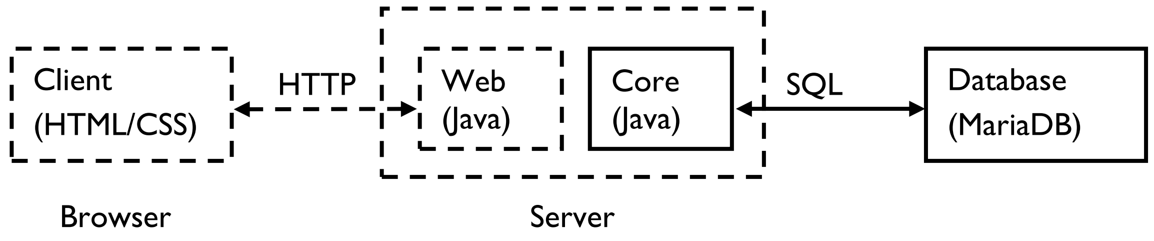
Databases Coursework 2

Overview

In this coursework you will work in groups to 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 four deliverables for this coursework. Everyone should submit an individual report. Choose one member of your group to submit the other deliverables.

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

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, as a group, want to change the schema later on.

The deliverables for this task are:

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

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 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 64-bit integer (a Java long).

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.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 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! 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. For example, the getUsers method has the syntax

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

There are no parameters so there's no failure case here. You want to run a SQL query to fetch all users and put them into a map of usernames to names (as specified in APIProvider.java). If you succeed and you map is in a variable called `map`, you can run

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

If there are no users, that's not a failure – in this case you just return an empty map. In the SQLException catch clause for the JDBC methods, you return `Result.fatal` if there's an exception.

For another example, 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 ranking from one to three stars.

Group "A" contains methods that every group of students should implement.

- A.1 "Person" methods.
  You should implement these first as they provide a self-contained introduction to the coursework and cover the main areas (fetching an individual item, fetching multiple items, creating a new item).
  - getUsers (*)
  - getPersonView (*)
  - createPerson (**)
- A.2 Forum-only methods
  These are also self-contained and you should implement them before A.3.
  - getSimpleForums (*)
  - createForum (**)
- A.3 Forums, topics, posts
  This section forms the main part of the application.
  - getForums (**)
  - getForum (**)
  - getSimpleTopic (**)
  - getLatestPost (**)
  - createPost (** - ***)
  - createTopic (***)
  - countPostsInTopic (*)

Group "B" contains extra methods for additional credit.

- B.1 "likes"
  This section adds a feature to "like" topics and posts.
  - likeTopic (**)
  - likePost (***)
  - getLikers (**)
  - getTopic (***)
- B.2 "joins"
  These are more complex views that include e.g. like counts. You should implement at most one of these, and even that only if you are confident in all your other solutions and have time to spare.
  - getAdvancedForums (***)
  - getAdvancedPersonView (***)
  - getAdvancedForum (***)

Task 3 – reports

Each group member must submit an individual report in PDF format. Aim for 1 page A4, maximal 2 pages. Discuss 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).

In addition, one member of your group should submit the group report as described in Task 1.

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.

I am assuming that you have a java development kit (JDK) version 7 or 8 and the ant build tool installed (these are installed on the lab machines). 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 `student.zip` from the databases website and unzip it (it doesn't matter where you 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. If you are on a lab machine, add the following line to the end of your `~/.bashrc` file, then close and re-open any terminals:

```
alias ant=/usr/bin/ant
```

4. In the `bb/` folder, run `ant compile` to compile the program.
5. Make sure the MariaDB server is running (`database start`) and run `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:

   a. `http://localhost:8000` displays `"Hello, World!"`

   b. `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 `bb/` 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

One person in your group should run these checks before submitting to SAFE.

1. Shut down any running MariaDB instances (`database stop`). Make a copy of your database (mysql/ folder), then delete this folder and recreate it from the original database ZIP file. Then start the MariaDB server and client, select the bb database and run your create-drop script to create the tables. Check that your script produces no errors when run several times in a row.

2. Re-run the initial cwk2 setup in a fresh folder and copy over the java source files 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 3.

Submission checklist

- ☐ ONE group member: group report (PDF) for Task 1.
- ☐ ONE group member: SQL create/drop script for Task 1.
- ☐ ONE group member: ZIP file of your Java code for Task 2.
  Source files only – no .class files in the ZIP file.
- ☐ EVERY group member: individual report for Task 3.

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" 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. You get this for a decent implementation of group "A".
- **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 group "A" methods is worth more than a badly done implementation of both groups.

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 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 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.

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.

Using your own machine

It is recommended that you work on coursework 2 on the lab machines as the whole application has been tested there. If you wish to use your own machine and you are confident with setting everything up yourself, the following instructions may help. If you do this, please bear in mind two things:

- Your submission must run on the lab machines. If you submit something that only works on your own machine, you will lose marks – test on a lab machine too!
- You will not get any "tech support" for installing and configuring MariaDB on your own machine.

If you have a UNIX/Linux machine (this includes Mac OS) then you might be able to install MariaDB, use the provided sample databases from the ZIP file and work with the coursework 2 files just like on a lab machine.

The following instructions should help you set up the coursework on your own machine under any operating system. If you do not understand what some of the steps mean, you should be using a lab machine.

1. Make sure you have an up-to-date Java (JRE + JDK) environment installed. Typing "javac -version" on the command line shows you if you have the JDK installed. You need at least version 7 (shown as "1.7" in the version output).
2. Install "ant" with your system's package manager if it is not installed already (typing "ant" on the command line will show you if it is).
3. Install MariaDB and create a user 'student' with no password and a database 'bb'. Give the student user full access (GRANT ALL) to the bb database.
4. Make sure that MariaDB either starts automatically with your machine, or that you know how to start and stop it. The "database" script will not work! (Usually when you install MariaDB the default is to start automatically, but there might be an option to choose this in the installer program.)
5. Unzip the student.zip file and edit the bb/build.xml file: comment out the <arg> item in the run target.
6. When downloading the libraries from lib/README.text, leave out the two JNA ones.
7. If your system requires this, allow applications to use port 3306 in your firewall settings (you might get a warning dialog the first time you run MariaDB and/or the Java application that you're working on).

The commands "ant compile" and "ant run" should now work as expected, the latter of course requires the database to be running. On your machine, the Java application will connect to MariaDB over port 3306 which is the usual way of connecting to a MySQL or MariaDB database.