**Тестування Unit test**

1. Необхідні для визначення помилок при написанні коду
2. Визначення помилок при модифікації коду
3. Тестування складних місць в коді, де логіка доволі заплутана і можуть бути потенційні проблеми
4. Тестування API, які ми підключаємо в проект, для перевірки їх коректності відпрацювання

**Плагін для тестування:**

1. Jest – <https://jestjs.io/>

**Встановлення в проект:**

1. npm I --save-dev jest babel-jest jest-transform-stub
2. Створюємо файл із такою ж назвою файла, як і файл, який ми тестуємо із приставкою name**.spec.js**
3. В .json файлі прописуємо в “script” – команду “test”: “jest” – знайде усі jest тести і виконає їх
4. Для тесту ми виносимо метод в нашому файлі назовні через **module.exports = functionName**
5. В файлі тестування ми приймаємо даний метод: **const functionName = required(**pathBasicFile**)**
6. Деталі в прикладі

**Методи для тестування загальні:**

1. It(‘коментар до результату тесту’, () => {

**expect(*methodWithParams*).toBe(*expecting result*);**

}); - метод для перевірки одного варіанту використання методу

1. **describe**(‘coment’, () => {it(…);}); - метод обгортка для тестів, в коментарі переважно вказується, що зберігають тести всередині
2. **beforeEach() –** метод для виконання певних дій перед тестом
3. **jest.mock**(‘шлях до файлу, асинхронні операції якого ми імітуємо’) – метод для імітації асинхронних операції, де ми підставляємо власні дані в фейкові асинхронні методи, щоб протестувати реакцію певного коду на фейковий асинхронний метод

**Методи для тестування технічні:**

1. expect() – передається будь-який тип данних для перевірки
2. .toBe() - передаємо оічікуваний результат примітивних типів даних
3. .toEqual() – передаємо очікуваний результат об’єктних типів даних
4. .toHaveBeenCalledWith() – перевіряємо, із якими параметрами викликається даний метод

**Команди для прописування в “script”:**

1. jest
2. jest --watch - тестить файли при кожній зміні з git репозиторієм
3. jest --watchAll - тестить файли при кожній зміні без git
4. jest --coverage --color – виведе нам інформацю про покриття тестами заданої області

**End to End testing**

**Ступінь заглиблення в тестування:**

1. **Unit** – тестування окремих методів
2. **Integration** – інтеграційні тести призначені для тестування ряд модулів чи методів, зв’язаних одним сенсом, працюють коректно. Перевірка функціоналу, а не окремого методу. Тест зв’язків, між модулями чи методами
3. **End to End** – тести для симуляції поведінки користувача, тестування UI, тестування певного функціоналу. Тестування взаємодії користувача із DOM і функціоналом

**FrameWork Cypress:** <https://www.cypress.io/>

**Встановлення:**

1. npm i cypress --save-dev
2. Добавляємо в ‘scripts’ – ‘cypress:open’: ‘cypress open’ – для зручності
3. Активуємо дану команду і створюється папка cypress
   1. Fixture – прописуємо фейкові статичні дані
   2. Integration – папка із тестами
   3. Plugins
   4. Support

**Основні методи тестування:**

Такі ж, як і в Unit тестах

**Технічні методи тестування:**

1. Visit() – метод для відвідування сторінок по посиланню
2. Get() – отримуємо елементи для роботи із ними !бажано на html елементи вішати атребути, які ми використаємо для роботи і тестування
3. Should() – в даний метод ми вписуємо спец фрази, для визначення стану об’єкта над яким проводимо маніпуляції
4. As() – переназиваємо елемент із яким ми працюємо, так як нам зручно, для повторного використання (в подальшому звертатимемось до даного елемента **@**newName)
5. Contain() – перевіряє, чи містить в собі контент, вписаний в даний метод