidad).
- No se admite ningún campo adicional, las operaciones `@SelectEntityById` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `id`.

La operación generada implementa la interfaz `SelectByIdOperation`, esta interfaz extiende de `Operation` y define la propiedad `id`. La definición de la interfaz `SelectByIdOperation` es:

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

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

@InsertEntity

Esta operación inserta una entidad (o vista de entidad) en una fuente de datos, típicamente una base de datos; las operaciones de este tipo asumen que reciben la entidad y no admiten campos adicionales, estas operaciones retornan el identificador del registro insertado (por defecto) o la cantidad de registros insertados (si se establece como `false` el valor de la propiedad

`returnLastInsertedId` de la anotación).

Parámetros:

- **`value`**: indica cual es el tipo de la entidad a ser insertada. **Nota**: en Java, cuando la propiedad de la anotación se llama value y es la única propiedad a ser usada, es posible omitir el nombre de la propiedad y especificar directamente el valor.
- **`returnLastInsertedId`**: si se establece en `false` indica que la operación debe retornar la cantidad de registros insertados (que es de tipo `Integer`), en caso contrario (por defecto), retorna el id del registro insertado (que es del tipo del campo que lleva la anotación `@Id` en la entidad).

Requisitos:

- Esta operación requiere que la entidad (o vista de entidad) insertada tenga declarado un único campo identificador, al menos que el campo `returnLastInsertedId` de la anotación haya sido establecido a false, en cuyo caso este requisito no se exige.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Campos de la operación generada:

- **`value`**: entidad a ser insertada.
- No se admite ningún campo adicional, las operaciones `@InsertEntity` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `value`.

La operación generada implementa la interfaz `InsertValueOperation`, esta interfaz extiende de `Operation` y define la propiedad `value`. La definición de la interfaz `InsertValueOperation` es:

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

```
}
```

Ejemplo: Si en la definición de la operación se fuese indicado se desea que el resultado sea la cantidad de registros insertados

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Campos de la operación generada:

- **`value`**: entidad a ser insertada.
- No se admite ningún campo adicional, las operaciones `@InsertEntity` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `value`.

La operación generada implementa la interfaz `JustInsertValueOperation`, esta interfaz extiende de `Operation` y define la propiedad `value`. La definición de la interfaz `JustInsertValueOperation` es:

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

@UpdateEntity

Esta operación actualiza una entidad (o vista de entidad) en una fuente de datos, típicamente una base de datos; las operaciones de este tipo asumen que reciben la entidad y no admiten campos adicionales, estas operaciones retornan la cantidad de registros actualizados (que es de tipo `Integer`). Los campos a actualizar son todos los campos de la entidad excepto el identificador que se utiliza para determinar cuál registro es el que se va a actualizar.

Parámetros:

- **`value`**: indica cual es el tipo de la entidad a ser actualizada. El id de la entidad indicada corresponderá al id del registro a ser actualizado. **Nota**: en Java, cuando la propiedad de la anotación se llama value y es la única propiedad a ser usada, es posible omitir el nombre

de la propiedad y especificar directamente el valor.

Requisitos:

- Esta operación requiere que la entidad (o vista de entidad) a ser actualizada tenga declarado un único campo identificador.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Campos de la operación generada:

- **`value`**: entidad a ser actualizada.
- No se admite ningún campo adicional, las operaciones `@UpdateEntity` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `value`.

La operación generada implementa la interfaz `UpdateValueOperation`, esta interfaz extiende de `Operation` y define la propiedad `value`. La definición de la interfaz `UpdateValueOperation` es:

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

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

@DeleteEntityById

Esta operación elimina una entidad (o vista de entidad) dado su identificador de una fuente de datos, típicamente una base de datos; las operaciones de este tipo asumen que reciben el identificador de la entidad (que es del tipo del campo que lleva la anotación `@Id` en la entidad) y no admiten campos adicionales, estas operaciones retornan la cantidad de registros eliminados (de tipo `Integer`).

Parámetros:

- **`related`**: indica cual es el tipo de entidad a ser eliminada.

Requisitos:

- Esta operación requiere que la entidad (o vista de entidad) a ser eliminada tenga declarado un único campo identificador.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Campos de la operación generada:

- **`id`**: identificador de la entidad que se desea eliminar (que es del tipo del campo que lleva la anotación `@Id` en la entidad).
- No se admite ningún campo adicional, las operaciones `@DeleteEntityById` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `id`.

La operación generada implementa la interfaz `DeleteByIdOperation`, esta interfaz extiende de `Operation` y define la propiedad `id`. La definición de la interfaz `DeleteByIdOperation` es:

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

@SaveEntity

Esta operación inserta o actualiza una entidad (o vista de entidad) en una fuente de datos, típicamente una base de datos; las operaciones de este tipo asumen que reciben la entidad y no admiten campos adicionales, estas operaciones retornan el identificador del registro insertado o actualizado (por defecto) o la cantidad de registros insertados o actualizados (si se establece como false el valor de la propiedad `returnLastInsertedId` de la anotación). Los campos a actualizar son todos los campos de la entidad excepto el identificador que se utiliza para determinar cuál registro es el que se va a actualizar, en caso de ser nulo el id se inserta un

registro nuevo.

Parámetros:

- **`value`**: indica cual es el tipo de la entidad a ser insertada o actualizada. **Nota**: en Java, cuando la propiedad de la anotación se llama value y es la única propiedad a ser usada, es posible omitir el nombre de la propiedad y especificar directamente el valor.
- **`returnLastInsertedId`**: si se establece en `false` indica que la operación debe retornar la cantidad de registros insertados o actualizados (que es de tipo `Integer`), en caso contrario (por defecto), retorna el id del registro insertado o actualizado (que es del tipo del campo que lleva la anotación `@Id` en la entidad).

Requisitos:

- Esta operación requiere que la entidad (o vista de entidad) a ser insertada o actualizada tenga declarado un único campo identificador.
- Esta operación requiere que el tipo de datos del campo identificador admita asignarle nulo, por lo que no puede ser `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, en lugar de ellos se puede usar `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` o cualquier otro objeto.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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;
    }

}
```

Campos de la operación generada:

- **`value`**: entidad a ser insertada o actualizada.
- No se admite ningún campo adicional, las operaciones `@SaveEntity` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `value`.

La operación generada implementa la interfaz `SaveValueOperation`, esta interfaz extiende de

`Operation` y define la propiedad `value`. La definición de la interfaz `SaveValueOperation` es:

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

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

Ejemplo: Si en la definición de la operación se fuese indicado se desea que el resultado sea la cantidad de registros insertados o actualizados

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Campos de la operación generada:

- **`value`**: entidad a ser insertada o actualizada.
- No se admite ningún campo adicional, las operaciones `@SaveEntity` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `value`.

La operación generada implementa la interfaz `JustSaveValueOperation`, esta interfaz extiende de `Operation` y define la propiedad `value`. La definición de la interfaz `JustSaveValueOperation` es:

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

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

@MergeEntity

Esta operación actualiza una entidad (o vista de entidad) en una fuente de datos, típicamente una base de datos; las operaciones de este tipo asumen que reciben la entidad y no admiten campos adicionales, estas operaciones retornan la cantidad de registros actualizados (que es de tipo `Integer`). Los campos a actualizar son todos los campos de la entidad cuyo valor sea distinto de

nulo excepto el identificador que se utiliza para determinar cuál registro es el que se va a actualizar.

Parámetros:

- **`value`**: indica cual es el tipo de la entidad a ser actualizada. El id de la entidad indicada corresponderá al id del registro a ser actualizado. **Nota**: en Java, cuando la propiedad de la anotación se llama value y es la única propiedad a ser usada, es posible omitir el nombre de la propiedad y especificar directamente el valor.

Requisitos:

- Esta operación requiere que la entidad (o vista de entidad) a ser eliminada tenga declarado un único campo identificador.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

```
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;
    }

}
```

Campos de la operación generada:

- **`value`**: entidad a ser actualizada.
- No se admite ningún campo adicional, las operaciones `@MergeEntity` deben ser definidas sin ningún campo dentro de ella, automáticamente se añadirá el campo `value`.

La operación generada implementa la interfaz `MergeValueOperation`, esta interfaz extiende de `Operation` y define la propiedad `value`. La definición de la interfaz `MergeValueOperation` es:

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

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

@Insert

Esta operación inserta un registro en una fuente de datos, típicamente una base de datos y retorna el identificador del registro insertado (por defecto) o la cantidad de registros insertados (si se establece como `false` el valor de la propiedad `returnLastInsertedId` de la anotación). Los valores a insertar en el registro son aquellos especificados como campos de la operación. Esta operación es mucho más flexible que `@InsertEntity` ya que permite especificar los campos a insertar en el nuevo registro.

Parámetros:

- **`related`**: indica cual es la entidad asociada al registro a insertar.
- **`returnLastInsertedId`**: si se establece en `false` indica que la operación debe retornar la cantidad de registros insertados (que es de tipo `Integer`), en caso contrario (por defecto), retorna el id del registro insertado (que es del tipo del campo que lleva la anotación `@Id` en la entidad).

Requisitos:

- Esta operación requiere que la entidad (o vista de entidad) asociada a la inserción tenga declarado un único campo identificador, al menos que el campo `returnLastInsertedId` de la anotación haya sido establecido a false, en cuyo caso este requisito no se exige.

Ejemplo: Para la definición de la operación

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

Se genera la clase que representa la operación:

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

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

    public InsertCalendarByTitle() { }

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

Ejemplo: Si en la definición de la operación se fuese indicado se desea que el resultado sea la cantidad de registros insertados

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

Se genera la clase que representa la operación:

```
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

Esta operación actualiza un registro en una fuente de datos, típicamente una base de datos y retorna la cantidad de registros actualizados (que es de tipo `Integer`). Los campos a actualizar son todos los campos de la operación que tengan la anotación `@SetValue`, si en esta anotación se establece la propiedad `ignoreWhenNull` a `true` solo se actualiza el campo cuando su valor en la operación sea distinto de nulo; todos aquellos campos de la operación que no posean la anotación `@SetValue` se utilizan para determinar cuál registro es el que se va a actualizar. Esta operación es mucho más flexible que `@UpdateEntity` ya que permite especificar los campos por los cuales se va buscar los registros a actualizar (pudiendo tener varios, que no necesariamente tengan que ser el identificador del registro), también permite especificar cuales son los campos a actualizar el registro.

Parámetros:

- **`related`**: indica cual es la entidad asociada al registro a actualizar.

La operación `@Update` actualiza todos los registros que coincidan con los campos suministrados en la operación, excepto aquellos campos que lleven la anotación `@SetValue`, estos campos son los que se van a actualizar con los valores suministrados en la operación.

