# Tests

# Why Do We Need Tests?

**First of all we want to validate the results of our code.**

We want to make sure our code returns the correct results in all cases (including edge cases).

Tests helps us achieve that.

# Test Across Development

## Test Types

1. Unit test
2. Integration test
3. E2E tests (End to End)

# Test Across development

**Unit Testing**
Unit testing is the practice of testing small pieces of code, typically individual functions.

**Integration Testing**
The idea is to test how parts of the system work together – the integration of the parts.

**E2E (End to End) Testing**
Testing of complete functionality of some application. In practice with web apps, this means using some tool to automate a browser, which is then used to click around on the pages to test the application. Combines both client side and server side.

# Unit Testing

## Description

**What?**
A unit tests should essentially just give the function that's tested some inputs, and then check that the function outputs is correct.

**When** should you use unit testing?
Ideally all the time.

**Why?**
A good set of unit tests do not only prevent bugs, but also improve your code design, and make sure you can later refactor your code without everything completely breaking apart.

# Unit Testing

## Characteristic

**Small**

Should test a small thing, a function or a class

**Isolated**

Unit tests should be independent of other units. This is typically achieved by mocking the dependencies. If your test uses some external resource, like the network or a database, it’s not a unit test.

# Let's See an example

**Your Task** - Write a function that accepts a string from a user, and returns how many vowels appear in a string. For this questions, vowels are { a, e, i, o ,u }

**For example:** Aabbuu -> a appears twice, u appears twice, so in total we have 4 vowels.

## Is This a Good Solution?

```
function count_vowels(str) {
   let counter;
   for (let i = 0; i < str.length; i++) {
      if (['a', 'e', 'i', 'o', 'u'].indexOf(str.charAt(i)) !== -1) {
         counter++;
      }
   }
   return counter;
}
```

# Is Our Code Working?

## How can we know?

We can run the function with different values, and make sure that the result that we get is what we expected to get.

A very simple option will be to run the function and print the result. For example if we run the function `count_vowels` with the parameter "cool", we should get 2.

```
console.log(count_vowels("cool"));
```

Let's run it!

Oops!

```
> console.log(count_vowels("cool"));
  NaN
```

Not what we wanted…

# Test Case

So tests helps us find bugs!
And make sure our code is valid.

We tested only one value.
Each scenario we want to test, is called a **Test Case**.

What other test cases should we have?

# Test Cases

1. Regular cases (average/standard input)

```
let str1 = "apple"; //contains vowels and non-vowels
```

2. Typical cases (input that relate to the specific problem)

```
let str2 = "aeiou";//contains all the vowels
let str3 = "e";//1 vowel
let str4 = "r";//1 non-vowel
let str5 = "eek";//2 similar vowels – duplicated values
```

3. Edge cases (problematic inputs, rare cases)

```
let str6 = "";//empty string
let str7 = "rtgykm mmmm llkk";//no vowels
```

# Code Structure

How should we write all these tests?

```
function count_vowels(str) {
   let counter;
   for (let i = 0; i < str.length; i++) {
      if (['a', 'e', 'i', 'o', 'u'].indexOf(str.charAt(i)) !== -1) {
         counter++;
      }
   }
   return counter;
}

console.log(count_vowels("apple"));
console.log(count_vowels("aeiou"));
console.log(count_vowels("e"));
console.log(count_vowels("r"));
console.log(count_vowels("eek"));
console.log(count_vowels(""));
console.log(count_vowels("rtgykm mmmm llkk"));
```

# Test Mechanism

## There are several problems with that approach:

1. It makes our code dirty. We have a lot of dev/debug code in the middle of a file.

2. It is very tiring to do manual tests. Because we print the result of each test, we need to know (how?) what is the expected result and check that it is what we got.

3. Not scalable. Writing 7 tests is ok, what if we had to write 7000?

# Tests Mechanism

**Conclusion #1:**
We want our tests to be separate from the "production code" (the code that will reach our end user).

Solution:
Write the tests in a separate file

**Conclusion #2:**
We want our tests to be automatic

Solution:
Use a method that will check if the test pass, and only notify us of failed tests

# Assert

## A simple solution

Will be to use the browser's native console.assert
Which writes an error message to the console if the assertion is false. If the assertion is true, nothing happens.

