Fundamentals of Jasmine and Unit Testing

A Day in the Life of a MERN Stack Developer

Joe, a developer at an IT company, is asked to develop a calculator application. Joe is also required to test the application using the Jasmine framework to ensure minimal errors. The company adopts scrum methodology for the development and release of its products.

Joe has to create the application that can perform multiple operations. During the daily stand-up meeting, he is asked to test the application in addition to other tasks.

Joe has to follow the agile methodology to complete his tasks.

In this lesson, we will learn how to solve this real-world scenario to help Joe complete his task effectively and quickly.

Learning Objectives

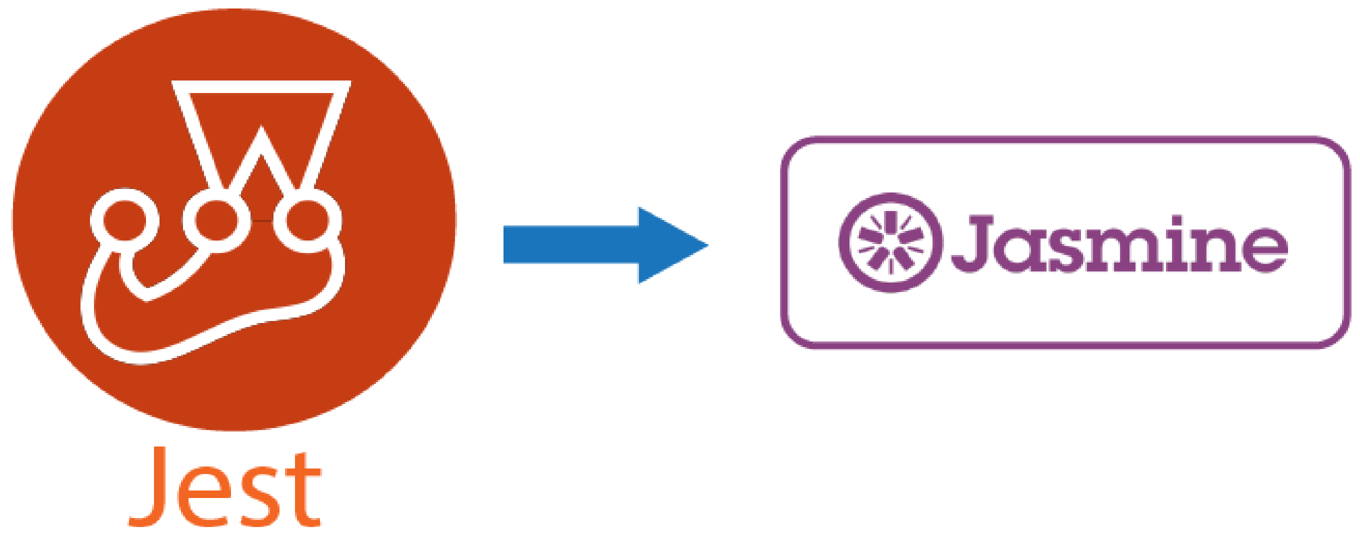
By the end of this lesson, you will be able to:

- Define Jasmine
- Differentiate unit and integration testing
- Classify Behavior Driven Development architecture
- Implement different structures of Jasmine

Introduction to Jasmine

What Is Jasmine?

- Open-source JavaScript framework
- Capable of testing any kind of JavaScript application
- Adheres to Behavior Driven Development (BDD) framework
- BDD is a procedure to ensure each line is properly tested



Jest is built on Jasmine.

Why Jasmine?

