

# Advanced Javascript CA1

Building a REST API.

Creative Computing 4<sup>th</sup> year

IADT

By Gary Thompson(N00161709)

# Contents

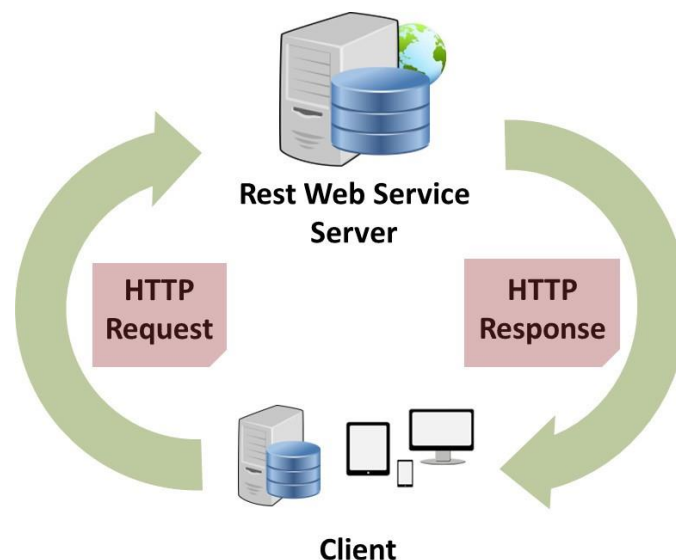
1. Introduction
2. Technologies
3. Implementation
4. Testing
5. Conclusion

# 1. Introduction

This Assignment was to create a REST API that uses Express, MongoDB, JWT and Passport. In order to understand this we must first explore what a REST API is. REST stands for representational state transfer, and API stands for Application programming interface. So a REST API is an API that conforms to the REST design principals which are:

1. Uniform interface or requests
2. The separation of client and server processes
3. They must be stateless so that a request contains all relevant information on the application side
4. They must be cachable for data such as tokens
5. They must have a layered system architecture, such as controllers and middleware
6. They must have on demand code, or executable code upon request.

Below is what a usual API request can look like:



There are a number of REST API applications, allowing a user to login or register to a service, showing a list of football matches, or user created notes being stored on a server, a REST API is an incredibly dynamic way to integrate applications and servers. This CA's REST API focuses on logging a user in and allowing them CRUD access for a miniature store's database.

What do you need to make use of a REST API? First a database with tables and parameters in place to fit the exact use case of your app. If creating a recipe API, your recipe table might contain a name, ingredients, levels of complexity and so on. There also needs to be a way to consume your API's output, for this CA our requests were handled through an API request client such as Postman, Insomnia or thunder client, however most practical uses of API's are consumed through a mobile app, a desktop app, or a web-based app. Lastly you will need an API itself to send requests and responses between your app and your database.

## 2. Technologies

There were several technologies used to create the REST API for this CA. The technologies used were:

1. MongoDB (Non-relational Database)
2. Mongoose (ODM library)
3. Node.js (Asynchronous JS backend environment)
4. Express.js (Web Application Framework for Node.js)
5. Passport.js (Authentication Middleware for Node.js)
6. JWT (JSON Web tokens for authentication)
7. Insomnia(API request/response client)

MongoDB is a non relational database, this sets it apart from the likes of a mySQL database which is a relational database. The difference between these two is that a non relational database like MongoDB represents data as JSON documents, whereas mySQL treats its data as tables and rows. Both are good but very different, MongoDB offers a lot of flexibility in terms of its key/value pairs and allowing documents to be “related” by nesting them inside each other or allowing each document to be part of an array whereas mySQL handles this by using its primary and secondary keys, allowing aspects of tables to relate to one another by their ids that are a separate field in a second table. MongoDB is also more optimized for speed, allowing a user to upload a vast amount of data all at once with functions like insertMany() as opposed to inserting data one row at a time with mySql.

Mongoose is built to make node and mongoDB applications more streamlined, contain less boilerplate, and have less dataleaks. It contains built in validation, query building and offers a schema based solution to modelling the data for an API. The main reason for using mongoose is that it allows you to create schemas. These are similar to java classes, if the object in question is the json object. The schema for your JSON documents map to the MongoDB database table and create the overall shape of the objects. For example, the miniature schema is used to show that, each miniature must have a name which will be a string, a faction which will also be a string, an inPrint value which will be a Boolean, a price value which will be a number, and an inStock Boolean. The example can be seen below.

```
const mongoose = require('mongoose');
//this is the miniature schema, which is a blueprint used for a
const miniatureSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  faction: {
    type: String,
    required: true,
  },
  inPrint: {
    type: Boolean,
    required: true,
    default: false,
  },
  price: {
    type: Number,
    required: true,
  },
  inStock: {
    type: Boolean,
    required: true,
    default: false,
  },
});
//this is then exported so that other files can use it.
module.exports = mongoose.model('Miniature', miniatureSchema);
```

Node.js is a backend JS environment that allows for a lot of synchronous concurrent code to be ran. In essence, whilst a normal PHP environment sends a file then waits for it to be returned, Node.js can send the file and immediately is ready to receive or send another request, it doesn't have to wait for the response first. It is what handles all the CRUD functionality in our application. Node works in events, an example of an event is somebody trying to access the port 9000 that we use as a gateway for our server. Node mainly operates on the Async, Await, and callback functions. Async and await work like promises that must be fulfilled before something is returned, and callback functions allow for something to be returned when it is ready without holding up the rest of the application. We also use Nodemon which notices changes in the environment and is able to restart the server upon saving.

