Falkirk FC Web App

Development Stage

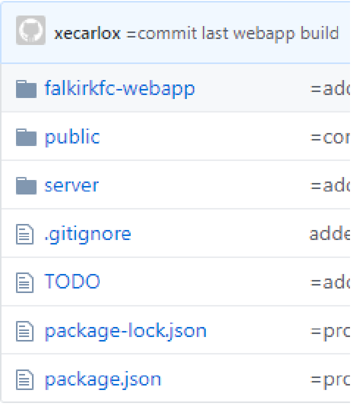
SOFTWARE DEVELOPMENT: GRADED UNIT 2

H48W35/011

Jose Fernandes
EC1637434

Contents

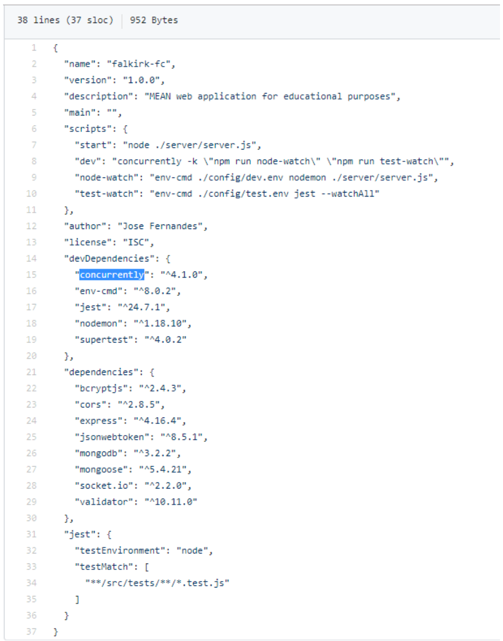
Business model code listing



xecarlox =commit last webapp build

falkirkfc-webapp =ad
public =cor
server =ad
.gitignore adde
TODO =ad
package-lock.json =prc
package.json =prc

The solution produced is a single page application (SPA) which communicates directly with the back-end restfull api app. Even though these two applications are not physically connected, they are developed and deployed in the same environment. The advantage of this is that there is opportunity to allow other types of app, such as mobile as Internet of Things ones applications, to communicate with. The front end SPA can be easily detached from the backend, being developed and deployed on other environment, and still communicate with the restfull api. The project's git repository (available at https://github.com/xecarlox94/FalkirkFC) contains "falkirkfc-webapp", "public" and "server" folders. The "falkirkfc-webapp" is where the SPA is developed and the public folder is where the SPA is compiled for production.

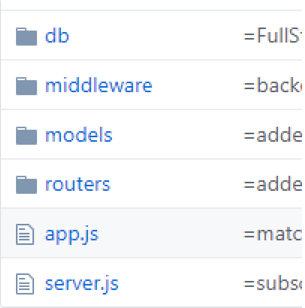


38 lines (37 sloc) | 952 Bytes

```
{
  "name": "falkirk-fc",
  "version": "1.0.0",
  "description": "MEAN web application for educational purposes",
  "main": "",
  "scripts": {
    "start": "node ./server/server.js",
    "dev": "concurrently -k \"npm run node-watch\" \"npm run test-watch\"",
    "node-watch": "env-cmd ./config/dev.env nodemon ./server/server.js",
    "test-watch": "env-cmd ./config/test.env jest --watchAll"
  },
  "author": "Jose Fernandes",
  "license": "ISC",
  "devDependencies": {
    "concurrently": "^4.1.0",
    "env-cmd": "^8.0.2",
    "jest": "^24.7.1",
    "nodemon": "^1.18.10",
    "supertest": "^4.0.2"
  },
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "cors": "^2.8.5",
    "express": "^4.16.4",
    "jsonwebtoken": "^8.5.1",
    "mongodb": "^3.2.2",
    "mongoose": "^5.4.21",
    "socket.io": "^2.2.0",
    "validator": "^10.11.0"
  },
  "jest": {
    "testEnvironment": "node",
    "testMatch": [
      "**/src/tests/**/*.test.js"
    ]
  }
}
```

The package.json file contains all the overall projects's information, configuration and scripts necessary. It also lists all the node pendencies needed to be installed and the respective version. The development dependencies are not installed in production environment, when pushed to a production server. The scripts are also important to initialize the application, for example the "start" script initializes the application in the production server. Additionally the testing configuration is also set up in this file.

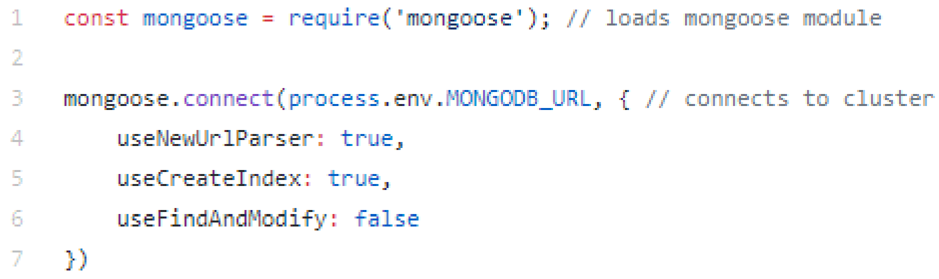
The .gitignore file is important to ignore files and folders which should not be commited, for safety and performance reasons. The "node_modules" folder must be always ignored, as they can be installed any time. This .ignore file also ignores the development and test environment variables needed, located in the local machine and loaded by the "env-cmd" module.



db =FullS
middleware =back
models =adde
routers =adde
app.js =matc
server.js =subs

The server folder contains db (database), middleware, models, routers folders.

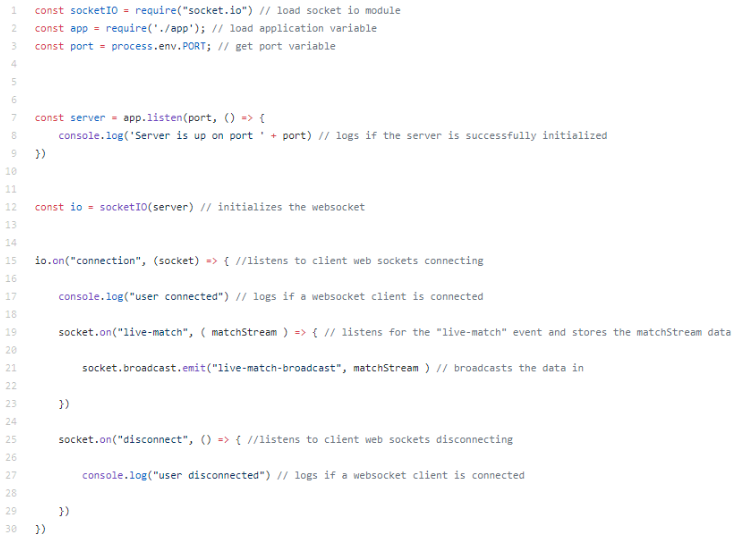
The database folder constains the mongoose.js file which connects the mongodb database which is provided by the mongoAtlas cluster service, using the uri string containing the location and credentials required to store and retrieve data.



```
const mongoose = require('mongoose'); // loads mongoose module

mongoose.connect(process.env.MONGODB_URL, { // connects to cluster
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false
})
```

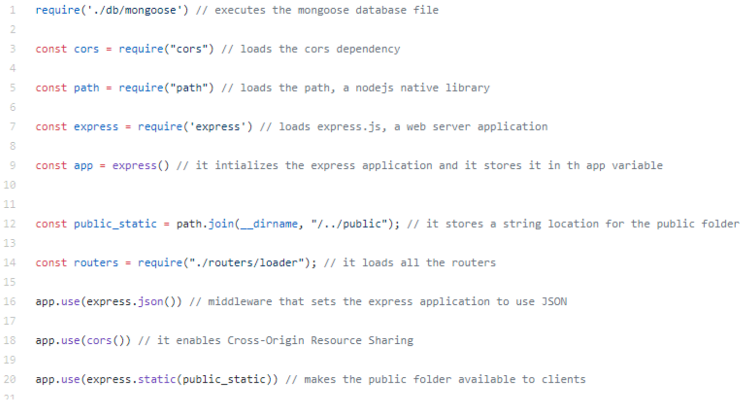
The “start script” executes the server.js file that loads the SocketIO module, stores the environment port variable along with the app.js file. The server is then started and stored in the server variable, by calling the listen function that allows the server to be available for users.

The web socket is also initialized by passing the existing server variable to integrate real-time communication in this application. This web socket is listening for the connection, disconnect and live-match events. When the live-match event is sent by one of the clients, the server will get the data stored in the “matchStream” parameter variable and it will broadcast this live data to all clients.



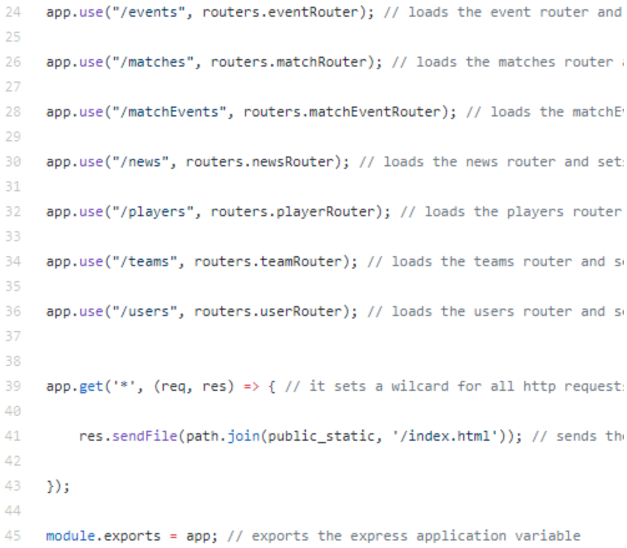
```javascript
const socketIO = require("socket.io") // load socket io module
const app = require('./app'); // load application variable
const port = process.env.PORT; // get port variable



const server = app.listen(port, () => {
    console.log('Server is up on port ' + port) // logs if the server is successfully initialized
})


const io = socketIO(server) // initializes the websocket


io.on("connection", (socket) => { //listens to client web sockets connecting

    console.log("user connected") // logs if a websocket client is connected

    socket.on("live-match", ( matchStream ) => { // listens for the "live-match" event and stores the matchStream data

        socket.broadcast.emit("live-match-broadcast", matchStream ) // broadcasts the data in

    })

    socket.on("disconnect", () => { //listens to client web sockets disconnecting

        console.log("user disconnected") // logs if a websocket client is connected

    })
})
```

The application variable runs the mongoose file, loads the cors, path and express modules, before creating the express application. Apart from sharing the same development environment, these two applications also share the same server. By default all the native webserver do not allow Cross-Origin Resource Sharing but by including cors module in the application middleware is capable of getting requests from the front end. The JSON is also enabled by calling the express json method. The public folder is also made available to clients by using the string containing the directory directory and the express static method. Furthermore, the routers integrating alongside each global url path.



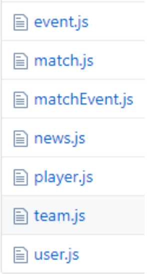
```javascript
require('./db/mongoose') // executes the mongoose database file

const cors = require("cors") // loads the cors dependency

const path = require("path") // loads the path, a nodejs native library

const express = require('express') // loads express.js, a web server application

const app = express() // it intializes the express application and it stores it in th app variable


const public_static = path.join(__dirname, "/../public"); // it stores a string location for the public folder

const routers = require("./routers/loader"); // it loads all the routers

app.use(express.json()) // middleware that sets the express application to use JSON

app.use(cors()) // it enables Cross-Origin Resource Sharing

app.use(express.static(public_static)) // makes the public folder available to clients
```

The application is set by default to always send the index.html file to the client’s browser by default. This ensures the SPA is always loaded before any other request made by the client.

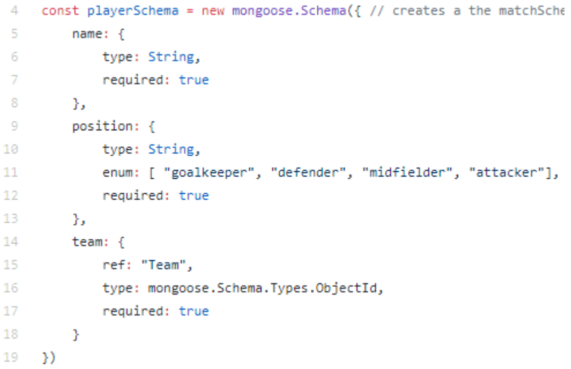


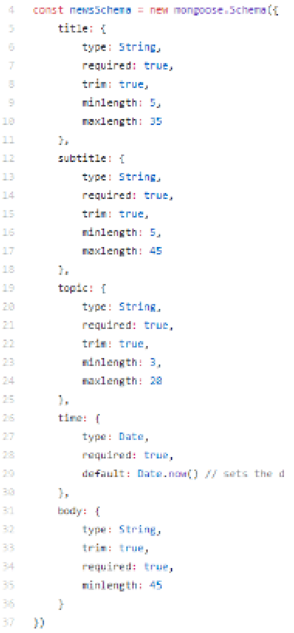
```javascript
app.use("/events", routers.eventRouter); // loads the event router and

app.use("/matches", routers.matchRouter); // loads the matches router

app.use("/matchEvents", routers.matchEventRouter); // loads the matchE

app.use("/news", routers.newsRouter); // loads the news router and set

app.use("/players", routers.playerRouter); // loads the players router

app.use("/teams", routers.teamRouter); // loads the teams router and s

app.use("/users", routers.userRouter); // loads the users router and s


app.get('*', (req, res) => { // it sets a wilcard for all http request

    res.sendFile(path.join(public_static, '/index.html')); // sends th

});

module.exports = app; // exports the express application variable
```

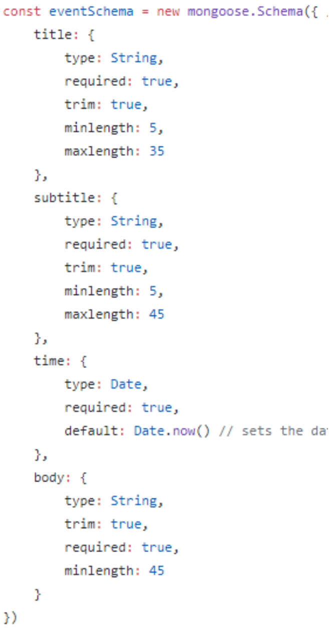
The models folder, inside server, contains all the models being used in this application. This includes the event, match, match events, news, player, team and user models. These models contain all the properties, including their validation, as well as instance and model methods.


```
event.js
match.js
matchEvent.js
news.js
player.js
team.js
user.js
```

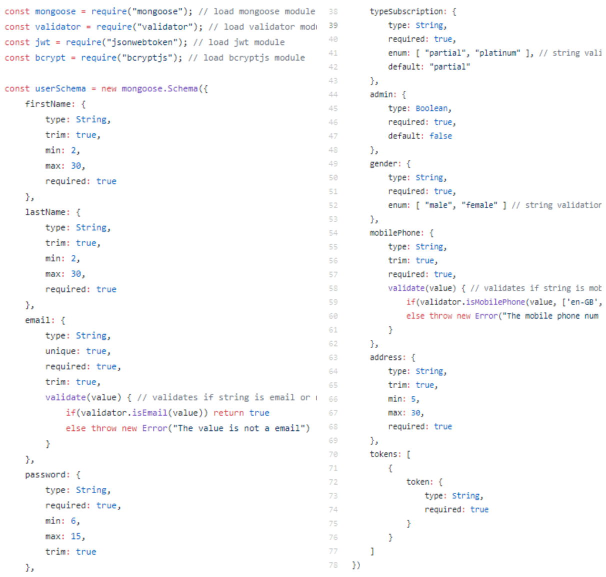
The event, news and player models only contains properties, specified in the image, alongside the default mongoose methods. The player model has additional string validation, that ensures the player's position property has only a few valid values.


```
const playerSchema = new mongoose.Schema({ // creates a the matchSche
    name: {
        type: String,
        required: true
    },
    position: {
        type: String,
        enum: [ "goalkeeper", "defender", "midfielder", "attacker"],
        required: true
    },
    team: {
        ref: "Team",
        type: mongoose.Schema.Types.ObjectId,
        required: true
    }
})
```


```
const newsSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true,
        trim: true,
        minlength: 5,
        maxlength: 35
    },
    subtitle: {
        type: String,
        required: true,
        trim: true,
        minlength: 5,
        maxlength: 45
    },
    topic: {
        type: String,
        required: true,
        trim: true,
        minlength: 3,
        maxlength: 20
    },
    time: {
        type: Date,
        required: true,
        default: Date.now() // sets the d
    },
    body: {
        type: String,
        trim: true,
        required: true,
        minlength: 45
    }
})
```

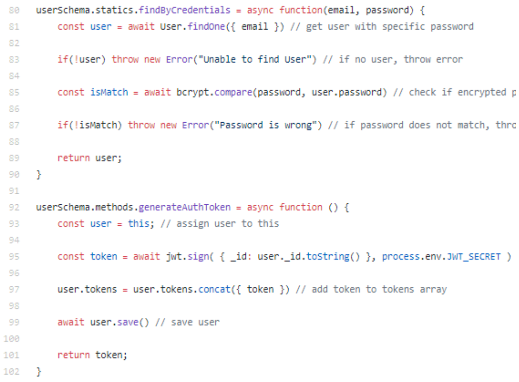

```
const eventSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true,
        trim: true,
        minlength: 5,
        maxlength: 35
    },
    subtitle: {
        type: String,
        required: true,
        trim: true,
        minlength: 5,
        maxlength: 45
    },
    time: {
        type: Date,
        required: true,
        default: Date.now() // sets the da
    },
    body: {
        type: String,
        trim: true,
        required: true,
        minlength: 45
    }
})
```

The User model is the most complex model because it is used as a normal resource as well as an authenticator throughout the application. As every model, it starts by loading the mongoose module but it adds extra modules such as validator, bcrypt and jsonwebtoken. Its string properties are the first name, last name, password, type of subscription, gender, mobile phone and address. The type of subscription and gender are both validated as enumerable strings. The email and mobile phone string properties are validated using the validator library. The admin property is a boolean that determines if the user is an administrator or a subscriber. The tokens array stores all the authentication tokens each user has.


```
const mongoose = require("mongoose"); // load mongoose module
const validator = require("validator"); // load validator modu
const jwt = require("jsonwebtoken"); // load jwt module
const bcrypt = require("bcryptjs"); // load bcryptjs module

const userSchema = new mongoose.Schema({
    firstName: {
        type: String,
        trim: true,
        min: 2,
        max: 30,
        required: true
    },
    lastName: {
        type: String,
        trim: true,
        min: 2,
        max: 30,
        required: true
    },
    email: {
        type: String,
        unique: true,
        required: true,
        trim: true,
        validate(value) { // validates if string is email or
            if(validator.isEmail(value)) return true
            else throw new Error("The value is not a email")
        }
    },
    password: {
        type: String,
        required: true,
        min: 6,
        max: 15,
        trim: true
    },
```
```
    typeSubscription: {
        type: String,
        required: true,
        enum: [ "partial", "platinum" ], // string vali
        default: "partial"
    },
    admin: {
        type: Boolean,
        required: true,
        default: false
    },
    gender: {
        type: String,
        required: true,
        enum: [ "male", "female" ] // string validation
    },
    mobilePhone: {
        type: String,
        trim: true,
        required: true,
        validate(value) { // validates if string is mob
            if(validator.isMobilePhone(value, ['en-GB',
            else throw new Error("The mobile phone num
        }
    },
    address: {
        type: String,
        trim: true,
        min: 5,
        max: 30,
        required: true
    },
    tokens: [
        {
            token: {
                type: String,
                required: true
            }
        }
    ]
})
```

Additionally, the user model has a model method that finds a user by email and password. If no user is found an Error is thrown. The password is also compared with the encrypted password stored in the database if not an Error will also be thrown.

