DOCUMENT_HEADING

12 files

(file list disabled)

src\Authentication.ts

```ts
import * as express from "express";
import dbQuery from "./dbQuery";

abstract class Authentication {

    // SOURCE: Credit to https://emn178.github.io/online-tools/sha256.html for simple hashing tool. Used to manually generate initial hashes, then hashes with salts.

    // check if provided credentials are valid for route being accessed
    public static authBarrier(isAdministratorLevel:boolean) {

        // since isAdministratorLevel needs to be passed with the default express parameters, a new modified function is returned which has access to both but also has the correct method signature for Express.
        // Source: Express docs, section "Configurable middleware" at bottom of page: https://expressjs.com/en/guide/writing-middleware.html
        return async (req: express.Request, res: express.Response, next: express.NextFunction) => {

            // if the parameters are not present in the request, set them equal to ""
            console.log(req.headers);

            // types for req.headers are string|string[]|any|(etc...), so cast to string since it will never be any other type (except null)
            const username:string = <string>req.headers["username"]??""; // fallback to "" if is null
            const passwordHash:string = <string>req.headers["passwordhash"]??"";


            // if valid, proceed to next layer in middleware stack and stop execution of this function
            if (await this.checkHash(username, passwordHash) && await this.checkUserLevel(username, isAdministratorLevel)) return next();

            res.status(403).send(`Resource forbidden: Current user does not have required user level for access to ${req.path}`); // forbidden, refused to serve requested resource, different to 401
        }

    }

    private static async checkHash(username:string, hash:string) {
        // if either are undefined or the user doesn't exist, storedHash will be undefined, making the final comparison false
        const storedHash = (await dbQuery.makeDBQuery(`SELECT password_hash FROM "Account" WHERE username = $1`, [username]))?.rows[0]?.password_hash;
        return storedHash == hash;
    }

    private static async checkUserLevel(username:string, isAdministratorRoute:boolean) {
        // isAdministratorRoute is true when trying to access a route that only administrator-
```

```
level users should be able to access
        const isUserAdminLevel = (await dbQuery.makeDBQuery(`SELECT administrator_level FROM
"Account" WHERE username = $1`, [username]))?.rows[0]?.administrator_level;
        // Admins should be able to access all routes so return true if admin
        if (isUserAdminLevel) {
            return true;
        } else { // not admin
            return isUserAdminLevel == isAdministratorRoute; // admin and trying to access
admin route?
        }
    }

    public static async getLoginSalt(username: string) {
        // ?. allows undefined to be returned (which is desired here)
        return (await dbQuery.makeDBQuery(`SELECT salt FROM "Account" WHERE username = $1;`,
[username]))?.rows[0]?.salt;
    }
}

export default Authentication;
```

src\Camera.ts

```
class Camera {

    public CameraID: number;
    public IPAddress: string;
    public EventURL: string;
    public ResponseFormat: string;
    public CarparkID: number;

    constructor(CameraID: number, IPAddress: string, EventURL: string, ResponseFormat: string,
CarparkID: number) {
        this.CameraID = CameraID;
        this.IPAddress = IPAddress;
        this.EventURL = EventURL;
        this.ResponseFormat = ResponseFormat;
        this.CarparkID = CarparkID;
    }
}

export default Camera;
```

