## @SqlUpdate

The @SqlUpdate annotation is used to annotate a method as a custom SQL update or insert statement.

An “update” method may accept any type or primitive as input parameters; however, the name of the parameter will used as the name of the replacement variable in the SQL statement, so they will need to be consistent.

The return type of an “update” method must be one of the following:

* void - the return type is ignored
* boolean - which will be true for a non-zero update record count, and false for 0.
* int or long - which will be the count of the records inserted or updated.

The value property of the annotation is used to provide the SQL string which will be compiled into the method. The method parameters will be used as replacement variables in the SQL using the parameter name prefixed with a colon (e.g. :firstName).

An “update” method would look something like the following:

@SqlUpdate('insert into pets (name,breed) values (:name,:breed)')  
**abstract** **boolean** addPet(String name, String breed)

The “update” methods also allow the use of the PreparedStatementSetter helper annotation. See its documentation above, for more information.

>> Should also accept ordered parameters based on SQL order and method parameter order - jinky but should support it

>> Need to add and support the “strategy” annotation property - defaults to Groovy (or whatever configured using @JdbcStrategy has for class)

>> How does the DataSource or JdbcTemplate get injected? Add a “source” property as a String - this will be the name of the field/property of the class containing either a DataSource (Groovy or Spring) or JdbcTemplate (Spring) instance to be used. Population of the field is left to the developer or framework. A default source name should be provided for each strategy.

## @SqlSelect

The @SqlSelect annotation is used to annotate a method as a custom SQL query.

A “select” method may accept any type or primitive as input parameters; however, the name of the parameter will used as the name of the replacement variable in the SQL statement, so they will need to be consistent.

A “select” method must return a single type or collection of a type that is appropriate to the RowMapper or ResultSetExtractor being used. If a RowMapper or ResultSetExtractor are not specified, Effigy will attempt to resolve the appropriate mapper based on the return type - if it cannot resolve a mapper an instance of the BeanPropertyRowMapper class will be used - this may or may not work for the configured scenario, but it is a minimal fallback point.

The return types currently supported by the default row mappers are the following: Byte, byte, Character, char, Short, short, Integer, int, Long, long, Float, float, Double, double, Boolean, boolean, and String. Any other return types will fallback to use the BeanPropertyRowMapper.

The value property of the annotation is used to provide the SQL string which will be compiled into the method. The method parameters will be used as replacement variables in the SQL using the parameter name prefixed with a colon (e.g. :firstName).

A “select” method using the default return type mapper resolution would look something like the following:

@SqlSelect('select count(\*) from people where age >= :min and age <= :max')  
**abstract** **int** countByAgeRange(**int** min, **int** max)

In order to configure a custom RowMapper or ResultSetExtractor, it must be specified using a secondary annotation: the @RowMapper or @ResultSetExtractor annotations.

>> Should also accept ordered parameters based on SQL order and method parameter order - jinky but should support it

>> Need to add and support the “strategy” annotation property - defaults to Groovy (or whatever configured using @JdbcStrategy has for class)

>> How does the DataSource or JdbcTemplate get injected? Add a “source” property as a String - this will be the name of the field/property of the class containing either a DataSource (Groovy or Spring) or JdbcTemplate (Spring) instance to be used. Population of the field is left to the developer or framework. A default source name should be provided for each strategy.

## @JdbcStrategy

Annotation used on a class to denote a custom default JdbcStrategy for the Grow annotations to be used as a default, which may be overridden by the specific annotation uses.

Currently the strategies are Groovy and Spring

## @RowMapper

The @RowMapper annotation is used with a @SqlSelect annotation to provide information about the RowMapper to be used.

There are three distinct configuration scenarios for mapper annotations, they can be defined by the annotations bean, or type properties, or by a combination of the type and factory properties.

The bean property will inject code into the repository to autowire a reference to the bean with the specified name. The mapper bean must be defined somewhere in the Spring context, and must implement the RowMapper interface. This bean will then be used as the RowMapper for the query.

The type property will inject code into the repository to use an instance of the specified class as the mapper. The class must implement the RowMapper interface.

