**Local Testing (Schemas) - Null Safety**

With a list of endpoints with optional parameters, we can start reviewing the use of these optional parameters, and see how they are handled by the endpoint, as they may have null safety issues. If we do identify any null safety issues, we can move to the more difficult part, which is trying to see how we may take advantage of such flaws.

**UserSchema**

In the previous section, we identified optional parameters in two different schemas: passwordResetSchema and UserSchema. All of the uses of both schemas are found in the users-controllers.js file under /controllers/. We will start with the more interesting/sensitive of the two schemas, which is UserSchema.

As mentioned in the previous section, this schema has four optional parameters: id, name, username, registrationDate, and is used to validate user input in 4 different endpoints: createUser, login, updateUserDetails, and requestPasswordResetLink. So, let's go through each of these, and see how it is handling potential null variables.

We will start with createUser. The first thing we notice is that it performs null checks for the name and username variables:

Code: js

// name/username are not set as required in the schema, so we need to check for them here

if (!name || !username) {

if (!name) {

<SNIP>

} else {

<SNIP>

}

}

This raises the question: Why weren't these parameters set as .required() in the schema instead of null-checking them here? The most likely answer is that these fields may not be required in all endpoints that use the UserSchema for input validation; hence, they are set as optional.

However, this is a bad implementation, as it may lead to null safety issues, and the better implementation would be to use conditional requirements for the schema. This means that we set a certain condition under which each parameter would be considered optional or required, like having the useUsername toggle, for example, which may set some of the parameters as optional. Another option would be to use a different schema for each use case, although this is not always ideal or efficient, given the amount of duplication it will cause.

In addition to the above, we have already established that the use of the not operator (!) has some pitfalls in JavaScript. For example, if we set the name to be 0, then the above condition would be triggered, even though we did not set it to null, and setting it to 1 or "0" or any other string would work, as demonstrated below:

A screenshot of a computer

Description automatically generated

While this is obvious in this case, as a name should never be "0", some other parameters that expect numeric values may face the same issue, like amount in the example we saw in the unexpected input sections. So, it is also advisable to use a specific pattern when validating each parameter, wherever possible, to avoid such edge cases. Furthermore, in case we need to perform local null checks, we should avoid using the not operator (!), and stick to strict equality (e.g. if (name === null || name === undefined)).

Other than name and username, the id and registrationDate are automatically generated and not passed as user input, so their usage is safe in this case. If we move to the login endpoint, we see that it accepts only two user inputs: email and password, both of which are required by the schema, so they are safe as well.

Next, we have updateUserDetails, which accepts 3 user input parameters: name, username, email, two of which are set as optional by the schema (name and username). The endpoint uses the schema validation to test the user input:

Code: js

const errors = await validateUserDetails({

email,

password: process.env.VALIDATION\_TEST\_PASSWORD,

name,

username,

});

if (errors) {

return next(errors);

}

We see that the code uses a pre-set random password for the required password field, since it is a required field, but is not used in this endpoint. This is another example of a bad schema design, as it is using a workaround to pass the required parameters, even though it does not need to. Once again, this is another case where conditional requirement must be used, which should set the password parameter as optional if the usePassword toggle is set to false.

Other than that, we see that both name and username are being used later on to update user details in the database, even though they are set as optional parameters by the schema, and we don't even see the endpoint checking whether they are null, like the createUser endpoint did above:

Code: js

const updateReq = await User.findByIdAndUpdate(

// use id from token to ensure users can only update their own account

req.user?.id,

{

email,

name,

username,

},

{

returnOriginal: false,

}

);

This is definitely an issue, as we can store null values in the database, and if anywhere in the code it retrieves these values while not expecting null values, it may lead to a run-time error or a process crash. We will further test this later on.

Finally, this schema is used in requestPasswordResetLink, which only accepts one parameter (email), and this parameter is marked as .required() in the schema, and may be considered safe as well.

**PasswordResetSchema**

Moving on to the second schema. Unlike the previous schema, this one is only used once within the same endpoint it is defined in. We have previously noticed that all of its parameters are set as optional, even though the entire schema is set as required. This means that we need to provide at least one of the parameters in order to pass the schema validation test, but it does not matter which parameter we choose to use.

Let's try to see where each of the parameters is being used in the endpoint. We see that the id parameter is being used to identify the user, and then to calculate the secret password-reset token:

Code: js

// retrieve user based on id

try {

user = await User.findById(id);

<SNIP>

}

<SNIP>

// generate password reset token based on password hash and user id

const hashedToken = md5(`${id}:${user?.password}`);

As for the token, it is mainly being used to validate against the secret token, as calculated above:

Code: js

// verify password reset token based on password hash and user id

// if not valid, return error

if (token && token !== hashedToken) {

return next({

message: "Invalid password reset token.",

statusCode: 403,

});

}

The password is simply used to modify the password once everything has been verified. So, the application basically accepts the user id and the secret token, then dynamically compares the sent token with the one it calculates based on secret user details stored in the databases (mainly the old password hash).

Most modern applications generate a random and long secret token upon a password reset request, and then store it in the database with an expiration date, to ensure the token cannot be guessed and that it expires on time. Still, some smaller applications use this dynamic approach to avoid storing tokens on the back-end, and simply calculate it on run-time using secret information that users cannot guess. Even though the first approach is recommended, this approach is not necessarily insecure, so we will not consider it as a security risk.

However, is this the only issue with the code? The main issue that got us here is that all of these parameters are considered optional. So, what would happen if any of them is set to null? This is what we will test in the next section.