src\Cameras.ts

```
import * as express from "express";
import Camera from "./Camera";
import Carpark from "./Carpark";
import dbQuery from "./dbQuery";
import Logs from "./Logs";
import Vehicle from "./Vehicle";

abstract class Cameras {

    public static cameras: Camera[] = [];
```

```
    public static async loadCameras() {
        const cameraRecords = (await dbQuery.makeDBQuery(`SELECT camera_id, ip_address,
event_url, response_format, carpark_id FROM "Camera";`, [])).rows;

        for (let i = 0; i < cameraRecords.length; i++) {
            const rec = cameraRecords[i];
            Cameras.addCamera(new Camera(rec.camera_id, rec.ip_address, rec.event_url,
rec.response_format, rec.carpark_id));
        }
    }

    private static addCamera(camera:Camera) {
        Cameras.cameras.push(camera);
    }

    private static getCameraIDFromIP(ip_address:string):number {

        for (let i = 0; i < Cameras.cameras.length; i++) {
            if (Cameras.cameras[i].IPAddress == ip_address) return
Cameras.cameras[i].CameraID;
        }

        return -1;

    }


    public static async processEvent(request:express.Request, response:express.Response) {
        const detectedNumberplate = request.body["Picture"].Plate.PlateNumber;
        const detectedVehicleImage = request.body["Picture"].NormalPic.Content;
        const detectedVehicleTimestamp = request.body["Picture"].SnapInfo.SnapTime;

        console.log("CAMERA ID: " + Cameras.getCameraIDFromIP(request.ip).toString());
        console.log("FREE SPACES: " + (await Carpark.getFreeSpaces()).toString());
        console.log(detectedNumberplate, request.body["Picture"].SnapInfo.Direction);


        if (request.body["Picture"].SnapInfo.Direction == "Reverse") {
            // vehicle exiting, no need to check numberplate
            Carpark.openGate(request, response)
            await Logs.updateLogRecordOnExit(detectedNumberplate, detectedVehicleImage,
detectedVehicleTimestamp);


        } else if (request.body["Picture"].SnapInfo.Direction == "Obverse") {
            // vehicle entering carpark
            if (await Vehicle.isKnown(detectedNumberplate)) {
                // something returned, duplicate numberplates not allowed in table therefore
only 1 record will be returned.

                console.log("KNOWN VEHICLE");
                if ((await Carpark.getFreeSpaces()) <= 0) {
                    await Logs.createRecordNoEntry(detectedNumberplate, detectedVehicleImage,
true, Cameras.getCameraIDFromIP(request.ip), detectedVehicleTimestamp);
                } else {
                    Carpark.openGate(request, response);
                    await Logs.createRecord(detectedNumberplate, detectedVehicleImage, true,
Cameras.getCameraIDFromIP(request.ip), detectedVehicleTimestamp);
                }
```

```
            } else {
                // nothing returned, unknown vehicle: Keep gate shut and notify reception (by setting known_vehicle to false), independent of free spaces
                console.log("UNKNOWN VEHICLE");
                await Logs.createRecordNoEntry(detectedNumberplate, detectedVehicleImage, false, Cameras.getCameraIDFromIP(request.ip), detectedVehicleTimestamp);
            }

        }

    }


}

export default Cameras;
```

