# Coursework Team Report

# Team A (Alexander McKenzie (S1507940), Oheneba Poku-Marboah (S2008061), James Devenport (S1423789))

## **Heroku Link:** [**https://cwk-checkr.herokuapp.com/**](https://cwk-checkr.herokuapp.com/)

## Link Design

**Listing of Links**

1. Domain/
2. Domain/login
3. Domain/logout
4. Domain/register
5. Domain/reg-success
6. Domain/view-coursework
7. Domain/view-coursework?selected=”courseworkID”
8. Domain/view-coursework?shareStudent=”studentID”&cwkId=”courseworkID”
9. Domain/add-coursework
10. Domain/add-coursework-success
11. Domain/edit-coursework
12. Domain/edit-coursework-success
13. Domain/remove-coursework
14. Domain/remove-coursework-success

Each piece of the application is purely coursework related. Hypothetically in the future, the URL format could be changed to something along the lines of “domain/course/add”, “domain/coursework/add”. However, considering that for this application only courseworks were relevant, this extra part to the URL seemed superfluous.

The URL naming is self-explanatory which means a user will easily be able to tell which page they are viewing simply by glancing at what the URL is called. The URLs are also fairly short and easy to read.

Each URL stems from the domain host (in this case, https://cwk-checkr.herokuapp.com/). The user can navigate to each URL with ease, either through an embedded hyperlink within the navigation bar or elsewhere on the page, or by manually entering the URL into the browser’s URL bar.

Using queries, or added URL sections, future pages can be added easily. For example, in links 7 and 8 the view-coursework link remains the same, but additional queries (selected, shareStudent, cwkId) cause the page to render differently. In number 7, a specific coursework is loaded instead of every coursework belonging to the logged in user. In number 8, the coursework corresponding to the entered student ID and coursework ID will be loaded. No coursework is loaded if either of these fields are incorrect.

In theory, this could be done by adding URL sections as well. E.g. if in edit-coursework a user wanted to edit only one coursework, the coursework name could be added in the URL - “/domain/edit-coursework/first-coursework”. If the user wanted to edit all coursework, this could be done in a similar way - “domain/edit-coursework/all”.

Session IDs have been deliberately omitted from the URL, only alphanumeric characters are present, and no input data is ever present within the URLs (these are handled through POST requests). An API for handling post requests has been designed by Ohe. These URLs should never be accessible to the user, but if they somehow are, they won’t do anything since they only respond to post requests.

Each URL – except for links 2, 3 and 4 – requires for the user to be logged in. This is validated with a session token. If the user session token is not present, the user is redirected to the login page. Once logged in, the user will have their request forwarded to whichever page they were trying to access previously. E.g. user tries to access “/domain/add-coursework” and gets redirected to “/login”. User logs in successfully and is now redirected to “domain/add-coursework”.

Test Reports

**Functional Testing was carried out for the following modules:**

1. Edit coursework
2. View coursework
3. Remove coursework
4. Add Coursework
5. Register
6. Login
7. View complete/incomplete coursework/milestones
8. Generate Shareable Link
9. View Shareable Link

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Action** | **Expected outcome** | **Status** | **Evidence** |
| 1.1 | Click dropdown that lists all available thus editable courseworks | Dropdown menu shows courseworks | OK |  |
| 1.2 | Edit coursework by adding milestone “for test reports” and click submit | Redirects to home page and adds milestone to coursework | ok |  |
| 1.3 | Click add milestone on edit coursework page opens input bar to write and option to not make any change | Opens input bar for milestone and button to remove the milestone | Ok |  |
| 2.1 | http://localhost:3001/view-coursework | Shows complete and incomplete courseworks | Ok |  |
| 3.1 | http://localhost:3001/remove-coursework | Shows dropdown menu to select which coursework you want to remove | ok |  |
| 3.2 | Go to <http://localhost:3001/remove-coursework> , select “CWK 3” and click Remove | Popup alert asks for confirmation | Ok |  |
| 3.3 | Click Remove Coursework with “CWK 3” selected in dropdown | Coursework “CWK 3”no longer present in list of courseworks | Ok |  |
| 4.1 | Go to <http://localhost:3001/add-coursework>, select “Course Two” in the course dropdown and “CWK Four” in the coursework dropdown, then click submit. | User is redirected to the application home page, “CWK Four” shows in the users coursework listings. | Ok |  |
| 4.2 | Go to <http://localhost:3001/add-coursework>, select “Course Two” in the course dropdown and “CWK Four” in the coursework dropdown, then click submit. | User is shown a message box highlighting that they cannot add an already added coursework to their courseworks collection. | Ok |  |
| 4.3 | Go to <http://localhost:3001/add-coursework>, select “Course Two” in the course dropdown and “CWK Two” in the coursework dropdown. Add milestone started and set to completed, add milestone finished and set to incomplete then click submit. | User is redirected to the home page, “CWK Two” shows in the users coursework listings. When on the view coursework page, “CWK Two” has two milestones showing: “started”, which is set as complete, and “finished” which is set as incomplete. | Ok | Inserting image...Inserting image... |
| 5.1 | Go to <http://localhost:3001/register>, enter nothing within the user registration fields and click submit | Unfilled fields are highlighted red, a new user is not registered. | Ok |  |
| 5.2 | Go to <http://localhost:3001/register,> enter full name as test ted, username as test user, and password as test pass | User “test user” is registered, user is shown a success screen and is then redirected to the login screen. | Ok |  |
| 5.3 | Go to <http://localhost:3001/register,> enter full name as test ted, username as test user, and password as test pass | A message box is shown indicating the requested username is already taken | Ok |  |
| 6.1 | Go to <http://localhost:3001/login>, enter test as username (incorrect username) and test pass as password | A message box is shown indicating that the user has entered an incorrect username/password combo | Ok |  |
| 6.2 | Go to <http://localhost:3001/login>, enter test user as username and test as password (incorrect password) | A message box is shown indicating that the user has entered an incorrect username/password combo | Ok |  |
| 6.3 | Go to <http://localhost:3001/login>, enter test user as username and test pass as password | User “test user” is logged in and redirected to the application home page | Ok |  |
| 7.1 | Set a new coursework with 3 milestones. 2 milestones are complete and 1 is incomplete | Coursework is added successfully | Ok |  |
| 7.2 | Validate the milestones are present in ‘view-coursework’ and labelled as complete/incomplete correctly | Milestones are visible. Complete milestones marked with a tick, incomplete milestones marked with a blank box | Ok |  |
| 7.3 | Change third milestone to complete in ‘edit-coursework’, then make sure this is reflected in ‘view-coursework’ | Milestone updated successfully. Reflected in view coursework screen as expected | Ok |  |
| 7.4 | Change coursework from incomplete to complete by adding a completion date as today’s date (29/05/2020). Validate this in the view coursework screen. | Completion date added and reflected in view coursework screen. Coursework moved from incomplete to complete section. Completion date is also visible. | Ok |  |
| 8.1 | Go into a specific coursework view (CWK Five) and validate that shareable link is present | Shareable link is visible along with coursework information | Ok |  |
| 9.1 | Check the shareable link under the same user (JDeven200) | Coursework was displayed correctly when the link <http://localhost:3001/view-coursework?shareStudent=4&cwkId=5> was navigated to | Ok |  |
| 9.2 | Log in as different user and use the shareable link to view the specified coursework (CWK Five) | Logged in as “Mc ALEX” and copied the shareable link <http://localhost:3001/view-coursework?shareStudent=4&cwkId=5>The coursework and its information was displayed correctly. | Ok |  |