## Example

```
console.assert(1 + 3 === 4, "1+3 is not 4");
```

# Assert Example

If we execute:

```
console.assert(1 + 3 === 4, "1+3 is not 4");
```

The expression `1 + 3 === 4` will be evaluated, and since it is truthy, nothing will happen.

If we execute:

```
console.assert(1 + 3 === 0, "1+3 is not 0");
```

The expression `1 + 3 === 0` will be evaluated, and since it is falsie, an error will be thrown with the error message.

Assertion failed: 1+3 is not 0

# Console.assert Problems

console.assert is very intuitive and simple.

But it has one major disadvantage:

Once one assertion has failed the program will stop executing the code.
If we try to run:

```
console.assert(1 + 3 === 0, "1+3 is not 0");
console.assert(1 + 3 === 2, "1+3 is not 2");
```

We will get only one error thrown.
Although we would like to know a more detailed status:
How many tests failed and which ones.

# Console.assert Problems

For that reason (and many more) developers developed test frameworks and test utilities.
Tests are now a **standard in the industry**.




Every company that respect itself uses tests in the development cycle.

# Node JS, nice to meet you.

Node.js is an engine to run JS outside of the browser.

Initially developed by Google and based on the Chrome engine.

It has numerous types of usages. In our case we will use it to run our tests!

# Tests Frameworks

Tests frameworks helps us implement test methodology that will help us write tests with the best practices!
There are a lot of tests frameworks and utilities:

Jest

Jasmine

We will learn

MOCHA

chai

**For that you will need to install NodeJS**

node JS

# Node JS

## Installing node JS

Go to node js site and install it. http://nodejs.org

After installing it verify it was installed properly:

1. Open your cmd (command line)
2. Type: `$ node -v`

You should get the version you have installed.
For example:

```
$ node -v
v10.15.1
```

Verify npm was successfully installed:
Type: `$ npm -v`

# NPM

## What is NPM?

**NPM** is a package manager which allows us to download packages (external libraries) directly to our project folder via the command line.

NPM is installed automatically when installing Node JS.

To install a package:

```
$ npm install <package_name>
```

When installing an npm package, npm will place it in your node_modules folder (it will create one if necessary).

# NPM

## What is NPM?

The best practice of working with NPM is to create a “package.json” file.

Calling the command:

```
$ npm init
```

After calling the command, npm will ask us a few questions about our project. We can just click ENTER to use the default answers.

# npm init

MINGW64:/c/matan/new-project

```
matan@TALEVIN MINGW64 /c/matan/new-project
$
```

Link to view

# Node JS

## Installing Mocha

Now we can install Mocha.

cd to your dev folder and run:

```
$ npm install mocha
```

# Test framework functionalities

## What do we need from a test framework?

- A better assert function.
- A mechanism to tell us which test failed.
- We also want to group our tests according to relevant topics.
  (In our example we might want to group the tests according to regular, typical and edge cases.)
- We would like to describe our tests to explain what they test.

# Mocha – first test

## Let’s see a simple test.

Let’s test the native js function `indexOf` :

```
let assert = require('assert’);

it('should return the index of the found element', function(){
   assert([1,2].indexOf(2) === 1, "index of 2 should be 1");
});
```

**Fetching node's assert function.**
**Using commonJS syntax for dependencies management**

**An it function (from Mocha) to define the test**

**The assert function.**
**Similar to console.assert**

# The it Function

Let's start with defining the test.
For that mocha offers us to use an `it` **function**:

## Syntax

```
it(test_description, tests_function);
```

## Example

```
it('should return -1 when value is not present', function(){
  //test code
});
```

# Running Tests in the Console

We can run the test in the cmd:
* note: we need to be in the project folder.

```
$ ./node_modules/mocha/bin/mocha <path/to/tests>
```

**Path to mocha. Should be inside your node_modules folder, inside your current project.**

For example:

```
Lotem@DESKTOP-AVN63ER MINGW64 /d/gitProjects/itc/april18 (master)
$ ./node_modules/mocha/bin/mocha lectures/tests/example.1.js


  √ should return the index of the found element

  1 passing (0ms)
```

**Mocha gives us a nice and clear summery of the tests**

We ran 1 test and it passed.

