PFS – Assignment 2

Jacqui Meacle, Nathan Kafer, Sophie Coyte

ZEIT3120

1 TABLE OF CONTENTS

2 SCOPE

The scope of this project is to design a secure application capable of tracking inventory compatible with nu-tracker-company Pty system of ng RFID tags and BYOD smartphones to access stock level and location of inventory.

3 BACKGROUND

The Inventory Application produced is a REST API. REST or Representational State Transfer is a framework of guidelines to build a web services interface. Our REST API includes the API Interface, which includes how the client interacts with the API, the API internals, which listens and responds to client requests, and the API Backend, which is structured to facilitate the storage, retrieval, modification and deletion of data. Our Inventory API utilizes Postman as the API Development Interface, Express as the API Internals, and MongoDB as the API Backend.

4 TECHNICAL DECISIONS

4.1 SYSTEM REQUIREMENTS

Windows, MacOS, Linux, Docker. Most computers should have the required computing power to run the localhost server to test functionality.

4.2 SYSTEM ARCHITECTURE

The MEAN Framework, with slight changes, has been used to develop the REST API. The MEAN framework is a JavaScript framework for developing web applications (Subramanian & Raj, 2022). All back-end properties of the framework have been implemented with the client-side architecture being altered to use the Postman API. The decision to use Postman during development was to ensure efficient testing of the API without the need to create an additional web application. The parts of the MEAN stack framework that are most influential in this project are M – Mongo DB, a document-based database and E – Express.js, a Node.js framework.

REST is an architectural style for the development of coupled applications over the internet. In our case it is the coupling of the Client-Side Server (Postman), Express Server, and Mongo DB Database. Several Architectural restraints are imposed on the system due to the fundamental design characteristics of a REST API. These include its uniform nature, client-server interdependence, stateless implementation, cacheable nature, and layered system approach (Subramanian & Raj, 2022). Figure 1 shows how each of these characteristics can be implemented through the API system.

4.2.1 Uniform Nature

Due to the decoupling of Client and Server, strict interfaces are put in place to ensure the efficient flow of data to and from the API (Subramanian & Raj, 2022). This uniform nature

enforces a mutual standard which ensure both the client, server, and database understand each other.

4.2.2 Client-Server Inter-dependence

The Client and Server must have the ability to operate inter dependently and must not have any dependency on each other (Subramanian & Raj, 2022). Both should have the ability to be developed, and altered, in separation of each other, without compromising operability. Benefits include portability and scalability (Subramanian & Raj, 2022).

4.2.3 Statelessness

A stateless nature ensures that no client information is stored on the server. Each request sent between client and server should be treated as new. In a situation where the end user signs in to authorize functionality, then each request must contain the relevant information to grant such authorization. This is done with JWT tokens, which is discussed below in Security Considerations.

4.2.4 Cacheable

Caching is the standardized practice of storing copies of file in a temporary storage location that data can be retrieved quickly without requesting from the web server. Within the context of a REST API, caching can be used to reduce bandwidth, reduce latency, reduce load on servers, and hide network failures (Subramanian & Raj, 2022).

4.2.5 Layered system

The layered system ensures that as each component operates independent of the other, and layers should be able to be added, removed, modified or reordered without affecting system functionality.