## Persistence

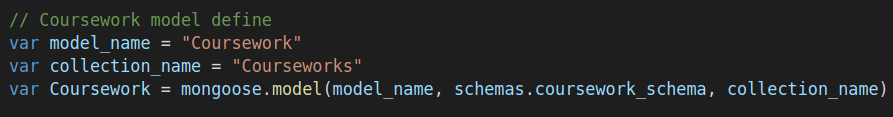
Datastore

MongoDB Atlas, a cloud-based MongoDB service, was used to persist data within the application. Mongoose, a MongoDB data modelling framework, was used to map document attribute keys to their values. This was done by first defining a collection of Mongoose models, which were then referenced for data storage and manipulation.

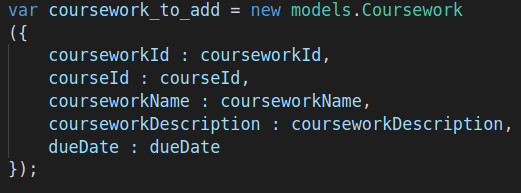
Defined data models referenced collections stored within MongoDB atlas, allowing direct interaction with these collections through models to ensure data was manipulated in a consistent manner (E.g. the same document attributes appeared within all documents in a single collection).

This is exemplified when analyzing the coursework model code below. Initially, a coursework model is defined, it references a Mongoose schema, used to lay out the attributes expected within a Mongoose model. When creating a new document based on the coursework model, the fixed attributes the document should contain (defined by the model's schema) are enforced. Then, when saving the created coursework model document, it is shown that the coursework model’s collection can easily be referenced within the model object for its manipulation.