Express.js is a Node.js framework that allows for server side web apps to be much faster. This makes things such as routing much more efficient. Express allows for the applications endpoints to be much more simplistic, with methods that mirror http methods such as GET for retrieving data, POST for pushing data, PATCH for updating existing data, and DELETE for deleting data. An example of how express can be useful is shown below.

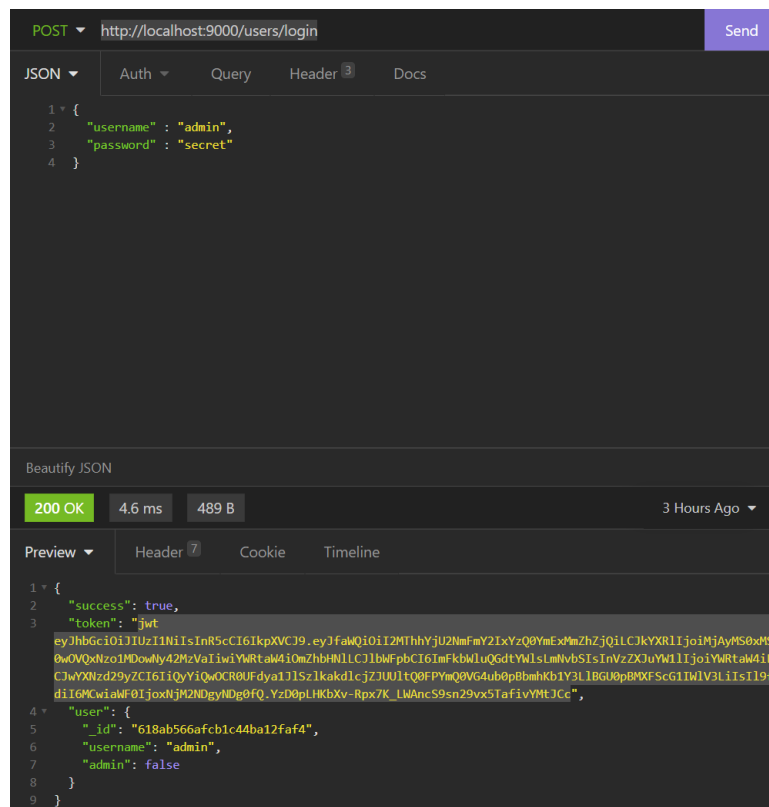
```
router.post("/register", userController.register);  
router.post("/login", userController.login);  
router.route("/user-profile").get(
```

This shows that in order to access the methods for the http request /register that is a post request, all that is needed is to access the register method inside the user controller.

Passport.js is an authentication middleware library for Node.js. This works in combination with JWT or JSON web tokens in order to provide this API with authentication. Authentication is a form of controlling who can access what. In this API, anyone without a login or a guest can view all of the miniatures in our database, however only authenticated users have access to create, update and delete operations. This is done with passport by creating strategies that allow for control of how the JWT is deployed. For example in this API, a strategy is created that combines the user ID with a JWT payload and returns it as a useable token.

Insomnia is an API request client that allows us to test the functionality of the API without needing a frontend application to consume it. This is done by sending requests to the server and testing API endpoints under a number of different situations. There are alternatives to Insomnia such as postman or thunder client. Insomnia stores requests such as : " <http://localhost:9000/users/login>"

And allows us to access these with various request types, in this instance a post, to allow the user to input their user details and have their user document and token returned along with a success message. This can be seen below:



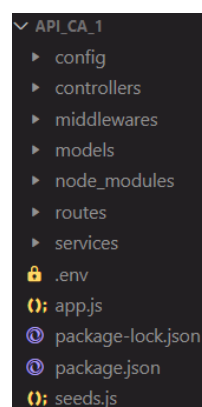
Insomnia allows for full control of the type of body, with a MongoDB database which deals in JSON documents, a JSON body must be used, as well as a header, which has things such as Authorization or content type passed in for the API to recognize. Finally the query itself is executed once all other pre requisites have been met to allow for the request to be processed and responded to with the correct output.

### 3. Implementation

The Implementation of this API was difficult, especially due to working with so many technologies for the first time, however with a combination of provided code, online documentation, tutorials, Nodemon error outputs, and prior JS development experience, the API was completed with a full suite of functionality.

The file structure for the API is as follows:

Some files are within a sub folder, whereas other files are accessed in the main folder. The first file being discussed is the db.config file in the config folder.



