Function rest parameter -> …args turns args into an array
The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

`map()` creates a new array from calling a function for every array element.

```
const numbers = [4, 9, 16, 25];
const newArr = numbers.map(Math.sqrt)
```

MockImplementation capture the original args from the original function

jest.spyOn() -> track original function with its parameters
.mockImplementation(originalFunctArgs -> {})

```
const mockFn = jest.fn(scalar => 42 + scalar);
mockFn(0); // 42
mockFn(1); // 43

mockFn.mockImplementation(scalar => 36 + scalar);

mockFn(2); // 38
mockFn(3); // 39
```

jest. fn(): Creates a new mock function from scratch for when you don't need the original implementation. jest. spyOn(): Observes and tracks calls to an existing function, allowing you to keep or override the actual implementation

What is innerhtml in jest?

2 purposes

- Strip out of only the outer html layers, not nested layers
- Convert the node object into a string
- So you can use toContain, because toContain is more for string

`toContain` in Jest is primarily used for checking if a string or array contains a specific substring or element, respectively.

```
  console.log(typeof tableHeaders[0]); //object
  console.log(typeof tableHeaders[0].innerHTML); //string
      expect(tableHeaders[0].innerHTML).toContain(
        MyMessage.header.download(),
      );
```

How to select table header and row as node list?

What is a node list?
https://www.w3schools.com/jsref/dom_obj_html_nodelist.asp

```
expect(agentImageTable).toBeDefined();


  const tableHeaders = agentImageTable.querySelectorAll("thead tr th");
```

```
  const MyMessage = Messages.macs.agentImages;
  expect(tableHeaders[0].innerHTML).toContain(MyMessage.header.download());

  expect(tableHeaders[2].innerHTML).not.toContain(
    MyMessage.header.packageArchitectureType(),
  );
  expect(tableHeaders[3].innerHTML).not.toContain(MyMessage.header.version());
  const tableRows = Array.from(
    agentImageTable.querySelectorAll("tbody tr td"),
  );
  console.log("type", typeof agentImageTable.querySelectorAll("tbody tr td"));

  console.log(
    "table",
    agentImageTable.querySelectorAll("thead tr th")[0].innerHTML,
  );
  expect(
    tableRows[0].getElementsByClassName("oui-margin-small-right"),
  ).toBeDefined();
```

How to load the dom into browser?

```
import { screen } from "@testing-library/dom";
```

```
screen.logTestingPlaygroundURL() to launch dom in browser;
```

https://blog.logrocket.com/using-react-testing-library-debug-method/

RequireActual import the actual implementation of a module, rather than its mocked version

In Jest, `requireActual` is a method used in conjunction with Jest's mocking functionality. Jest is a popular JavaScript testing framework developed by Facebook. When writing tests, it's common to mock dependencies, such as external libraries or modules, to isolate the code being tested. Mocking allows you to replace actual dependencies with fake implementations, making it easier to control and test the behavior of the code under different conditions.

Sometimes, however, you may want to partially mock a module, meaning you want to mock certain parts of it but keep other parts intact. This is where `requireActual` comes into play.

`requireActual` is used to import the actual (original) implementation of a module, rather than its mocked version. This is useful when you want to mock only certain functions or properties of a module, while keeping the rest of the module unchanged.

Here's an example of how `requireActual` can be used in Jest:

javascript
```
// myModule.js
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}
```

javascript
```
// myModule.test.js
jest.mock('./myModule', () => {
  const actualModule = jest.requireActual('./myModule');
  return {
    ...actualModule,
    add: jest.fn((a, b) => 42) // Mocking the add function
  };
});

const { add, subtract } = require('./myModule');

test('add function is mocked', () => {
  expect(add(1, 2)).toBe(42); // This will use the mocked add function
});
```

```
test('subtract function works as usual', () => {
  expect(subtract(5, 3)).toBe(2); // This will use the actual subtract function
});
```

In this example, the `add` function is mocked to always return `42`, while the `subtract` function remains unchanged. The `requireActual` function is used to import the actual implementation of `myModule` so that we can override only the `add` function while keeping the rest of the module intact for testing.