# Mocha – first test

## Let's add a failing test to see how it looks

```
let assert = require('assert’);

it('should return the index of the found element', function(){
   assert([1,2].indexOf(2) === 1, "index of 2 should be 1");
});

it('should return -1 when value is not present', function(){
   let msg = "error with return value for a non existing value";
   assert([1].indexOf(3) === null, msg);
});
```

And now run it in the console

# Failed Tests Info

```
Lotem@DESKTOP-AVN63ER MINGW64 /d/gitProjects/itc/april18 (master)
$ ./node_modules/mocha/bin/mocha lectures/tests/example.1.js


  √ should return the index of the found element
  1) should return -1 when value is not present

  1 passing (16ms)
  1 failing

  1) should return -1 when value is not present:

      AssertionError [ERR_ASSERTION]: error with return value for an unexisting value
      + expected - actual

      -false
      +true

      at Context.<anonymous> (lectures\tests\example.1.js:9:5)
```

**Failed**

**Failed tests description**

**What happened?**
**Why did the test failed?**

We can see that 1 test passed and 1 test failed.
We can see the failing test description and the assertion error.

# Chai

Now that we have a failed test we need to fix it.
In order to do that we would like to know 2 things:

1. What we got => we call it **actual**
2. What we should have got => **expected**

Chai will help us get an informative error message.

Chai is available through npm, so you can install it with:

```
$ npm install chai
```

And then import it in your code:

```
let expect = require('chai').expect;
let assert = require('chai').assert;
```

# Chai assert.equal

Now we can use chai:
Let's write an erroneous function with a test:

```
let assert = require('chai').assert;

function add(a,b){
   return a * b;
}

it('should return the sum of 2 numbers', function(){
   assert.equal(add(1,2), 3 , "sum of 1,2");
});
```

**Using a specified method `assert.equal` help us get a detailed error .**

**Syntax:**
`assert.equal(actual, expected)`

And run it in the console.

# Chai failing test results

```
1) should return the sum of 2 numbers

0 passing (31ms)
1 failing

1) should return the sum of 2 numbers:

    sum of 1,2
    + expected - actual

    -2
    +3

    at Context.<anonymous> (lectures\tests\chai-example.js:8:12)
```

We can see here in detailed the actual and the expected values.

# Chai assert.expect

Chai has another syntax that is also popular in other test libraries:

```
let expect = require('chai').expect;

function add(a,b){
   return a * b;
}

it('should return the sum of 2 numbers', function(){
  expect(add(1,2)).to.equal(3);
});
```

# More Chai Syntax

With Chai we can check many other things:

```
it('should return a number', function(){
   assert.typeOf(add(1,2), "number");
});

it('should return a number', function(){
   expect(add(1,2)).to.be.a("number");
});

it('should have length of the 2 concatenated arrays', function(){
   expect([1].concat([2])).to.have.lengthOf(2);
});

it('should have have length property', function(){
   expect([1]).to.have.property("length");
});
```

**Type checking**

**Length checking**

**Check for a property**

# Let's implement all our test cases

1. Regular cases (average/standard input)

```
let str1 = "apple"; //contains vowels and non-vowels
```

2. Typical cases (input that relate to the specific problem)

```
let str2 = "aeiou";//contains all the vowels
let str3 = "e";//1 vowel
let str4 = "r";//1 non-vowel
let str5 = "eek";//2 similar vowels – duplicated values
```

3. Edge cases (problematic inputs, rare cases)

```
let str6 = "";//empty string
let str7 = "rtgykm mmmm llkk";//no vowels
```

