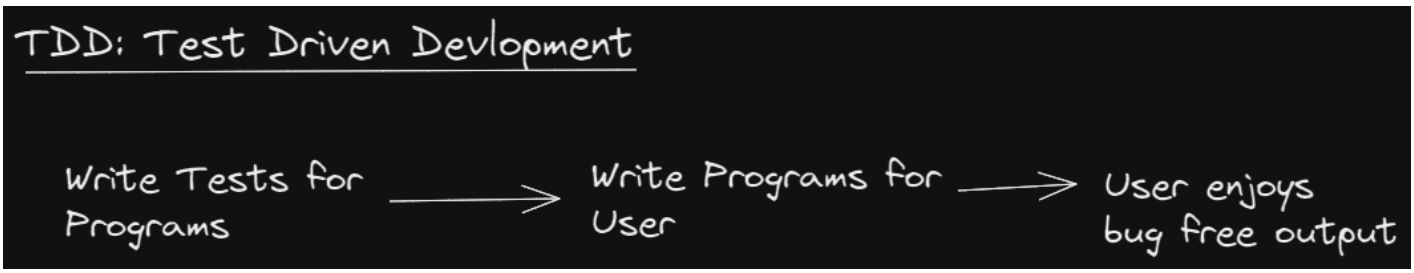
Testing Notes

Why do we write tests?

- To save time during manual work.
- We can automate testing by writing a test that reduces the need for manual testing.
- Test cases don't have to be updated regularly just like documentation.
- We can easily add new features or refactor the code with testing without breaking other features.
- Automated testing works on every change we make and make sure that things are working as expected.
- Test runs on the command line which means the end user will not receive the bad build.
- The code quality improves significantly with Test Driven Development(TDD).

What is test-driven development?

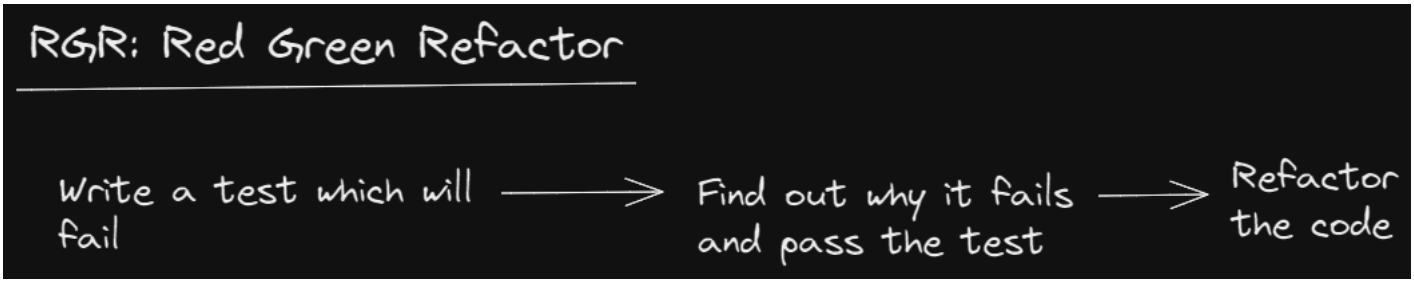
- You first write a test for a function.
- Then you make the test fail.
- Then you write the function to make that test pass.



What is RGR?

- Red — think about *what* you want to develop (this may cause test cases to fail hence red)

- Green — think about *how* to make your tests pass
- Refactor — think about *how* to improve your existing implementation (since the test for the code is already there we can easily refactor)



What is the syntax of test function?

`test()` is a function with two arguments:

1. name of test
2. callback - returns `true` or `false`

The output of the function should print the result of the test.

How to write a test function without the jest library?

- Suppose we want to test the below sum function.

```
function sum (sum1, sum2) {
    return sum1 + sum2;
  };
```

COPY

- Now we will write a test function that takes the name of the test as a string and a callback function that will help us in validating the output of the above sum function.

- The callback returns a boolean value to indicate if the test is failed or passed.

