Querying Databases

This page explains the **SQL Adapter** in Bridge context. If you were looking for the same information regarding the PAS Designer, refer to SQL Adapter in the Designer guide.

In order to make an activity interact with a database, use the <<SQLAdapter>> stereotype for an action node. The tagged value **sql** then will contain the SQL statement (see Performing Static SQL Statements). However, there is also the possibility to give the SQL statement as input string (see Performing Dynamic SQL Statements).

The SQL adapter has the tagged value **alias**. Aliases are stereotyped UML artifacts. Their purpose is to link E2E adapters (defined in the activity diagram) with configuration settings of accessed backend systems. The aliases are defined and configured in the component diagram.
For more details, see SQL Deployment.

Example File (Builder projectAdd-ons/SQL):

<your example path>\Add-ons\SQL\uml\sqlQueries.xml

The example accesses the **Employee.sqlite** database. This SQLite database is installed together with the SQL examples.

Example databases can be found in:

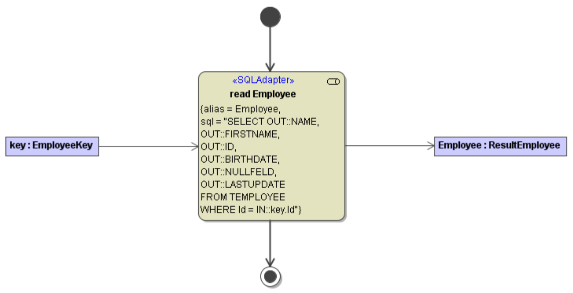
<your example path>\Add-ons\SQL\resources\templates\binaries

Performing Static SQL Statements

In the example below, the <<SQLAdapter>> action node is named **read Employee**. It will access the database that is associated with the alias **Employee** in the component diagram.

The input object of the action node is **key**, which is of complex type **EmployeeKey**. It is used in the SQL script that is entered in the script section of the action node. The last line of the action script references the **Id** attribute of the **key** parameter. The resulting output of the query is returned to the caller in the output object **Employee** of type **ResultEmployee**.

Figure: SQL Adapter – Read Example



```
{alias = Employee,
sql = "SELECT OUT::NAME,
OUT::FIRSTNAME,
OUT::ID,
OUT::BIRTHDATE,
OUT::NULLFELD,
OUT::LASTUPDATE
FROM TEMPLOYEE
WHERE Id = IN::key.Id"}
```

Writing SQL Queries

In the SQL script, all input variables have to be prefixed with `IN::`. All columns that are returned from the database should be prefixed with `OUT::`. This prefix is optional, but enables consistency checks during the compilation phase.
In the example above , the xUML Model Compiler checks if the columns `NAME`, `FIRSTNAME`, `ID`, `BIRTHDATE`, `NULLFELD`, `LASTUPDATE` can be mapped to attributes having the same name in the **ResultEmployee** class. If the names do not match, the Model Compiler will display an error message.

On this Page:

Related Pages:

- SQL Deployment
- Database-Specific Mappings

After receiving the result from the DBMS at runtime, the Bridge 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.
In the above example, only one record is expected to be returned from the DBMS because the <<SQLAdapter>> action returns one object only. If the DBMS were to return multiple records, the xUML Runtime would throw an error.

If you want to make the database table partly variable, you can use the tagged values **schema** and **table Qualifier** (see SQL Deployment).

- **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 Bridge 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::TEMPLOYEE` in SQL statements. If you do not prefix the table name by `TABLE::`, the tablename is used as it is.

Blob Handling

Some databases handle BLOB columns differently to their other types. Therefore, the xUML Model Compiler must recognize BLOB columns. This is achieved by marking the column name within the IN:: keyword, i.e. `IN:<blob column name>`.

Example File (Builder projectAdd-ons/SQL):

⤓	<your example path>\Add-ons\SQL\uml\sqlBlobs.xml

If you want to write database records containing columns that are represented by type **Blob**, you have to use the following in your SQL query

- for inserts

  ```
  IN:<column_name>:<variable_name>
  ```

  for example:

  ```
  INSERT INTO E2ETYPES
  (E2E_BLOB) VALUES (IN:E2E_BLOB:e2etypes.e2e_blob)
  ```