The db.config file is very straight forward. It contains a single string which is the URL of our mongo db database, and exports it so that it is a global variable accessible anywhere within the app.

```
module.exports = {  
  db: "mongodb://localhost/MiniaturesDBex",  
};
```

The .env file is a file we want locked from the app and database so that no users can see the contents. There is just a few variables saved in there such as the url of the database, the secret key, the port and the db name.

```
DB_URL=mongodb://localhost/MiniaturesDBex  
DB_NAME=MiniaturesDBex  
PORT=9000  
SECRET_KEY=secretauth
```

The package.json contains all of the dependencies used for this API. This contains the library or plugin, and the version number. Whenever we use the NPM install command, whatever is installed will automatically be added to this file as well as the most up to date version number. The project number, author and other information is also stored here.

```
{  
  "name": "ajs_cal",  
  "version": "1.0.0",  
  "description": "",  
  "main": "app.js",  
  "scripts": {  
    "start": "nodemon app.js"  
  },  
  "author": "Gthomp",  
  "license": "ISC",  
  "dependencies": {  
    "bcrypt": "^5.0.1",  
    "bcryptjs": "^2.4.3",  
    "dotenv": "^10.0.0",  
    "express": "^4.17.1",  
    "express-unless": "^1.0.0",  
    "jsonwebtoken": "^8.5.1",  
    "mongodb": "^4.1.4",  
    "mongoose": "^6.0.12",  
    "mongoose-unique-validator": "^3.0.0",  
    "passport": "^0.5.0",  
    "passport-jwt": "^4.0.0"  
  },  
  "devDependencies": {  
    "nodemon": "^2.0.14"  
  }  
}
```

The Seeds.js file is a script that can be run to populate the database with data, so it can be used right off the bat. This seed is only for the Miniature table so a user will still have to register traditionally in order to be able to manipulate the data within besides viewing it. When this script is run by “nodemon start seeds.js” first, a connection is established.

```
mongoose  
  .connect(dbConfig.db, {  
    useNewUrlParser: true,  
    useUnifiedTopology: true,  
  })  
  .then(() => {  
    console.log("Seed Connected");  
  })  
  .catch((err) => {  
    console.log(err);  
  });
```

Then all of the data in the database is deleted, and a JSON array called `seedMiniatures` is pushed instead that contains 5 JSON objects representing miniatures.

```
const seedDB = async () => {
  await Miniature.deleteMany({});
  await Miniature.insertMany(seedMiniatures);
};

seedDB().then(() => {
  mongoose.connection.close();
});
```

Here is the `seedMiniatures` array.

```
const seedMiniatures = [
  {
    name: "Lorgar",
    faction: "Word Bearers",
    inPrint: false,
    price: 45.99,
    inStock: false,
  },
  {
    name: "Dante",
    faction: "Blood Angels",
    inPrint: true,
    price: 18.99,
    inStock: true,
  },
  {
    name: "Tigirius",
    faction: "Ultra Marines",
    inPrint: true,
    price: 24.99,
    inStock: false,
  },
  {
    name: "Typhus",
    faction: "Death Guard",
    inPrint: true,
    price: 18.99,
    inStock: true,
  },
  {
    name: "Konrad Kurze",
    faction: "Night Lords",
    inPrint: true,
    price: 45.99,
    inStock: false,
  },
];
```



Finally there is the app.js. This is the main script or index that is launched and ties the other parts of the API together. It calls in imports for express, mongoose, the db config file, our error handler and the dotenv config.

```
const express = require("express");
const mongoose = require("mongoose");
const dbConfig = require("../config/db.config");
require("dotenv").config();

const errors = require("../middlewares/errors");

const app = express();
```

Then the database connection is initialized globally, passing in the NewUrlParser to allow the old url parser to be used incase an issue is run into with the newer one that mongo.db created as the old one is deprecated. It also uses unified topology to set up a connection string and begin to do operations immediately rather than waiting for a connection method. Then it prints the database is connected unless there is an error in which case the error is returned.

```
mongoose
  .connect(dbConfig.db, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(
    () => {
      console.log("Database Connected");
    },
    (error) => {
      console.log("Database cant be connected: " + error);
    }
  );
```

The middleware is also called here, with the app.use function. This is what allows the middleware to be called globally. Users.routes, and miniatures is called, as well as express and errorHandler.

```
app.use(express.json());

app.use("/users", require("../routes/users.routes"));

app.use(errors.errorHandler);

app.use("/miniatures", require("../routes/miniatures"));
```

Finally in the app.js, a listener is set up to watch all traffic coming and going through port 9000 which is the port our database runs off in the localhost.

```
app.listen(process.env.port || 9000, function () {
  console.log("Ready to Go");
});
```

In order to have an API you have to have models that reflect the data in the database. This API has two, the miniature.js, and the user.model. The miniature.js is a model that reflects the miniature objects in the database. Mongoose is used to create a miniatureSchema that is a blueprint for all miniature objects to follow. The schema requires 5 properties, a name which is a string, a faction which is a string, an inPrint value which is a Boolean, a price which is a number, and an inStock value which is a Boolean. The defaults for both Booleans was set to false, as if this was a real miniature shop, and something was added accidentally, presenting that it is in stock when it isn't would cause problems. The schema was exported and the final schema can be seen below.

```
const miniatureSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  faction: {
    type: String,
    required: true,
  },
  inPrint: {
    type: Boolean,
    required: true,
    default: false,
  },
  price: {
    type: Number,
    required: true,
  },
  inStock: {
    type: Boolean,
    required: true,
    default: false,
  },
});
```

Next there is the user.model, and this is slightly more complex as not only is there a schema, but some methods that the schema must access in order to attain full functionality. The schema contains a username string, an email string, a password string, a date which is a date.now (shows date of registry) and an admin which is a Boolean variable. The username email and password are all required to register, and the email must be unique, however the data is only to show the exact moment that a user registered, and the admin variable is a handover from a different piece of functionality that no longer is relevant.

```
const userSchema = new Schema({
  username: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  date: {
    type: Date,
    default: Date.now(),
  },
  admin: {
    type: Boolean,
    default: false,
  },
});
```