Set up file

Math.test.js
Or math.spec.js will be executed by jest or vitest

Basic testing

3A - arrange, act, assert

```
export function add(numbers) {
 let sum = 0;

 for (const number of numbers) {
   sum += number;
 }
 return sum;
}
```

```
import { expect, it } from "vitest";
import { add } from "./math";
//3A - arrange, act, assert
it("adds up number", () => {
 //arrange
 const numbers = [1, 2, 3];
 //acc, curr, initial value
```

```
 const expectedResult = numbers.reduce((acc, cur) => acc + curr, 0);
 //act
 const result = add(numbers);
 //assert
 expect(result).toBe(expectedResult);
});
```

You can use test to capture errors in your code

```
export function add(numbers) {
 let sum = 0;

 for (const number of numbers) {
   //force number conversion
   sum += +number;
 }
 return sum;
}
```

```
import { expect, it } from "vitest";
import { add } from "./math";
//3A - arrange, act, assert
it("adds up number", () => {
 //arrange
 const numbers = [1, 2, 3];
 //acc, curr, initial value
 const expectedResult = numbers.reduce((acc, cur) => acc + cur, 0);
 //act
 const result = add(numbers);
 //assert
 expect(result).toBe(expectedResult);
});

it("should yield nan if one invalid number is provided", () => {
 const inputs = ["invalid", 1];
 const result = add(inputs);
 expect(result).toBeNaN();
});
```

```
it("should yield correct sum if an array of num string values is provided", () => {
 const numbers = ["1", "2"];
 const result = add(numbers);
 //convert string to number
 const expectedResult = numbers.reduce((acc, cur) => +acc + +cur, 0);
 expect(result).toBe(expectedResult);
});
```

USe tothrow to check error of your wrapper function

```
it("should yield 0, if an empty array is provided", () => {
 const numbers = [];

 const result = add(numbers);
 expect(result).toBe(0);
});

it("should throw an error if no input is passed", () => {
 //wrapper function for the function
 const resultFn = () => {
   add();
 };
 //test for wrapper function
 //toThrow will execute resultFn
 expect(resultFn).toThrow();
});
```

```
it("should throw an error if provided with multiple para", () => {
 const num1 = 1;
 const num2 = 2;
 const resultFn = () => {
   add(num1, num2);
 };
 expect(resultFn).toThrow();
});
```

Test error message of toThrow

```
it("should throw an error if provided with multiple para", () => {
 const num1 = 1;
 const num2 = 2;
 const resultFn = () => {
   add(num1, num2);
 };

 //how check the right error, error message, instance of class, etc
 //error is because numbers are not iterable like an array
 expect(resultFn).toThrow(/is not iterable/);
});
```

toBeTypeOf and toBeNAN();

```
export function transformToNumber(value) {
 return +value;
}
```

```
import { expect, it } from "vitest";
import { transformToNumber } from "./numbers";
it("convert stirng to value", () => {
 const input = "3";
 //   const expectedResult = +input;
 expect(transformToNumber(input)).toBeTypeOf("number");
});

it("convert stirng to value", () => {
 const input = "string";
 //   const expectedResult = +input;
 expect(transformToNumber(input)).toBeNaN();
});
```

You should only test your own code, you do not test third party codes

We can also change our code, not 3 party codes

A Word About Code Coverage

An important aspect of testing is to achieve good **code coverage**. This means, that you want to write tests for the majority of your code (both code files and line of code).

There are tools that help you measure your code coverage - actually Vitest comes with a built-in functionality:
https://vitest.dev/guide/features.html#coverage

It is worth noting though, that the goal is not necessarily 100% coverage. There always can be some code that doesn't need any tests (e.g., because it merely calls other functions that are tested already).

In addition, achieving (close to) full code coverage also isn't any guarantee that you wrote good tests.

- You could cover 100% of your code with meaningless tests after all.
- -Or you could missing important tests (that should test important behaviors). The code would still technically be covered by tests in such scenarios.

So don't see a high amount of code coverage as the ultimate goal!

