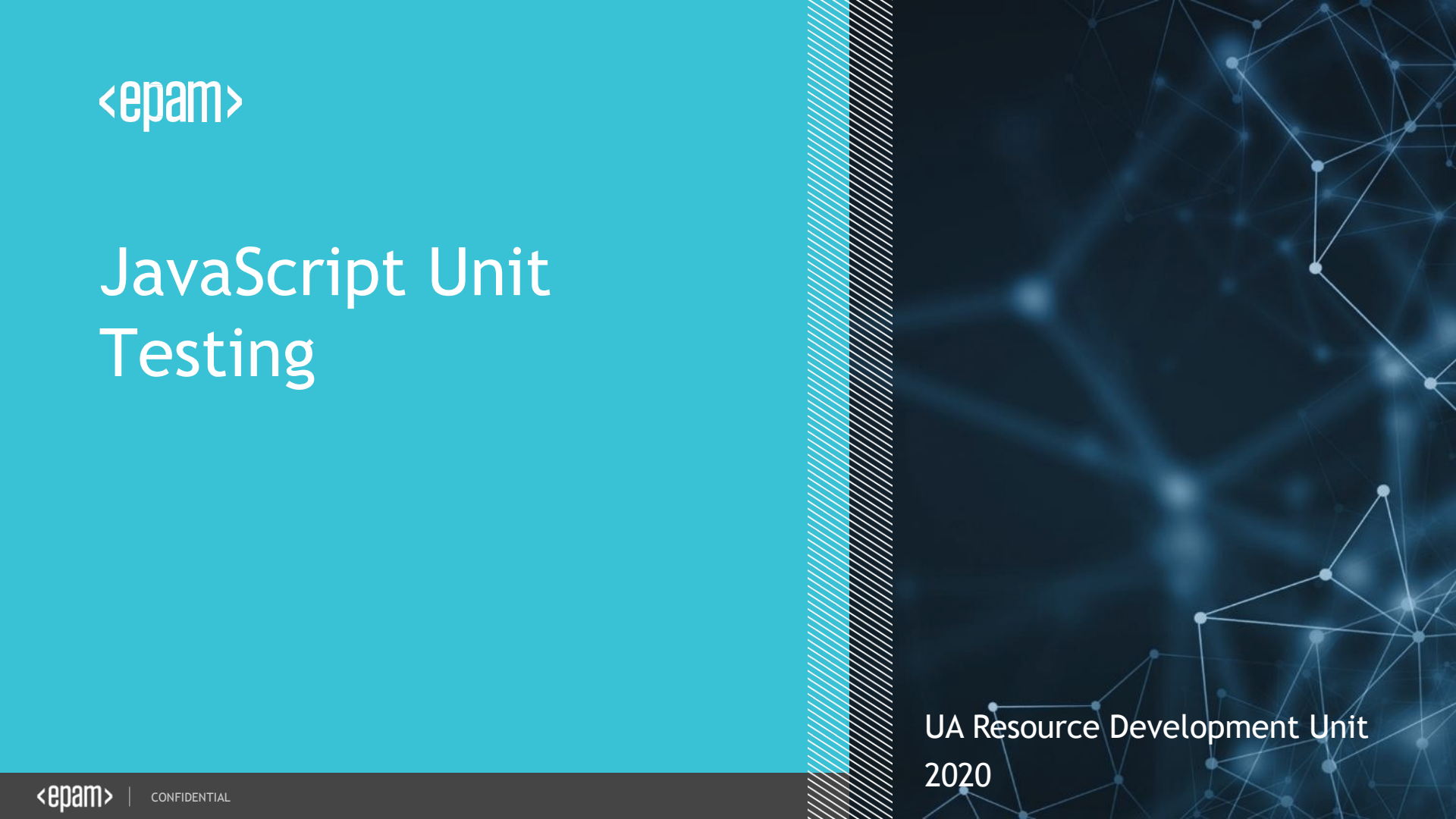

JavaScript Unit Testing
UA Resource Development Unit
2020

AGENDA

1. Software Testing Overview
2. Benefits of Software Testing
3. Testing Types
4. Unit/Integration/E2E Testing
5. Testing Pyramid Concept
6. Testing Tools & Frameworks
7. JS Unit Testing with Jasmine Framework
8. Jasmine functions
9. Matchers
10. Organizing your specs
11. Spying on your code
12. Testing asynchronous code
13. Test reports
14. Useful links

Software Testing Overview

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.

Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation.

Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use.

Wikipedia - Software testing

Benefits of Software Testing

1. Working product
2. Better Quality
3. Satisfied Customer
4. Increased development velocity
5. Reliability of Software
6. Prevents code aging
7. Improve User Experience

Testing Types

Functional Testing

- Unit
- Integration
- System (functional)
- Sanity
- Smoke
- Interface
- Regression
- Beta/Acceptance

Non-functional testing

- Performance
- Load
- Stress
- Volume
- Security
- Compatibility
- Install
- Recovery
- Reliability
- Usability
- Compliance
- Localization

Unit/Integration/E2E Testing

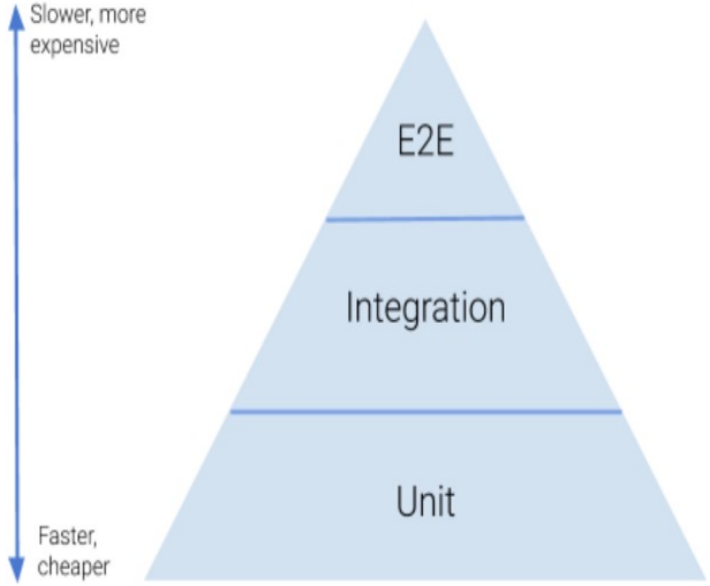
Unit testing: is testing of an individual software component or module. It is typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code.

Integration testing: testing of all integrated modules to verify the combined functionality after integration.

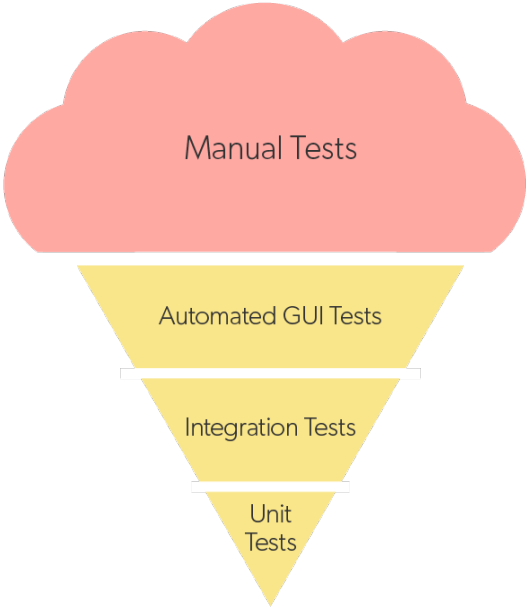
Functional/End-to-End: involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Testing Pyramid Concept

Testing Pyramid
(pattern)