Parámetros de `@SetValue`:

- **ignoreWhenNull**: indica si se debe omitir actualizar este campo cuando su valor es nulo; es decir, si es `false`, el campo se actualiza con el valor suministrado en la operación, incluso cuando es nulo, en cuyo caso el nuevo valor será nulo; ahora si es `true`, se actualiza con el valor suministrado siempre y cuando este sea no nulo, pero si es nulo, no se actualiza el valor. Establecer `ignoreWhenNull` a `true` es útil para conseguir que ciertos campos sean actualizados de manera opcional. El valor por defecto es `false`.

  Nota: Al establecer `ignoreWhenNull` a `true` es requerido que el campo adicionalmente tenga la anotación `@Optional` que indica que el campo es opcional, y por ende puede admitir nulo.

Ejemplo: Para una operación que permite actualizar la descripción de un evento dado su título y el

identificador de calendario al que pertenece

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

Se genera la clase que representa la operación:

```
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;
    }
}
```

Ejemplo: Para una operación que permite actualizar el título de un calendario y opcionalmente su descripción dato su identificador

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

Se genera la clase que representa la operación:

```
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

Esta operación elimina un registro de una fuente de datos, típicamente una base de datos y retorna la cantidad de registros eliminados (que es de tipo `Integer`). Todos los campos de la operación se utilizan para determinar cuál o cuáles son los registros a eliminar. Esta operación es mucho más flexible que `@DeleteEntityById` ya que permite especificar los campos por los cuales se va buscar los registros a eliminar (pudiendo tener varios, que no necesariamente tengan que ser el identificador del registro).

Parámetros:

- **`related`**: indica cual es la entidad asociada al registro a eliminar.

Ejemplo: Para la definición de la operación

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

    Integer calendarId;
    String title;

}
```

Se genera la clase que representa la operación:

```
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;
    }
}
```

Información complementaria para la definición de la operación

Si se desea, es posible alterar algunos aspectos del código generado para conseguir que sea tratado de manera diferente mediante una serie de anotaciones complementarias

Información que se pueden añadir sobre la operación:

- **`abstract`**: si se marca la operación como abstracta provoca que la operación generada sea abstracta.
- **`extends`**: si se añade una clase base (usando `extends`) a la operación provoca que la operación generada extienda de esa clase base.
- **`implements`**: si se implementan interfaces (usando `implements`) en la entidad provoca que la operación generada implemente las interfaces indicadas.
- **`@Doc`**: si se añade esta anotación a la entidad provoca que el texto indicado en ella sea escrito como documentación de la operación generada. Esta anotación recibe un array de string, en donde cada item del array corresponde a una línea de la documentación.
- **`@Deprecated`**: si se añade esta anotación a la operación provoca que la entidad generada sea marcada como obsoleta.
- **`@Manually`**: si se añade esta anotación se indica al backend (típicamente la base de datos) que no se debe generar una implementación para esta operación, ya que la implementación será provista manualmente por el programador.
- Cualquier anotación que esté fuera del paquete de anotaciones de Uaithne (`org.uaithne.annotations.*`) será generado en la operación resultante.
- Si se desea que una operación no tenga resultado se debe de usar como tipo de resultado la clase `Void`, cuyo único valor válido es nulo.

Información que se puede añadir sobre los campos:

- **`@Optional`**: indica que el campo es opcional, por lo que admite nulo. Si un campo no posee la anotación `@Optional` se asume que es obligatorio, y en consecuencia no admite nulo.
- **`@ExcludeFromConstructor`**: indica que el campo no debe ser incluido en el constructor de la clase de la operación generada. Los campos marcados con `@Optional` son automáticamente excluidos del constructor de la clase de la operación generada.
- **`@ExcludeFromObject`**: indica que este campo no debe ser incluido en la clase de la operación generada, su existencia tiene otros propósitos y en ningún caso debe formar parte del código Java generado para la operación.
- **`@ExcludeFromToString`**: indica que este campo no debe ser incluido en la implementación del método `toString` de la clase generada.
- **`@DefaultValue`**: indica cual es el valor por defecto a ser asignado al campo durante la inicialización de la clase de la operación generada. El valor indicado con esta anotación debe entenderse que se traduce en una asignación del valor al campo antes que se ejecute el constructor.

  Ejemplo:

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

  Genera:

  ```
  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;
    }
}
```

La anotación `@DefaultValue` recibe un string que será interpretado en función del tipo de dato del campo, si el tipo es **`boolean` o `Boolean`** el texto debe ser `true` o `false`, si es un tipo **numérico** (`byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, `float`, `Float`, `double`, `Double`) el texto debe ser un número, si se trata de **`BigInteger` o `BigDecimal`** el texto es el que se va a usar para inicializar la nueva instancia de estas clases, si es **`char` o `Character`** el texto debe ser un carácter que será apropiadamente encomillado, si se trata de **`String`** el texto debe ser una secuencia de caracteres que será apropiadamente encomillado, si se trata de **cualquier otro tipo** el texto se tratará como el código Java requerido para realizar la inicialización.

- **`@Doc`**: si se añade esta anotación al campo provoca que el texto indicado en ella sea escrito como documentación de este en la operación generada. Esta anotación recibe un array de string, en donde cada item del array corresponde a una línea de la documentación.
- **`@Deprecated`**: si se añade esta anotación al campo provoca que la operación generada este sea marcado como obsoleto.
- **`@Manually`**: si se añade esta anotación se indica al backend (típicamente la base de datos) que no se debe generar ninguna lógica asociada a este campo, ya que la implementación será provista manualmente por el programador. Si este campo no debe ser usado en lo absoluto por la capa de acceso a datos ya que obtiene su valor en capas superiores, se debe establecer el a `true` la propiedad `onlyProgrammatically` de esta anotación.
- **transient**: se se marca el campo no serializable provoca que en la operación generada este campo sea marcado como no serializable.
- Cualquier anotación que esté fuera del paquete de anotaciones de Uaithne (org.uaithne.annotations.*) será generado en la operacion resultante.

Características de las operaciones

- Todas las operaciones generadas implementan directa o indirectamente la interfaz `Operation`.
- Todas las operaciones generadas reimplementan en método `toString` que retorna el nombre de la operación y el valor de cada uno de los campos, excepto aquellos campos que tengan la anotación `@ExcludeFromToString` o `@ExcludeFromObject`.
- Todas las operaciones generadas reimplementan el método `equals` y `hashCode`, en la nueva implementación de estos métodos se considera que dos operaciones son iguales si su tipo y el valor de cada uno de sus campos son iguales, son excluidos aquellos campos que tengan la anotación `@ExcludeFromObject` o que estén marcados como `transient`.
- Todas las operaciones generadas tienen un constructor sin argumentos.

- Todas las operaciones generadas tienen un constructor que tiene todos campos que sean obligatorios, es decir, aquellos que no tengan la anotación `@Optional`, son excluidos del constructor aquellos campos que tengan la anotación `@ExcludeFromObject`. Los campos en el constructor aparecen en el mismo orden en el cual aparecen declarados en la entrada del generador.
- Todas las operaciones generadas implementan cada uno de los métodos requeridos por el framework para su funcionamiento, es decir, implementa cada uno de los métodos de la interfaz `Operation` (solo en algunas configuraciones de Uaithne).
- Todas las operaciones generadas son instanciables al menos que sean definidas como abstractas en la entrada del generador, en consecuencia, para usar las operaciones basta con crear una nueva instancia de esta (no hace falta crear clases que extiendan de esta).

Operaciones generadas para una entidad

Si la entidad está definida dentro de un módulo se puede generar por defecto una serie de operaciones (la anotación `@Entity` permite indicar cuales se deben generar, también se puede activar que se generen por defecto en la configuración de Uaithne), en el nombre de la operación la cadena *`«Entity»`* se reemplaza por el nombre de la entidad. Si existe una operación de igual nombre no se genera la operación de la entidad con nombre coincidente.

- **`Insert`*`«Entity»`***: Inserta una entidad en la fuente de datos, retorna el Id del registro insertado. Esta operación es equivalente a la generada con `@InsertEntity`.
- **`Update`*`«Entity»`***: Actualiza una entidad en la fuente de datos, retorna la cantidad de registros afectados. Esta operación es equivalente a la generada con `@UpdateEntity`.
- **`Delete`*`«Entity»``ById`**: Elimina una entidad de la fuente de datos dado su id, retorna la cantidad de registros eliminados. Esta operación es equivalente a la generada con `@DeleteEntity`.
- **`Select`*`«Entity»``ById`**: Recupera una entidad de la fuente de datos dado su el id y retorna la entidad o nulo si no lo encuentra. Esta operación es equivalente a la generada con `@SelectEntityById`.
- **`Save`*`«Entity»`***: Inserta o actualiza una entidad en la fuente de datos, si el identificador de la entidad que recibe la operación es nulo inserta un nuevo registro, si no actualiza el registro con ese identificador, retorna el identificador del registro insertado. Esta operación es equivalente a la generada con `@SaveEntity`.
- **`Merge`*`«Entity»`***: Actualiza los campos de una entidad en la fuente de datos cuyos valores sean distintos de nulo. Esta operación actualiza el registro de la entidad cuyo identificador coincida con el indicado dentro de esta, esta operación solo actualiza los campos que posean valor en la entidad, es decir, aquellos cuyo valor sea diferente de nulo, los campos con valores nulos serán ignorados. Esta operación es equivalente a la generada con `@MergeEntity`.
- **`JustInsert`*`«Entity»`***: Inserta una entidad en la fuente de datos, retorna la cantidad de registros insertados sin tratar de recuperar el id del registro insertado. Esta operación es equivalente a la generada con `@InsertEntity` estableciendo como `false` el valor de la propiedad `returnLastInsertedId` de la anotación.
- **`JustSave`*`«Entity»`***: Inserta o actualiza una entidad en la fuente de datos en función de si el identificador de la entidad que recibe la operación es nulo inserta un nuevo registro, si no actualiza el registro con ese identificador; y retorna la cantidad de registros afectados sin tratar de tratar de recuperar el id del registro insertado. Esta operación es equivalente a la generada con `@SaveEntity` estableciendo como `false` el valor de la propiedad

`returnLastInsertedId` de la anotación.

Para poder generar las operaciones para una entidad es necesario que esté contenida dentro de un módulo y que tenga un único campo marcado con la anotación `@Id` definido por esta o por cualquier entidad de la que extienda directa o indirectamente.

Para poder generar las operaciones `Save` o `JustSave` es necesario que el tipo de datos del campo marcado con la anotación `@Id` sea un tipo de datos al que se le pueda asignar nulo (por lo que no puede ser `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, en lugar de ellos se puede usar `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` o cualquier otro objeto).

Generador de queries

Al utilizar el generador de código es posible indicar a Uaithne que genere la implementación de los ejecutores de acceso a la base de datos con las queries SQL incluidas, reduciendo así aún más el trabajo requerido por parte del programador para tener el backend del sistema funcionado.

