**Technical Documentation**

Models

TimeRecord - a record which belongs to a user, a project and date which users create to record how they are spending their time.

User - a user of the system. There are different authorisation levels for users within the system to restrict and grant access to different user types. This allows each user to have the minimum amount of access required for them to perform their job role. Each user is assigned to a department so they gain access to create time records for projects within that department.

Project - a project represents a project that is being worked on. The project belongs to a client. Users assign time records projects. A project belongs to a client.

Project Stage - a project stage belongs to a project. Project stages allow for more detailed tracking of time, because users can assign time to a specific stage within each project rather than just to the project as a whole.

Department - the department of the organisation that the client belongs to. Assigning clients to departments creates better organisation, and also creates better authentication access. Users of the system can be assigned to a department, and therefore access all clients, and the clients’ projects within the department.

Client - the client represents a real-world client of the organisation. The client belongs to a department within the system / organisation, and the client has projects within it.

WhiteListIpAddressessMiddleware – all requests to the API are passed through this middleware. It controls access to the API, based on the request’s origin IP. All IPs are blacklisted by default, and only the organisation’s public static IP has been whitelisted. This middleware prevents any requests being made to the API that hasn’t originated from the organisation’s intranet.

User authentication

When the user logs in with valid credentials, an access token for the api is generated for that user account. The token is attached to a http-only strict cookie, and returned in the response to the frontend so the cookie is stored in the browser. The user model and access token returned to the frontend are stored in the localStorage with the variables )*user* and *\_token.* Any axios request on the frontend must be sent with the *withCredentials* attribute set to true. Any axios request from the frontend will be sent with the token’s cookie, which is read by the *BearerTransfer* middleware server-side to transfer the token from the cookie to the request’s *Authorization* header. Then the *auth:api* middleware reads the token from the *Authorization* header to authenticate the user that the request has come from, and blocks any unauthenticated requests from the route. The *admin* middleware can be used in conjunction with the *auth:api* middleware to check that the authenticated user in the request has admin privileges. When the user logs out, the previously generated access token is revoked and the frontend cookie is deleted

There are 2 user authentication levels. All standard users are level 1, which gives the user permission to act as a normal employee - they can manage their own time records for dates and projects. Users with level 2 are manager level users, who can create their own time records, along with managing projects, departments, clients and users.

Frontend Middleware

All middleware applied to frontend routes is applied with each routes’ *meta* property. The middleware is then handled within the router, utilising the *beforeEach* method to check the middleware is met before allowing the request to continue to server the route’s component.

requiresAuth - this middleware checks that the user is logged in, by reading the access token stored in the localStorage of the frontend. If this is present, the frontend request is allowed, else it is not allowed and the user is redirected to the login page.

requiresAuth - this middleware checks that the logged in user has a user level of 2. If the user has a user level of 2, the frontend request is allowed, else the user is redirected to the home page.

Frontend Routes

|  |  |  |  |
| --- | --- | --- | --- |
| **Route** | **Purpose** | **Props** | **Middleware** |
| / | Displays the login page | N/A | N/A |
| /login | Displays the login page | N/A | N/A |
| /home | Displays the home page (dashboard) | N/A | requiresAuth |
| /projects | Lists all projects | N/A | requiresAuth  requiresAuthLevel2 |
| /project/{id}/view | Show more in depth details of a project | *id:* the id of the project model being viewed | requiresAuth  requiresAuthLevel2 |
| /users | List all users | N/A | requiresAuth  requiresAuthLevel2 |
| /clients | List all clients | N/A | requiresAuth  requiresAuthLevel2 |
| /departments | List all departments | N/A | requiresAuth  requiresAuthLevel2 |
| /:pathMatch(.\*)\* | This is the fallback route. If somebody visits any route not defined, they will be filtered into this route which serves them a user-friendly 404 page | *N/A* | N/A |

API responses

All responses from the API are returned in a JSON format.

Laravel’s *findOrFail* method is used to find a model with the associated ID, or return a 404 http response if the model cannot be found. All responses from the api are returned with the appropriate http response code such as 401, 403, 200, etc.

Validation / Error Handling

Data in the request payload to the api is validated on the server-side. If the data in the request is invalid, user friendly validation error messages are generated and returned in the json response so they can be presented to the user on the frontend.

Other checks are performed in the server-side methods to make sure the request can be successfully carried out, and catch errors so the system can fail gracefully and return user-friendly error messages to the frontend.