Available in different versions

Open-source framework

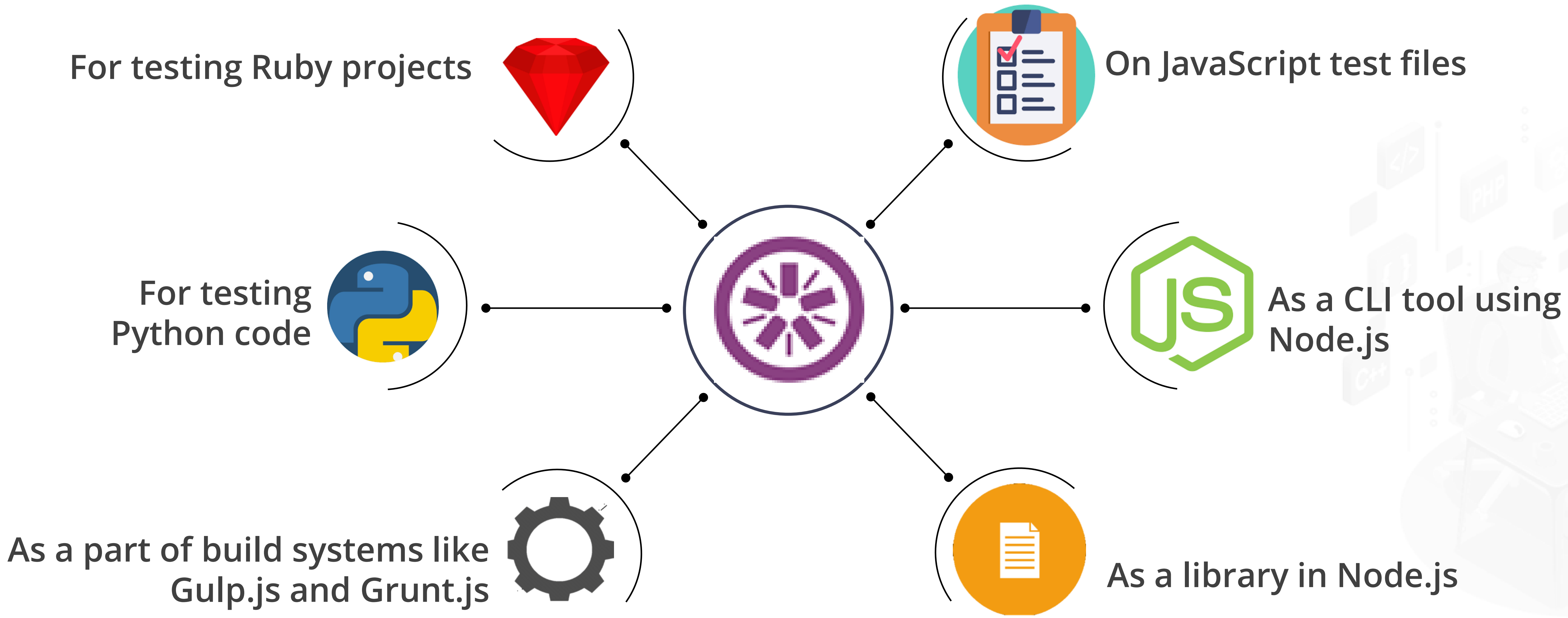
Heavily influenced by Rspec, JS Spec, and Jspec

Independent of any other JavaScript framework

Does not require any DOM

Clean and obvious syntax

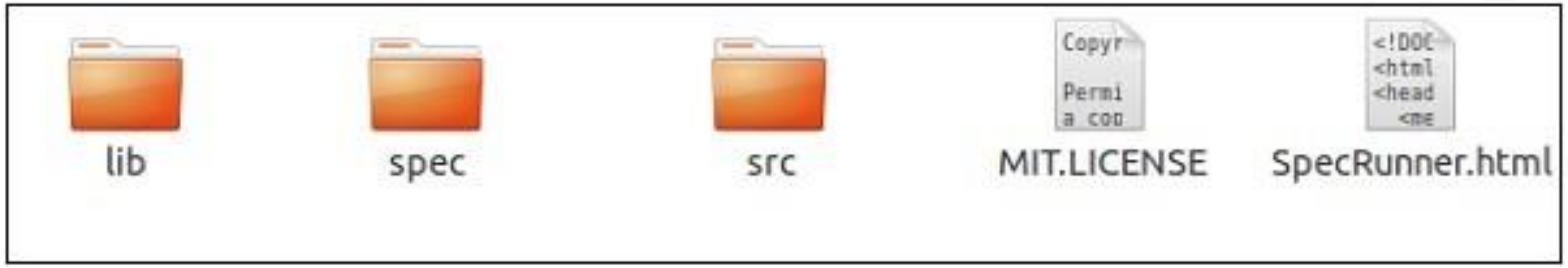
When to Use Jasmine?