Para generar la capa de acceso a base se genera la implementación del executor con el código Java necesario para invocar la base de datos mediante el uso de MyBatis y un fichero XML de MyBatis con las queries SQL a ser ejecutadas por cada operación.

Bases de datos soportadas:

- 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, 10.11, 10.12, 10.13, 10.14)
- Derby (10.6, 10.7, 10.8, 10.9, 10.10)
- MySql (5.5, 5.6, 5.7)

Para activar la generación de la capa de acceso a la base de datos basta con añadir la anotación `@MyBatisMapper` sobre la clase de definición del módulo.

```
@OperationModule
@MyBatisMapper
public class _events {

}
```

La inclusión de la anotación @MyBatisMapper provoca que se genere un fichero java con la implementación de acceso a MyBatis y el XML de mapeo de MyBatis por cada una de las base de datos que se haya incluido en la configuración de Uaithne.

Queries por defecto para cada operación

La idea central tras la generación de las queries es que para cada operación se genera una query por defecto que mediante anotaciones el programador puede ir indicándole al generador los cambios que requiera hasta conseguir la query deseada.

@Operation

Para esta operación no se genera ningún código, su implementación debe ser provista por el programador. Este mismo efecto se puede conseguir añadiendo la anotación `@Manually` sobre cualquier otra operación.

@SelectOne

En esta operación todos los campos de la entidad resultante conforman la cláusula `select` de la query y todos los campos de la operación conforman la cláusula `where` unidos entre ellos con el operador `and`, y si la operación tiene el parámetro `limit` a `true` se añade el código adicional necesario para que la query retorne únicamente el primer registro encontrado.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

Pero si la operación tuviese el parámetro `limit` a `true`:

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

Generaría para PostgreSQL la query:

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

@SelectMany

En esta operación todos los campos de la entidad resultante conforman la cláusula `select` de la query y todos los campos de la operación conforman la cláusula `where` unidos entre ellos con el operador `and`, y si la operación tiene el parámetro `distinct` a `true` se añade el modificador `distinct` a la cláusula `select`.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

```
select
```

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

Pero si la operación tuviese el parámetro `distinct` a `true`:

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

Generaría para PostgreSQL la query:

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

@SelectPage

En esta operación se usan dos queries distintas: una para contar la cantidad de registros y otra para obtener los datos de la página. Para contar la cantidad de registros se hace un `select count(*)` y se incluyen todos los campos de la operación para conformar la cláusula `where` unidos entre ellos con el operador `and`. La query para obtener los datos de la página contiene todos los campos de la entidad resultante en la cláusula `select` y todos los campos de la operación conforman la cláusula `where` unidos entre ellos con el operador `and`, se añade el código adicional necesario para que la query retorne únicamente los datos de la página solicitada. Si la operación tiene el parámetro `distinct` a `true` se añade el modificador `distinct` a la cláusula `select` de ambas queries.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query que cuenta la cantidad de registros:

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

También se genera la query que trae los datos de la página solicitada:

```
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>
```

Pero si la operación tuviese el parámetro `distinct` a `true`:

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

Generaría para PostgreSQL la query que cuenta la cantidad de registros:

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

También generaría la query que trae los datos de la página solicitada:

```
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

En esta operación se hace un `select count(*)` que incluye todos los campos de la operación en la cláusula `where` de la query unidos entre ellos con el operador `and`.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

@SelectEntityById

En esta operación todos los campos de la entidad resultante conforman la cláusula `select` de la query e incluye el campo `id` (que es añadido automáticamente a la operación) en la cláusula `where`, y si la operación tiene el parámetro `limit` a `true` se añade el código adicional necesario para que la query retorne únicamente el primer registro encontrado.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

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

Pero si la operación tuviese el parámetro `limit` a `true`:

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

Generaría para PostgreSQL la query:

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

@InsertEntity

En esta operación todos los campos de la entidad que contiene en el campo `value` (que es añadido automáticamente a la operación) que va a ser insertada, se especifican en el `insert` con sus respectivos valores, exceptuando los campos marcados con la anotación `@Id` que su presencia depende de la estrategia de generación del id.

Ejemplo: Para la definición de la operación (donde el id es autogenerado)

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

Se genera para PostgreSQL la query:

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

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

Nota: retornar el id del registro insertado puede requerir la ejecución de una segunda query dependiendo del manejador de la base de datos.

@UpdateEntity

En esta operación todos los campos de la entidad que contiene en el campo `value` (que es añadido automáticamente a la operación) se traducen en actualizaciones de campos en la tabla excepto los campos marcados con la anotación `@Id` que se utilizan en la cláusula `where` unidos entre ellos con el operador `and`.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

```
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

En esta operación se hace un `delete` donde la cláusula `where` incluye el campo `id` (que es añadido automáticamente a la operación).

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

@SaveEntity

En esta operación se usan dos queries distintas: una para insertar un registro y otro para actualizarlo, se inserta si la entidad que contiene en el campo `value` (que es añadido automáticamente a la operación) el campo anotado con `@Id` es nulo (por lo que se inserta un nuevo registro) o es no nulo (por lo que se actualiza el registro con el id suministrado).

Para la query de inserción se incluyen todos los campos de la entidad en el `insert` con sus respectivos valores, exceptuando los campos marcados con la anotación `@Id` que su presencia depende de la estrategia de generación del id.

Para la query de actualización se incluyen todos los campos de la entidad que se traducen en actualizaciones de campos en la tabla excepto los campos marcados con la anotación `@Id` que se utilizan en la cláusula `where` unidos entre ellos con el operador `and`.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query de inserción:

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

También se genera la query de actualización:

```
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}
```

Nota: retornar el id del registro insertado puede requerir la ejecución de una segunda query

dependiendo del manejador de la base de datos.

@MergeEntity

En esta operación todos los campos de la entidad que contiene en el campo `value` (que es añadido automáticamente a la operación) se traducen en actualizaciones de campos en la tabla, siempre y cuando su valor sea distinto de nulo, excepto los campos marcados con la anotación `@Id` que se utilizan en la cláusula `where` unidos entre ellos con el operador `and`.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

```
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

En esta operación todos los campos de la operación se especifican en el `insert` con sus respectivos valores.

Ejemplo: Para la definición de la operación (donde el id es autogenerado)

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

Se genera para PostgreSQL la query:

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

Nota: retornar el id del registro insertado puede requerir la ejecución de una segunda query dependiendo del manejador de la base de datos.

@Update

En esta operación todos los campos de marcados con la anotación `@SetValue` se traducen en actualizaciones de campos en la tabla, y los campos que no posean esa anotación se utilizan en la cláusula `where` unidos entre ellos con el operador `and`. Si se establece el parámetro `ignoreWhenNull` a `true` de la anotación `@SetValue` solamente se actualiza el campo si este tiene valor, de ser nulo este es ignorado.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

@Delete

En esta operación se hace un `delete` donde la cláusula `where` contiene todos los campos de la operación unidos entre ellos con el operador `and`.

Ejemplo: Para la definición de la operación

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

    Integer calendarId;
    String title;

}
```

Se genera para PostgreSQL la query:

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

Personalización de las queries generadas

Uaithne permite alterar las queries generadas usando para ello una serie de anotaciones complementarias que cambian el comportamiento por defecto del generador.

Anotaciones básicas:

- **`@MappedName`**: Permite indicar el nombre del campo o tabla en la base de datos.
- **`@Manually`**: Indica al generador de queries que ignore la operación o un campo.
- **`@ValueWhenNull`**: Indica al generador de queries que use el valor indicado en esta anotación cuando el valor dado sea nulo.
- **`@ForceValue`**: Indica al generador de queries que use el valor dado en lugar del suministrado.

Anotaciones que solo afectan la inserción:

- **`@HasDefaultValueWhenInsert`**: Indica al generador de queries que este campo tiene un valor por defecto en la base de datos, por lo que si el valor dado del campo es nulo debe ser usado el valor por defecto de la base de datos en la query de `insert`.

Anotaciones que permiten adaptar el `where`:

- **`@Optional`**: Indica al generador que el valor de este campo es opcional, por lo que debe ser tratado apropiadamente, por ejemplo, solo debe ser tenido en cuenta en el `where` si su valor es distinto de nulo.
- **`@Comparator`**: Permite indicar al generador de queries cual es la regla de comparación de un campo con el valor en la base de datos, por defecto, si no se usa esta anotación se compara por igualdad, pero si el tipo de dato del campo es una lista comprueba que el valor de la columna esté en la lista.
- **`@CustomComparator`**: Permite especificar al generador de queries una regla de comparación compleja a ser usada para comparar el valor del campo con el de la base de datos.

Anotación para ordenar los datos:

- **`@OrderBy`**: Indica al generador que este campo es una cláusula de ordenamiento de los registros resultantes por la consulta.

@MappedName

Esta anotación permite especificar el nombre en la base de datos de una entidad o campo.

En una entidad Uaithne usa por defecto como nombre de tabla el nombre de la entidad, pero si se añade esta anotación sobre una entidad (o vista de entidad) se usará el valor de esta anotación como nombre de la tabla en la base de datos.

En un campo, ya sea de una entidad (o vista de entidad) o de una operación, Uaithne usa por defecto el nombre del campo como nombre de la columna en la base de datos, pero si se añade esta anotación sobre el campo se usará el valor de esta anotación como nombre de la columna en la base de datos.

Ejemplo: Para la definición de la entidad

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

Se genera para la operación de `@SelectEntityById` en PostgreSQL la query:

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

Tip: el valor suministrado en la anotación `@MappedName` representa un fragmento SQL, por lo que es posible por ejemplo que para una entidad el valor suministrado sea un `JOIN` o para un campo sea un fragmento SQL que calcula en valor; siempre que tenga sentido para las operaciones.

