**COMP5013 Web Application Development: Coursework**

**Allocation**

This coursework is worth 100% of the marks for module COMP5013.

You will also need skills from the following modules:

* **COMP4004** (Python Programming)
* **COMP4003** (Design reporting)

**Specification**

**Task**

Your task is to design, develop and test the front- and back-end implementations for a web application as described below.

The coursework is intended to familiarise you with the fundamentals of how the web and web applications work, not to duplicate the commercial web development process. The back-end of the application must be written in Python 3 using the flask framework. The front end must make use of JavaScript.

Your application will make use of a database. You will be provided with a back-end database schema that can be run under sqlite. Sqlite would not normally be used for a deployed web application, but you may use it for this coursework as it is simpler than running an actual SQL server and SQL is not the focus of this module.

**Application**

The web application you must produce is a semantically enhanced debate forum. Users can create accounts with a username and password, post messages, and read messages that other users have posted. However, this application differs from typical internet forums in that it allows messages to be connected to each other in multiple meaningful ways. There is no existing online service which exactly matches this specification, but a system with similar features can be seen at <http://kialo.com/> .

The forum is divided into **topics**. Each topic can contain one or more **claims**. Each **claim** has a title and some text, plus additional **replies** can be posted to it. A user can post new topics, new claims on an existing topic, or new replies on an existing claim.

When posted, a reply must be related to either the claim it is posted to, or another reply on the same claim. If the reply is related to the claim, the user can choose if it is a *clarification*, *supporting argument*, or *counterargument*. If the reply is related to another reply, the user can choose if it is *evidence*, *support*, or a *rebuttal*. When posting a reply, the user must explicitly choose one of these types, and it must be stored in the database alongside the text of the reply.

There is no limit to the “depth” with which a reply can be associated with another reply. A reply may be made to a reply to a reply.

In addition, claims can be related to other claims, as *opposed* or *equivalent*. **Note that each claim can potentially be related to more than one other claim.**

The precise functionality to implement is as follows:

1. On visiting the first page of the site, the user must see a suitable site banner and user interface, plus a list of *topic*s.
2. Clicking on a topic should display:
   * all **claims** posted in that topic, with the most recently updated (newly posted or with a new **reply**) at the top.
   * a summary of the opposed/equivalent relationships between these claims.
3. Clicking on a claim should display:
   * the claim header and top message,
   * all related **replies**, arranged according to their relationship with the topic. Replies related directly to the claim should appear below it. Replies related to other replies should appear below, or within, the reply they relate to, with the type of relationship clearly visible.
   * links leading to related opposed/equivalent claims.
4. Each topic, claim, or reply should also display the username of the posting user, and the time of posting. Replies should also show their relationship type to the claim or to the previous reply.
5. Topic and claim pages should be directly linkable. If a user copy and pastes the URL of a topic or claim page, anyone else visiting that URL should arrive at the same topic or claim page.
6. A user must be able to create an account, giving a username and password. You may ask the user for their name and e-mail address also. There is no need to verify e-mail addresses.
7. A user must be able to log into an existing account with the correct username and password.
8. Using a direct link to a topic or claim page, as described in item 5, should not change the users logged-in status. In particular, they should **not** be logged in as the user who copy and pasted the URL.
9. The front page, topic, and claim pages should be accessible without logging in. If a user is already logged in, visiting these pages should not log them out.
10. The user interface to register an account or log in should appear as a “log in” button in the corner of each regular page. When clicked, this button should expand into a box overlaying the existing page, offering the username and password prompts and any others necessary. The page should not be reloaded during this process.
11. When a user is logged in, the log in button should be replaced by a log out button.
12. Once a user has logged in, they should be able to post topics, claims, and replies.
13. When posting a reply, the user should be able to write the text of the reply and choose what it is a reply to (the claim, or another reply) and what type of relation it has (from those listed above). A posted reply **must** be related to either the claim, or another reply by one of the listed relation types.
14. When posting a claim, the user should be able to write the header message of the claim and choose to relate it to one **or more** other claims. A claim is not required to relate to any other claims, although it must be part of a topic.
15. If separate URLs are used for the pages for posting replies, claims, and topics, then directly linking to them when not logged-in should send the user to the front page. The user should not have the opportunity to enter text before being sent to the front page.
16. The site should not be vulnerable to deep linking, nor SQL, JavaScript (XSS) or HTML injection.
17. The user interface should be of a standard suitable for a modern web application.
18. The application should meet reasonable standards for accessibility on desktop platforms. It does not need to be accessible via mobile platforms.

**Please note the following differences from “standard” forums:**

* The user must state the type of relationship a reply has to another reply, or to a claim.
* Claims can relate to other claims. Users must state the type of relationship in this case also.
* Each claim can relate to more than one other claim.

