UNIVERSITY OF PLYMOUTH

Lecture 10: Server-Side Testing

COMP3006: Full-Stack Development

David Walker

School of Engineering, Computing and Mathematics
david.walker@plymouth.ac.uk

Semester 1, 2020-21

Introduction

Today's topics

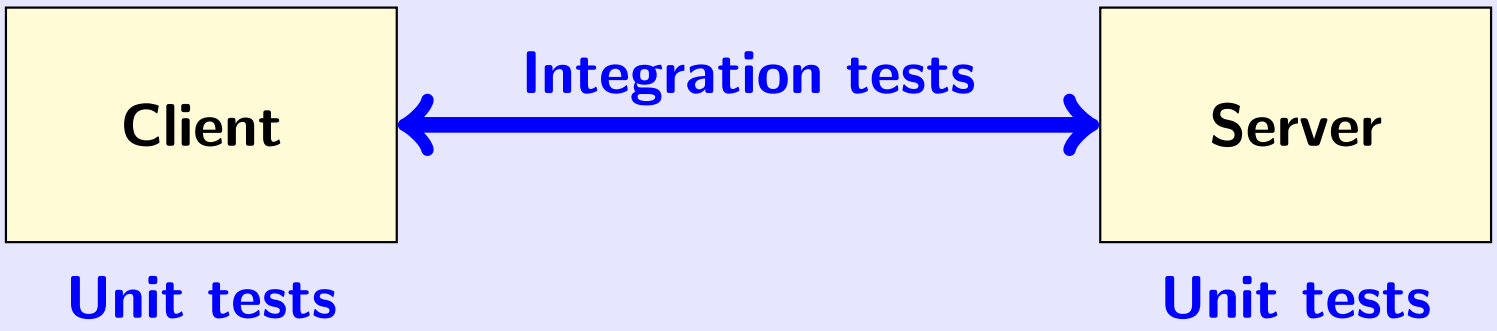
1. Server-Side Testing
2. Server-Side Unit Testing
3. Mock Objects
4. Integration Testing

Session learning outcomes

By the end of today's lecture you will:

- Implement unit tests for server-side JavaScript code
- Use mock objects to represent other parts of the system within tests
- Test the connections between components of a system using integration tests

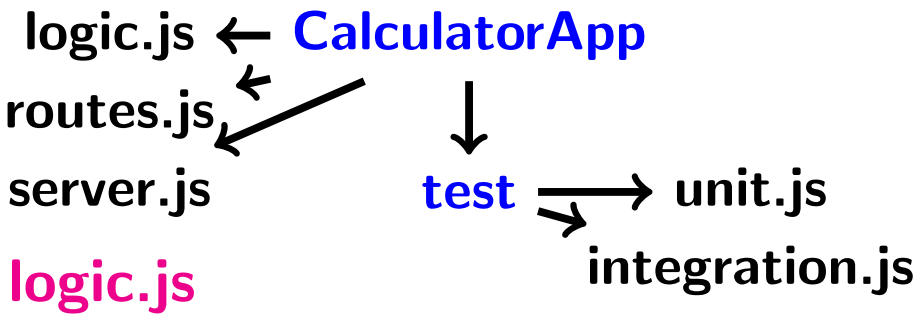
Full-Stack Testing



System tests

e.g. **behaviour-driven tests**, **UX tests**, **load testing**. . .

A sample full-stack application



logic.js

```
function square(x) {
  return x * x;
}

module.exports.square = square;
```

routes.js

```
let logic = require("./logic");

function square(req, res) {
  let num = req.params.number;
  res.send(
    logic.square(num).toString())
}

module.exports.square = square;
```

server.js

```
let express = require("express");
let routes = require("./routes");

app = express();

app.get("/square/:number", routes.square);

module.exports.app = app;
```

Unit testing

The same as unit testing on the client side

What is a test?

- A test executes a unit of code and checks whether the result of execution was as expected
  - Observed value
  - Expected value
  - Assertion (equal, OK...) and feedback message

Test business logic

- Test should be **atomic**
- Tests should not depend on the order they occur in
- Should be a lightweight test of the code
- Support regression testing

Testing the `square` function

```
let chai = require("chai");
let logic = require("../logic");

suite("Test square function", function() {

  test("Test the square function", function() {
    let result = logic.square(3);
    chai.assert.isNumber(result, "Result should be numeric");
    chai.assert.equal(result, 9, "3x3 should equal 9");

    result = logic.square(5);
    chai.assert.isNumber(result, "Result should be numeric");
    chai.assert.equal(result, 25, "5x5 should equal 25");
  });

});
```