Test call back function with done (async code with call back)

```
import jwt from 'jsonwebtoken';

//jwt.sign invoke doneF when it is done
export function generateToken(userEmail, doneFn) {
 jwt.sign({ email: userEmail }, 'secret123', doneFn);
}
```

```
import { expect, it } from "vitest";

import { generateToken } from "./async-example";

it("generate a token with done", (done) => {
 const testEmail = "test@gmail.com";
```

```
 //jest will not wait for call back function to finish, async
 generateToken(testEmail, (err, token) => {
   //so will not assert here
   //so we need below structure to do it otherwise will have timeout
   try {
     expect(token).toBe(2);
     done();
   } catch (err) {
     done(err);
   }
 });
});
```

Test promise

```
export function generateTokenPromise(userEmail) {
 const promise = new Promise((resolve, reject) => {
   jwt.sign({ email: userEmail }, 'secret123', (error, token) => {
     if (error) {
       reject(error);
     } else {
       resolve(token);
     }
   });
 });

 return promise;
}
```

```
it("should generate a token", () => {
 const testEmail = "test@gmail.com";
 //wrap your promise and resolve your promise
 expect(generateTokenPromise(testEmail)).resolves.toBeDefined();
});
```

Use async await to test promise

```
it("should generate a token", async () => {
 const testEmail = "test@gmail.com";
```

```
 const token = await generateTokenPromise(testEmail);
 expect(token).toBeDefined();
});
```

Has property test on objec

```
export class User {
 constructor(email) {
   this.email = email;
 }

 updateEmail(newEmail) {
   this.email = newEmail;
 }

 clearEmail() {
   this.email = '';
 }
}
```

```
it('should have an email property', () => {
 const testEmail = 'test@test.com';

 const user = new User(testEmail);

 expect(user).toHaveProperty('email');
});
```

Hooks, before all

```
import { it, expect, beforeAll, beforeEach, afterAll, afterEach } from "vitest";

import { User } from "./hooks";

//below without hooks, we not pass the tests
const testEmail = "test@test.com";
let user;

//befor all test do sth
```

```
beforeAll(() => {
 user = new User(testEmail);
});

//before each test, do something
beforeEach(() => {
 user = new User(testEmail);
});

afterEach(() => {
 console.log("clean up");
});

//perform clean up work
afterAll(() => {});

it("should update the email", () => {
 const newTestEmail = "test2@test.com";
 user.updateEmail(newTestEmail);
 expect(user.email).toBe(newTestEmail);
});

it("should have an email property", () => {
 expect(user).toHaveProperty("email");
});

it("should store the provided email value", () => {
 expect(user.email).toBe(testEmail);
});

it("should clear the email", () => {
 user.clearEmail();
 expect(user.email).toBe("");
});

it("should still have an email property after clearing the email", () => {
 user.clearEmail();

 expect(user).toHaveProperty("email");
});
```

Use concurrent to speed up the test

```
//all test in the suite will be conducted in parallel
// describe.concurrent();
//below test will run concurrently with other tests
it.concurrent("should update the email", () => {
 const newTestEmail = "test2@test.com";
 user.updateEmail(newTestEmail);
 expect(user.email).toBe(newTestEmail);
});

it.concurrent("should have an email property", () => {
 expect(user).toHaveProperty("email");
});
```

SPies and mocks

How to solve problem for side effect

The problem of side effect, we do not want to write to file

```
import path from 'path';
import { promises as fs } from 'fs';

export default function writeData(data, filename) {
 const storagePath = path.join(process.cwd(), 'data', filename);
 return fs.writeFile(storagePath, data);
}
```

Below writes to our file

```
import { it, expect } from "vitest";
import writeData from "./io";

it("should exe write file", () => {
 const testData = "test";
 const fileName = "test.txt";
 //return below waits for jest to await, why undefined
 //it has side effect - write data to harddrive - we do not want it to be write
```

```
 return expect(writeData(testData, fileName)).resolves.toBeUndefined();
});
```

We are not interested in testing writeFile function file, we only want to test if right parameter has been called and we have called the writeData function