The type and factory properties used together will inject code that will call the static factory method on the specified class to retrieve an implementation of RowMapper which will be used by the query.

If multiple properties are configured outside the scope of these scenarios, the precedence order will be bean, then type; factory will be ignored if the type property is not specified.

The singleton property is used to specify whether or not the generated mapper is shared across multiple calls (singleton=true) or has a new instance created for each use (singleton=false); the default is true. One thing to make special note of, is that for the case when the beanproperty is specified along with the singleton property having a value of false, the configured bean in your application context should be configured as a prototype bean, otherwise you are not really getting a new instance with each call.

Configured RowMapper instance are allowed to access the arguments passed into the method; to do this, the arguments property must be set to true, which will force the singleton property to be false (prototype). See the description of the singleton property above for more information. Mappers that accept method arguments must either implement the ArgumentAwareHelper or provide a method with the following signature:

**void** setMethodArguments(Map<String,Object> args)

The arguments will be injected at runtime using this method. It is up to the implementation to make proper use of them. Obviously, mappers making use of this construct are no longer stateless.

An example of using the @RowMapper annotation would be the following:

@SqlSelect('select a,b,c from some\_table where d=:d and e < :e')  
@RowMapper(type=AbcMapper)  
**abstract** Collection<Abc> findByDAndE(String d, **int** e)

>> Change “bean” property to “instance” - this will then be the name of a field or property of the class containing the mapper to be used: how it gets populated is up to the developer and the framework being used.

>> Double-check the singleton functionality - some of that might change here.

## @ResultSetExtractor

The @ResultSetExtractor annotation is used with a @SqlSelect annotation to provide information about the ResultSetExtractor to be used.

There are three distinct configuration scenarios for extractor annotations, they can be defined by the annotations bean, or type properties, or by a combination of the type and factory properties.

The bean property will inject code into the repository to autowire a reference to the bean with the specified name. The extractor bean must be defined somewhere in the Spring context, and must implement the ResultSetExtractor interface. This bean will then be used as the ResultSetExtractor for the query.

The type property will inject code into the repository to use an instance of the specified class as the extractor. The class must implement the ResultSetExtractor interface.

The type and factory properties used together will inject code that will call the static factory method on the specified class to retrieve an implementation of ResultSetExtractor which will be used by the query.

If multiple properties are configured outside the scope of these scenarios, the precedence order will be bean, then type; factory will be ignored if the type property is not specified.

The singleton property is used to specify whether or not the generated extractor is shared across multiple calls (singleton=true) or has a new instance created for each use (singleton=false); the default is true. One thing to make special note of, is that for the case when the beanproperty is specified along with the singleton property having a value of false, the configured bean in your application context should be configured as a prototype bean, otherwise you are not really getting a new instance with each call.

Configured ResultSetExtractor instances are allowed to access the arguments passed into the method; to do this, the arguments property must be set to true, which will force the singleton property to be false (prototype). See the description of the singleton property above for more information. Extractors that accept method arguments must either implement the ArgumentAwareHelper or provide a method with the following signature:

**void** setMethodArguments(Map<String,Object> args)

The arguments will be injected at runtime using this method. It is up to the implementation to make proper use of them. Obviously, extractors making use of this construct are no longer stateless.

An example of using the @ResultSetExtractor annotation would be the following:

@SqlSelect('select a,b,c from some\_table where d=:d and e < :e')  
@ResultSetExtractor(type=AbcExtractor, factory='getExtractor')  
**abstract** Collection<Abc> findByDAndE(String d, **int** e)

Note that when an extractor is used, the return type of the method should match, or at least be compatible with the return type of the ResultSetExtractor since the extractor is used to build the return value explicitly.

>> Change “bean” property to “instance” - this will then be the name of a field or property of the class containing the mapper to be used: how it gets populated is up to the developer and the framework being used.

>> Double-check the singleton functionality - some of that might change here.

## @PreparedStatementSetter

The @PreparedStatementSetter annotation is used with a @SqlSelect annotation to provide information about the PreparedStatementSetter to be used. This is an optional annotation and may be used in conjunction with a @RowMapper or@ResultSetExtractor annotation.