For testing Ruby projects
On JavaScript test files
For testing Python code
As a CLI tool using Node.js
As a part of build systems like Gulp.js and Grunt.js
As a library in Node.js

How to Use Jasmine?

- Download the standard library files and implement the same in your application.
- On successful download, unzip the zip file to find sub-folders inside the file.



Unit Testing

- Performed during the development of an application by developers
- Is the first level of testing
- Follows a white-box testing technique
- Helps fix bugs in the initial development cycle
- Enables developers to make quick changes in the code base

Unit testing tools:

JUnit 5 PHPUnit JMockit

Integration Testing

- Executed by testers
- Tests integration between software modules
- Individual units of a program are combined and tested as a group
- Follows black-box testing
- Should be performed by bottom-up and top-down methods
- Detects the errors related to the interface

Integration testing tools:

LDRA

Protractor
end to end testing for AngularJS

CITRUS

Unit Testing vs. Integration Testing

Features	Unit Testing	Integration Testing
Functionality	A small module or piece of a code of an application is tested	Individual modules are combined and tested as a group
Speed	Fast	Slow
Complexity	Less complex	More complex
Dependency	No dependency	Requires dependencies
Test conductor	Conducted by developers	Conducted by testers
Order of testing	Performed at the beginning stage	Performed after unit testing
Maintenance	Low maintenance	High maintenance

Setup and Write a Jasmine Test

Duration: 30 min.

Problem Statement:

You are given a project to install and demonstrate the Jasmine test.

Assisted Practice: Guidelines

Steps to perform installation and testing of Jasmine:

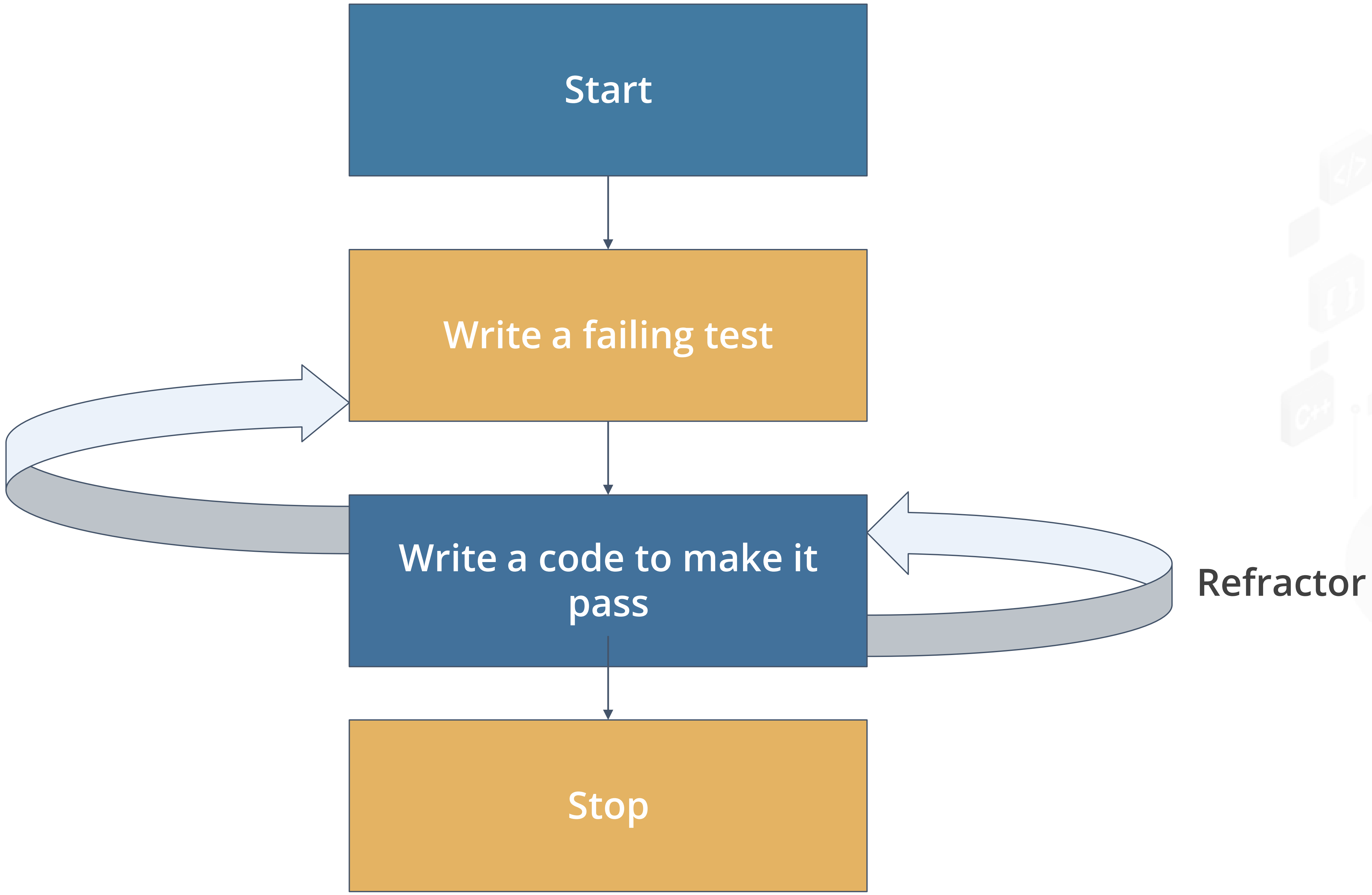
1. Download Jasmine
2. Write a test using Jasmine
3. Execute the test