# Count vowels test cases

```
it('should return vowel count in a mixed string', function(){
    expect(count_vowels("apple")).to.equal(2);
});

it('should return 1 in a 1 vowel string', function(){
    expect(count_vowels("e")).to.equal(1);
});

it('should return 0 in a 1 non-vowel string', function(){
    expect(count_vowels("r")).to.equal(0);
});

it('should return 2 in a duplicated vowel string', function(){
    expect(count_vowels("cheek")).to.equal(2);
});
it('should count all the vowels', function(){
    expect(count_vowels("aeiou")).to.equal(5);
});
it('should return 0 for the empty string', function(){
    expect(count_vowels("")).to.equal(0);
});
it('should return 0 for a string with no vowels', function(){
    expect(count_vowels("rtgykm mmmm llkk")).to.equal(0);
});
```

# Arranging Tests

Now all the tests are one after the other without any structure.

The more we write tests the more we need a way to arrange them, and group them according to themes.

For that we have the describe function.
To keep them

# The Describe function

Mocha offers us to use a `describe` function:

## Syntax

```
describe(message, tests_function);
```

## Example

```
describe("Test functionality", function(){
   //test code
});
```

With the describe function we can arrange our tests according to topics. Inside each describe we can put as many `it`'s as we want.

# Tests nested inside a describe container

```
describe("Typical cases", function(){

    it('should return 1 in a 1 vowel string', function(){
       expect(count_vowels("e")).to.equal(1);
    });
    it('should return 0 in a 1 non-vowel string', function(){
       expect(count_vowels("r")).to.equal(0);
    });
    it('should return 2 in a duplicated vowel string', function(){
       expect(count_vowels("cheek")).to.equal(2);
    });
    it('should count all the vowels', function(){
       expect(count_vowels("aeiou")).to.equal(5);
    });

});
```

# We can have many describe blocks

```
describe("Regular cases", function(){
    it('should return vowel count in a mixed string', function(){
       expect(count_vowels("apple")).to.equal(2);
    });
});

describe("Edge cases", function(){
    it('should return 0 for the empty string', function(){
       expect(count_vowels("")).to.equal(0);
    });

    it('should return 0 for a string with no vowels', function(){
       expect(count_vowels("rtgykm mmmm llkk")).to.equal(0);
    });
});
```

# Nested Describes

```
describe("Vowel Couter", function () {
  describe("Typical cases", function () {
    describe("1 vowel", function () {
      it('should return 1 in a 1 vowel string', function () {
         expect(count_vowels("e")).to.equal(1);
      });
      it('should return 1 in a string that begins with a vowel',function(){
        expect(count_vowels("egg")).to.equal(1);
      });
      it('should return 1 in a string that ends with a vowel',function () {
        expect(count_vowels("pre")).to.equal(1);
      });
    });

    it('should return 0 in a 1 non-vowel string', function () {
      expect(count_vowels("r")).to.equal(0);
    });
  });

  describe("Regular cases", function () {});

  describe("Edge cases", function () {});
});
```

Level 1

Level 2

Level 3

Level 3

Level 2

Level 2

# Arranging the Tests

After running in the console we get a nice printing with the hierarchy:

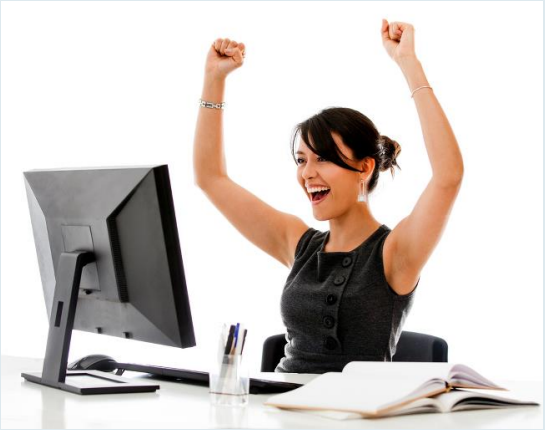
```
Vowel Couter
  Typical cases
    √ should return 0 in a 1 non-vowel string
    √ should return 2 in a duplicated vowel string
    √ should count all the vowels
    1 vowel
      √ should return 1 in a 1 vowel string
      √ should return 1 in a string that begins with a vowel
      √ should return 1 in a string that ends with a vowel
  Regular cases
    √ should return vowel count in a mixed string
  Edge cases
    √ should return 0 for the empty string
    √ should return 0 for a string with no vowels
```

# Test to help us during refactors

Tests are very helpful when we refactor code.
[* We can define refactor as changing the code implementation, without changing the behavior.]

