JUTA

The JavaScript Unit Testing Specification

Last update: 07/30/2017

What is JUTA?

JUTA stands for

JavaScript Unit Testing API.

It is a high level specification to write unit testing for JEC programs:

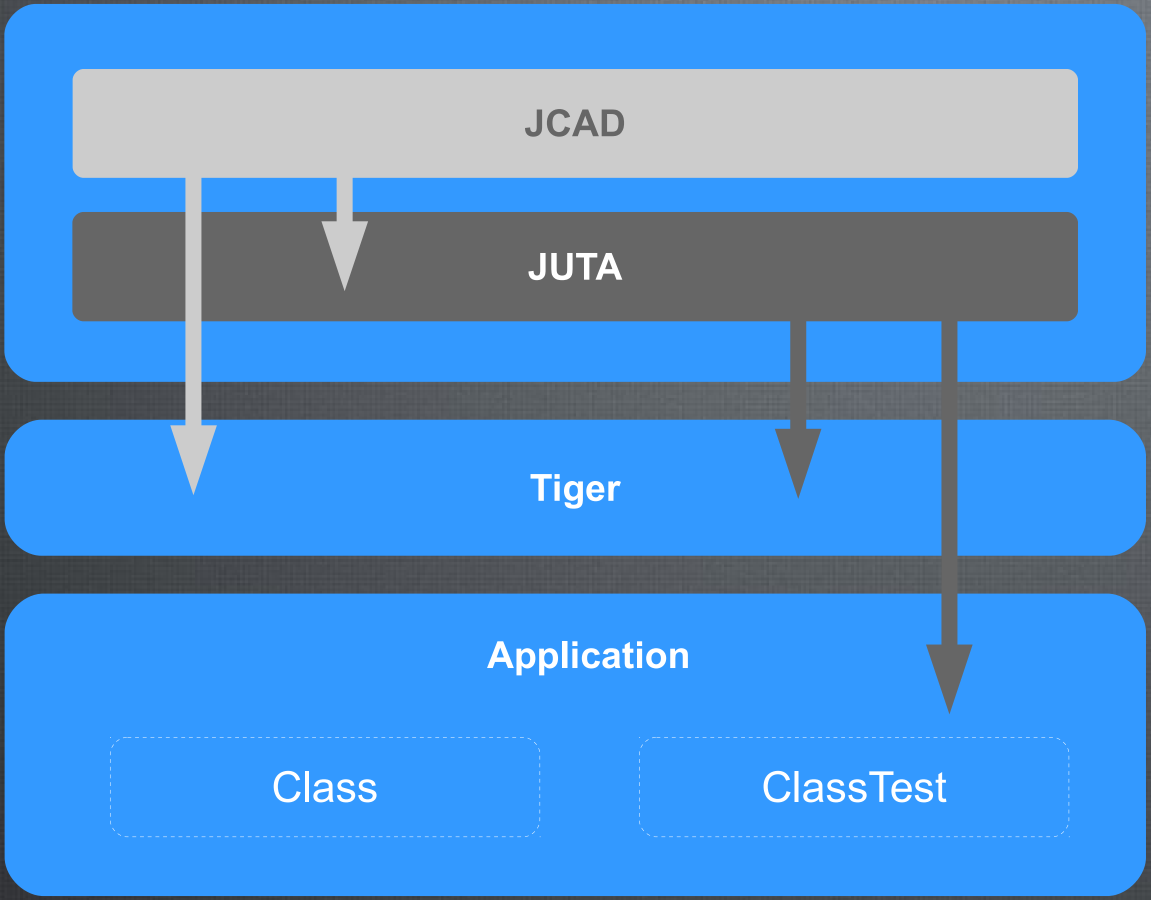
- easy-to-use
- portable
- based on TypeScript decorators
- object-centric

It provides an abstraction layer for all popular JavaScript unit testing frameworks (e.g. Mocha.js, Jasmine, etc.).

Tiger is the default JUTA implementation, built on top of **Mocha.js**.

JUTA Architecture

JUTA is built over the JCAD[1] API.



JCAD turns TypeScript decorators into an abstraction layer.

[1] JavaScript Connector API for Decorators

The Object-Centric Approach

The OOP approach takes benefits of encapsulation for better designing unit testing:

- it uses POJOs[1] to implement test suites
- it isolates each test case within an object member

[1] Plain Old JavaScript Objects

```
import { TestSuite, Test } from "jec-juta";
import { expect } from "chai";

@TestSuite({
  description: "Test the methods of the Greetings class"
})
export class GreetingsTest {

  @Test({
    description: "should return 'Hello World!'"
  })
  public sayHelloTest():void {
    let greetings:Greetings = new Greetings();
    expect(greetings.sayHello()).to.equal("Hello World!");
  }
}
```

List of JUTA Annotations 1/2

Basic annotations:

Annotation	Target	Description
`@TestSuite`	Class	When a class is annotated with `@TestSuite`, all tests in that class will be added to the test runner.
`@Test`	Method	Marks a method of a test class as part of the test suite to be run by the test runner.
`@TestSuitesConfig`	Class	Provides configuration for all test suites available in the current test path. You typically use the `@TestSuitesConfig` annotation to specify the list of groups that test classes belong to.
`@Async`	Field	Indicates that the associated test case must be run asynchronously.
`@DataProvider`	Method	Marks a method as supplying data for a test method. The annotated method must return an array of objects where each object can be assigned the parameter list of the test method.