We could use spy and mocks

- Spy - wrapper around original function or function placeholder to see if a function has been called, or what parameters have been received
- Mock - provide replacement for API to test specific behavior

Use a spy function (placeholder function) to check whether it has been called

```
import writeData from './util/io.js';

export function generateReportData(logFn) {
 const data = 'Some dummy data for this demo app';
 if (logFn) {
   logFn(data);
 }

 return data;
}

export async function storeData(data) {
 if (!data) {
   throw new Error('No data received!');
 }
 await writeData(data, 'data.txt');
}
```

Below check if logFn has been called

```
import { it, describe, vi, expect } from "vitest";
//vi=jest
import { generateReportData } from "./data";
describe("generate report data", () => {
```

```
 it("should generate report logfn is provided", () => {
   //create an empty function
   const logger = vi.fn();
   generateReportData(logger);
   //expect our placeholder fm to be called
   expect(logger).toHaveBeenCalled();
 });
});
```

Mock the whole module and test if its function has been called

```
import path from "path";
import { promises as fs } from "fs";

export default function writeData(data, filename) {
 const storagePath = path.join(process.cwd(), "data", filename);
 return fs.writeFile(storagePath, data);
}
```

Mock the whole module fs

```
import { it, expect, vi } from "vitest";
import writeData from "./io";
import { promises as fs } from "fs";
//import { promises } from "fs"; const { readFile, writeFile } = promises;

//mock will find fs module and replace it with empty module
//it will not create a test.text into data folder
vi.mock("fs");

it("should exe write file", () => {
 const testData = "test";
 const fileName = "test.txt";
 // return expect(writeData(testData, fileName)).resolves.toBeUndefined();
 writeData(testData, fileName);
 expect(fs.writeFile).toBeCalled();
});
```

Vi.mock (“fs”)

- This mock is only available in this mock file
- It is hoisted to the top like js in vitest

For spy function, we can also use toHaveBeenCallTimes and toHaveBeenCalledWith

- Test how many time this function has been called
- Test if a parameter has been used for this function

Mock function usually is empty, we custom the implementation of the function

```
import path from "path";
import { promises as fs } from "fs";

export default function writeData(data, filename) {
 const storagePath = path.join(process.cwd(), "data", filename);
 return fs.writeFile(storagePath, data);
}
```

Mock module's function

```
import { it, expect, vi } from "vitest";
import writeData from "./io";
import { promises as fs } from "fs";

vi.mock("fs");
//mocking path module and its join mehtod
//must put in default key because we use import default method
//mock (module module, fake custom module)
vi.mock("path", () => {
 return {
   default: {
     join: (...args) => {
       return args[args.length - 1];
     },
   },
 };
});

it("should exe write file", () => {
 const testData = "test";
```

```
 const fileName = "test.txt";
 writeData(testData, fileName);
 //assert the writeFile function to have receive the following arguments
 expect(fs.writeFile).toHaveBeenCalledWith(fileName, testData);
});
```

Make a global custom mock fn implementation

Make a global folder, when vi.mock("fs") - will always look at the file first then mock itself
__mock__/fs.js

//below promise resolve to nothing when called fs.writeFile

```
import { vi } from "vitest";

export const promises = {
 writeFile: vi.fn(() => {
   return new Promise((resolve, reject) => {
     //resolve to nothing
     resolve();
   });
 }),
};
```

Io.test.js

- vi.mock("fs") will first look into __Mock__ to see if there is a fs.js module
- If it does, then it will use the implementation
- If not, it will then mock an empty module

```
import { it, expect, vi } from "vitest";
import writeData from "./io";
import { promises as fs } from "fs";

//mock("fs") will first look into __Mock__ to see if a file fs.js exist
//before making it an empty module
vi.mock("fs");
vi.mock("path", () => {
 return {
   default: {
     join: (...args) => {
       return args[args.length - 1];
     },
```

```
  },
 };
});

it("should return a promise with no value if", () => {
 const testData = "test";
 const fileName = "test.txt";
 writeData(testData, fileName);
 // expect(fs.writeFile).toHaveBeenCalledWith(fileName, testData);
 return expect(fs.writeFile(fileName, testData)).resolves.toBeUndefined();
});
```