src\Carpark.ts

```
import * as express from "express";
import { Server } from "http";

import Authentication from "./Authentication";
import dbQuery from "./dbQuery";
import Cameras from "./Cameras";
import dbPool from "./dbPool";
import Logs from "./Logs";
import Tenant from "./Tenant";


class Carpark {


    public static server: express.Application;
    public static httpServer: Server;

    private static CarparkID: number;
    private static TotalSpaces: number;

    public static tenants:Tenant[];


    constructor(CarparkID: number, TotalSpaces: number) {

        Carpark.CarparkID = CarparkID;
        Carpark.TotalSpaces = TotalSpaces;
        // the start method must be run at a higher level
    }


    public static async start() {
        // spin up cameras - must be done before server starts (below)
        await Cameras.loadCameras();

        Carpark.server = express();
        Carpark.loadRoutes();
```

```
        // get the server's listen port from .env file. Could not be defined in file (returns undefined), so automatically assign 8000 in this case as the default.
        const serverListenPort:number = (process.env.SERVERLISTENPORT !== undefined)? parseInt(process.env.SERVERLISTENPORT):(8000);

        // start listening and store HTTPServer instance for graceful shutdown
        Carpark.httpServer = Carpark.server.listen(serverListenPort, "0.0.0.0", () => { console.log("server listening") });

        // if any request takes longer than
        // Carpark.httpServer.on('connection', function(socket) {
        //  socket.setTimeout(10000, () => {console.log("Request timed out");});});
        // });
    }

    // count number of Log records without exit timestamps (still in carpark), get total spaces in carpark and return difference between them.
    public static async getFreeSpaces() { // not getUsedSpaces because this can be used directly in Cameras.processEvent(), 1 extra call here.
        const totalSpaces = (await dbQuery.makeDBQuery(`SELECT total_spaces FROM "Carpark";`, [])).rows[0].total_spaces;
        const usedSpaces = (await dbQuery.makeDBQuery(`SELECT COUNT(*) AS used_spaces FROM "Log" WHERE exit_timestamp IS NULL;`, [])).rows[0].used_spaces; // count Log records without an exit_timestamp
        return totalSpaces - usedSpaces;
    }


    public static async getCarparkRecords() {
        return (await dbQuery.makeDBQuery(`SELECT carpark_id, total_spaces FROM "Carpark";`, []));
    }

    // previous implementation used to update the counter in the database. Free spaces are now derived from data in base on demand.
    /*
    private async editCarparkSpaceCounter(increment:1|-1, cameraAddress:string) {
        console.log(`UPDATE "Carpark" SET used_spaces = used_spaces + ${increment.toString()} FROM "Camera" WHERE "Carpark".carpark_id = "Camera".carpark_id AND "Camera".ip_address = '${cameraAddress}';`);

        await dbQuery.makeDBQuery(`UPDATE "Carpark" SET used_spaces = used_spaces + $1 FROM "Camera" WHERE "Carpark".carpark_id = "Camera".carpark_id AND "Camera".ip_address = '$2';`, [increment.toString(), cameraAddress])
    }
    */

    public static openGate(req: express.Request, res: express.Response) {
        // code to send request to activate camera relay
    }

    private static async openGateFromClient(req: express.Request, res: express.Response) {
        // can only open gate for unknown vehicle that has not entered (entry_tsp==exit_tsp)
        const LogID:number = parseInt(req.params.LogID);
        const logToModify = await Logs.getLogByID(LogID);

        if ((logToModify.entry_timestamp.toString() == logToModify.exit_timestamp.toString()) && !logToModify.known_vehicle) {

            Logs.setExitTimestampNullForLogID(LogID);
            Carpark.replySuccess(res);
        } else {
```

```
            res.status(403)
                .send("Cannot open gate for a vehicle that has already entered or is authorised")
                .end()
        }

    }


    private static replyQueryError(err: Error, res:express.Response) {
        // bad query so send status 400 with error message
        console.log("QUERY ERROR\n", err);

        res.status(400)
            .send("Query failed with error: " + err.message)
            .end();
    }

    private static replySuccess(res: express.Response) {
        res.status(200)
            .end();
    }

    private static loadRoutes(): void {