**Defining the coursework model**



**Defining a coursework model document**

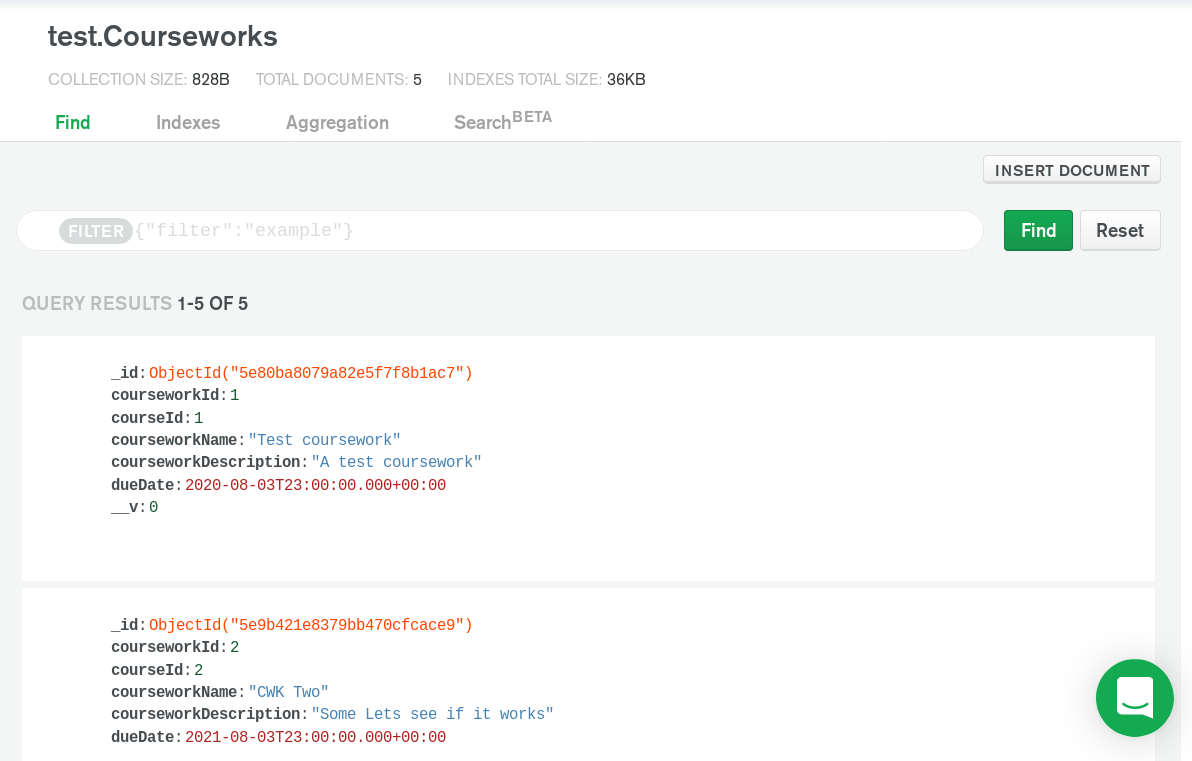


**Saving the defined coursework model document in the MongoDB Atlas courseworks collection**



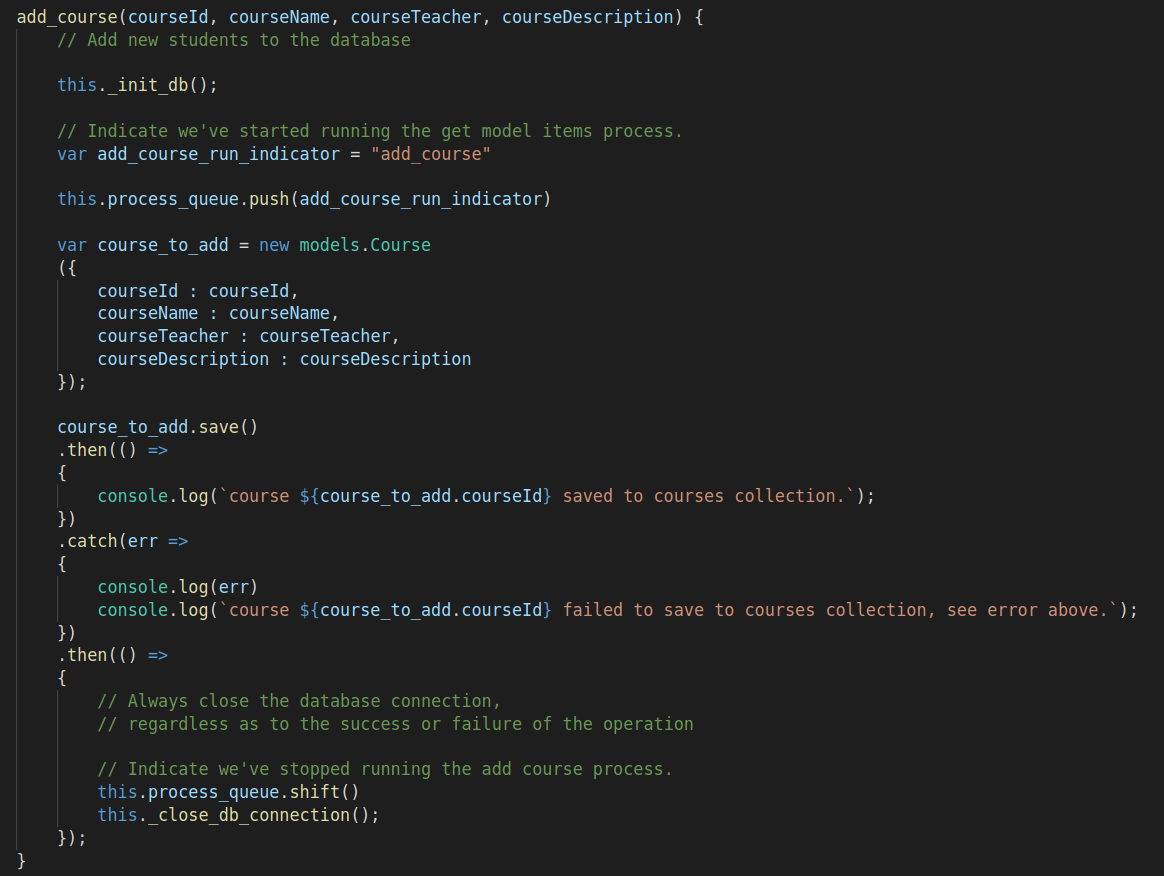
As mentioned above, Mongoose data models directly reference collections within MongoDB Atlas. This is exemplified below, where it is shown that the MongoDB atlas Courseworks collection is directly being interacted with through it being referenced in the Coursework model – shown through the populated collection.

**The populated MongoDB Atlas courseworks collection**



### Database functions

The DAO class was used to implement all database interaction functionality used to manipulate data stored within the database. For global collections, which were referenced across all user accounts, and stored student, coursework and course document collections, only add functionality was implemented. This was because these global collection documents did not need to be edited or deleted as part of the applications core functionality. Therefore, the deletion and editing of these global documents could be managed in the MongoDB Atlas database. A code snippet of the add\_course method, used to add global courses to the courses collection, is shown to highlight this.

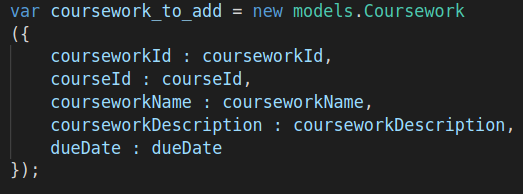


In order to allow users to manipulate global collection documents in a manner that was stereotyped to them, e.g. adding milestones to a coursework, code was added to

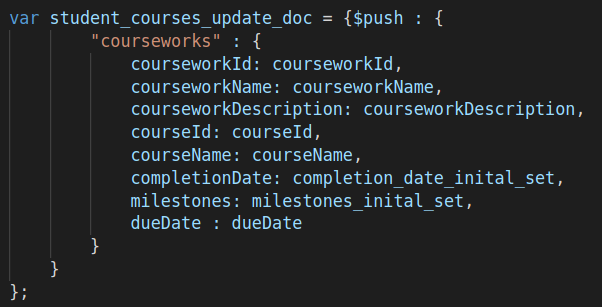
* control the association of globally stored documents to individual students
* Undertake localized definition of document attributes relevant only to individual students (E.g. milestones for a coursework)
* Manipulate these global documents within individual student documents (e.g. removing a coursework from a user courseworks list).

