JavaScript Part 9: Modules, Error Handling and Debugging

Programs grow, and so do their problems. Four steps keep a growing program healthy: split, guard, inspect and fix. Each step answers one simple question, and the free developer tools that every browser includes are enough to practise them. In this part, Reem, a software engineer on an invoicing team, uses them to find out why the total of some invoices shows NaN.

1. Split

A module is a JavaScript file that keeps its own variables private and shares only what it chooses to export. Another file brings those parts in with an import statement, naming what it needs and the file it comes from. In a web page we load the main file with a script tag whose type is module, and the page should be served from a web address, because browsers do not load modules from files opened directly. The question is: what belongs together, and what should other files be able to use? Small modules with one clear job are easier to test and reuse, and they cannot overwrite each other's variables by accident. A module that exports only a few well-named functions gives the rest of the program a simple surface to use.

Example: A company keeps separate drawers for contracts, invoices and letters, and every drawer has a label. Nothing from the contracts drawer ends up among the letters by accident. Staff open only the drawer they need and take out one folder. This is how modules work: each file keeps its own variables private and shares only the functions it exports.

At work: Reem's invoicing code lives in one very long file. She asks the Split question and separates it into three modules: one for tax rules, one for formatting amounts and one for building the invoice. Each module exports only its main functions, and the invoice module imports the other two. Colleagues can now change the tax rules without touching the layout of the invoice.

2. Guard

Things will go wrong at runtime, so good code decides in advance what to do. The try block holds code that might fail, and the catch block receives the error object when it does, with a name and a message. The finally block runs in every case, which makes it the place for cleanup. We can also raise our own error, for example `throw new Error("Line without a price")`, which creates an error object with a clear message. The question is: what could go wrong here, and what should happen when it does? A catch block that ignores the error hides the problem and is worse than no handling at all. Errors should be caught where we know how to respond, and a friendly message should reach the user.

Example: A pharmacist checks every prescription for a missing dose before preparing it. When something is missing, the pharmacist calls the doctor instead of guessing, so the customer gets a clear answer instead of a wrong medicine. The check takes a few seconds, while a wrong guess could cost a lot. This is the guard idea: expect the failure, and decide in advance what to do when it happens.

At work: Reem's total function now checks every line of the invoice. She asks the Guard question and throws an error with the message line without a price when a price is missing. The code that builds the invoice catches that error and shows the accountant a clear message instead of a total of NaN. A finally block hides the loading indicator in every case.

3. Inspect

To fix a problem, we must first see it. The console.log function prints a value, and console.table prints an array of objects as a table, which is ideal for the lines of an invoice. The Console panel of the browser

This document is a property of Asaseeyat Academy - © Copy Right Protected.

developer tools shows errors with the file name and the line number, and the stack shows which functions were running. In the Sources panel we can set a breakpoint on a line, so that the program pauses there, and then we can read the value of every variable and step forward one line at a time. The debugger statement pauses the program in the same way when the developer tools are open. The question is: what is the code really doing, compared with what we believe it does? Looking at the actual values ends the guessing.

Example: A car technician does not replace parts at random but connects a diagnostic tool and reads what the engine reports, for example which part is failing. Then the repair is aimed at the real cause, not at a guess. A breakpoint gives us the same connection to code, because it pauses the program on a line and lets us read every value. That is why looking at the actual values ends the guessing.

At work: Reem prints the lines of the invoice with console.table and sees that one line has no price at all. She asks the Inspect question and sets a breakpoint in the total function to watch the values line by line. The stack in the error message shows which functions led to the failing line. Within minutes she knows which line breaks the total.

4. Fix

Fixing starts with reproducing the problem every time, using the smallest example that fails. Reading the error message carefully often gives the answer: a ReferenceError means that a name is not defined, a TypeError means that a value is not of the type the code expected, such as reading a property of undefined, and a SyntaxError means that the code is not valid, so the file does not run at all. Then we form a guess about the cause, change one thing and test again. The question is: what is the smallest change that fixes the cause, not only the symptom? After the fix, we repeat the test that failed and keep it as a check for the future. Changing several things at once makes it impossible to know which change helped.

Example: A doctor first asks when the pain started and what makes it worse, then tests one idea at a time. Giving three medicines together would hide which one worked, so nobody would know what to repeat next time. In the same way, a programmer reproduces the problem, changes one thing and tests again. This is why we change one thing at a time.

At work: Reem reproduces the problem with one invoice that has a single line. The error message and the table show that a property name was spelled differently in the data, so the price was undefined and the arithmetic produced NaN. She asks the Fix question and corrects the data mapping where the name is created, rather than hiding NaN at the end. She keeps the failing invoice as a permanent test, so the problem cannot return unnoticed.

Conclusion

Reem's invoicing code is now split into clear modules, guarded against missing data and easy to inspect. Split asks what belongs together and what others may use, and guard asks what could go wrong and what should happen then. Inspect asks what the code is really doing, and fix asks for the smallest change that removes the cause. In the next part, we leave the browser and run JavaScript with Node.js and the npm package manager.

Sources

- MDN Web Docs: JavaScript modules – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
- MDN Web Docs: try...catch statement – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch
- MDN Web Docs: Error – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error
- Chrome for Developers: Debug JavaScript – https://developer.chrome.com/docs/devtools/javascript

This document is a property of Asaseeyat Academy - © Copy Right Protected.