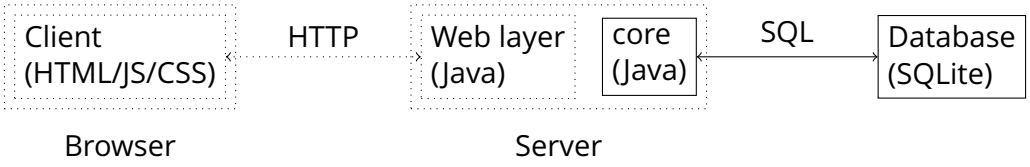
COMS20700 — Coursework 3

In this coursework you will design and implement server-/database-layer features of an application that every CS unit should have: an online forum for students to ask and answer questions relating to their courses.

We will be using SQLite for this assignment even though it is not the best choice for such an application: it has no concurrency support, nor any role-based access control. For all of this coursework, you do not need to worry about concurrency.

The general application architecture is as follows. You will work on the parts marked in solid lines, the parts with dotted lines will be provided for you.



A more detailed explanation of how the application handles a request from start to end will follow in Section 4.

1 Tasks

This coursework involves three tasks.

The deliverables for this coursework are as follows. With the exception of the individual reports, choose one of your group members who submits them on behalf of your group.

- A PDF report for tasks 1 and 2.
- A SQL create-drop script for task 1.
- The source code (java files) of your implementation (task 2).
- An individual PDF report (task 3) for each group member.

1.1 Task 1 — design the database schema

Your first task is to design a database schema for the forum application based on the features you will implement in task 2. There are different, valid choices how to do this and depending on how you choose to design the schema, different parts of task 2 will become easier or harder.

In your report I am looking for a description of why you chose to design the schema the way you did. You can address questions here like which elements you chose to normalise or not, and how this makes the API easier or harder to implement and/or to keep the database consistent.

You should first study the API (see task 2) and decide which of the features you want to implement first, then use those choices in designing the database schema. As you move on to other features you may, as a group, want to change the schema later on.

This task is group work. Your deliverable for this task is a PDF report and a SQL create-drop script, to be uploaded by one member of your group and containing the following information:

- The names and student IDs of all group members.
- An ER diagram of your database schema, in whatever notation you choose.
- A create-drop script in a `.sql` file that sets up your database schema when run on an empty SQLite database.
- A description (aim for 1–2 pages A4, maximum 3 pages) of the design choices involved in your schema.

As an additional restriction, the `Person` table has the following schema. It is shared with other university projects so its schema may not be modified.

```
CREATE TABLE Person (
    id INTEGER PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    username VARCHAR(10) NOT NULL UNIQUE,
    stuId VARCHAR(10) NULL
);
```

The `stuId` is the number on the back of a student's UCard and is NULL for staff. `id`, being a primary key, is a 64-bit integer (a Java `long`).

1.2 Task 2 — implementation

The specification of the API that you will implement is in the class named

`uk.ac.bris.cs.databases.api.APIProvider`

The comments in this class and in all other classes in the `api` package together form the application specification. For example, the `getForums()` method is defined in the `APIProvider` class to return a list of `ForumSummaryView` objects ordered by title; the `ForumSummaryView` class itself specifies that the `lastTopic` field should be NULL if there are no topics in this forum.

All your implementation must be in the package

`uk.ac.bris.cs.databases.api.cwk3`

or subpackages of this package. You will need one class that implements the API and you may make any number of other "helper" classes in order to have clean and well-designed code.

You should implement the API in a class `API` in the `cwk3` package as follows:

```
package uk.ac.bris.cs.databases.cwk3;

// imports

public class API implements APIProvider {

    private final Connection c;

    public API(Connection c) {
        this.c = c;
    }

    // TODO implement the methods

}
```

In the methods you can then assume that the connection field has been set to an open database connection. In other words you can do the following in the methods:

```
final String STMT = ...;
try (PreparedStatement p = c.prepareStatement(STMT)) {
    // do stuff
    // and return a result
} (catch SQLException e) {
    // handle it
}
```

Note that auto-commit will be turned off for the connection object that you get from the server class. In other words you need to take care of committing or rolling back any transaction that involves writing to the database.

The deliverable for task 2 is a ZIP (or TAR.GZ, 7Z) file containing your java files of the `api` package (and any subpackages). Make sure you include only java sources, not `.class` files.

