Develop a Reliable Backend with Node and Express

Mocha

Learning Objectives

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

- Examine Mocha and its components to gain insights into Mocha's structure and inner workings for effective utilization in testing scenarios
- Grasp the diversity of tests in Mocha to select and employ the most suitable tests for different contexts
- Comprehend the fixtures and plugins within Mocha, enabling efficient and enhanced testing practices
- Configure Mocha for testing with expertise, ensuring seamless integration into any testing setup

A Day in the Life of a MERN Stack Developer

Mr. Robin is working as a MERN stack developer in an organization looking to establish and maintain consistency in a product's performance and functional attributes with great database management. The goal is to test the full stack application based on the varied user needs.

To achieve all the above goals, along with some additional concepts, you will be learning a few features in this lesson that will help Robin test his application.

Introduction to Mocha

What Is Mocha?

Mocha is a JavaScript testing framework that runs both on Node.js and in the browser. It can be further described as:

- It provides developers with a flexible and easy-to-use platform for creating and running tests for JavaScript applications.
- It provides clarity to everyone on the team about their roles and responsibilities.
- It provides a way to write a structured code for testing the applications thoroughly, classifying them into test suites and test cases.

Features of Mocha.js Framework

- Offers a complete testing solution for JavaScript applications
- Runs tests in both Node.js and the browser
- Supports various testing approaches to accommodate developers' preferences
- Offers strong hooks to enable adaptable setup and teardown of test fixtures
- Is highly extensible, enabling the addition of custom functionality through plugins
- Provides a wide variety of built-in reporters and permits custom reporters to meet particular requirements

Installation of Mocha

You can install Mocha in the following two ways:

With npm globally

```
Syntax:
npm install --global Mocha
```

As a development dependency

```
Syntax:
npm install --save-dev Mocha
```

Mocha Syntax for Writing Tests

Mocha syntax is unique, as its blocks (describe and it) are identical to Jasmine's (both are quite similar to RSpec).

```
describe 'Task instance', ->    task1 = task2 = null
it 'should have a name', ->        task1 = new Task
'open the door'        task1.name.should.equal 'open
the door'
```

Modifying the JSON Output

JSON.parse() method is used for modifying the JSON output. Following are the various properties of the object options available with the JSON.parse() method:

Property	Purpose
inflate	This enables or disables the deflated or compressed bodies' handling. The default value is True.
limit	This option controls the maximum size of the request body.
reviver	As the second argument, this option is passed to the JSON.parse method.
strict	This toggles the acceptance of arrays or objects.
type	This specifies the media type for the middleware to be parsed.

Hello World Unit Test with Mocha

Duration: 10 min.

Problem Statement:

You have been asked to install the Mocha third party module and test Node.js application using that module

Assisted Practice: Guidelines

Steps to Perform:

1. Check whether Node.js is installed or not
2. Create the project structure and package.json file and installing Mocha module
3. Create a simple Node.js program and test file

Mocha Overview

Run Cycle Overview: Serial and Parallel Mode

A run cycle is a series of activities that take place when a test is run.

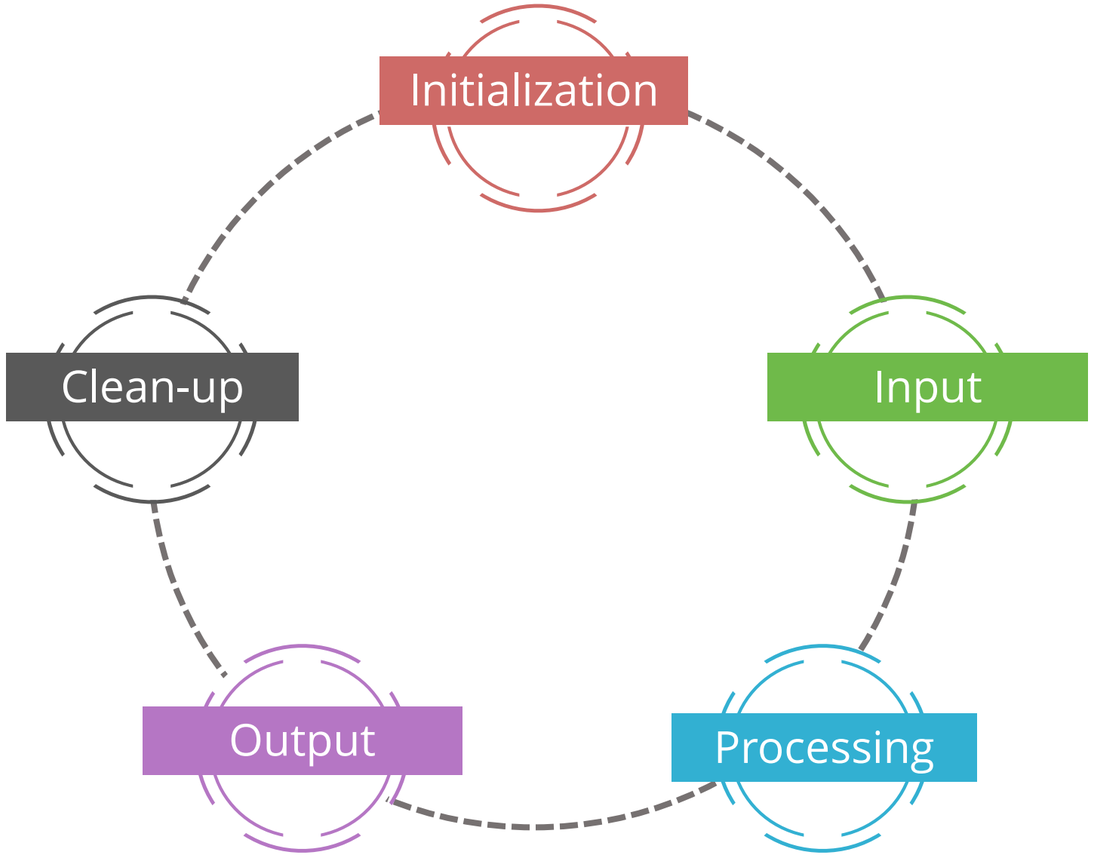
In serial mode, tests are run sequentially, with one test finishing before the next one begins.

In parallel mode, tests are run concurrently to reduce the overall time required to conduct tests.

Parallel mode can be used when multiple tests need to be run simultaneously.