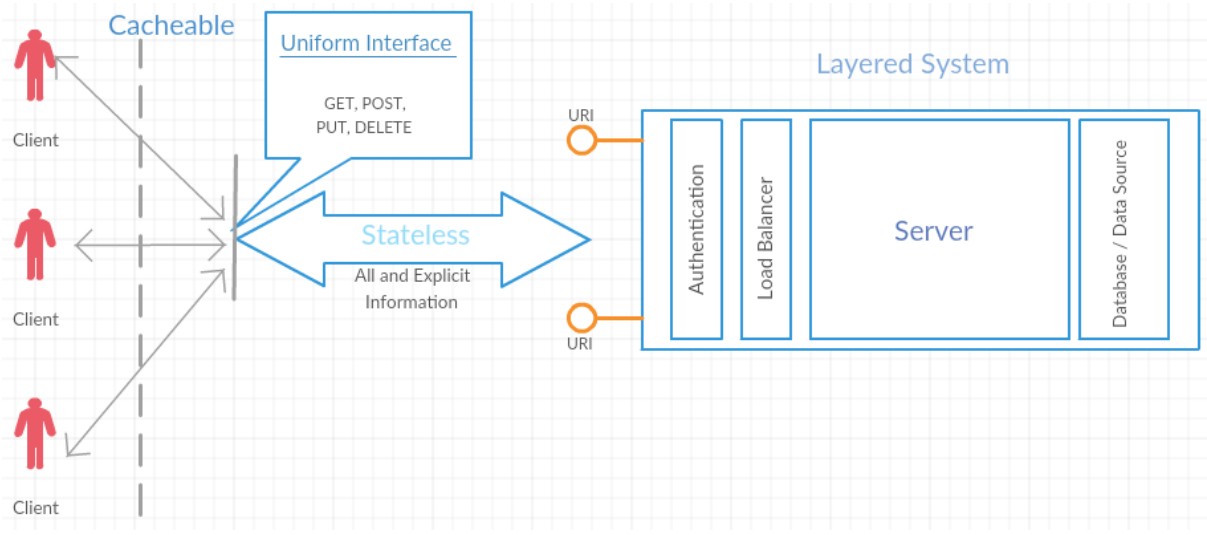

Figure 1: REST Architectural Style Constraints (Subramanian & Raj, 2022)

4.3 System Design

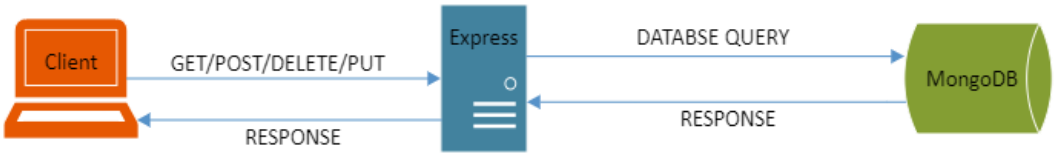

Figure 2: Flow of Data Diagram

Figure 1 shows the flow of data through the API system. The Client is the component that is the requester of a service and sends requests for various types of services to the server. The Server is the service provider and provides the requested services to the client as per the HTTP requests.

4.4 System Components

4.4.1 Express

The Express Development Environment includes the installation of Node.js, the NPM Package Manager, on a local computer. Express.js is a server framework for Node.js that forms the backend component of the MEAN stack as mentioned above. Express as a modern framework offers many advantages within the implementation of this application. The main features are versatility, usability and accessibility of its design, especially in comparison to other back-end languages such as PHP. Express.js has high interoperability with MongoDB and has advanced built-in security features recognized by industry as this language is the building block of PayPal, the world's most used secure payment system. The only disadvantage of implementing Express.js as the main scripting language of this application is the unfamiliarly of the framework within the development team, although the framework is still the most appropriate to use for this application as the development team is familiar with the base language JavaScript that Express.js is derived from.

4.4.2 MongoDB

MongoDB is the serverless database that securely stores the inventory of the application. As a non-sql database the document structure of MongoDB is highly flexible and scalable. Key value pairs of documents allow easy access in any programming language's native data structures, making it a highly flexible database and documents producing a JSON object for each data entry negates the need of an object-relational mapping technique. Further, the lack of SQL is an inherent security bonus as only 10% of databases are non-sql, therefore infrastructure and viability of an industry based on malicious code injection on these types of databases is much less developed. MongoDB also has its own security features consistent with the REST API characteristics as discussed above. Although MongoDB also has flaws in the use of this project. Memory allocation for data storage of individual JSON object entries is high and would likely become a consideration if this application were to be deployed on the scale of a real warehouse. Unfamiliarly, with non-relational databases and the complexity of performing join statements increases the difficulty of achieving first normal form leading to the duplication of data which will only create higher memory overhead. With memory allocation being the only considerable disadvantage of MongoDB and the cheap price of cloud storage, it is clear that MongoDB is a suitable database manager for this application.

4.4.3 Postman

Postman is the API chosen to host the application; it is ideal for the REST API testing. Using an API reduces the number of scripts created for development and removes the detail in programming, it has in-built security features and has cloud-hosted sharing capabilities that make it ideal for integration with BYOD phones as outlined in the project scope. The Postman UI is quite easy to use and has a high speed of service. The only disadvantage the Postman API UI faces over a standard web app is the interface is confusing for non-programmers but can easily be taught with a 10-minute tutorial to employees using this interface.

4.5 System Development

The Following features outline the key design components integral for the flow of data from client to server, to database, back to server, and then back to client.

