ASTON

Интенсив-курс по React JS

astondevs.ru

Занятие 7. Дополнительные темы

1. ООП
2. ФП
3. Принципы программирования
4. Рефакторинг
5. Тестирование

Объектно-ориентированное программирование

ООП.

Это парадигма программирования, основанная на представлении программы в виде совокупности объектов и их взаимодействии. Выделяют 3 основных принципа:

Инкапсуляция: Свойство системы позволяющее объединить данные и методы внутри класса и скрыть реализацию от пользователя.

Наследование: Возможность создавать класс(наследник) на основе другого класса(родитель).

Полиморфизм: Возможность объектов с одинаковой спецификацией иметь различную реализацию.

А также выделяют и 4й принцип : **Абстракция**: разделении несущественных деталей реализации подпрограммы и характеристик существенных для корректного ее использования.

Аксессоры (геттеры и сеттеры) — это методы, задача которых контролировать доступ к полям. Это дает возможность снабдить такие методы дополнительными обработками.

Функциональное программирование

ФП. Это парадигма программирования, предполагающая обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы(в отличии от императивного ООП, где описывается процесс вычислений как последовательное изменение состояний.

Основные концепции ФП:

- Функциональный аргумент / Функция с функциональным значением
- Чистые функции
- Иммутабельность данных
- Композиция
- Функции высшего порядка
- Каррирование
- Частичное применение

Функциональное программирование

Чистые функции - это функции, которые не содержат побочных эффектов, не изменяют данные и при передаче одних и тех же аргументов выдают один и тот же результат.

```
function sum(a, b) {
  return a + b;
}
```

Композиция функций - это подход в функциональном программировании, который подразумевает вызов одних функций в качестве аргументов других, для создания сложных составных функций из более простых.

```
const add5 = x => x + 5;
const mult3 = x => x * 3;

add5(mult3(3)); //14
```

Функции высшего порядка - это функция принимающая, создающая внутри себя и/или возвращающая другую функцию

```
const useFuncToArguments = (func, x, y) => func(x, y);
const add = (x, y) => x + y;
useFuncToArguments(add, 5, 10); //15
```

Функциональное программирование

Каррирование в функциональном программирование — это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом.

```
const add = (a, b, c) => a + b + c;
add(1, 2, 3); // 6

const carryAdd = a => b => c => a + b + c;
carryAdd(1)(2)(3); //6
```

Частичное применение - это преобразовании функции в другую функцию, обладающую меньшим числом аргументов. Некоторые аргументы такой функции оказываются зафиксированными.

```
const add = (a, b, c) => a + b + c;
add(1, 2, 3); // 6

const addFiveToAgruments = (a, b) => add(5, a, b);
addFiveToAgruments(0, 1) // 6
```

Принципы программирования

DRY(don’t repeat yourself)

Не повторяй себя. «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы»

```
for ( let i = 0; i < data.length; i++ ) {
  if (data[i].includes(KEY)) {
    main.push(data[i])
  }
}
```

Выносим в функцию. И вызываем с нужным нам аргументом.

```
function pushSuitableArrayElementsToMain(array) {
  for ( let i = 0; i < array.length; i++ ) {
    if (array[i].includes(KEY)) {
      main.push(array[i])
    }
  }
}
```

Принципы программирования

KISS(keep it simple, stupid)

Простота кода – превыше всего, потому что простой код – наиболее понятный. Не стоит заблуждаться и думать, что принцип подразумевает самое простое написание кода. Все с точностью да наоборот. Простой и понятный код писать довольно тяжело.

KISS работает во всем. Не только в написании кода, но и в проектировании, и даже в общении с заказчиком. Иногда лучше предложить более простой вариант.

Принципы программирования

YAGNI(you aren’t gonna need it)

Тебе это не понадобится. Принцип гласит о том, чтобы не добавлять в проект функциональность, которая сейчас не нужна.
Выполняя задачи поставленные по ТЗ не нужно добавлять функциональность, которая там не описана.

SOLID

S	Принцип единственной ответственности (The Single Responsibility Principle) Каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.
O	Принцип открытости/закрытости (The Open Closed Principle) «программные сущности … должны быть открыты для расширения, но закрыты для модификации.»
L	Принцип подстановки Барбары Лисков (The Liskov Substitution Principle) «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.» Наследующий класс должен дополнять, а не изменять базовый.
I	Принцип разделения интерфейса (The Interface Segregation Principle) «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.»
D	Принцип инверсии зависимостей (The Dependency Inversion Principle) Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Рефакторинг

Процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы.

Если вы заметили в коде нарушения основных принципов хорошего(чистого) кода, то нужно проводить его рефакторинг, применяя эти самые принципы к коду. -Повторение кода? Вынести его в функцию или класс

-Magic numbers? Вынести их в константы.

-Переменные с именами a,b,c ? Переименуйте их согласно тому, Для чего они нужны. -Функция огромная и делает 10 несвязных действий? Разбейте на несколько функций.

Список и описание основных рефакторингов можно найти в книге М. Фаулера “**Рефакторинг кода на JavaScript**”

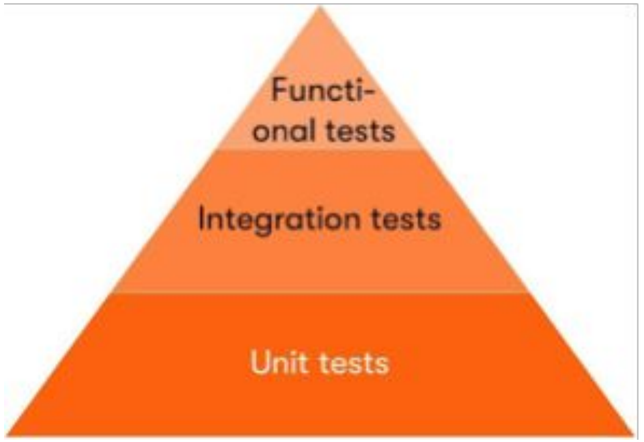
Для более подробного понимания чистого кода, следует почитать книгу Р. Мартина “**Чистый код**”

Тестирование

Тестирование - это

процесс оценки того, что все части приложения ведут себя так, как описано в требованиях.