Ejemplo: Para la definición de la entidad

```
@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;
}
```

Se genera para la operación de `@SelectEntityById` en PostgreSQL la query:

```
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

Esta anotación permite indicar al generador de acceso a la base de datos que es el programador quién se va a hacer cargo del elemento anotado, por lo que debe ser ignorado durante la generación.

Si se coloca sobre una operación indica a Uaithne que no se genere el acceso a la base de datos para la operación en cuestión, por lo que la implementación en java de esta operación debe ser provista por el programador.

Si se coloca sobre un campo, ya sea de una entidad (o vista de entidad) o de una operación, provoca que el campo en cuestión sea ignorado en la query generada (como si no existiera).

Parámetros:

- **onlyProgrammatically**: Indica al generador que este campo nunca va a ser usado en una query SQL (solo tiene utilidad en la generación experimental de procedimientos almacenados). Valor por defecto: `false`.

Tip: Es posible marcar un campo como `@Manually` y luego usarlo en un fragmento de query SQL que sea suministrado por el programador, permitiendo de esta forma hacer queries más complejas de las inicialmente soportadas por Uaithne.

@ValueWhenNull

Esta anotación permite indicar al generador que use el valor suministrado en esta anotación en la query cuando el campo (de una entidad o de una operación) sobre el cual se aplica vale nulo.

Nota: Esta anotación requiere que el campo sobre el que se aplica admita nulo, por lo que debe tener también la anotación `@Optional`.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

Tip: el valor suministrado en la anotación `@ValueWhenNull` representa un fragmento SQL, por lo que es posible, por ejemplo, que el fragmento SQL sea la llamada a una función SQL que retorna

el valor a ser usado finalmente.

@ForceValue

Esta anotación permite indicar al generador que use el valor suministrado en esta anotación independientemente del valor que tenga el campo (de una entidad o de una operación).

Nota: Esta anotación es similar a `@ValueWhenNull` solo que el valor se va a aplicar siempre, independientemente si el valor del campo es nulo o no, tampoco requiere que el campo posea la anotación `@Optional`.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

Tip: el valor suministrado en la anotación `@ForceValue` representa un fragmento SQL, por lo que es posible, por ejemplo, que el fragmento SQL sea la llamada a una función SQL que retorna el valor a ser usado finalmente.

@HasDefaultValueWhenInsert

Esta anotación permite indicar que el campo de una entidad (o vista de entidad) tiene un valor por defecto en la base de datos, por lo que al insertar el registro, si el valor de este campo es nulo se debe usar el valor por defecto de la base de datos.

Ejemplo: Para la definición de la entidad

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

Se genera para la operación de `@InsertEntity` en PostgreSQL la query:

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

@Optional

Esta anotación le indica a Uaithne que el valor de un campo es opcional, por lo que admite nulo, si se utiliza en los campos de una operación indica que ese campo debe ser ignorado cuando es nulo si el uso del campo condiciona el resultado de la operación, es decir, cuando este campo forma parte del `where` o del `order by` de la query.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

```
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

Esta anotación permite especificar a Uaithne como debe comparar el valor del campo de una operación con el valor de la columna en la base de datos cuando el campo se usa como parte del `where` de la query. Por defecto, ni no se suministra esta anotación la comparación se hace por igualdad, pero si el tipo de dato del campo es una lista comprueba que el valor de la columna esté en la lista.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

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

Valores posibles:

- **`EQUAL`**: El valor de la columna debe ser igual al valor del campo. SQL:

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

- **`NOT_EQUAL`**: El valor de la columna no debe ser igual al valor del campo. SQL:

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

- **`EQUAL_INSENSITIVE`**: El valor de la columna debe ser igual al valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **`NOT_EQUAL_INSENSITIVE`**: El valor de la columna no debe ser igual al valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **`SMALLER`**: El valor de la columna debe ser menor que el valor del campo. SQL:

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

- **`LARGER`**: El valor de la columna debe ser mayor que el valor del campo. SQL:

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

- **`SMALL_AS`**: El valor de la columna debe ser menor o igual que el valor del campo. SQL:

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

- **LARGER_AS**: El valor de la columna debe ser mayor o igual que el valor del campo. SQL:

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

- **IN**: El valor de la columna debe estar contenido en la lista suministrada como valor del campo. SQL:

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

- **NOT_IN**: El valor de la columna no debe estar contenido en la lista suministrada como valor del campo. SQL:

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

- **LIKE**: El valor de la columna debe ser similar al patrón dado como valor del campo. SQL:

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

- **NOT_LIKE**: El valor de la columna no debe ser similar al patrón dado como valor del campo. SQL:

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

- **LIKE_INSENSITIVE**: El valor de la columna debe ser similar al patrón dado como valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **NOT_LIKE_INSENSITIVE**: El valor de la columna no debe ser similar al patrón dado como valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **START_WITH**: El valor de la columna debe empezar por el patrón dado como valor del campo. SQL:

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

- **NOT_START_WITH**: El valor de la columna no debe empezar por el patrón dado como valor del campo. SQL:

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

- **END_WITH**: El valor de la columna debe terminar por el patrón dado como valor del campo. SQL:

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

- **NOT_END_WITH**: El valor de la columna no debe terminar por el patrón dado como valor del campo. SQL:

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

- **START_WITH_INSENSITIVE**: El valor de la columna debe empezar por el patrón dado como

valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **NOT_START_WITH_INSENSITIVE**: El valor de la columna no debe empezar por el patrón dado como valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **END_WITH_INSENSITIVE**: El valor de la columna debe terminar por el patrón dado como valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **NOT_END_WITH_INSENSITIVE**: El valor de la columna no debe terminar por el patrón dado como valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **CONTAINS**: El valor de la columna debe contener el patrón dado como valor del campo. SQL:

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

- **NOT_CONTAINS**: El valor de la columna no debe contener el patrón dado como valor del campo. SQL:

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

- **CONTAINS_INSENSITIVE**: El valor de la columna debe contener el patrón dado como valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

- **NOT_CONTAINS_INSENSITIVE**: El valor de la columna no debe contener el patrón dado como valor del campo ignorando la distinción entre mayúsculas y minúsculas. SQL:

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

Nota: En Sql Server se reemplaza `||` por + como operador de concatenación, y en PostgreSql se utiliza `ilike` en lugar de la combinación de `lower` y `like`.

@CustomComparator

Esta anotación permite especificar a Uaithne como debe comparar el valor del campo de una operación con el valor de la columna en la base de datos cuando el campo se usa como parte del `where` de la query. La diferencia con `@Comparator` es que `@CustomComparator` permite especificar la regla de comparación en lugar de usar una predefinida.

El patrón de comparación corresponde a un fragmento SQL con unas secuencias incrustadas que permiten incluir cierta información de la columna o el campo en el código SQL final, esta secuencia debe estar contenida entre `[[` y `]]`, por ejemplo, `[[value]]` es sustituido por el valor del campo en el SQL generado.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query:

```
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}
```

Secuencias incrustadas posibles:

- **`column`**
- `COLUMN`

  Nombre de la columna en la base de datos que se haya especificado con `@MappedName`, de no tener retorna el nombre del campo en el objeto Java.

- **`name`**
- `NAME`

  Nombre del campo en el objeto Java.

- **`value`**
- `VALUE`

  Valor del campo en el objeto Java.

  Esta secuencia toma el valor de la anotación `@ForceValue`, de no tener el campo esta anotación, toma el valor de la anotación `@ValueWhenNull`, y de no tener esta otra, toma el valor del campo, equivaliendo en este último caso a:

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

- **`jdbcType`**
- `JDBC_TYPE`

  Tipo de dato JDBC del campo.

El valor de esta secuencia siempre empieza por `,jdbcType=` seguido del nombre del tipo JDBC del campo que se lee de la anotación `@JdbcType`, de no tener se aplica el que mejor corresponda al tipo de dato del campo.

- **typeHandler**
- TYPE_HANDLER

  TypeHandler de MyBatis si se especificó uno para el campo usando la anotación `@MyBatisTypeHandler`

  El valor de esta secuencia siempre empieza por `,typeHandler=` seguido del nombre completo de la clase `TypeHandler` especificada en el campo con la anotación `@MyBatisTypeHandler`, de no tener equivale a un string vacío.

- **valueNullable**
- VALUE_NULLABE

  Valor del campo en el objeto Java ignorando la anotación `@ValueWhenNull`

- **valueWhenNull**
- VALUE_WHEN_NULL

  Valor de la anotación `@ValueWhenNull`

- **unforcedValue**
- UNFORCED_VALUE

  Valor del campo en el objeto Java ignorando la anotación `@ForceValue`

- **unforcedNullbaleValue**
- UNFORCED_NULLABLE_VALUE

  Valor del campo en el objeto Java ignorando las anotaciones `@ForceValue` y `@ValueWhenNull`

- **forcedValue**
- FORCED_VALUE

  Valor de la anotación `@ForceValue`

@OrderBy

Esta anotación marca uno o varios campos de una operación como parte de la cláusula order by de un select, por lo que se puede usar en las operaciones `@SelectOne`, `@SelectMany` y `@SelectPage`.

Un campo marcado con la anotación `@OrderBy` debe ser de tipo string y su valor debe ser el contenido de la cláusula `order by` de la query, pero con la diferencia de que se debe usar el nombre de los campos de la entidad resultante en lugar del nombre de las columnas en la base de datos, Uaithne traduce automáticamente el `order by` antes de ejecutarlo en la base de datos a su correspondiente SQL, impidiendo en el proceso cualquier riesgo de SQL Injections y a la vez escondiendo los detalles de la base de datos a las capas superiores.

Ejemplo: Para la definición de entidad y la operación

```
@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;
}
```

Se genera para PostgreSQL la query:

```
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}
```

La propiedad `orderResultBy` de la operación puede recibir como string un listado separado por coma de los nombre de los campos de la entidad `Event` resultante, y cada uno de ellos puede tener el modificador `asc` o `desc`, sin hacer distinción entre mayúsculas y minúsculas. El valor de campo es traducido a su correspondiente versión SQL antes de que se llame al MyBatis, por lo que `${orderResultBy}` contendrá la versión SQL de la cláusula SQL suministrada a la operación.

Ejemplo de valores válidos para el campo `orderResultBy`:

- `title` que es traducido a SQL como `column_title`
- `title desc` que es traducido a SQL como `column_title desc`
- `title asc` que es traducido a SQL como `column_title asc`

- `start desc, title` que es traducido a SQL como `column_start desc, column_title`

Personalización de la generación del identificador de registros

Uaithne soporta múltiples estrategias para la generación del identificador de un registro en la base de datos, partiendo de una estrategia base especificada en la configuración Uaithne que puede ser campos autogenerados a usar secuencias con patrones de nombre predefinidos, a poder controlar, con ayuda de una serie de anotaciones poder controlar caso a caso cómo manejar la generación del identificador del registro.

Limitación: Uaithne solo soporta un único campo identificador de por entidad para poder generar automáticamente las queries y operaciones que reciben o retornan el identificador. Para eludir esta limitación es posible utilizar varias estrategias:

- En **tablas con** claves compuestas, pero dónde un campo que conforma la clave primaria en la base de datos puede actuar como **clave alterna**: en estos casos basta con usar en Uaithne la clave alterna como identificador, teniendo así un único identificador a efectos prácticos. Esta estrategia es útil cuando, por diseño, se desea utilizar claves compuestas en la base de datos pero la tabla posee su propio identificador único.
- En las tablas con claves compuesta, **donde no sea posible aplicar la estrategia anterior** se debe evitar usar operaciones que usen automáticamente el identificador del registro, es decir, no se pueden usar las operaciones que hacen Insert, Update, Delete, Save, Merge de entidades, en su lugar hay que usar las operaciones de Insert, Update y Delete no restringidas a entidades, es decir, aquellas que se construyen con las anotaciones `@Insert`, `@Update` y `@Delete`.

