**Code Review (Null Variables) - Null Safety**

Let's start by trying to identify variables that can have a null value at some stage. These generally occur in two ways:

1. Null variables: Declaring a nullable variable (e.g. let const;) without an initial value
2. Optional parameters: When a function has a parameter that is optional with no default value (i.e. can be set to null)

The key point for us is to identify ones that may have null safety issues, and then filter those depending on whether we have control over them. At this stage of the module, we are mainly interested in input that we have a direct/indirect control over.

**JavaScript Common Null Pitfalls**

Before we start identifying potential null issues with our JavaScript application, we need to learn a bit more about common pitfalls of this language. Unfortunately, JavaScript is considered to be one of the worst languages when it comes to null safety. In addition to allowing nullable variables, JavaScript also supports three different types that do not hold a value, which may lead to run-time errors. They are:

1. null
2. undefined
3. NaN

This makes checking for null variables quite tricky in JavaScript, and can often lead to unexpected outcomes. As showcased in [this article](https://javascript.plainenglish.io/there-is-a-bug-in-javascript-since-day-one-typeof-null-9b18da349cc6), if we check the type of null in JavaScript, we would get an empty object, while undefined would simply return undefined. Furthermore, running typeof on NaN returns a number, even though NaN literally means Not-a-Number!

On top of all of that, JavaScript mostly doesn't even produce a run-time error when dealing with a null value, and simply continues its execution as if the variable is assigned a value, due to the flawed implementation of [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) in JavaScript. We saw an example of this in the unexpected input section, when the application continue execution with the total value being NaN, and returned it to us in a message.

Finally, JavaScript sometimes even parses null values to a type (e.g. int), which can completely change the intended execution logic. For example, if an int variable is being used while having a null value, then JavaScript would simply turn it to 0. This can lead to many types of logic bugs, like comparing the null variable (now 0) with >=, which would result in true, as we will see later on. Another example is if it attempts to obtain the cost of an item from the database and fails to do so, then it may consider the cost of the item to be 0, which could allow malicious users to trick it to buy items for free.

As we are dealing with JavaScript in this module, it will be very interesting to evaluate our code against all of these potential null issues, so let's see how we can go about identifying any of them.

**Flawed Null Checks**

[This article](https://synergycodes.com/blog/how-to-do-your-null-checks-safely-in-javascript) also explains some cases where checking for null variables may not always work as expected. For example, if we only checked if a value is undefined, and the variable was actually null, then the condition (user === undefined) would be false, as null and undefined have different characteristics, as mentioned above. So, the application must check against both null and undefined using strict equality === for both; otherwise, the test may fail. The same thing applies to NaN.

Let's take the below code as an example:

Code: js

const { count } = req.body;

if (!count) {

throw new Error();

}

The code tries to obtain the value of count from the POST request body, and if this value was not sent (i.e. !count), then it would throw an error. However, since JavaScript considers 0 to be equal to false and 1 to be equal to true, if the sent value of count was 0 or '0' "as a string", then the code would still throw an error, even though the value was not null! This is because JavaScript would evaluate (!0) to be if not false (i.e. if true), which would pass the if condition.

This shows that even the use of the not operator (!) is not safe, and we should instead use !== or === and check against both null and undefined.

**Identifying Null Variables**

Let's start with the simplest case, and get a list of variables that are declared without an initial value. In JavaScript this is usually done with the use of let or var, as using const requires setting a value (since it's constant and can't be changed). So, we can use the VSCode regex search with the pattern (let|var) [A-Za-z]\*; and also limit it to the src/src/controllers/\*\* directory, and it should locate all such variables: A screen shot of a computer program

Description automatically generated

As we can see above, the search yielded 42 results found in 9 different files. The first result, as we can see in the screenshot above, attempts to set the value of user to the output of User.findById(userId), and if no output is returned (i.e. it is still null), it throws an error and stops execution:

Code: js

if (!user) {

throw new Error();

}

All errors in this web application are passed to the next() function, which is whatever function passed to the current function as the next parameter, as seen below:

Code: js

export async function createUserToken(userId, next)

If we backtrack this next function, we will see that it was passed down from the login() function, which eventually leads to the Express app.use function on line 68 in app.js. This basically returns an HTTP error to the user with the error string provided by the next/catch function. Once it detects a null, it'll throw an error, which will stop execution and return whatever error was thrown to the user.

**Exercise:** Try to review the other results from above using the same process we just did, and see if any of them skips the null check "e.g. if (!user)". If you find any, then it can be shortlisted as a potential null safety issue.

Going through the rest of the results, we will notice that the application is generally consistent in checking for null variables before proceeding with execution. However, it is always validating that a variable is not null through the use of if (!variable), which, as has been discussed earlier, can lead to unexpected outcomes in some cases, which may end up causing a logic bug.

We will not focus on these, as their values are usually derived from database functions or other functions, which means that it is unlikely to return a null value, as then the database would return an error and cause the catch clause to throw an error and stop execution. Still, it may be worth keeping those as a last resort, as there may be some cases where they would cause a logic bug.

**Note:** In this module, we are trying to cover different approaches and potential ways to identify the logic bugs we are discussing. While our exercises may not suffer from all of them, it is still important to learn these approaches/techniques, as well as learn the way of thinking when identifying such bugs, as they may be found in other applications you test.