Throughout the module, we will rely on [VSCode](https://code.visualstudio.com/download) and some of its plugins, which should also be helpful in your future code review and whitebox pentesting exercises. The main plugins that we will use are the following:

* [RapidAPI](https://marketplace.visualstudio.com/items?itemName=RapidAPI.vscode-rapidapi-client)
* [Docker for VScode](https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker)

The web application we will be testing throughout the module uses the MERN stack (Mongo, Express, React, Node). We wanted to avoid having you to go through the hustle of setting all of these tools and technologies on your own machine, and then reconfiguring them for each exercise.

So, we created a separate Docker image for each exercise that includes both the front-end and back-end codes for the web application, as well as setting up the Mongo database and everything else. This also allows easy debugging of the web application, and easy restart/reset in case anything goes wrong in our testing.

So, in addition to the above tools, please make sure that you have [Docker](https://docs.docker.com/engine/install/) setup and installed on your machine.

**Note:** Dockerizing web applications is an important skill to learn, so you can use this opportunity to learn how it's done and use it in your future whitebox pentesting exercises. Another option for larger web applications is to set up a test VM replica of the production server, though this usually takes a lot of effort and isn't easily deployable elsewhere.

**Running Locally**

Once you have the above tools set up and running, you can download the provided zip archive at the end of this section, extract its content, and then open the folder using VScode, using File>Open Folder, or through the terminal with code ./validation\_logic\_disparity.

Once that's done, you can open the Dockerfile within VSCode, right click on the opened window, and select Build Image. This will prompt you to enter a tag for the image, so you can either keep the default value of the folder's name, or use a common host for all exercises under this module (e.g. application\_logic\_flaws:validation\_logic\_disparity). Once you do so, the docker image will start building, which should take 5-15 minutes depending on your machine and internet speeds.

A screenshot of a computer

Description automatically generated

Once the build is done, you can find the new image in the Docker icon in the VSCode Sidebar on the right, under IMAGES. You can right-click on and select Run, and the image should start up. Give it a few seconds to load up everything, and then visit http://localhost:5000/ with your browser, and login with the credentials provided in the Dockerfile, as that user may have some privileges like cubes or payment cards added under their user. Some tests and exercises may require you to create your own user, so that's another option.

A screen shot of a computer

Description automatically generated

**Tip:** You may also select Run Interactively to keep an eye on the logs, like the back-end server error messages, which will show up in a window within VSCode.

**Local Testing**

As part of the whitebox pentesting process, we usually need to perform local testing and debugging, as mentioned in the previous section. Luckily, the Docker images we just setup also enable easy debugging of our code. All we need to do is go to the Run and Debug tab in the VSCode sidebar, and click on the Run icon next to the text Docker: Attach to Node. As the Docker image is already configured for debugging, this will attach to our web application for debugging, and the bottom bar of VSCode should now be red to indicate we are in debug mode. You may also pause, restart, and detach from the debug session using the hovering session buttons.

A screen shot of a computer

Description automatically generated

In the coming sections, we will go through more details on how to utilize debugging in our local testing. But as an example, we can add breakpoints by opening the file we want to add a breakpoint in, and go to any line and click on the line number, and this should add a breakpoint. Now, whenever the code reaches this line, the application should break and stop on that point, so we can examine everything and then resume its execution.

**Note:** It is important to note that even though we can set breakpoints using the code files opened in VSCode, any code changes will not be reflected in the running container. Though this isn't something we will need to do in this module, if you ever need to modify code in a running container, go to the Docker tab, find the running image under CONTAINERS, then you can click on it to expand it and show its files. You can find the web application files under /app, and right-click and select Open to open and modify any of them within VSCode.

Another thing we may need to modify and debug throughout our local testing is the database. To do so, we can go to the Docker tab in VSCode, find our running image under CONTAINERS, right-click on it and select Attach Shell, and this should open a new window terminal in VSCode and drop us into a shell within the running container. Now, we can simply run the mongosh command, and should have a MongoDB shell with access to our database.

A screen shot of a computer

Description automatically generated

**Note:** This module assumes that you are already familiar with MongoDB and how to run basic commands, as well as understand the basic usage of MongoDB with NodeJS. If you are not, you may give it a quick read online to familiarize yourself with it, or check out the [Introduction to NoSQL Injection](https://academy.hackthebox.com/module/details/171) module, which introduces MongoDB and gives a brief about how to use it.

**Application Structure**

Before we move on to our first topic, it is worthwhile to take a quick look on the general structure of the application we will be testing. We can start by opening src/src/app.js, and going through it.

We will see that the application starts by importing some files and libraries. After that, the application sets up Node/Express along with other configurations, as well as setting up bodyparser and CORS.

Code: js

const app = express();

const port = parseInt(process.env.PORT || "5000");

const \_\_filename = fileURLToPath(import.meta.url);

const \_\_dirname = path.dirname(\_\_filename);

// set up body parser and cors

app.use(bodyParser.json());

app.use(cors());

// serve public assets

app.use(express.static(path.resolve(\_\_dirname, "public")));

Then, the application sets up the main API middleware with app.use() and specifies the routes file for each route. Then, it handles 404 and general app errors.

Code: js

// set up API routes

app.use("/api/auth", authRoutes);

<SNIP>

// forward all other requests to react app, so react router can handle them 'will handle 404 errors'

app.use((req, res, next) => {

res.sendFile(path.resolve(\_\_dirname, "public", "index.html"));

});

// handle 404 errors

app.use((req, res, next) => {

res.status(404).json({

message: "Could not find this route.",

});

});

// handle next() errors and general errors

app.use((error, req, res, next) => {

<SNIP>

});

Finally, it sets up the MongoDB connection and starts listening on the port specified at the beginning. We can read the comments to get a better idea of exactly what each code block does.

Code: js

// start the Express server & db connection

set("strictQuery", false);

connect(process.env.DB\_URL ?? "", {

dbName: process.env.DB\_NAME,

user: process.env.DB\_USER,

pass: process.env.DB\_PASS,

})

.then(() => {

<SNIP>

});

So, the main thing that we need to focus on is the APIs and their routes middleware. We can CMD/CTRL click on any of them, like authRoutes, and VSCode should open it in its corresponding file.

Code: js

const router = express.Router();

// secure private routes for content (use req.user in private controllers)

router.use(verifyToken);

router.get("/update\_token", updateUserToken);

If we examine the routes middleware, we will see that it is linking each sub-route to a specific function, like /update\_token to updateUserToken, which we can also CMD/CTRL click on to further review, as we will do in upcoming sections.

Take some time to navigate through the code and familiarize yourself of how it's running. In general, the files we will be most interested in are under /controllers, as well as /routes and /models.

With that, you should be able to run this application and debug it, as well as have a general idea of its general structure. In the next section, we will start with our first topic and will go through the code to identify potential issues and logic bugs.