        Carpark.server.use(express.json( { limit: "10mb" } ));


        // this will resolve to "/NotificationInfo/TollgateInfo" with Dahua cameras (true for this project)
        Carpark.server.post(Cameras.cameras[0].EventURL, async (req, res) => {
            console.log(req.body);
            console.log(new Date().toString());
            res.json({"Result":true});
            try {
                await Cameras.processEvent(req, res);
                this.replySuccess(res);
            } catch (error:any) {
                //this.replyQueryError(error, res);
                console.error(error);
                res.status(200);
                res.end() // for now
            }
        })

        Carpark.server.post("/NotificationInfo/KeepAlive", (req: express.Request, res: express.Response) => {
            Carpark.replySuccess(res);
        })


        // for requests from ReceptionUI.

        // Carpark.server.get("/query/:table", async (req:express.Request, res:express.Response) => {
        //  // query string will hold fields

        // })
        const authBarrierAdminRoute = Authentication.authBarrier(true);
        const authBarrierBaseRoute = Authentication.authBarrier(false);
```

```
        Carpark.server.get("/loginSalt", async (req: express.Request, res: express.Response)
=> {
            const username:string|any = req.query.username;

            res.send((await Authentication.getLoginSalt(username)));
            res.status(200);
            res.end();
        })

        Carpark.server.post("/login", authBarrierBaseRoute, async (req: express.Request, res:
express.Response) => {
            // essentially a check so the user can know immediately if their credentials do
not work; there is no is_logged_in state or token to change
            res.status(200).send("Valid credentials");
            res.end();
        })

        Carpark.server.get("/tenantDataFromLogID/:LogID", authBarrierBaseRoute, async
(req:express.Request, res:express.Response) => {
            // returns tenant records joined with vehicle records
            res.status(200);
            const tenantData = await
Tenant.getTenantDataFromLogID(parseInt(req.params.LogID));
            console.log(tenantData);

            try {
                res.json({ // undefined will be picked up on client and interpreted as vehicle
not belonging to tenant if no records returned
                    "TenantID": tenantData[0]?.tenant_id,
                    "Forename": tenantData[0]?.forename,
                    "Surname": tenantData[0]?.surname
                })
                res.end();
            } catch (error:any) {
                this.replyQueryError(error, res);
            }

        });

        Carpark.server.get("/carparkStatistics", async (req:express.Request,
res:express.Response) => {
            res.status(200);
            try {
                res.json({
                    "FreeSpaces": await this.getFreeSpaces(), // will always return a record
                    "TotalSpaces": this.TotalSpaces
                })
                res.end();
            } catch (error:any) {
                this.replyQueryError(error, res);
            }
        })

        Carpark.server.get("/logData/:recordCount", authBarrierBaseRoute, async
(req:express.Request, res:express.Response) => {
            // returns the most recent req.params.recordCount number of records in the Log
table (highest id)
            console.log("R E Q U E S T    FOR    L O G S");

            res.status(200);
            await Logs.loadLogs(parseInt(req.params.recordCount)); // load fresh logs from db
            res.json(Logs.getLogs());
```

```
            res.end()
        });

        Carpark.server.get("/entryCount/:TenantID", async (req: express.Request, res:
express.Response) => {
            // TODO: JOIN with Tenant and Vehicle for more information.
            const data = (await dbQuery.makeDBQuery(`SELECT numberplate, COUNT(*),
MAX(entry_timestamp) FROM "Log" GROUP BY numberplate;`, [])).rows;

        })

        Carpark.server.post("/openGate/:LogID", authBarrierBaseRoute, async (req:
express.Request, res: express.Response) => {this.openGateFromClient(req, res)});
    }



    public static async shutdown() {
        this.httpServer.close();
        await dbPool.dbPool.end();
        process.disconnect();
        process.exit(0);
    }


}

export default Carpark;
```