Anotaciones que permiten controlar la generación del identificador:

- **`@Id`**: Esta anotación indica cuál es el campo identificador. Si se establece el parámetro `autogenerated` a `false` indica que este identificador no es autogenerado, por ende su valor debe ser suministrado al momento de insertar el registro.
- **`@IdSequenceName`**: Esta anotación indica al generador que para la generación del identificador se debe usar la secuencia de nombre indicado como valor de la anotación.
- **`@IdQueries`**: Esta anotación indica que para la generación del identificador se debe usar las queries suministradas.

Estrategias de generación del identificador de un registro soportadas por Uaithne:

- **Manual**: En la cual el identificador debe ser provisto manualmente antes de insertar el registro en la base de datos como valor del campo en la entidad. Para activar este modo se debe establecer a `false` la propiedad `autogenerated` de la anotación `@Id`.
- **Columnas autoincrementales en la base de datos**: En la cual el identificador es generado automáticamente por el tipo de datos de la columna en la base de datos. Este es el modo por defecto en las base de datos con soporte a tipos de datos autoincrementales (se puede cambiar en la configuración de Uaithne). Nota: es posible emular este comportamiento en la base de datos que no lo soportan mediante 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.
- **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`).

@Id

Esta anotación permite especificar a Uaithne que el campo de una entidad (o vista de entidad) sobre el cual se aplica es el identificador del registro.

Parámetros:

- **autogenerated**: Indica si el campo es autogenerado.Si se establece a `false` (por defecto su valor es `true`) indica a Uaithne que el identificador del registro va a ser el valor que tenga el campo, por lo que el valor debe ser incluido en la query de insersión y debe ser suministrado antes de que se ejecute el insert.

@IdSecuenceName

Esta anotación permite especificar a Uaithne que utilice la secuencia de nombre especificado como parámetro para generar el identificador del registro.

Nota:

Esta anotación se puede colocar tanto sobre el campo como sobre la entidad, al colocarlo sobre la entidad modifica el comportamiento del campo anotado con `@Id` incluso si este se encuentra en una clase base.

@IdQueries

Esta anotación permite especificar a Uaithne el código SQL para generar el siguiente identificador en el momento de realizar la inserción y la query SQL para recuperar el identificador del registro insertado.

Parámetros:

- **selectNextValue**: Indica cual el el fragmento SQL a ser incrustado en la query de inserción para generar el identificador del registro a ser insertado.
- **selectCurrentValue**: Indica cual es la query SQL que se debe de ejecutar para recuperar el valor del registro recién insertado.

Nota:

Esta anotación se puede colocar tanto sobre el campo como sobre la entidad, al colocarlo sobre la entidad modifica el comportamiento del campo anotado con `@Id` incluso si este se encuentra en una clase base.

Personalización del código MyBatis

Uaithne utiliza MyBatis para realizar el acceso a la base de datos, y en cualquier sitio donde el generador permita especificar un fragmento o una query SQL es posible incrustar instrucciones de MyBatis (no permitido en la generación experimental de procedimientos almacenados).

Nombres de los columnas:

Al hacer un `select` la transformación del resultado en objetos Java se hace mediante el nombre de la columna, por lo que es necesario que el nombre de la columna en el resultado de la query sea el mismo que el nombre de la propiedad en el objeto que recibe el valor, si los nombres no coinciden es necesario incluir la cláusula `as` para renombrar el campo en la query.

Caracteres especiales:

- **`>`**: Uaithne transforma automáticamente el carácter `>` en `>` para que las queries escritas como entradas de Uaithne sean válidas en el XML de MyBatis.
- **`<`**: Uaithne transforma automáticamente el carácter `<` en `<` para que las queries escritas como entradas de Uaithne sean válidas en el XML de MyBatis.
- **`{[`**: Si se desea escribir el carácter `<` como parte de un fragmento XML se debe sustituir por `{[`. Ejemplo: Para escribir la etiqueta XML `<set>` en el XML generado de MyBatis es necesario escribir `{[set]}` en la entrada de Uaithne.
- **`]}`**: Si se desea escribir el carácter `>` como parte de un fragmento XML se debe sustituir por `]}`. Ejemplo: Para escribir la etiqueta XML `</set>` en el XML generado de MyBatis es necesario escribir `{[/set]}` en la entrada de Uaithne.

Anotaciones que permite controlar el código MyBatis:

- **@JdbcType**: Permite especificar el tipo de dato JDBC del campo.
- **@MyBatisTypeHandler**: Permite especificar para un campo cuál es el TypeHandler encargado de transformar el valor del objeto java en el valor que recibe la base de datos. Limitación: Esta anotación no afecta el proceso de conversión del resultado de la base de datos en en valor a recibir el objeto Java.
- **@MyBatisCustomSqlStatementId**: Permite especificar el `statementId` a ser ejecutado por la operación que recibe la anotación. El uso de esta anotación provoca que Uaithne no genere la query en el XML de MyBatis y provoca que el código Java invoque el `statementId` suministrado (en lugar del que se generaría de no usar esta anotación).

@JdbcType

Permite especificar a Uaithne el tipo de datos JDBC que se debe usar, para el campo sobre el cual se aplica la anotación, en los XML de MyBatis, si no se especifica ninguno se le asigna uno automáticamente en función de tipo (se puede cambiar en la configuración de Uaithne), tanto cuando es el tipo en sí mismo como cuando es argumento genérico de los tipos `List` o `ArrayList` o como tipo de un array:

- **Varchar**: Para el tipo `java.lang.String`
- **Timestamp**: Para los tipos `java.util.Date`, `java.sql.Timestamp`
- **Date**: Para el tipo `java.sql.Date`
- **Time**: Para el tipo `java.sql.Time`
- **Boolean**: Para los tipos `boolean`, `java.lang.Boolean`

- **Tinyint**: Para los tipos `byte`, `java.lang.Byte`
- **Smallint**: Para los tipos `short`, `java.lang.Short`
- **Integer**: Para los tipos `int`, `java.lang.Integer`
- **Bigint**: Para los tipos `long`, `java.lang.Long`
- **Real**: Para los tipos `float`, `java.lang.Float`
- **Double**: Para los tipos `double`, `java.lang.Double`
- **Numeric**: Para los tipos `java.math.BigDecimal`, `java.math.BigInteger`
- **Char**: Para los tipos `char`, `java.lang.Character`

@MyBatisTypeHandler

Permite especificar a Uaithne que use, para el campo sobre el cual se aplica la anotación, el TypeHandler especificado como parámetro en los XML de MyBatis.

Limitación:

Esta anotación solo afecta a la conversión del valor desde Java a Sql pero no el contrario, si se desea que afecte también en el sentido contrario se debe especificar un TypeHandler global en la configuración de MyBatis.

@MyBatisCustomSqlStatementId

Permite especificar a Uaithne que para la operación sobre la cual se aplica esta anotación se use el `statementId` de MyBatis suministrado, esto provoca que no se genere para esta operación una entrada en el XML de MyBatis con la query, sino que se usa el `statementId` suministrado y que debe ser incluido en la configuración de MyBatis manualmente añadiendo otro fichero XML que lo contenga.

Parámetros:

- **`value`**: `StatementId` a ser usado por la operación.
- **`countSatementId`**: Para las operaciones `@SelectPage` este parámetro permite especificar el `statementId` que retorna la cantidad de registros existentes, para esta operación el parámetro `value` debe tener el statementId retorna el contenido de la página seleccionada.
- **`saveInsertStatementId`**: Para las operaciones `@Save` este parámetro permite especificar el `statementId` que inserta el registro, para esta operación el parámetro value debe tener el `statementId` que actualiza el registro.

Accediendo a campos desde fragmentos SQL

Uaithne permite acceder al valor de cualquier campo en cualquier fragmento o query SQL, aumentando así la flexibilidad de la query generada.

El patrón de acceso a un campo debe ser una secuencia de caracteres contenida entre `{{` y `}}`, por ejemplo, `{{field}}` es sustituido por el valor del campo en el SQL generado. Esta secuencia de caracteres puede estar compuesta de hasta tres partes (como mínimo debe tener una), cada parte debe estar separada por el carácter `:` quedando el formato de la siguiente forma:

`{{field}}`

`{{field:rule}}`

`{{field:rule:arg}}`

Dónde:

- **`field`**: corresponde al nombre del campo en la entrada de Uaithne.
- **`rule`**: indica la regla ser usada para generar el código SQL de referencia al campo. Si no se especifica ninguna regla es lo mismo que se fuese especificado la regla `value`.
- **`arg`**: permite especificar para algunas reglas un argumento a ser usada por ésta para generar el código SQL de acceso al campo.

Ejemplo:

Si se desea acceder al valor del campo `title` basta con escribir `{{title}}` en el SQL que recibe Uaithne para que sea transformado en a `#{title,jdbcType=VARCHAR}` en el XML de MyBatis. También se puede escribir `{{title:value}}` para conseguir el mismo resultado.

Reglas de acceso básicas:

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

  Genera la regla de comparación del campo tal como tal como se ha definido en la operación, es decir, usa comparación por igualdad o `IN`, pero si se el campo contiene la anotación `@Comparator` o `@CustomComparator` se genera la regla especificada.

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

  Esta regla permite acceder al mismo comportamiento que la anotación `@CustomComparator`, recibiendo como argumento el patrón de comparación. Ejemplo:

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

  genera el código:

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

- **`ifNull`**
- `IF_NULL`

  Esta regla genera la apertura de un `if` que verifica si el valor es nulo. Ejemplo:

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

  genera el código:

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

- **ifNotNull**
- IF_NOT_NULL

  Esta regla genera la apertura de un `if` que verifica si el valor es no nulo. Ejemplo:

  `{{field:ifNotNull}}`

  genera el código:

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

- **endIf**
- END_IF

  Esta regla genera el cierre de un `if`. Ejemplo:

  `{{field:endIf}}`

  genera el código:

  ```
  </if>
  ```

Reglas de acceso a las secuencias incrustadas de comparación

También se puede especificar como regla de acceso cualquiera de las secuencias incrustadas disponibles en la anotación `@CustomComparator` sin usar los caracteres `[[` o `]]`. Estas son:

- **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

Reglas de acceso a los comparadores predefinidos:

También se puede especificar como regla de acceso cualquier comparador permitido en la anotación `@Comparator` o algunos de sus alias, generando la regla de comparación para el campo con la columna en la base de datos.

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

  Equivale al comparador `EQUAL` de la anotación `@Comparator`.

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

  Equivale al comparador `NOT_EQUAL` de la anotación `@Comparator`.

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

  Equivale al comparador `EQUAL_INSENSITIVE` de la anotación `@Comparator`.

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

  Equivale al comparador `NOT_EQUAL_INSENSITIVE` de la anotación `@Comparator`.

- <
- smaller
- **SMALLER**

  Equivale al comparador `SMALLER` de la anotación `@Comparator`.

- >

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

  Equivale al comparador `LARGER` de la anotación `@Comparator`.

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

  Equivale al comparador `SMALL_AS` de la anotación `@Comparator`.