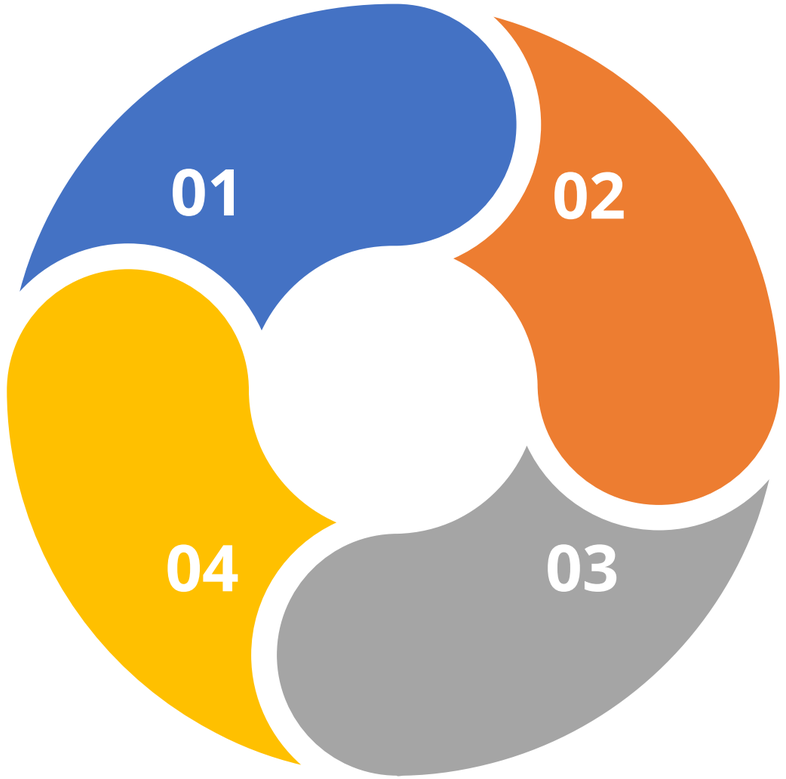
Serial Mode

The run cycle in serial mode is usually executed as follows:



Parallel Mode

Parallel run cycles typically need extra steps to coordinate the execution of several threads, such as:



Task allocation

Thread creation

Thread synchronization

Thread termination

Double Callbacks

Double callbacks occur when a procedure is called more than once without waiting for the first callback to finish.

This situation leads to overlapping or simultaneous execution of the same callback function, potentially causing unexpected behavior or issues in the program.

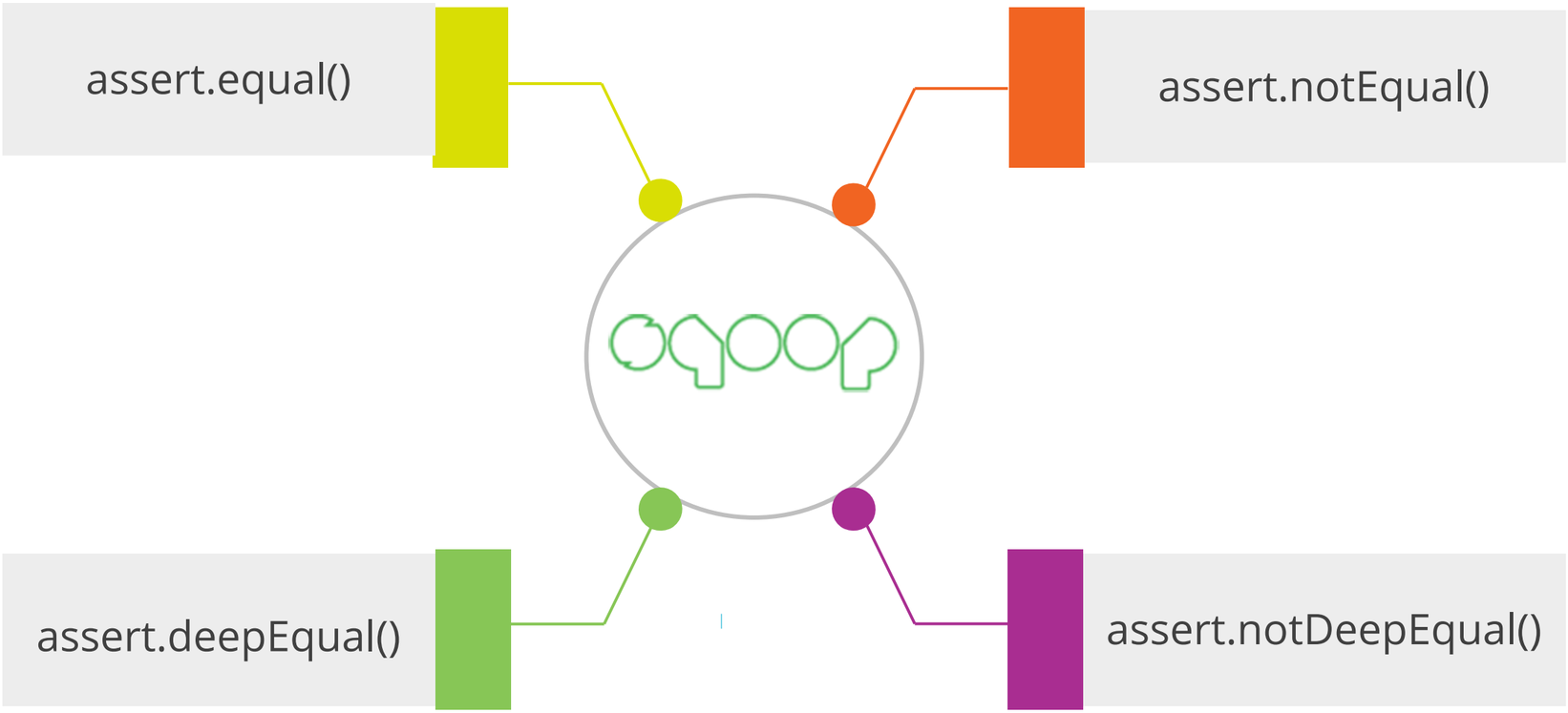
How to Catch Double Callbacks?