For example, when adding a coursework to a student, the coursework is appended to the students courseworks collection and, whilst doing so, additional document attributes localized to the individual student are added to the document, allowing the student to manipulate the coursework document locally as required.

**Original coursework document saved in the global courseworks collection**



**Localized coursework document added to a student's local courseworks collection, where additional attributes are added to the document**



The utilization of global collections of documents was useful, as it would allow for their extendibility to further application functionality if required. For example, adding the functionality to output a course announcement to all students. A course in the global courses collection could have an announcement associated with it, whilst its courseID attribute could then be used to identify students that have a course in their localized courses collection with the same courseID, the announcement made could then be pushed to these students.

Schema

We have a designated service for accessing the database. In the endpoint server, whenever access is needed to the database, a data access object, DAO, is imported into the module that needs it. The data access object has methods for all actions that the backend needs to be able to undertake. All functionality that our application requires the database to be able to do is written beforehand into the DAO class. These functions include the ability to add a student, add a course and coursework, add course and coursework to student, edit coursework in student, add coursework to student and to get items from the database.

|  |
| --- |
| var student\_schema = new mongoose.Schema({  studentNo: Number,  name: String,  username: String,  passwordHash: String,  courseworks: [{  courseworkId: Number,  courseworkName: String,  courseworkDescription: String,  courseId: Number,  courseName: String,  dueDate: Date,  completionDate: Date,  milestones : [  {  milestoneTitle : String,  complete : Boolean  }  ]  }],  courses: [{  courseId: Number, courseName: String  }]  }); |

|  |
| --- |
| var course\_schema = new mongoose.Schema({  courseId: Number,  courseName: String,  courseTeacher: String,  courseDescription: String  }); |

|  |
| --- |
| var coursework\_schema = new mongoose.Schema({  courseworkId: Number,  courseId: Number,  courseworkName: String,  courseworkDescription: String,  dueDate: Date  }); |

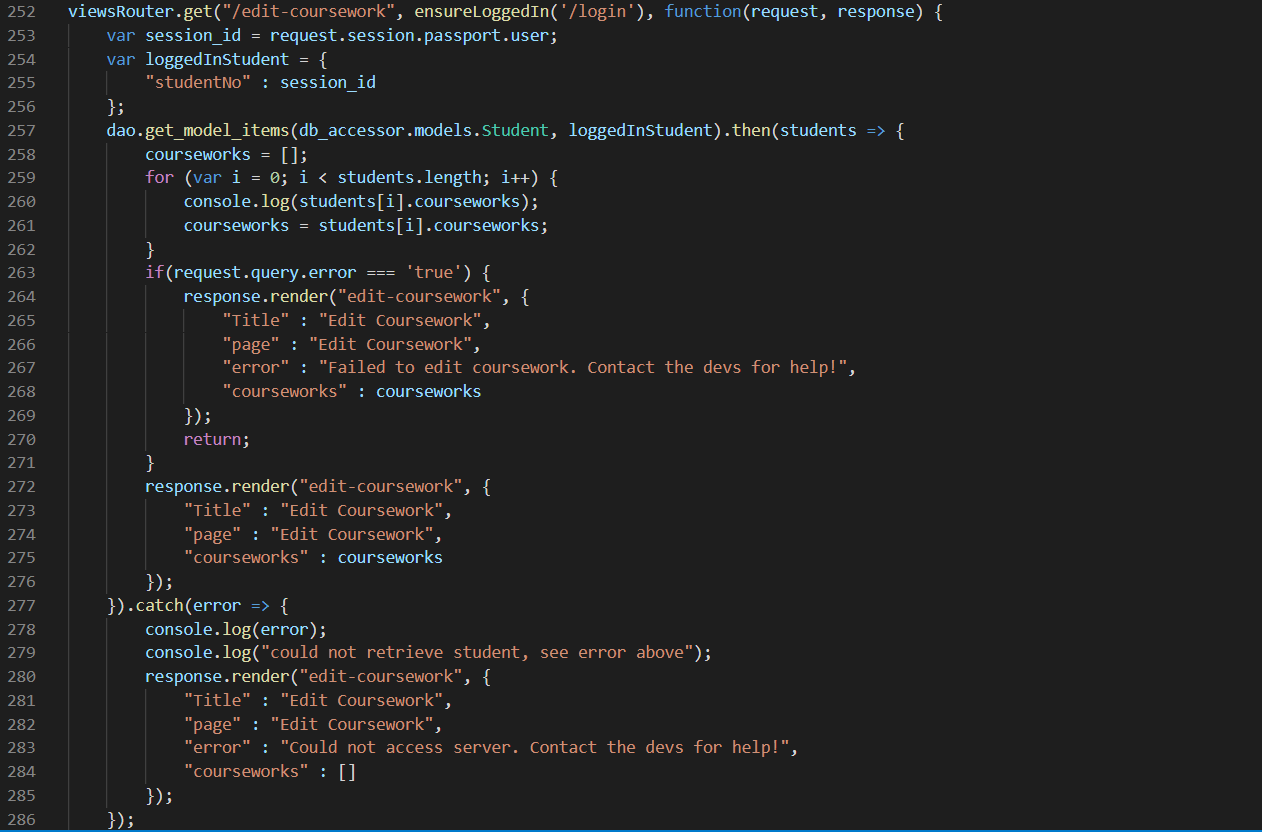
This is the document structure of our database. The course and coursework schema are global collections which contain all courses and all courseworks which are stored within the application. The courses/courseworks a student has are contained within the courses and courseworks arrays found within the Student schema.