- for updates

  ```
  <column_name> = IN:<column_name>:<variable_name>
  ```

  for example

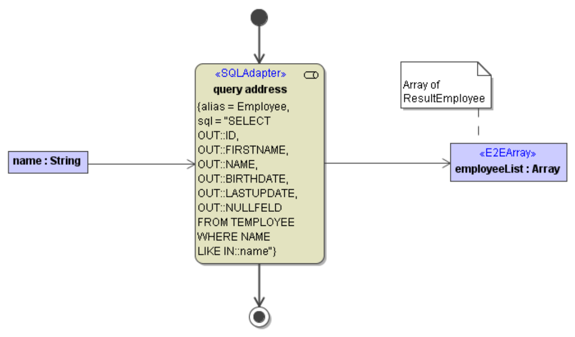
  ```
  UPDATE E2ETYPES
  SET E2E_BLOB = IN:E2E_BLOB:e2etypes.e2e_blob
  WHERE ID = IN::id
  ```

Note that the usage of **`BLOB::`** is deprecated and will not work with multiple blobs in one query.
Be aware of the fact that some DBMS do not support more than one blob column.

Selecting Multiple Records

If you expect a query to return more than one record, your result object must be an array (base type **Array**). The type of the array elements, in this case the complex type **ResultAddress**, is defined on the array object by the tagged value **arrayElement** (stereotype is <<E2EArray>>). The implicit mapping from columns to object attributes works like in the first example.

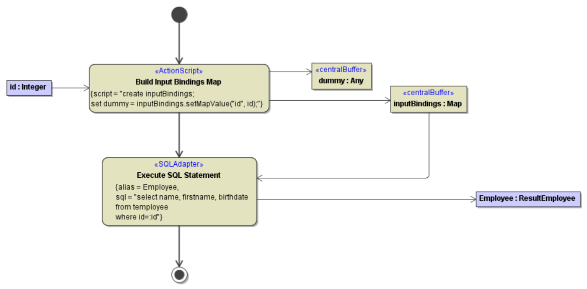
Figure: SQL Adapter – Query Example



Parameterized SQL Statements

You can use the SQL Adapter with parameterized statements that get their parameters via a map.

Figure: Dynamic SQL with Parameterized Statement



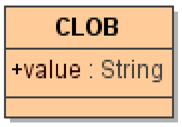
Within the SQL statement, replace values by **:<name of the parameter>**. Provide parameter/value pairs in a map called **inputBindings**. The xUML Runtime will identify the map by this name and automatically replace the parameters in the SQL string by the values given by the map.

Do not use numeric parameter identifiers, e.g. **:12**. However, identifiers like **:id2** are allowed.

You can use parameters with all kinds of SQL statements: with static and dynamic usage of the SQL adapter as well as with all kinds of database access (query, update, delete, ...). Please also respect all restrictions applying to the usage of the static and dynamic SQL Adapter, e.g. to Blob handling.

Using CLOBs in a Parameterized SQL Statement

CLOB values need a special treatment.



They cannot be directly inserted to the map - you have to use the indirect way via a CLOB object:

```
create aCLOB;
set aCLOB.value = inputCLOBValue;
set dummy = inputBindings.setMapValue("parameterName", aCLOB);
```

Now you can use parameter **parameterName** in your SQL statement to access the CLOB value.

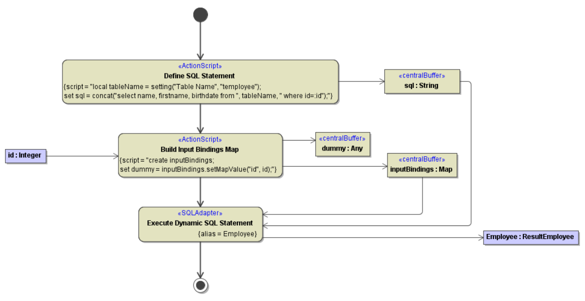
Performing Dynamic SQL Statements

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** tag.

Figure: Example of a Dynamic SQL Statement


«ActionScript»
Define SQL Statement
{script = "local tableName = setting("Table Name", "temployee");
set sql = concat("select name, firstname, birthdate from ", tableName, " where id=:id");"}
«centralBuffer»
sql : String
id : Integer
«ActionScript»
Build Input Bindings Map
{script = "create inputBindings;
set dummy = inputBindings.setMapValue("id", id);"}
«centralBuffer»
dummy : Any
«centralBuffer»
inputBindings : Map
«SQLAdapter»
Execute Dynamic SQL Statement
{alias = Employee}
Employee : ResultEmployee