4.5.1 Methods

The Uniformed Nature between Client and Server detail how the client requests particular resources through the API. These requests are specified through pre-defined HTTP methods such as GET, POST, PUT and DELETE. The following rules should be considered when designing the requests:

Table 1: HTTP Methods. Adapted from (Gupta, 2022)

HTTP Method	Action
GET	Obtain information about a resource Should be idempotent – Multiple requests will produce the same result until the state of the system has been changed by a POST or PUT Request
POST	Create a New Resource Not Idempotent – Multiple Identical requests will result in two different resources.
PUT	Update a Resource API may create resource if it doesn't exist
DELETE	Delete a Resource

Note, some methods which obtain Database resources use a Post Method. This is to ensure that no scalability issues arise, i.e., maximum URL length. It can also be noted that sending data through GET requests expose queries to server logs. This may be undesirable by certain clients due to the nature of the data they are requesting.

4.5.2 Headers

HTTP Headers let the client and server pass additional information with a HTTP request or response. There are HTTP standard headers that should be included in all request and response headers. These are all present within our system and include: content-type, content-length, last-modified, ETag, and expiration.


```
▼ Request Headers
  Content-Type: "application/json"
  User-Agent: "PostmanRuntime/7.29.0"
  Accept: "*/*"
  Postman-Token: "90d17fa4-58a6-4569-a57a-141dfe6a8394"
  Host: "localhost:5000"
  Accept-Encoding: "gzip, deflate, br"
  Connection: "keep-alive"
  Content-Length: "62"
  Cookie: "jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyOTU0ZTk1YmYyYzNkMDAzNDhlZDVlZiIsInVzZXJuYW1lIjoiamFjd
  wOH0.YpGFxsmvT7NSmB6Ea3vYFDZIv0g6FdZDwcEwNkbHijA"
```


Figure 3 Example HTTP Request Header


```
▼ Response Headers
  X-Powered-By: "Express"
  Content-Type: "text/plain; charset=utf-8"
  Content-Length: "24"
  ETag: "W/"18-bmduU+Fgav4baWaPjk+nRlkYmOs""
  Date: "Sun, 05 Jun 2022 05:35:11 GMT"
  Connection: "keep-alive"
  Keep-Alive: "timeout=5"
```


Figure 4 Example HTTP Response Header

4.5.3 Routing

The REST API utilizes Express Routers to navigate the flow of data from the client to the relevant controller, which will gather the requested data, and return the required response.

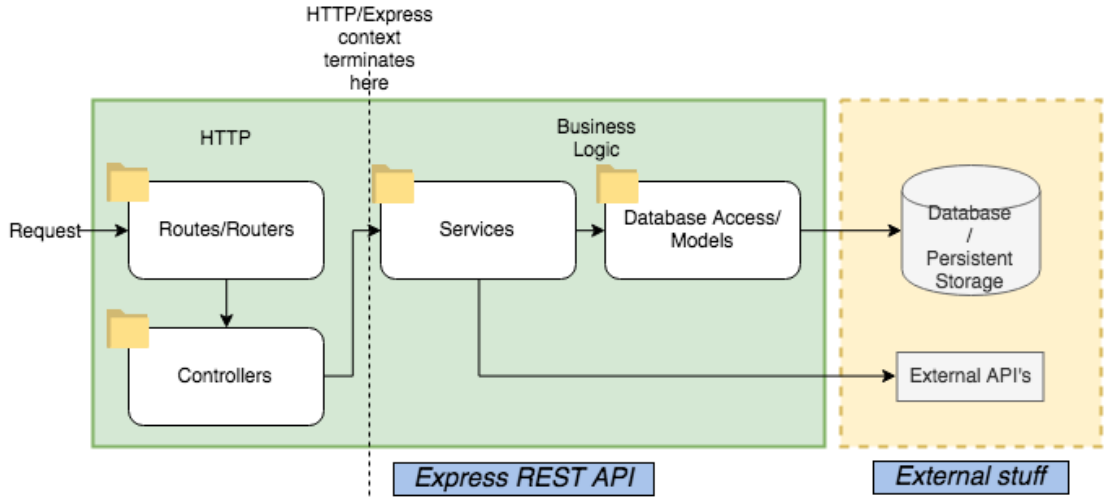

Figure 5 (API Architecture Diagram, 2022)

An Express route is declared within the API which associates a HTTP verb (GET, POST, PUT, DELETE), a URL path, and a function that performs the requested functionality. Essentially, it is the mechanism within the API which routes the HTTP request to the correct section of code to handle the request. An example of this can be shown in the figures below.

```
// Get ALL
router.route("/allInventoryItemType").get(userAuth, getAllItemType);
```