This architecture meant that there were less duplicate items contained within the database. It also prevented the database from becoming cluttered and difficult to navigate which is useful when considering future scalability. Finally, this architecture allowed for local course/coursework names, descriptions, dates, etc. to be changed in one student’s collection while not affecting the global entry, or any other students who were working on any specific course/coursework.

Modularity

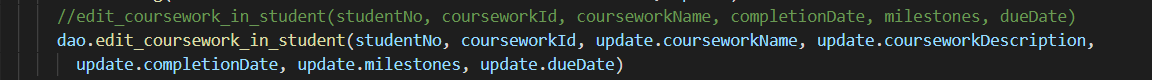
The DAO class is exported to the rest of the application. When a method is needed, the DAO is declared within the controller that requires it. The appropriate method is then called.

For retrievals, a Promise is returned with a .then() call for a successful resolution and a .catch() for a refusal. Appropriate functionality is coded in each of these calls. The functionality within these calls usually instructs the app where to redirect the response depending on whether the DAO was successful or unsuccessful in retrieving a collection.



Here we can see that upon a successful call without any ‘error’ query, the edit coursework screen will be rendered with no error message. If an ‘error’ query is present, this means that the Student collection was retrieved correctly, but something went wrong when editing the coursework. The edit coursework page is then rendered with the error message “Failed to edit coursework. Contact the devs for help!” if this is the case. When a Student collection cannot be retrieved, the edit coursework screen is rendered with an empty “courseworks” array, and an error message which states “Could not access server. Contact the devs for help!”.

For storage functions, no Promise object is returned. The parameters sent to the storage function are used to update the specified entry within the collection. This can be seen in the edit-coursework POST request handler.



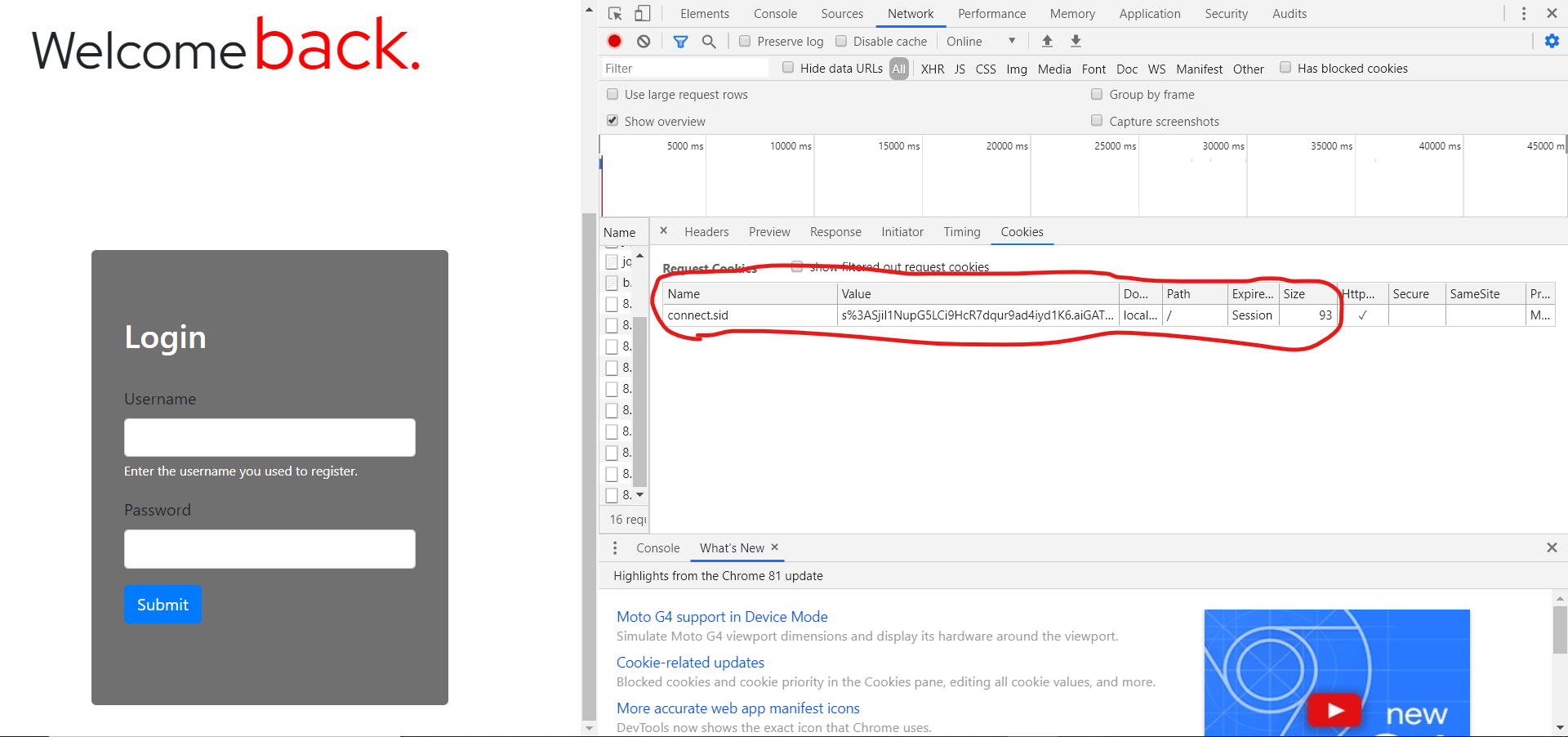
Here, the coursework parameters have been collated in an object called ‘update’ with this object’s fields being the updated coursework values. CourseworkId and dueDate will never change. Everything else can be modified by the user. We can see that the ‘edit\_coursework\_in\_student’ method within the DAO has been called and the updated coursework ID, name, description, completion date, milestones and due date are all used as function parameters.