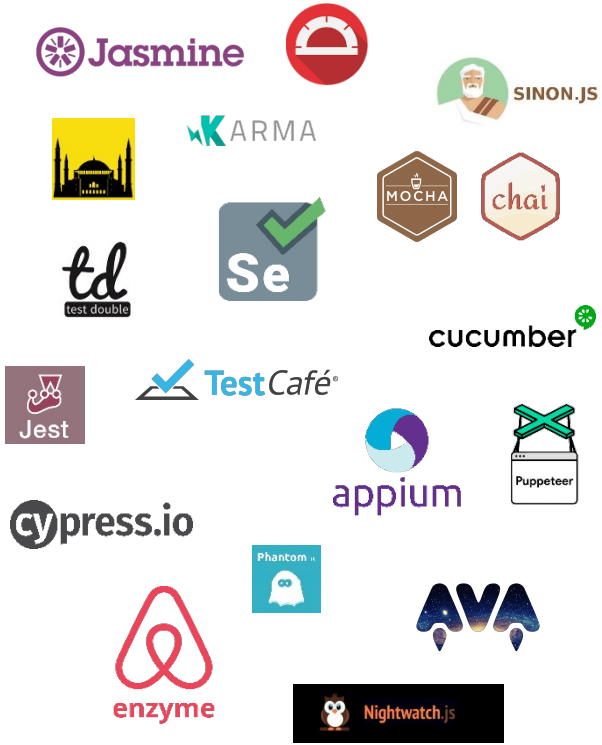


Inverted Testing Pyramid
(anti pattern)



Testing Tools & Frameworks

- launch your tests in the Browser or Node.js env
  (Karma, Jasmine, Jest, TestCafe, Cypress)
- testing structure providers help you arrange your test files
  (Mocha, Jasmine, Jest, Cucumber, TestCafe, Cypress)
- assertion functions check if the results a test returns are as expected *(Chai, Jasmine, Jest, Unexpected, TestCafe, Cypress)*
- generate and display test progress and results
  (Mocha, Jasmine, Jest, Karma, TestCafe, Cypress)
- provide mocks, spies, and stubs
  (Sinon, Jasmine, enzyme, Jest, testdouble)
- generate and compare snapshots of component and data structures *(Jest, Ava)*
- generate code coverage reports *(Istanbul, Jest, Blanket)*
- browser Controllers simulate user actions for Functional Tests *(Nightwatch, Nightmare, Phantom, Puppeteer, TestCafe, Cypress)*



JS Unit Testing with Jasmine Framework

Suite

A Jasmine suite is a group of test cases that can be used to test a specific behavior of the JavaScript code (a JavaScript object or function). This begins with a call to the Jasmine global function `describe` with two parameters – first parameter represents the title of the test suite and second parameter represents a function that implements the test suite.

```
describe("Test suite", () => {
          ...
});
```

Spec

A Jasmine spec represents a test case inside the test suite. This begins with a call to the Jasmine global function `it` with two parameters – first parameter represents the title of the spec and second parameter represents a function that implements the test case.

```
describe("Test Suite",() => {
  it("test spec", () => {
    …
  });
});
```

Expectations

Spec contains one or more expectations. Each expectation represents an assertion that can be either `true` or `false`. In order to pass the spec, all of the expectations inside the spec have to be `true`. If one or more expectations inside a spec is `false`, the spec fails.

```
describe("Test Suite", () => {
  it("test spec", () => {
    expect( expression ).toEqual(true);
  })
});
```

Skipped Specs and Suites

Jasmine also allows the developers to skip one or more than one test cases. These techniques can be applied at the Spec level or the Suite level. Depending on the level of application, this block can be called as a Skipping Spec and Skipping Suite respectively.

```
xdescribe("Test Suite", () => {
        ...
});

describe("Test Suite", () => {
        xit("test spec", () => {
        ...
    });
});
```

Matchers

toBe():
Expects the actual value to be equal (===) to the expected value.

toEqual():
Similar to the toBe method but it uses deep equality comparison.

toBeFalsy():
Expects the actual value to be falsy.

```
describe('test.js', () => {
  it('is true', () => {
    expect(true).toBe(true);
  });

  it('has same keys and values', () => {
    expect(obj1).toEqual(obj2);
  });

  it('is falsy', () => {
    expect(0).toBeFalsy();
  });
});
```

Matchers

toBeTruthy():
Expects the actual value to be truthy.

toBeNull():
Expects the actual value to be null.

toBeDefined():
Expects the actual value to be defined.

```
describe('test.js', () => {
  it('is truthy', () => {
    expect({}).toBeTruthy();
  });

  it('is null', () => {
    expect(result).toBeNull();
  });

  it('is defined', () => {
    expect(result).toBeDefined();
  });
});
```

Matchers

toContain():
Expects the actual value to contain a specific value.

not:
This is a chain method that can be used with all matchers.