Completing all of the above to a reasonable standard will result in a mark up to 70% depending on coding and design quality.

To gain the remaining 30% marks, you must design and implement the website using AJAX. Data should be fetched from the server in JSON format using AJAX requests, and used by JavaScript code to dynamically update the visible page. You must design the communication protocol used. You can choose which parts of the site will use AJAX communication, but AJAX should be the standard mode of operation of the site; for any function that does not use AJAX, you must describe the reason why not in your design report.

**Restrictions**

Your website must be built using HTML 5 and JavaScript (ECMAscript) on the client, and Flask, sqlite and Python 3 on the server.

Your web site must run in a Python virtual environment built by running the following commands from a Command Prompt or PowerShell in a suitable working directory:

pip install virtualenv

virtualenv webapp

cd webapp

scripts\activate

pip install flask

Having entered these commands, you must not copy additional external Python files into the virtual environment nor pip install any other modules. You may use all modules that are path of the standard Python installation, and all modules that are supplied with Flask.

sqlite does not need to be installed as it is already built into recent versions of Python.

It is recommended that you also use this virtual environment for development. Regardless, you must test your site within the virtual environment before submitting it. This is the virtual environment in which your code will be tested by the assessor. If it does not work, marks will not be awarded.

In addition to the libraries in the above virtual environment, you may use the jQuery library for JavaScript. You may copy the jQuery library’s JavaScript file into the virtual environment.

You will also need to install the pre-provided database schema. To do this, download the provided database schema from Moodle into a file 5013dbinit.sql, place it in the virtual environment directory, and then run the following Python program in that directory.

import sqlite3

db = sqlite3.connect("debate.sqlite")

cursor = db.cursor()

with open("5013dbinit.sql") as f:

cursor.executescript(f.read())

db.commit()

db.close()

Once you have done this, you can delete this program file and the .sql file. Doing so may be a good idea, as running this program a second time will reset the database, deleting any data added to it.

**You may not use any other frameworks or libraries** other than those installed by the above commands. You may use all Python built-in libraries and all libraries supplied with Flask. You may **not** use Flask extensions that are not part of the Flask install package (eg, flask-sqlalchemy), nor independent Python libraries (eg, peewee), nor CSS libraries (eg tailwind), nor JavaScript frameworks (eg, bootstrap or angular) , nor other JavaScript libraries (eg, popper).

**Any functionality implemented by, or depending on, code that uses an unauthorised library will be treated as absent.** This may dramatically reduce your mark.

Although libraries of this kind are often used in commercial web development, the objective of this exercise is to assess your understanding of the core web technologies that underly all of these tools. Learning this will enable you to use any library in the future and to adapt when a library does not provide the precise function required (which is also extremely common!)

**Plagiarism**

This is individual work. You must not copy or share code with other students. Do not copy code from online sources, answers, tutorials or existing open source forum software. Copied code will result in the plagiarism process being invoked and you may be asked to attend an online meeting to verify that your understanding of your code is consistent with you having written it.

**You must be very careful with the use of online tutorials on this module.** Your primary learning source should be the module notes and the lecturer and tutorial staff. There are a very large number of online tutorials on the topic of web development, but many are incorrect, out-of-date, or badly written. Accurate, up-to-date, well-written information is not typically available for free.

**Sites which present code and then explain it are often cheating sites, not tutorials.** Retroactively explaining large amounts of code is not an effective learning method, but is often used as a trick to “justify” presenting code actually intended to be cut and pasted. Often the explanations are extremely poor or even incorrect and will confuse your understanding further.

**Do not use pre-implemented forums as a base for your coursework.** The additional features that this forum application has require a different database structure. The provided database design has the correct structure. Attempting to adapt an existing forum implementation will not only result in plagiarism penalties but will actually be harder than implementing it yourself.

**Do not use code generated by AI.** This can also result in plagiarism proceedings being invoked. No current generative AI produces fully correct output for this coursework specification, and many make distinctive errors that will allow the use of AI to be identified.

**Submission**

There are three components to submit:

* a preliminary report to be submitted by the end of week 8.
* the actual website to be submitted by the end of week 12.
* a final report to also be submitted by the end of week 12.

“The end” of a week is **1pm on Friday** of that week.

The preliminary report is worth 20% of the coursework. The implementation is worth 60%, and the final report is also worth 20%.

Note that marking on this module will **not** be anonymous as it is necessary to relate your submissions to each other.

**Preliminary Report**

The preliminary report should cover the design and structure of implementation you intend to use to produce the selected functionality. This should include:

* an overview of the intended implementations at client and server (20% of the mark).
* how tasks will be divided between the code on the client and the code on the server (40%);
* wireframes representing the user interface (40%).

