**WDD 330 Web Frontend Development II**
**Samuel Palacios**
**Week 05**
**May 20th, 2022**

# Readings Assignment Notes

## 1. Ch10: Testing and Debugging

- ✓ We have software that is compiled and another that is interpreted. Both the compiler and the interpreter analyze the source code and check for lexical, syntactic, and semantic errors. On that account, when a program passes compiler or interpreter validations, it would be expected to work correctly, deterministically (**same input, same output).** However, the real world is not simple, and when an application runs there are different types of errors or exceptions.

| | |
|---|---|
| **System Errors** | It is a failure or error caused by factors external to the application, for example a failure of the software or hardware of the server where the application resides, a network failure, or a misconfigured firewall, etc. |
| **Programmer Errors** | In the case of interpreted programming, this type of error is more common, or it may be a programming that is lexically, syntactically, and semantically correct, but does not comply with the business rules because it was not done according to the design, that is an error, of programming. |
| **User Error** | They are errors that the user commits in data entry and selection of options, many of them will happen because the software allows them, the application allows them, that is why it is important to enter validations and adequate control points that help the user to reduce the number of mistakes you will make. |
| **Exceptions** | An exception is an error in real time, this type of exception can be caused by any of the types of error mentioned above that occurs at some point in time, for example if someone is going to make an electronic payment and at that |

| | |
|---|---|
| | precise moment If there is a communication failure, an exception will be generated (which can be controlled by the application, if it is not controlled the application can even close) because there is no communication. |
| **Stack Trace** | When an error or an exception occurs, the detail of said error or exception can be seen in a stack of traces of all the code that failed those reports all the active elements in the stack of the interpreter or execution engine that failed and the detail of the cause why they failed, this helps the programmers to debug the code or to know what could have happened. |
| **Warnings** | These are precautionary elements, they are not errors, but they warn us of one. For example, if there is a bad programming practice but, in some contexts, it needs to be done, there may be a warning, but it still won't be an error and the code will work. This type of warning can be silenced or ignored, although in many cases it will be useful. |

- ✓ After understanding what errors and exceptions are, we understand that **testing** and **debugging** is something we must do if we want the software application, we make to be successful in a productive environment. The issue of debugging is clear, when there is a scenario not considered and the application throws an exception, it will be necessary to review the logs and do debugging tests until it is corrected; but when one or more users receive errors or experience problems, the preference or confidence in the tool can drop, in some cases dramatically...then it becomes necessary to test, to test to prevent this from happening and to reduce the probability of experiencing problems already in a productive environment.

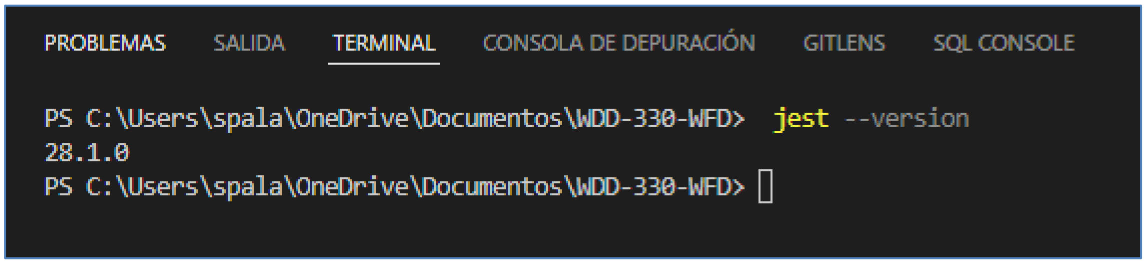
- ✓ **"Strict mode"** was created in JavaScript to make applications more robust by forcing the programmer to follow best practices and disallow semantic errors such as using the wrong data type switch or making incorrect declarations. Without this mode statement, the JavaScript interpreter ignores several things that the programmer might do wrong and that are historically allowed and ignored by the JavaScript interpreter. Therefore, it is recommended to use it to ensure that there will be no syntactic errors and above all semantics.

