Web Platform Development Coursework 2 Report

M3I322955-19-B

Conor McGowan – S1709217

Samuel Brown - S1713974

Zygimantas Domarkas - S1718169

**Link Design**

Our original intention for the link design was to query the username and somehow insert this username into the link to uniquely identify each page. Moving forward we removed the need for a username as we feel it didn’t quite make sense given how the application functioned and with the scope of the application being restricted.

We ended up having most of the main functionality tied in with the use of link design. The application utilises buttons to trigger the functions for things like adding a new coursework project. For example, upon clicking the “Add a new project” button which has an href element equal to “/add”, the page will then be redirected to “localhost:9000/add”. The controller will contain multiple GET request handlers and a POST request, with most of the data being sent to the client rather than reading input from the client. The GET request handler for ‘/add’ will then return with a response containing the mustache template to render, in this case, “new-project”. The user will be presented with the template containing a form for a new project, once submitted, the addProject method will be used to create a new project entry and add it to the database. The client will then be redirected to “/projects”, with “/” also redirecting to this if the client ends up there. On this page, a method is used to fetch a list of all the projects in the database to be displayed in a dynamic list using mustache templates. Other functionality also works in a similar way, with the links for the edit and delete functions having hrefs of “edit/{{project}}” and “delete/{{project}}” respectively.

The plan was to implement a feature to allow the user to copy a projects shareable link and submit it into a form, the link would then be retrieved from the URL and used to render the specific project for that link. This would also work if they replaced the link in the URL with a link relating to another project, rendering that one instead. Sadly we were not able to get this fully functioning due to a problem with passing the variables but the methods are all in place for this.

**Description of Persistence**

As specified in the coursework spec the application has been developed with Nedb, Node.js and Node Express. We have persisted our data by pushing it onto a Nedb database, using a function to update the database or add to it when the user makes an amendment.

The database runs from a specified file rather than an in-memory database, allowing data to persist even when the applications is closed down. The database stores the projects in a very simple structure, each project will have a project name, a module name, a target date for the student to complete the project by and a date for the actual deadline. Each project also stores however many tasks the user has defined, using an array of strings within each entry, with an auto generated identifier field. The user must define at least one task and all other fields are required.

Below are two example lines from the database, showing the format of how a project is stored:

{"project":"IP3 Group Report","module":"Integrated Project 3","intendedDate":"24/04/20","actualDueDate":"29/04/20","tasks":[{"task":"Design Stage"},{"task":"Initial Implementation"},{"task":"Integration of Back-End with Front-End"}],"\_id":"zhjspc1TwaxyyYDx"}

{"project":"RSPI Presentation","module":"RSPI","intendedDate":"14/05/20","actualDueDate":"22/05/20","tasks":[{"task":"Create Powerpoint"},{"task":"Record voiceover"}],"\_id":"Cc3nH5KhzuYbKU2E"}

To add something to the database the user clicks a button, the function for the project data access object is called to render the new project moustache template, which then renders the order form, they are then prompted with a form which they fill in and submit, the details from this form are then used to create an entry that is added to the database by a function.

Removal of a task is done by a similar process to the one above.

The schema for the database is below.

A screenshot of a cell phone

Description automatically generated

A decision was made to only use one table so as to keep the database simple, efficient and robust. It could have been done in two tables but for the sake of efficiency and ease of use we kept it simple, we believe this design choice benefited us greatly in the development of the deliverable. The tasks were stored as a string early in development but once the projects were being rendered correctly and everything was working, we then changed the tasks to be stored as an array of strings instead. This made add and editing new projects very easy and got rid of the possibility for having a second table to store the tasks relating to a specific project.