FULL STACK

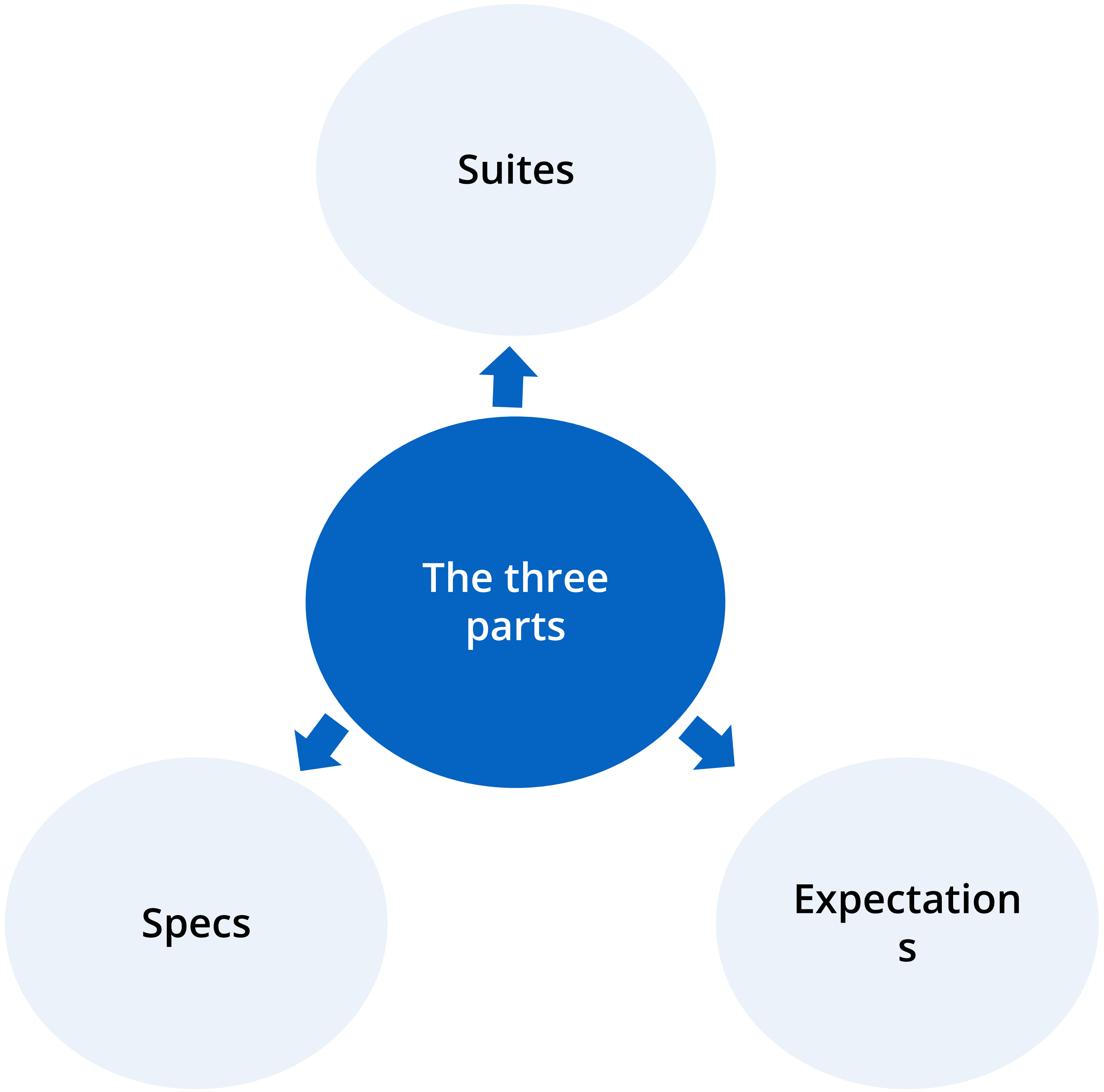
Jasmine Structures

BDD Architecture

Jasmine follows the Behavioral Driven Development (BDD) framework.



Jasmine Tests



Suites, Specs, and Expectations

Suites

A test suite begins with a call to the global Jasmine function *describe* with two parameters:

- String: A name or title for a spec suite
- Function: A block of code that implements suite

Specs

Defined by calling the global Jasmine function *it* with two parameters:

- String: A title for a spec
- Function: A spec or test

Expectations

- Built with the *expect* function which takes a value.
- Chained with a *matcher* function, which takes an expected value.

Suites, Specs, and Expectations

Syntax

```
//Suite
  describe(’Spec name', function () {
//Spec
      it(’what you are testing', function () {
//Expectation
           expect(Value).matcher();
});
});
```