Through this use of modularity when implementing database persistence, it means that team members who were not directly involved with the creation of the database methods are still able to use these methods without knowing exactly how they work. This helped avoid confusion and saved development time. This modularity also improved application cohesiveness. There were no parts of the application which were dependent on the DAO being there; it was pluggable and could be used as needed.

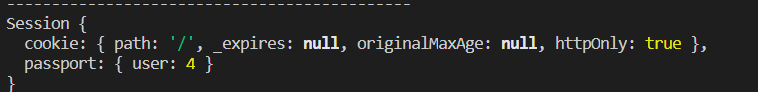
## An appraisal of application security

### Session Management and User Request Authentication

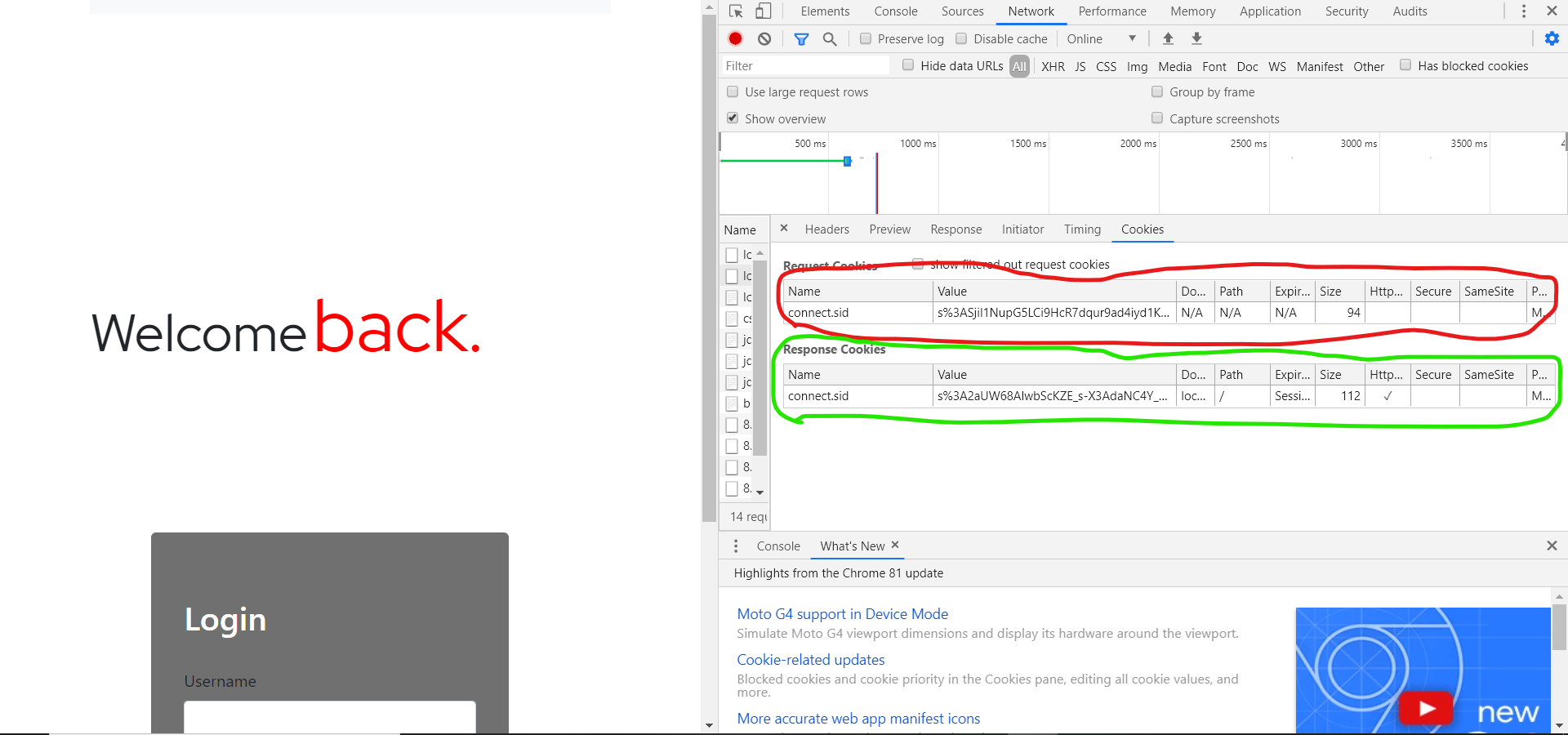
Whenever a user opens the application in their browser, a new session is created within express. This session has a ‘user’ token applied to it upon a successful login. The session itself will expire after a given amount of time. This is how the user is validated on each subsequent request after logging in. If the ‘user’ token is not present on this session, then the user will be redirected to the login screen since they are no longer validated within the application. Upon logging out, the ‘user’ token is removed, and the session is destroyed. This prevents an old session from being used to gain unauthorised access to the application. The user token being created can be seen below, as well as the user token and session being destroyed upon logging out. Another session is created once the previous session is destroyed on the following request.