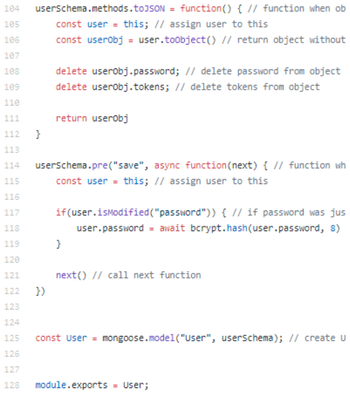
The instance method generateAuthToken generates an authentication token and stores it in the tokens array of its user.


```
userSchema.statics.findByCredentials = async function(email, password) {
    const user = await User.findOne({ email }) // get user with specific password

    if(!user) throw new Error("Unable to find User") // if no user, throw error

    const isMatch = await bcrypt.compare(password, user.password) // check if encrypted p

    if(!isMatch) throw new Error("Password is wrong") // if password does not match, thro

    return user;
}

userSchema.methods.generateAuthToken = async function () {
    const user = this; // assign user to this

    const token = await jwt.sign( { _id: user._id.toString() }, process.env.JWT_SECRET )

    user.tokens = user.tokens.concat({ token }) // add token to tokens array

    await user.save() // save user

    return token;
}
```

The instance method toJSON is actually a method override which remove all the instance methods and deletes the password and the tokens before every time the user object is sent via JSON.

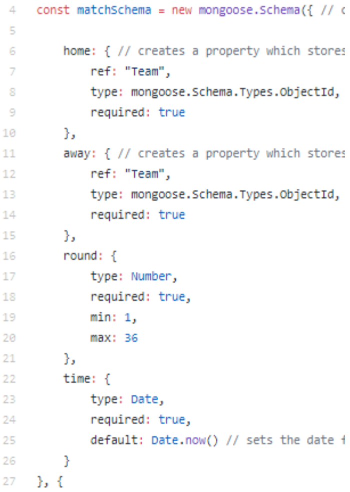
A middleware is also integrated to verify if the user password property was modified and if it was the current password will be encrypted again.

As all the model, this model is created from its schema and exported to be used throughout the application



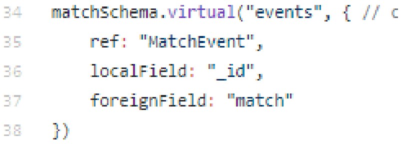
```
userSchema.methods.toJSON = function() { // function when ob
    const user = this; // assign user to this
    const userObj = user.toObject() // return object without

    delete userObj.password; // delete password from object
    delete userObj.tokens; // delete tokens from object

    return userObj
}

userSchema.pre("save", async function(next) { // function wh
    const user = this; // assign user to this

    if(user.isModified("password")) { // if password was jus
        user.password = await bcrypt.hash(user.password, 8)
    }

    next() // call next function
})


const User = mongoose.model("User", userSchema); // create U


module.exports = User;
```

The Match model is what relates the teams, players and match events. Its properties home and away store the id of the teams that played that match. The round property identifies which round the game was played and the time identifies the time when the game was played.



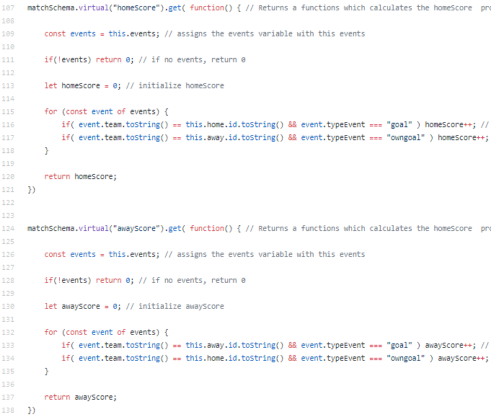
```
const matchSchema = new mongoose.Schema({ // c

    home: { // creates a property which stores
        ref: "Team",
        type: mongoose.Schema.Types.ObjectId,
        required: true
    },
    away: { // creates a property which stores
        ref: "Team",
        type: mongoose.Schema.Types.ObjectId,
        required: true
    },
    round: {
        type: Number,
        required: true,
        min: 1,
        max: 36
    },
    time: {
        type: Date,
        required: true,
        default: Date.now() // sets the date f
    }
}, {
```

The events virtual property relates all the match events with the respective match. The match event stores additional information which will store information needed for further calculation.



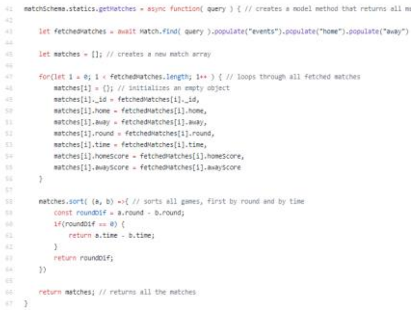
```
matchSchema.virtual("events", { // c
    ref: "MatchEvent",
    localField: "_id",
    foreignField: "match"
})
```

The properties home and away score are not native of the object but they are calculated each time the object is requested. These properties are returned by a function that loops through events and determines the scores of the match

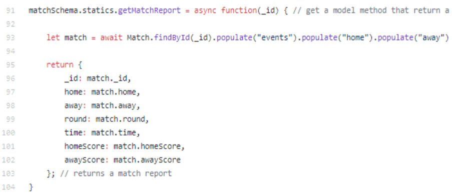


```
matchSchema.virtual("homeScore").get( function() { // Returns a functions which calculates the homeScore  pr

    const events = this.events; // assigns the events variable with this events

    if(!events) return 0; // if no events, return 0

    let homeScore = 0; // initialize homeScore

    for (const event of events) {
        if( event.team.toString() == this.home.id.toString() && event.typeEvent === "goal" ) homeScore++; //
        if( event.team.toString() == this.away.id.toString() && event.typeEvent === "owngoal" ) homeScore++;
    }

    return homeScore;
})


matchSchema.virtual("awayScore").get( function() { // Returns a functions which calculates the homeScore  pr

    const events = this.events; // assigns the events variable with this events

    if(!events) return 0; // if no events, return 0

    let awayScore = 0; // initialize awayScore

    for (const event of events) {
        if( event.team.toString() == this.away.id.toString() && event.typeEvent === "goal" ) awayScore++; //
        if( event.team.toString() == this.home.id.toString() && event.typeEvent === "owngoal" ) awayScore++;
    }

    return awayScore;
})
```

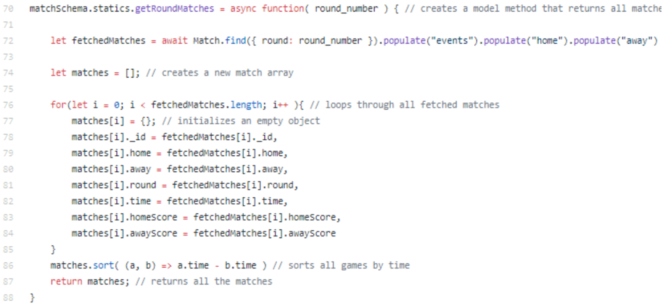
The match model has also extra model methods that include getMatchReport, getMatches and getRoundMatches. The getMatchreport finds a match by id and populates all the virtual fields namely events, home and away fields. The getMatches populates all virtual fields returns all the matches and sorts them by round and time. The getRoundMatches works the same as getMatches but it returns matches of each round and sorts them by time.



```
matchSchema.statics.getMatches = async function( query ) { // creates a model method that returns all m

    let fetchedMatches = await Match.find( query ).populate("events").populate("home").populate("away")

    let matches = []; // creates a new match array

    for(let i = 0; i < fetchedMatches.length; i++ ) { // loops through all fetched matches
        matches[i] = {}; // initializes an empty object
        matches[i]._id = fetchedMatches[i]._id,
        matches[i].home = fetchedMatches[i].home,
        matches[i].away = fetchedMatches[i].away,
        matches[i].round = fetchedMatches[i].round,
        matches[i].time = fetchedMatches[i].time,
        matches[i].homeScore = fetchedMatches[i].homeScore,
        matches[i].awayScore = fetchedMatches[i].awayScore
    }

    matches.sort( (a, b) =>{ // sorts all games, first by round and by time
        const roundDif = a.round - b.round;
        if(roundDif == 0) {
            return a.time - b.time;
        }
        return roundDif;
    })

    return matches; // returns all the matches
}
```



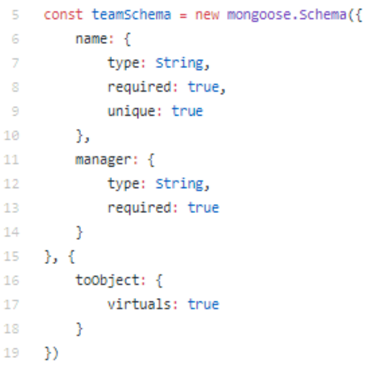
```
matchSchema.statics.getMatchReport = async function(_id) { // get a model method that return a

    let match = await Match.findById(_id).populate("events").populate("home").populate("away")

    return {
        _id: match._id,
        home: match.home,
        away: match.away,
        round: match.round,
        time: match.time,
        homeScore: match.homeScore,
        awayScore: match.awayScore
    }; // returns a match report
}
```



```
matchSchema.statics.getRoundMatches = async function( round_number ) { // creates a model method that returns all matche

    let fetchedMatches = await Match.find({ round: round_number }).populate("events").populate("home").populate("away")

    let matches = []; // creates a new match array

    for(let i = 0; i < fetchedMatches.length; i++ ){ // loops through all fetched matches
        matches[i] = {}; // initializes an empty object
        matches[i]._id = fetchedMatches[i]._id,
        matches[i].home = fetchedMatches[i].home,
        matches[i].away = fetchedMatches[i].away,
        matches[i].round = fetchedMatches[i].round,
        matches[i].time = fetchedMatches[i].time,
        matches[i].homeScore = fetchedMatches[i].homeScore,
        matches[i].awayScore = fetchedMatches[i].awayScore
    }
    matches.sort( (a, b) => a.time - b.time ) // sorts all games by time
    return matches; // returns all the matches
}
```

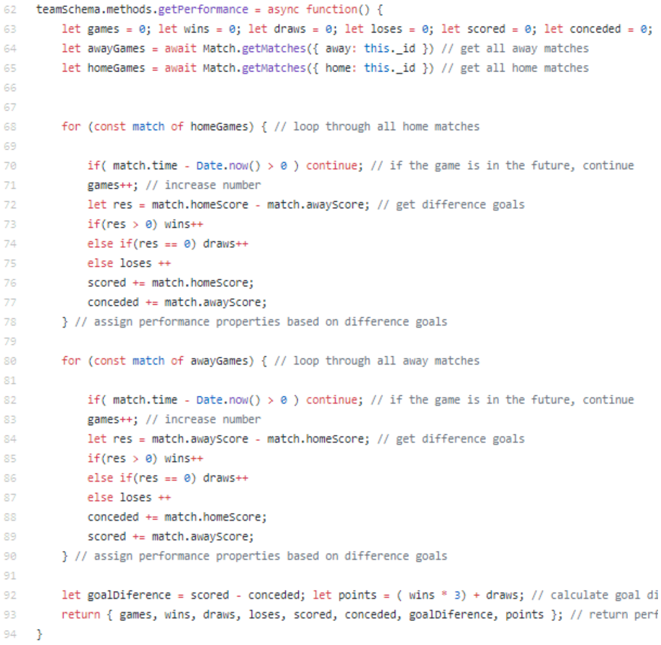
The team model has the name and manager string types. The name property describes the clubs name and it is set to be a unique string to avoid duplication.

The players virtual property is also set, which aggregates players to their team. This way it is possible to load entire squads.



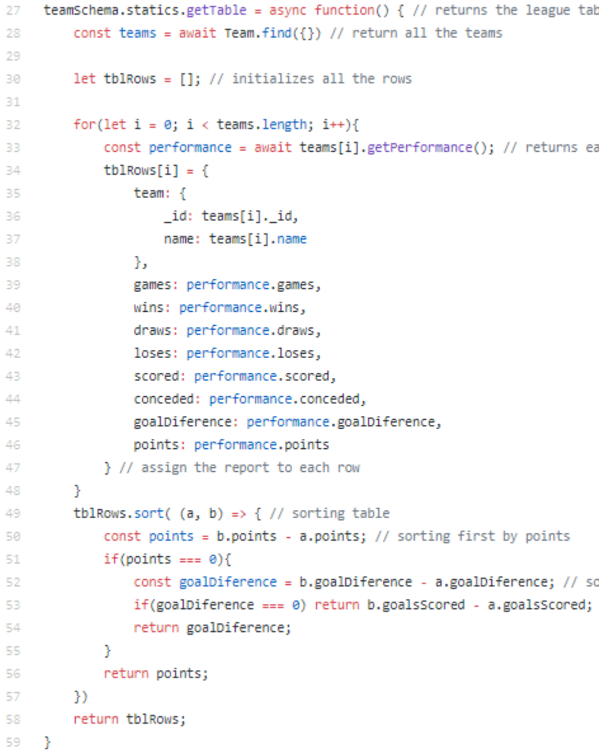
```
const teamSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        unique: true
    },
    manager: {
        type: String,
        required: true
    }
}, {
    toObject: {
        virtuals: true
    }
})
```

The team model has instance the getPerformance method to return the team’s performance in the league. It accomplishes that by loading all the away and home matches, after loading the Match model at the file’s beginning. Following that the away home matches will be looped through individually calculating the games, wins, draws, loses, scored and conceded variables. The goal difference and points will be calculated after looping all the games.



```
teamSchema.methods.getPerformance = async function() {
    let games = 0; let wins = 0; let draws = 0; let loses = 0; let scored = 0; let conceded = 0;
    let awayGames = await Match.getMatches({ away: this._id }) // get all away matches
    let homeGames = await Match.getMatches({ home: this._id }) // get all home matches


    for (const match of homeGames) { // loop through all home matches

        if( match.time - Date.now() > 0 ) continue; // if the game is in the future, continue
        games++; // increase number
        let res = match.homeScore - match.awayScore; // get difference goals
        if(res > 0) wins++
        else if(res == 0) draws++
        else loses ++
        scored += match.homeScore;
        conceded += match.awayScore;
    } // assign performance properties based on difference goals

    for (const match of awayGames) { // loop through all away matches

        if( match.time - Date.now() > 0 ) continue; // if the game is in the future, continue
        games++; // increase number
        let res = match.awayScore - match.homeScore; // get difference goals
        if(res > 0) wins++
        else if(res == 0) draws++
        else loses ++
        conceded += match.homeScore;
        scored += match.awayScore;
    } // assign performance properties based on difference goals

    let goalDiference = scored - conceded; let points = ( wins * 3) + draws; // calculate goal di
    return { games, wins, draws, loses, scored, conceded, goalDiference, points }; // return perf
}
```

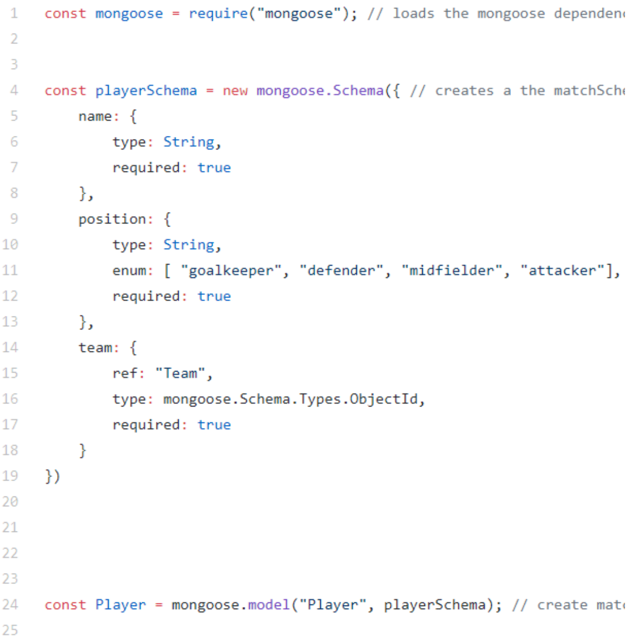
The getTable model method, gathers all teams and loops through them returning an array of object containing the id and name of each team alongside its performance. Following that the array is then sorted by points, goal difference and goals scored.



```
teamSchema.statics.getTable = async function() { // returns the league tab
    const teams = await Team.find({}) // return all the teams

    let tblRows = []; // initializes all the rows

    for(let i = 0; i < teams.length; i++){
        const performance = await teams[i].getPerformance(); // returns ea
        tblRows[i] = {
            team: {
                _id: teams[i]._id,
                name: teams[i].name
            },
            games: performance.games,
            wins: performance.wins,
            draws: performance.draws,
            loses: performance.loses,
            scored: performance.scored,
            conceded: performance.conceded,
            goalDiference: performance.goalDiference,
            points: performance.points
        } // assign the report to each row
    }
    tblRows.sort( (a, b) => { // sorting table
        const points = b.points - a.points; // sorting first by points
        if(points === 0){
            const goalDiference = b.goalDiference - a.goalDiference; // sc
            if(goalDiference === 0) return b.goalsScored - a.goalsScored;
            return goalDiference;
        }
        return points;
    })
    return tblRows;
}
```

The player model has the name and position attributes as strings. The position string attribute is a string enumerable to ensure the player only has certain positions. This data is necessary for the user to identify each player in the web application.