Suites

Specs

Expectations

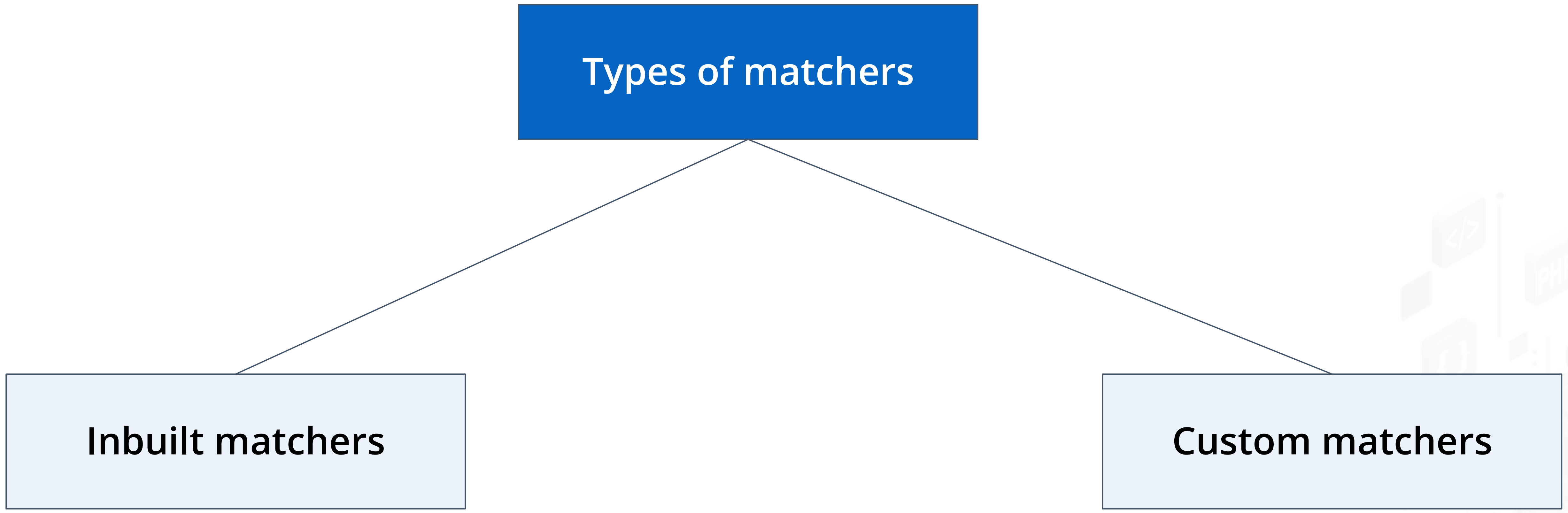
Matchers

- Implements a boolean comparison between the actual value and the expected value
- Reports to Jasmine and then passes or fails the spec
- Evaluates a negative assertion by chaining the call to *expect* with a *not* before calling the matcher

Examples of matchers:

- toBe(true)
- toEqual(10)
- toMatch(/bar/)
- toBeGreaterThan(6)
- toBeLessThan(4)
- not.toBe(true)
- toThrow()

Matchers



- Inbuilt in the Jasmine framework
- Can be used implicitly

- Not present in the inbuilt system library of the Jasmine framework
- Should be defined explicitly

Inbuilt and Custom Matchers

Syntax

```
    describe("A suite", function() {
         it("contains spec ", function() {
                expect(true).toBe(true);
    });
        it("contains spec ", function() {
                expect(true).toBe(true);
    });
});
```

Inbuilt Matchers

Syntax

```
var customMatchers = {
      customMatcherName: function (util,customEqualityTesters) {
     return {
               compare: function (actual, expected) {
               return {
                   pass: actual=== expected,
                   message: "Expected actual to equal expected "
               };
        }};
        }
});
         beforeEach(function() {
          jasmine.addMatchers(customMatchers);
});
```

Custom Matchers

Stubs

- Spy is a special function that records how it is called.
- Stub can be considered as a spy with behavior.

Stubs can be used to:

- Control individual method behavior for a specific test case
- Prevent a method from making side effects such as communicating with the outside world

Spies

- Jasmine tests have double functions called spies.
- Exists in the *describe* or *it* block and will be removed after each *spec.*
- We use spies when we want to track:
  - If a function has been called by the system under test (SUT)
  - How many times it has been called
  - Which arguments were passed

Syntax

```
variable= {

variable1: function(value) {

 variable3 = value;

}

spyOn(variable, ‘statement ');
```

Spies

- There are special matchers for interacting with spies:
  - The *toHaveBeenCalled* matcher will return true if the spy was called.