Figure 6: src/routes/route.js

src > models > JS itemtype.js > ...

```
1   const Mongoose = require("mongoose");
2
3
4   const ItemTypeSchema = new Mongoose.Schema({
5     _id: Mongoose.Types.ObjectId,
6     itemName: String,
7     itemSel: Number,
8     itemCost: Number,
9     supplier: String,
10  });
11
12  const ItemType = Mongoose.model("itemtype", ItemTypeSchema);
13
14  module.exports = ItemType;
15
```

Figure 7: src/models/itemtype.js

```
exports.getAllItemType = (req, res, next) => {
  try {
    const query = ItemType.find();
    // execute the query at a later time
    query.exec(function (err, result) {
      if (err) return handleError(err);
      var transresult = result.map(function(ItemType) {
          return ItemType.toJSON();
      });
      res.status(200).send(transresult);
    })

  } catch (error) {
      if (error instanceof Error) {
          res.status(500).send(error.message);
      } else {
          res.status(500).send('Unexpected Error');
      }
  }
};
```

Figure 8: src/controllers/posts.js

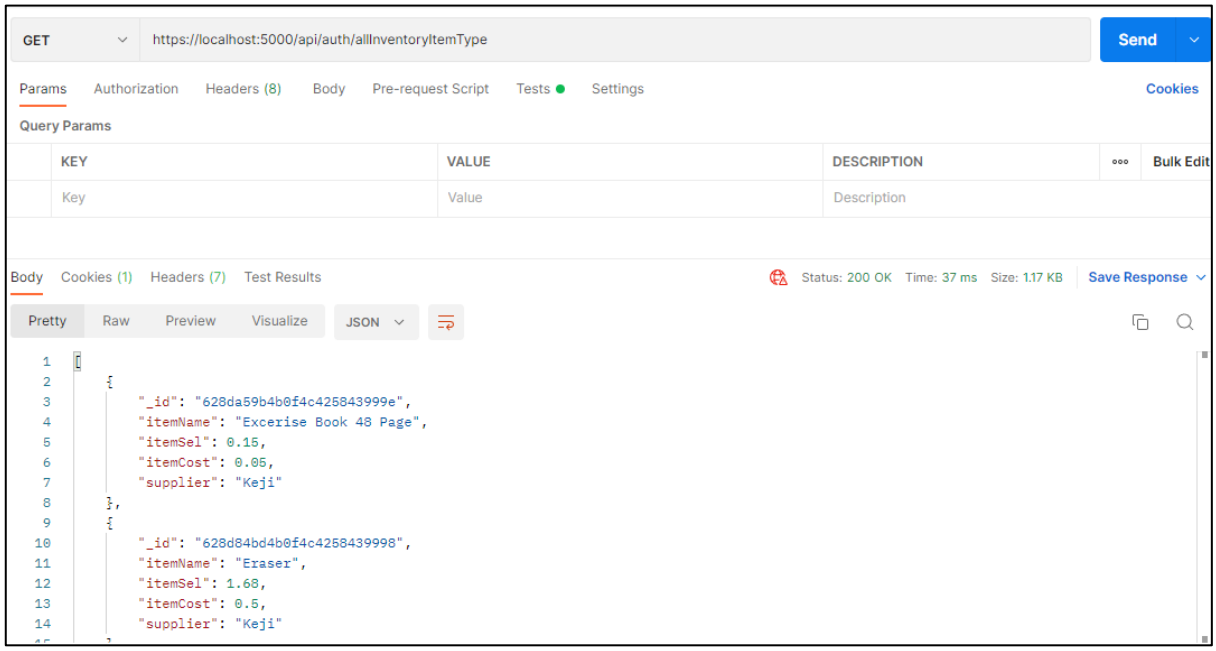

Figure 9: Postman Test

It is possible for parameters to be passed as URL parameters. However, all data is sent as a JSON object within the request body. An example of this can be seen in the figure below.

Figure 10: Example Postman Request/Response

It is important to note that each request is passed through the relevant security middleware's to ensure the client has the authorization to request such information. This is discussed further in the Security Considerations section.

4.5.4 Controllers

A detailed description of the available controllers can be found in the Application Documentation Swagger File. However, a brief overview is also detailed in the table below. A controller can be considered a service endpoint that the client is requesting. Each controller takes the request body, extracts the input data, and performs the relevant database queries to produce a response. If the controller successfully produces a result for the client, it sends the response under status 200. If the controller is unsuccessful in producing a meaningful result for the client, it sends a response under the description of a different error code. These are explained in detail in the next section.