The API contains methods organised into three different difficulty levels.

- Level 1 (6 methods). These are simple SQL statements. Every group should aim to implement all of these. I recommend that you implement these first to get a feel for the application structure.
- Level 2 (9 methods). These are slightly more involved than level 1, either because the SQL statement is more complex or there is writing as well as reading involved. Every group should aim to implement all of these.
- Level 3 (5 methods) These are more complex methods for extra credit. Make sure you have implemented all level 1 and 2 methods before you tackle any of these.

Each group member should implement at least one level 1 method and at least one level 2 method on their own. Beyond this, it is up to you whether you implement the other methods individually or as groups.

To track work allocation, include a table of who implemented which methods in your group PDF report. This table will also be used during marking if the distribution of work was not equal among the group.

1.3 Task 3 — individual reports

Each group member should write a short individual report (aim for 1–2 pages A4, maximum 3 pages) on their work on the project, addressing the following points:

- Which tasks were particularly easy/hard or interesting/boring?
- What did you learn during this coursework?
- What is your opinion of JDBC now that you have worked with it for a while? If you were designing your own database API what would you definitely do the same, or definitely do differently?
- Anything else you would like to mention (relating to the coursework).

You should also include a table showing the contribution of each group member as a percentage, which I will consult while marking.

1.4 Submission checklist

- Group PDF report contains names/student IDs, ER diagram, group report and table of who implemented which methods.
- SQLite create-drop script runs on an empty SQLite database.
- ZIP file contains all java source files that you have implemented yourselves, and only these files.
- Application runs when following the setup procedure (part 2) below.
- Individual PDFs contain a table of everyone's contributions (percentages) and individual reports.

2 Setup Procedure

Run part 1 when you first download the coursework files. Run part 2 before you submit to make sure that your application works.

2.1 Part 1 — initial setup

I am assuming that you have a java development kit (JDK) version 7 or 8 and the `ant` build tool installed. The following instructions use the command line; you can also use an IDE such as netbeans, eclipse or IDEA if you are more familiar with that.

1. Download `cwk3.zip` from the databases website and unzip it.
2. In the folder `bb/lib/` there is a README file indicating the dependencies of the project. Download these as instructed and place the jar files in `bb/lib/`.
3. In the folder `bb/database/`, open a console and type the command `sqlite3 database.sqlite3` then run the commands `.schema` and `.quit` to set up an empty database. The schema command forces an empty database to be created in the file indicated.
4. In the `bb/` folder, run `ant compile` to compile followed by `ant run` to start the application. If successful this should print `[java] Server started, Hit Enter to stop.` and you should be able to access the following URLs in your browser:
   - `http://localhost:8000` displays "Hello, World!"
   - `http://localhost:8000/forums` displays the error "Error: java.lang.UnsupportedOperationException : Not supported yet." This comes from the method `getSimpleForums()` in the API.

2.2 Part 2 — final checks before submission

1. Re-run the initial setup in a fresh folder.
2. In the `database/` subfolder of the folder that you just created, run the following where `setup.sql` is your create-drop script (copy the script to that folder):

   ```
   sqlite3 database.sqlite3
   .read setup.sql
   .schema
   ```

   Check that the schema command shows all the tables you need, then continue:

   ```
   .read setup.sql
   .schema
   .quit
   ```

   This ensures that your create-drop script works both on an empty database and to rebuild a database that already contains tables.
3. Copy over the java source files that you wrote to the `cwk3/` folder under `src/`.
4. `ant compile` then `ant run` and test some use cases such as creating and looking up users, forums, topics and posts.
5. If all this runs fine then you're good to submit. Make sure that you submit your create-drop script as an SQL file and a zip of the source files that you copied over in step 3.

3 API notes

JDBC methods throw a `java.sql.SQLException` if something goes wrong. The API methods wrap one (sometimes more than one) SQL query and use result objects to signal one of three states: success, failure (e.g. a query to look up an object did not return any rows) and fatal (invalid query, database down etc.). The result object is implemented in `api.Result` and should be used as follows.

For a method that should return some data, run the SQL query, extract the values you need from the `ResultSet` into the object(s) that the api should return and then run `return Result.success(data);` where `data` is the object you want to return.