List of JUTA Annotations 2/2

Fixture annotations:

Annotation	Target	Instantiation Policy	Description
`@BeforeClass`	Static Method	SINGLE	Indicates that the annotated static method will be run before the first test method in the current class is invoked.
`@AfterClass`	Static Method	SINGLE	Indicates that the annotated static method will be run after the last test method in the current class is invoked.
`@BeforeAll`	Method	MULTIPLE	Indicates that the annotated method will be run before the first test method in the current class is invoked.
`@AfterAll`	Method	MULTIPLE	Indicates that the annotated method will be run after the last test method in the current class is invoked.
`@Before`	Method	---	Indicates that the annotated method will be run before each test method in the current class is invoked.
`@After`	Method	---	Indicates that the annotated method will be run after each test method in the current class is invoked.

Test Isolation Principle

By default, JUTA creates only one instance of the test class to execute all `@Test` methods.

To apply the test isolation principle, you set the `instantiationPolicy` property of the `@TestSuite` decorator to `InstantiationPolicy.MULTIPLE`.

This will force the test runner to execute each `@Test` method in a new instance:

```
import { TestSuite, Test, InstantiationPolicy } from "jec-juta";

@TestSuite({
  description: "All test methods will be run in a new instance of GreetingsTest",
  instantiationPolicy: InstantiationPolicy.MULTIPLE
})
export class GreetingsTest {

  // Your test cases here...

}
```

Assertions

JUTA allows you to use any assertion library you wish.

You just have to import an assertion library and to use it in the body of a `@Test` method:

```
import { TestSuite, Test } from "jec-juta";
import { expect } from "chai";

@TestSuite({
  description: "Test assertions"
})
export class AssertionTest  {

  @Test({
    description: "should validate the result of the sum"
  })
  public testEqual():void {
    expect(2).to.equal(1 + 1);
  }
}
```

Disabling Tests

Sometimes you want to temporarily disable a test or a group of tests. Both, `@TestSuite` and `@Test`, decorators implement a `disabled` property that prevents tests executions:

```
import { TestSuite, Test } from "jec-juta";

@TestSuite({
  description: "Some test methods in this test class will be ignored"
})
export class GreetingsTest {

  @Test({
    description: "this test case will be run"
  })
  public sayHelloTest():void {}

  @Test({
    description: "this test case will be ignored",
    disabled: true
  })
  public toStringTest():void {}
}
```

Asynchronous Testing

You can test asynchronous code by using the `@Async` decorator, associated with a callback method:

```
@Test({
  description: "asynchronous test case"
  timeout: 6000
})
public asyncMethodTest(@Async done:Function):void {
  this.db.findUser(10, (user:User))=>{
    expect(user.name).to.equal("DOE");
    done();
  });
}
```

the `@Async` decorator can be passed as parameter of methods associated with the following decorators:

- `@Test`
- `@BeforeAll`
- `@BeforeClass`
- `@Before`
- `@AfterAll`
- `@AfterClass`
- `@After`

Ordering Tests

By setting the `testOrder` property of the `TestSuiteParams` interface, you can specify the execution order of test method invocations.

```
import { TestSuite, Test, TestsSorter } from "jec-juta";

@TestSuite({
  description: "Test methods are executed in numeric ascending order",
  testOrder: TestSorters.NAME_ORDER_ASCENDING
})
export class MyClassToTest {

  @Test({
    description: "this test will be run first"
    order: 1
  })
  public method1ToTest():void {}

  @Test({
    description: "this test case will be run after method1ToTest"
    order: 2
  })
  public method2ToTest():void {}
}
```

Running Tests

You must install a JUTA implementation for running tests.
Tiger is the default JEC implementation, built on the top of Mocha.js.

You configure Tiger with a basic script file in order to run tests:

```
import { TestStats } from "jec-juta";
import { Tiger, TigerFactory } from "jec-tiger";

let factory:TigerFactory = new TigerFactory();
let tiger:Tiger = factory.create();
tiger.process((stats:TestStats) => {
  if(stats.error) console.error(stats.error);
});
```

Add the script reference to your `package.json` file:

```
"scripts": {
  "test": "mocha test-config"
}
```

You start the Tiger test runner with the npm `test` command:

```
$ npm test
```

Where to go from here?

For more information and documentation on JUTA and the Tiger framework visit:

- JUTA Wiki
- Tiger Framework
- Sample project

JUTA and the Tiger framework are parts of the JEC project:

- JEC project on GitHub

JEC implementations that are
Based on JUTA and Tiger:

- jec-glasscat
- jec-glasscat-core
- jec-glasscat-cli
- jec-sandcat
- jec-wildcat

JEC
JavaScript Entreprise Container