  - The *toHaveBeenCalledWith* matcher will return true if the argument list matches any of the recorded calls to the spy.

Syntax of special matchers

```
expect(variable.statement).toHaveBeenCalled();
expect(variable.statement).toHaveBeenCalledWith(value);
```

Spies

Spies matchers	Function
and.callThrough	By chaining the spy with *and.callThrough*, the spy will track all calls and represents the actual implementation.
and.returnValue	By chaining the spy with *and.returnValue*, all calls to the function will return a specific value.
and.callFake	By chaining the spy with *and.callFake*, all calls to the spy will be delegated to the supplied function.
and.throwError	By chaining the spy with *and.throwError*, all calls to the spy will throw the specified value as an error.
and.stub	When a calling strategy is used for a spy, the original stubbing behavior can be returned at any time with *and.stub*.

Source: https://jasmine.github.io/2.0/introduction.html

Spies

Spies matchers	Function
jasmine.createSpy	When there is no function to spy on, *jasmine.createSpy* can create a *bare* spy which can track calls, arguments, and other general tasks that are performed by spies.
jasmine.createSpyObj	In order to create a mock with multiple spies, use *jasmine.createSpyObj* and pass an array of strings. It returns an object that has a property for each string which is a spy.

Source: https://jasmine.github.io/2.0/introduction.html

Mocks

- Preprogrammed with expectations that form a specification of the expected calls
- Throws an exception in case of an unexpected call
- Checks during verification to ensure that all expected calls are received
- Verifies itself to check whether it has been used correctly by the SUT
- Tests interaction of SUT with a collaborator that communicates with the outside world

Syntax

```
<pre lang="javascript">var variable1 = function(){};
variable1.prototype.callMe = function() {};
var variable1 = mock( variable1 );
variable1.callMe();
 expect( variable1.callMe ).toHaveBeenCalled();
```

Stubs vs. Spies vs. Mocks

	Stubs	Spies	Mocks
Verification	State	Behavior	Behavior
Testing	Manual	Manual	Automatic

Setup and Teardown

In order to help a test suite DRY up any duplicated setup and teardown code, Jasmine provides the global beforeEach, afterEach, beforeAll, and afterAll functions.

