- **Routing APIs** -> Routing refers to how an application's APIs respond to requests from an endpoint. Routing methods correspond to the HTTP methods (GET, PUT, POST, PATCH, DELETE), etc.
- In express, the **app** object is used to call routing methods. Eg - app.get() is used to handle GET requests, app.post() is used to handle post requests and so on.
- A routing method can specify a **handler function.** When the application receives a request at that endpoint, the actions in that handler function are executed.
- A routing method can chain several handler functions together. In order to execute them one after the other, the **next()** method must be called.
- The **app.all()** method is used to load middleware for all HTTP methods. The **next()** method must always be used in it.
- It is to be noted that all these methods take the **req, res** and **next** arguments. It is possible to get parameters from the **request body**, send a **response** and move on to the next method.
- The routing methods work like a waterfall. So when a request comes at the endpoint, then express keeps checking all the routing methods till it finds the one with the correct route. The handler function in that routing method is then executed.
- It is possible to use **wildcards** in the route paths.

The response methods in express can send a response to the client and terminate the request-response cycle.
The response methods are provided below:

Error Handling APIs -> Express catches and handles errors that occur both synchronously and asynchronously using a default error handler.

- Errors that occur inside synchronous code inside route handlers and middleware are automatically caught by express' default error handler. The errors caught inside asynchronous functions must be sent to **next()** for them to be handled by the default error handler.
- If anything other than the string 'route' is passed to the **next()** then it will regard that parameter as an error.
- Errors in asynchronous functions must be caught and given to Express. This can be done using the **try … catch** block (overhead) or **Promise**, else Express won't handle the error at all.
- The custom made error handler is placed at the end after all the other middleware.
- If an error is passed to next() and isn't handled in a custom error handler, then it will be written to the client as a stack trace.
- A custom error handler function takes 4 arguments (**req, res, err, next**).

JSON Web Token (JWT) ->
- A JWT is generated using a CSPRNG found in a library like **crypto**.
- It has 3 parts -> A base64 encoded header, the second part contains the data as a base64 encoded string and the 3rd part contains a signature to verify its integrity.
- The secrets for the Access Token and refresh token are generated using the crypto module. They are 64 bits long.

Authentication works as follows:

- We connect to the mongodb using mongoose.
- Login credentials are sent.
- The login credentials are verified.
- A signed JWT is sent by the server.
- This signed JWT usually contains the username, a user role and the access_token_secret used as a signature. This is the access token.
- This access token is sent to the user along with the role as JSON, it is stored in the application state. A new access token is issued every time the user logs in.

Refresh Token Issue:

- The Refresh Token is obtained from the HTTPOnly Cookie where it is stored.
- If the user is still logged in, then the username, user role, refresh token and the refresh token secret are verified using jwt.verify().

Authorisation:

- The access token is obtained from the authorisation header. It is decided and verified against the access token secret. The username and role are verified with those in the Access Token. If authorisation is successful, then the user is allowed to access the resource.

The APIs required for this are: **crypto** (to generate the token secrets), **dotenv** (to load the token secrets from .env), **jsonwebtoken** (to create and verify JWTs), **mongoose** (to access the user credentials stored in mongodb), **bcrypt** (to store the password as a hash), **cookie-parser** (to create the HTTPOnly Cookie).

Other APIs

User Registration -> This API will allow users to register with a username and a password. A user verification email is sent to the user with a link. On clicking the link, the user's account will be activated.

Implementation details:

- The **nodemail** package is needed to send the account verification email.
- We first set up an SMTP server.
- Generate the email verification link using the **rand**() function from the **math** module. This link has a domain name and an ID (generated by **rand**).
- Send the email using SMTP Transport to the user's email address.
- When the user clicks on the email verification link, we verify the domain and the ID. If they match, then the user's account is created by using **createOne()** of **mongoose**. The user is also assigned a profile of "user". The user's password is encrypted using **bcrypt**.

This API needs **mongoose** (to access the user credentials stored in mongodb)**, nodemailer** (to send the verification email)**, bcrypt** (to compare the entered password's hash with actual password) and **math** (to generate the ID).

Session Management (Part of Login and Refresh) -> A session is a period of time for which the user interacts with the application. All the user data and preferences saved by the user during the session are stored in the session variable. When the user logs out, the session variable is destroyed.

Implementation Details:

- We need to install the **express-session**, **cookie-session** and **session-file-store** packages.
- The **express-session** middleware creates a session. It is initialised with 3 configuration options; *secret* (set a secret key for the session), *resave* (Decides if the session should be saved to store on every request), *saveUninitialized* (Decides if uninitialized sessions should be saved).
- After initialising the session middleware, the session object must be created and stores user data and preferences.
- The session middleware stores session data in server memory, in a cookie or on Redis. When a user logs in, the session middleware creates a session object and assigns it a unique ID. The session middleware uses the session ID to retrieve the session data from the server or session store.
- The session timeout is set using *maxAge*, expressed in **ms**.
- The data is stored in the *req.session* object and can be retrieved by accessing it
- The session is destroyed using the *req.session.destroy()* method.
- We can secure session data by using secure cookies, encrypting session data, and implementing HTTPS encryption.

The APIs needed here are: **express-session** (to create a session), **cookie-session** and **session-file-store** (to save the session file).

Product API -> This API will allow administrators to do CRUD operations on products while it will allow customers to read information about products. The products will be of different categories.

Implementation Details:

- The products will be stored as documents on **MongoDB** and be interacted with using the **mongoose** ODM.
- The API will allow a customer to download 1 or more products from the database and as many for administrators.
- The API will allow **administrators** and **users** to search for products using either a product name or some keywords.

The APIs needed here are: **mongoose**.