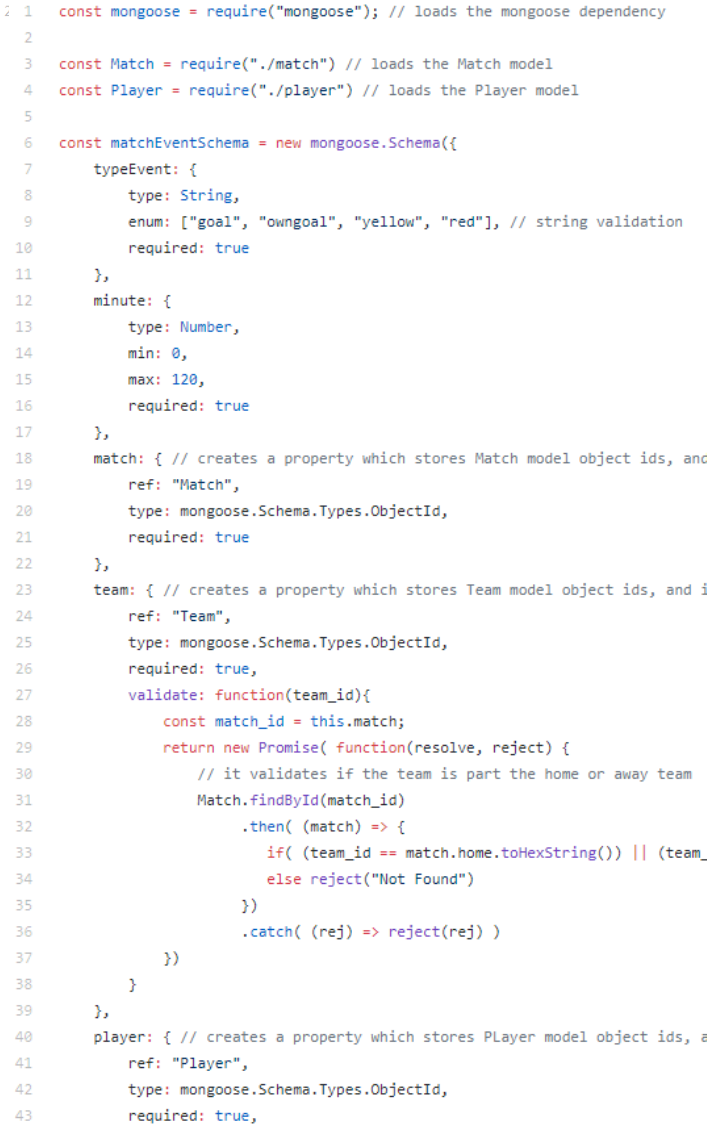
The player model also has a team field which store its team's object id, to relate player and teams.



```
const mongoose = require("mongoose"); // loads the mongoose dependen


const playerSchema = new mongoose.Schema({ // creates a the matchSch
    name: {
        type: String,
        required: true
    },
    position: {
        type: String,
        enum: [ "goalkeeper", "defender", "midfielder", "attacker"],
        required: true
    },
    team: {
        ref: "Team",
        type: mongoose.Schema.Types.ObjectId,
        required: true
    }
})




const Player = mongoose.model("Player", playerSchema); // create mat

```

The match event model is composed of a typeEvent string type enumerable field, minute number field as well as objects id for the match, team and player models.

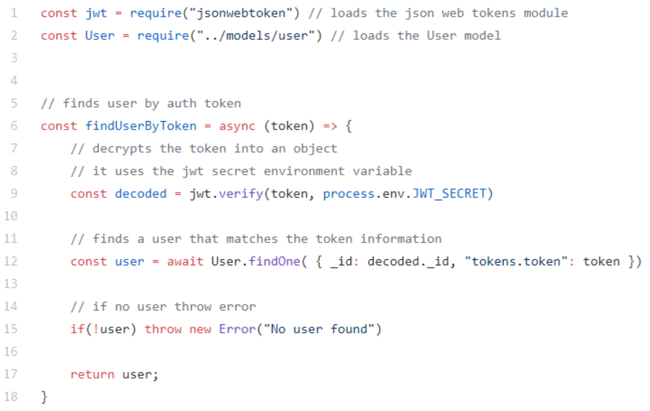
The relationship established between match event, match, team and player is the key for whole system has it calculates all data necessary for a league. The validators will ensure the player that commits the event, is part of the team which is playing the match. The models are loaded to be then used asynchronously in the validation of each match event.



```
const mongoose = require("mongoose"); // loads the mongoose dependency

const Match = require("./match") // loads the Match model
const Player = require("./player") // loads the Player model

const matchEventSchema = new mongoose.Schema({
    typeEvent: {
        type: String,
        enum: ["goal", "owngoal", "yellow", "red"], // string validation
        required: true
    },
    minute: {
        type: Number,
        min: 0,
        max: 120,
        required: true
    },
    match: { // creates a property which stores Match model object ids, anc
        ref: "Match",
        type: mongoose.Schema.Types.ObjectId,
        required: true
    },
    team: { // creates a property which stores Team model object ids, and i
        ref: "Team",
        type: mongoose.Schema.Types.ObjectId,
        required: true,
        validate: function(team_id){
            const match_id = this.match;
            return new Promise( function(resolve, reject) {
                // it validates if the team is part the home or away team
                Match.findById(match_id)
                    .then( (match) => {
                        if( (team_id == match.home.toHexString()) || (team_
                        else reject("Not Found")
                    })
                    .catch( (rej) => reject(rej) )
            })
        }
    },
    player: { // creates a property which stores PLayer model object ids, a
        ref: "Player",
        type: mongoose.Schema.Types.ObjectId,
        required: true,
```

A fundamental part of this server is the http handling, that is done by the individual routers that manipulate each specific resource using the models. Authentication and Authorization are necessary for the secure and correct manipulation of data, including personal user information, and are achieved using express middleware that runs before any http handler function, manipulating the http request object.

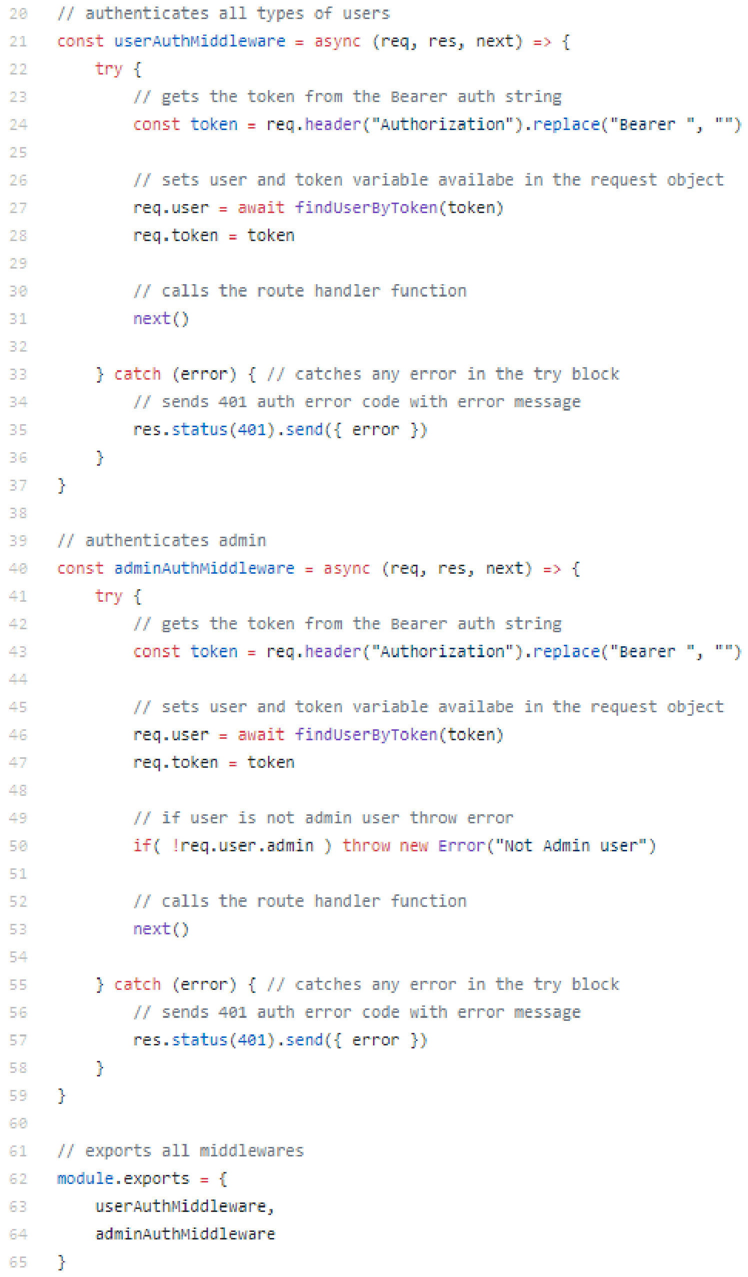
The auth authentication file is in the middleware folder, inside the server folder. This file loads the jwt module and user model and it declares a function which decrypts the authentication token and returns the user which matches the token information.



```
const jwt = require("jsonwebtoken") // loads the json web tokens module
const User = require("../models/user") // loads the User model


// finds user by auth token
const findUserByToken = async (token) => {
    // decrypts the token into an object
    // it uses the jwt secret environment variable
    const decoded = jwt.verify(token, process.env.JWT_SECRET)

    // finds a user that matches the token information
    const user = await User.findOne( { _id: decoded._id, "tokens.token": token })

    // if no user throw error
    if(!user) throw new Error("No user found")

    return user;
}
```

The userAuthMiddleware and adminAuthMiddleware are asynchronous functions, as the finUserByToken, which consume promises by using the keywords “await” to stop the execution of a promise or throw a promise error and “async” that transforms the function into a promise itself. The middleware functions decryt the token string and if a user matching the token data is found the next http handler function is executed otherwise an error code 401 (an authentication code) is sent to the front-end application. By sending the 401 response, any other http handler is stopped without causing warm to the system’s data.

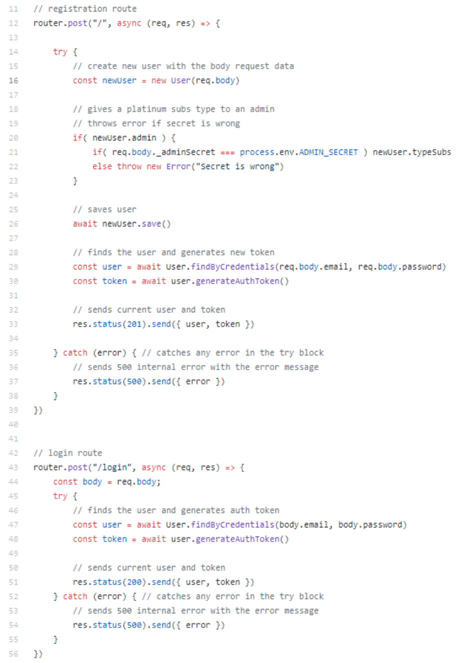
The userAuthMiddleware is meant to authenticate subscribers and administrators but the adminAuthMiddleware only authenticates administrators. It will additionally throw an error informing that the current user is not an administrator.



```
// authenticates all types of users
const userAuthMiddleware = async (req, res, next) => {
    try {
        // gets the token from the Bearer auth string
        const token = req.header("Authorization").replace("Bearer ", "")

        // sets user and token variable availabe in the request object
        req.user = await findUserByToken(token)
        req.token = token

        // calls the route handler function
        next()

    } catch (error) { // catches any error in the try block
        // sends 401 auth error code with error message
        res.status(401).send({ error })
    }
}

// authenticates admin
const adminAuthMiddleware = async (req, res, next) => {
    try {
        // gets the token from the Bearer auth string
        const token = req.header("Authorization").replace("Bearer ", "")

        // sets user and token variable availabe in the request object
        req.user = await findUserByToken(token)
        req.token = token

        // if user is not admin user throw error
        if( !req.user.admin ) throw new Error("Not Admin user")

        // calls the route handler function
        next()

    } catch (error) { // catches any error in the try block
        // sends 401 auth error code with error message
        res.status(401).send({ error })
    }
}

// exports all middlewares
module.exports = {
    userAuthMiddleware,
    adminAuthMiddleware
}
```

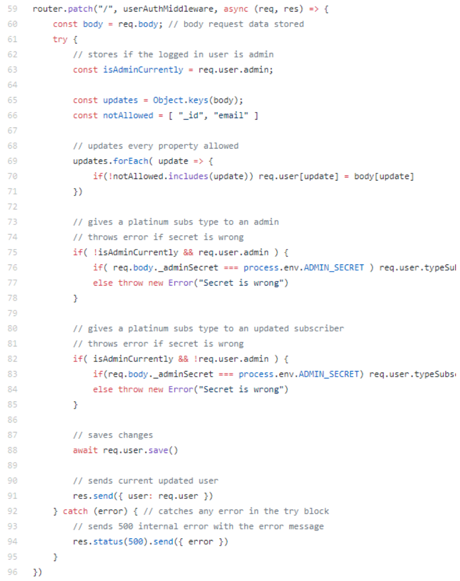
The most important router in this application is the user router, located in the routers folder which is inside the server folder. This file initiates an express router, by loading the express module, and it exports a router that is added to the express application located in the app file. This design allows to integrate many routers, controlling each resource, to extend the application functionalities. The router also loads the authentication middleware function and the user model. The user model is integrated in the authentication system, but it is also used as a resource that is managed by any administrator.

The only routers in this application which do not have a middleware set is the login and the register ones. These routers must be available because they authenticate a user logging in or registering in the application. In the register route, a user is created using the request body and subsequently the user is checked to determine if it is admin. If the user is admin and the admin secret is true the admin user will be assigned with a “platinum” subscription and saved. Afterwards, as in the login route, the user is found using the email and password and it is generated a new authentication token. The response body in these two routes is the current user object authenticated and the token to be used on the http headers.



```
// registration route
router.post("/", async (req, res) => {

    try {
        // create new user with the body request data
        const newUser = new User(req.body)

        // gives a platinum subs type to an admin
        // throws error if secret is wrong
        if( newUser.admin ) {
            if( req.body._adminSecret === process.env.ADMIN_SECRET ) newUser.typeSubs
            else throw new Error("Secret is wrong")
        }

        // saves user
        await newUser.save()

        // finds the user and generates new token
        const user = await User.findByCredentials(req.body.email, req.body.password)
        const token = await user.generateAuthToken()

        // sends current user and token
        res.status(201).send({ user, token })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})


// login route
router.post("/login", async (req, res) => {
    const body = req.body;
    try {
        // finds the user and generates auth token
        const user = await User.findByCredentials(body.email, body.password)
        const token = await user.generateAuthToken()

        // sends current user and token
        res.status(200).send({ user, token })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

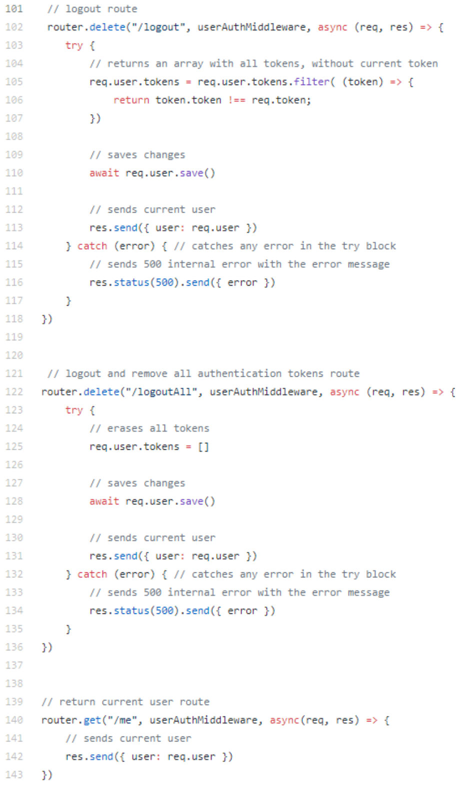
The update route (using the patch http verb) uses the userAuthMiddleware to ensure only authorized users can access this functionality. The function updates all allowed fields to be updated as well as change the subscription as the user changes from admin to normal subscriber an vice-versa.



```
router.patch("/", userAuthMiddleware, async (req, res) => {
    const body = req.body; // body request data stored
    try {
        // stores if the logged in user is admin
        const isAdminCurrently = req.user.admin;

        const updates = Object.keys(body);
        const notAllowed = [ "_id", "email" ]

        // updates every property allowed
        updates.forEach( update => {
            if(!notAllowed.includes(update)) req.user[update] = body[update]
        })

        // gives a platinum subs type to an admin
        // throws error if secret is wrong
        if( !isAdminCurrently && req.user.admin ) {
            if( req.body._adminSecret === process.env.ADMIN_SECRET ) req.user.typeSul
            else throw new Error("Secret is wrong")
        }

        // gives a platinum subs type to an updated subscriber
        // throws error if secret is wrong
        if( isAdminCurrently && !req.user.admin ) {
            if(req.body._adminSecret === process.env.ADMIN_SECRET) req.user.typeSubs
            else throw new Error("Secret is wrong")
        }

        // saves changes
        await req.user.save()

        // sends current updated user
        res.send({ user: req.user })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

The logout and logoutAll routes are authenticated for all user and have similar intents. The logout route filters the token which is currently authenticating the session and it logs the user out. The logoutAll route deletes all tokens present in the user object and logs out the user from the application.

The me route uses the user authenticated in the users middleware and sends its object to the client.

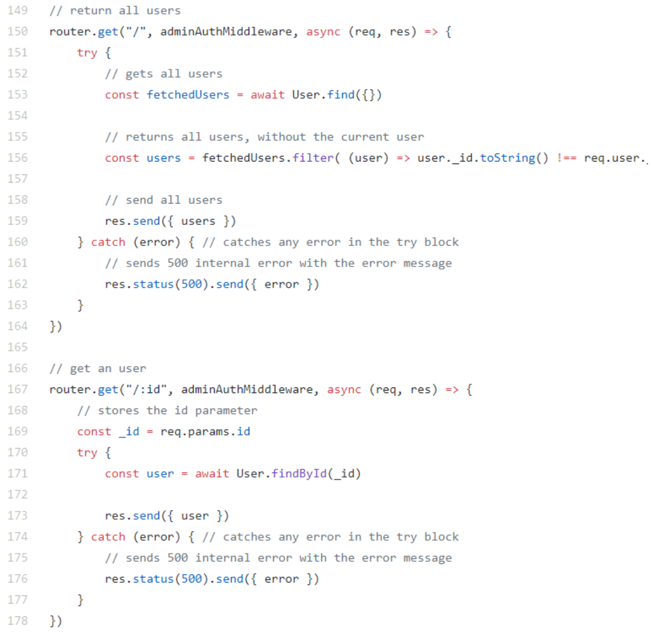


```
// logout route
router.delete("/logout", userAuthMiddleware, async (req, res) => {
    try {
        // returns an array with all tokens, without current token
        req.user.tokens = req.user.tokens.filter( (token) => {
            return token.token !== req.token;
        })

        // saves changes
        await req.user.save()

        // sends current user
        res.send({ user: req.user })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})


// logout and remove all authentication tokens route
router.delete("/logoutAll", userAuthMiddleware, async (req, res) => {
    try {
        // erases all tokens
        req.user.tokens = []

        // saves changes
        await req.user.save()

        // sends current user
        res.send({ user: req.user })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})


// return current user route
router.get("/me", userAuthMiddleware, async(req, res) => {
    // sends current user
    res.send({ user: req.user })
})
```

The user’s router is also responsible for manipulating the users as resources. All of these routes have the administrator middleware meaning that only a user with the administrator type of account can use them. The administrator can get all users, one specific, update or delete the users accounts in the system.

The get route, fetches asynchronously all the users in the system and it filters its own account in this search. This filtering is only to reduce redundancy of functionality in the authentication routes.