Syntax

```
describe("A spec using beforeEach and afterEach",
function() {
  var variable1 = 0;
  beforeEach(function() {
    variable1 += 1;
  });
  afterEach(function() {
    variable1 = 0;
  });
  it("is just a function, so it can contain any code",
function() {
    expect(variable1).toEqual(1);
  });
  it("can have more than one expectation", function() {
    expect(variable1).toEqual(1);
    expect(true).toEqual(true);
  });
});
```

beforeEach block

afterEach block

Setup and Teardown

Syntax

```
describe("A spec using beforeAll and afterAll", function() {
  var variable1;
  beforeAll(function() {
    variable1 = 1;
  });
  afterAll(function() {
    variable1 = 0;
  });
  it("sets the initial value of foo before specs run", function() {
    expect(foo).toEqual(1);
    variable1 += 1;
  });
  it("does not reset foo between specs", function() {
    expect(variable1).toEqual(2);
  });
});
```

beforeAll block

afterAll block

Setup and Teardown

- Another way to share variables between a *beforeEach, it,* and *afterEach* is through the *this* keyword.
- Each spec's *beforeEach,it,* and *afterEach* has the *this* as the same empty object that is set back to empty for the next spec's *beforeEach,it,* and *afterEach*.

Syntax

```
describe("A spec", function() {
  beforeEach(function() {
    this.variable1 = 0;
  });
  it("can use the `this` to share state", function() {
    expect(this.variable1).toEqual(0);
    this.variable2 = "statement";
  });
  it("`this` created for the next spec", function() {
    expect(this.variable1).toEqual(0);
    expect(this.variable2).toBe(undefined);
  });
});
```

this keyword

Describe Blocks

Nesting describe blocks can be nested, with specs defined at any level.

Syntax

```
describe("A spec", function() {
   var variable1;
        beforeEach(function() {  variable1 = 0;   variable1 += 1;  });
          afterEach(function() {  variable1 = 0;  });
              it("is just a function, so it can contain any code", function() {
                   expect(variable1).toEqual(1);  });
               it("can have more than one expectation", function() {
                    expect(variable1).toEqual(1);  expect(true).toEqual(true); });
       describe("nested inside a second describe", function() {
            var variable2; beforeEach(function() {  variable2 = 1; });
               it("can reference both scopes as needed", function() {
                  expect(variable1).toEqual(variable2);
       });
    });
});
```

Before a spec is executed, Jasmine walks down the tree executing each *beforeEach* function in order.

After the spec is executed, Jasmine walks through the *afterEach* function similarly.

Working with Jasmine Structures

Duration: 90 min.

Problem Statement:

You are given a project to demonstrate the structures of Jasmine test.

Assisted Practice: Guidelines

Steps to follow the structure test of Jasmine:

1. Write a Javascript code to add and subtract two numbers
2. Test the code
3. Write Jasmine Expect statement
4. Use Jasmine after and before the statement

Key Takeaways

- Jasmine is an open-source JavaScript framework capable of testing any kind of JavaScript application.
- Unit testing is done by the developers, whereas integration testing is done by the testers.
- Jasmine follows the Behavioral Driven Development (BDD) framework.
- Jasmine has different structures such as matchers, spies, stubs, mocks, setup, and teardown.

Create and Test Calculator Application

LESSON-END PROJECT

Duration: 40 min.

Problem Statement:

Create an application to create a calculator and test it. Perform the following:

- Create a JavaScript calculator application
- Implement addition, subtraction, multiplication, and division operations
- Implement the Jasmine statement
- Execute Jasmine

Before the Next Class

Course: Hands-On Full-Stack Web Development with GraphQL and React

You should be able to:

- Define GraphQL
- Explain Apollo GraphQL
- Implement Apollo Client with React
- List Apollo Client Local and Remote Data