InPUTj Tutorial

SQL Database Transactions

Requirements

Below are the requirements for performing SQL database transactions via InPUTj:

- *Java SE* version 6 or higher (http://www.java.com/en/download/)
- InPUTj in the Java classpath (https://github.com/feldob/InPUT/)
- A JDBC driver in the Java classpath (e.g. the *PostgreSQL* JDBC driver, http://jdbc.postgresql.org/download.html)
- Access to a SQL database (e.g. a PostgreSQL database, http://www.postgresql.org/download/) where you have permission to execute the following types of statements:
  - `CREATE INDEX`
  - `CREATE SCHEMA`
  - `CREATE TABLE`
  - `DELETE`
  - `INSERT`
  - `SELECT`
  - `UPDATE`

  To import, only the `SELECT` statement is required.

Introduction

Many of the InPUTj methods that perform database transactions have a parameter of the type `java.sql.Connection`. If a JDBC driver is in the Java classpath, a `Connection` object can be created by calling the method `getConnection(String)` or `getConnection(String, String, String)` of the class `java.sql.DriverManager` (if you're using a JDBC driver lower than version 4.0, the driver must first be loaded by calling the method `Class.forName(String)` with the fully qualified name of one of the driver's `java.sql.Driver`-implementing classes as argument). Both of those methods takes a database URL of the form `jdbc:<`*`subprotocol`*`>:<`*`subname`*`>` as their first argument (e.g. `jdbc:postgresql://localhost:5432/postgres` if you're using the PostgreSQL JDBC driver and wish to connect to a PostgreSQL database named *postgres* on the local machine through port 5432), and the latter method additionally takes a username and a password as arguments.

When using InPUTj, all kinds of SQL database transactions (export, import, updating and deletion) can be handled via instances of the class `se.miun.itm.input.util.sql.DatabaseAdapter`. `DatabaseAdapter` has two constructors. One of them takes a database `Connection` as argument and creates a `DatabaseAdapter` that isn't thread-safe, and the other one takes the same argument as the former along with a `boolean` that specifies whether the `DatabaseAdapter` shall be thread-

safe or not.

All of the InPUTj methods that perform database transactions throw `SQLInPUTException`:s whenever a database error occurs. In addition, some of them throw `IOInPUTException`:s whenever an I/O error occurs and some throw `InPUTException`:s whenever another kind of error occurs. `SQLInPUTException` and `IOInPUTException` are both subclasses of `InPUTException`, and all three of these classes are members of the package `se.miun.itm.input.model`.

In order to more easily manage, export and import multiple related code-mapping objects, code-mapping `Document`:s can be added to instances of the class `se.miun.itm.input.model.mapping.Framework`. The `Framework` class implements the `se.miun.itm.input.aspects.Identifiable` and `java.util.Set` interfaces, and supports all "optional" set operations. When a code-mapping object is exported, data about which `Framework` it belongs to, if any, is also exported. This way, different sets of code mappings can easily be selected when importing `Exportable`:s.
`Framework` has two constructors, which both takes as first argument the ID that the `Framework` shall have. Aside from the ID, one of them takes a `java.util.Collection` of code-mapping `Document`:s as argument and the other one takes an arbitrary number of code-mapping `Document`:s, either as an array or as a sequence of arguments.

InPUT stores `Document`:s and `Exportable`:s in the same way in SQL databases.

Export

To export a `Document` or `Exportable` object to a SQL database, simply create a `DatabaseAdapter` and call its `export` method with the `Document`/`Exportable` as argument. This way, both the `Document`/`Exportable` itself and all objects that it's dependent on are exported to the database. The `export` method returns the value `true` wrapped in a `Boolean` if the export was successful, and the value `false` wrapped in a `Boolean` if there already is an object in the database that has the same ID and type as the one that was given as argument.