The steps to catch double callbacks are as follows:

- Create a flag or a lock property to maintain a note of whether the function or listener has already been executed
- Initialize the lock or flag property to false
- Verify whether the signal or lock value is true inside the method or listener
- Respond accordingly as a double reply has happened (For example, logging an error, displaying a message to the user, and so on)
- Set the flag or lock variable to true and carry out the intended feature if it is untrue
- Reset the signal or lock variable to false once the feature is complete so that the function or listener can be called again in the future

Types of Assertions

In Mocha, assertions are used to validate the outcomes or expected behavior of your code. Different kinds of assertion functions used to test a particular behavior are:



Types of Assertions: assert.equal()

assert.equal()

- It is a method used in unit testing frameworks such as Mocha or Jest to assert that a given value is identical to an expected value.
- In frameworks such as Mocha, it can be used in a test case to ensure that a function or piece of code produces the desired result.
- When the two values being compared are identical, the test will pass. In non-strict mode, it functions the same as the == operator.

assert.equal(): Example

Example

```
const assert = require('assert');

describe(‘multiplication', function() {
  it('should return the multiplication of two numbers',
function() {
    assert.equal(1 * 2, 2);
    assert.equal(10 * 20, 200);
    assert.equal(0 + 0, 0);
  });
});
```

Types of Assertions: assert.notEqual()

```
assert.notEqual()
```

- It is a method in the assertion library integrated into the Mocha testing framework.
- It is used to state that two numbers are not the same.
- The assert.notEqual() method is the inverse of the prior example's equal method.
- When the two values you're comparing are equal or identical, it will issue an assertion error.

assert.notEqual(): Example

Example

```
const assert = require('assert');

describe('my test', function() {
  it('should check if a is not equal to b', function() {
    const a = 250;
    const b = 400;
    assert.notEqual(a, b, ‘a should not be equal to b');
  });
});
```

Types of Assertions: assert.DeepEqual()

assert.deepEqual()

- It is a method in Node.js's built-in assert module that checks two values for deep equivalence.
- It compares the real and expected values and throws an error if they are not profoundly equal.
- It ensures deep equality to check even child components in the actual and expected parameter values.
- It compares the result of an array-based function to an expected array.
- It allows the test to pass when both arrays are identical.

assert.DeepEqual(): Example

Example

```
const assert = require('assert’);
const actual = { foo: 'bar', baz: { qux: 42 } };
const expected = { foo: 'bar', baz: { qux: 42 } };
assert.deepEqual(actual, expected, 'Objects should be deeply equal');
```

Types of Assertions: assert.notDeepEqual()

assert.notDeepEqual()

- It is a Node.js assert module method to test if two values are not profoundly equal, that is, if their properties and nested properties are not equal.
- As a result, this test will fail if the real and expected values are not equal.

assert.notDeepEqual(): Example

After installing the express module, use the command to check the express version in the command prompt.

Example

```
const assert = require('assert');

describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present',
function() {
      assert.deepEqual([21,22,23].indexOf(4), -1);
    });
  });
});
```

Assertion Libraries

Assertion libraries are used to determine whether a set of conditions is true or false and are frequently used in unit testing.

Examples of Assertion libraries are:

Chai

Jest

Jasmine

Asynchronous Code

Asynchronous code testing in Mocha necessitates extra effort. To indicate the completion of an asynchronous action using the **done function** or return a **Promise**.

When the asynchronous action is finished, the done function is sent to the test function and invoked.

Synchronous Code

It runs on a single thread and prevents other code from running until it completes.

In Mocha, synchronous code can be tested using the **it()** function's default behavior.

Arrow Functions

Arrow functions in Mocha can be used to define test cases and hooks, among other things.

Example: Using arrow functions to define test cases

```
describe(‘Sample test suite', () => {
  it('should return true when given true', () => {
    const result = myFunction(true);
    assert.strictEqual(result, true);
  });
});
```

Arrow Functions

Arrow functions in Mocha can be used to define test cases and hooks, among other things.

Example: Using arrow functions to define test cases

```
describe('Sample test suite', () => {
  let myValue;

  beforeEach(() => {
    myValue = 62;
  });

  it('must use the correct value', () => {
    assert.strictEqual(myValue, 62);
  });
});
```

Hooks

Hooks are preconditions and postconditions that are executed before and after the tests.

Inbuilt BDD Hooks

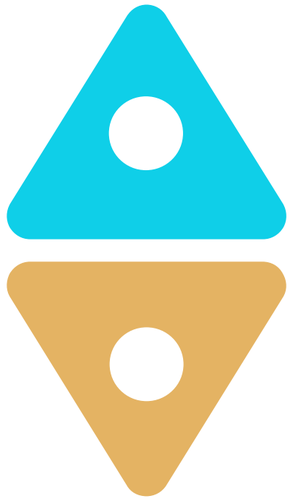
Behavior Driven Development (BDD) is a software development methodology that enables testers and business analysts to build test cases. Mocha provides several inbuilt BDD hooks:

before Hook	Runs once before all tests in the suite
after Hook	Runs once after the tests in the suite
beforeEach Hook	Runs before each test in the suite
afterEach Hook	Runs after all tests in the suite

Root-Level Hooks

Root-level hooks are routines that run before or after all test suites or individual tests.
In Mocha, there are two kinds of root-level hooks:

before and **beforeEach**



after and **afterEach**

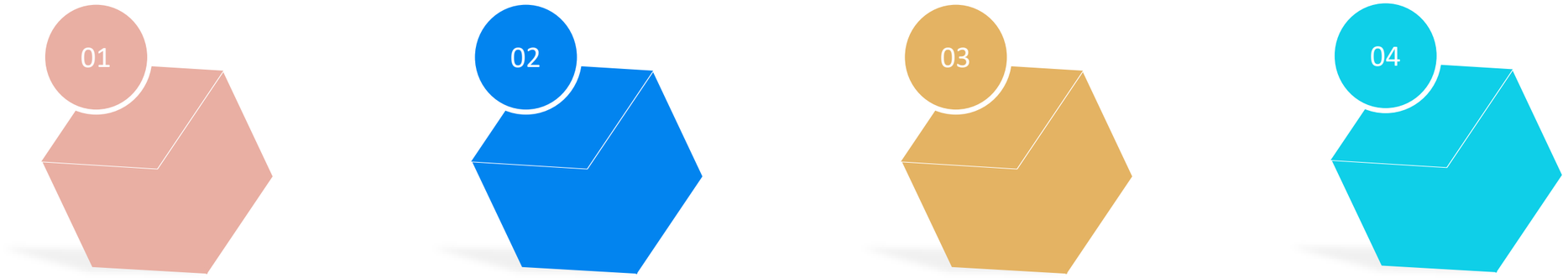
Delayed Root Suite

It is a way to organize tests that must be run asynchronously, because they contain setup code that takes some time to complete.

```
describe("Test delayed root suite", function() {
  before(function(done) {
    setTimeout(done, 1000);
  });
  it("should do something", function() {
  });
  it("should do nothing", function() {
  });
});
const MochaInstance = new Mocha();
MochaInstance.addFile("my-test-file.js");
MochaInstance.run();
```