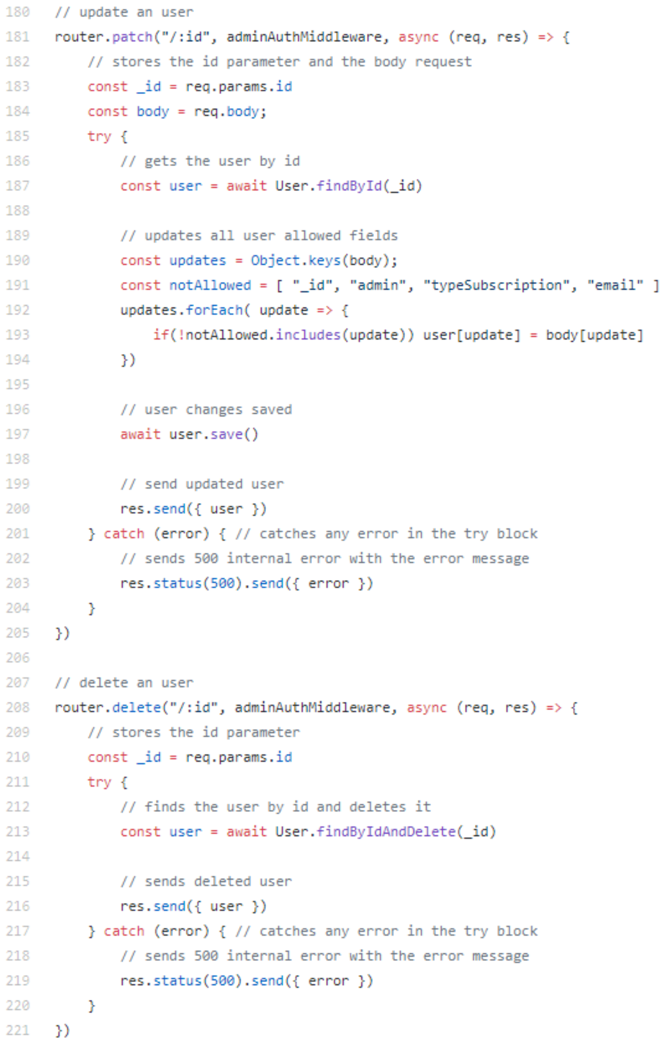
The users can be fetched individually by their object id to allow further management operation such as viewing the personal information, updating user information and even deleting the account.



```
// return all users
router.get("/", adminAuthMiddleware, async (req, res) => {
    try {
        // gets all users
        const fetchedUsers = await User.find({})

        // returns all users, without the current user
        const users = fetchedUsers.filter( (user) => user._id.toString() !== req.user._

        // send all users
        res.send({ users })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

// get an user
router.get("/:id", adminAuthMiddleware, async (req, res) => {
    // stores the id parameter
    const _id = req.params.id
    try {
        const user = await User.findById(_id)

        res.send({ user })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

Any user can be updated by the administrator but the “_id”, “admin”, “typeSubscription” and “email” are not allowed to be changed because they are meant to be only changed by the user or not changed at all. The function will loop through the array of proposed updates and if will only assign changes only if the changes are allowed. The user is then asynchronously save and sent back to the client.

Any user account, subscriber or a admin, can be deleted by an administrator, using the delete route. Such account will no longer exist and the user wont be able to logging in again.

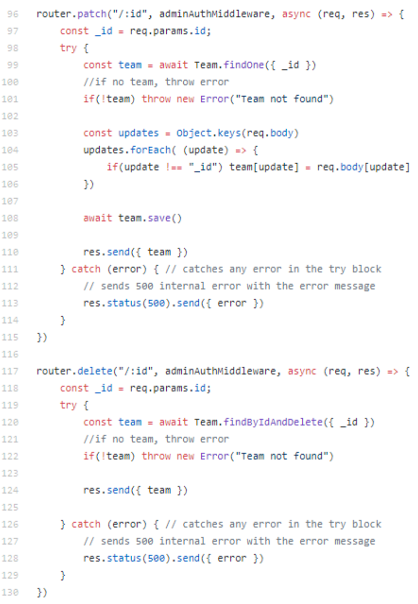


```
// update an user
router.patch("/:id", adminAuthMiddleware, async (req, res) => {
    // stores the id parameter and the body request
    const _id = req.params.id
    const body = req.body;
    try {
        // gets the user by id
        const user = await User.findById(_id)

        // updates all user allowed fields
        const updates = Object.keys(body);
        const notAllowed = [ "_id", "admin", "typeSubscription", "email" ]
        updates.forEach( update => {
            if(!notAllowed.includes(update)) user[update] = body[update]
        })

        // user changes saved
        await user.save()

        // send updated user
        res.send({ user })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

// delete an user
router.delete("/:id", adminAuthMiddleware, async (req, res) => {
    // stores the id parameter
    const _id = req.params.id
    try {
        // finds the user by id and deletes it
        const user = await User.findByIdAndDelete(_id)

        // sends deleted user
        res.send({ user })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

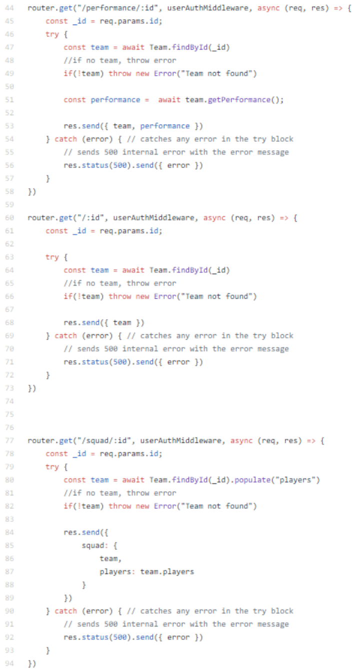
The other router existent in the application manage the matches, teams, players, match events, news and events. In these routers, all the router which get one or more instances of the respective models are available to all users but all the routes that create, update or delete instances of the same models are only available to administrator user accounts.

This router has routes to get all teams, one team, the league table, performance per team and squad (team and its players). All these routes use the get http verb, so for security they have the user authentication middleware to all users to use it.

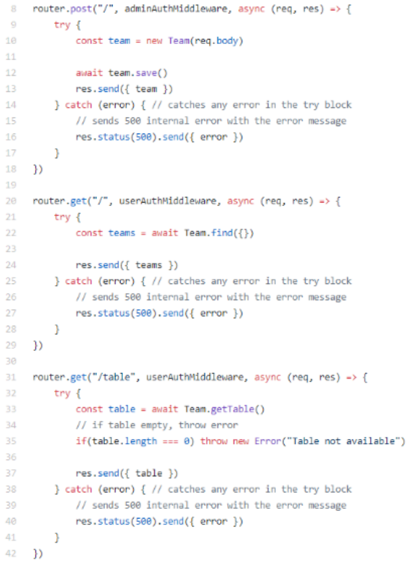
The remaining routes have the admin authentication middleware, to disable no admin users from create, update or delete teams.



```
router.patch("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const team = await Team.findOne({ _id })
        //if no team, throw error
        if(!team) throw new Error("Team not found")

        const updates = Object.keys(req.body)
        updates.forEach( (update) => {
            if(update !== "_id") team[update] = req.body[update]
        })

        await team.save()

        res.send({ team })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.delete("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const team = await Team.findByIdAndDelete({ _id })
        //if no team, throw error
        if(!team) throw new Error("Team not found")

        res.send({ team })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```



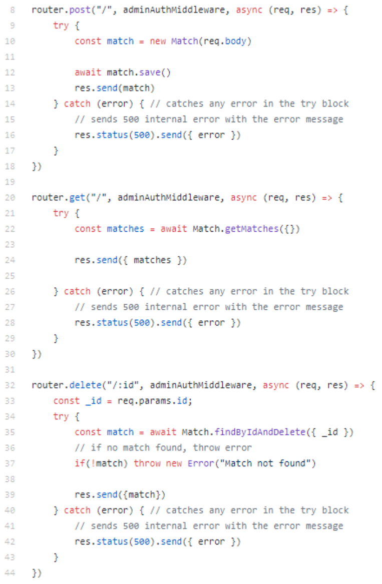
```
router.get("/performance/:id", userAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const team = await Team.findById(_id)
        //if no team, throw error
        if(!team) throw new Error("Team not found")

        const performance =  await team.getPerformance();

        res.send({ team, performance })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/:id", userAuthMiddleware, async (req, res) => {
    const _id = req.params.id;

    try {
        const team = await Team.findById(_id)
        //if no team, throw error
        if(!team) throw new Error("Team not found")

        res.send({ team })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})



router.get("/squad/:id", userAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const team = await Team.findById(_id).populate("players")
        //if no team, throw error
        if(!team) throw new Error("Team not found")

        res.send({
            squad: {
                team,
                players: team.players
            }
        })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```



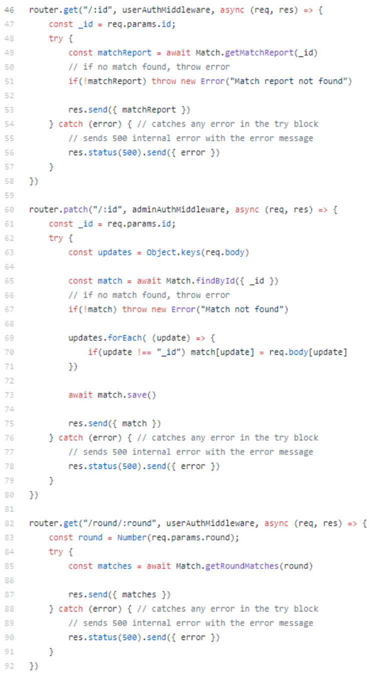
```
router.post("/", adminAuthMiddleware, async (req, res) => {
    try {
        const team = new Team(req.body)

        await team.save()
        res.send({ team })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/", userAuthMiddleware, async (req, res) => {
    try {
        const teams = await Team.find({})

        res.send({ teams })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/table", userAuthMiddleware, async (req, res) => {
    try {
        const table = await Team.getTable()
        // if table empty, throw error
        if(table.length === 0) throw new Error("Table not available")

        res.send({ table })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

In the match router get the matches per round and get a match report of a specific match.

An Administrator can create, delete and update matches. Additionally, the admin also has exclusive access to the get all matches because this functionality is not relevant, as users are only interested in the matches per round.



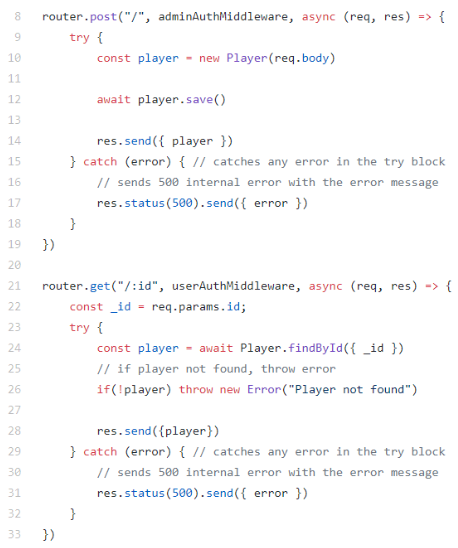
```
router.post("/", adminAuthMiddleware, async (req, res) => {
    try {
        const match = new Match(req.body)

        await match.save()
        res.send(match)
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/", adminAuthMiddleware, async (req, res) => {
    try {
        const matches = await Match.getMatches({})

        res.send({ matches })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.delete("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const match = await Match.findByIdAndDelete({ _id })
        // if no match found, throw error
        if(!match) throw new Error("Match not found")

        res.send({match})
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```



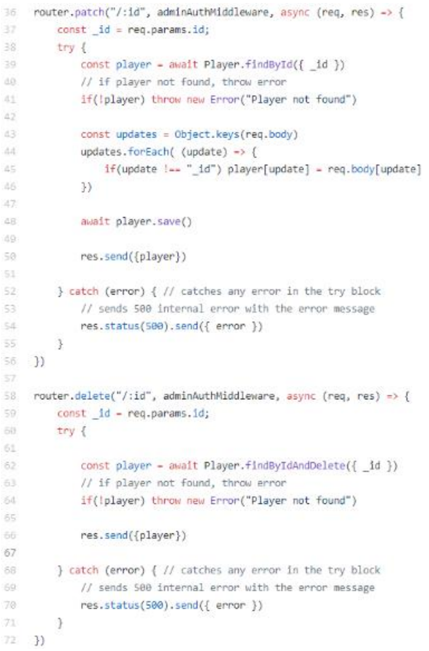
```
router.get("/:id", userAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const matchReport = await Match.getMatchReport(_id)
        // if no match found, throw error
        if(!matchReport) throw new Error("Match report not found")

        res.send({ matchReport })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.patch("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const updates = Object.keys(req.body)

        const match = await Match.findById({ _id })
        // if no match found, throw error
        if(!match) throw new Error("Match not found")

        updates.forEach( (update) => {
            if(update !== "_id") match[update] = req.body[update]
        })

        await match.save()

        res.send({ match })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/round/:round", userAuthMiddleware, async (req, res) => {
    const round = Number(req.params.round);
    try {
        const matches = await Match.getRoundMatches(round)

        res.send({ matches })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

The player router allows all the users to get a player individually. It is not necessary to get all players or players of a team because the team router already has a get squad that get all the team's players.

The administrator can create, update and delete players.



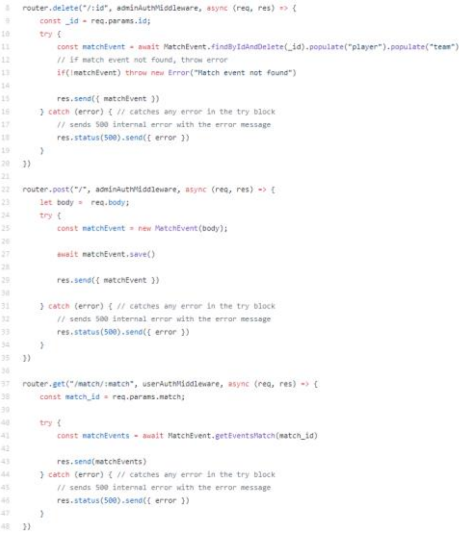
```
router.post("/", adminAuthMiddleware, async (req, res) => {
    try {
        const player = new Player(req.body)

        await player.save()

        res.send({ player })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/:id", userAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const player = await Player.findById({ _id })
        // if player not found, throw error
        if(!player) throw new Error("Player not found")

        res.send({player})
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```



```
router.patch("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const player = await Player.findById({ _id })
        // if player not found, throw error
        if(!player) throw new Error("Player not found")

        const updates = Object.keys(req.body)
        updates.forEach( (update) => {
            if(update !== "_id") player[update] = req.body[update]
        })

        await player.save()

        res.send({player})

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.delete("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {

        const player = await Player.findByIdAndDelete({ _id })
        // if player not found, throw error
        if(!player) throw new Error("Player not found")

        res.send({player})

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

All the users can get all match events of a match.

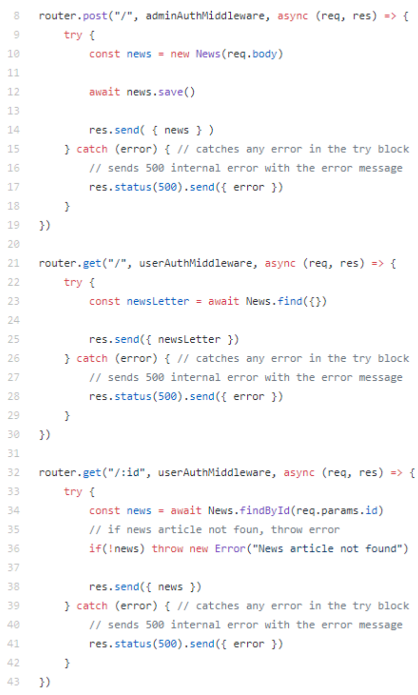
The administrator can create and delete match events. It is a simple router as the match event is a very simple model, event though it is extremely important for the whole system.



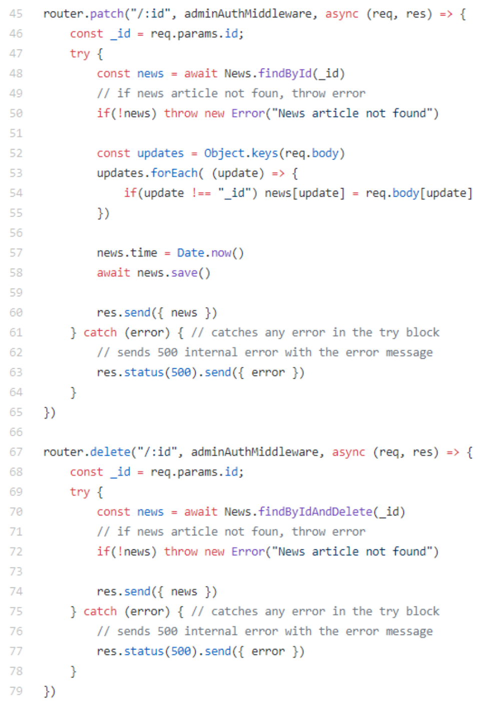
```
router.delete("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const matchEvent = await MatchEvent.findByIdAndDelete(_id).populate("player").populate("team")
        // if match event not found, throw error
        if(!matchEvent) throw new Error("Match event not found")

        res.send({ matchEvent })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.post("/", adminAuthMiddleware, async (req, res) => {
    let body =  req.body;
    try {
        const matchEvent = new MatchEvent(body);

        await matchEvent.save()

        res.send({ matchEvent })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/match/:match", userAuthMiddleware, async (req, res) => {
    const match_id = req.params.match;

    try {
        const matchEvents = await MatchEvent.getEventsMatch(match_id)

        res.send(matchEvents)
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

All users can get all news or individual news articles

The administrator can create, update and delete news articles.



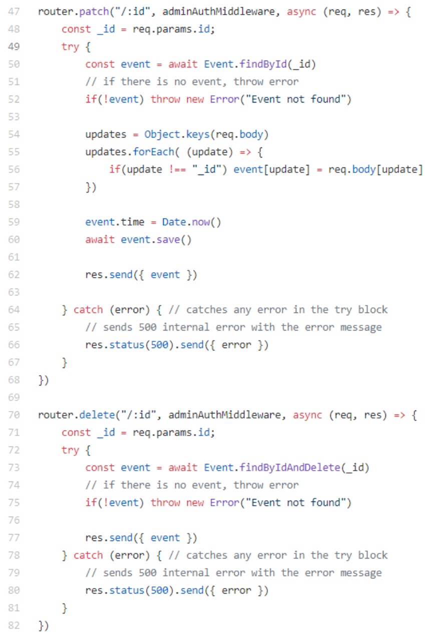
```
router.post("/", adminAuthMiddleware, async (req, res) => {
    try {
        const news = new News(req.body)

        await news.save()

        res.send( { news } )
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/", userAuthMiddleware, async (req, res) => {
    try {
        const newsLetter = await News.find({})

        res.send({ newsLetter })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/:id", userAuthMiddleware, async (req, res) => {
    try {
        const news = await News.findById(req.params.id)
        // if news article not foun, throw error
        if(!news) throw new Error("News article not found")

        res.send({ news })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```



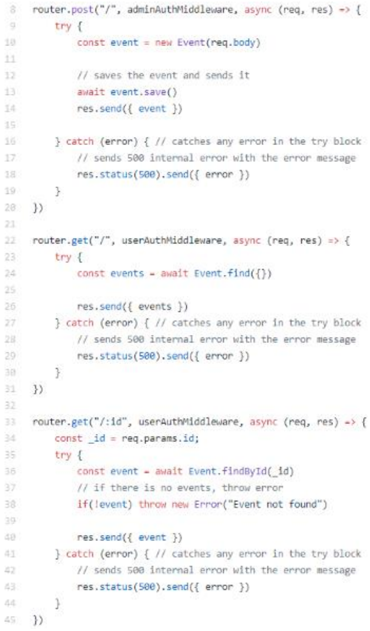
```
router.patch("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const news = await News.findById(_id)
        // if news article not foun, throw error
        if(!news) throw new Error("News article not found")

        const updates = Object.keys(req.body)
        updates.forEach( (update) => {
            if(update !== "_id") news[update] = req.body[update]
        })

        news.time = Date.now()
        await news.save()

        res.send({ news })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.delete("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const news = await News.findByIdAndDelete(_id)
        // if news article not foun, throw error
        if(!news) throw new Error("News article not found")

        res.send({ news })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

All user can get all events or get an individual event.