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

  Equivale al comparador `LARGER_AS` de la anotación `@Comparator`.

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

  Equivale al comparador `IN` de la anotación `@Comparator`.

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

  Equivale al comparador `NOT_IN` de la anotación `@Comparator`.

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

  Equivale al comparador `LIKE` de la anotación `@Comparator`.

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

  Equivale al comparador `NOT_LIKE` de la anotación `@Comparator`.

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

  Equivale al comparador `LIKE_INSENSITIVE` de la anotación `@Comparator`.

- `notIlike`
- `NOT_ILIKE`
- `notLikeInsensitive`
- **`NOT_LIKE_INSENSITIVE`**

  Equivale al comparador `NOT_LIKE_INSENSITIVE` de la anotación `@Comparator`.

- `startWith`
- **`START_WITH`**

Equivale al comparador `START_WITH` de la anotación `@Comparator`.

- `notStartWith`
- **`NOT_START_WITH`**

Equivale al comparador `NOT_START_WITH` de la anotación `@Comparator`.

- `endWith`
- **`END_WITH`**

Equivale al comparador `END_WITH` de la anotación `@Comparator`.

- `notEndWith`
- `NOT_END_WITH`

Equivale al comparador `NOT_END_WITH` de la anotación `@Comparator`.

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

Equivale al comparador `START_WITH_INSENSITIVE` de la anotación `@Comparator`.

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

Equivale al comparador `NOT_START_WITH_INSENSITIVE` de la anotación `@Comparator`.

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

Equivale al comparador `END_WITH_INSENSITIVE` de la anotación `@Comparator`.

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

Equivale al comparador `NOT_END_WITH_INSENSITIVE` de la anotación `@Comparator`.

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

Equivale al comparador `CONTAINS` de la anotación `@Comparator`.

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

Equivale al comparador `NOT_CONTAINS` de la anotación `@Comparator`.

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

  Equivale al comparador `CONTAINS_INSENSITIVE` de la anotación `@Comparator`.

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

  Equivale al comparador `NOT_CONTAINS_INSENSITIVE` de la anotación `@Comparator`.

Personalizando las cláusulas de las queries con @CustomSqlQuery

Mediante el uso de la anotación `@CustomSqlQuery` Uaithne permite modificar parcial o totalmente la query generada de la operación sobre la cual se aplica, o añadir fragmentos SQL en la cláusula deseada. El nombre de los parámetro de la anotación corresponde al nombre de la cláusula a ser reemplazada, pero existe la posibilidad de añadir código antes o después de la cláusula mediante el uso de los parámetros cuyo nombre empieza por `before` y `after` seguido del nombre de la cláusula.

Las cláusulas soportadas son:

- **query**: que reemplaza toda la query generada por la suministrada.
- **select**: que reemplaza el contenido de la cláusula `select` de una query select.
- **from**: que reemplaza la cláusula from de la query, ya sea en una query de select, insert, update, o delete. Este punto refiere al nombre de la tabla en la que se realiza la acción.
- **where**: que reemplaza el contenido (o añade una si no tiene) el contenido de la cláusula `where` de una query de select, update o delete.
- **groupBy**: añade la cláusula `group by` de una query de select. En este fragmento SQL se puede incluir la subcláusula `having` si el `group by` requiere una.
- **orderBy**: que reemplaza el contenido (o añade una si no tiene) la cláusula `order by` de una query de select.
- **insertInto**: que reemplaza el contenido de la cláusula `into` de una query de insert. Este punto refiere a los elementos que se colocan dentro del primer par de paréntesis al escribir `insert into table (...) values (...)`
- **insertValues**: que reemplaza el contenido de la cláusula `values` de una query de insert. Este punto refiere a los elementos que se colocan dentro del segundo par de paréntesis al escribir `insert into table (...) values (...)`
- **updateSet**: que reemplaza el contenido de la cláusula `set` de una query de update. Este punto refiere al listado de campos a ser actualizados y su respectivos valores.

Otros parámetros de `@CustomSqlQuery`:

- **tableAlias**: permite añadir un alias a la tabla en la query, lo que provoca que que añada el alias en el `from` y en cada lugar donde se use el nombre de la columna de un campo se prefije con el alias seguido con un punto.

- **excludeEntityFields**: permite especificar una lista con el nombre de los campos que se desea que sean ignorados durante la generación de la query.
- **isProcedureInvocation**: permite especificar si se trata de una invocación a un procedimiento almacenado, por lo que se debe marcar como `CALLABLE` en el XML de MyBatis, si no se le de ningún valor el tipo se detecta automáticamente el función del tipo de operación (`TRUE` para las operaciones especiales de procedimiento almacenado, `FALSE` para el resto).

Para mostrar cómo cada parámetro altera el resultado de la query generada se va a usar la siguientes queries (se omite el tipo de dato JDBC para facilitar la lectura):

```
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}
```

En la query modificada se va a colocar entre `[` y `]` en contenido de la query que se ve reemplazado por el valor del parámetro, y si se usa `[...]` es porque el valor se añade en ese punto. Se resalta en rojo los cambios realizados en la query. Las queries tachadas es porque no aplica.

query

Sustituye la query entera por el valor suministrado.

```
[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

Añade el valor suministrado al inicio de la 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

Añade el valor suministrado al final de la 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

Sustituye el select de la query por el valor suministrado.

```
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

Añade el valor suministrado al inicio del 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

Añade el valor suministrado al final del 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

Sustituye el from de la query por el valor suministrado.

```
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

Añade el valor suministrado al inicio del 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

Añade el valor suministrado al final del 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

Sustituye el where de la query por el valor suministrado.

```
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

Añade el valor suministrado al inicio del 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

Añade el valor suministrado al final del 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

Añade el valor suministrado como group by a la 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

Añade el valor suministrado al inicio del group by, si no tiene un group by añade uno.

```
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

Añade el valor suministrado al final del group by, si no tiene un group by añade uno.

```
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

Sustituye el order by de la query por el valor suministrado, si no tiene un order by añade uno.

```
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

Añade el valor suministrado al inicio del order by, si no tiene un order by añade uno.

```
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

Añade el valor suministrado al final del order by, si no tiene un order by añade uno.

```
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

Sustituye el into de la query de insert por el valor suministrado.

```
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

Añade el valor suministrado al inicio del into en el 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

Añade el valor suministrado al final del into en el 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

Sustituye el values de la query de insert por el valor suministrado.

```
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

Añade el valor suministrado al inicio del values en el 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

Añade el valor suministrado al final del values en el 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

Sustituye el set de la query de update por el valor suministrado.

```
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

Añade el valor suministrado al inicio del set en el 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

Añade el valor suministrado al final del set en el 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

Añade el valor suministrado como alias del nombre de la tabla especificado en la entidad, y prefija el acceso a las columnas manejadas por la entidad con este valor seguido con un punto.

```
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

Ignora durante la generación de la query los campos de nombre suministrado como una lista a esta anotación que provengan de la entidad. Si en el ejemplo siguiente se manda a ignorar el campo de nombre title el resultado sería (se muestra tachado lo que no se incluiría en la 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

Permite indicar al generador que la query se trata de una llamada a un procedimiento almacenado

y por ende se debe marcar como `CALLABLE` el `statementType` en el XML de MyBatis. Si su valor es `TRUE` indica que se debe marcar como `CALLABLE`, si es `FALSE` indica que no se debe marcar como `CALLABLE`, si no se especifica valor o se especifica `UNSPECIFIED` indica que se detecta automáticamente en función del tipo de operación (`TRUE` para las operaciones especiales de procedimiento almacenado, `FALSE` para el resto). **Nota**: este parámetro no altera de ninguna manera la 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}
```

Accediendo al contexto de la aplicación desde SQL

Es posible especificar en la configuración de Uaithne la entidad que representa el contexto de ejecución de la aplicación y acceder a ella desde cualquier fragmento o query SQL que reciba Uaithne o MyBatis.

El contexto de la aplicación es una entidad que contiene valores específicos a la ejecución actual, como por ejemplo, el login del usuario, esta información es complementaria a la operación pero no forma parte de esta, y normalmente se usa para almacenar información relativa al usuario, que es independiente de la acción que esté ejecutando.

Para acceder al contexto de la aplicación basta con escribir `_app.` seguido del nombre del campo en la entidad de contexto usando la misma sintaxis de acceso al valor de un campo de una entidad u operación, es decir usando `{{` y `}}` en Uaithne o `#{` y `}` en MyBatis.

Ejemplo: Para la definición de la entidad que se ha configurado como la entidad que representa el contexto

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

Para acceder al valor de la propiedad login en el contexto de ejecución de la aplicación hay que escribir:

Usando la sintaxis de acceso a campos de Uaithne:

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

Usando la sintaxis de acceso a campos de MyBatis:

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

El uso del contexto de ejecución de la aplicación se puede hacer desde cualquier fragmento o query SQL, e incluso desde cualquier XML de MyBatis escrito manualmente, y siempre debe ser de solo lectura, suele ser bastante útil si se utiliza en las anotaciones `@DefaultValueWhenNull` o `@ForceValue`.

Invocación de procedimientos almacenados sencillos

La invocación de procedimientos almacenados simples, que solo tienen parámetros de entrada, pero no poseen parámetros de salida se crean utilizando las operaciones ya antes descritas que mejor defina la lógica del procedimiento almacenado y utilizando la anotación `@CustomSqlQuery` para indicar la query de invocación del procedimiento almacenado o la anotación `@MyBatisCustomSqlStatementId` para indicar el `statementId` de MyBatis que contiene la query de invocación del procedimiento almacenado.

Operaciones típicamente usadas para invocar procedimientos almacenados son:

- **`@Insert`**: Permite invocar procedimientos almacenados que realizan cambios en la base de datos, pudiendo ser su acción principal, entre otras cosas, insertar uno o varios registros en una o varias tablas.
- **`@Update`**: Permite invocar procedimientos almacenados que realizan cambios en la base de datos, pudiendo ser su acción principal, entre otras cosas, actualizar uno o varios registros en una o varias tablas.
- **`@Delete`**: Permite invocar procedimientos almacenados que realizan cambios en la base de datos, pudiendo ser su acción principal, entre otras cosas, eliminar uno o varios registros en una o varias tablas.

Uso de la anotación `@CustomSqlQuery`:

- **`query`**: donde se coloca la query de invocación del procedimiento almacenado, que suele lucir como (dependiendo del manejador de base de datos la sintaxis es diferente):
    - **Mayoría base de datos si se marca `isProcedureInvocation` a `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`**: opcional, si se establece a `True` indica que la query generada se debe invocar usando un `CallableStatement` de JDBC. Al generar el código con MyBatis si se establece esta propiedad a `TRUE` provoca que en la query generada se

coloque como `statementType` el tipo `CALLABLE`. El valor por defecto es `FALSE` excepto para las queries complejas de acceso a la base de datos donde es `TRUE`.