Mocha Structure

A Mocha test suite is composed of the following components:



Test Suite

Test Specs

Test Hooks

Assertions

Mocha Structure: Example

An example of simple Mocha suite:

```
describe('Math', function() {
  let n1 = 20;
  let n2 = 8;

  beforeEach(function() {
    console.log('Before starting each test spec');
  });

  afterEach(function() {
    console.log('After starting each test spec');
  });
```

Mocha Structure: Example

An example of simple Mocha suite:

```
before(function() {
   console.log('Before test suite');
 });

 after(function() {
   console.log('After test suite');
 });
```

Mocha Structure: Example

An example of simple Mocha suite:

```
 it('should add two numbers', function() {
    let result = n1 + n2;
    assert.equal(result, 25);
  });

  it('should subtract two numbers', function() {
    let result = n1 - n2;
    assert.equal(result, 7);
  });
});
```

Types of Tests in Mocha

Various tests that may be run in Mocha include:

01 Pending tests

02 Exclusive tests

03 Inclusive tests

04 Retry tests

05 Parallel tests

Pending Tests

Pending tests in Mocha can be identified by using the **pending** keyword instead of **describe** or **it**.

```
describe('My test suite', () => {
  it('do something', () => {
  });

  it('do nothing', () => {
  });
  it('should do this thing - PENDING');

  it('should do another thing - PENDING  more work');
});
```

Exclusive Tests

In Mocha, exclusive tests can be created using **it.only()** function.

```
describe('Sample test suite', function() {
  it.only('should run this test ', function() {
      });
  it('should not run this test', function() {
  });
});
```

It.only() function allows you to run only the specified test case and exclude all other tests.

Inclusive Tests

These are created to execute a particular set of test cases, despite the failure of one or more of the test cases.

```
describe('My test suite', function() {
  it(` perform test case 1', function() {
  });
    it(`perform test define. Only case 2', function() {
  });
  it(' perform test case 3', function() {
  });
});
```

Retry Tests

To retry a test in Mocha, the **retry** option is used.

```
describe('retries', function () {
   this.retries(5);

  beforeEach(function () {
    browser.get('http://www.google.com');
  });

  it('should succeed on the 4th try', function () {
     this.retries(2);
    expect($('.foo').isDisplayed()).to.eventually.be.true;
  });
});
```

Parallel Tests

There are several ways to accomplish parallelism with Mocha, including command-line options, plugins, and programmatically using Node.js.

```
describe('retries', function () {
   this.retries(5);

  beforeEach(function () {
    browser.get('http://www.google.com');
  });

  it('should succeed on the 4th try', function () {
     this.retries(2);
    expect($('.foo').isDisplayed()).to.eventually.be.true;
  });
});
```

Parallel Tests

Parallel tests in Mocha can be accomplished by executing multiple test files or suites concurrently.

01 Use the **--parallel** command-line option

```
mocha --parallel test
```

Parallel Tests

Parallel tests in Mocha can be accomplished by executing multiple test files or suites concurrently.

02	Use a plugin like **mocha-parallel-tests**

```
npm install mocha-parallel-tests
```

Dynamically Generating Tests

In Mocha, users can utilize the **it function**, which is used to define a test case, to dynamically generate tests.

```
describe('My test suite', function() {
  for (let j = 0; j < 10; j++) {
    it(`should return ${i} when calling myFunction with ${j}`, function() {
      const result = myFunction()j;
      assert.equal(result, j);
    });
  }
});
```

The **it function** accepts two parameters: a test case description and a function containing the test logic.

Timeouts in Mocha

The **‘this.timeout()’** method in Mocha allows you to establish a timeout for each test case.

This method accepts a milliseconds argument and sets a timeout for the current test case.

Users can shut down the testing environments and pay only for the time they are up and running.

Timeouts in Mocha

An example of simple 3000 milliseconds (3 secs)

```
it(‘complete in 3 seconds', function() {
  this.timeout(3000);
});
```

Diffs in Mocha

When a test case fails, Mocha displays the difference between the expected and actual values that triggered the failure.

This aids in swiftly determining what went wrong in the test scenario.

When executing Mocha from the command line, use the --diff option to allow diffs.

Users can also enable diffs programmatically by setting the diff option in the Mocha configuration object to true.

Diffs in Mocha

Example

```
const Mocha = new Mocha({
  diff: true
});
```

Arrow Functions

Problem Statement: **Duration: 10 min.**

You have been asked to perform testing with expression and arrow style for a Node.js sample app using Mocha

Assisted Practice: Guidelines

Steps to Perform:

1. Create the project structure and package.json file and installing Mocha module
2. Create simple Node.js program and test file

Asynchronous and Synchronous Testing

Problem Statement: **Duration: 10 min.**

You have been asked to perform asynchronous and synchronous testing for a Node.js sample app using Mocha

Assisted Practice: Guidelines

Steps to Perform:

1. Create the project structure and package.json file and installing Mocha module
2. Create a simple Node.js program and test file to test synchronous and asynchronous code

Assertion Libraries

Problem Statement: **Duration: 10 min.**

You have been asked to check Node.js program using assertion library.

Assisted Practice: Guidelines

Steps to Perform:

1. Create the project structure and package.json file and installing Mocha module
2. Create and test the code in the same file with different assertion library

Testing a Promise

Problem Statement:

Duration: 10 min.