**Have you ever ?**
Made a small change to a function and broke you app?

We already defined the expected behavior with the tests and after the refactor we can run the tests to make sure we didn’t break anything.



# Tests as documentation

Tests also use as documentation.

Tests are examples of how to use a function and what is the expected outcome.

The tests should specify how the function should behave in all cases, even edge cases.

# Test Structure

## Keep it clean and organized

Each test has a structure:

1. Arrange – setup all of the required variables for the function to work
2. Act - test (activate function)
3. Assert – validate the result: assert, expect…

# Test Structure

## Example

Let's say we wrote a function to find family members.
The function `findFamilyMembers(people, lname)`
Receives an array of persons:

```
class Person {
  constructor(fname, lname, age) {
     this.fname = fname;
     this.lname = lname;
     this.age = age
  }
}
```

and a last name, and returns an array with all the people with the specified last name.

```
findFamilyMembers([new Person("Yosi", "Banai", 72)], "Banai");
// Person { fname: “Yosi”, lname: “Banai”, age: 72 }
```

# Test Structure

## Let's look at a test structure

```
it('should return 1 family member', function () {
  // Arrange
  let people = [
      new Person("Yosi", "Banai", 72),
      new Person("Yehudit", "Ravitz", 58),
      new Person("Riki", "Gal", 57),
  ];

  let expectedFamilyMember = people[0];

  // Act
  let familyMembers = findFamilyMembers(people, "Banai");

  // Assert
  expect(familyMembers[0]).to.equal(expectedFamilyMember);
});
```

Setup

We want to keep the data separated

Set expected result

Test

Validations

# Configure tests to run with npm

In the package.json file we created earlier we need to add the following command in the scripts category:

"mocha <path/to/tests>"

for example:

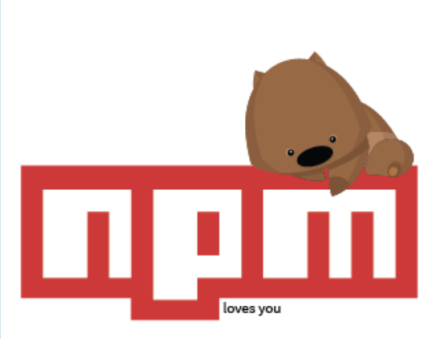
```
"scripts": {
   "test": "mocha app/tests"
},
```

Command name

Command to run

Afterwards, in the command line run:

```
$ npm test
```




**With npm we can configure our own commands!**

# Example

## package.json

```
{
  "name": "itc-bootcamp",
  "version": "0.0.0",
  "scripts": {
    "test": "mocha tests"
  },
  "dependencies": {
    "chai": "^4.2.0",
    "mocha": "^5.2.0"
  }
}
```

Out test folder which is called “tests”

# Mocha + Chai Cheat Sheet

## Run test in command line:

```
$ ./node_modules/mocha/bin/mocha <path/to/tests>
```

## Install instruction

```
$ npm install mocha
$ npm install chai
```

## Import chai dependencies

```
let expect = require('chai').expect
let assert = require('chai').assert
```

## Run tests with npm

```
Add to the package.json file:
  "scripts": {
     "test": "mocha app/tests"
  }
```

## In the command line:

```
$ npm test
```

## Type checking with assert

```
describe("Typical cases", function(){
  it('should return a number', function(){
     assert.typeOf(add(1,2), "number");
  });
});
```

## Type checking with expect

```
it('should return a number', function(){
   expect(add(1,2)).to.be.a("number");
});
```

## Check length

```
it('should have length', function(){
expect([1].concat([2])).to.have.lengthOf(2);
});
```

## Check if property exists

```
it('should have length property',
function(){
   expect([1]).to.have.property("length");
});
```