The administrator user can create, update and delete events.



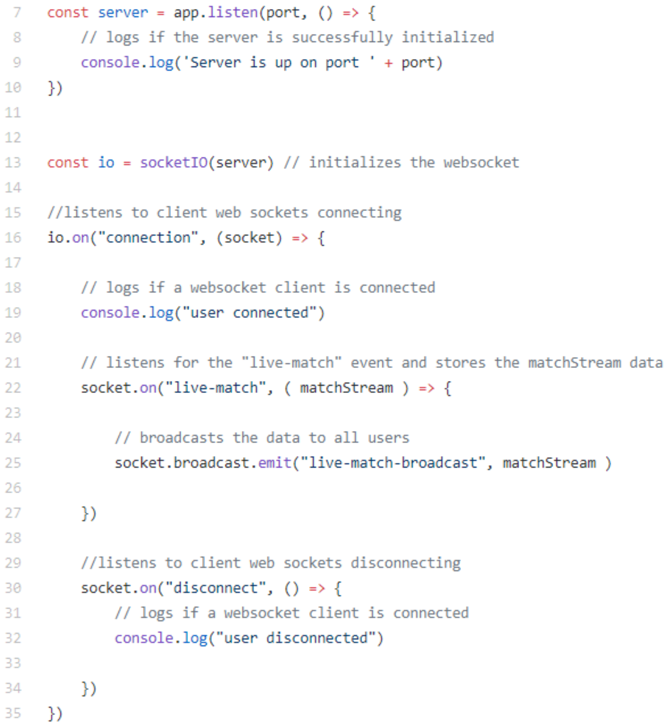
```
router.patch("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const event = await Event.findById(_id)
        // if there is no event, throw error
        if(!event) throw new Error("Event not found")

        updates = Object.keys(req.body)
        updates.forEach( (update) => {
            if(update !== "_id") event[update] = req.body[update]
        })

        event.time = Date.now()
        await event.save()

        res.send({ event })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.delete("/:id", adminAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const event = await Event.findByIdAndDelete(_id)
        // if there is no event, throw error
        if(!event) throw new Error("Event not found")

        res.send({ event })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```



```
router.post("/", adminAuthMiddleware, async (req, res) => {
    try {
        const event = new Event(req.body)

        // saves the event and sends it
        await event.save()
        res.send({ event })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/", userAuthMiddleware, async (req, res) => {
    try {
        const events = await Event.find({})

        res.send({ events })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})

router.get("/:id", userAuthMiddleware, async (req, res) => {
    const _id = req.params.id;
    try {
        const event = await Event.findById(_id)
        // if there is no events, throw error
        if(!event) throw new Error("Event not found")

        res.send({ event })
    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

The real-time data is secured by using socketio that takes the server variable and listens for connections by listening to the “connection” event. When a node connects to this web socket it will listen for the “disconnect” and “live-match” events.

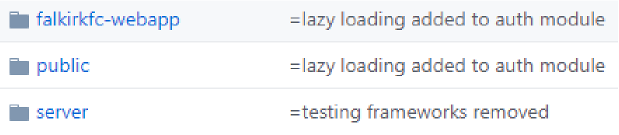
When the “live-match” event is triggered the server web socket will take the “matchStream” data and it will broadcast the data to all connected socketio clients node, apart from the socket that originally emitted the matchStream.



```
const server = app.listen(port, () => {
    // logs if the server is successfully initialized
    console.log('Server is up on port ' + port)
})


const io = socketIO(server) // initializes the websocket

//listens to client web sockets connecting
io.on("connection", (socket) => {

    // logs if a websocket client is connected
    console.log("user connected")

    // listens for the "live-match" event and stores the matchStream data
    socket.on("live-match", ( matchStream ) => {

        // broadcasts the data to all users
        socket.broadcast.emit("live-match-broadcast", matchStream )

    })

    //listens to client web sockets disconnecting
    socket.on("disconnect", () => {
        // logs if a websocket client is connected
        console.log("user disconnected")

    })
})
```

View model code listings

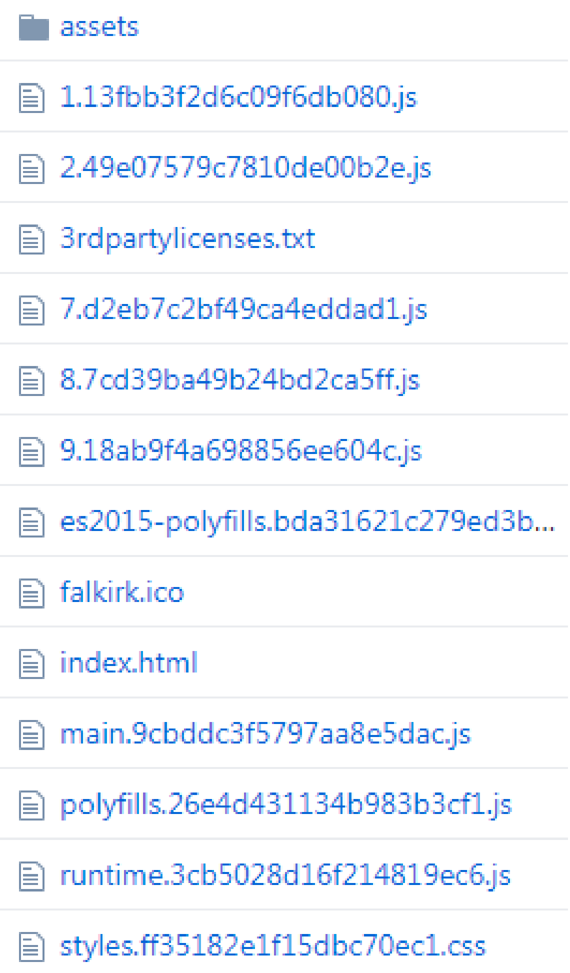
The route of the project has both the Single Page Application, the server application and the public folder.



falkirkfc-webapp	=lazy loading added to auth module
public	=lazy loading added to auth module
server	=testing frameworks removed

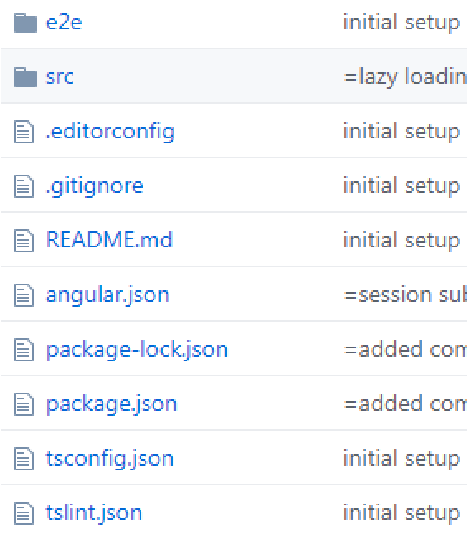
The public folder is the one resource available to the users, containing all the angular javascript compiled bundles, the index html file together with other assets such as images, logos and icons. These files have their names hashed for security reasons and the main javascript bundle is the one which is the first that renders the application inside the index html template file. The main other bundles which are numbered are the other application models that are lazy loaded, namely the auth, the admin and the subscriber modules.

The Lazy Loading is a software design pattern used to increase the performance of many types of applications. In Angular applications the lazy loading is used to lower the application's loading time over the internet by only downloading the absolute necessary for immediate use, in this case the main bundle and the css file, as well as the cross-browser modules such as the EcmaScript2015, polyfills and runtime files. The css file is also the result of the compilation of the sass files from the angular custom theme and from custom css modifications. The main bundle has the all the core angular modules and main configuration as well as the landing page. The index html only has the header tags as its body will be rendered by the angular bundles. The assets page contains all the images and logos, apart from the Falkirk tab icon which is located in the public directory.

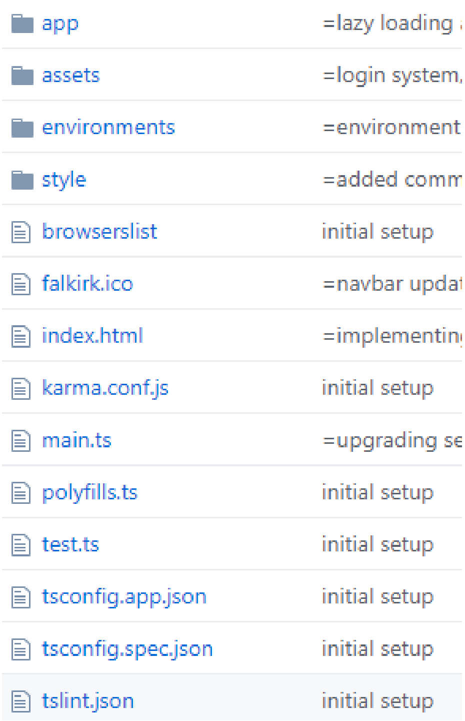


assets
1.13fbb3f2d6c09f6db080.js
2.49e07579c7810de00b2e.js
3rdpartylicenses.txt
7.d2eb7c2bf49ca4eddad1.js
8.7cd39ba49b24bd2ca5ff.js
9.18ab9f4a698856ee604c.js
es2015-polyfills.bda31621c279ed3b...
falkirk.ico
index.html
main.9cbddc3f5797aa8e5dac.js
polyfills.26e4d431134b983b3cf1.js
runtime.3cb5028d16f214819ec6.js
styles.ff35182e1f15dbc70ec1.css

The falkirk-webapp contains mostly native angular configuration which allows production and development of the application located in the src folder.



e2e	initial setup
src	=lazy loadin
.editorconfig	initial setup
.gitignore	initial setup
README.md	initial setup
angular.json	=session sul
package-lock.json	=added con
package.json	=added con
tsconfig.json	initial setup
tslint.json	initial setup

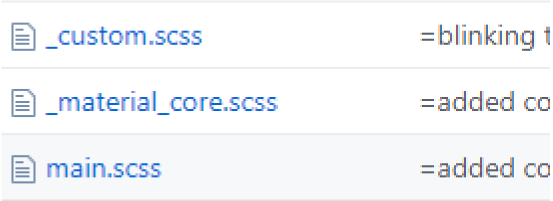


app	=lazy loading
assets	=login system,
environments	=environment
style	=added comm
browserslist	initial setup
falkirk.ico	=navbar updat
index.html	=implementin
karma.conf.js	initial setup
main.ts	=upgrading se
polyfills.ts	initial setup
test.ts	initial setup
tsconfig.app.json	initial setup
tsconfig.spec.json	initial setup
tslint.json	initial setup

The src folder contains mostly angular configuration files that set up the development and production. The environment folder contains two files which set the SPA environment files to sets the base url when producing or developing (which is different from the server port, because angular development local server runs on a different port). The same assets is exactly the same as the one in the public folder, as well as the Falkirk icon. The styles folder is a custom folder where it contains the sass files, which imports the angular material sass files from the node modules.

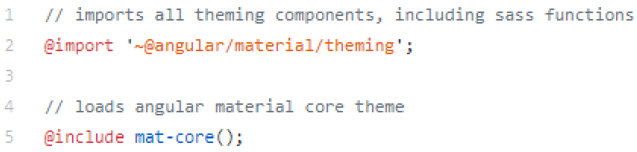
The app folder is where the all the SPA application code is going to be compiled from.

The content in the styles has all the sass files which import angular material and sets the custom theme, as well as some custom css. The main sass files will be compiled into the public css files.



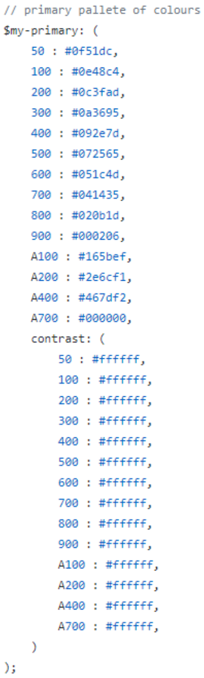
_custom.scss	=blinking t
_material_core.scss	=added co
main.scss	=added co

The material_core file will import the theming module from the angular material library. The include function will instantiate the material core components.



```
// imports all theming components, including sass functions
@import '~@angular/material/theming';

// loads angular material core theme
@include mat-core();
```

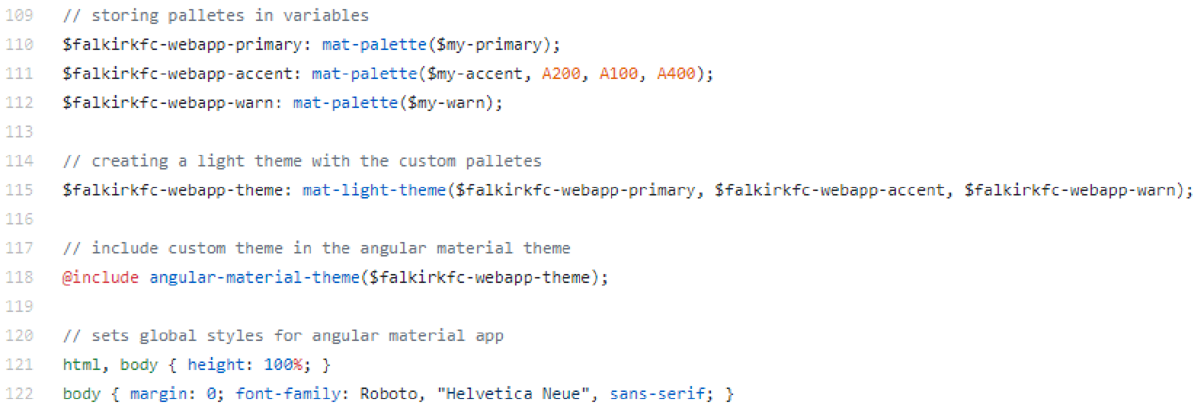
The angular material theming is based on pallets, instead of just basic colours. This makes the theme is more dynamic as the colour scheme is more complex. The Custom theme is based on three pallets, the primary, the accent and the warn pallets. The primary colour is based on the Falkirk main blue colour, the accent is based on its secondary yellow kit and the red colour is based on kits of other kits which also included a red colour. Every colour pallet needs to have different values, defined in a map, of the same colour based together with a nested map containing the respective contrast colours.



```
// primary pallete of colours
$my-primary: (
    50 : #0f51dc,
    100 : #0e48c4,
    200 : #0c3fad,
    300 : #0a3695,
    400 : #092e7d,
    500 : #072565,
    600 : #051c4d,
    700 : #041435,
    800 : #020b1d,
    900 : #000206,
    A100 : #165bef,
    A200 : #2e6cf1,
    A400 : #467df2,
    A700 : #000000,
    contrast: (
        50 : #ffffff,
        100 : #ffffff,
        200 : #ffffff,
        300 : #ffffff,
        400 : #ffffff,
        500 : #ffffff,
        600 : #ffffff,
        700 : #ffffff,
        800 : #ffffff,
        900 : #ffffff,
        A100 : #ffffff,
        A200 : #ffffff,
        A400 : #ffffff,
        A700 : #ffffff,
    )
);
```

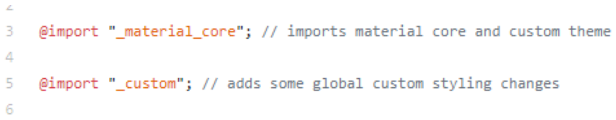
After defining the primary, accent and war pallet maps, the same ones are stored in the variables using the mat-pallete function, an angular material method, which takes a map and returns pallet.

The intended theme was meant to be the light angular material theme. The Angular material mat-light-theme takes the pallets and returns the Falkirk custom application theme. This variable will be then finally instantiated by using the angular-material-theme that takes the custom theme and includes it in the sass content to be loaded in the public's folder final css bundle.



```scss
// storing palletes in variables
$falkirkfc-webapp-primary: mat-palette($my-primary);
$falkirkfc-webapp-accent: mat-palette($my-accent, A200, A100, A400);
$falkirkfc-webapp-warn: mat-palette($my-warn);

// creating a light theme with the custom palletes
$falkirkfc-webapp-theme: mat-light-theme($falkirkfc-webapp-primary, $falkirkfc-webapp-accent, $falkirkfc-webapp-warn);

// include custom theme in the angular material theme
@include angular-material-theme($falkirkfc-webapp-theme);

// sets global styles for angular material app
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }
```

The main scss file will import the material custom theme and the content from the custom file, which only includes a few minor changes.

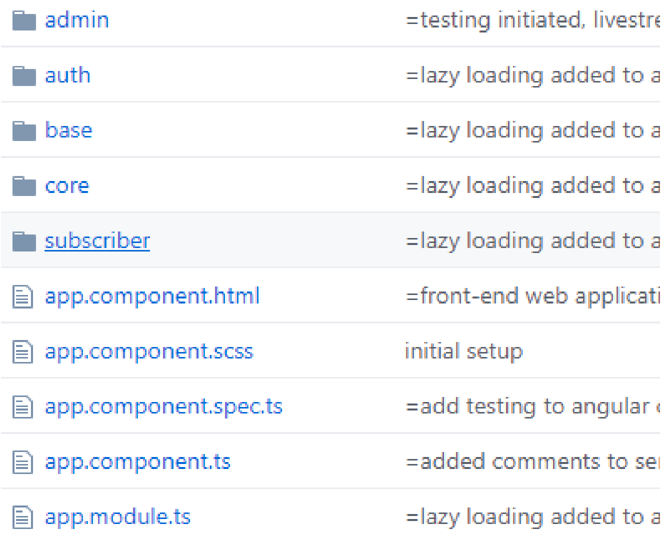


```scss
@import "_material_core"; // imports material core and custom theme

@import "_custom"; // adds some global custom styling changes
```

The main view model is inside the app model, where it is possible to see the many modules' folders alongside the app module file as well as the app component's html, scss, spec and ts files.

The app module is the main module that bootstraps the app component together with the other modules. The app component is the container that will the whole view model.