Link to Swagger File (Live, Free Trial May Have Expired)

Postman Collections and Exported Swagger File can be found in Source Files or GitHub Link. https://github.com/BigRonnyColeman/Programming-For-Security

Authentication Required	Controller Name	HTTP Method	Route Path	Request Body	Response Body
None	Login	POST	https://localhost:5000/logout	JSON	JSON
None	Info	GET	https://localhost:5000/api/auth/getinfo		String
User	Register	POST	https://localhost:5000/api/auth/register	JSON	JSON
User	AllItemType	GET	https://localhost:5000/api/auth/allInventoryItemType		JSON Array
User	AllItem	GET	https://localhost:5000/api/auth/allInventoryItem		JSON Array
User	allBoxes	GET	https://localhost:5000/api/auth/allBoxes		JSON Array
User	ItemTypeByID	POST	https://localhost:5000/api/auth/ItemTypeByID	JSON	JSON
User	ItemByID	POST	https://localhost:5000/api/auth/ItemByID	JSON	JSON
User	BoxByID	POST	https://localhost:5000/api/auth/BoxByID	JSON	JSON
User	ItemTypeByName	POST	https://localhost:5000/api/auth/ItemTypeByName	JSON	JSON
User	ItemTypeBySupplier	POST	https://localhost:5000/api/auth/ItemTypeBySupplier	JSON	JSON
User	ItemTypeBySell	POST	https://localhost:5000/api/auth/ItemTypeBySell	JSON	JSON
User	ItemTypeByCost	POST	https://localhost:5000/api/auth/ItemTypeByCost	JSON	JSON
User	BoxesByItemType	POST	https://localhost:5000/api/auth/BoxesByItemType	JSON	JSON
User	LocationByItemType	POST	https://localhost:5000/api/auth/LocationByItemType	JSON	JSON
User	ItemCount	POST	https://localhost:5000/api/auth/ItemCount	JSON	JSON
User	ItemsByItemType	POST	https://localhost:5000/api/auth/ItemsByItemType	JSON	JSON
User	ItemsByBoxID	POST	https://localhost:5000/api/auth/ItemsByBoxID	JSON	JSON
User	AddItems	POST	https://localhost:5000/api/auth/AddItems	JSON	JSON
User	MoveItem	PUT	https://localhost:5000/api/auth/MoveItem	JSON	JSON
User	MoveBox	PUT	https://localhost:5000/api/auth/MoveBox	JSON	JSON
User	SellItem	POST	https://localhost:5000/api/auth/SellItem	JSON	JSON
User	UpdateRFID	POST	https://localhost:5000/api/auth/UpdateRFID	JSON	JSON
User	BoxByRFID	POST	https://localhost:5000/api/auth/BoxByRFID	JSON	JSON
Admin	updateUser	PUT	https://localhost:5000/api/auth/updateUser	JSON	JSON
Admin	deleteUser	DELETE	https://localhost:5000/api/auth/deleteUser	JSON	JSON
Admin	Users	GET	https://localhost:5000/api/auth/getUsers		JSON
Admin	deleteItemType	DELETE	https://localhost:5000/api/auth/deleteItemType	JSON	String
Admin	UpdateCost	PUT	https://localhost:5000/api/auth/UpdateCost	JSON	JSON
Admin	UpdateSell	PUT	https://localhost:5000/api/auth/UpdateSell	JSON	JSON
Admin	AddItemType	POST	https://localhost:5000/api/auth/AddItemType	JSON	JSON

4.5.5 Error Codes

Status codes are used to convey the status of the results of a request to the client. Error Codes should be sent as part of the response if the server does not successfully complete the service requested by the client. HTTP defines standard status codes which can be divided into five categories.

Table 2 HTTP Status Codes (Gupta, 2022)

1XX	Informational	Communicates Transfer Protocol-Level Information
2XX	Success	Indicates that the clients request was accepted successfully
3XX	Redirection	Indicates that the client must take some additional action in order to complete their request
4XX	Client Error	Indicates a client Error
5XX	Server Error	The Server takes responsibility for these error status codes

Specific examples implemented in our REST API are detailed in the table below.