If something went wrong that is the caller's fault, e.g. passing an empty/null parameter where this is not allowed or passing an id that does not refer to any object in the database, run `return Result.failure(message);`. The message can be anything you choose and it will be displayed in the browser.

If something goes wrong that is not the caller's fault, e.g. a SQLException during a read operation, run `return Result.fatal(message);` with a message — the message can be `e.toString()` where e is the exception you got, if you like.

For example, here is how one could implement a fictional API method:

```
public Result<Long> getId(String username) {
    if (username == null || username.equals("")) {
        return Result.failure("Need a valid username");
    }

    final String SQL = "SELECT id FROM User WHERE uname = ?";
    try (PreparedStatement p = c.prepareStatement(SQL)) {
        p.setString(1, username);
        ResultSet r = p.executeQuery();
        if (r.next()) {
            Long id = r.getLong("id");
            return Result.success(id);
        } else {
            return Result.failure("No user with this name");
        }
    } catch (SQLException e) {
        return Result.fatal("Something bad happened: " + e);
    }
}
```

First we validate the parameter to make sure it's not null or empty so we definitely won't get a null pointer problem later on. If this check fails, it was the caller's fault so we return failure. Otherwise we do the standard JDBC procedure and return the id wrapped in a `Result.success` object if there is such a user or a failure if not. If there was no such user, it's the caller's fault for not providing us with a valid username. If there's something else wrong (e.g. database down, typo in our SQL query) then we return a fatal error. And of course we use a prepared statement so as not to be vulnerable to SQL injection.

Some methods (such as ones that write to the database) do not need to return any data. In this case you just `return Result.success();` when you're satisfied that the job is done. These methods are declared to return a plain `Result` rather than a `Result<Something>`, e.g. `createForum`.

In methods that write values such as `addNewPerson`, it is the caller's fault if they try to create a new person with a username that already exists (there is a unique constraint on usernames). The pattern to use here is first check with a SELECT whether such a user already exists and return failure in this case. Then issue the INSERT. Any problem with the INSERT, whether due to a constraint violation or a

database problem or a typo in your SQL will result in a SQLException — one could parse the error message returned in the exception to see if it was a constraint violation but this is ugly and depends on the driver/database's error message format. SELECT then INSERT is the standard pattern to use in this case.

The view objects that the API methods return inside the result wrappers, such as `ForumView`, are immutable value objects. This means that you have to extract all the necessary data first and then create the object with its single constructor that takes all the required values. Some of these constructors contain checks that will throw an exception if you pass an empty or null value for a required field. You do not need to catch these exceptions — you need to code in such a way that they can't happen in the first place.

In some cases, such as `getTopic()`, the view object (`TopicView`) contains a collection of child view objects (`PostView`). In this case, you probably want to create the collection of child objects first and then create the parent object at the end, since the children already have to exist when you call the parent constructor.

4 Overview of the application

This section is for information only. The web layer is not part of the coursework.
Here's what happens when you visit `http://localhost:8000/forums` in your browser with the application running.

1. The `Server` class in the `web` package is the main class of the application. It starts nanohttpd listening on port 8000 and it opens a database connection to textttdatabase/database.sqlite3 in the main method. It also sets up the freemarker template engine for the page templates in `resources/`. The server class also creates an instance of the API with the open database connection and registers it with the application context so that handlers can access it. This all happens once when the application starts.
2. The `addMappings()` method defines handlers for different URL paths. We asked for `/forums` so an instance of `ForumsHandler` is created.
3. Nanohttpd calls the `get()` method on the forums handler, which is implemented in the `AbstractHandler` superclass. This delegates to `handle()` which in turn calls `render()` which is implemented in `SimpleHandler`, the direct superclass of the forums handler. `render()` first checks if an URL parameter is needed: for example, to view an individual forum using the URL `/forum/100`, the parameter 100 is required. In this case we don't need one (`ForumsHandler.needsParameter()` returns false) so we pass control to the `simpleRender` method.
4. The forums handler class asks the application context for the API implementation and calls the `getForums()` method to get the data. It passes this back

to `SimpleHandler`, requesting that if successful the data should be displayed with the `ForumsView.ftl` template (which lives in `resources/`).