There are three distinct configuration scenarios for setter annotations, they can be defined by the annotations bean, or type properties, or by a combination of the type and factory properties.

The bean property will inject code into the repository to autowire a reference to the bean with the specified name. The extractor bean must be defined somewhere in the Spring context, and must implement the PreparedStatementSetter interface. This bean will then be used as the PreparedStatementSetter for the query.

The type property will inject code into the repository to use an instance of the specified class as the setter. The class must implement the PreparedStatementSetter interface.

The type and factory properties used together will inject code that will call the static factory method on the specified class to retrieve an implementation of PreparedStatementSetter which will be used by the query.

If multiple properties are configured outside the scope of these scenarios, the precedence order will be bean, then type; factory will be ignored if the type property is not specified.

The singleton property is used to specify whether or not the generated setter is shared across multiple calls (singleton=true) or has a new instance created for each use (singleton=false); the default is true. One thing to make special note of, is that for the case when the beanproperty is specified along with the singleton property having a value of false, the configured bean in your application context should be configured as a prototype bean, otherwise you are not really getting a new instance with each call.

Configured PreparedStatementSetter instances are allowed to access the arguments passed into the method; to do this, the arguments property must be set to true, which will force the singleton property to be false (prototype). See the description of the singleton property above for more information. Setters that accept method arguments must either implement the ArgumentAwareHelper or provide a method with the following signature:

**void** setMethodArguments(Map<String,Object> args)

The arguments will be injected at runtime using this method. It is up to the implementation to make proper use of them. Obviously, mappers making use of this construct are no longer stateless.

An example of using the @PreparedStatementSetter annotation would be the following:

@SqlSelect('select a,b,c from some\_table where d=:d and e < :e')  
@ResultSetExtractor(type=AbcExtractor, factory='getExtractor')  
@PreparedStatementSetter(type=DandESetter)  
**abstract** Collection<Abc> findByDAndE()

Note that when an setter is used, there are no method arguments (or they are ignored) unless the annotation has arguments set to true; see the description of the arguments property above, for more information.

>> Change “bean” property to “instance” - this will then be the name of a field or property of the class containing the mapper to be used: how it gets populated is up to the developer and the framework being used.

>> Double-check the singleton functionality - some of that might change here.

## @RowMapperFactory

method, field, property - uses static version of DSL to generate row mapper (groovy or spring)

params: name, dsl, strategy

Should also be a dynamic version of the DSL for both strategies

## @ResultSetExtractorFactory

method, field, property - this this even possible? These are mainly for joins and for the case fo the factory should be mostly a combination of mappers with prefixes

Not sure this will lend itself to a DSL , but worth a look

>> May want to call them Mappers and Extractors (as interfaces) to differentiate between this and Spring’s stuff

each annotated row mapper builder should have a DSL (the same) which may be used static or dynamically to create mappers

(see also ResultSetExtractor - should have a similar entity for that in both cases)

(prepared statement setters?

should also have (here or in vanilla) a ResultSet builder DSL for testing

class Person {

String firstName

String lastName

int age

long highScore

@RowMapper(

value=Person

dsl={

ignore 'highScore'

map 'firstName' fromString 'given\_name' using { -> }

}

)

static RowMapper<Person> mapper(){}

}

map <property-name> [[from | fromXXX] <db-field>] [using <convert-closure>]

ignore <property-name> (one or more)

“using” closure accepts arguments (dbFieldValue, resultSet, mappedObj) and should return the value for the property being mapped

“from <db-field>” - explicitly state the name of the db field where the data is to be pulled; the type will be resolved by the the property type or default to Object (rs.getObject())

“fromXXX <db-field>” - explicitly states the name of the field and the type of result set getter to be used (e.g. fromString ‘foo’ will call rs.getString(‘foo’))

(the “from” call may be a string value for a named field, or a number for the index number of the result set value)

“ignore” - ignores and will not map the specified properties (one or more)

* all properties are mapped by default to a database field named in underscore style (firstName : first\_name) - might want a way to specify case (upper/lower)
* the type of object being mapped is passed into the annotation and used to determine the default mapping config - the DSL is applied on top of the default (meaning that for some objects you may not need any configuration beyond the mapped type)