The app component, as any other component is composed of the typescript file that operates the logic of the component and html file as well. The scss file, that gives the style to the component's html file, is also compiled in the public's style css bundle but its styles will only have a local effect in the component file, without affect event the children components. The spec file is used to run the angular's unit testing.



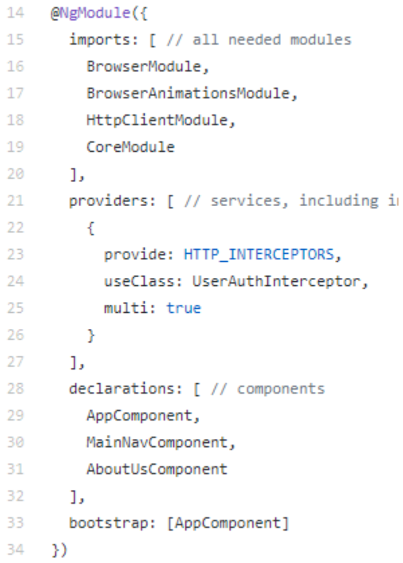
admin	=testing initiated, livestre
auth	=lazy loading added to a
base	=lazy loading added to a
core	=lazy loading added to a
subscriber	=lazy loading added to a
app.component.html	=front-end web applicati
app.component.scss	initial setup
app.component.spec.ts	=add testing to angular
app.component.ts	=added comments to se
app.module.ts	=lazy loading added to a

The app module is the most important module in an angular application and it will load all the modules necessary to be load immediately. This module does not load the other modules that are lazily loaded. The module uses a NgModule decorator (decorator is a design pattern) and it imports the browser module (necessary to interact with the browser features), browser animations module (necessary for angular material module), http client module (necessary to do http requests to the server API) and the core module which is a custom module to centralize some of the code as well.

The providers field imports all the services. In this case an http interceptor is set with special configuration, because it will intercept every http request the application will do, even in the other modules. The services that are instantiated in the app module are global to all modules and they behave as singletons (a design pattern) which ensure each service is only instantiated once, avoiding bugs and errors

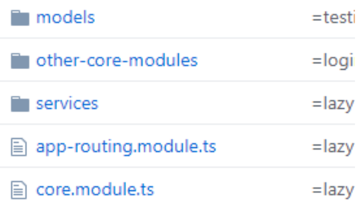
The declaration array imports all the components which the module loads. The app module loads the app, the main nav and the about us components.

The app module file instantiate the Falkirk web app by bootstrapping the app component, file which will contain the view model.



```
@NgModule({
  imports: [ // all needed modules
    BrowserModule,
    BrowserAnimationsModule,
    HttpClientModule,
    CoreModule
  ],
  providers: [ // services, including i
    {
      provide: HTTP_INTERCEPTORS,
      useClass: UserAuthInterceptor,
      multi: true
    }
  ],
  declarations: [ // components
    AppComponent,
    MainNavComponent,
    AboutUsComponent
  ],
  bootstrap: [AppComponent]
})
```

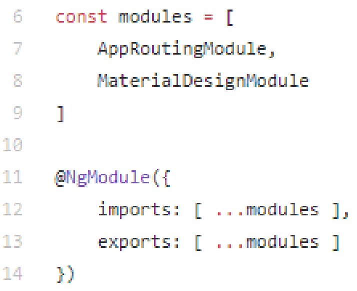
The core folder has the core module file, the app routing module, the service’s folder, the model’s folder, and other modules folders to be imported.



```
models                  =testi
other-core-modules      =logi
services                =lazy
app-routing.module.ts   =lazy
core.module.ts          =lazy
```

The core module is just a wrapper module that imports and exports the app routing and the material design modules. The wrapper module’s intent, as this core module, is to reduce the amount of code in the app module.

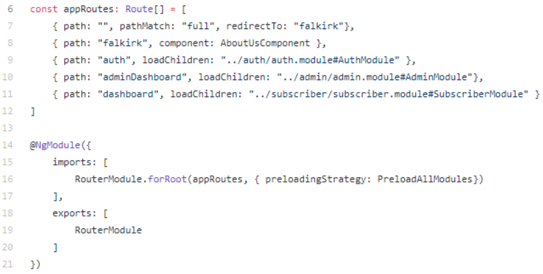
The array of modules is pushed in the imports and exports fields using the new JavaScript features. This feature allow to reduce the code if any individual module is also added.



```
const modules = [
    AppRoutingModule,
    MaterialDesignModule
]

@NgModule({
    imports: [ ...modules ],
    exports: [ ...modules ]
})
```

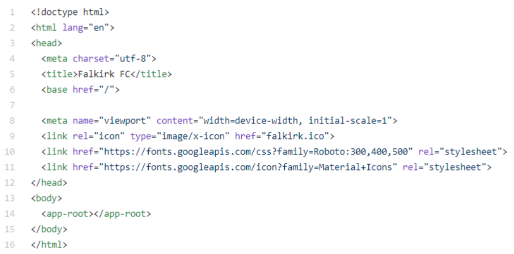
The app routing module declares an array of Routes to define the root router.

The path “/” redirects the router to the “/falkirk” route which renders the About Us Component. The “auth” path is the root path for the lazy loaded Auth module by loading its children. The “adminDashboard” path is the root path for the lazy loaded Admin module by loading its children. The “dashboard” path is the root path for the lazy loaded Subscriber module.



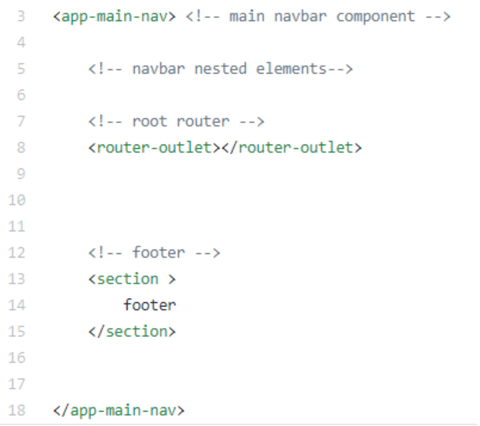
```
const appRoutes: Route[] = [
    { path: "", pathMatch: "full", redirectTo: "falkirk"},
    { path: "falkirk", component: AboutUsComponent },
    { path: "auth", loadChildren: "../auth/auth.module#AuthModule" },
    { path: "adminDashboard", loadChildren: "../admin/admin.module#AdminModule"},
    { path: "dashboard", loadChildren: "../subscriber/subscriber.module#SubscriberModule" }
]

@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes, { preloadingStrategy: PreloadAllModules})
    ],
    exports: [
        RouterModule
    ]
})
```

The index html page, located in the “src” folder, is the single page where the Single Page Application works. It contains meta data information in the head section and it only contains the “app-root” tag. This tag is where the app component is rendered, therefore where all the Single Page Application is rendered at. This is the same html file which is sent by the server to the client to display the view model and the html page that communicates with the server using asynchronous programming, without requesting any other html page.



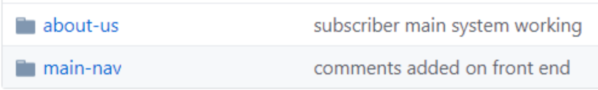
```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Falkirk FC</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="falkirk.ico">
  <link href="https://fonts.googleapis.com/css?family=Roboto:300,400,500" rel="stylesheet">
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The app component, html template, has a “router-outlet” tag and a footer section nested inside the “app-main-nav” component. The “router-outlet” tag is where the “app-routing module” will render the child components, according with the browser’s route path. The SPA routers avoid the page by using the router module route requests will determine which component the “router-outlet” must render. This specific “router-outlet” renders components from all the three lazy loaded modules.



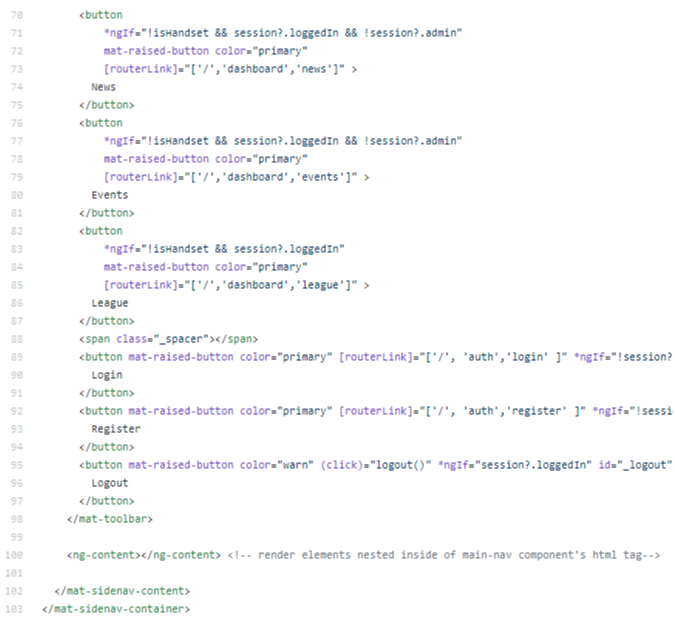
```html
<app-main-nav> <!-- main navbar component -->

    <!-- navbar nested elements-->

    <!-- root router -->
    <router-outlet></router-outlet>



    <!-- footer -->
    <section >
        footer
    </section>


</app-main-nav>
```

The base folder contains the about us and the main-nav components. The about us is the landing page that shows some general information about the club.

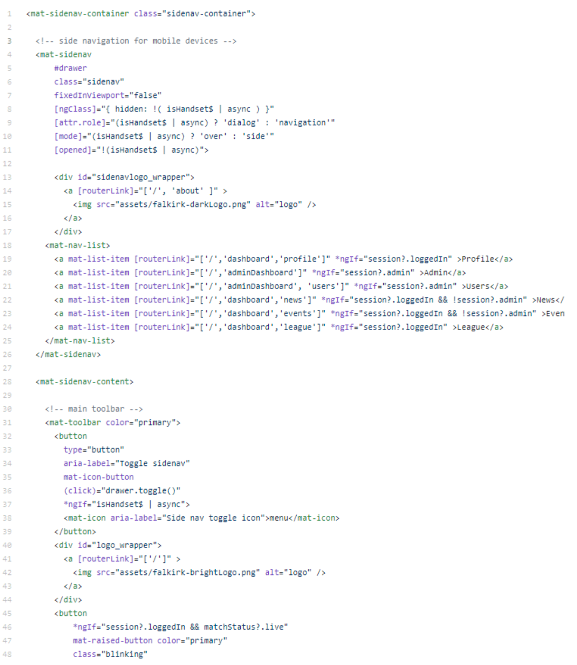


about-us	subscriber main system working
main-nav	comments added on front end

The main-nav html template contains the “angular material” navbar component. It is composed by a “sidenav”and a toolbar “angular material components”. The side nav will appear if the client view has the dimensions of a mobile device, otherwise it on appear on other screen sizes. When the client uses a mobile device he toolbar links will not be displayed, and the “sidenav” equivalent links will appear to allow navigation to mobile devices. Both templates have logic that determines how the “main-nav” will behave if many different scenarios. The “ng-content” tag is the tag that lets the angular application render other components nested inside of the “main-nav” tag.



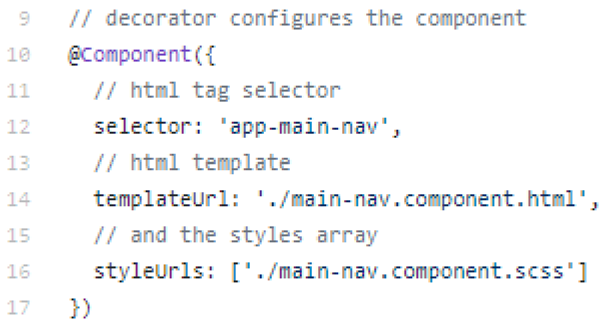
```html
      <button
          *ngIf="!isHandset && session?.loggedIn && !session?.admin"
          mat-raised-button color="primary"
          [routerLink]="['/','dashboard','news']" >
        News
      </button>
      <button
          *ngIf="!isHandset && session?.loggedIn && !session?.admin"
          mat-raised-button color="primary"
          [routerLink]="['/','dashboard','events']" >
        Events
      </button>
      <button
          *ngIf="!isHandset && session?.loggedIn"
          mat-raised-button color="primary"
          [routerLink]="['/','dashboard','league']" >
        League
      </button>
      <span class="_spacer"></span>
      <button mat-raised-button color="primary" [routerLink]="['/', 'auth','login' ]" *ngIf="!session?
        Login
      </button>
      <button mat-raised-button color="primary" [routerLink]="['/', 'auth','register' ]" *ngIf="!sessi
        Register
      </button>
      <button mat-raised-button color="warn" (click)="logout()" *ngIf="session?.loggedIn" id="_logout"
        Logout
      </button>
    </mat-toolbar>

    <ng-content></ng-content> <!-- render elements nested inside of main-nav component's html tag-->

  </mat-sidenav-content>
</mat-sidenav-container>
```



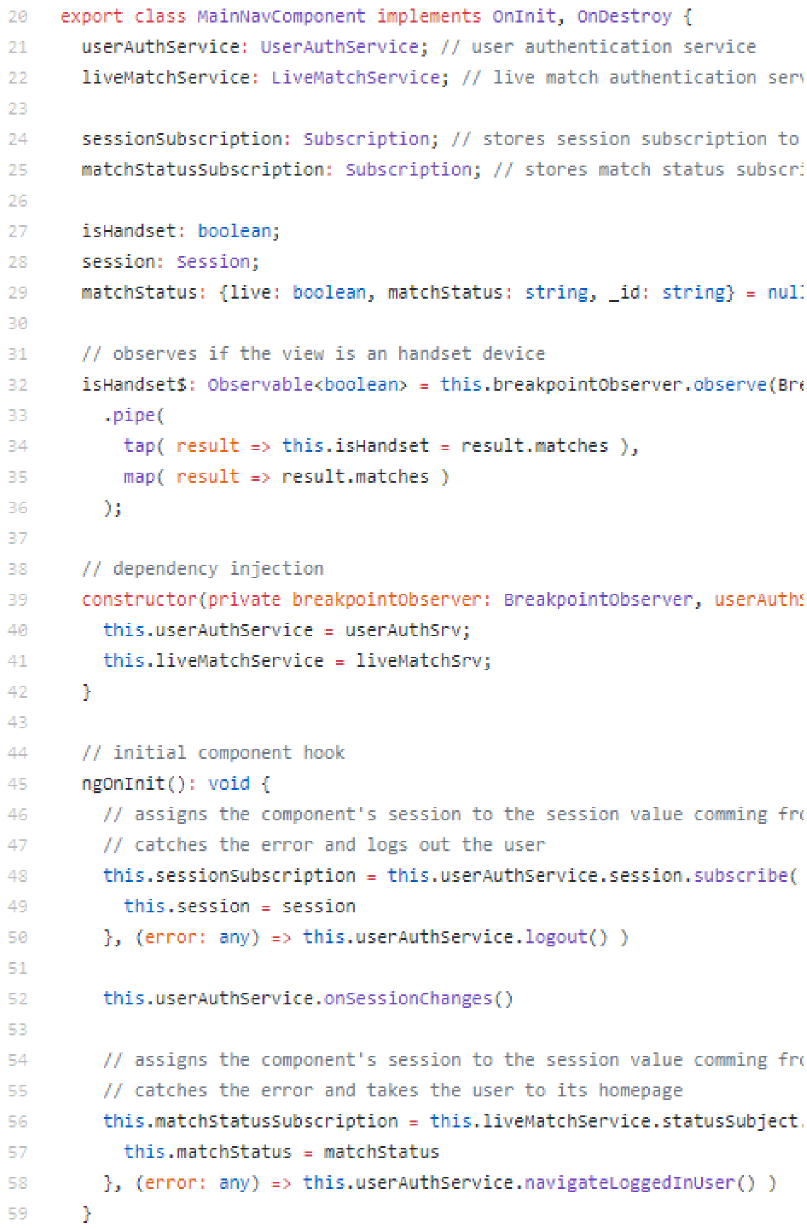
```html
<mat-sidenav-container class="sidenav-container">

  <!-- side navigation for mobile devices -->
  <mat-sidenav
      #drawer
      class="sidenav"
      fixedInViewport="false"
      [ngClass]="{ hidden: !( isHandset$ | async ) }"
      [attr.role]="(isHandset$ | async) ? 'dialog' : 'navigation'"
      [mode]="(isHandset$ | async) ? 'over' : 'side'"
      [opened]="!(isHandset$ | async)">

      <div id="sidenavlogo_wrapper">
        <a [routerLink]="['/', 'about' ]" >
          <img src="assets/falkirk-darkLogo.png" alt="logo" />
        </a>
      </div>
    <mat-nav-list>
      <a mat-list-item [routerLink]="['/','dashboard','profile']" *ngIf="session?.loggedIn" >Profile</a>
      <a mat-list-item [routerLink]="['/','adminDashboard']" *ngIf="session?.admin" >Admin</a>
      <a mat-list-item [routerLink]="['/','adminDashboard', 'users']" *ngIf="session?.admin" >Users</a>
      <a mat-list-item [routerLink]="['/','dashboard','news']" *ngIf="session?.loggedIn && !session?.admin" >News</
      <a mat-list-item [routerLink]="['/','dashboard','events']" *ngIf="session?.loggedIn && !session?.admin" >Even
      <a mat-list-item [routerLink]="['/','dashboard','league']" *ngIf="session?.loggedIn" >League</a>
    </mat-nav-list>
  </mat-sidenav>

  <mat-sidenav-content>

    <!-- main toolbar -->
    <mat-toolbar color="primary">
      <button
        type="button"
        aria-label="Toggle sidenav"
        mat-icon-button
        (click)="drawer.toggle()"
        *ngIf="isHandset$ | async">
        <mat-icon aria-label="Side nav toggle icon">menu</mat-icon>
      </button>
      <div id="logo_wrapper">
        <a [routerLink]="['/']" >
          <img src="assets/falkirk-brightLogo.png" alt="logo" />
        </a>
      </div>
      <button
          *ngIf="session?.loggedIn && matchStatus?.live"
          mat-raised-button color="primary"
          class="blinking"
```

The component’s template logic refers to the component’s typescript file. This file instantiates the component and controls the html template content.

The “main-nav” component, as any other angular component has a component decorator (a programming design pattern) that declare the html selector, the html template and its stylesheet files. Following the component’s decorator, the class is declared and exported to be available to its feature module. Inside the class, the all the variables, including any primitive type, services, dependencies and so on, are declared so that the template can have access to them. All the dependencies, such as services, must be injected in the component vie the constructor to avoid the creation of more than one instance, hence the fact that services are designed to behave as the design pattern’s singleton.