5. The API call returns a `Result` object that indicates whether the call was successful. If not, `SimpleHandler` creates an error message to show to the user. If successful, `SimpleHandler` calls `renderView()` which is implemented in the superclass `AbstractHandler`.
6. `renderView()` runs the freemarker template engine with the requested template. The `ForumsView.ftl` template contains HTML mixed with parameters that get injected from the data object returned by the handler. In this case, the data object is a list of `ForumSummaryView` objects that contain a title, an id and a `SimpleTopicSummaryView` of the last updated topic in the relevant forum, which contains the topic id, forum id and topic title.
7. The template operation returns a `View` containing the HTML page to be displayed. This goes back to the `handle()` method in `AbstractHandler` which writes out the HTTP response.

5 Marking

This project will be marked according to university marking guidelines for masters' level units. A PDF of the exact scheme is available at `http://goo.gl/2WCxBE` under "Table 1 and Table 2" and the rough correspondance of marks to outcomes is the following:

up to 40% All or major parts of the assignment not completed as required, little if any evidence of relevant knowledge or abilities.

40%–50% Some relevant work done but clearly below expectations.

51%–60% Good enough for a pass, but fewer tasks competed or of lower quality than expected.

61%–65% You have done more or less everything you were asked to.

65% This (not 100%) is the mark you get if you've done everything you were asked to, no more and no less.

66%–70% Good work, better than required in at least one aspect.

71%–85% Excellent work, clearly exceeds the expected outcome.

above 85% Outstanding work, beyond what we would expect any student to achieve on this unit.

Top marks reflect quality, not quantity. There are page limits on the reports and there is a fixed number of API functions to implement. A particularly well done implementation of the level 1 and 2 API calls is worth more than a badly done implementation of all 3 levels.

5.1 Things to avoid

The following mistakes will result in marks being deducted:

- API calls that are vulnerable to SQL injection. If in doubt, stick to prepared statements.
- API calls that throw unexpected excpetions, e.g. `NullPointerException` if the caller passes in a null parameter (you should check the inputs and return failure if you get a bad one).
- Running SQL queries in a loop when there is a good way to avoid this.

  If you find yourself doing SQL in a loop because your schema makes it hard to write a particular API call, this is an opportunity to call a group meeting and discuss adapting the schema (and it gives you something to write about for task 1).
- Using multiple queries when there is an obvious way to use a single one, e.g. with a join.

  You do not have to use exactly one query per API call, indeed in some cases it's necessary to use more than one (SELECT then INSERT as described earlier).

  If you want to find a particular topic and the name of the person starting it, for a particular choice of schema the following SQL may do what you want:

```
SELECT ... FROM Topic JOIN Person ON Topic.author = Person.id
```

  In such a case, if you run two queries `SELECT .. FROM Topic` and then `SELECT ... FROM Person WHERE id =` with the author of the topic you just retrieved, you will be marked down for unnecessarily using two queries. In cases where there is not an obvious solution, don't be afraid to use more than one query though.
- SQL issues such as joining on a table that you're never using in the query, joining on something that's not a candidate key etc.
- API calls that return wrong results or do not match the specification (i.e. results not sorted the way they're supposed to be).
- Bad coding practices, e.g. inconsistent indentation, huge methods that would be better off split into smaller parts.
- Bad use of JDBC, e.g. not closing statements or result sets (and not using try-with-resources) or writing to the database without guaranteeing either a commit or a rollback before the method returns.

5.2 Things to aim for

The following are examples of features that will give marks.

- Correct and efficient use of SQL and JDBC.
- API implementation meets the specification including in corner cases (e.g. `getForum()` on a forum that has no topics).
- Correct handling of errors, including bad inputs and SQLExceptions.
- Good coding style: small methods that do one thing each, consistent indentation etc.
- Correct use of public/private (e.g. in the API implementation only the API methods and the constructor should be public).
- Repeated tasks split off into methods or classes of their own rather than copy-pasting the code.
- Some documentation (javadoc) on any methods or classes that you create yourselves. You do not need to document the API methods — this is already done in the `APIProvider` class.

Writing unit tests for the API is *not* part of the coursework.

5.3 Libraries

The coursework should be completed using only functionality available in the Java runtime (and the xerial driver for SQLite, though you do not need to talk to the driver directly). You should not need any third-party libraries and in particular you should not use any other database or ORM libraries.

This is the end of coursework 3.