There is a unique validator plugin for the email which must be unique to show that the email is already in use as opposed to crashing the api, this is handled.

```
userSchema.plugin(uniqueValidator, { message: "Email is already in use" });
```

A generate hash method is also used using bcrypt. This method takes the password that is entered by the user and encrypts it by hashing, so that no user or even inspecting the database can show the true contents of the password. The password for most users is "secret" but when inspected in the database the password shows "\$2b\$08\$X6ko1I/aF5SRi0i0c0oGE0FrMPcaoQu.ce09v015UapWUCV0fBda"

This shows that our encryption works and that user security is effective.

```
userSchema.methods.generateHash = (password) => {  
  return bcrypt.hashSync(password, bcrypt.genSaltSync(8), null);  
};
```

We can also use bcrypt and mongoose to compare the password that the user entered with the password that is stored in the database to ensure their password is valid. This works by getting the user entered password and pushing it through bcrypts hashing functionality, then comparing that string value with the string value in the database and returning it. This is used for a user that is already registered and logging back in.

```
userSchema.methods.validatePassword = async (username, password) => {  
  let thisuser = await mongoose.model("user", userSchema).findOne({ username: username });  
  return bcrypt.compareSync(password, thisuser.password);  
};
```

The user schema is saved in the User constant and exported to be used in other places of the application.

```
const User = mongoose.model("user", userSchema);  
module.exports = User;
```

Next there is routes, these are where our specific endpoints are used and given access to the controllers. This is where the functionality lies for what happens when a specific url is entered. There is a users.routes and a miniatures.js. The users.routes main purpose is to handle the endpoints and requests of the user table of the database. There are three routers inside here, all of which refer to the userController which contains the methods for each endpoint. There is a register, login, and user-profile router, which has a method such as post or get for what type of request they are fulfilling.

```
router.post("/register", userController.register);  
router.post("/login", userController.login);  
router.route("/user-profile").get(userController.userProfile);
```

This is good coding practice as everything has a clear destination. The miniatures.js route is not as well constructed. The reason for this is that instead of referring each endpoint to a method inside a controller, the miniatures controller doesn't exist, and all miniature specific methods exist in the miniatures.js routes file, which is a mistake that compromises the structure and readability of the code, however its function remains the same. Each method is contained in the router itself rather than referring to the method in the controller. Each method here is incredibly important, and contains some of the CRUD functionality for the miniatures table. The first is get all miniatures.

```
router.get("/", async (req, res) => {  
  try {  
    const miniatures = await Miniature.find();  
    res.json(miniatures);  
  } catch (err) {  
    res.send("Error" + err);  
  }  
});
```

This is the get all miniatures method, that is connected directly to the .get(/) request. It is asynchronous and runs until the Miniature.find() function runs. It then responds with a json document full of miniatures. It is within a try catch which means it will try to carry out its primary functionality but if it fails, it will catch rather than crash, and send an error message. Next is the find miniature by id function.

```
router.get("/:id", async (req, res) => {
  try {
    const miniature = await Miniature.findById(req.params.id);
    res.json(miniature);
  } catch (err) {
    res.send("Error" + err);
  }
});
```

This uses /:id in the request to pass in to the findById function, rather than the find function in get all miniatures. This is passed into req.params.id so that the function knows which miniature to retrieve, and again runs to completion unless an error occurs. Next is the post miniature function.

```
router.post("/", async (req, res) => {
  const token = await getToken(req.headers);
  if (!token) {
    return res.status(401);
  }
  let user = await jwt.verify(token, process.env.SECRET_KEY);

  if (user) {
    const miniature = new Miniature({
      name: req.body.name,
      faction: req.body.faction,
      inPrint: req.body.inPrint,
      price: req.body.price,
      inStock: req.body.inStock,
    });

    try {
      const a1 = await miniature.save();
      res.json(a1);
    } catch (err) {
      res.send("Error");
    }
  }
});
```

This is quite different to the get miniatures function. First of all in order to access it, a token is needed, meaning that you must be an authenticated user to create a new miniature object. This is then verified with jwt and if you are a logged in user, a new Miniature object is created with a name, faction, inPrint, price, and inStock value. The value is saved as a1 and printed as a result in a try catch. Next is the update Miniature function.

```
router.patch("/:id", async (req, res) => {
  const token = getToken(req.headers);
  if (!token) return res.status(401);
  const user = jwt.verify(token, process.env.SECRET_KEY);
  if (user)
    try {
      const miniature = await Miniature.findById(req.params.id);
      miniature.name = req.body.name;
      miniature.faction = req.body.faction;
      miniature.inPrint = req.body.inPrint;
      miniature.price = req.body.price;
      miniature.inStock = req.body.inStock;
      const a1 = await miniature.save();
      res.json(a1);
    } catch (err) {
      res.send("Error");
    }
});
```

This is the update miniature function, this has the same token and user verification as post, however it uses patch to update existing information in the database. It finds the id of an existing miniature, and then pulls the body of that miniature, and replaces any parts that are edited by the user so that if a miniature name was changed, but everything else was constant, it would not delete any other properties that weren't changed. This is then saved as a1 and returned as a json object. Finally for miniatures there is the delete function.

```
router.delete('/:id', async (req, res) => {
  const token = getToken(req.headers);
  if (!token) return res.status(401);
  const user = jwt.verify(token, process.env.SECRET_KEY);
  if (user) {
    try {
      const miniature = await Miniature.findById(req.params.id);
      miniature.deleteOne({ _id: req.params.id }).then(
        res.status(200).json({
          message: "Deleted",
        })
      );
    } catch (err) {
      res.send("Error" + err);
    }
  }
});
```

This is the delete miniature function, this uses delete and uses the id of the miniature to find the specific object to delete. It has the same user and token verification as post and patch, and uses the deleteOne function, passing in the id required to find the object and remove it from the database. It then prints a status 200 "Deleted" to show that it worked, unless there is an error which is then caught.