MockImplementationOnce, will have mock the function once, and switch back to empty function

```
import { it, describe, vi, expect } from "vitest";
//vi=jest
import { generateReportData } from "./data";
describe("generate report data", () => {
 it("should generate report logfn is provided", () => {
   const logger = vi.fn();
   //mock implementation once, will mock one time, and switch back to empty mock
   logger.mockImplementationOnce(()=>{})
   generateReportData(logger);
   expect(logger).toHaveBeenCalled();
 });
});
```

More on mock and deeper

Practice to test on the class you wrote

```
export class HttpError {
 constructor(statusCode, message, data) {
   this.statusCode = statusCode;
   this.message = message;
   this.data = data;
```

```
 }
}

export class ValidationError {
 constructor(message) {
   this.message = message;
 }
}
```

```
import { describe, expect, it } from "vitest";
import { HttpError } from "./errors";
describe("class httpError", () => {
 it("should contain the code, message, data", () => {
   const testStatus = 1;
   const testMessage = "test";
   const testData = { key: "test" };
   const httpError = new HttpError(testStatus, testMessage, testData);
   // expect(httpError).toHaveProperty("statusCode");
   // expect(httpError).toHaveProperty("message");
   // expect(httpError).toHaveProperty("data");
   expect(httpError.statusCode).toBe(testStatus);
   expect(httpError.message).toBe(testMessage);
   expect(httpError.data).toBe(testData);
 });

 it("should have test data undefined if no parameter is provided", () => {
   const testStatus = 1;
   const testMessage = "test";
   const httpError = new HttpError(testStatus, testMessage);
   expect(httpError.statusCode).toBe(testStatus);
   expect(httpError.message).toBe(testMessage);
   expect(httpError.data).toBeUndefined();
 });
});
```

Test if a function has throw error

validation.js

```
import { ValidationError } from './errors.js';

export function validateNotEmpty(text, errorMessage) {
```

```
 if (!text || text.trim().length === 0) {
   throw new ValidationError(errorMessage);
 }
}
```

Test if a specific message has been thrown

```
import { it, expect } from "vitest";
import { validateNotEmpty } from "./validation";

it("should throw an error if an empty string is provided", () => {
 const testInput = "";
 //wrapper for the function
 const validationFn = () => {
   validateNotEmpty(testInput);
 };
 //test if the wrapper function have thrown
 expect(validationFn).toThrow();
});

it("should throw an specific message if there is an error", () => {
 const testInput = "";
 const testMessage = "message";
 const validationFn = () => {
   validateNotEmpty(testInput, testMessage);
 };
 //test if the specific message was thrown
 expect(validationFn).toThrow(testMessage);
});
```

Mock api request to third party

Http.js

```
import { HttpError } from "./errors.js";

export async function sendDataRequest(data) {
 //fetch is function available in browser, sending request to backend api
 //fetch is a globally available object
 const response = await fetch("https://dummy-site.dev/posts", {
   method: "POST",
   headers: {
```

```
    "Content-Type": "application/json",
  },
  body: JSON.stringify(data),
});

const responseData = await response.json();

if (!response.ok) {
  throw new HttpError(
    response.status,
    "Sending the request failed.",
    responseData
  );
}

return responseData;
}
```

Http.test.js

```
import { vi, it, expect } from "vitest";
import { sendDataRequest } from "./http";

const testResponseData = { testKey: "testData" };
//fake the api request
const testFetch = vi.fn((url, options) => {
 return new Promise((resolve, reject) => {
   const testResponse = {
     ok: true,
     json() {
       return new Promise((resolve, reject) => {
         resolve(testResponseData);
       });
     },
   };

   resolve(testResponse);
 });
});
//mock a global available object/fn
vi.stubGlobal("fetch", testFetch);
```

```
it("should return the response data we sent", () => {
 const testData = { key: "req" };
 return expect(sendDataRequest(testData)).resolves.toEqual(testResponseData);
});
```

In this example project, the global `fetch()` API / function is used.

You can, of course, also use third-party libraries in frontend JavaScript projects though. For example, the axios library is a very popular library for sending HTTP requests from the frontend.