Если мы говорим, о тестировании кода, то это:

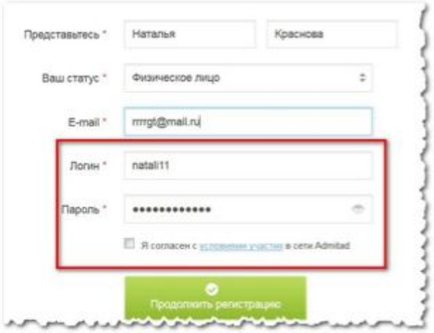


Тестирование

Функциональное тестирование

Например при вводе всех корректных данных

и нажав на кнопку, мы сможем протестировать функцию регистрации

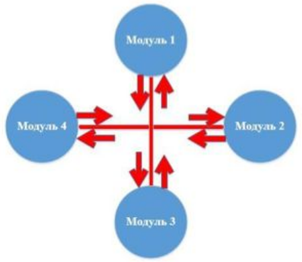


Интеграционное тестирование

Unit tests.

— процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода.



Jest

Jest

— среда тестирования JavaScript с упором на простоту. Можно использовать с чистым JS и с любым фреймворком. Пример простейшего теста на функцию add5.

```
test('testing simple function', ()=> {
  expect(add5(5)).toEqual(10)
});
```

Есть множество вариантов того, как будет проходить сравнение значений. Все их множество есть в документации, но самые часто используемые:

toEqual()

toBe()

toBeFalsy() / toBeTruthy()

toContain()

toBeDefined()

toHaveBeenCalled()

toHaveBeenCalledTimes()

toHaveBeenCalledWith()

React Testing Library

React Testing Library — это набор вспомогательных функций, позволяющий тестировать React-компоненты не полагаясь на их внутреннюю реализацию.

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // подготавливаем DOM-элемент, куда будем рендерить
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // подчищаем после завершения
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

```
import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";
```

```
let container = null;
beforeEach(() => {
  // подготавливаем DOM-элемент, куда будем рендерить
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // подчищаем после завершения
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders with or without a name", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Hey, stranger");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Hello, Jenny!");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Hello, Margaret!");
});
```

Группировка тестов

Кроме test, есть еще **describe, beforeEach, beforeAll, afterEach, afterAll**.

```
beforeAll(() => console.log('1 - beforeAll'));
afterAll(() => console.log('1 - afterAll'));
beforeEach(() => console.log('1 - beforeEach'));
afterEach(() => console.log('1 - afterEach'));
test('', () => console.log('1 - test'));
describe('Scoped / Nested block', () => {
  beforeAll(() => console.log('2 - beforeAll'));
  afterAll(() => console.log('2 - afterAll'));
  beforeEach(() => console.log('2 - beforeEach'));
  afterEach(() => console.log('2 - afterEach'));
  test('', () => console.log('2 - test'));
});

// 1 - beforeAll
// 1 - beforeEach
// 1 - test
// 1 - afterEach
// 2 - beforeAll
// 1 - beforeEach
// 2 - beforeEach
// 2 - test
// 2 - afterEach
// 1 - afterEach
// 2 - afterAll
// 1 - afterAll
```

Моковые данные

Часто, чтобы протестировать какую-то функцию, нам нужно передать в нее какие-то данные, которые в реальном приложение поступают в нее извне. Для этого создаются моковые данные. Например мы тестируем сагу, которая принимает в себя пришедший с бэка ответ и распарсивает его и кладет данные в наш store. Для этого нам нужно замокать ответ.

```
export const mockResponseBody = {
  result: {
    sum: 1000,
    commission: 10,
    owner: 'Jhon Jhonson'
  },
  state: 'first',
  isSuccess: true,
};
```

```
describe('Тестирование логики car записи', () => {
  test('saga setPayFields производит запись полей ответа в store', () => {
    const saga = setPayFields(mockResponseBody);

    expect(saga.next().value).toEqual(put(putState(mockResponseBody.state)));
    expect(saga.next().value).toEqual(put(change(FORM, 'sum', mockResponseBody.result.sum)));
    expect(saga.next().value).toEqual(put(change(FORM, 'commission', mockResponseBody.result.commission)));
    expect(saga.next().value).toEqual(put(change(FORM, 'owner', mockResponseBody.result.owner)));
    expect(saga.next().done).toBeTruthy();
  });
});
```

Enzyme и snapshot тестирование компонентов

Enzyme и snapshot тестирование компонентов

snapshot тестирование используется для проверки правильности рендеринга наших реакт компонентов.

Enzyme — это библиотека, которая предоставляя удобные функции рендеринга компонентов. Enzyme разработан в Airbnb.

Enzyme позволяет рендерить компоненты в коде. Для этого есть несколько удобных функций, которые выполняют разные варианты рендеринга:

- - полный рендеринг (как в браузере, full DOM rendering);
- - упрощенный рендеринг (shallow rendering);
- - статический рендеринг (static rendering).

```
const wrapper = render(<MyComponent />);

test('snapshot testing of MyComponent', () => {
  expect(wrapper).toMatchSnapshot();
})
```