The methods for the user table are contained in the user controller. First there is the register function. This begins with requiring a body, and allowing it to compose of email, username and password. All of these must not be null, if they are a specific error is returned for each field.

```
exports.register = (req, res) => {
  const { body } = req;
  let { email, username, password } = body;

  if (!email) {
    return res.json({
      success: false,
      message: "Error: Email cannot be blank.",
    });
  }
  if (!username) {
    return res.json({
      success: false,
      message: "Error: Username cannot be blank.",
    });
  }
  if (!password) {
    return res.json({
      success: false,
      message: "Error: Password cannot be blank.",
    });
  }
}
```

Then some quality of life features, email will ignore capitalization and remove spaces in order to eliminate any user error, for security reasons username was kept as is to avoid any accidental logins.

```
email = email.toLowerCase();
email = email.trim(); // remove spaces
```

Then a new user is created, with the parameters, email, username, password and saved whilst passing in the user object and an error upon saving and hashing the password. If an error is detected

a server error message is sent back, if it succeeds, a token is generated using passport and returned with the message "Account created for user".

```
let newUser = new User();
newUser.email = email;
newUser.username = username;
newUser.password = newUser.generateHash(password);
newUser.save((err, user) => {
  if (err) {
    return res.json({
      success: false,
      message: "Error: Server error.",
    });
  }
  return res.json({
    success: true,
    token: "jwt " + user.token,
    message: "Account created for user",
  });
});
```

The next method is the login method, which gets the User schema passed in from the user model, and then uses the findOne method to retrieve the user object associated with the username entered in. If there's an error or no user found, these are handled.

```
User.findOne({ username }, (err, user) => {
  if (err) throw err;
  if (!user) {
    res.status(401).json({
      success: false,
      message: "Authentication failed. User not found.",
    });
  }
});
```

Otherwise, if the user entered password is validated against what is stored in the database with the validPassword function, then assign the token to the user, sign it and output the user object, otherwise state that authentication has failed.

```
} else if (user.validPassword(username, password)) {
  // check password match
  let token = jwt.sign(user.toJSON(), process.env.SECRET_KEY);
  res.json({
    success: true,
    token: "jwt " + token,
    user: {
      _id: user._id,
      username: user.username,
      admin: user.admin,
    },
  });
} else {
  res.status(401).json({
    success: false,
    message: "Authentication failed. Wrong password or username.",
  });
}
```

Finally for the user controller there is the userProfile function. This verifies a token to ensure that a user is authenticated and does it with the verify jwt function. If there is no token or there is no user, an error is sent, otherwise a message stating the user is authorized prints.

```

exports.userProfile = (req, res) => {
  const token = getToken(req.headers);
  if (!token) return res.status(401);
  const user = jwt.verify(token, process.env.SECRET_KEY);

  if (user) {
    return res.status(200).json({ message: "Authorized User" });
  } else {
    return res.status(401);
  }
};

```

Finally, there is middleware's, which are a step in the API that lays between the client and server. There are three middleware's in this API, auth.js, errors.js and passport.js.

Auth.js is how we get out token this is used in conjunction with passport to create the full authentication for this api. It has a single getToken method, that is exported to other areas of the api such as the user controllers and the miniature.js route. This is used in the header and its primary function is to split the token from the other components of auth such as the ids or "jwt" to get the token by itself. This allows for authorization regardless of any ids that are passed in with the token. It only runs if there is a header, and a header with the Authorization field and returns null if the token isn't split correctly.

```

const getToken = (headers) => {
  if (headers && headers.authorization) {
    let parted = headers.authorization.split(" ");
    if (parted.length === 2) {
      return parted[1];
    } else {
      return null;
    }
  } else {
    return null;
  }
};

```

Errors.js has some error handling to help trouble shoot queries for some general issues. If a string error occurs, status 400 is returned.

```

if (typeof err === "string") {
  return res.status(400).json({ message: err });
}

```

If a validation error occurs, a status 400 is also sent as a result.

```

if (typeof err === "ValidationError") {
  return res.status(400).json({ message: err.message });
}

```

If a user is unauthorized, then a 401 forbidden response is sent back.

```

if (typeof err === "UnauthorizedError") {
  return res.status(401).json({ message: err.message });
}

```

If there are any other errors that aren't caught by the above, then an error 500 status is sent.

```
return res.status(500).json({ message: err.message });
}
```

Finally, there is passport.js. This is the middleware that controls the jwt token generation. It does this by using the opts var, otherwise known as options. This is a toolkit of various methods that can only be used with passport such as jwtFromRequest or secretOrKey. First the jwt bearer is extracted from the header, followed by the secret key from the .env file.

```
var opts = {};
opts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken("jwt");
opts.secretOrKey = secret;
```

