# 4 Development

## 4.1 Development of the Back-end

In order to develop a functional back-end, a basic Node.js project was setup along with the Express framework as a starting point. This would act as the server for the API endpoints which the front-end will make requests to. The first functionality implemented was the ability to create, edit and delete tasks.

### 4.1.1 Task Methods

```
async function createTask(task, uid) {
    try {
        // Creating a reference to the specified user Tasks database.
        const tasksDbRef = collection(db, "Tasks", uid, "Tasks")
        await addDoc(tasksDbRef, {
            title: task.title,
            duration: parseInt(task.duration),
            priority: task.priority,
            completed: false
        });
    } catch (error) {
        console.log(error);
        throw new Error(error);
    }
}
```

*Figure 28: createTask Back-end API Method*

Figure 28 is a snippet of code used by the back-end to create a new task for the user in the tasks database table. By using try / catch blocks, errors can be handled effectively and reduce the chances of the server crashing if an error is encountered [20]. Try / catch blocks also allow for much improved debugging as they provide useful information about the error, making it easier to identify and fix issues within the code. The code for deleting and editing a task is based on the code in Figure 28 as both are manipulating data from the same database table.

After creating the methods for creating, editing and deleting tasks, an endpoint was created which will allow the front-end to access the method. Firstly, a controller was created for the tasks which is used to handle the HTTP request and response as well as call the relevant method:

```
class TasksApiController {
    createTask(req, res) {
        createTaskApi(req.body, req.authId)
            .then(uid => {
                res.status(200).send(uid)
            })
            .catch(err => {
                res.status(400).send(err)
            })
    }
}
```

*Figure 29: TasksApiController*

All methods used by the back-end require a controller and follow the same structure as Figure 29.

Followed by the routing endpoints:

```
const tasks_router = express.Router();
tasks_router.use(bodyParser.json());

const tasksApiController = require('./tasksApiController');

tasks_router.post(
    '/tasks/createtask',
    authenticate_token,
    tasksApiController['createTask']);

tasks_router.post(
    '/tasks/deletetask',
    authenticate_token,
    tasksApiController['deleteTask']);

tasks_router.post(
    '/tasks/edittask',
    authenticate_token,
    tasksApiController['editTask']);


module.exports = tasks_router;
```

*Figure 30: Tasks Router and Endpoints*

In addition to creating, updating and deleting a task, methods are required to mark a task as complete, as well as allow a user to set a weekly goal on the hours spent completing tasks:

```
async function markCompleted(task, uid) {
    try {
        const tasksDocRef = doc(db, "Tasks", uid);
        const tasksColRef = collection(tasksDocRef, "Tasks")
        await setDoc(doc(tasksColRef, task.id), {
            completed: true,
            completedTime: serverTimestamp(),
            startTime: null
        }, {
            merge: true
        });

    } catch (error) {
        throw new Error(error);
    }
}
```

*Figure 31: markCompleted Back-end API Method*

When markCompleted is called, the completed Boolean is updated to true and the completedTime field is set equal to a timestamp generated by the Firebase server. The startTime field is also updated to null, which is used by the planner function when determining where a task should be placed on the timeline. The merge flag is used by Firebase to determine whether to merge a document or overwrite it.

```
async function setGoal(body, uid) {
    try {
        const preferencesDocRef = doc(db, "UserPreferences", uid);
        await setDoc(preferencesDocRef, {
            goal: body.goal
        }, {
            merge: true
        });

    } catch (error) {
        throw new Error(error);
    }
}
```

*Figure 32: setGoal Back-end API Method*

The setGoal method simply updates the user's goal field in their associated UserPreferences document with the given value specified by the user.

### 4.1.2 Authentication Middleware

In order to reduce the risk of unauthorised changes and access to the database, the back-end will require some form of authentication. After some research, authentication tokens were chosen for this process [21]. Fortunately, Firebase has inbuilt authentication functionality that allows for the generation of authentication tokens for users. This token can then be included in the 'Authorization' header of each HTTP request. The back-end can then utilise the Firebase Admin SDK verifyIdToken method, validating any HTTP request by checking that the user's authentication token included in the header. As seen in the tasks_router post endpoints in Figure 30, 'auth_token' is referenced as a parameter, which means the authentication middleware will be executed before the relevant method is called.

```
const getAuthToken = (req, res, next) => {
    if (
        req.headers.authorization &&
        req.headers.authorization.split(' ')[0] === 'Bearer'
    ) {
        req.authToken = req.headers.authorization.split(' ')[1];
    } else {
        req.authToken = null;
    }
    next();
}

module.exports = (req, res, next) => {
    getAuthToken(req, res, async () => {
        try {
            const { authToken } = req;
            const userInfo = await admin.auth().verifyIdToken(authToken);
            req.authId = userInfo.uid;
            return next();
        } catch (e) {
            return res.status(401).send(
                {
                    error: 'You are not authorized to make this request'
                });
        }
    });
}
```

*Figure 33: Authentication Middleware*

In Figure 33, the getAuthToken function checks if there is an 'authorization' field in the request headers and if the authorization scheme is 'Bearer'. If both conditions are met, it extracts the token and stores it in the 'req.authToken' property; otherwise, it sets 'req.authToken' to null.

The exported middleware function calls the getAuthToken function to extract the token and then moves on to the asynchronous callback. Inside the callback, it first seperates the authToken from the req object. Then, it tries to verify the token using Firebase Admin SDK's verifyIdToken method, which returns a decoded token containing user information if the token is valid. If the token is verified, the user's UID is extracted from the decoded token and stored in req.authId. The middleware then moves on to the next function in the chain using the next() call. If an error occurs during the verification process, the middleware sends a HTTP 401 Unauthorized response with an error message indicating that the user is not authorized to make the request. This response can then be used by the front-end to indicate an error has occurred.

### 4.1.3 User Authentication

Next, under user authentication a method for creating a new user was implemented. This code is also very similar to the createTask method however, it involves adding data to the AccountInformation table in the database. First, the Firebase method createUserWithEmailAndPassword to create a new user is called. Following this, a new entry is created for the user in AccountInformation and UserPreferences which is used to store save any user settings.

```
async function createAccount(newUser) {
    try {
        const cred = await createUserWithEmailAndPassword(auth,
newUser.email, newUser.password);

        await setDoc(doc(db, "AccountInformation", cred.user.uid), {
            firstName: newUser.firstName
        });

        await setDoc(doc(db, "UserPreferences", cred.user.uid), {
            goal: 40
        });
    } catch (error) {
        throw new Error(error);
    }
}
```

*Figure 34: Create Account Method*

### 4.1.4 Planner Methods

In order to implement the planner functionality, the back-end will require the methods to set and remove the start time of a task. By giving a task a start time, the front-end will be able to render the task on a timeline at the correct time allocated by the user.

```
async function setStartTime(task, uid) {
    try {
        const tasksDbRef = collection(db, "Tasks", uid, "Tasks")
        await setDoc(doc(tasksDbRef, task.id), {
            startTime: task.startTime
        }, {
            merge: true
        });

    } catch (error) {
        throw new Error(error);
    }
}
```

*Figure 35: setStartTime Back-end API Method*

And removing the start time:

```
async function removeStartTime(task, uid) {
    try {
        const tasksDbRef = collection(db, "Tasks", uid, "Tasks")
        await setDoc(doc(tasksDbRef, task.id), {
            startTime: null
        }, {
            merge: true
        });

    } catch (error) {
        throw new Error(error);
    }
}
```

*Figure 36: removeStartTime Back-end API Method*