✓ When testing, one of the objectives is to detect failures. Failures can occur in any part of the flow of an application, from when it is started until any of the users use it or even finish using it. **Debugging** can be done in different ways:

| | |
|---|---|
| **Displaying Alerts** | This is a simple way, it is to be able to display the error or exception in a message within the application, this is quite practical debugging, although it is not recommended for productive environments since it is not comfortable for the user to see the exceptions on the screen. |
| **Using the Console** | All browsers have a console where you can display output that is not visible to the user except by opening this console, it is very similar to the command line when you open a program from there. In JavaScript you get output to there with the "console.log()" statement. |
| **Instancing Error Objects** | At least in JavaScript, it is possible to work with the "error" object, which allows us to obtain better information about the error, such as the complete message and the stack trace, which is valuable information for debugging. |
| **Throwing Exceptions** | There is also a way to throw exceptions, which are checkpoints at which it is not recommended that the program sequence continue, so the code is stopped by throwing an exception. In JavaScript it is done via the "throw" statement. In some languages, doing this forces exception handling with try, catch, and finally. |
| **Exception Handling** | Related to the previous one, when exceptions are thrown or occur at runtime, it is good practice to handle those exceptions to tell the user what is happening in a better way or to give a better treatment. For this, the "try-catch-finally" blocks are used, which allow the code inside *try* to be executed and if an exception is thrown, it can be analyzed in *catch*, and regardless of whether the exception is thrown or not, execute in *finally*. a definitive action. |
| **Browser Debugger** | There is a live debugger in every browser which allows you to set breakpoints and see the different values of variables, you can also use the "debugger" directive to do this and see the results in the console and browser developer tools like Google Chrome and Firefox. |

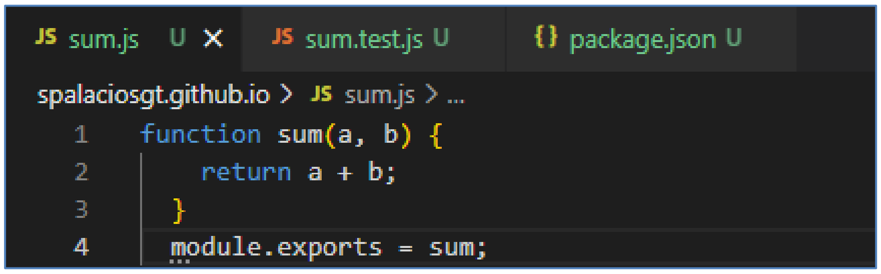
✓ The development of tests is something that has boomed in recent years. The purpose is very clear, if the software passes 100% of the tests, it is guaranteed that it will have a certain stability, a certain predictable operation; then, when there are applications whose source code is huge and a small change is made, it is valuable to know if after that change said software will still pass 100% of the tests. From this account, there are currently software development methodologies where even the tests are developed before the functionalities, that is, the development is directed by the tests as it happens in **DDT** (Test-Driven Development).

✓ Usually, to develop tests, a library or framework is used that allows them to be carried out. There are unit, integration, and automated tests. In JavaScript, a well-known framework to do them is **"Jest".** The tests are simple, an action is carried out and it is verified that the results are adequate, for example a sum function if 1 and 2 are sent to it, it should give 3 as a result to "assert", otherwise it would give a "fail" .

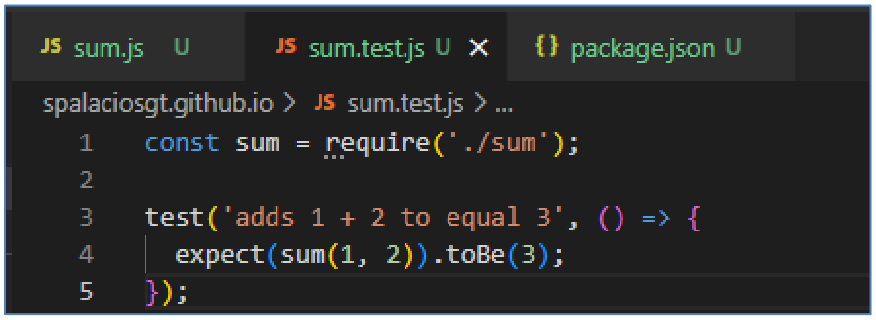


```
PS C:\Users\spala\OneDrive\Documentos\WDD-330-WFD>  jest --version
28.1.0
PS C:\Users\spala\OneDrive\Documentos\WDD-330-WFD> 
```

✓ The following is a small exercise of creating a test in **JavaScript** with **Jest.**

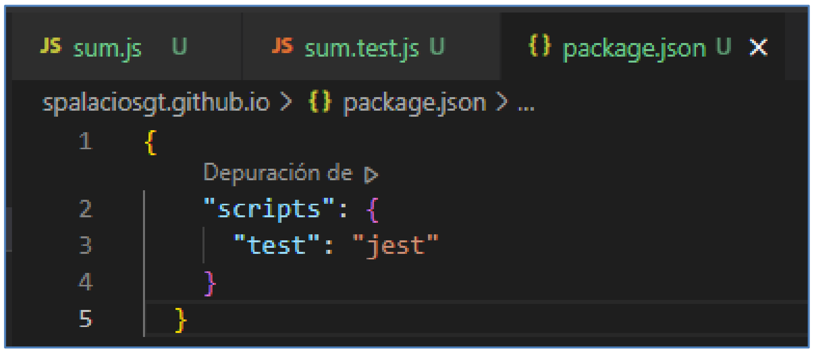


```
function sum(a, b) {
    return a + b;
  }
  module.exports = sum;
```