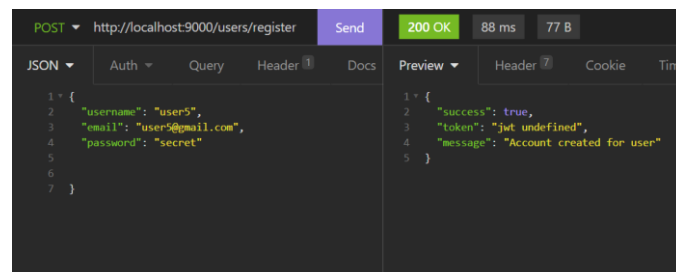
Next a new JwtStrategy is created which allows for the token to contain the payload, or a random encoded JSON object that sits in the token. The user's id is also passed into the payload. If there is an error then return done:false, if there is a user object then return done and the user object, otherwise return done, false.

```
new JwtStrategy(opts, function (jwt_payload, done) {
  User.findOne({ _id: jwt_payload.id }, function (err, user) {
    if (err) return done(err, false);
    if (user) {
      done(null, user);
    } else {
      done(null, false);
    }
  });
});
```

## 4. Testing

The testing chapter discusses testing endpoints and trying to break the API using error checking.

First every endpoint will be tested in insomnia. Using a username of user5, an email of [user5@gmail.com](mailto:user5@gmail.com) and a password of secret, first register will be tested. This is posted in the JSON body and the url <http://localhost:9000/users/register> with a post request gets the output:



The user complete with hashed password can be seen in the mongodb database:



```

1  _id: ObjectId("618b1e4daf8508cf096aa04b")      ObjectId
2  date: 2021-11-10T01:04:05.835+00:00           Date
3  admin: false                                    Boolean
4  email: "users@gmail.com"                       String
5  username: "users"                              String
6  password: "$2b$08$9w5HXkqUfmyWpBmInuZ1e5opSBdvJorgCtDe83Souaq8NC9dsg36" String
7  _v: 0                                           Int32

```

CANCEL UPDATE

What if an email is left out, or an email that already exists is entered? The following can be observed:

```
{
  "success": false,
  "message": "Error: Email cannot be blank."
}

{
  "success": false,
  "message": "Error: Server error."
}
```

This shows that the register is safe against user misinput. What about login?

The user5 username and password secret are entered to login and output the token for that user.

[illegible]

If a user enters the wrong username it will return a usernot found:

And if a username or password are blank, the following will return:

```

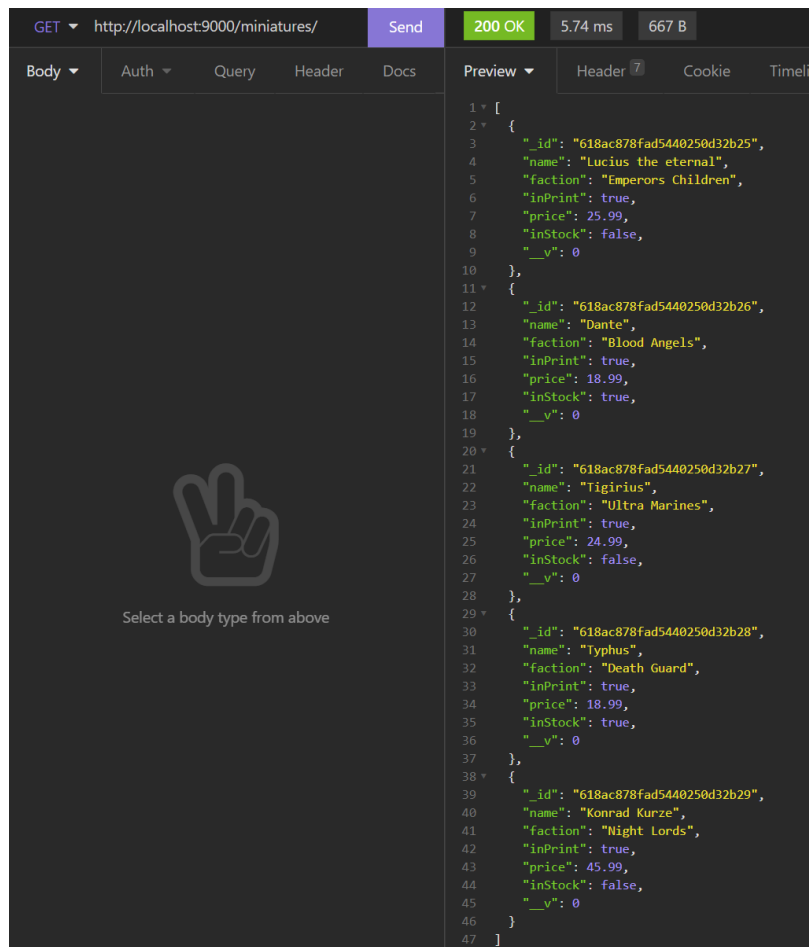
"username" : "user5",
"password" : ""
2      "success": false,
3      "message": "Error: Password cannot be blank"
4  }

```

Next if a user authentication check is carried out, it will return an authorized user once the token is passed in.

If there is no token the request times out which should be error checked.