The SQL string must not contain `IN::` or `OUT::` qualifiers. For instance, a valid SQL string might look like: `set sqlStatement = "select NAME, FIRSTNAME from TEMPLOYEE where Id=5";`

Using a Dynamic Table Name (Security Considerations)

It may be that at development time the name of the table to query is not yet known. 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", "temployee");
set sqlStatement = concat("select NAME, FIRSTNAME, DEPARTMENT from ",
tableName, " where id=:id");
```

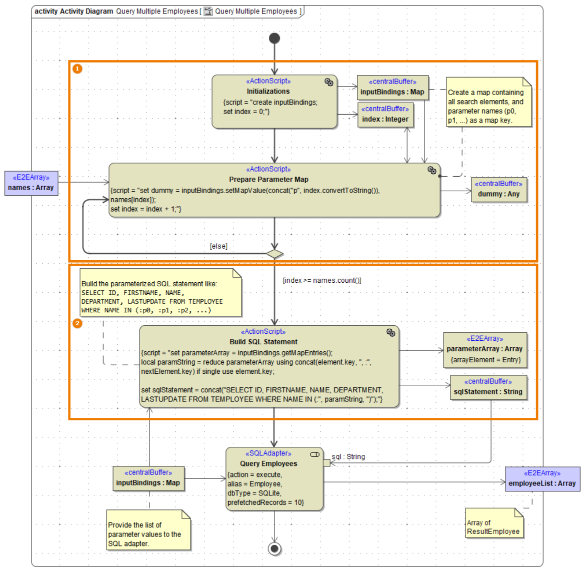
Security remark: From a security point of view it is important to control the building of the SQL statement.

- Getting the table name from a service setting is safe, getting the table name via an operation parameter would be not.
- It is also recommended to **use SQL parameters** (see Parameterized SQL Statements) and not to use `concat()` to add operation parameters directly to the `where` clause.

Using Parameterized SQL With the IN Clause

The xUML Runtime does not allow to build dynamic SQL statements like `select NAME, FIRSTNAME from TEMPLOYEE where name in (IN::<a list of concatenated values>)`. SQL statements like this will not return the expected results or not work at all.
If you want to build a dynamic SQL statement and use the `IN` clause, you need use parameterized SQL as described above (see Parameterized SQL Statements).

The activity diagram below shows an example implementation from the **sqlQueries** example.


activity Activity Diagram Query Multiple Employees [Query Multiple Employees]
1
«ActionScript»
Initializations
{script = "create inputBindings;
set index = 0;"}
«centralBuffer»
inputBindings : Map
«centralBuffer»
index : Integer
Create a map containing all search elements, and parameter names (p0, p1, ...) as a map key.
«E2EArray»
names : Array
«ActionScript»
Prepare Parameter Map
{script = "set dummy = inputBindings.setMapValue(concat("p", index.convertToString()), names[index]);
set index = index + 1;"}
«centralBuffer»
dummy : Any
[else]
[index >= names.count()]
2
Build the parameterized SQL statement like:
SELECT ID, FIRSTNAME, NAME, DEPARTMENT, LASTUPDATE FROM TEMPLOYEE WHERE NAME IN (:p0, :p1, :p2, ...)
«ActionScript»
Build SQL Statement
{script = "set parameterArray = inputBindings.getMapEntries();
local paramString = reduce parameterArray using concat(element.key, ", :", nextElement.key) if single use element.key;
set sqlStatement = concat("SELECT ID, FIRSTNAME, NAME, DEPARTMENT, LASTUPDATE FROM TEMPLOYEE WHERE NAME IN (:", paramString, ")");"}
«E2EArray»
parameterArray : Array
{arrayElement = Entry}
«centralBuffer»
sqlStatement : String
«SQLAdapter»
Query Employees
{action = execute,
alias = Employee,
dbType = SQLite,
prefetchedRecords = 10}
sql : String
«centralBuffer»
inputBindings : Map
«E2EArray»
employeeList : Array
Provide the list of parameter values to the SQL adapter.
Array of ResultEmployee

1. Build a map that contains a list of parameters that represent the values from the `IN` clause.
   Implement a loop to create the map. The map key mustn't be numeric: The example above uses a concatenation of "p" and the index.
2. Build the parameterized SQL statement and concatenate the parameters from the map.
   Get the map entries to an auxiliary array and reduce this array to a string containing the list of parameters. Then, you can use this string to populate the list of values for the `IN` clause.

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.

Mapping of Database Fields

In general, database-specific types are mapped to the Bridge 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.