In case you're working with such a library, instead of a global value, you can mock that library as you learned in the previous section (i.e., use `vi.mock('axios')`, provide a `__mocks__/axios.js` file if necessary etc.).

Test if your string is convert to string,

```
import { HttpError } from "./errors.js";

export async function sendDataRequest(data) {
 //fetch is function available in browser, sending request to backend api
 //fetch is a globally available object
 const response = await fetch("https://dummy-site.dev/posts", {
   method: "POST",
   headers: {
     "Content-Type": "application/json",
   },
   body: JSON.stringify(data),
 });

 const responseData = await response.json();

 if (!response.ok) {
   throw new HttpError(
     response.status,
     "Sending the request failed.",
     responseData
   );
 }
```

```
 return responseData;
}
```

```
import { vi, it, expect } from "vitest";
import { sendDataRequest } from "./http";

const testResponseData = { testKey: "testData" };
//fake the api request
const testFetch = vi.fn((url, options) => {
 return new Promise((resolve, reject) => {
   if (typeof options.body !== "string") {
     //reject and return below string is it is not a string
     return reject("Not a string.");
   }

   const testResponse = {
     ok: true,
     json() {
       return new Promise((resolve, reject) => {
         resolve(testResponseData);
       });
     },
   };

   resolve(testResponse);
 });
});
//mock a global available object/fn
vi.stubGlobal("fetch", testFetch);

it("should return the response data we sent", () => {
 const testData = { key: "req" };
 return expect(sendDataRequest(testData)).resolves.toEqual(testResponseData);
});


it("should convert data to json before sendin", async () => {
 const testData = { key: "req" };
 let errorMessage;
 try {
```

```
    await sendDataRequest(testData);
  } catch (error) {
    //rejects generates an error
    errorMessage = error;
  }
  expect(errorMessage).not.toBe("Not a string.");
});
```

Modify your mockfetch implementation once

- Mark ok to be false
- When it is false, check if the reject error is an instance of the httpError class you created

```
//alter fetchMock implementation once
it("shoud throw an http error in case of non okay response", () => {
  testFetch.mockImplementationOnce((url, options) => {
    return new Promise((resolve, reject) => {
      const testResponse = {
        ok: false,
        json() {
          return new Promise((resolve, reject) => {
            resolve(testResponseData);
          });
        },
      };

      resolve(testResponse);
    });
  });
  const testData = { key: "req" };
  return expect(sendDataRequest(testData)).rejects.toBeInstanceOf(HttpError);
});
```

Use local mock values to for testing

App.js

```
import { extractPostData, savePost } from './posts/posts.js';

const formElement = document.querySelector('form');
```

```
export async function submitFormHandler(event) {
 event.preventDefault();

 const formData = new FormData(formElement);
 try {
   const postData = extractPostData(formData);
   await savePost(postData);
 } catch (error) {
   showError(error.message);
 }
}

formElement.addEventListener('submit', submitFormHandler);
```

Posts.js

```
import { sendDataRequest } from '../util/http.js';
import { validateNotEmpty } from '../util/validation.js';

export function savePost(postData) {
 postData.created = new Date();
 return sendDataRequest(postData);
}

export function extractPostData(form) {
 const title = form.get('title');
 const content = form.get('content');

 validateNotEmpty(title, 'A title must be provided.');
 validateNotEmpty(content, 'Content must not be empty!');

 return { title, content };
}
```

Posts.tests.js

```
import { beforeEach, describe, expect, it, test } from "vitest";
import { extractPostData } from "./posts";
const testTitle = "test title";
const testContent = "test content";
let testFormData;
```

```
describe("extrapost data", () => {
 beforeEach(() => {
   //reset the testFormData to always to its original value after multiple test
   testFormData = {
     title: testTitle,
     content: testContent,
     get(identifier) {
       return this[identifier];
     },
   };
 });
 it("extract title and content from form data", () => {
   const data = extractPostData(testFormData);
   expect(data.title).toBe(testTitle);
   expect(data.content).toBe(testContent);
 });
});
```