- Log the name and test result.

```
const test = (testname, cb) => {
    console.log("testing...", testname);
     const bool = cb();
     return bool ? `${testname}... passed` : `${testname}... failed`;
    };
```

COPY

- We will call the function.

```
console.log(test("should add two numbers", () => sum(2, 3) === 5));
```

COPY

- The output will be `test passed` since 2 + 3 gives 5.

```
Output - "test passed"
```

COPY

How to write a test using the jest library?

- Create a file with the file name to be tested followed by `test.js`.
- Suppose we want to test the `App.js` file so its test file will be `App.test.js`.
- We will first create a function in `App.js` and export it for testing. Here we have created a function that returns the sum of 2 numbers.

```
export function sum(num1, num2){
return num1 + num2;
}
```

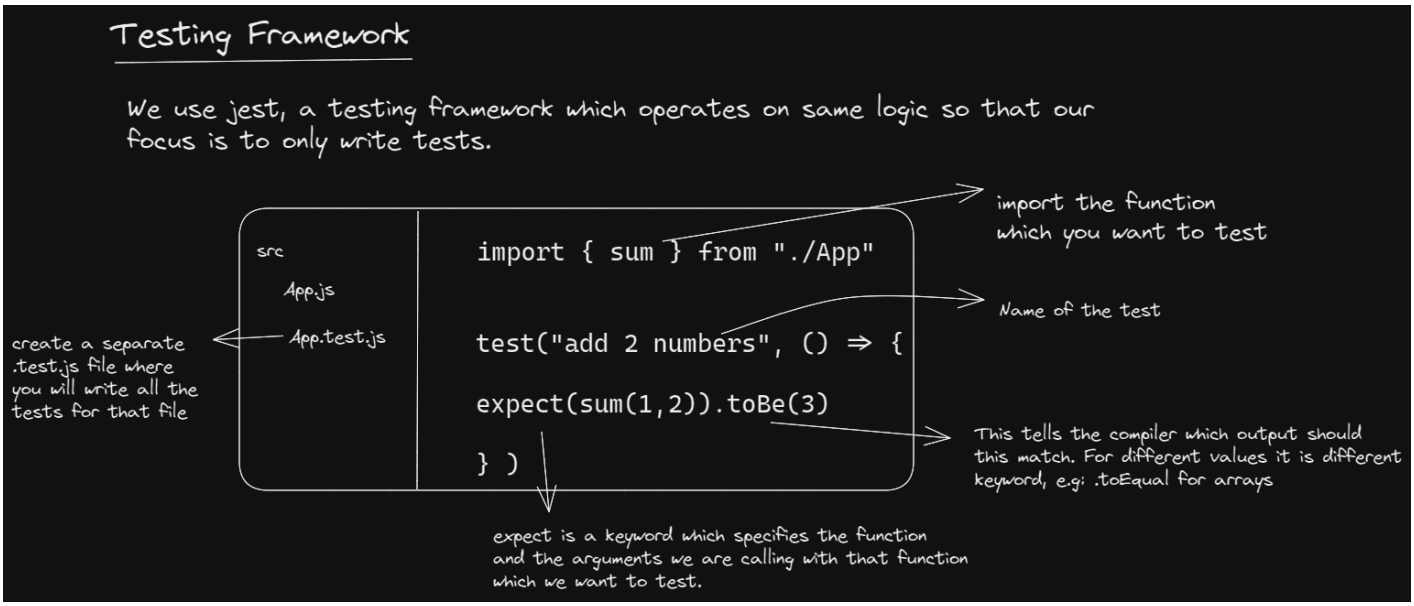
COPY

- Now we will write a test function. The test function takes the string and a callback function which will validate the result of the function.

```
test("should add two numbers", ()=>{
expect(add(2,3)).toBe(5);
})
```

COPY

- The output will be test passed since 2 + 3 will return 5.



Matchers

`toBe` is used to test exact equality such as numbers or strings.

```
expect().tobe(3)
```

COPY

`toEqual` recursively checks every field of an object or array.

```
expect().toEqual({one: 1, two: 2})
```

COPY

For more such examples of matchers for different datatypes, refer to Matchers from JEST documentation

What is `describe()` while testing?

