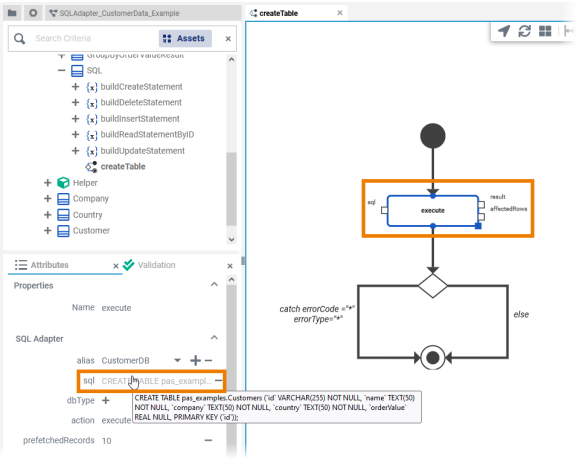
# Querying SQL Databases

In order to make an activity interact with a database, drag an SQL operation to a diagram pane and apply the stereotype **SQL Adapter** to it. The attribute **sql** then will contain the SQL statement (see Performing SQL Statements). However, there is also the possibility to give the SQL statement as input string (see Static versus Dynamic SQL).

## Performing SQL Statements

In the example below, the **execute** operation will access the database that is associated with the alias **CustomerDB**. In the attributes panel attribute **sql** allows to enter an SQL script, which in our example creates a new table to the associated database. The resulting output of the query is returned to the caller via the **result** pin if there is any.




### Writing SQL Queries

After receiving the result from the DBMS at runtime, the xUML Runtime checks if the columns can be mapped to the attributes of the output object (upper and lower case is not distinguished). If the xUML Runtime cannot map the result sets to objects, it will throw an error. This is also the case if a query returns multiple records and the receiving object is not an array (see Selecting Multiple Records further below).

### Using a Variable Table Name

If you want to make the database table partly variable, you can use the attributes **schema** and **tableQualifier** (see SQL Adapter Reference).

- **schema** is a string that prefixes tables and stored procedures. It changes the table name to <schema>.<table name>, e.g. `S1.TEMPLOYEE`.
- **tableQualifier** is a string that prefixes tables. It changes the table name to <tableQualifier><table name>, e.g. `TQ1TEMPLOYEE`.

Both values can be changed on the deployed service. Also, a combination of both is possible: <schema>.<tableQualifier><table name>.

> This works only if the tables are marked using the `TABLE::` keyword, e.g `TABLE::Customers` in SQL statements. If you do not prefix the table name by `TABLE::`, the table name is used as it is.

### Blob Handling

Some databases handle BLOB columns differently to their other types.In this case, you cannot stick with ANSI SQL but need to provide the blob content in the way requested by the related database.

- MySQL

**On this Page:**



**SQLAdapter_CustomerData_Example**

Click the icon to download a simple example model that shows the usage of the SQL adapter in **Sc heer PAS** *Designer* .

**Related Pages:**

- Attributes Panel
- Database-Specific Mappings

```
"INSERT INTO Image"
       +" (ID, Subject, Body)"
       +" VALUES (1, '"+subject+"'"
       +", x'<blob content>')");
```

- Oracle

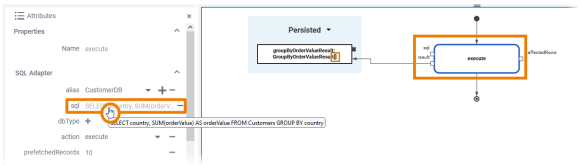
```
"INSERT INTO Image"
       +" (ID, Subject, Body)"
       +" VALUES (1, '"+subject+"'"
       +", '<blob content>')");
```

- SQL Server

```
"INSERT INTO Image"
       +" (ID, Subject, Body)"
       +" VALUES (1, '"+subject+"'"
       +", <blob content>)");
```

## Selecting Multiple Records

If you expect a query to return more than one record, your result object must be an array (xUML base type **Array**). Open the attributes panel on the corresponding variable to assign a type, in this case the complex type **groupByOrderValueResult**, and enable the **Array** attribute (see Attributes Panel). The implicit mapping from columns to object attributes works like in the first example.

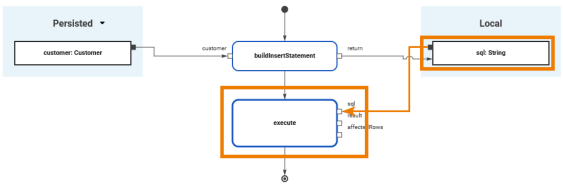



# Static versus Dynamic SQL

Entering the SQL statement directly to the **sql** attribute of the adapter operation provides the statement in a static way. However, it is also possible to provide the SQL Adapter with an input string that contains the SQL statement. You can use this to build full dynamic SQL statements.

> ⚠ Note that it can be a security issue, if the input SQL string (or parts of it) comes as an input parameter from outside the service. This would give the caller the possibility to inject malicious code.

The following figure shows an example of a dynamically generated SQL statement. The adapter expects the input parameter **sql**, if it does not have a static SQL statement given as value of the **sql** attribute.




A valid SQL string might look like: `set sqlStatement = "select name, company, country from Customers where id='1234'";`

## Using a Dynamic Table Name (Security Considerations)

Sometimes the name of the table to query might not yet be known at development time. In this case you can use dynamic SQL and build statements that get the table name from e.g. a service setting:

```
local tableName = setting("Table Name", "Customers");
set sqlStatement = concat("select name, company, country from ",
tableName, " where id='", id, "'");
```



**Security remark:** From a security point of view it is important to control the building of the SQL statement. Getting the table name from a service setting is considered safe, getting the table name via an operation parameter would be not.

# SQL Adapter Output

For `SELECT` statements, the SQL adapter needs an output record class (or an array of this class, if the statement creates multiple output records) to store the adapter output to. Here, the SQL Adapter tries to match the table column names with the attribute names of the output class.
For all types of statements there is an additional output parameter **affectedRows**. This parameter returns the number of rows affected by the SQL statement. This comes in handy if the modeler must take into account if e.g. updates had some effect or not**,** e.g. **affectedRows** is equal to zero no records have been updated.

## Mapping of Database Fields

In general, database-specific types are mapped to the xUML base types like described on Database-Specific Mappings.

If you query a database and want to store the query results in object attributes of base type **Boolean**, the xUML Runtime tries to map each table column type to the Boolean attribute type. For instance, if a database query returns a string or numeric representation of a Boolean table field like **true**, **false**, **1** (for true), or **0** (for false), the xUML Runtime will map these values to the Boolean attribute values true or false accordingly.