Table 3 API Status Codes. Adapted from (Gupta, 2022)

HTTP Method	**CRUD**	**Collection Resource**	**Single Resource**
POST	Create	201 (Created),	Avoid using POST on a single resource
GET	Read	200 (OK). Use pagination, sorting, and filtering to navigate big lists	200 (OK), single user. 404 (Not Found), if ID not found or invalid
PUT	Update/Replace	405 (Method not allowed), unless you want to update every resource in the entire collection of resource	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid
PATCH	Partial Update/Modify	405 (Method not allowed), unless you want to modify the collection itself	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid
DELETE	Delete	405 (Method not allowed), unless you want to delete the whole collection — use with caution	200 (OK). 404 (Not Found), if ID not found or invalid

5 SECURITY CONSIDERATIONS

Web application vulnerabilities generally involve a system flaw or weakness embedded within the structural design of the underlying API architecture. These vulnerabilities are unlike other common types of vulnerabilities, such as network or asset, and generally arise because web applications require greater levels of accessibility, which often comes at the expense of lowered security. The sheer quantity of web application security breaches that regularly occur has been the primary cause for industry standards, such as the widely employed OWASP Top 10, to be created. For the purposes of this project, the OWASP Top 10 was used as the primary set of guidelines toward providing security against adverse actors. It must be noted that the primary means of addressing these matters was through trusted middleware libraries, which shouldn't be considered without insecurities themselves.

5.1 BROKEN OBJECT AND FUNCTION LEVEL AUTHORIZATION

With broken access object and function level control flaws, unauthenticated or unauthorized users may have access to sensitive files and systems, or user privilege settings. Configuration errors and insecure access control practices are hard to detect as automated processes cannot always test for them. This API manages access control using a sign-on issued JSON Web Token (JWT). Upon providing the correct credentials, each subsequent request made by the user will include the JWT in a cookie header, which details the user ID and privileges encoded using BASE64. This allows the user to access routes, services, and resources that are permitted with that token until it expires.

Figure 11: src/controllers/ post.js

Figure 12: src/routes/ route.js

5.2 BROKEN AUTHENTICATION AND SESSION MANAGEMENT

Incorrectly implemented authentication and session management calls can be a huge security risk. If attackers notice these vulnerabilities, they may be able to easily assume legitimate users' identities. Often, this works in conjunction with other components of the OWASP Top 10, such as injection, sensitive data exposure or broken access control. This API implements items such as minimum password length (6 characters), though could benefit from the added security of multi-factor authentication or added password complexity. Upon providing the correct credentials, the session is ultimately managed by the JWT issued to the user, which is a culmination of the user, the users session ID, and their privilege levels.

Figure 13: src/controllers/ post.js

5.3 SENSITIVE DATA EXPOSURE

Many APIs rely on data transmission methods and storage databases, which if not properly configured, can be exploited to gain access to usernames, passwords, and other sensitive information. To prevent sensitive data exposure, this API uses the hash algorithm 'BCRYPT' to store its passwords in the API database. BCRYPT has been considered the industry standard for password hashing for 17 years now, primarily due to its slow

Figure 14: src/controllers/ post.js

calculation speed- which limits the number of calculations per minute an attacker may perform. Additionally, all requests containing sensitive data are transmitted via the POST method and encrypted via HTTPS. To ensure that any sensitive global variables remain hidden, a .env file is used. It must be noted that when uploading this to a public forum such as GitHub, this mustn't be uploaded with its contents.

Figure 15: src/controllers/ post.js

```
exports.register = async (req, res, next) => {
  const { username, password } = req.body;
  if (password.length < 6) {
    return res.status(400).json({ message: "Password less than 6 characters" });
  }
  bcrypt.hash(password, 10).then(async (hash) => {
    await User.create({
      username,
      password: hash,
    })
```

```
_id: ObjectId("629549c0bf2c3d00348ed5e4")
username: "admin"
password: "$2a$10$7pk.e9jo.nvymiJKKGpW/.fndjZMJumdW1LVp.zY7M82PL3t8mqyq"
role: "admin"
__v: 0
```

Figure 16: User stored in DB

5.4 Lack of Resources and Rate Limiting

Lack of Resources and Rate Limiting is one of the greatest impacts to API server performance. If not handled correctly, these matters can allow for Denial of Service (DoS) and authentication flaws. This API uses a "rateLimiter" middleware library to handle this and is currently configured to accept 10 requests every 2 minutes from a single IP- anymore and the user will have to wait until the next 2-minute interval.

```
//Rate Limiting
const limiter = rateLimit({
  windowMs: 2 * 60 * 1000, // 15 minutes
  max: 10, // Limit each IP to 2 requests per `window` (here, per 15 minutes)
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers
  handler: function (req, res, /*next*/) {
    return res.status(429).json({
      error: 'You sent too many requests. Please wait a while then try again'
    })
  }
})
```

Figure 17: src/server.js

5.5 Mass Assignment

Binding client provided data (e.g., JSON) to data models, without proper properties filtering based on an allow list, usually leads to Mass Assignment. In this API, mass assignment is entirely governed by access control, authorization, and by ensuring that internal processes and matters remain internal.