toBeNaN():
Expects the actual value to be NaN (Not a Number).

```
describe('test.js', function() {
  it('contains 1', function() {
    const array = [1, 2, 3];
    expect(array).toContain(1);
  });

  it('is not true', function() {
    expect(true).not.toBe(false);
  });

  it('is NaN', function() {
    expect('4'*5).toBeNaN();
  });
});
```

Matchers

toThrow():
Expects a function to throw something.

toThrowError():
Expects a function to throw an error.

toMatch():
Expects the actual value to match a regular expression.

```
describe('test.js', () => {
  it('throws', () => {
    const err = message => throw new Error(message);
    expect(err).toThrow();
  });

  it(‘throws an error', () => {
    const err = message => throw new Error();
    expect(err).toThrowError();
  });

  it('matches regex', function() {
    const STR = 'my string';
    expect(STR).toMatch(/string$/);
  });
});
```

Organizing specs

beforeAll():
Runs **once** before all of the specs in the describe are run.

afterAll():
Runs **once** after all of the specs in the describe are run.

```
describe('test.js', () => {
  beforeAll(() => {
    this.boolean = true;
  });

  afterAll(() => {
    this.boolean = false;
  });

  describe('someMethod()', () => {
    it('is true', () => {
      expect(this.boolean).toBe(true);
    });
  })
});
```

Organizing specs

beforeEach():
Runs before each of the specs in the describe in which it is called.

afterEach():
Runs after each of the specs in the describe in which it is called.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    beforeEach(() => {
      this.boolean = true;
    });

    afterEach(() => {
      this.boolean = false;
    });

    it('is true', () => {
      expect(this.boolean).toBe(true);
    });
  })
});
```

Spying on code (stubs/test doubles)

toHaveBeenCalled():
Expect the actual (a Spy) to have been called.

toHaveBeenCalledWith():
Expect the actual (a Spy) to have been called with particular arguments at least once.

toHaveBeenCalledTimes():
Expect the actual (a Spy) to have been called the specified number of times.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    it('calls spy', () => {
      spyOn(calc, 'add').and.stub();
      const result = calc.sum('2+3');
      expect(calc.add).toHaveBeenCalled();
      expect(calc.add).toHaveBeenCalledWith('2');
      expect(calc.add).toHaveBeenCalledTimes(2);
    });
  });
});
```

Spying on code (stubs/test doubles)

callThrough():
Tell the spy to call through to the real implementation when invoked.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    it('calls spy', () => {
      spyOn(calc, 'add').and.callThrough();
      const result = calc.sum('2+3');
      expect(calc.add).toHaveBeenCalled();
      expect(result).toEqual(5);
    });
  });
});
```

Spying on code (stubs/test doubles)

callFake():
Tell the spy to call a fake implementation when invoked.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    it('calls spy', () => {
      spyOn(calc, 'add').and.callFake(() => {
        return 9;
      });
      const result = calc.sum('2+3');
      expect(calc.add).toHaveBeenCalled();
      expect(result).toEqual(9);
    });
  });
});
```

Spying on code (stubs/test doubles)

returnValue(Value):
Tell the spy to return the value when invoked.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    it('calls spy', () => {
      spyOn(calc, 'sum').and.returnValue(3);
      const result = calc.sum('1+1');
      expect(calc.add).toHaveBeenCalled();
      expect(result).toEqual(3);
    });
  });
});
```

Spying on code (stubs/test doubles)

returnValues:
Tell the spy to return one of the specified values (sequentially) each time the spy is invoked.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    it('calls spy', () => {
      spyOn(calc, 'sum').and.returnValues(4);
      calc.sum('1+1');
      const result = calc.sum('1+1');
      expect(calc.add).toHaveBeenCalled();
      expect(result).toEqual(4);
    });
  });
});
```

Spying on code (stubs/test doubles)

throwError()
Tell the spy to throw an error when invoked.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    it('calls spy', () => {
      spyOn(calc, 'sum').and.throwError(
        'Error'
      );

      expect(() => {
        calc.sum('1+1')
      }).toThrowError('Error');
    });
  });
});
```

Spying on code (stubs/test doubles)

spyOnProperty()
Install a spy on a property installed with Object.defineProperty onto an existing object.