Below is a code snippet that illustrates how to export a `Document` and an `Exportable` to a database:

```
Connection connection;
DatabaseAdapter adapter;
Document document;
Exportable exportable;

// Initializes all the variables except 'adapter'.
...

try {
      adapter = new DatabaseAdapter(connection);

      adapter.export(document);
      adapter.export(exportable);
} catch (InPUTException e) {
      // Handles the exception.
      ...
}
```

Import

To import an `Exportable` object from a SQL database, simply create a `DatabaseAdapter` and call one of its import methods. `DatabaseAdapter` has four single-parameter methods called `importDesign`, `importDesignSpace`, `importExperiment` and `importInPUT`, respectively, which import and return an `IDesign`, `IDesignSpace`, `IExperiment` and `IInPUT`, respectively. Each of them takes as argument the ID of the `Exportable` that shall be imported, and each of them has a corresponding two-parameter method with the same name and return type that takes the same argument along with the ID of the `Framework` that the code mappings shall be selected from. All of the import methods imports both the specified `Exportable` and all objects that it's dependent on, including the code mappings. If a `Framework` is not specified, only frameworkless code mappings are imported. If the specified `Exportable` or any of its selected code mappings is not in the database, `null` is returned.

Below is a code snippet that illustrates how to import `IDesign`, `IDesignSpace`, `IExperiment`, and `IInPUT` objects from a database:

```
Connection connection;
DatabaseAdapter adapter;
IDesign design1, design2;
IDesignSpace space1, space2;
IExperiment experiment1, experiment2;
IInPUT inPUT1, inPUT2;
String      designID,
            experimentID,
            frameworkID,
            inPUTID,
            spaceID;

// Initializes 'connection' and the string variables.
...

try {
      adapter = new DatabaseAdapter(connection);

      design1 = adapter.importDesign(designID);
      design2 = adapter.importDesign(designID, frameworkID);

      space1 = adapter.importDesignSpace(spaceID);
      space2 = adapter.importDesignSpace(spaceID, frameworkID);

      experiment1 = adapter.importExperiment(experimentID);
      experiment2 = adapter.importExperiment(experimentID, frameworkID);

      inPUT1 = adapter.importInPUT(inPUTID);
      inPUT2 = adapter.importInPUT(inPUTID, frameworkID);
} catch (InPUTException e) {
      // Handles the exception.
      ...
}
```

Updating

To update a `Document/Exportable` object in a SQL database, simply create a `DatabaseAdapter` and call its `update` method with the updated version of the `Document/Exportable` as argument. This way, both the `Document/Exportable` itself and all objects that it's dependent on are updated. The `update` method returns the value `true` if the update was successful, and the value `false` if there is no `Document/Exportable` in the database that has the same ID and type as the one that was given as argument.

Below is a code snippet that illustrates how to update `Document/Exportable` objects in a database:

```
Connection connection;
DatabaseAdapter adapter;
Document document;
Exportable exportable;

// Initializes all the variables except 'adapter'.
...

try {
      adapter = new DatabaseAdapter(connection);

      adapter.update(document);
      adapter.update(exportable);
} catch (InPUTException e) {
      // Handles the exception.
      ...
}
```

Deletion

To delete a `Document`/`Exportable` object from a SQL database, simply create a `DatabaseAdapter` and call its `delete` method with an argument consisting of a `Document`/`Exportable` that has the same ID and type as the one that shall be deleted. The `delete` method returns the value `true` if the deletion was successful, and the value `false` if there was no `Document`/`Exportable` in the database that had the same ID and type as the one that was given as argument.

Below is a code snippet that illustrates how to delete `Document`/`Exportable` objects from a database:

```
Connection connection;
DatabaseAdapter adapter;
Document document;
Exportable exportable;

// Initializes all the variables except 'adapter'.
...

try {
      adapter = new DatabaseAdapter(connection);

      adapter.delete(document);
      adapter.delete(exportable);
} catch (InPUTException e) {
      // Handles the exception.
      ...
}
```