**Session being created. The value can be seen to be “s%3ASjil1NupG5LCi9HcR7dqur9ad4iyd1K6.aiGATi7QroEuiYoOvXWSs7obmvRPy%2F9mEsZSGjJDjyk”.**



**User token being added upon a successful login. The user token is the user’s student ID in the application.**



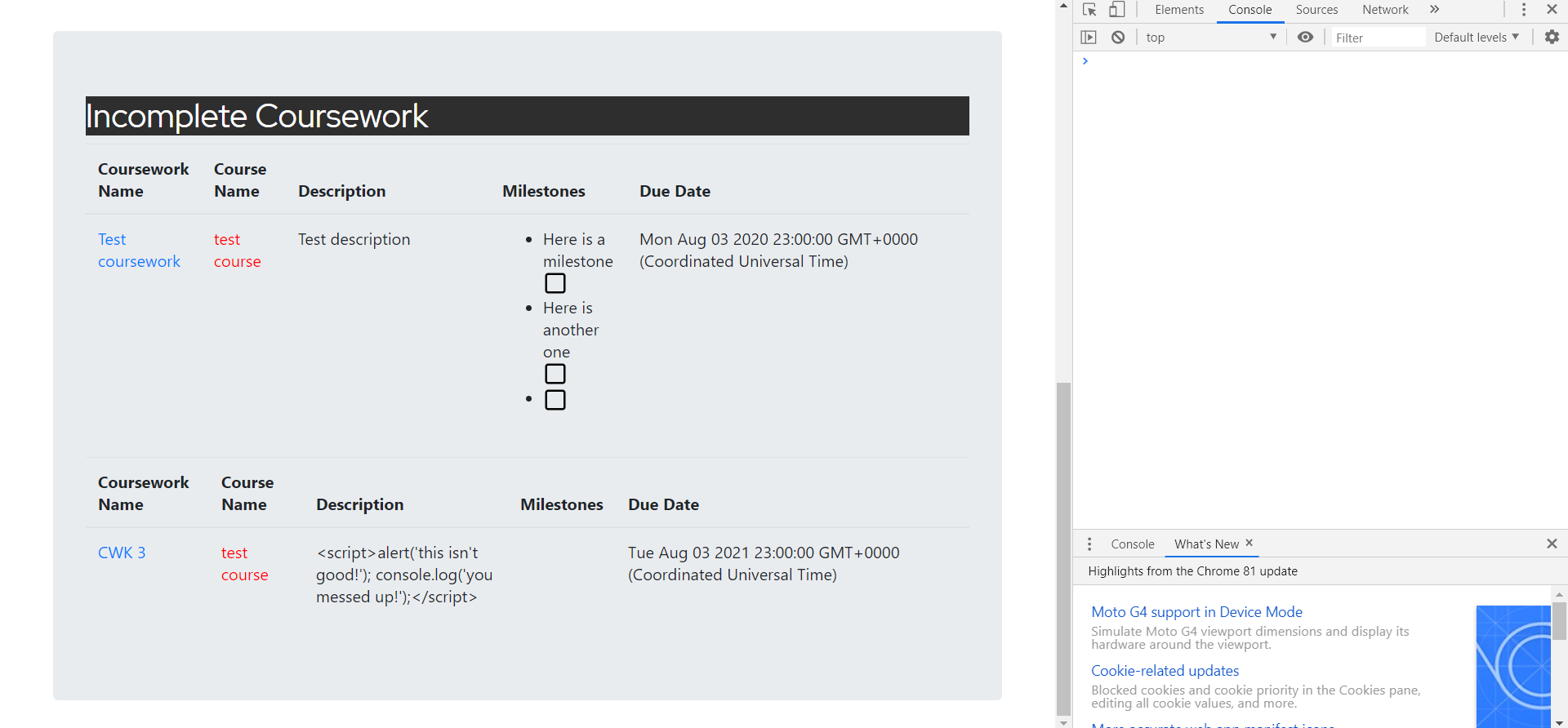
**Once logged out, the destroyed session can be seen within the request (red) followed by the new session created by the response (green). This new session value is “s%3A2aUW68AlwbScKZE\_s-X3AdaNC4Y\_CrHj.ZX7aA%2FFmFRDOComSPENNXkCxIQPmzcD7XNtZU3DUnyo”.**