You have been asked to test a Node.js app with promise object using Mocha

Assisted Practice: Guidelines

Steps to Perform:

1. Check whether Node.js is installed or not
2. Create the project structure and package.json file and installing Mocha module
3. Create Node.js app which returns the promise data with fake REST API program and test file to check the promise

Fetch API with Multiple Done()

Problem Statement: **Duration: 10 min.**

You have been asked to install node-fetch-commonJS to call REST API and test that API using Mocha

Assisted Practice: Guidelines

Steps to Perform:

1. Create the node app.js file which calls the third-party fake REST API node-fetch-commonJS module
2. Create a test file to check the REST API using done

Hooks in Mocha

Problem Statement:

Duration: 10 min.

You have been asked to install Mocha and verify its hook function

Assisted Practice: Guidelines

Steps to Perform:

1. Create the node app.js file, which is called a simple function
2. Create a test file to check this function with the hook concept

Mocha With Chai

Problem Statement: **Duration: 10 min.**

You have been asked to install Mocha with Chai library and test with different styles

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to Perform:

1. Create the project structure and package.json file and installing Mocha module
2. Test the code using assertion style, expect style, and should style

Dynamic Test Case Generation

Problem Statement: **Duration: 10 min.**

You have been asked to install Mocha and generate dynamic test case for testing Node.js application

Assisted Practice: Guidelines

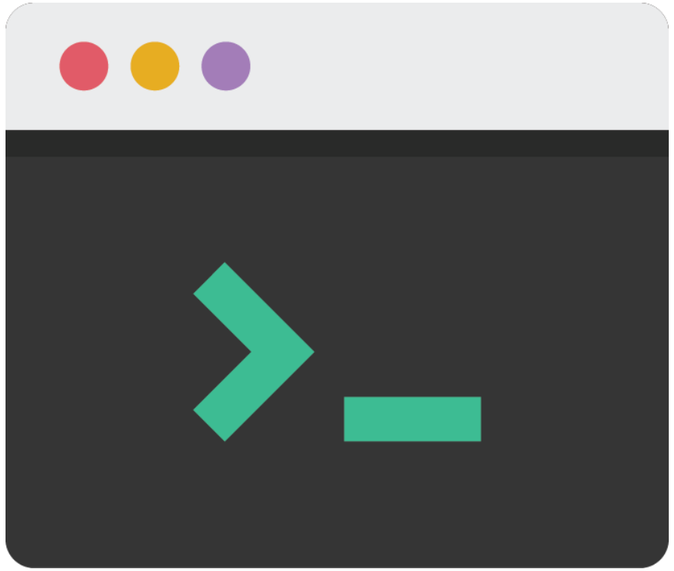
Steps to Perform:

1. Create the project structure and package.json file and installing Mocha module
2. Test the code and generating a test case

Command-Line Usage

What Is Command Line?

It's a program that accepts commands and sends them to the computer's operating system for execution.



The text-based computer interface is known as the command line.

List of Mocha Commands

mocha
This command runs all the tests in the current directory.

01

mocha --grep <pattern>
This command runs only the tests that match the specified pattern.

05

mocha <file>
This command runs a specific test file.

02

mocha –watch
This commaand watches for changes in the test files and automatically re-runs the tests.

04

mocha <directory>
This command runs all the tests in a specific directory.

03

MOCHA

List of Mocha Commands

06

mocha --timeout <ms>
This commanad sets the maximum time allowed for a test to run before it times out.

10

mocaha –colors
This command enables colors in the test output.

07

mocha --reporter <reporter> This command specifies the reporter used to display test results.

09

mocha –bail
This command stops running tests after the first test failure.

08

mocha --ui <interface>
This command specifies the user interface used to write tests.

Plugins, Fixtures, Interfaces, and Reporters in Mocha

Editor Plugins

Editor plugins enhance the testing process by offering capabilities like syntax highlighting, code completion, and test discovery.

Editor Plugins

Some of the famous editor plugins are as follows:

01 Mocha Sidebar

02 Mocha Test Explorer

03 Sublime Mocha

04 Atom Mocha Test Runner

05 Mocha Snippets

Global Fixture

A global fixture is a piece of code that runs once before each test in a test suite. It can provide a common environment or state required by all the tests.

The **before hook** defines global fixtures at the top level of a Mocha test suite.

Global Fixture

Properties of the global fixtures are as follows:

01 It works identically in parallel mode, watch mode, and serial mode.

02 It doesn't share a context with tests, suites, or other hooks.

03 It executes only once.

Global Fixture: Types

The types of global fixtures are as follows:

01

Global setup fixtures

02

Global teardown fixtures

Global Fixture

The code below can create a global setup fixture:

```
exports.mochaGlobalSetup = async function () {
  this.server = await startSomeServer({port:
process.env.TEST_PORT});
  console.log(`server successfully working on port
${this.server.port}`);
};
```

