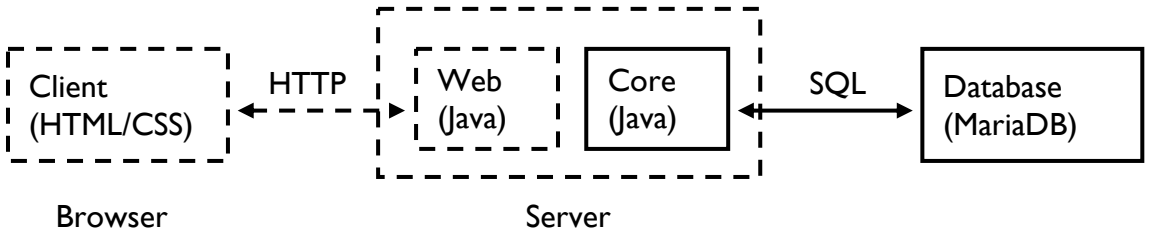
# Databases Coursework 2, 2020 (Individual Assessment Version)

## Overview

In this coursework you will design and implement server-/database-layer features of an online forum application.
The general application architecture is as follows. You will work on the parts marked in solid lines, the parts with dashed lines will be provided for you.




There are up to four deliverables for this coursework, depending if task 3 is attempted.

- A single PDF report for tasks 1 and 3 (the latter if this is attempted).
- A SQL create/drop script for task 1.
- A ZIP file containing the source code (java files) of your implementation (task 2).
- A SQL create/drop script for task 3 (if attempted).

**Please submit these files to Blackboard as a single zip file.**
(see 'Submission Checklist' section later for more details)

## Tasks

### Task 1 – design the 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.
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 want to change the schema later on.
The deliverables for this task are:

- A create/drop script as a ".sql" file.
- A schema description in your report (in the PDF report file).

In your report, we are 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 may refer to anything relevant from any of the Databases lectures.
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 AUTO_INCREMENT,
  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 32-bit integer (a Java int).

## Task 2 – implement the methods

The methods that you should implement are specified in the interface

```
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. The APIProvider interface contains a description of each method to implement. 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.cwk2
```

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 cwk2 package as follows:

```
package uk.ac.bris.cs.databases.cwk2;
// 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 initialised 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 the exception
}
```

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 file containing your java files of the api package (and any subpackages). Make sure you include only java sources, not .class files.

### The Result class

Many of the API methods return an instance of the result class. This class can represent one of three states:

- success – data is available.
- failure – the method caller did something wrong.
- fatal – something went wrong that is not the caller's fault.

All JDBC methods can throw the checked exception `java.sql.SQLException` so you will need to run your database code in try/catch blocks and handle this exception. You should also use the try-with-resources pattern to automatically close statements that you are done with.
In your SQLException catch block, you should return a `Result.fatal` since an SQLException should never be able to be caused by bad input (after all, we care about security). The following code snippet will cause the web interface to display an indication of what went wrong:

```
catch (SQLException e) {
    return Result.fatal(e.getMessage());
}
```

The most likely cause to trigger an SQLException, at least when you start developing, is a syntax error in your SQL.
When you implement a method, the first thing you should do is check any input parameters.

- You do not need to check the connection (c) object – you can assume that this is always valid. If something is wrong with the connection, your method will not get called in the first place.
- If a parameter has restrictions such as "must not be null and/or empty" that do not require a database lookup to validate, check these first. If anything is wrong, return a `Result.failure` with an appropriate message.
- If a parameter is supposed to point to a database object (such as the username in getPersonView) then you need to check whether such a person exists in the database or not. If they do not but the method specification says they should, you should return a `Result.failure` with an appropriate message.

