

SQLite Deployment Options



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.

Using SQLite as DBMS there are two different options that may be used:

- The ordinary way would be to reference a database file located in the file system.
- Alternatively the database file can be added to the project as a resource using the resource importer.

Holding the database in memory only, without creating a database file, is not possible. The following paragraphs describe when using these options makes sense and how it is done in the component diagram.

Using tag [options](#) on the [SQLAlias](#) the following SQLite specific options can be set:

- `BusyTimeout=<time in milliseconds>` . The default is 60000 milliseconds. Used to avoid some [well known problems](#).
- SQLite PRAGMA statements: Used mainly for performance tuning. Details see below.

Example File (Builder projectAdd-ons/SQL):



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

On this Page:

- [SQLite Pragma Statements](#)
- [Ordinary SQLite Deployment Using a Database File in the File System](#)
- [SQLite Deployment Using a File Resource](#)
- [Known Problems Using SQLite](#)

Related Pages:

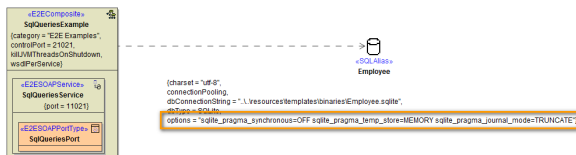
- [SQLite PRAGMA statement](#)
 - [sqlite_pragma_synchronous=OFF](#)
 - [sqlite_pragma_temp_store=MEMORY](#)
 - [sqlite_pragma_journal_mode=TRUNCATE](#)
- [xUML Service Settings](#)

SQLite Pragma Statements

The [SQLite PRAGMA statement](#) is a SQL extension specific to SQLite and used to modify the behavior of the SQLite database. The syntax is `sqlite_pragma_<pragma_name>=<pragma_value>`. The main use case is performance tuning. For example, the following PRAGMA options speed up inserting data (but also reduce data safety):

```
sqlite_pragma_synchronous=OFF, sqlite_pragma_temp_store=MEMORY,  
sqlite_pragma_journal_mode=TRUNCATE
```

The tag option is set in the component diagram, for example:



The meaning of these statements is:

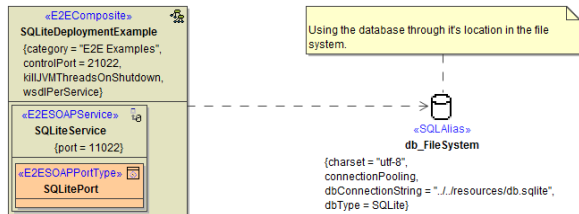
- [sqlite_pragma_synchronous=OFF](#): Disable wait for writes to complete (may increase performance by factor 50). Potential of database corruption on power failure.
- [sqlite_pragma_temp_store=MEMORY](#): Store temporary tables to memory.
- [sqlite_pragma_journal_mode=TRUNCATE](#): The TRUNCATE journaling mode commits transactions by truncating the rollback journal to zero-length instead of deleting it.

If you want to speed up your persistent state database, please look at option **Internal State DB Synch** on [Persistent State Components](#).

Ordinary SQLite Deployment Using a Database File in the File System

The ordinary way of using an SQLite database is the specification of a database file located in the file system (e.g. **C:\temp\db.sqlite**). The component diagram for the file system deployment option as follows:

Figure: SQLite Deployment Using a Database File in the File System



SQLite Deployment Using a File Resource

The SQLite database file (e.g. **db.sqlite**) can be imported using the resource importer. The import procedure is described in the E2E Builder User Guide, you can select the option "binary file" during the import.

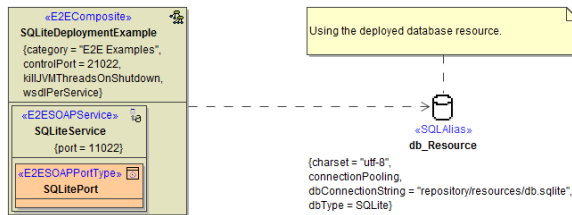
Using this special way, the database itself will be included in the repository during deployment. This has the following implications:

- The structure and initial data of the database need to be clear and set to an initial state before importing the database file as a resource.
- The idea of a resource is providing certain data to a model or service, in contrast to storing data from a model or service. Therefore, during each deployment process, the *.sqlite file is added to the repository and deployed to the xUML Runtime in its initial, original state (which it had when being imported). This means: During service execution, the database can be read and written to, however any changes will be overwritten during the next deployment procedure. This might be ok, if you use the database for management of temporary data in a service context, however it might not be suitable if the data needs to be persisted longer.
- Within the component diagram, the database can be referenced using a relative path (e.g. **.repository/resources/db.sqlite**) instead of an absolute path (e.g. **C:\temp\db.sqlite**). Therefore, no settings need to be adapted when deploying the service to different machines / environments.

Once deployed, you can replace the resource changing the resource path in the **SQL Adapter Connection** settings of the xUML service. See [xUML Service Settings](#) for more information on changing the settings of a service.

The component diagram for the resource deployment option looks as follows.

Figure: SQLite Deployment Using a File Resource



Known Problems Using SQLite

Using the SQL Adapter with SQLite database, you may get the following error:

```
[SQLSM][6][Error Message: 5 "database is locked". SQL Statement: ...]
```

This occurs, if multiple threads or processes want to read/write the SQLite database simultaneously. In case of concurrent writes, one write will fail. The xUML Runtime will retry to execute the write for 60 seconds. After the time-out, the message above is written to the bridgeserver log.

There are two possible approaches to solve this conflict (they may also be combined):

- Add value `BusyTimeout=<time in milliseconds>` to the tag **options** of the `<<SQLAlias>>`. Default is 60000 milliseconds - increase this value.
- Re-model the service to have short database transactions (including select) and add explicit commits to unlock the database frequently.