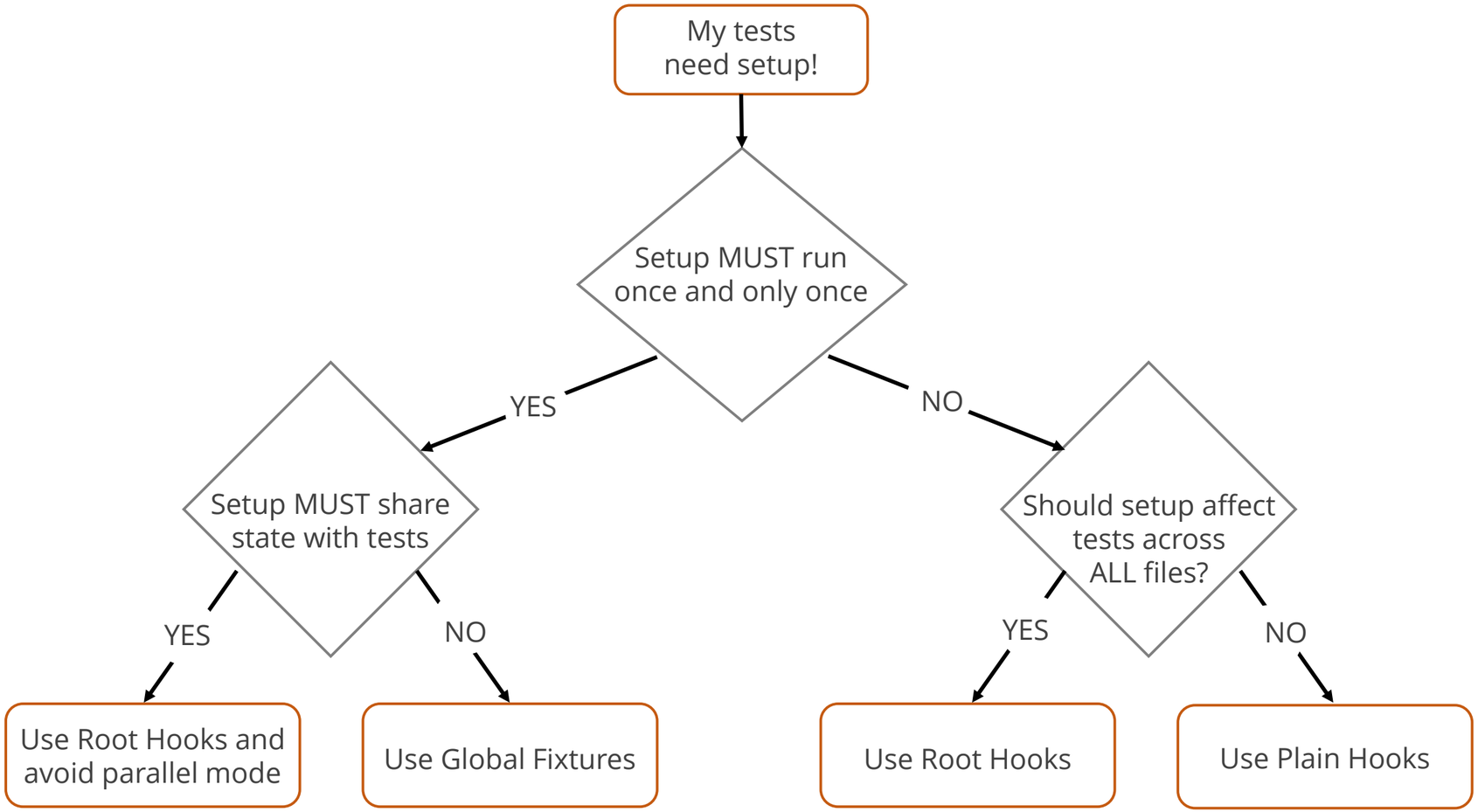
Global Fixture

The code below can create a global teardown fixture:

```
exports.mochaGlobalTeardown = async function () {
  await this.server.stop();
  console.log('server not working!');
```

Test Fixture Decision: Tree Wizard

The use of hooks, root hook plugins, or global fixtures can be decided using this flowchart:

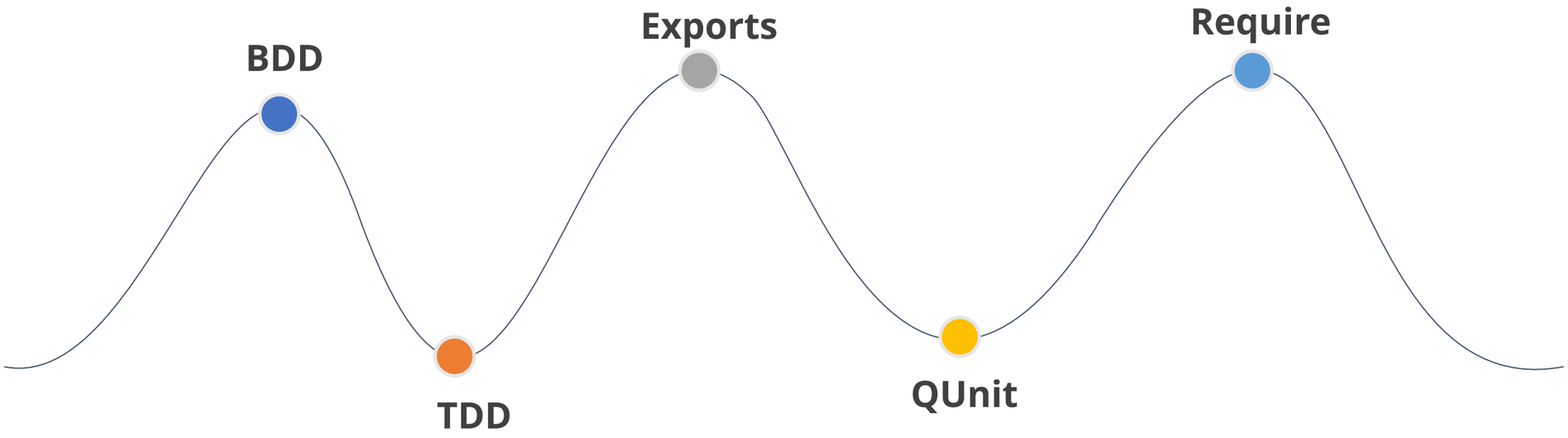


Interface

Mocha's interface module defines test style, offering flexibility with multiple interfaces like BDD, TDD, QUnit. It structures tests as per preferred styles or conventions.

Interface

The interfaces in Mocha are:

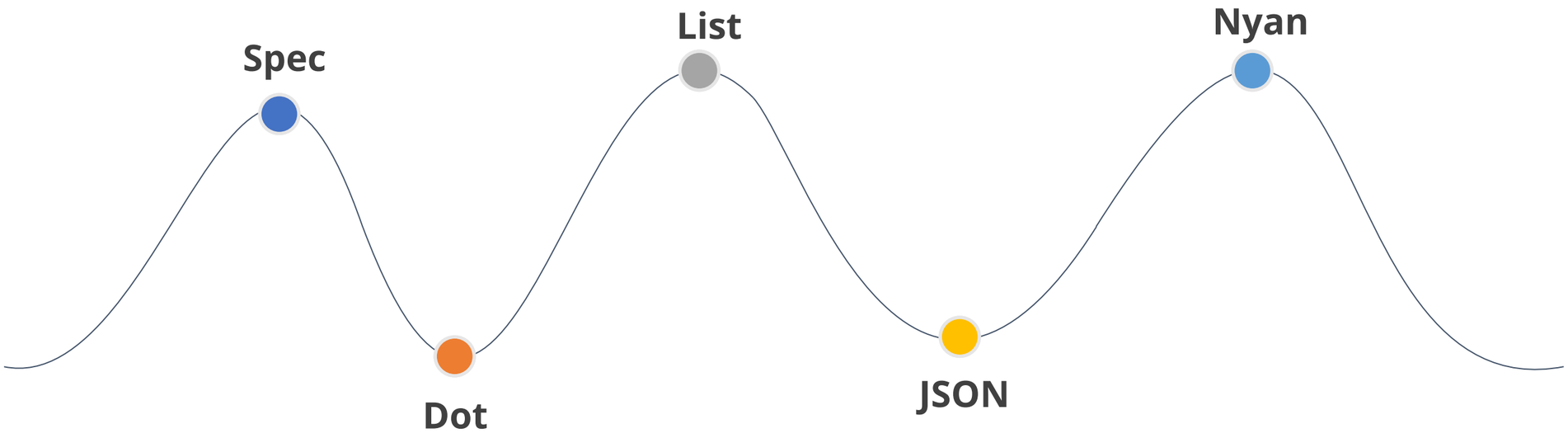


Reporters

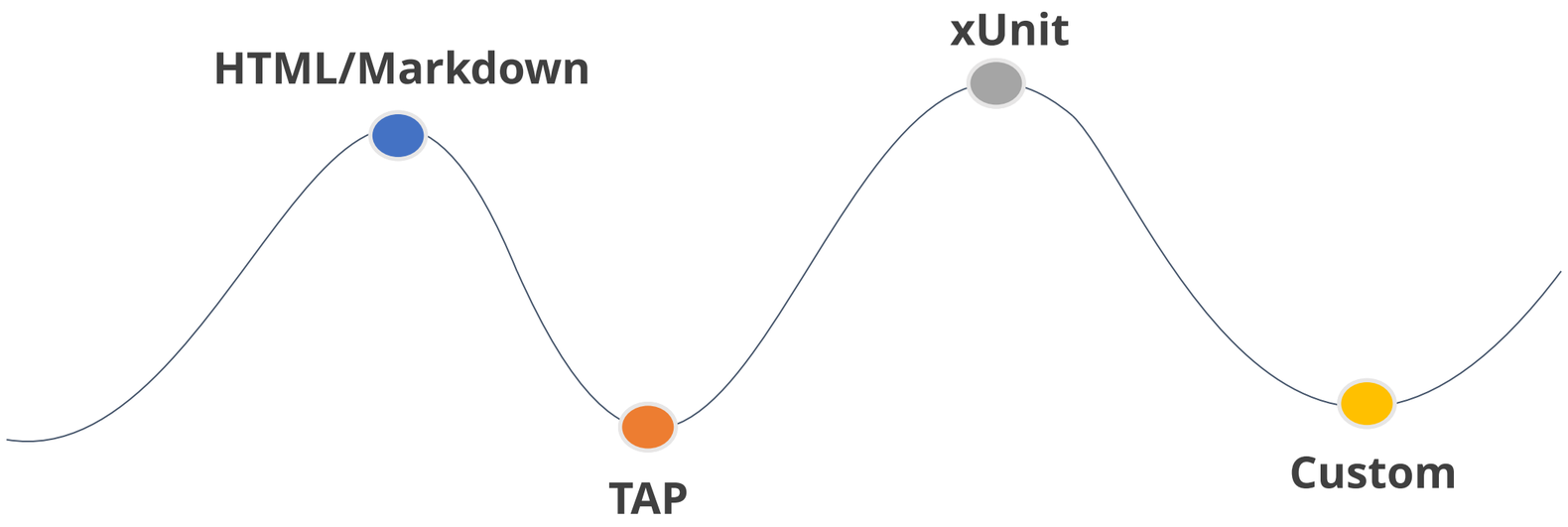
Mocha **reporters** adjust to the terminal window and disable **ANSI-escape** coloring when the **stdio** streams are not associated with a **TTY**.

Reporters

Following are some built-in reporters in Mocha:



Reporters



Reporters in Mocha

Problem Statement: **Duration: 15 min.**

You have been asked to test Node.js with Mocha module and generate a different output using reporter concept.

Assisted Practice: Guidelines

Steps to Perform:

1. Create the project structure and package.json file and install the Mocha module
2. Create JavaScript and test the files

Configuring Mocha

Node.js Native ESM Support

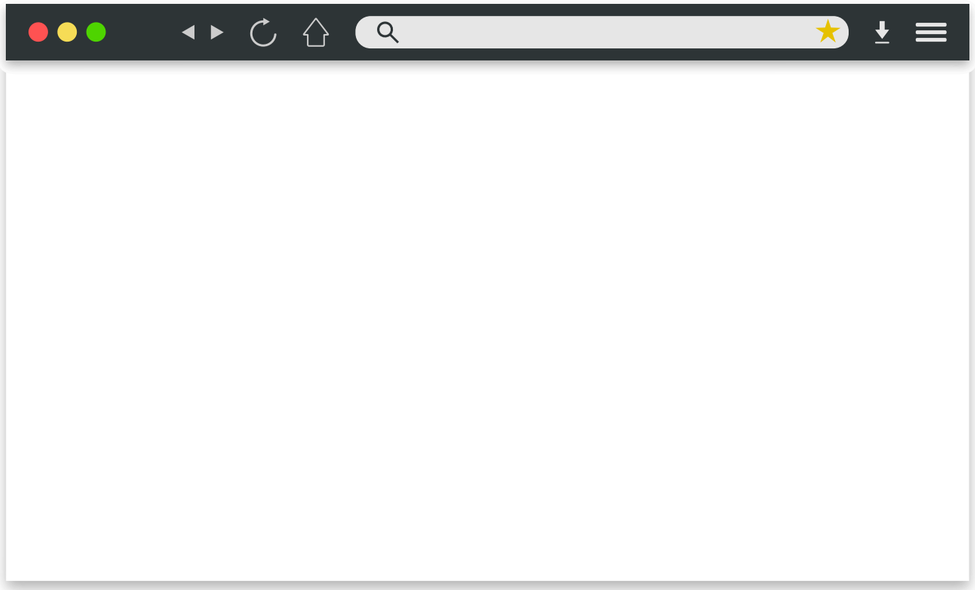
Mocha allows the creation of tests utilizing the **ES** modules in addition to the **CommonJS** module.

```
// test1.mjs
import {add} from './add.mjs';
import assert from 'assert';

it(‘adds numbers from an es module', () => {
  assert.equal(add(4, 8), 6);
});
```

Running Mocha in Browser

The Mocha library and other libraries or test files you want to run must be included in your HTML page to function in a browser.