You are allowed to use more than one SQL query per method – this is not coursework 1 anymore! However, try not to use multiple queries when there is an obvious way to use a single one, e.g. with a join. One pattern is to start off with queries to check all parameters, return failure if something's wrong, then run the queries to do the method's actual job. But sometimes you can combine the "checking" and "working" queries into one.
If all parameters are ok and your method could complete its job, you should `return Result.success(value);` where `value` is whatever value your method was supposed to produce.
As a starting point, we provide the implementation for two of the methods, namely getUsers and addNewPerson. To get an idea of how the implementation works, the getUsers method has the syntax

```
public Result<Map<String, String>> getUsers();
```

There are no parameters so there's no failure case here. First, we run a SQL query to fetch all users and put them into a map (stored in a variable called `data)` of usernames to names (as specified in APIProvider.java).
We then run

```
  return Result.success(data);
```

If there are no users, that's not a failure – in this case we just return an empty map. In the SQLException catch clause for the JDBC methods, we return `Result.fatal` if there's an exception.
As an example of a method whose implementation you have to provide, the method `getPersonView(String username)` has the precondition "username cannot be empty". Not empty (not the empty string) always includes not null, but not vice versa. So the first thing you can do in this method is:

```
if (username == null || username.isEmpty()) {
    return Result.failure("getPersonView: username cannot be empty");
}
```

Note that we're checking for null first, as calling isEmpty (or any other method) on a null parameter would cause a NullPointerException.

### Structure of the methods

The methods are as follows. Methods have a difficulty (level) ranking from one to three stars.

- 1 "Person" methods.
  We provide the code for two of these methods, whereas you need to implement the third one.
  - GetUsers (*) [already provided]
  - addNewPerson (**) [already provided]
  - getPersonView (*)
- 2 Forum-only methods
  These are self-contained and you should implement them before the next subtask (point 3).
  - getForums (*)
  - createForum (**)
- 3 Forums, topics, posts
  This section forms the main part of the application.
  - getForum (**)
  - getTopic (**)
  - createPost (** - ***)
  - createTopic (***)
  - countPostsInTopic (*)

## Task 3 – "like" functionality

For extra credit, imagine that your application has now been deployed and is live for end users, but now you are required to add further functionality: a "like" feature. This functionality allows users to "like" either a post or a topic.

For this task, you will need to revise the database schema designed in task 1 in order to record the likes, write additional SQL queries and text (for the subtasks described below), but you won't need to modify the actual application; i.e. there is NO code writing required.

You should attempt this task only if you have successfully completed the earlier tasks and have time to spare. Note that implementing all methods in task 2 well will get you higher marks than attempting both task 2 and task 3 with significant mistakes.

The subtasks are as follows:

a) Update your DB schema to accommodate the "like" feature and explain the functionality of the changes in the report (max. 2 pages).
b) Provide functional SQL queries in your report for TWO of the following operations, noting any particular highlights of your query design (max. 2 pages):
    - Recording a "like" for a post.
    - Getting the total number of topics and posts liked by a specific user.
    - Getting the names of all people who have liked a specific topic, ordered alphabetically.
c) Deploying new schema functionality to an existing application can be dangerous. Why? How does your schema design mitigate these issue(s)? (max. 1 page in report).

The deliverables for this tasks are:

- A create/drop script as a ".sql" file for your revised schema.
- Your PDF report containing:
    - the description of the revised schema in your report;
    - your SQL queries and the answer to part c above.

## Setup and submission procedures

### Initial setup

You should run this procedure once when you start working on this coursework to set up your development environment. We are assuming that you will be using a personal computer for your development rather than a University lab machine.

This coursework requires the Java 11 JDK to be installed on your PC/Mac, along with the ant build tool and the database virtual machine (DB VM) we have already been using for Coursework 1. If you have any issues with getting your environment set up, please see the Support section at the end of this document for help.

The following instructions use the command line; you can also use an IDE such as Netbeans, Eclipse or IntelliJ if you are more familiar with that.

1. Download `student.zip` from the databases website and unzip it (it doesn't matter where you unzip it).
2. In the folder `cw2-student/lib/` there is a README file indicating the dependencies of the project. Download these as instructed and place the jar files in `cw2-student/lib/`.
3. In the `cw2-student/` folder, run `ant compile` to compile the program.
4. Make sure the MariaDB server on the DB VM is running (ensure you can log in using the mysql command) and run `ant run` to start the application. If successful, this should print some information lines ending in:

```
[java] Server started, Hit Enter to stop.
```

…and you should be able to access the following URLs in your browser:

a. `http://localhost:8000` displays `"Hello, World!"`
b. `http://localhost:8000/people` displays a list of users (since the getUsers method has already been implemented for you).
c. `http://localhost:8000/forums` displays an error message (since this method is not implemented yet).