- We can club multiple tests belonging to the same category or function with the help of `describe()`.
- `describe()` is just a collection of multiple tests with a title.

```
describe(string, callback)
```

COPY

- `describe()` takes 2 parameters, first is a string that acts like the title for the test cases and the callback takes the multiple test cases.
- We can have nested `describe()` as well in order to further club the tests.

How to handle edge cases while testing?

- Suppose we are writing a function that capitalizes the first letter of a string. We have to test it for 3 cases.

Input	Expected Output
hello	Hello
world	World
“”	“”

- First, we will write `capitalize()` function inside `App.js` and export it for testing.

```
export function capitalize(str) {
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

COPY

- Now we will write the test for the function in `app.test.js`.
- We will enclose our test cases for `capitalize()` function inside `describe()`.
- Now we can write test cases for the 3 different cases separately within the test function callback.

```
describe("capitalize", () => {
    test("capitalizes the first letter of a string", () => {
      expect(capitalize("hello")).toBe("Hello");
      expect(capitalize("world")).toBe("World");
    });

    test("returns an empty string if input is empty", () => {
      expect(capitalize("")).toBe("");
    });
  });
```

COPY

What is `it()` in testing?

Another way of writing `test()` using the jest library is `it()`.

How to write a test for the reducer function?

- Since reducer is a function that returns an object state, we can easily test the function.
- To test the reducer function we take the approach of - Arrange, Act, and Assert.
- In `Arrange`, we first get the initial values we want to call the reducer function with.
- Then we perform the `Act` ****by calling the reducer function with `state` and `action` and get an output.
- During `**Assert**`, we check whether the expected value is as expected or not.
- With the help of TDD, we will first write the test for the reducer. In this case, we will write a test for adding an item to the cart.
- The initial state for the reducer will be as follows-

```
export const initialState = {
  items: [],
  totalPrice: 0,
  totalQuantity: 0
```

```
};
```

COPY

- Now, we will write the test with the help of TDD, for the reducer for the `ADD_TO_CART` case.

```
describe("testing cart", () => {
  it("should add item to the cart", () => {

     **// Arrange**
    const initialState = {
      items: [{ product: "book", price: 200 }],
      totalPrice: 200,
      totalQuantity: 1
    };

    const action = {
      type: "ADD_TO_CART",
      payload: {
        item: { product: "shades", price: 399 }
      }
    };

    // Act
    const updatedState = cartReducer(initialState, action);

    // Assert
    expect(updatedState).toEqual({
      items: [
        { product: "book", price: 200 },
        { product: "shades", price: 399 }
      ],
      totalPrice: 599,
      totalQuantity: 2
    });
  });
});
```

COPY

- At this point, the test will fail since we haven't written the case for `ADD_TO_CART`.

- Now, we will write the case for adding an item(`ADD_TO_CART`) to the cart inside the `cartReducer()` function and validate the function.

```
function cartReducer(state = initialState, action) {
  switch (action.type) {
    case "ADD_TO_CART":
      return {
        ...state,
        items: [...state.items, action.payload.item],
        totalPrice: state.totalPrice + action.payload.item.price,
        totalQuantity: state.totalQuantity + 1
      };

    default:
      break;
  }
}
```

COPY

- Once the case is written, the test will again pass.

- This is called `RGR(red-green-refactor)`.

Best practices for testing

- The time taken by a `test` matters. Therefore, the `test` should be easier to run and should be fast.
- Even if we change the order of `tests`, the tests should work as expected.
- Each `test` should be independent of one another and there should be no dependency of one `test` on another.
- The tests should be readable and must follow `AAA (arrange action assert).`
- All `tests` should be automatic and must not demand inputs to complete.
- One `test` should be responsible for testing one thing.

What is code coverage?

- The percentage of code that has been tested vs the percentage of code that has not been tested in the completed codebase is known as code coverage.
- Don’t aim for 100% code coverage. 85% and above code coverage is very much acceptable.
- Must have test cases for P0(priority 0) functions like `reducer` functions and `utility` functions.

What is `test.only()`?

If there are multiple tests written, and we just want to test a single `test` function, we can replace `test()` with `test.only()`.