https://shorturl.at/kAHNV

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Unit testing using Chai and Mocha**

**Microservices in Node.js**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

================================================================

Unit testing using Chai and Mocha

================================================================

* Chai and Mocha serve different purposes within the testing ecosystem.
* Chai focuses on assertions, providing a flexible and extensible library,
* While Mocha serves as a comprehensive testing framework with support for various test structures, hooks, and asynchronous testing.
* Often, they are used together, with Mocha providing the test structure and Chai providing the assertion capabilities.

Chai

https://www.chaijs.com/

https://www.chaijs.com/api/assert/

1. Assertion Library: Chai is primarily an assertion library, providing a set of functions to make assertions in your tests.
2. Extensible: Chai is extensible, allowing you to use different styles of assertions, such as should, expect, or assert.
3. Styles of Assertions:
   1. expect: BDD style assertions.
   2. should: Chainable assertions.
   3. assert: Classic assert style.
4. Chaining: Chai's should style allows for natural language chaining of assertions.
5. Plugins: Chai supports plugins, enabling you to extend its functionality according to your needs.
6. Integration: Can be used with different testing frameworks, not limited to Mocha.
7. Readable Output: Chai provides readable and informative error messages when assertions fail.
8. Browser Compatibility: Chai can be used in both Node.js and browser environments.

Behavioral Driven Development (BDD), Test-Driven Development (TDD), and Acceptance Test-Driven Development (ATDD)

Mocha

https://mochajs.org/

1. Testing Framework: Mocha is a testing framework for Node.js, providing a test structure and runner.
2. Describe and It Blocks: Mocha uses describe and it blocks to structure and organize test suites.
3. Hooks: Mocha supports various hooks like before, after, beforeEach, and afterEach to set up and tear down test fixtures.
4. Async Testing: Mocha has built-in support for asynchronous testing using callbacks, Promises, or async/await.
5. Multiple Reporters: Mocha supports multiple reporters for test output, including spec, dot, and tap.
6. Timeouts: Mocha allows you to set timeout values for individual tests and suites.
7. Parallel Execution: Mocha can run tests in parallel, improving test suite performance.
8. Generators Support: Mocha supports generators, which can be useful for testing asynchronous code.
9. Browser Compatibility: Mocha can be used in both Node.js and browser environments, making it versatile.
10. Custom Reporters: You can create custom reporters for Mocha, tailoring the test output to your needs.
11. Wide Adoption: Mocha is widely adopted in the Node.js community and is often the default choice for testing frameworks.
12. Test Coverage Integration: Mocha can be integrated with tools like Istanbul for test coverage reporting.

*Use Mongooseeg*

Install dev dependencies (chai@4.3.3 chai-http@4.3.0 mocha@8.3.1)

> npm install --save-dev chai chai-http mocha

Change database to “testdb”

\*\*\*url.js\*\*\*

module.exports = "mongodb+srv://admin:admin@mdb.vtkja.mongodb.net/testdb"

Download mocha configuration file from

https://github.com/mochajs/mocha/blob/master/example/config/.mocharc.json

paste in working directory and remove comments

create directory ‘test’ to keep all test files

create following directory structure

<>

test

product.test.js

in package.json update script test as

"scripts": {

"test": "mocha --exit",

. . .

}

Fire command

>npm test -> make sure that there will not be any errors

Have a trial run

\*\*\*product.test.js\*\*\*

//import chai

const chai = require('chai')

const expect = chai.expect

describe("First test collection", () => {

it("Should test values", () => {

let expectedValue = 10

let actualValue = 5

expect(actualValue).to.be.equal(expectedValue)

})

})

Now create a simple test api

\*\*\*productApis.js\*\*\*

//trial api

const trial = async(req,res)=>{

res.status(200).send('Trial api responce')

}

. . .

module.exports = {

trial,

. . .

}

\*\*\*productRoutes.js\*\*\*

. . .

//Trial route

router.get("/fetch/trial",productApi.trial)

. . .

Update server.js with exporting rest object

\*\*\*server.js\*\*\*

. . .

module.exports = app

- - x - -

\*\*\*product.test.js\*\*\*

. . .

describe('Trial api testing', () => {

it("Trial api status code 200", (done) => {

chai.request(server)

.get('/fetch/trial')

.end((req, res) => {

console.log(res.body)

//res.should.have.status(200) //test 1

const actualValue = res.body.message

expect(actualValue).to.be.equal('Trial api responce updated') //test 2

done()

})

})

})

. . .

Now Time to test remaining apis

\*\*\*product.test.js\*\*\*

. . .