src\Log.ts

```
class Log {

    public EventID: number;
    public CameraID: number;
    public VehicleID: number;
    public Numberplate: string;
    public EntryTimestamp: Date;
    public ExitTimestamp: Date;
    public EntryImageBase64: string;
    public ExitImageBase64: string;
    public KnownVehicle: boolean;
    // https://stackoverflow.com/a/42884828 to store dates/times
    // client.query will return a timestamp string in the promise result rows

    constructor(EventID: number, CameraID: number, VehicleID: number, Numberplate: string,
EntryTimestamp: Date, ExitTimestamp: Date, EntryImageBase64: string, ExitImageBase64: string,
KnownVehicle: boolean) {
        this.EventID = EventID;
        this.CameraID = CameraID;
        this.VehicleID = VehicleID;
        this.Numberplate = Numberplate;
        this.EntryTimestamp = EntryTimestamp;
        this.ExitTimestamp = ExitTimestamp;
        this.EntryImageBase64 = EntryImageBase64;
        this.ExitImageBase64 = ExitImageBase64;
```

```
        this.KnownVehicle = KnownVehicle;
    }

}

export default Log;
```

src\Logs.ts

```
import dbQuery from "./dbQuery";
import Log from "./Log";

abstract class Logs {

    private static logs: Log[] = [];


    public static getLogs():Log[] {
        return Logs.logs;
    }

    public static async loadLogs(count:number) { // loads the most recently created logs into Logs.logs[]
        var logRecords:any[] = [];

        const max_log_id:number = (await dbQuery.makeDBQuery(`SELECT log_id FROM "Log" ORDER BY log_id DESC LIMIT 1;`, [])).rows[0].log_id;


        // clear Logs.logs[], then populate with new Log() instances
        Logs.logs = [];

        for (let i = max_log_id; i > max_log_id-count; i--) {

            console.log("Getting log id " + i);

            const rec = (await dbQuery.makeDBQuery(`SELECT log_id, camera_id, numberplate, entry_timestamp, exit_timestamp, vehicle_id, entry_image_base64, exit_image_base64, known_vehicle FROM "Log" WHERE log_id = $1;`, [i.toString()])).rows[0];

            if (rec == undefined) {count+=1; continue};
            Logs.addLog(new Log(rec.log_id, rec.camera_id, rec.vehicle_id, rec.numberplate, rec.entry_timestamp, rec.exit_timestamp, rec.entry_image_base64, rec.exit_image_base64, rec.known_vehicle));
        }

    }

    private static addLog(log:Log) {
        Logs.logs.push(log);
    }

    public static async getLogByID(log_id:number) { // throws error if none returned
        return (await dbQuery.makeDBQuery(`SELECT log_id, numberplate, entry_timestamp, exit_timestamp, known_vehicle FROM "Log" WHERE log_id = $1;`, [log_id.toString()])).rows[0];
    }

    public static async getLatestLogByNumberplate(numberplate:string) { // throws error if
```

```
none returned
        return (await dbQuery.makeDBQuery(`SELECT log_id, numberplate, entry_timestamp,
exit_timestamp FROM "Log" WHERE numberplate = $1 ORDER BY log_id DESC LIMIT 1;`,
[numberplate])).rows[0];
    }

    public static async setExitTimestampNullForLogID(log_id:number) {
        await dbQuery.makeDBQuery(`UPDATE "Log" SET exit_timestamp = NULL WHERE log_id = $1;`,
[log_id.toString()]);
    }

    public static async createRecord(numberplate:string, image:string, knownVehicle: boolean,
camera_id: number, timestamp:string) {
        // TODO: need to match vehicle_id
        const secondsString = Logs.timestampStringToSeconds(timestamp).toString();
        await dbQuery.makeDBQuery(`INSERT INTO "Log" (numberplate, entry_timestamp,
entry_image_base64, known_vehicle, camera_id) VALUES ($1, to_timestamp($2), $3, $4, $5);`,
[numberplate, secondsString, image, knownVehicle.toString(), camera_id.toString()]);
    }

    public static async createRecordNoEntry(numberplate:string, image:string, knownVehicle:
boolean, camera_id: number, timestamp:string) {
        // TODO: need to match vehicle_id
        // for when a vehicle is detected but doesn't enter (i.e. no free spaces available or
unauthorised). Set entry and exit timestamps to be equal.
        const secondsString = Logs.timestampStringToSeconds(timestamp).toString();
        await dbQuery.makeDBQuery(`INSERT INTO "Log" (numberplate, entry_timestamp,
entry_image_base64, exit_timestamp, known_vehicle, camera_id) VALUES ($1, to_timestamp($2),
$3, to_timestamp($4), $5, $6);`,
                [numberplate, secondsString, image, secondsString, knownVehicle.toString(),
camera_id.toString()]);
    }



    public static async updateLogRecordOnExit(numberplate:string, image:string,
timestamp:string) {
        console.log(`UPDATE "Log" SET exit_timestamp =
to_timestamp(${Logs.timestampStringToSeconds(timestamp)}), exit_image_base64 = '<image data
here>' WHERE log_id = (SELECT MAX(log_id) FROM "Log" WHERE "Log".numberplate =
'${numberplate}');`);
        await dbQuery.makeDBQuery(`UPDATE "Log" SET exit_timestamp = to_timestamp($1),
exit_image_base64 = $2 WHERE log_id = (SELECT MAX(log_id) FROM "Log" WHERE "Log".numberplate =
$3);`, [Logs.timestampStringToSeconds(timestamp).toString(), image, numberplate]);
    }

    private static timestampStringToSeconds(timestampString:string) {
        // camera gives SnapTime in format "YYYY-MM-DD HH:mm:ss" e.g. "2023-03-03 18:15:04".
Convert to millis since unix epoch, divide by 1000 to get seconds.
        return new Date(timestampString).getTime()/1000;
    }

}

export default Logs;
```

src\Server.ts

```
// import * as express from 'express';
// import main from "./index";
```

```
// const router = express.Router();


// export default router;
// export default new Server().express;


// https://medium.com/@pmhegdek/oop-in-typescript-express-server-d9368b97740e
```

src\Tenant.ts

```
import dbQuery from "./dbQuery";

class Tenant {
    public TenantID:number;
    public Forename:string;
    public Surname:string;

    constructor(TenantID: number, Forename: string, Surname: string) {
        this.TenantID = TenantID;
        this.Forename = Forename;
        this.Surname = Surname;
    }

    public static async getTenantDataFromLogID(log_id:number) {
        return (await dbQuery.makeDBQuery(`SELECT log_id, forename, surname FROM "Tenant",
"Vehicle", "Log" WHERE "Log".log_id = $1 AND "Log".numberplate = "Vehicle".numberplate AND
"Vehicle".tenant_id = "Tenant".tenant_id LIMIT 1;`, [log_id.toString()])).rows;
    }
}

export default Tenant;
```