```
// decorator configures the component
@Component({
  // html tag selector
  selector: 'app-main-nav',
  // html template
  templateUrl: './main-nav.component.html',
  // and the styles array
  styleUrls: ['./main-nav.component.scss']
})
```



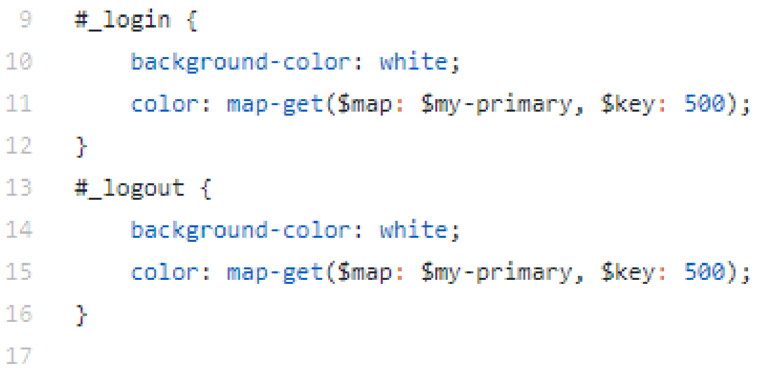
```
export class MainNavComponent implements OnInit, OnDestroy {
  userAuthService: UserAuthService; // user authentication service
  liveMatchService: LiveMatchService; // live match authentication ser

  sessionSubscription: Subscription; // stores session subscription to
  matchStatusSubscription: Subscription; // stores match status subscr

  isHandset: boolean;
  session: Session;
  matchStatus: {live: boolean, matchStatus: string, _id: string} = nul

  // observes if the view is an handset device
  isHandset$: Observable<boolean> = this.breakpointObserver.observe(Br
    .pipe(
      tap( result => this.isHandset = result.matches ),
      map( result => result.matches )
    );

  // dependency injection
  constructor(private breakpointObserver: BreakpointObserver, userAuth
    this.userAuthService = userAuthSrv;
    this.liveMatchService = liveMatchSrv;
  }

  // initial component hook
  ngOnInit(): void {
    // assigns the component's session to the session value comming fr
    // catches the error and logs out the user
    this.sessionSubscription = this.userAuthService.session.subscribe(
      this.session = session
    }, (error: any) => this.userAuthService.logout() )

    this.userAuthService.onSessionChanges()

    // assigns the component's session to the session value comming fr
    // catches the error and takes the user to its homepage
    this.matchStatusSubscription = this.liveMatchService.statusSubject.
      this.matchStatus = matchStatus
    }, (error: any) => this.userAuthService.navigateLoggedInUser() )
  }
```

Advanced data structures

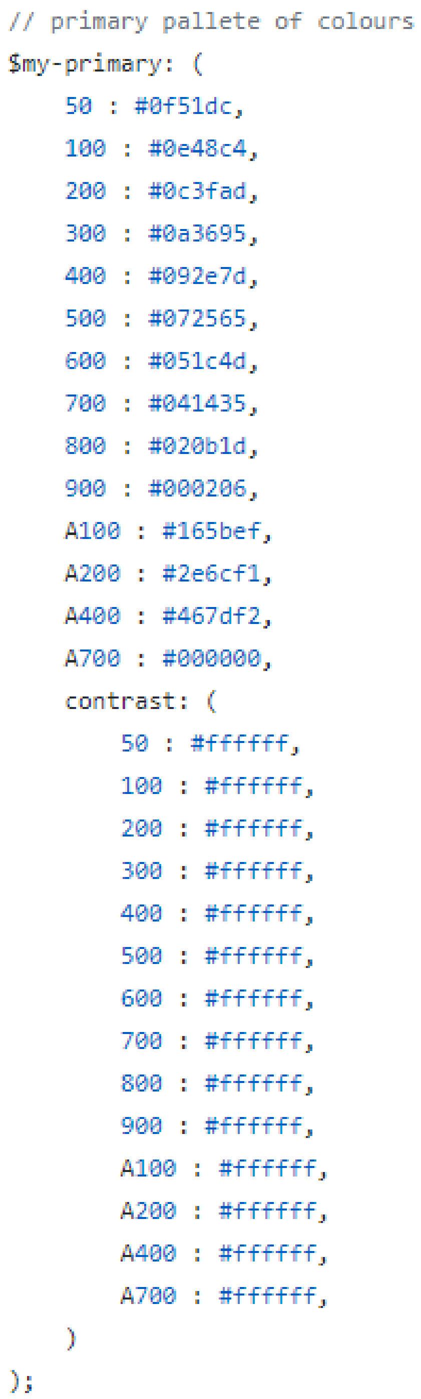
The advanced data structure used in the development was actually in the styling side of the Single Page Application. The Sass programming language was the map advanced data structure to store any kind of data.

In the mat-core sass file the map is used to declare a map, containing colour variables and a nested map (named contrast) containing correspondent contrast colours, which will be converted into a pallet using mat-pallet angular material function.

In the custom sass file in that same folder, the same map used to create the primary pallet is used again to modify and highlight the login and logout style. The colour for the two CSS classes is returned from the map, using the sass map-get native function.



```
#_login {
    background-color: white;
    color: map-get($map: $my-primary, $key: 500);
}
#_logout {
    background-color: white;
    color: map-get($map: $my-primary, $key: 500);
}

```



```
// primary pallete of colours
$my-primary: (
    50 : #0f51dc,
    100 : #0e48c4,
    200 : #0c3fad,
    300 : #0a3695,
    400 : #092e7d,
    500 : #072565,
    600 : #051c4d,
    700 : #041435,
    800 : #020b1d,
    900 : #000206,
    A100 : #165bef,
    A200 : #2e6cf1,
    A400 : #467df2,
    A700 : #000000,
    contrast: (
        50 : #ffffff,
        100 : #ffffff,
        200 : #ffffff,
        300 : #ffffff,
        400 : #ffffff,
        500 : #ffffff,
        600 : #ffffff,
        700 : #ffffff,
        800 : #ffffff,
        900 : #ffffff,
        A100 : #ffffff,
        A200 : #ffffff,
        A400 : #ffffff,
        A700 : #ffffff,
    )
);
```

Use of unfamiliar Libraries

Rxjs

This library is integrated in the angular framework to provide asynchronous coding. The library is essential to create real-time experiences through the use of observables. It is possible to create custom observables or use angular ones to subscribe to streams of data. The library was crucial for the development of the live score feature.

Git

The git repository system was essential to do version control and to push the Falkirk repository to GitHub and to the Heroku server.

SocketIO

This library is the JavaScript implementation of web socket which allow the admin to broadcast a live match stream to all users. This library has the server web socket and the client web socket incorporated together.

Angular Material

Angular material is a library maintained by angular that provides an custom angular theme built on sass, and an cli that creates angular components.

Bcryptjs

The Bcrytjs JavaScript is used to hash the password before being saved in the database. This is extre

Cors

This library lets the SPA and the rest API communicate in the same server by allowing Cross Origin Resource Shraring. Without this library the SPA would need to be installed in other server to use the rest API resources.

JSON Web Token

JSON web tokens has a JavaScript library that encrypts data using the HS256 algorithm. This algorithm encrypts data that can be only decrypt using a key string which was defined in the server, as an environment variable.

Mongoose

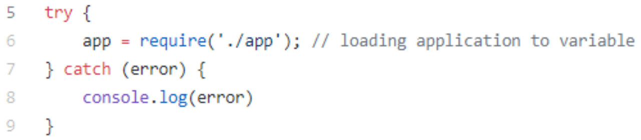
The mongoose library is used to interact with the MongoDB database API. The use of this library made the development more flexible by allowing the creation of complex data models that can hold information about a football club and its competitions.

Validator

The validator library is used to validate some of the fields located in the mongoose models. Without this library it would be difficult to validate a potential large volume of possibilities that need to be handled.

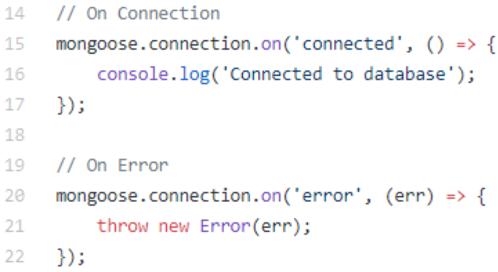
Error handling and Validation

Application error handling



```
try {
    app = require('./app'); // loading application to variable
} catch (error) {
    console.log(error)
}
```

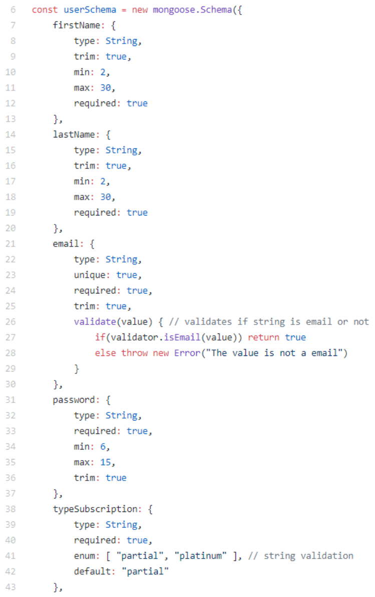
The server.js file has a try catch blocks to handle any error from loading the express application. This is specially important as there is many components that can fail. With that set up any uncaught error by any of the components will be caught in the server file.



```
// On Connection
mongoose.connection.on('connected', () => {
    console.log('Connected to database');
});

// On Error
mongoose.connection.on('error', (err) => {
    throw new Error(err);
});
```

For an example, the database which connects with a external cloud cluster throws an error if the connection is not established perfectly. This exception will be caught in the server file, when loading the express application.

Model validation



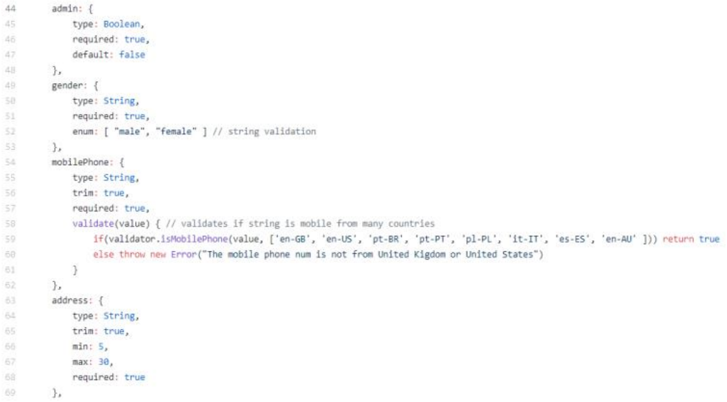
```
const userSchema = new mongoose.Schema({
    firstName: {
        type: String,
        trim: true,
        min: 2,
        max: 30,
        required: true
    },
    lastName: {
        type: String,
        trim: true,
        min: 2,
        max: 30,
        required: true
    },
    email: {
        type: String,
        unique: true,
        required: true,
        trim: true,
        validate(value) { // validates if string is email or not
            if(validator.isEmail(value)) return true
            else throw new Error("The value is not a email")
        }
    },
    password: {
        type: String,
        required: true,
        min: 6,
        max: 15,
        trim: true
    },
    typeSubscription: {
        type: String,
        required: true,
        enum: [ "partial", "platinum" ], // string validation
        default: "partial"
    },
```

Every mongoose model is validated before it is saved or updated in the database. In the case of user, all its fields are strings, apart from admin boolean field.

The type is a validating field as it will throw an error in the data does not match the specified type. All its string field are validated using “required” validating field, what means they need to be specified ate least otherwise mongoose will throw a validation error.

The “min” and “max” fields validate the length of the string and the “enum” validating field creates an enumerable string that will throw an error if the given string does not match one of the strings in the enumerable.

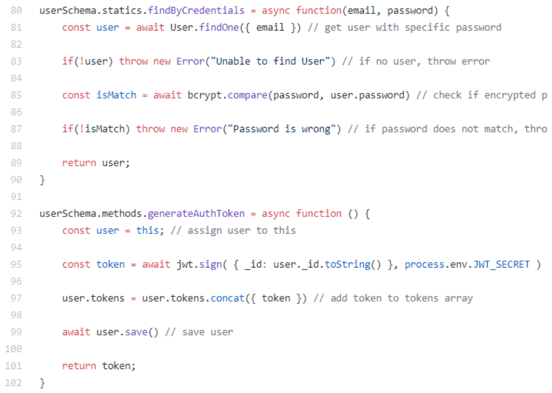
The validate function is a field that can be customized by the developer to either return true if the value (which already passed all built-in validations) is still valid or if a exception should be thrown. The “email” and “mobilePhone” are validated using the validator library which uses an algorithm to determine if phone numbers and email addresses are valid



```
    admin: {
        type: Boolean,
        required: true,
        default: false
    },
    gender: {
        type: String,
        required: true,
        enum: [ "male", "female" ] // string validation
    },
    mobilePhone: {
        type: String,
        trim: true,
        required: true,
        validate(value) { // validates if string is mobile from many countries
            if(validator.isMobilePhone(value, ['en-GB', 'en-US', 'pt-BR', 'pt-PT', 'pl-PL', 'it-IT', 'es-ES', 'en-AU' ])) return true
            else throw new Error("The mobile phone num is not from United Kigdom or United States")
        }
    },
    address: {
        type: String,
        trim: true,
        min: 5,
        max: 30,
        required: true
    },
```

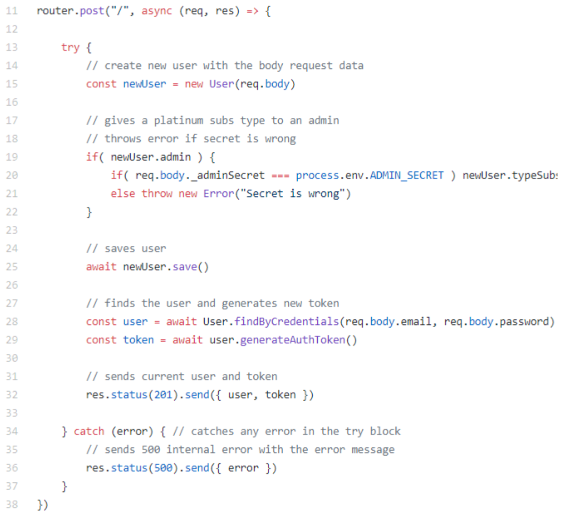
http router error handling

Furthermore, the user model has, apart from all the built-in validation and functions that throw exceptions, some custom functions as "findByCredentials" and "generateAuthToken" also throw exceptions not only due to the implicit throw statement but also due to the use of libraries that have their own exception throwing, such as the "jwt" and "bcrypt" ones.



```
userSchema.statics.findByCredentials = async function(email, password) {
    const user = await User.findOne({ email }) // get user with specific password

    if(!user) throw new Error("Unable to find User") // if no user, throw error

    const isMatch = await bcrypt.compare(password, user.password) // check if encrypted p

    if(!isMatch) throw new Error("Password is wrong") // if password does not match, thro

    return user;
}

userSchema.methods.generateAuthToken = async function () {
    const user = this; // assign user to this

    const token = await jwt.sign( { _id: user._id.toString() }, process.env.JWT_SECRET )

    user.tokens = user.tokens.concat({ token }) // add token to tokens array

    await user.save() // save user

    return token;
}
```

For these reasons all application's router handler functions for every resource run inside the try catch blocks that are going to catch any exception or error and safely return a status code 500 with the error message as a feedback. If this try catch blocks were not placed the server would crash and the client would not get any answer at all.

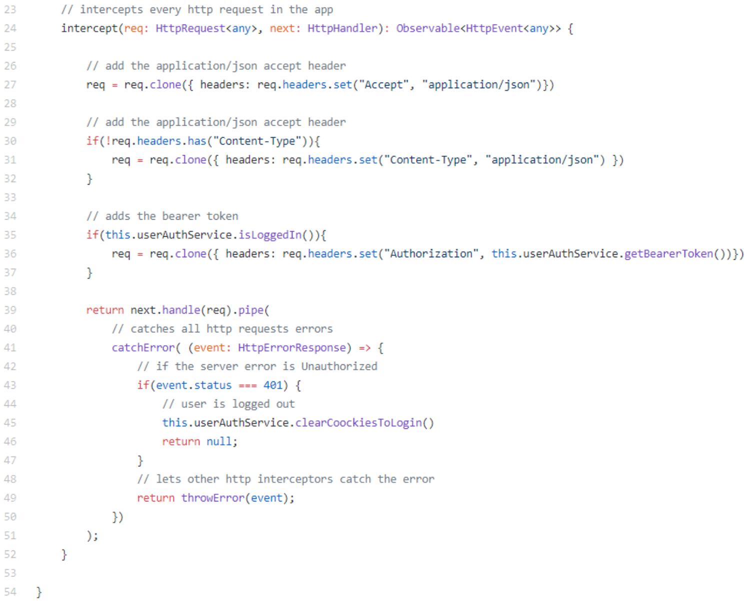


```
router.post("/", async (req, res) => {

    try {
        // create new user with the body request data
        const newUser = new User(req.body)

        // gives a platinum subs type to an admin
        // throws error if secret is wrong
        if( newUser.admin ) {
            if( req.body._adminSecret === process.env.ADMIN_SECRET ) newUser.typeSub:
            else throw new Error("Secret is wrong")
        }

        // saves user
        await newUser.save()

        // finds the user and generates new token
        const user = await User.findByCredentials(req.body.email, req.body.password)
        const token = await user.generateAuthToken()

        // sends current user and token
        res.status(201).send({ user, token })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

http interceptor error handling

The http interceptor, located inside the web application's core folder, uses the interceptor interface that implements the intercept method that will intercept every single http request in the SPA to modify the headers but also to catch a specific type of error.

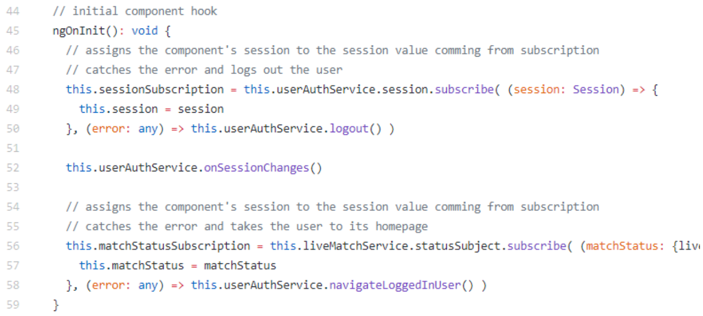
At the line 39, the next variable returns an observable that is modified using the "pipe" method. Inside the "pipe" method, the "catchError" function will catch every http error and it will check if the http request returned an unauthorized 401 http code and if that is true, the user will be logged off from the web application. The "catchError" method throws all the remaining errors to be dealt with by other http interceptors.



```
23      // intercepts every http request in the app
24      intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
25
26          // add the application/json accept header
27          req = req.clone({ headers: req.headers.set("Accept", "application/json")})
28
29          // add the application/json accept header
30          if(!req.headers.has("Content-Type")){
31              req = req.clone({ headers: req.headers.set("Content-Type", "application/json") })
32          }
33
34          // adds the bearer token
35          if(this.userAuthService.isLoggedIn()){
36              req = req.clone({ headers: req.headers.set("Authorization", this.userAuthService.getBearerToken())})
37          }
38
39          return next.handle(req).pipe(
40              // catches all http requests errors
41              catchError( (event: HttpErrorResponse) => {
42                  // if the server error is Unauthorized
43                  if(event.status === 401) {
44                      // user is logged out
45                      this.userAuthService.clearCoockiesToLogin()
46                      return null;
47                  }
48                  // lets other http interceptors catch the error
49                  return throwError(event);
50              })
51          );
52      }
53
54  }
```

observable error handling

Inside the navbar's typescript component, the "ngOnInit" contains two observables that need to be subscribed for the value as well as the error.