**Documentation of test reports**

|  |  |  |
| --- | --- | --- |
| **Test Subject** | **Intended Outcome** | **Actual Outcome** |
| Courseworks can be added | User adds a Module, Coursework name, Intended due date, actual due date and submit | Fully working and implemented. User can add courseworks with due dates and details. |
| Courseworks can be removed | Users can delete courseworks by pressing the delete button (Rubbish bin button, top right above coursework) | User presses button, coursework is deleted. |
| Courseworks can be modified | Press edit button (pencil button top right) and any detail can be edited | User presses button, they are taken to a edit form where they can change any detail they like |
| A listing of all incomplete courseworks and their milestones | User logs on and is presented with a home page where this is visible | The users default view fulfils this request. |
| Users can share a project with other users by sending a link | User opens a share box, inputs a course identifier and a link is generated | Not working fully, button works and a field for the link generation exists but the variable does not pass and the user never reaches the shared project view. A page for this view exists. Each project has a unique link, we failed to get this fully implemented but did all we could. |
| Users can view the title of coursework, the module to which it belongs, milestones and two due dates, all customisable. | The user is presented with an easy to use userface which makes all of the criteria apparent | The user can easily see the courseworks, can also edit them at the push of a button. |

**Appraisal of security**

The five biggest threats to a web application are

* Injection
* Broken authentication and session management
* Sensitive data exposure
* Xml external entity
* Broken access control

There is no opportunity in this coursework to inject directly anything malicious into a database. The fields do not needed to be validated because the database is stored locally and if a malicious user intended to drop the tables, for example, they’d only succeed in ruining their own experience. An injection attack occurs when database commands are executed by the database engine, usually inputting commands into a form (A user setting their name as drop\_tables in SQL or the equivalent in other syntax) will cause all tables to be dropped from the database, a side effect of this is also a lot of embarrassment for the designer of the database and form.

The use of parameterised queries could completely eliminate the injection risk if the testing and parameterisation was in depth enough. Keeping your database up to date will also help prevent this occurring.

There is no considerations for sessions in this coursework, each user is verified as themselves only by the username they have and there is no sensitive data moving to a server or being displayed to trusted individuals so there is no real need for any measures here. A session hijack is a type of attack that could exploit a valid authentication key to gain unauthorised access to a form which only a valid user can see (A users profile, bank account, admin panel etc.) and as a result, the facilitation of malicious actions. This is only one sort of session exploit, there are multiple others

To prevent this risk, we could implement a number of measures, one such measure is to encourage users to have antivirus software and keep their browsers and related software updated, however, relying on users to employ good practice is never a good practice. There is another method implemented on the server side in which the requests of a session are fingerprinted and in addition the IP and SSL session ID is logged, there is a number of points allocated to each piece of valid data and when a request falls below that threshold, the session terminates as invalid (Different SSL and IP than the original authenticated user)

Sensitive data exposure occurs when plaintext information is exposed to a third party. Under the Data Protection act measures must be put in place to mitigate the risk of this happening and the data controller is responsible for not revealing any of the data subject’s information.

Encryption could be employed here, hashing and salting would remove the risk of plaintext data being put out as a result of a blunder of attack. There are numerous good practices one could employ to ensure that even if the attacker got the encrypted text he would have to unhash it and unsalt it which would prove to be incredibly hard. Advances in computing are on the verge of rendering it impossible to unencrypt encrypted data through the use of quantum computing. However in the past the new development of “unbreakable” encryption has been shortly broken.

An XML External Entity attack is a type of attack that allows an attacker to interfere with an application processing of xml data, it often allows an attacker to view files or interact with backends, this is an incredibly large vulnerability as it gives near unrestricted access to the backend.

There are no ways to completely eliminate the threat of an XXE attack, but again, a number of good practices can be adopted. The use of less complex data formats and increased verification would help greatly.

Access control errors allow the bypassing of checks by modifying the url or page or a custom tool etc. This is another large vulnerability in the same vein as an XEE attack.

The only way to avoid such XEE attacks is denying access to functionality as a default and requiring users to validate properly for access to features. Enforcing the use of server side code or a serverless API will also mitigate the risk of this occurring by a large margin.

**Declaration**

I declare that all work submitted for this coursework is the work of Conor McGowan, Samuel Brown, Zygimantas Domarkas alone unless stated otherwise