@Insert

Permite invocar procedimientos almacenados que realizan cambios en la base de datos, pudiendo ser su acción principal, entre otras cosas, insertar uno o varios registros en una o varias tablas y se debe usar en conjunción con la anotación `@CustomSqlQuery` donde se indica la query de invocación del procedimiento almacenado o la anotación `@MyBatisCustomSqlStatementId` para indicar el `statementId` de MyBatis que contiene la query de invocación del procedimiento almacenado.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query, indicando a MyBatis que `statementType` es de tipo `CALLABLE`:

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

@Update

Permite invocar procedimientos almacenados que realizan cambios en la base de datos, pudiendo ser su acción principal, entre otras cosas, actualizar uno o varios registros en una o varias tablas y se debe usar en conjunción con la anotación `@CustomSqlQuery` donde se indica la query de invocación del procedimiento almacenado o la anotación `@MyBatisCustomSqlStatementId` para indicar el `statementId` de MyBatis que contiene la query de invocación del procedimiento almacenado.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query, indicando a MyBatis que `statementType` es de tipo `CALLABLE`:

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

@Delete

Permite invocar procedimientos almacenados que realizan cambios en la base de datos, pudiendo ser su acción principal, entre otras cosas, eliminar uno o varios registros en una o varias tablas y se debe usar en conjunción con la anotación `@CustomSqlQuery` donde se indica la query de invocación del procedimiento almacenado o la anotación `@MyBatisCustomSqlStatementId` para indicar el `statementId` de MyBatis que contiene la query de invocación del procedimiento almacenado.

Ejemplo: Para la definición de la operación

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

Se genera para PostgreSQL la query, indicando a MyBatis que `statementType` es de tipo `CALLABLE`:

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

Invocación de procedimientos almacenados complejos

Uaithne ofrece cuatro tipos de operaciones especiales para invocar procedimientos almacenados que tienen parámetros de salida, donde los parámetros de salida se pueden usar como resultado de la operación.

Operaciones para invocar procedimientos almacenados con parámetros de salida:

- **`ComplexSelectCall`**: Permite invocar un procedimiento almacenado que realiza una consulta en la base de datos sin realizar cambios en esta.
- **`ComplexInsertCall`**: Permite invocar un procedimiento almacenado que realiza cambios en la base de datos, que pudieran tratarse de un insert (no es obligatorio).
- **`ComplexUpdateCall`**: Permite invocar un procedimiento almacenado que realiza cambios en la base de datos, que pudieran tratarse de un update (no es obligatorio).
- **`ComplexDeleteCall`**: Permite invocar un procedimiento almacenado que realiza cambios en la base de datos, que pudieran tratarse de un delete (no es obligatorio).

Importante: Este tipo de operaciones no genera la query automáticamente por lo que debe ser provista usando `@CustomSqlQuery(query="...")` indicándole la query o con `@MyBatisCustomSqlStatementId`. indicando el `statementId` de MyBatis que contienen en un fichero no generado la query a ser ejecutada.

El objeto que recibe MyBatis en este tipo de operaciones es diferente al resto de operaciones ofrecidas por Uaithne, es estas operaciones se crea un objeto especial que contiene las propiedades:

- **`operation`**: contiene el objeto con la información de la operación a ser ejecutada.

- **`result`**: contiene o contendrá el resultado de la operación. Si el resultado de la operación es una entidad (o vista de entidad) esta propiedad se preinicializa (se puede cambiar usando la propiedad `initResult` disponible en la anotación de la operación) llamando al constructor por defecto.

El que la operación reciba este objeto contenedor provoca que al escribir la query se tenga que especificar explícitamente el origen de la información al escribir la query, prefijando el nombre del campo con `operation.` o `result.` según sea el caso.

Ejemplo: Cómo acceder a un campo desde MyBatis

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

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

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

Para dar soporte a los parámetros de entrada y salida, Uaithne añade al objeto contenedor cualquier propiedad de igual nombre y tipo de dato que exista entre la operación y el resultado, siempre que este sea una entidad (o vista de entidad). También se añade cualquier subpropiedad que cumpla con los criterios que estén contenidas dentro de una propiedad (siempre que sea una entidad) contenida dentro de la operación (de haber múltiples coincidencias se queda con la primera, según el orden que el programador las añadió).

Ejemplo: Cómo acceder a un campo de entrada y salida desde MyBatis

Si se tiene una operación que tiene una propiedad (o una entidad con una propiedad) llamada `start` de tipo `java.util.Date`, se crea una propiedad de igual nombre en el objeto contenedor, que permite acceder al valor de esta desde la operación, pero al establecer el valor lo escribe en el resultado, no hace falta usar ningún prefijo para usarla:

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

Las propiedades de entrada y salida permiten acceder desde MyBatis la operación y el resultado como una única propiedad, leyendo el valor de la operación, pero al ser modificado el valor se almacena en el resultado (sin modificar la operación).

Limitación:

Al usar estas propiedades de entrada y salida puede ser necesario especificar el tipo de dato Java (usando `javaType=`), ya que de no hacerlo, MyBatis no lo tiene en cuenta, por lo que el tipo de datos es el por defecto, que no necesariamente coincide con el del resultado, provocando un error oscuro en tiempo de ejecución que indica que no se puede hacer `cast` del objeto o que no se puede invocar un método con ese tipo de dato.

@ComplexSelectCall

Permite crear una operación para invocar un procedimiento almacenado que tiene parametros de salida como si fuese un `select`, es decir, sin efecto colateral en la base de datos. **Nota**: esta operación invoca el método `selectOne` de MyBatis, y este método puede provocar que no se haga `commit` de la transacción en la base de datos, quedando los cambios que haga el procedimiento almacenado pendientes de commit si este realiza algún cambio.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación
- **`initResult`**: indica si el resultado se debe inicializar (llamando al constructor sin argumentos) antes de de invocar al procedimiento almacenado, permitiendo así usar sus propiedades en la invocación del procedimiento almacenado.

  De ser **`TRUE`** el resultado se inicializa llamando al constructor sin argumentos.

  De ser **`FALSE`** el resultado no se inicializa, por lo que su valor será nulo, 0 o `false` según su tipo de dato y no se podrá acceder a ninguna de sus propiedades en la invocación del procedimiento almacenado.

  De ser **`UNSPECIFIED`** (por defecto) el resultado se inicializa si se trata de una entidad (o vista de entidad), en caso contrario no se inicializa.

Ejemplo: Para la definición de entidad y la operación

```
@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;
}
```

Se genera para PostgreSQL la query, indicando a MyBatis que `statementType` es de tipo `CALLABLE`:

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

Importante:

- Los **parámetros de entrada** están presente en la operación y en la query se escriben prefijando el nombre del campo con "`operation.`"
- Los **parámetros de salida** están presente en la entidad resultante y en la query se escriben prefijando el nombre del campo con "`result.`"
- Los **parámetros de entrada y salida** están presente en la operación y en la entidad resultante, y en la query se escriben sin especificar ningún prefijo

@ComplexInsertCall

Permite crear una operación para invocar un procedimiento almacenado que tiene parametros de salida como si fuese un `insert`, es decir, tiene efecto colateral en la base de datos. **Nota**: esta operación invoca el método `insert` de MyBatis, pero JDBC no hace distinción entre insert, update o delete.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación
- **`initResult`**: indica si el resultado se debe inicializar (llamando al constructor sin argumentos) antes de de invocar al procedimiento almacenado, permitiendo así usar sus propiedades en la invocación del procedimiento almacenado.

  De ser **`TRUE`** el resultado se inicializa llamando al constructor sin argumentos.

  De ser **`FALSE`** el resultado no se inicializa, por lo que su valor será nulo, 0 o `false` según su tipo de dato y no se podrá acceder a ninguna de sus propiedades en la invocación del procedimiento almacenado.

  De ser **`UNSPECIFIED`** (por defecto) el resultado se inicializa si se trata de una entidad (o vista de entidad), en caso contrario no se inicializa.

Ejemplo: Para la definición de entidad y la operación

```
@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;
}
```

Se genera para PostgreSQL la query, indicando a MyBatis que `statementType` es de tipo `CALLABLE`:

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

Importante:

- Los **parámetros de entrada** están presente en la operación y en la query se escriben prefijando el nombre del campo con "`operation.`"
- Los **parámetros de salida** están presente en la entidad resultante y en la query se escriben prefijando el nombre del campo con "`result.`"
- Los **parámetros de entrada y salida** están presente en la operación y en la entidad resultante, y en la query se escriben sin especificar ningún prefijo

@ComplexUpdateCall

Permite crear una operación para invocar un procedimiento almacenado que tiene parametros de salida como si fuese un `update`, es decir, tiene efecto colateral en la base de datos. **Nota**: esta operación invoca el método `update` de MyBatis, pero JDBC no hace distinción entre insert, update o delete.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación
- **`initResult`**: indica si el resultado se debe inicializar (llamando al constructor sin argumentos) antes de de invocar al procedimiento almacenado, permitiendo así usar sus propiedades en la invocación del procedimiento almacenado.

  De ser **`TRUE`** el resultado se inicializa llamando al constructor sin argumentos.

  De ser **`FALSE`** el resultado no se inicializa, por lo que su valor será nulo, 0 o `false` según su tipo de dato y no se podrá acceder a ninguna de sus propiedades en la invocación del procedimiento almacenado.

  De ser **`UNSPECIFIED`** (por defecto) el resultado se inicializa si se trata de una entidad (o vista de entidad), en caso contrario no se inicializa.

Ejemplo: Para la definición de entidad y la operación

```
@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;
}
```

Se genera para PostgreSQL la query, indicando a MyBatis que `statementType` es de tipo

CALLABLE:

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

Importante:

- Los **parámetros de entrada** están presente en la operación y en la query se escriben prefijando el nombre del campo con "`operation.`"
- Los **parámetros de salida** están presente en la entidad resultante y en la query se escriben prefijando el nombre del campo con "`result.`"
- Los **parámetros de entrada y salida** están presente en la operación y en la entidad resultante, y en la query se escriben sin especificar ningún prefijo

@ComplexDeleteCall

Permite crear una operación para invocar un procedimiento almacenado que tiene parametros de salida como si fuese un `delete`, es decir, tiene efecto colateral en la base de datos. **Nota**: esta operación invoca el método `delete` de MyBatis, pero JDBC no hace distinción entre insert, update o delete.

Parámetros:

- **`result`**: indica cual es el tipo de datos del resultado de la operación
- **`initResult`**: indica si el resultado se debe inicializar (llamando al constructor sin argumentos) antes de de invocar al procedimiento almacenado, permitiendo así usar sus propiedades en la invocación del procedimiento almacenado.

  De ser **`TRUE`** el resultado se inicializa llamando al constructor sin argumentos.