In the get miniatures request, there is no auth needed so a full list of all miniatures is sent back:



GET http://localhost:9000/miniatures/ Send 200 OK 5.74 ms 667 B

Body Auth Query Header Docs Preview Header 7 Cookie Timeline

Select a body type from above

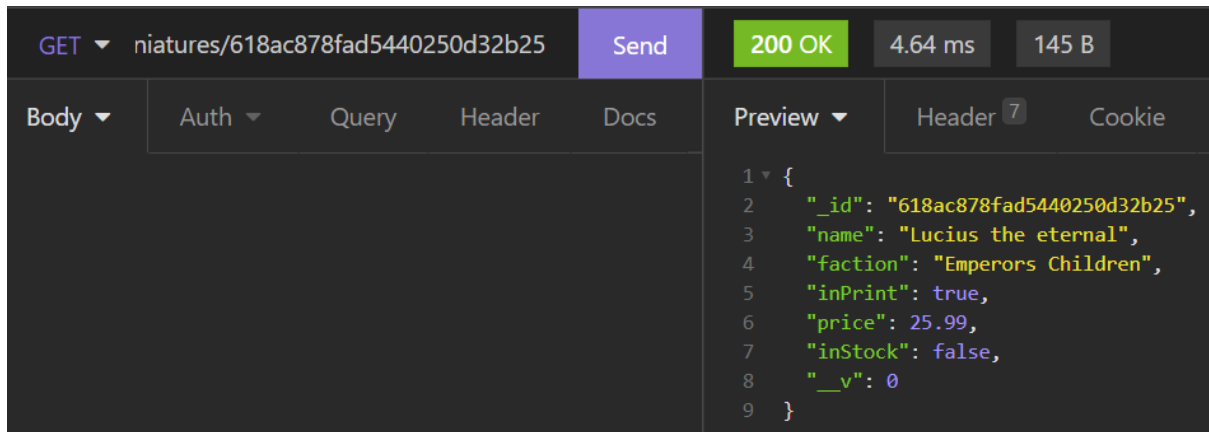
```
1 * [  
2 * {  
3   "_id": "618ac878fad5440250d32b25",  
4   "name": "Lucius the eternal",  
5   "faction": "Emperors Children",  
6   "inPrint": true,  
7   "price": 25.99,  
8   "inStock": false,  
9   "__v": 0  
10  },  
11 * {  
12   "_id": "618ac878fad5440250d32b26",  
13   "name": "Dante",  
14   "faction": "Blood Angels",  
15   "inPrint": true,  
16   "price": 18.99,  
17   "inStock": true,  
18   "__v": 0  
19  },  
20 * {  
21   "_id": "618ac878fad5440250d32b27",  
22   "name": "Tigirius",  
23   "faction": "Ultra Marines",  
24   "inPrint": true,  
25   "price": 24.99,  
26   "inStock": false,  
27   "__v": 0  
28  },  
29 * {  
30   "_id": "618ac878fad5440250d32b28",  
31   "name": "Typhus",  
32   "faction": "Death Guard",  
33   "inPrint": true,  
34   "price": 18.99,  
35   "inStock": true,  
36   "__v": 0  
37  },  
38 * {  
39   "_id": "618ac878fad5440250d32b29",  
40   "name": "Konrad Kurze",  
41   "faction": "Night Lords",  
42   "inPrint": true,  
43   "price": 45.99,  
44   "inStock": false,  
45   "__v": 0  
46  }  
47  ]
```

This is also reflected to be the same list in the database.

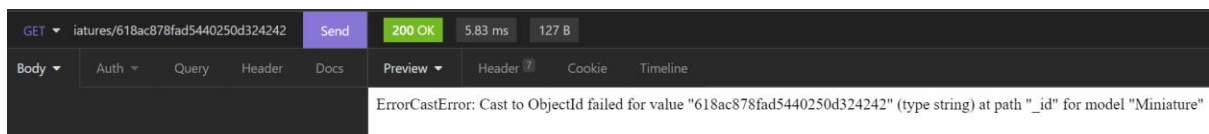
```
_id: ObjectId("618ac878fad5440250d32b25")  
name: "Lucius the eternal"  
faction: "Emperors Children"  
inPrint: true  
price: 25.99  
inStock: false  
__v: 0  
  
_id: ObjectId("618ac878fad5440250d32b26")  
name: "Dante"  
faction: "Blood Angels"  
inPrint: true  
price: 18.99  
inStock: true  
__v: 0  
  
_id: ObjectId("618ac878fad5440250d32b27")  
name: "Tigirius"  
faction: "Ultra Marines"  
inPrint: true  
price: 24.99  
inStock: false  
__v: 0  
  
_id: ObjectId("618ac878fad5440250d32b28")  
name: "Typhus"  
faction: "Death Guard"  
inPrint: true  
price: 18.99  
inStock: true  
__v: 0  
  
_id: ObjectId("618ac878fad5440250d32b29")  
name: "Konrad Kurze"  
faction: "Night Lords"  
inPrint: true  
price: 45.99  
inStock: false  
__v: 0
```

For the get Miniature by id request, an id needs to be passed into the request url.

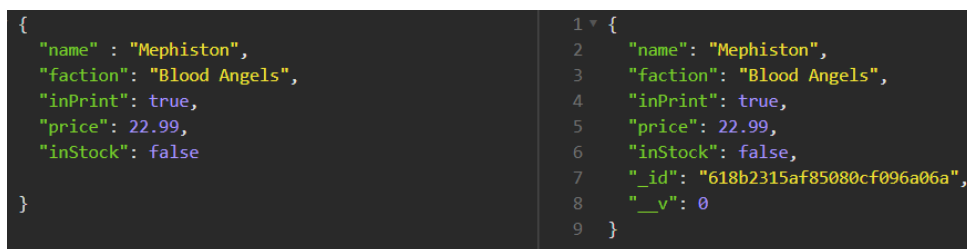
This needs no auth as anyone can access it.