//import chai

const chai = require('chai')

const expect = chai.expect

const server = require('../server')

const chaiHttp = require('chai-http') //refer https://www.chaijs.com/plugins/chai-http/

chai.use(chaiHttp)

describe("First test collection", () => {

it("Should test values", () => {

let expectedValue = 10

let actualValue = 5

expect(actualValue).to.be.equal(expectedValue)

})

})

describe('Trial api testing', () => {

it("Trial api status code 200", (done) => {

chai.request(server)

.get('/fetch/trial')

.end((req, res) => {

expect(res.status).to.be.equal(200) //test1

const actualValue = res.body.message

expect(actualValue).to.be.equal('Trial api responce') //test 2

done()

})

})

})

describe('Initial Fetch', () => {

it('Initially db contain 0 records', (done) => {

chai.request(server)

.get("/fetch")

.end((req,res)=>{

expect(res.status).to.be.equal(200) //test1

expect(res.body).to.be.a('array') //test2

expect(res.body.length).to.be.eql(0)//test3

done()

})

})

})

describe('Insert Record', () => {

it('Should insert a product', (done) => {

let product = {

"p\_id": 111,

"p\_name": "P\_one",

"p\_cost": 10000

}

chai.request(server)

.post('/insert')

.send(product)

.end((req,res)=>{

expect(res.status).to.be.equal(200) //test1

done()

})

})

})

describe('Update Record', () => {

it('Should update a product', (done) => {

let product = {

"p\_id": 111,

"p\_name": "P\_one\_updated",

"p\_cost": 11111

}

chai.request(server)

.put('/update')

.send(product)

.end((err, res) => {

expect(res.status).to.be.equal(200) //test1

expect(res.body.update).to.be.equal('success')

done()

})

})

})

describe('Delete Record', () => {

it('Should delete a product', (done) => {

let product = {

"p\_id": 111

}

chai.request(server)

.delete('/delete')

.send(product)

.end((err, res) => {

expect(res.status).to.be.equal(200) //test1

expect(res.body.delete).to.be.equal('success')

done()

})

})

})

================================================================

Microservices in Node.js

================================================================

* Microservices in Node.js refer to an architectural approach where a software application is broken down into small, independent services that communicate with each other over a network.
* Each microservice is designed to perform a specific business function and can be developed, deployed, and scaled independently.
* Node.js, being a JavaScript runtime that is well-suited for building scalable and efficient server-side applications, is often used to implement microservices.

Key characteristics of microservices in Node.js include:

* Independence: Each microservice operates independently, with its own codebase, database, and dependencies. This independence allows teams to develop, deploy, and scale services separately, without affecting the entire application.
* Decentralized Data Management: Microservices typically have their own databases, which can be of different types (SQL, NoSQL, etc.). This decentralization of data management helps avoid dependencies on a single, monolithic database.
* API Communication: Microservices communicate with each other through well-defined APIs (Application Programming Interfaces). RESTful APIs or other communication protocols are commonly used to enable interaction between microservices.
* Scalability: As each microservice is independent, it can be scaled independently based on its specific workload. This allows for efficient resource utilization and better performance.
* Technology Diversity: Different microservices within a system can be implemented using different technologies or programming languages. Node.js is often chosen for its efficiency and non-blocking I/O capabilities, but other technologies can be used where they are better suited.
* Resilience and Fault Isolation: If one microservice fails, it does not necessarily bring down the entire application. The failure is contained within that specific service, and the rest of the application can continue to operate.

Monolithic and microservices architectures are two different approaches to designing and building software applications. Here are the key differences between monolithic and microservices architectures:

Architecture:

* Monolithic: In a monolithic architecture, the entire application is developed, deployed, and scaled as a single, unified unit. All components, such as the user interface, business logic, and data access layer, are tightly interconnected within the same codebase.
* Microservices: Microservices architecture breaks down the application into small, independent services that operate as separate entities. Each microservice focuses on a specific business capability and communicates with other microservices through APIs.

Development and Deployment:

* Monolithic: Developers work on a single codebase, and the entire application is deployed as a single unit. Updates or changes to any part of the application may require redeploying the entire system.
* Microservices: Development and deployment are independent for each microservice. Teams can work on and deploy microservices separately, allowing for faster development cycles and more frequent releases.

Scalability:

* Monolithic: Scaling a monolithic application typically involves replicating the entire application, even if only a specific component requires additional resources.
* Microservices: Microservices can be individually scaled based on their specific needs. This allows for more efficient resource utilization and better scalability.

Technology Stack:

* Monolithic: A monolithic application generally uses a uniform technology stack for all components.
* Microservices: Microservices provide flexibility in choosing different technologies for different services, allowing teams to use the most appropriate tools for each microservice.

Data Management:

* Monolithic: A monolithic application usually has a centralized database that all components access.
* Microservices: Each microservice can have its own database, promoting decentralization. Microservices communicate with each other through well-defined APIs.

Fault Isolation:

* Monolithic: A failure in any part of the monolithic application can potentially impact the entire system.
* Microservices: Failures are isolated to individual microservices, limiting the impact on the overall system. Other microservices can continue to function even if one service encounters issues.

Complexity and Maintenance:

* Monolithic: As applications grow, monolithic codebases can become complex and challenging to maintain. Changes to one part of the application may affect other parts.
* Microservices: Microservices promote modularity and ease of maintenance. Each microservice can be developed and maintained independently, reducing the overall complexity.

Communication:

* Monolithic: Components in a monolithic application communicate through direct function calls or shared memory.
* Microservices: Microservices communicate through well-defined APIs, often using lightweight protocols like HTTP/REST or message queues.

================================================================

================================================================