  De ser **`FALSE`** el resultado no se inicializa, por lo que su valor será nulo, 0 o `false` según su tipo de dato y no se podrá acceder a ninguna de sus propiedades en la invocación del procedimiento almacenado.

  De ser **`UNSPECIFIED`** (por defecto) el resultado se inicializa si se trata de una entidad (o vista de entidad), en caso contrario no se inicializa.

Ejemplo: Para la definición de entidad y la operación

```
@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;
}
```

Se genera para PostgreSQL la query, indicando a MyBatis que `statementType` es de tipo `CALLABLE`:

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

Importante:

- Los **parámetros de entrada** están presente en la operación y en la query se escriben prefijando el nombre del campo con "`operation.`"
- Los **parámetros de salida** están presente en la entidad resultante y en la query se escriben prefijando el nombre del campo con "`result.`"
- Los **parámetros de entrada y salida** están presente en la operación y en la entidad resultante, y en la query se escriben sin especificar ningún prefijo

Misceláneos

Related vs Extends

Se puede hacer extends de entidades para añadir más campos, con `related` (presente como una propiedad en las anotaciones `@Entity` y `@EntityView`) se puede hacer un subconjunto de ésta.

En operaciones o en entidades con `related` los campos mantienen las anotaciones establecidas en las entidades fuente, siempre que se llamen igual, excepto la anotación `@Optional`.

Anotaciones para controlar campos con una semántica especial

@InsertDate

Indica que el campo representa la fecha de inserción del registro, por lo que al generar la query de insert este campo siempre es tenido en cuenta y su valor se sustituye por la fecha y hora actual de la de de datos (no se lee el valor del campo para realizar el insert). Su tipo de datos debe ser `java.util.Date`, `java.sql.Date`, `java.sql.Time` o `java.sql.Timestamp`.

Si se crean operaciones `@Insert` que referencian a una entidad con este campo, en el insert se añade automáticamente este campo (aunque no esté presente en la operación).

Esta anotación se suele combinar con la anotación `@ExcludeFromObject` para que se tenga en cuenta durante la generación de la query pero se ignore por completo en los objetos Java.

@InsertUser

Indica que el campo representa la usuario que ordenó la inserción del registro, por lo que al generar la query de insert este campo siempre es tenido en cuenta. Su tipo de datos debe ser `String`.

Para dar valor a este campo se suele usar con la anotación `@ValueWhenNull` o `@ForceValue` con la cual se lee el usuario actual del contexto de ejecución de la aplicación.

Si se crean operaciones `@Insert` que referencian a una entidad con este campo, en el insert se añade automáticamente este campo (aunque no esté presente en la operación).

Esta anotación se suele combinar con la anotación `@ExcludeFromObject` para que se tenga en cuenta durante la generación de la query pero se ignore por completo en los objetos Java.

@UpdateDate

Indica que el campo representa la fecha de actualización del registro, por lo que al generar la query de update este campo siempre es tenido en cuenta y su valor se sustituye por la fecha y hora actual de la de de datos (no se lee el valor del campo para realizar el update). Su tipo de datos debe ser `java.util.Date`, `java.sql.Date`, `java.sql.Time` o `java.sql.Timestamp`.

Si se crean operaciones `@Update` que referencian a una entidad con este campo, en el update se añade automáticamente este campo (aunque no esté presente en la operación).

Esta anotación se suele combinar con la anotación `@ExcludeFromObject` para que se tenga en

cuenta durante la generación de la query pero se ignore por completo en los objetos Java.

Probablemente un campo con esta anotación también requiera el uso de la anotación `@Optional` para indicar que puede no tener valor en la base de datos si no ha sido actualizado aún, al menos que el campo también esté marcado con la anotación `@InsertDate`.

@UpdateUser

indica que el campo representa la usuario que ordenó la actualización del registro, por lo que al generar la query de update este campo siempre es tenido en cuenta. Su tipo de datos debe ser `String`.

Para dar valor a este campo se suele usar con la anotación `@ValueWhenNull` o `@ForceValue` con la cual se lee el usuario actual del contexto de ejecución de la aplicación.

Si se crean operaciones `@Update` que referencian a una entidad con este campo, en el update se añade automáticamente este campo (aunque no esté presente en la operación).

Esta anotación se suele combinar con la anotación `@ExcludeFromObject` para que se tenga en cuenta durante la generación de la query pero se ignore por completo en los objetos Java.

Probablemente un campo con esta anotación también requiera el uso de la anotación `@Optional` para indicar que puede no tener valor en la base de datos si no ha sido actualizado aún, al menos que el campo también esté marcado con la anotación `@InsertUser`.

@DeleteDate

Indica que el campo representa la fecha de eliminación del registro, por lo que al generar la query de update (para realizar el borrado lógico) este campo siempre es tenido en cuenta y su valor se sustituye por la fecha y hora actual de la de de datos (no se lee el valor del campo para realizar el update). Su tipo de datos debe ser `java.util.Date`, `java.sql.Date`, `java.sql.Time` o `java.sql.Timestamp`.

El uso de esta anotación requiere que la entidad posea un campo con la anotación `@DeletionMark` para activar el borrado lógico.

Si se crean operaciones `@Delete` que referencian a una entidad con este campo, en el update (para realizar el borrado lógico) se añade automáticamente este campo (aunque no esté presente en la operación).

En aquellas entidades con borrado lógico donde la marca de borrado sea la fecha de borrado se acompaña esta anotación con las anotaciones `@DeletetionMark` y `@Optional`.

Esta anotación se suele combinar con la anotación `@ExcludeFromObject` para que se tenga en cuenta durante la generación de la query pero se ignore por completo en los objetos Java.

Probablemente un campo con esta anotación también requiera el uso de la anotación `@Optional` para indicar que puede no tener valor en la base de datos si no ha sido eliminado aún, al menos que el campo también esté marcado con la anotación `@InsertDate` o `@UpdateDate` o ambos.

@DeleteUser

Indica que el campo representa la usuario que ordenó la eliminación del registro, por lo que al generar la query de update (para realizar el borrado lógico) este campo siempre es tenido en cuenta. Su tipo de datos debe ser `String`.

El uso de esta anotación requiere que la entidad posea un campo con la anotación `@DeletionMark` para activar el borrado lógico.

Para dar valor a este campo se suele usar con la anotación `@ValueWhenNull` o `@ForceValue` con la cual se lee el usuario actual del contexto de ejecución de la aplicación.

Si se crean operaciones `@Delete` que referencian a una entidad con este campo, en el update (para realizar el borrado lógico) se añade automáticamente este campo (aunque no esté presente en la operación).

En aquellas entidades con borrado lógico donde la marca de borrado sea el usuario de borrado se acompaña esta anotación con las anotaciones `@DeletetionMark` y `@Optional`.

Esta anotación se suele combinar con la anotación `@ExcludeFromObject` para que se tenga en cuenta durante la generación de la query pero se ignore por completo en los objetos Java.

Probablemente un campo con esta anotación también requiera el uso de la anotación `@Optional` para indicar que puede no tener valor en la base de datos si no ha sido eliminado aún, al menos que el campo también esté marcado con la anotación `@InsertUser` o `@UpdateUser` o ambos.

@DeletionMark

Indica que el campo representa la marca de borrado lógico del registro.

El uso de un campo con esta anotación provoca que Uaithne genere un update de borrado lógico (mediante un update al valor de este campo) en lugar de un delete de borrado físico, el uso de esta anotación también provoca que se incluya automáticamente en el where de los update y select una cláusula que asegure que el registro no esté borrado, y en los insert se incluya indicando que el registro no ha sido borrado.

Los tipos de datos soportado son booleano o cualquier otro tipo de dato anotado con `@Optional`.

Para manejar el valor de esta anotación en el caso de que su tipo de datos no sea booleano se suele combinar con la anotación `@DeleteDate` o `@DeleteUser`.

Esta anotación se suele combinar con la anotación `@ExcludeFromObject` para que se tenga en cuenta durante la generación de la query pero se ignore por completo en los objetos Java.

@IgnoreLogicalDeletion

En las operaciones con esta anotación la query generada se debe realizar ignorando cualquier campo anotado con `@DeletionMark`, por lo que si se trata de un update o un select no se incluye en el where la comprobación de que el registro no esté borrado, pero si se trata de una operación de borrado se realiza el borrado físico mediante un delete en lugar de hacer un borrado lógico.

@Version

Indica que el campo representa la marca que indica la versión de un registro. Su funcionamiento está aún por definir, por ahora solo se usa para indicar en la entidad de su propósito, pero en las queries generadas es siempre ignorado.

Otras anotaciones para controlar mejor las queries

@PageQueries

Permite especificar por separado las query de una operación `@SelectPage` permitiendo indicar la query que recupera los datos de la página y por separado la query que permite contar la cantidad de registros existentes.

Parámetros:

- **`selectCount`:** Query para consultar la cantidad de registros existentes.
- **`selectPage`:** Query a ser ejecutada para recuperar los datos de la página de datos.

~~@EntityQueries~~

Probablemente se deba marcar como obsoleta, permite especificar las queries de las operaciones que se generan automáticamente para una entidad, es mejor usar en este caso las operaciones de entidades en combinación con `@CustomSqlQuery` que permite lograr mejores resultados.

Parámetros:

- **`selectById`:** Query a ser ejecutada para recuperar el registro dado su identificador.
- **`insert`:** Query de inserción de un registro nuevo.
- **`lastInsertedId`:** Query que permite recuperar el identificador del registro que se acaba de insertar.
- **`update`:** Query de actualización de un registro en la base de datos.
- **`deleteById`:** Query a ser ejecutada para eliminar un registro dado su identificador.
- **`merge`:** Query a ser ejecutada para actualizar un registro en la base de datos, pero solo actualizado aquellas columnas cuyos campos en la entidad no sean nulas.

~~@Query~~

Probablemente se deba marcar como obsoleta, permite especificar la query de una operación, es mejor usar la anotación `@CustomSqlQuery` en su lugar.

Java Beans Validations

- Se puede usar las anotaciones de Java Beans Validations sobre los campos de las entidades u operaciones, estas se generan en las clases resultantes.
- Se saca mucho provecho de que los campos mantienen las anotaciones establecidas al entidades al ser usadas en las operaciones o en entidades con related, siempre que se llamen igual, evitando así que tener que repetir la información de las anotación de Java Beans Validations, basta con colocarlas en las entidades y en cualquier otro campo en las operaciones o entidades con related que no esté en la entidad original.

Configuración de Uaithne

- Configuración de Uaithne: `@UaithneConfiguration`
- Generación de clases comunes usando `@SharedLibrary`
- Generación de clases comunes de MyBatis usando `@SharedMyBatisLibrary`
- Configuración de MyBatis: es necesario el uso de `ApplicationParameterDriver` y `RetainIdPlugin` para que todo funcione correctamente

Ω fin Ω

Diseñado por: **Juan Luis Paz Rojas** - https://github.com/juanluispaz