The most important task in the preliminary report is to refer to “HTTP dynamic scaling” at least three times and break down the functionality between the client and server, and you must make sure to consider the functionality in sufficient detail to do this. For example, “a list of topics will be displayed” is inadequate detail, as this is a multi-step process involving communication with the server and interaction with the database.

You should also include wireframes of the user interface you aim to create. Wireframes are not prototypes: wireframes should be drawn by hand or using a simple art program, not created in HTML. Non-header areas of text can be represented by dotted lines. Please take care when drawing your wireframes to ensure they represent the intended look of your site. Avoid leaving large gaps or margins in your wireframes as they imply leaving large areas of your web page blank.

It is not necessary to include source code in the report. The report will be your main opportunity to gather feedback before the submission (although you may also ask for help in practical sessions) so make sure to provide detail on any decisions you are not sure about, so that the assessor can provide useful feedback.

The preliminary report should be submitted via Moodle by the end of Week 8. You may have already started developing your implementation before this submission.

**Web application**

Your web application should be uploaded as a .zip file to Moodle by the end of week 12. To prepare your .ZIP file, do the following steps:

1. Prepare a fresh virtual environment via the commands described in the “restrictions” section.
2. Copy your website and database files into the virtual environment, activate it, and ensure that your website works inside it.
3. Create a dump of your sqlite database using the following commands at a Python console (replacing dbfile.db with the name of your database):
4. import sqlite3
5. db = sqlite3.connect('dbfile.db')
6. with open("dump.sql", "w") as f:
7. for line in db.iterdump():
8. f.write(line + '\n')
9. Ensure that the dump.sql file is within the virtual environment directory.
10. Delete the directories Include, Lib, Scripts, and tcl. **Do not forget to delete these directories - if you leave them in, the ZIP file may be too big to submit to Moodle.**
11. ZIP the virtual environment directory.

Marking for the website implementation is divided as follows (with percentages of the website mark):

* **28%** - Functionality: whether or not the functionality described in the specification above is implemented and working. Please read the functionality specification carefully as this includes all parts of the functionality described, such as information displayed on a post, named relationships between replies and claims, and ability for claims to relate to multiple other claims.
* **28%** - Code quality. Code should be clear to read, efficiently structured and make efficient use of the database. “Antipatterns” such as repeated blocks, one-time loops, expensive read-backs, etc. should be avoided. This also includes web security issues.
* **14%** - Interaction and site design. The website’s aesthetic appearance should be considered, and interaction paths with the user should be user-friendly. For example: Relationships between replies and claims, or claims and claims, should be shown to the user in a clearly understandable way; and entered data, such as the text of a post, should not be lost by needing to log in.
* **30%** - AJAX implementation. Quality of internal messaging protocols and enhancements to the user experience delivered by the use of AJAX.

**Final Report**

The final report should be no longer than 6 sides of A4 at standard font size, and should address the following issues:

The main aspect of your final report should be to **describe how you have tested your site.** This should include a formal test plan and results as covered on previous programming and software engineering, and any fixes you made. This should include testing both the functionality and accessibility of your site. You should also test that your site works when multiple sessions interact with it at once, by using multiple browsers at once on your computer. This is worth 70% of the final report marks. Note: if you present your test plan and results in a table, lines and extra space taken by that table do not count against the limit of 6 sides of A4.

**Describe the accessibility measures added to your site** and who they are intended for. The measures must be documented in WCAG as correct ways to increase accessibility. You could also describe any additional accessibility features that might be added with more development time. This is worth 10% of the final report marks. Note that the absence of inaccessible features that are not relevant to the specification (such as video) cannot be listed here and is not worth any marks.

**Describe how you would adapt your site for mobile phone users.** You do not have to write actual changes to the CSS, HTML, or JavaScript, but you should describe the changes you would make and the means by which they would adapt the site. It is not acceptable to write that you would replace your site with an app, This is worth 10% of the final report marks.

**Choose one of the major JavaScript frameworks** such as ReactJS or Angular, and research it specifically. Describe how you would modify your site to work within this framework and what difference it would make to implementation of your site. Note that your description must be specific to **your site** - a general description of the framework will score no marks. This is worth 10% of the final report marks.

The final report should be uploaded to Moodle by the end of week 12.