Once the Mocha library has been added to the HTML file, make a new **test file** and add it using the script tag.

Running Mocha in Browser

The below example shows how to include Mocha and run a test file in a browser:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mocha in Browser</title>
    <!-- include Mocha library -->
    <script
src="https://unpkg.com/mocha/mocha.js"></script>
    <!-- include test file -->
    <script src="test.js"></script>
    <!-- set up Mocha -->
    <script>
      mocha.setup('bdd');
      // run tests when page is loaded
      window.addEventListener('load', function () {
        mocha.run();
      });
    </script>
  </head>
  <body>
    <div id="mocha"></div>
  </body>
</html>
```

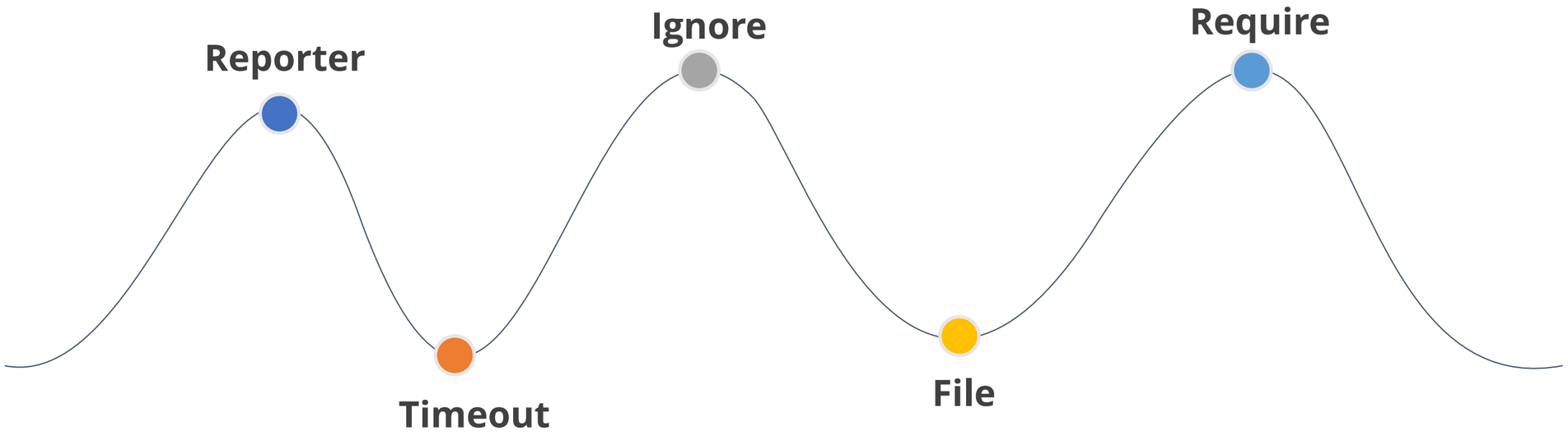
Configuration File in Mocha

Mocha configures the test runner's settings and options using a configuration file. It is usually named **mocha.opts** or **mocha.config.js** and should be placed in the project’s root directory.

It can be written in various formats, like JSON, YAML, or JavaScript.

Configuration File in Mocha

The configuration file allows for setting options such as:



Configuration File in Mocha

Here is a sample of the **mocha.config.js** file:

```
module.exports = {
  reporter: 'spec',
  timeout: 4000,
  ignore: ['node_simplimodule'],
  require: ['test/simplisetup.js']
}
```

Test Directory

A test directory in Mocha is a directory where test files are kept.

Typically, test files are kept in a separate directory under the name **test** or **tests**.

Test Directory

Follow these steps to build a test directory in Mocha:

Create a new directory in your project directory and name it **test** or **tests**

Create one or more test files inside the test directory, each containing the tests that must be executed

Add a test script that launches Mocha and specifies the location of test files in the project's **package.json** file

Run the tests by using the **npm** test command

Errors in Mocha

Some of the common error codes that may occur while using Mocha are as follows:

ReferenceError: This error occurs when you try to use a variable that has not been defined.

TypeError: This error occurs when you try to operate on a value that is not the expected type.

AssertionError: This error occurs when an assertion fails.

Errors in Mocha

TimeoutError: This error occurs when a test takes longer than the configured timeout to complete.

SyntaxError: This error occurs when there is a syntax error in your code.

UnhandledPromiseRejectionWarning: This error occurs when a promise is rejected.

Express.js App Testing

Problem Statement: **Duration: 20 min.**

You have been asked to create REST APIs using Express.js and test those REST APIs using Mocha with SuperTest

Assisted Practice: Guidelines

Steps to Perform:

1. Create the project structure and package.json file and install mocha and SuperTest party module
2. Create the REST API using Express.js
3. Test the REST APIs using Mocha with SuperTest

Running Mocha in Browser

Problem Statement:

Duration: 15 min.

You have been asked to run the Mocha testing file on the browser

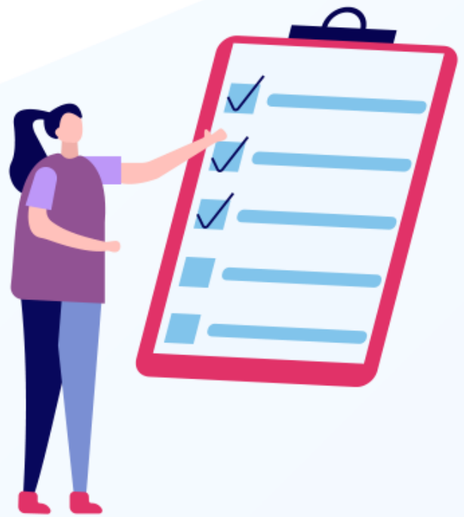
Assisted Practice: Guidelines

Steps to Perform:

1. Create the project structure in VS code and an index.html file
2. Add external third-party Mocha and Chai library

Key Takeaways

- Mocha is a JavaScript testing framework that runs both on Node.js and in the browser.
- Mocha offers strong hooks to enable adaptable setup and teardown of test fixtures.
- A run cycle is a series of activities that take place when a test is run.
- Assertion libraries are used to determine whether a set of conditions is true or false and are frequently used in unit testing.
- TimeoutError, SyntaxError, TypeError, and AssertionError are some of the common errors that occur in Mocha.



Thank You