An error is returned if a nonexistent id is passed into the request:

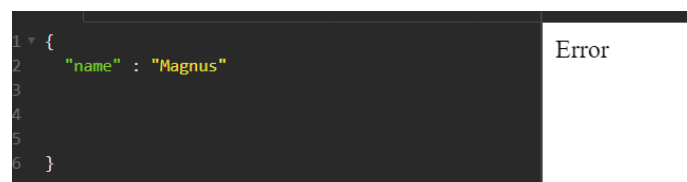


The post miniatures request needs authorization to allow it to be accessed if there is no token passed in the request will time out, but if there is a token a new miniature is posted to the database.



```
_id: ObjectId("618b2315af85080cf096a06a")
name: "Mephiston"
faction: "Blood Angels"
inPrint: true
price: 22.99
inStock: false
__v: 0
```

If any of the required fields are left out an error will be sent.



The patch miniature works the same, as the lack of a token will send the request to timeout, but if there is a token, the change can be perceived in insomnia and the database. If Dante 30.00, that change can be made.

```
_id: ObjectId("618ac878fad5440250d32b26")
name: "Dante"
faction: "Blood Angels"
inPrint: true
price: 18.99
inStock: true
__v: 0
```

```
{
  "name": "Dante",
  "faction": "Blood angels",
  "inPrint": true,
  "price": 30.00,
  "inStock": true
}
```

```
1 {
2   "_id": "618ac878fad5440250d32b26",
3   "name": "Dante",
4   "faction": "Blood angels",
5   "inPrint": true,
6   "price": 30,
7   "inStock": true,
8   "__v": 0
9 }
```

```
_id: ObjectId("618ac878fad5440250d32b26")
name: "Dante"
faction: "Blood angels"
inPrint: true
price: 30
inStock: true
__v: 0
```

This only works if all fields are present however, leaving out fields will result in an error.

PATCH tures/618ac878fad5440250d32b26 Send 200 OK 3.63 ms 5 B

JSON Auth Query Header 2 Docs Preview Header 7 Cook

```
1 {
2   "name": "dante",
3   "faction": "Blood Angels",
4   "price": 25.99
5
6 }
```

Error

The delete works very simply, when using an authenticated token, and passing an id into the query, the miniature is deleted. This example showing the previously edited dante being deleted.

DELETE tures/618ac878fad5440250d32b26 Send 200 OK 4.84 ms 21 B

Body Auth Query Header 1 Docs Preview Header 7 Cookie

```
1 {
2   "message": "Deleted"
3 }
```

<pre> _id: ObjectId("618ac878fad5440250d32b25") name: "Mephiston" faction: "Blood angels" inPrint: true price: 30 inStock: false __v: 0 </pre>
<pre> _id: ObjectId("618ac878fad5440250d32b27") name: "Tigirius" faction: "Ultra Marines" inPrint: true price: 24.99 inStock: false __v: 0 </pre>
<pre> _id: ObjectId("618ac878fad5440250d32b28") name: "Typhus" faction: "Death Guard" inPrint: true price: 18.99 inStock: true __v: 0 </pre>
<pre> _id: ObjectId("618ac878fad5440250d32b29") name: "Konrad Kurze" faction: "Night Lords" inPrint: true price: 45.99 inStock: false __v: 0 </pre>

If an id is passed in that doesn't currently exist, an error is thrown.

DELETE	tures/618ac878fad5440250d32b26	Send	200 OK	4.96 ms	56 B
Body	Auth	Query	Header 1	Docs	Preview
					Header 7
					Cookie
					Timeline
					ErrorTypeError: Cannot read property 'deleteOne' of null

A timeout also occurs if auth is left empty.

## 5. Conclusion

In conclusion, a REST API for a miniature shop was created for this CA. The API had full user login, registration, and authentication with CRUD for a Miniature table, parts of which were only accessible if authorized. The concept of a REST API was discussed as well as the technologies used. Implementation was explained and broken down, and extensive testing was carried out to show the functionality of the API. This CA overall was a success, and whilst the API is robust, it could use more bespoke error handling, as well as some expanded functionality such as a logout, or more comprehensive miniature related functionality. All references can be seen below:

<http://www.passportjs.org/docs/>

<https://www.ibm.com/cloud/learn/rest-apis>

<https://mongoosejs.com/docs/>

<https://www.abeautifulsite.net/posts/hashing-passwords-with-nodejs-and-bcrypt>

<https://medium.com/swlh/everything-you-need-to-know-about-the-passport-jwt-passport-js-strategy-8b69f39014b0>

<https://javascript.plainenglish.io/seeding-mongodb-database-from-node-the-simplest-way-3d6a0c1c4668>

<https://openbase.com/js/bcrypt/documentation>

<https://www.tabnine.com/code/javascript/functions/express/Express/delete>

<https://www.youtube.com/watch?v=eYVGoXPq2RA>

[https://www.youtube.com/watch?v=ZEg03f1o\\_vQ](https://www.youtube.com/watch?v=ZEg03f1o_vQ)