### Compiling and running

You can compile the program from the `cw2-student/` folder with `ant compile` and run it with `ant run`. You can stop the running program by pressing ENTER in the terminal in which you launched it.

## Final checks before submission

You should run these checks before submitting your final zip to the CW2 BlackBoard page (note that all VM operations are also described in the README.txt file included in the DB VM folder zip):

1. Test your create/drop script(s) on a fresh database VM, but first ensure you have backed up (exported) your bb schema to a file on your PC/Mac (not just within the VM), along with any other schema that you wish to retain (see the Import/Export section in the DB VM README file). Then run the following commands in the dbvm folder (this destroys ALL existing database data!]:
    a. Type: `vagrant destroy`
    b. Type: `vagrant up`
    c. After the new VM is deployed and running, log in to MariaDB as normal using the mysql command, select the bb database and import your bb schema file several times to ensure no issues.
2. Re-run the initial cwk2 setup in a fresh folder and copy over the java source file(s) that you wrote to the correct subfolder under this fresh folder.
3. `ant compile` then `ant run` and test some use cases such as creating and looking up users, forums, topics and posts.
4. 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 2.

## Submission checklist

- ☐ **Create a single zip file for upload, named your UoB username (e.g. ab12345.zip), containing the following files:**

- ☐ Report (PDF) for Task 1 and optionally Task 3 if you have attempted it.
  **filename: report.pdf**
- ☐ SQL create/drop scripts for Task 1 and optionally Task 3 if you have attempted it.
  **filename(s): task1.sql, task3.sql**
- ☐ ZIP file of your Java code for Task 2.
  (Source files only – no .class files in the ZIP file.)
  **filename: code.zip**

**Submission deadline: Friday 15th May 2020, 7pm BST**

Blackboard Submission Page:

https://www.ole.bris.ac.uk/webapps/assignment/uploadAssignment?content_id=_4113737_1&course_id=_237225_1&group_id=&mode=view

## Marking notes

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".

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 task 2 methods is worth more than a badly done implementation of both tasks 2 and 3.

### 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.

### Things to avoid

The following mistakes will result in marks being deducted:

- API calls that are vulnerable to SQL injection. Use prepared statements.
- API calls that throw unexpected exceptions, 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 consider 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 are 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 (by not using try-with-resources) or writing to the database without guaranteeing either a commit or a rollback before the method returns.
- Submitting to Blackboard incorrectly, such as uploading separate non-zipped files, using RAR instead of zip, or wrong naming conventions.

### Libraries

The coursework should be completed using only functionality available in the Java runtime (and the MariaDB driver, 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.

## Additional information

*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 the database in the main method. It also sets up the freemarker template engine for the page templates in resources/. The server class then 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 de1nes 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() 1rst 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 SimpleTopic-SummaryView 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.

## Support

If you need any technical support or clarification on this assessment, FIRST check the unit discussion boards (forum) on BlackBoard:

https://www.ole.bris.ac.uk/webapps/discussionboard/do/conference?toggle_mode=read&action=list_forums&course_id=_237225_1&nav=discussion_board_entry&mode=view

For an individual support session if you're really stuck, book an appointment at the following page with a member of the unit staff during the duration of the assessment:

https://outlook.office365.com/owa/calendar/COMSM0016@bristol.ac.uk/bookings/

You will need to enter your UoB email address and have a personal Skype account ID available for communication with one of our staff during the session.

Note that initially, we will only be running support sessions during scheduled unit lab hours, i.e. Mondays 2-5pm BST.

### Database Virtual Machine (DB VM)

Finally, for the unit's DB VM, see the 'Lab Database Setup' section of the unit webpage on BlackBoard:

https://www.ole.bris.ac.uk/bbcswebdav/courses/COMSM0016_2019_TB-2/web/index.html

Note that the DB VM supports Windows 10 PCs running the Linux subsystem and Mac computers, or any PC that can run the latest versions of Oracle VirtualBox and Vagrant, which both must be installed before the DB VM setup.sh script is run. It's also recommended that the MySQL client is installed, but this is not entirely necessary – see the DB VM README.txt file for more information. For technical support of the DB VM, please check the Forum or book a support appointment, as outlined above.