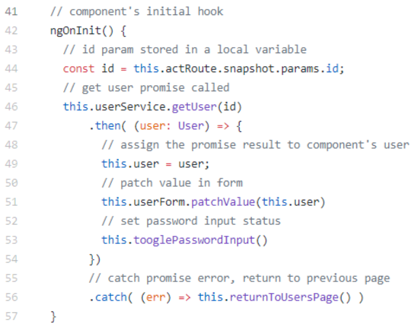


```
// initial component hook
ngOnInit(): void {
  // assigns the component's session to the session value comming from subscription
  // catches the error and logs out the user
  this.sessionSubscription = this.userAuthService.session.subscribe( (session: Session) => {
    this.session = session
  }, (error: any) => this.userAuthService.logout() )

  this.userAuthService.onSessionChanges()

  // assigns the component's session to the session value comming from subscription
  // catches the error and takes the user to its homepage
  this.matchStatusSubscription = this.liveMatchService.statusSubject.subscribe( (matchStatus: {liv
    this.matchStatus = matchStatus
  }, (error: any) => this.userAuthService.navigateLoggedInUser() )
}
```

The "session" observable (reacts to any change in the user session in the web application), is subscribed to update the component's session variable, but if any error happens with the session the user will be completely logged off.

The "statusSubject" observable (updates the live match status), is subscribed to update the component's matchStatus variable, but it will take the user to his homepage if an error occurs. This behaviour takes place, in case of error, to avoid unexpected results if the user is in the live match page.

Promise error handling

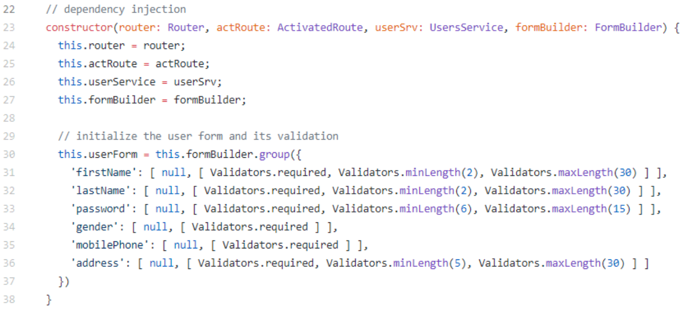


```
// component's initial hook
ngOnInit() {
  // id param stored in a local variable
  const id = this.actRoute.snapshot.params.id;
  // get user promise called
  this.userService.getUser(id)
      .then( (user: User) => {
        // assign the promise result to component's user
        this.user = user;
        // patch value in form
        this.userForm.patchValue(this.user)
        // set password input status
        this.tooglePasswordInput()
      })
      // catch promise error, return to previous page
      .catch( (err) => this.returnToUsersPage() )
}
```

Promises are asynchronous code that needs to also get its exceptions caught using its own features, just as observables.

In the user edit component, located inside the "ngOnInit" hook the user service's "getUser" function returns a promise that resolves by assigning a user to the component's user variable, patching the value in the form and by changing the password input status. If this promise rejects the admin will be redirect to the users list page.

Form validation



```
// dependency injection
constructor(router: Router, actRoute: ActivatedRoute, userSrv: UsersService, formBuilder: FormBuilder) {
  this.router = router;
  this.actRoute = actRoute;
  this.userService = userSrv;
  this.formBuilder = formBuilder;

  // initialize the user form and its validation
  this.userForm = this.formBuilder.group({
    'firstName': [ null, [ Validators.required, Validators.minLength(2), Validators.maxLength(30) ] ],
    'lastName': [ null, [ Validators.required, Validators.minLength(2), Validators.maxLength(30) ] ],
    'password': [ null, [ Validators.required, Validators.minLength(6), Validators.maxLength(15) ] ],
    'gender': [ null, [ Validators.required ] ],
    'mobilePhone': [ null, [ Validators.required ] ],
    'address': [ null, [ Validators.required, Validators.minLength(5), Validators.maxLength(30) ] ]
  })
}
```

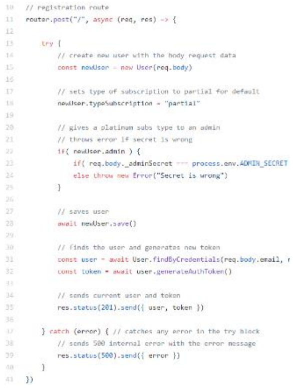
Before any data is send to the server, it is important the data gets validated even before the form accepts the input.

In the user edit component, the user form is built by declaring all its fields and its validators. The "Validators" built-in class has many important validators, including the "required", the "minLength" and the "maxLength". These validators ensure the data is going to be sent to the server is already validated in the most basic terms.

Internal documentation

Page Commenting

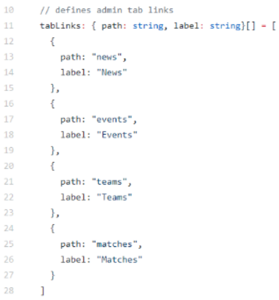
Comments are statements which are not processed by the computers as their only intent is to give more insight to the reader about what the program is doing, statement by statement. As an example, the registration route has many comments which let the reader clearly understand the process being undertaken.



```
// registration route
router.post("/", async (req, res) => {

    try {
        // create new user with the body request data
        const newUser = new User(req.body)

        // sets type of subscription to partial for default
        newUser.typeSubscription = "partial"

        // gives a platinum subs type to an admin
        // throws error if secret is wrong
        if( newUser.admin ) {
            if( req.body._adminSecret === process.env.ADMIN_SECRET
            else throw new Error("Secret is wrong")
        }

        // saves user
        await newUser.save()

        // finds the user and generates new token
        const user = await User.findByCredentials(req.body.email, r
        const token = await user.generateAuthToken()

        // sends current user and token
        res.status(201).send({ user, token })

    } catch (error) { // catches any error in the try block
        // sends 500 internal error with the error message
        res.status(500).send({ error })
    }
})
```

Consistent Indentation

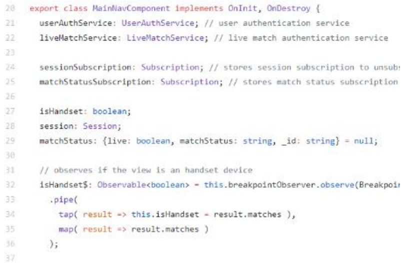
Indentation is a technique in programming that extends program statement on many lines to better describe data structures, functions or logic. In the admin dashboard component, the tab's array of links is indented in many lines what improves greatly the code's readability.



```
// defines admin tab links
tabLinks: { path: string, label: string}[] = [
  {
    path: "news",
    label: "News"
  },
  {
    path: "events",
    label: "Events"
  },
  {
    path: "teams",
    label: "Teams"
  },
  {
    path: "matches",
    label: "Matches"
  }
]
```

Code grouping

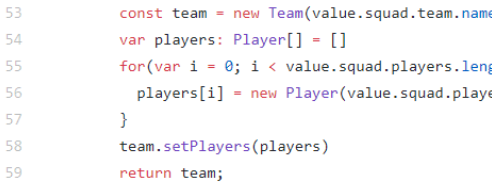
The code grouping happens when statements, that are related, are grouped. This technique separates section that have different intents and highlights what each sections of the code do. In the main navbar component, the services, the subscriptions the variables and the observable are all grouped to provide a clearer readability.



```
export class MainNavComponent implements OnInit, OnDestroy {
  userAuthService: UserAuthService; // user authentication service
  liveMatchService: LiveMatchService; // live match authentication service

  sessionSubscription: Subscription; // stores session subscription to unsub
  matchStatusSubscription: Subscription; // stores match status subscription

  isHandset: boolean;
  session: Session;
  matchStatus: {live: boolean, matchStatus: string, _id: string} = null;

  // observes if the view is an handset device
  isHandset$: Observable<boolean> = this.breakpointObserver.observe(Breakpoi
    .pipe(
      tap( result => this.isHandset = result.matches ),
      map( result => result.matches )
    );

```

Consistent variable naming

The variable naming is important to easily understand what type of data the variable is holding, as well as knowing what kind of object is going to be instantiated from a class. In the team's service, it is easy to spot where the players and teams are stored and which classes instantiate these variables. The iterator variable "i" is also a good example of good variable naming as it allows the reader to read the looping statement better.



```
const team = new Team(value.squad.team.name
var players: Player[] = []
for(var i = 0; i < value.squad.players.leng
  players[i] = new Player(value.squad.playe
}
team.setPlayers(players)
return team;
```

Testing

Test Plan

The strategy taken was to view the projects requirements fill up the generic user, administrator and subscriber test tables with tests case scenarios and the expected results.

Following the test preparation, the testing began by creating a new account and just test the generic user tests cases. After testing the generic user test cases the user account status was changed to administrator and then the administrator test cases began as well. Following the successful completion of the administrator test cases, a new user account was created to stay as a subscriber. This new account was created to test successfuly all subscribers test cases but also to be tested at the same time with the previous admin account to test the live stream test cases for admin and subscriber.

Test run

Generic User

Test case description	**Expected result**	**Actual result**
Potential user registers	Registers account and logs in user	Success
User logs in the web app	Logs in user	Success
Logs out	Logs the user out of the session	Success
Account details update	Updates user account and saves them	Success
Changes password	Changes password and saves it	Success

Administrator

Test case description	Expected result	Actual result
Status changes to admin	Changes account status from user to admin and saves it	Success
View other users	Views other user’s profiles	Success
Updates other users	Updates other user’s account data	Success
Deletes other users	Delete other user’s accounts	Success
Creates article	Publishes new news article	Success
Updates article	Updates news article publication	Success
Delete article	Deletes news article publication	Success
Creates new event	Publishes new event	Success
Update event	Updates event publication	Success
Deletes event	Delete event publication	Success
Creates new team	Creates a new team	Success
Edit team	Updates team’s profile information	Success
Deletes team	Deletes a team	Success
Creates player	Add player to an existing team	Success
Updates a player	Updates player profile information	Success
Deletes a player	Deletes a player profile	Success
Creates a match	Adds two teams and assigns a round	Success
Creates match events in match	Creates match events that update the match’s score and the league’s table	Success
Deletes match event in match	Deletes match events that update the match’s score and the league’s table	Success
Broadcasts a live match	Broadcasts a live match stream for all logged in users if editing the match	Success

Subscriber

Test case description	Expected result	Actual result
Views newsletter	Views the newsletter	Success
Views a news article	Views a news article	Success
Views the event list	Views the event list	Success
Views a team	Views a team profile	Success
Views a player	Views a player profile	Success
Views the table	Views the league classification	Success
Views fixtures	Views leagues round matches	Success
Views a match	Views a match report	Success
Views match events	Views match events on the match reports	Success
Creates a match	Adds two teams and assigns a round	Success
Creates match events in match	Creates match events that update the match’s score and the league’s table	Success
Deletes match event in match	Deletes match events that update the match’s score and the league’s table	Success
Broadcasts a live match	Broadcasts a live match stream for all logged in users if editing the match	Success

User Documentation

User

How to become a user?

To become a user it is necessary to go to the register page and sign up for an account by enter all the required personal information, which includes first and second names, email, password, gender, mobile phone and address.

How to sign in?

To sign in it is necessary to go to the login page and enter the email and password.

How do I change my account details?

After signing in the application, the user must go to the profile page, by using the link in the navbar toolbar or in the side navbar. Inside the profile page, the user must click the edit profile button to edit the account details. In this page the user can change the first and second names, address, phone number and the gender.

How do I change my password?

The user must go to the edit profile page and press the "change password" button, then it is possible to change the account password.

Administrator

How do I get the administrator account?

The user must go to the edit profile page and press the "change password" button, then it is possible to change the account status. The user must press the button "change account status", select the type of status wanted and type the admin secret.

How do I manage other users in the application?

When signed in, the administrator can should go to the users' page by pressing the "users" button in the navbar or in the side navbar. The page has a list of all the users who signed up for the application. Then the admin can go to the user profile to view the details, edit the user information or even delete the user. If the administrator wants to edit the user's personal information he can do it so by pressing the "update" button that leads to a form where many personal information fields can be updated, namely the first and second names, password, phone number, gender and address.

How do I manage the news in the application?

To manage the news, the administrator must go to the admin panel by pressing the “admin” button, either in the navbar or in the side navbar. After going to the admin dashboard, the admin must press the news button, located in the admin dashboard tab. In this page, the admin can create new news articles, by pressing “create new Article”, or select any of the news and view its content. In the news article page, it is possible to update the article or to delete it.

How do I manage the events in the application?

To manage the events, the user must go to the events’ page located in the admin dashboard. This page allows the publication of new event, by pressing the “create new Event” button, or to select any existing events in the list. The admin can access any event’s page to view its content, as well as update it or even to delete the news article.

How do I manage the teams in the application?

To manage the teams, the admin must go to the teams’ page located in the admin dashboard. This page contains all the existing teams as well as the option to create a new team. To edit a team’s information the admin must select one of the teams to go to its page. In each team’s page there is the option to edit the team, by updating its content, or even to delete it.

How do I manage the players in the application?

To manage players, the admin must go to the team that the player plays currently for. When inside a team’s page, it is possible to see the full squad and if the intent is to add a player to the squad it is necessary to press the “add player to team” button. If the intent is to edit or to delete a player, the admin needs to select one of the players from the list and, when inside the player’s page, press the “edit” or the “delete” buttons.

How do I manage the matches in the application?

To manage the matches the admin must go to the match page by pressing the “matches” tab button, located in the admin dashboard. This page contains a list of all matches, sorted by rounds and a button to create a new match. If the admin intends to create a new match, it is necessary to select to teams and its round. To edit or delete the match, the admin needs to go inside a match’s page and press the “edit” or “delete” buttons.

An Administrator will always broadcast to every client when a match page is accessed and he is responsible for updating the match report by adding or removing match events. Goals and own goals will change the match´s score and the subscriber will be able to keep up with the live result.

Subscriber

How do I view the newsletter?

To view the newsletter, the subscriber needs to press the “news” button, located in the navbar or side navbar. The subscriber will be taken to a list of events that he can further select to view each article’s content.

How do I view the events?

To view the events, the subscriber needs to press the “events” button, located in the navbar or in the navbar. This button will show a list of events that the subscriber can select to view its content.

How do I view the league table, fixtures, teams and players?

To access all the league information the subscriber only needs to go to the League section by pressing the “league” button located in the navbar or in the side navbar. This takes the current user to a page with a tab for table and for fixtures.

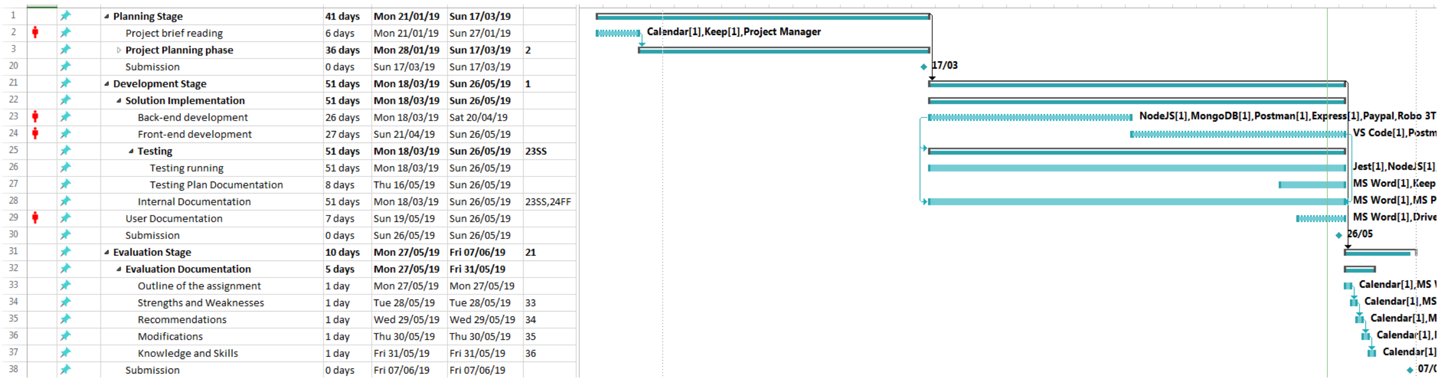
The table contains the current teams’ position in the league classification alongside the more data related. The subscriber can select one of the teams and view its players as well.

The fixtures display all the matches per each round, including its scores. A match page contains the score, and match events done by players. The subscriber can view any player or team that features in that match.

How can I keep up with a live result?

To keep up with a current match live score, the subscriber only needs to press the “live” blinking button in the navbar. This button will take the user to the live match report.

Updated Gant chart



1		📌	**Planning Stage**	**41 days**	**Mon 21/01/19**	**Sun 17/03/19**	
2	🧍	📌	Project brief reading	6 days	Mon 21/01/19	Sun 27/01/19	
3		📌	**Project Planning phase**	**36 days**	**Mon 28/01/19**	**Sun 17/03/19**	2
20		📌	Submission	0 days	Sun 17/03/19	Sun 17/03/19	
21		📌	**Development Stage**	**51 days**	**Mon 18/03/19**	**Sun 26/05/19**	1
22		📌	**Solution Implementation**	**51 days**	**Mon 18/03/19**	**Sun 26/05/19**	
23	🧍	📌	Back-end development	26 days	Mon 18/03/19	Sat 20/04/19	
24	🧍	📌	Front-end development	27 days	Sun 21/04/19	Sun 26/05/19	
25		📌	**Testing**	**51 days**	**Mon 18/03/19**	**Sun 26/05/19**	23SS
26		📌	Testing running	51 days	Mon 18/03/19	Sun 26/05/19	
27		📌	Testing Plan Documentation	8 days	Thu 16/05/19	Sun 26/05/19	
28		📌	Internal Documentation	51 days	Mon 18/03/19	Sun 26/05/19	23SS,24FF
29	🧍	📌	User Documentation	7 days	Sun 19/05/19	Sun 26/05/19	
30		📌	Submission	0 days	Sun 26/05/19	Sun 26/05/19	
31		📌	**Evaluation Stage**	**10 days**	**Mon 27/05/19**	**Fri 07/06/19**	21
32		📌	**Evaluation Documentation**	**5 days**	**Mon 27/05/19**	**Fri 31/05/19**	
33		📌	Outline of the assignment	1 day	Mon 27/05/19	Mon 27/05/19	
34		📌	Strengths and Weaknesses	1 day	Tue 28/05/19	Tue 28/05/19	33
35		📌	Recommendations	1 day	Wed 29/05/19	Wed 29/05/19	34
36		📌	Modifications	1 day	Thu 30/05/19	Thu 30/05/19	35
37		📌	Knowledge and Skills	1 day	Fri 31/05/19	Fri 31/05/19	36
38		📌	Submission	0 days	Fri 07/06/19	Fri 07/06/19	