Postman tests

There is a collection of Postman requests which can be used to test the API of the endpoints. This importable collection can be found the appendices.

API Middleware

NOTE: All requests are passed through the *BearerTransfer* middleware and the *throttle* middleware.

BearerTransfer - this middleware takes the user’s account token which is sent as a cookie in the request to the api, and transfers the token to the *authorization* header.

auth:api - only allows requests to the route that have been sent by an authorised user. This relies on the *BearerTransfer* middleware to take the access token from a cookie within the request, and transfer it to the *authorization* header which the *auth:api* middleware users to validate the request is authorised.

checkSoftDeletedUser - this checks that the authorised user has not been soft deleted. This middleware stops requests being allowed if the user that has been authorised in the request has been soft deleted.

throttle - this middleware is used to control the amount of requests being sent to the api by utilising rate limiting. This helps protect against malicious behaviour such as DDOS and brute force attacks.

admin - this middleware checks that the authorised api user has admin privileges.

API Endpoints, Routes and Methods

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Controller::method** | **Request Type** | **Middleware** | **Route** | **Dynamic URL Data** | **Request Payload Data** | **Purpose** |
| User | UserController::authUser | GET | Auth:api  checkSoftDeletedUser | /user | N/A | N/A | Return the user model for the authenticated user |
| User | UserController::logout | GET | Auth:api  checkSoftDeletedUser | /logout | N/A | N/A | Revoke the user’s access token and delete the access token cookie |
| User | UserControllre::get | GET | Auth:api  checkSoftDeletedUser  admin | /user/{*userId*} | *userId*: the id of the user model in the database that is being restored | *N/A* | Return the user model details |
| User | /UserController::index | GET | Auth:api  checkSoftDeletedUser  admin | /users | N/A | N/A | Return all user records from the database (that have not been soft deleted) |
| User | UserController::login | POST | N/A | /login | N/A |  | Validate user credentials and generate an access token for the API |
| User | UserController::restore | POST | Auth:api  checkSoftDeletedUser  admin | /user{*userId*}/restore | *userId*: the id of the user model in the database that is being restored |  | Mark the user model as restored in the database |
| User | UserController::softDelete | Delete | Auth:api  checkSoftDeletedUser  admin | /user/{*userId*}/delete | *userId*: the id of the user model in the database that is being deleted |  | Mark the user model as deleted in the database |
| User | UserController::register | PUT | Auth:api  checkSoftDeletedUser  admin | /user/create | N/A |  | Register a new user into the system |
| TimeRecord | TimeRecordController::getTimeRecordsByDate | GET | Auth:api  checkSoftDeletedUser | /timeRecords/date/{*date*} | *date:* The date which time records will be returned for in the format of yyyy-mm-dd | *N/A* | Return all of the time records for the authenticated user that were assigned to the chosen date |
| TimeRecord | TimeRecordController::create | PUT | Auth:api  checkSoftDeletedUser | /timeRecord/create | N/A |  | Create a time record for the authenticated user |
| TimeRecord | TimeRecordController::destroy | DELETE | Auth:api  checkSoftDeletedUser | /timeRecord/{*id*}/delete | *id:* the ID of the time record that is being deleted |  | Mark the time record as deleted (soft deleted) |
| Project | ProjectController::index | GET | Auth:api  checkSoftDeletedUser | /projects | N/A | N/A | Get all of the projects in the database that have not been marked as deleted (soft deleted) |
| Project | ProjectController::get | GET | Auth:api  checkSoftDeletedUser  admin | /project/{*id*} | *id:* the ID of the project that will be retrieved from the database | N/A | Retrieve the project from the database |
| Project | ProjectController::getAnalytics | GET | Auth:api  checkSoftDeletedUser  admin | /project/{*id*}/analytics | *id*: the ID of the project that will have its analytics generated for | N/A | Generate analytics for the project |
| Project | ProjectController::create | PUT | Auth:api  checkSoftDeletedUser  admin | /project/create | N/A |  | Create a new project |
| Project | ProjectController::destroy | DELETE | Auth:api  checkSoftDeletedUser  admin | /project/{*id*}/delete | *id:* the ID of the project that is being deleted |  | Mark the project as soft deleted |
| ProjectStage | ProjectStageController::create | PUT | Auth:api  checkSoftDeletedUser  admin | /projectStage/create | N/A |  | Create a new project stage |
| ProjectStage | ProjectStageController::destroy | DELETE | Auth:api  checkSoftDeletedUser  admin | /projectStage/{*id*}/delete | *id:* the ID of the project stage that is being deleted |  | Mark the project stage as soft deleted |
| Client | ClientController::index | GET | Auth:api  checkSoftDeletedUser  admin | /clients | N/A | N/A | Return all of the clients from the database that have not been soft deleted |
| Client | ClientController::create | PUT | Auth:api  checkSoftDeletedUser  admin | /client/create | N/A |  | Create a new client |
| Client | ClientController::destroy | DELETE | Auth:api  checkSoftDeletedUser  admin | /client/{*id*}/delete | *id:* the ID of the client that is being deleted |  | Mark the client as soft deleted |
| Department | DepartmentController::index | GET | Auth:api  checkSoftDeletedUser  admin | /departments | N/A | N/A | Return all of the departments from the database that have not been soft deleted |
| Department | DepartmentController::create | PUT | Auth:api  checkSoftDeletedUser  admin | /department/create | N/A |  | Create a new department |
| Department | DepartmentController::addUsers | POST | Auth:api  checkSoftDeletedUser  admin | /department/{*id*}/addUsers | *id:* the ID of the department that the users will be allowed access to |  | Assign users access to a department, so they can access the projects assigned to this department |
| Department | DepartmentController::addClients | POST | Auth:api  checkSoftDeletedUser  admin | /department/{*id*}/addClients | *id:* the ID of the department that the clients will be allowed access to |  | Assign a client to a department |
| Department | DepartmentController::delete | DELETE | Auth:api  checkSoftDeletedUser  admin | /department/{*id*}/delete | *id:* the ID of the department that is being soft deleted |  | Mark the department as soft deleted |