**Overall Mark Scheme**

| **Component** | **% of Component** | **% of Coursework** |
| --- | --- | --- |
| **Proposal Report** |  | **20%** |
| Implementation Overview | 20% | 4% |
| Client/Server Breakdown | 40% | 8% |
| User interface wireframes | 40% | 8% |
| **Implementation** |  | **60%** |
| Functionality | 28% | 16.8% |
| Code Quality | 28% | 16.8% |
| Interaction Design | 14% | 7.4% |
| AJAX | 30% | 19% |
| **Final Report** |  | **20%** |
| Testing | 70% | 14% |
| Accessibility | 10% | 2% |
| Mobile Adaptation | 10% | 2% |
| Framework Usage | 10% | 2% |

**Learning Outcomes**

* **Apply and evaluate the principles of web application technologies, architectures and deployment strategies.** Should be demonstrated by your design choices in your implementation, and the framework evaluation section of your final report.
* **Discuss and implement in practice the principles of web design, web accessibility and web application security policies.** Should be demonstrated by your design choices in your implementation, and the mobile and accessibility sections of your final report.
* **Apply appropriate methods, programming techniques, Application programming Interfaces (APIs), and relevant server-side technologies to design, implement and deploy a web application.** Should be demonstrated by successful completion of your implementation.

**Tips**

You will need to have completed the **practical exercises** as well as the lectures before starting the coursework. Some of the technical aspects of implementation are taught only through the practicals. Remember that you can always ask for help on the practicals. Completing the practicals will leave you with a working web-based to-do list application that stores its data in a database. This can be used as the foundation for your coursework.

The lecture material and practicals do not directly walk you through the coursework, but it can be completed by applying the skills they teach.

* Think about database **transactions** - what is the list of actions that the user (possibly an admin user) can take that alter the database? How can you write each one as a query? Think back to **COMP4003** and consideration of use cases.
* What input does each transaction need, and how can you get that from the user over the web? **Sessions 1-4** will help here.
* How will you send the input to the server? First, in what form? How can you make it easy for your program to work with but also secure? Again, see **sessions 1-4**. Second, how will you actually send it over the web? See **sessions 5 and 8** for your options.
* Can you now write the Python program for the transaction and assign it to be a Flask route? See **sessions 5 and 6** as well as general Python programming from **COMP4004**.
* How will you return the response to the user? Will you use a web template (**session 6**) or maybe a JavaScript program that reads from the server using AJAX (**session 9**)?
* Is the transaction “stateful” - meaning that it changes the treatment of other transactions after it, such as logging in? If so, what do you need to store in the Flask session variable (**session 6**) to allow this?
* You will be expected to be aware of security throughout your implementation - remember your work on **COMP5020**. There is no separate mark for security because security is so important in web programming that no insecure implementation can be good!

**Previous Common Errors**

Here are some previous mistakes frequently made by students on Web Development courseworks. Please be careful not to repeat them!

**Preliminary report**

* Repeating what is stated in the specification as if it was your decision rather than what you have been told to do.
* Telling the story of how a function will be implemented, or stating that you intend to implement it, but not how the implementation will work.
* Failing to break down tasks into client and server components or attributing all of a multi-stage task to one of these; for example “the client logs into the site” which actually requires multiple steps and cannot be purely done on the client.
* Describing what is **stored** on the client or server, but not what functionality they have.
* Over-emphasising the choice of technologies, rather than how they are used.
* Confusing the “user” and the “client”. While in business terms the “client” of a system can mean the user/customer, it does not mean this in web development; it means the web browser and the computer running it.
* Using informal diagrams with no defined meaning. If you decide to use diagrams, make sure to use standard diagrams - you learned UML diagrams on **COMP4003**. If you decide to make up your own diagram, make sure that each symbol has only one meaning - this is especially important for arrows.
* Stating that validation is done on the client - a major security error.
* Stating that the “user will be logged in” without indicating what method is used to signal this.
* Large amount of dead space on UI designs.
* UI designs that don’t show how replies or claims are related to each other.
* Showing UI prototypes instead of wireframes.
* Using colour coding alone to describe the status of UI items. This is inaccessible to colour blind users.
* Where AJAX is used, not providing a structure for the JSON messages that will be used.

**Web site**

* Using unauthorized libraries.
* Failing to implement all basic functionality.
* Failing to allow claims to be related to more than one other claim.
* Storing user logged in status in the form, rather than the session. This is a major security error.
* Functions on the server that require the user to be logged in don’t actually check that the user is logged in; they just assume that the user could not have reached the link if not logged in. This is also a major security error since links can be guessed and/or copied.
* Not making Topic and Claim pages directly linkable, which is part of the specification.
* Using colour coding alone for reply types, which is inaccessible to colour blind users.
* Making the on-screen fields for reply/claim text too short for messages to be entered.
* AJAX URLs don’t check access rights or that the user is logged in.

**Final Report**

* Describing colour coding as an accessible feature.
* Listing the general principles of mobile friendliness, framework features, etc. but not how they would be used on this particular site.
* Testing complete functionality paths instead of individual interactions. While testing paths is a good idea, remember that users may not follow them; web requests can be sent in any order as a result of shared links, the back button, etc.
* Not testing edge, exceptional or error cases.