Figure 18: src/controllers/ post.js

Particularly, admin privileges govern 2 primary things:

1. The manual updating of stock.
2. Admin privileges of other users

If this API was to be professionally deployed this, a situation where an admin user is compromised would have to be considered. To address this, perhaps this API could benefit from more than two layers of access control.

5.6 Security Misconfiguration

Just like misconfigured access controls, more general security configuration errors are huge risks that give attackers quick, easy access to sensitive data and site areas. Mitigating security misconfiguration is inherently difficult, as it is often a result of incompetence or unfamiliarity with the system. In the case of this API, this is perhaps the greatest cause for security concern due to our teams limited experience with the tools used.

5.7 INJECTION

Injection occurs when an attacker can insert their own code into a program because it is unable to determine code inserted in a certain way from its own code. If successful, attackers can use injection attacks to access secure areas and confidential information as though they are trusted users. Though NoSQL databases such as Mongo DB often offer performance and scaling benefits, these databases are highly vulnerable to injection attacks. In the example below, the attacker would be able to bypass the username and password requirement by injecting a JSON object that is true if not equal to 1 (true if the fields are incorrect).

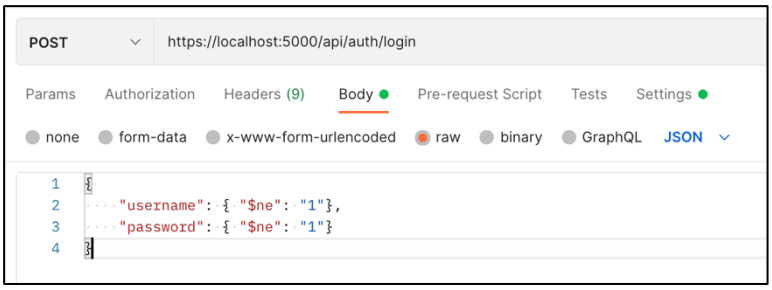

Figure 20: Postman NoSQL injection.

Figure 19: src/controllers/ post.js

```
exports.login = async (req, res, next) => {
  var { username, password } = req.body;
  //Cast to string to prevent object Injection (in conjunction with middleware)
  username = String(username);
  password = String(password);

  // Check if username and password is provided
  if (!username || !password) {
    return res.status(400).json({
      message: "Username or Password not present",
    });
  }

  try {
    const user = await User.findOne({ username });

    if (!user) {
      res.status(400).json({
        message: "Login not successful",
        error: "User not found",
      });
    } else {
      // comparing given password with hashed password
      bcrypt.compare(password, user.password).then(function (result) {
        if (result) {
          const maxAge = 3 * 60 * 60;
          const token = jwt.sign(
            { id: user._id, username, role: user.role },
            jwtSecret,
            {
              expiresIn: maxAge, // 3hrs in sec
            }
          );
          res.cookie("jwt", token, {
            httpOnly: true,
            maxAge: maxAge * 1000, // 3hrs in ms
          });
          res.status(201).json({
            message: "User successfully Logged in",
            user: user._id,
            role: user.role,
          });
        } else {
          res.status(400).json({
```

```
{
    "message": "Login not successful",
    "error": "User not found"
}
```

Figure 21: Unsuccessful NoSQL Injection

By this logic, an attack would be able to gain admin level privileges without any knowledge of the username or password. To account for this, the API implements the express-mongo-sanitise middleware library after first casting all body parameters to strings, so that they aren't read as objects. The added layer of security that the middleware provides will remove all "$" and "." characters from the body, parameter, header, and query requests.

To mitigate the risk of header malformities, the 'Helmet' API middleware library was implemented to protect all incoming and outbound request headers from attacks such as XSS. Helmet is a collection of several smaller middleware functions that set security-related HTTP response headers. Some examples include:

- helmet.contentSecurityPolicy which sets the Content-Security-Policy header. This primarily helps prevent cross-site scripting attacks among many other things.
- helmet.hsts which sets the Strict-Transport-Security header. This helps enforce secure (HTTPS) connections to the server.
- helmet.frameguard which sets the X-Frame-Options header. This provides clickjacking protection.

5.8 IMPROPER ASSET MANAGEMENT

This API utilizes modern principles and technologies. Particularly, it implements Node.js express with community developed middleware- rather than programming it.

5.9 Insufficient Logging and Monitoring

Failing to log errors or attacks and poor monitoring practices can introduce a human element to security risks. Threat actors count on a lack of monitoring and slower remediation times so that they can carry out their attacks before you have time to notice or react. This API records all requests and errors that occur on the server into a log file using the 'Winston' API middleware library. This includes login failures, access control failures, and server-side input validation failures so that any suspicious activity can be identified.