```
describe('test.js', () => {
  describe('someMethod()', () => {
    it('calls spy', () => {
      const spy = spyOnProperty(calc, total, 'get');
      calc.sum('1+2');
      expect(spy).toHaveBeenCalled();
    });
  });
});
```

Testing asynchronous code

Using promises:

Functions passed to **beforeAll**, **afterAll**, **beforeEach**, **afterEach**, and **it** can **return a promise** that should be resolved when the async work is complete.

Spec will not start until the promise returned from the call to **beforeEach** above is settled. And this spec will not complete until the promise that **it** returns is settled. If the promise is rejected, the spec will fail.

```
describe('promises', () => {
  beforeEach(() => {
    return Promise.resolve();
  });

  it('returns promise result', () => {
    return thenable().then(smth => {
       expect(smth).toBe(expectation);
    });
  });
});
```

Testing asynchronous code

Using async/await:

Functions passed to **beforeAll**, **afterAll**, **beforeEach**, **afterEach**, and **it** can be declared **async** in environments that support **async/await**.

This spec will not start until the promise returned from the call to **beforeEach** above is settled. And this spec will not complete until the promise that it returns is settled.

```
describe('async/await', () => {
  beforeEach(async () => {
    await doSomethingAsync();
  })

  it('await for the result', async () => {
    expect(await smthAsync()).toBe(expectation);
  });
});
```

Testing asynchronous code

Using done callback:

This is a lower-level mechanism and tends to be more error-prone, but it can be useful for testing callback-based code or for tests that are inconvenient to express in terms of promises. If the function passed to Jasmine takes an argument (traditionally called done), Jasmine will pass a function to be invoked when asynchronous work has been completed.

(outdated approach)

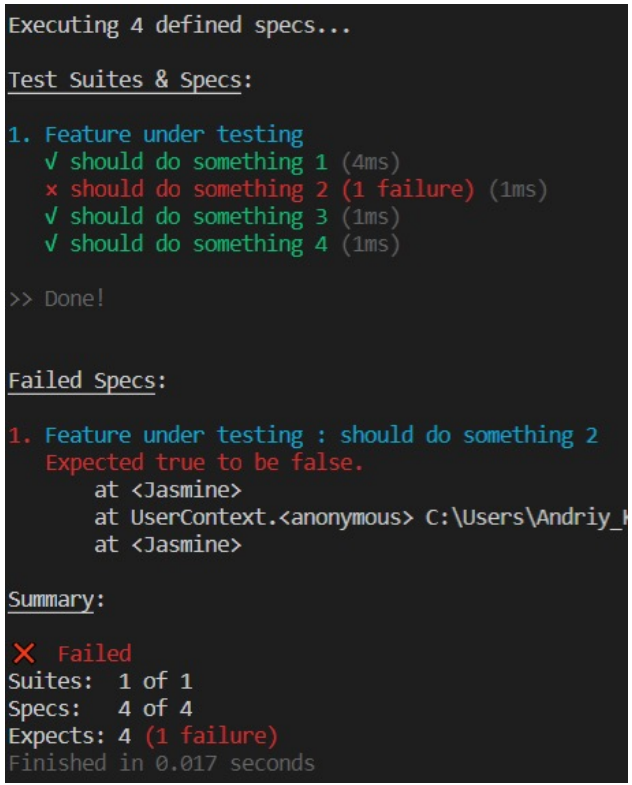
```
describe('done callback',  function () {
  it('call done', function (done) {
    doSomethingAsync(function (optParams) {
      expect(optParams).toBe(expectations);
      done();
    });
  });
});
```

Test reports

A Test report is an organized summary of testing objectives, activities, and results. It is created and used to help stakeholders (product manager, analysts, testing team, and developers) understand product quality and decide whether a product, feature, or a defect resolution is on track for release.

Allows to track:

- execution report
- code coverage
- possible defects



Default Jasmine reporter

Useful links

- Why testing is important in: Benefits of Software Testing
- Types Of Software Testing: Different Testing Types With Details
- Article Test Pyramid: the key to good automated test strategy
- Great overview of JavaScript Testing tools and popular frameworks
- Worth to read! Writing Testable Code
- Unit Testing/TDD/BDD

FE Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

serhii_shcherbak@epam.com

With the note [FE Online UA Training Course Feedback]

Thank you.

Q&A

UA Frontend Online LAB