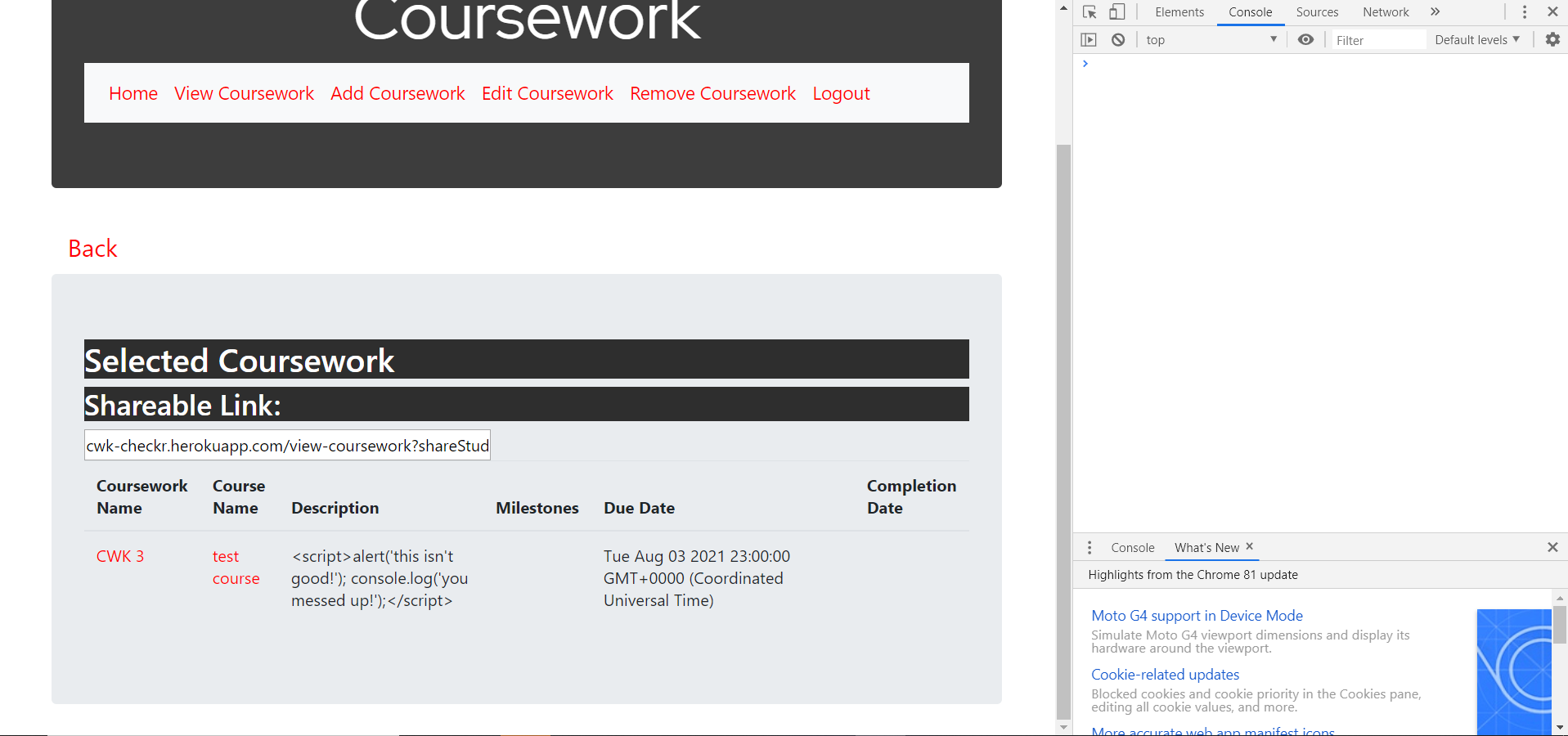
### Cross-Site Scripting (XSS) Prevention

XSS attacks are a serious web application vulnerability. Within express, carats (< >) are encoded in the URL to prevent script tags from being used. This prevents reflected XSS attacks from occurring. However, stored XSS attacks are still possible. This is when a script is embedded within an input box – a comment, or a description for instance – and stored in the database. Then, when the information is retrieved again, the script payload executes on the client side.

To prevent this from occurring, no raw data retrieved from the database is presented on the front end. Everything is passed as a parameter and inserted into a mustache tag. This ensures that any scripts embedded within a document are only ever treated as raw text and nothing else, which prevents stored XSS attacks from working.

This can be seen below in an example. Notice the ‘Description’ box containing a script which triggers an alert box and logs data to the console, but no alert box and no console logging takes place since the text in the description box is being treated as pure html because of the parameterisation + mustache tags.





### Database security

**MongoDB atlas integrated security**

The utilisation of MongoDB Atlas allowed the application to benefit from its integrated security features for the storage of database data. These features included:

**Isolated networking**

MongoDB atlas clusters, collections of servers used to host the Atlas service and store its data, are deployed in secure cloud environments with dedicated firewalls. Access to Atlas’ clusters, and therefore the data held within them, must first be explicitly granted, often through IP whitelisting. This helped protect against potential outside attempts to access the application database, as whitelisting user computers denies access to any unidentified machines automatically (assuming IP spoofing etc. Is not used).

**End-to-end encryption**

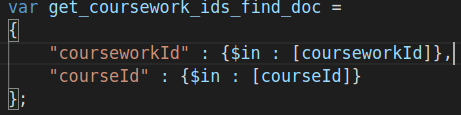
When transferring data to MongoDB Atlas databases, and retrieving this data, data is encrypted during transport using Transport Layer Security (TLS). This ensured data security, as any data intercepted by malicious entities through communications interception (e.g. man-in-the-middle attack) would’ve be in an encrypted state, hiding the ‘real’ content of stolen data.

Further database security was implemented using the following methods:

**Mongo document storage**

Due to the use of Mongo’s No-SQL, document storage-based design, widely used SQL injection attacks where malicious users enter SQL queries (e.g. within application input fields) to be executed within the applications backend database were completely mitigated.

Within database code, Mongo queries were consistently used in a parameterised manner. Pre-defined query documents were used that accepted parameters rather than explicitly building query document strings from user input at runtime. This ensured user inputs were always treated as strings, rather than as Binary JSON (BSON) objects, which is the object type used to represent mongo queries. This prevented users injecting malicious Mongo queries into the database for execution. An example parameterised mongo query used within the DAO’s get\_model\_items method is shown below.



### Secure password storage

The application requires username and password authentication in order to be used. This requires saving both values into the database. There are a lot of risks involved in saving passwords as plain text into the database. Any breach of data leaves all your users involved in that breach exposed to have their accounts accessible by malevolent parties. Because users tend to practice the bad habit of using the same password for multiple accounts, a data breach involving plain text passwords could ruin the lives of many by making personal and financial accounts vulnerable. Due to the numerous reasons it was decided that passwords should not be saved as plain text, but rather as password hashes, within the database.

The user’s passwords were hashed with bcrypt. Bcrypt takes any passed in and turns it into a string that is always the same length. When done with a salt it creates a second layer of security which requires the salt in order to decrypt. The salt prevents automated hacking attempts (e.g. rainbow tables) from working; if a hacker wants in, they will have to attempt a brute force attack which can take weeks, sometimes months. Whenever the user logs in we use bcrypt compare to check if the passed in password is the same as that of the hash saved in the database.

**Word Count: 3360**