Figure 22: src / server.js

```
//Error logging (has to be after routing)
app.use(expressWinston.errorLogger({
  transports: [
    new winston.transports.Console()
  ],
  format: winston.format.combine(
    winston.format.colorize(),
    winston.format.json()
  )
}));
```

Figure 23: Winston logger middleware.


```
{"level":"\u001b[32minfo\u001b[39m","message":"POST /api/auth/login 400 12ms","meta":{"req":{"headers":
{"accept":"*/*","accept-encoding":"gzip, deflate, br","connection":"keep-alive","content-
length":"61","content-type":"application/
json","cookie":"_auth_verification=%7B%22nonce%22%3A%22jMl87LXP2-7GeAlGRkvvKxEL2KpH-XcvQ1o1-
OJohUo%22%2C%22state%22%3A%22eyJyZXR1cm5UbyI6Imh0dHA6Ly9sb2NhbGhvc3Q6ODA4MCJ9%22%7D.V1heherKjBPJRLVGFK0
OHVk1kuQFk7LVGrmnC4b-hiA;
jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyOTU0OWMwYmYyYzNkMDAzNDhlZDVlNCIsInVzZXJuYW1lIjoiYW
RtaW4iLCJyb2xlIjoiYWRtaW4iLCJpYXQiOjE2NTQ0NzkyNDIsImV4cCI6MTY1NDQ5MDA0Mn0.5qoVQPnGDcehuw3r_UXHVyLL92q6P
2UhhZUb389NKVg","host":"localhost:5000","postman-token":"7fd74e2e-4329-43c8-8306-235aac57a652","user-
agent":"PostmanRuntime/7.29.0"},"httpVersion":"1.1","method":"POST","originalUrl":"/api/auth/
login","query":{},"url":"/api/auth/login"},"res":{"statusCode":400},"responseTime":12}}
```


Figure 24: src/server.js

```
//HTTPS request Logging
app.use(expressWinston.logger({
  transports: [
    new winston.transports.File({
      // Create the log directory if it do
      filename: 'logs/example.log'
    })
  ],
  format: winston.format.combine(
    winston.format.colorize(),
    winston.format.json()
  ),
  meta: true, // optional: control whether you want to log the meta data about the request (default to true)
  msg: "HTTP {{req.method}} {{req.url}}", // optional: customize the default logging message. E.g. "{{res.statusCo
  expressFormat: true, // Use the default Express/morgan request formatting. Enabling this will override any msg i
  colorize: false, // Color the text and status code, using the Express/morgan color palette (text: gray, status: 
  ignoreRoute: function (req, res) { return false; } // optional: allows to skip some log messages based on reques
}));
```

6 REFERENCES

API Architecture Diagram. (2022). [Image]. Retrieved 7 May 2022, from https://www.coreycleary.me/_next/static/media/Express-REST-API-Struc.aa7ecaa0c41dbb7344c70665a5f5e259.png.

Farmer, K. (2021). *What is Postman, and Why Should I use it?.* Retrieved 6 May 2022, from https://www.digitalcrafts.com/blog/student-blog-what-postman-and-why-use-it Retrieved 6 June 2022, from

Gupta, L. (2022). *HTTP Methods - REST API Tutorial*. REST API Tutorial. Retrieved 7 June 2022, from https://restfulapi.net/http-methods/.

Gupta, L. (2022). *HTTP Status Codes - REST API Tutorial*. REST API Tutorial. Retrieved 7 June 2022, from https://restfulapi.net/http-status-codes/.
IBM. (2019). *MEAN Stack*. Retrieved 6 June 2019, from https://www.ibm.com/au-en/cloud/learn/mean-stack-explained

MongoDB. (2022). *Advantages of MongoDB*. Retrieved 6 June 2022, from https://www.mongodb.com/advantages-of-mongodb

MongoDB. (2022). *What is the MEAN Stack?.* Retrieved 22 May 2022, from https://www.mongodb.com/advantages-of-mongodb

OWASP. (2019). OWASP API Top 10 2019. Retrieved 3 May 22, from https://owasp.org/www-project-api-security/

Subramanian, H., & Raj, P. (2022). *Hands-On RESTful API Design Patterns and Best Practices* [Ebook]. Packt Publishing. Retrieved 7 June 2022, from https://learning.oreilly.com/library/view/hands-on-restful-api/9781788992664/.