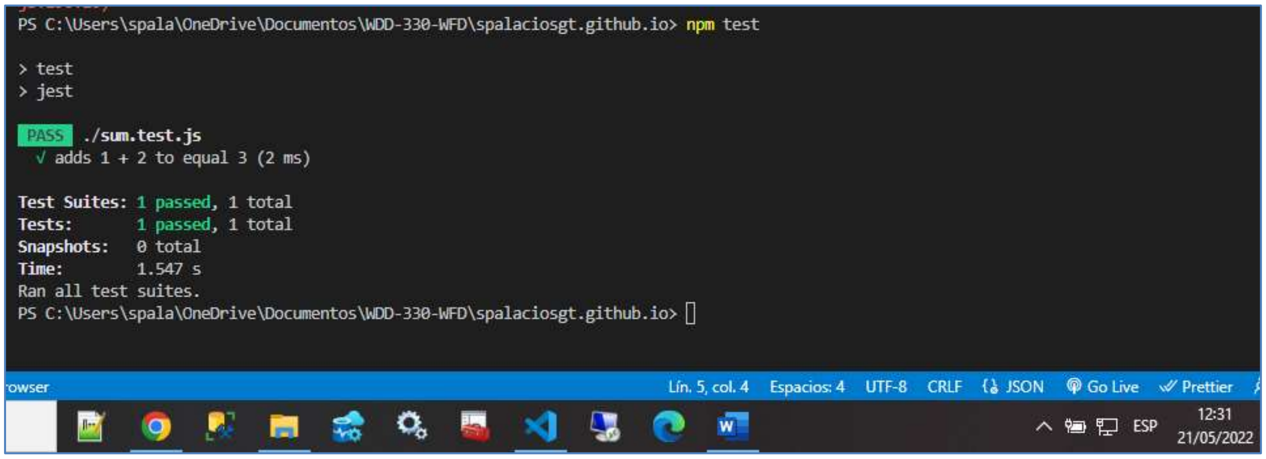


```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```



```
{
    "scripts": {
      "test": "jest"
    }
  }
```

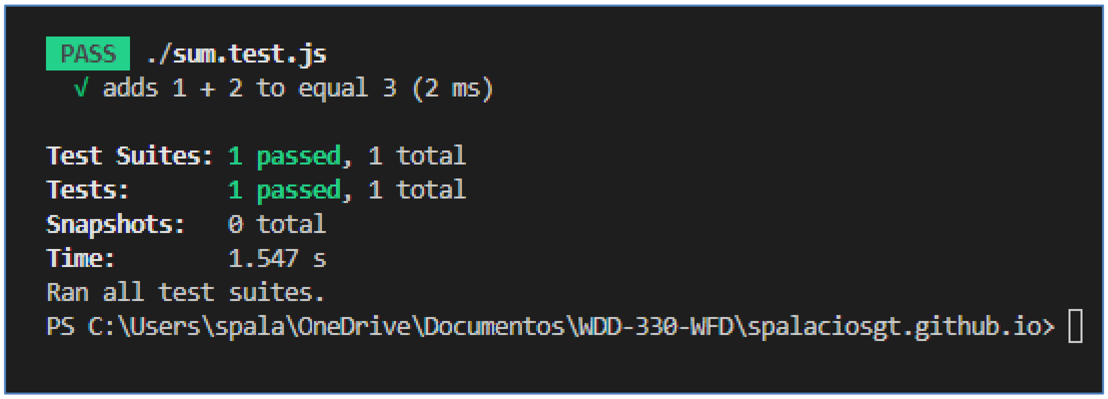


```
PS C:\Users\spala\OneDrive\Documentos\WDD-330-WFD\spalaciosgt.github.io> npm test

> test
> jest

 PASS  ./sum.test.js
  √ adds 1 + 2 to equal 3 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.547 s
Ran all test suites.
PS C:\Users\spala\OneDrive\Documentos\WDD-330-WFD\spalaciosgt.github.io> 
```



```
 PASS  ./sum.test.js
  √ adds 1 + 2 to equal 3 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.547 s
Ran all test suites.
PS C:\Users\spala\OneDrive\Documentos\WDD-330-WFD\spalaciosgt.github.io> 
```