Soft Deletion

No instances of models in this system can be hard deleted via the api - the method and route does not exist. When a user performs a deletion via the user interface, the instance of the model is marked as deleted with the current timestamp, allowing the record to be recovered. Any retrieval database interactions via the api ignore records that are marked as deleted, so only non-deleted models are returned in the api response. A cron job (or other script run periodically automatically) to drop all records from the database that have been marked as deleted for a chosen amount of time. This prevents the database from getting bloated, whilst still providing the user a chance to reverse their deletion. If a user is soft deleted, any of their requests to the api are caught in the *checkSoftDeletedUser* middleware and restrict their access.

Frameworks & Languages

Backend

The backend of the system is a RESTful api created using the Laravel 8 framework. This framework is written in PHP (version 7.4 or higher) and follows the MVC methodology.

Frontend

The frontend is written in VueJS (version 3) using the Composition api syntax. This framework is a template-based framework, and is written in JavaScript, HTML and CSS.

Database

The database had been designed to work well with OOP and MVC, which is the architecture of this system. The database is an SQL, relational, database which is designed to be hosted on a PhpMyAdmin web server. The database scheme and relationships are defined within the migrations of the laravel project. Laravel’s *eloquent* system is used as the data access layer within the api to easily query the database.

Source Control and Version Management

The source control and version management is achieved by using GIT with Microsoft Azure. The project can be found at <https://dev.azure.com/oasisazure/Timesheets>. The project contains 2 repos - the frontend and the backend. There are 2 key branches on each repo - the development branch and the master branch. The development branch is used to commit units of functionality that haven’t been released yet, and then development changes are merged into the master branch when the functionality is being deployed.

List of Appendices

|  |  |  |
| --- | --- | --- |
| **Appendix ID** | **Name / Description** | **Location** |
| 1 | Risk analysis and security appraisal | Security and Risk Analysis.docx |
| 2 | Requirements specification | Design/Timesheets System Requirements Specification.docx |
| 3 | Technical Documentation Timesheets Project | Technical Documentation Timesheets Project.docx |
| 4a | UML Class Diagram | Design/UML Class Diagram.drawio |
| 4b | Entity Relationship Diagram | Design/Entity Relationship Diagram.drawio |
| 5a | Static Wireframe | Design/Wireframe.indd |
| 5b | Interactive Prototype | Design/Timesheets Prototype.xd |
| 6a | PostMan collection of tests | Testing/ Timesheet.postman\_collection.json |
| 6b | Full System Test Record #1 | Testing/Timesheets Project Executed Test Records - 11.04.22.docx |
| 6c | Full System Test Record #2 | Testing/Timesheets Project Executed Test Records - 02.05.22.docx |