src\Vehicle.ts

```
import dbQuery from "./dbQuery";

abstract class Vehicle {
    public VehicleID:number;
    public Numberplate:string;
    public TenantID:number;

    constructor(VehicleID: number, Numberplate: string, TenantID: number) {
        this.VehicleID = VehicleID;
        this.Numberplate = Numberplate;
        this.TenantID = TenantID;
    }

    public static async getAllData() {
        return await dbQuery.makeDBQuery(`SELECT vehicle_id, numberplate, tenant_id FROM
"Vehicle";`, [])
    };

    public static async getData(numberplate:string) {
        return await dbQuery.makeDBQuery(`SELECT vehicle_id, numberplate, tenant_id FROM
"Vehicle" WHERE numberplate = $1;`, [numberplate]);
```

```
    }

    public static async isKnown(numberplate:string) {
        if ((await Vehicle.getData(numberplate)).rows.length > 0) return true;
        return false;
    }
}

export default Vehicle;
```

src\dbPool.ts

```
const { Pool } = require("pg");

// Referenced https://northflank.com/guides/connecting-to-a-postgresql-database-using-node-js
and
// https://node-postgres.com/apis/pool for below and to create sample query snippet in
index.ts

abstract class dbPool {
    public static dbPool: typeof Pool;

    public static async createPool() {
        // connect to local Postgresql database using credentials in local .env file
        dbPool.dbPool = await new Pool({
            host: process.env.PG_HOST,
            port: process.env.PG_PORT,
            user: process.env.PG_USER,
            password: process.env.PG_PASSWORD,
            database: process.env.PG_DATABASE,
            ssl: false,
        });
    }

}

export default dbPool;
```

src\dbQuery.ts

```
import dbPool from "./dbPool";

abstract class dbQuery  {

    // for use by client in the case that something (i.e. new tenant/vehicle) needs to be
added to db
    public static generateInsertQuery(table:string, parameters:object):string {

        const keys = Object.keys(parameters);
        const values = Object.values(parameters);

        const query = `INSERT INTO ${ table } (${ keys }) VALUES (${ values })`;
        return query;
    }
```

```
    public static async makeDBQuery(query:string, parameters:Array<string>) {
        // console.log(`QUERY: ${query}`)
        const result = await dbPool.dbPool.query(query, parameters)

        return result;
    }

}

export default dbQuery;
```

src\index.ts

```
import Carpark from "./Carpark";
import dbPool from "./dbPool";

require("dotenv").config();


class Main {

    private carpark!: Carpark;

    constructor() {

        this.start();

    }

    // load object data from db and create all necessary objects (new Carpark() creates new Camera()s etc..)
    // can and should only be called from constructor
    private async start() {
        await dbPool.createPool();

        const carpark_records = (await Carpark.getCarparkRecords()).rows;
        console.log(carpark_records);

        // first create a Carpark object for each record in DB (there will never be more than one, but for completeness)
        for (let i = 0; i < carpark_records.length; i++) {
            const carpark_record = carpark_records[i];
            this.carpark = new Carpark(carpark_record.carpark_id, carpark_record.total_spaces);
        }
        await Carpark.start()
        console.log("STARTED");
    }

}



const app:Main = new Main();

process.on("SIGTERM", () => {
    console.log("Exiting...");
```

```
    Carpark.shutdown();
})
process.on("SIGINT", () => {
    console.log("Exiting... int");
    Carpark.shutdown();
})
// https://northflank.com/guides/connecting-to-a-postgresql-database-using-node-js
// https://medium.com/bb-tutorials-and-thoughts/how-to-build-nodejs-rest-api-with-express-and-postgresql-typescript-version-121b5a11c9a6
// https://www.digitalocean.com/community/tutorials/setting-up-a-node-project-with-typescript


// sample query
// (async () => {
//  const dbClient = await getClient();
//  const log_data = await dbClient.query("SELECT * FROM \"Log\""); // referencing mixed-case table names https://stackoverflow.com/a/695312/7169383
//  console.log(log_data.rows);
//  await dbClient.end();
// })
```