Running the tests

Use the `mocha` application to run the tests

- The **`mocha`** application has a range of test interfaces – specify the TDD one
- Tests are stored in a directory called **`test`**
- Mocha will execute all of the tests within the directory

```
Davids-MacBook-Pro:CalculatorApp djw213$ mocha -ui tdd test/


  Test square function
    ✓ Test the square function


  1 passing (11ms)
```

Mock objects

Use mock objects when the real object

- has non-deterministic behaviour
- is difficult to set up
- has behaviour that is hard to trigger (e.g., network error)
- is slow
- has (or is) a user interface
- does not yet exist

Implementing mock objects

1. Use an interface to describe the object
2. Implement the interface for the production code
3. Implement the interface in a mock object for the unit test

Sinon example – timers

Control the system time for time/date-specific tests

1. Initialise the date/time as required
2. Run the test
3. Restore the actual date/time

```
let chai = require("chai");
let sinon = require("sinon");
let logic = require("../logic");

suite("Test message generator", function() {

    test("Check morning message correct", function() {
        let date = new Date(2020, 11, 1, 10, 0, 0, 0);
        let clock = sinon.useFakeTimers(date);
        let msg = logic.greetingMessage();
        chai.assert.equal("Good morning", msg, "Wrong 10am msg");
        clock.restore();
    });
```

Sinon example – timers

```
test("Check afternoon message correct", function() {
    let date = new Date(2020, 11, 1, 14, 0, 0, 0);
    let clock = sinon.useFakeTimers(date);
    let msg = logic.greetingMessage();
    chai.assert.equal("Good afternoon", msg, "Wrong 2pm msg")
    clock.restore();
})

test("Check evening message correct", function() {
    let date = new Date(2020, 11, 1, 21, 0, 0, 0);
    let clock = sinon.useFakeTimers(date);
    let msg = logic.greetingMessage();
    chai.assert.equal("Good evening", msg, "Wrong 9pm msg");
    clock.restore();
});
```

```
Davids-MacBook-Pro:HelloWorld djw213$ mocha -ui tdd test


  Test message generator
    ✓ Check morning message correct
    ✓ Check afternoon message correct
    ✓ Check evening message correct
```

Sinon example – spies

Inspect the calling of functions

- e.g. has a function been called

```
let routes = require("../routes"); // (and chai/sinon)

suite("Test Express router", function() {

    test("GET greetingRoute", function() {
        let request = {};
        let response = {};
        response.send = sinon.spy();

        routes.greetingRoute(request, response);
        chai.assert.isTrue(response.send.calledOnce);
    });

    test("GET
```

```
Davids-MacBook-Pro:HelloWorld djw213$ mocha -ui tdd test


  Test Express router
    ✓ GET greetingRoute
```

Possible integration failures

Possible integration failures

- Incorrect method invocation
- Methods invoked correctly but in the wrong sequence
- Timing failures – **race condition**
- Throughput/capacity problems

To test these, your integration tests will...

- Read/write to a database
- Call a web service
- Interact with files/directories on the hard drive

Testing the `square` route

```
let chai = require("chai");
let chaiHttp = require("chai-http");
let server = require("../server");

chai.use(chaiHttp);

suite("Test routes", function() {

  test("Test GET /square", function() {
    let app = server.app;

    chai.request(app).get("/square/3")
          .end(function(error, response) {
            chai.assert.equal(response.status, 200, "Wrong status
            chai.assert.equal(response.text, "9", "Wrong text");
          });
  });

});
```

```
Davids-MacBook-Pro:CalculatorApp djw213$ mocha -ui tdd test/


  Test routes
    ✓ Test GET /square

  Test square function
    ✓ Test the square function


  2 passing (77ms)
```

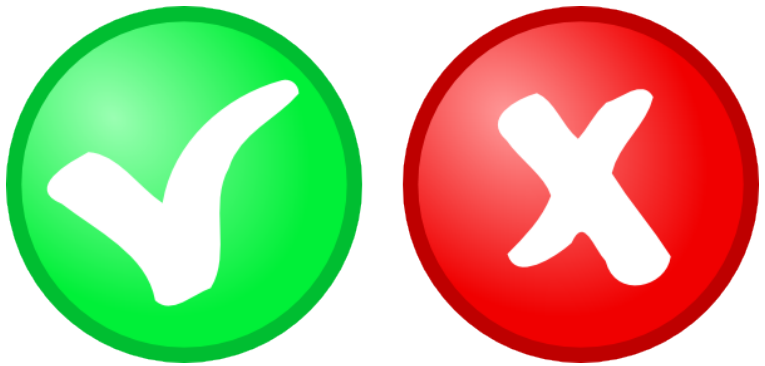
Integration testing a database

Set up a test database to run integration tests against it

1. **Before any of the tests run** set up a test instance of the database
2. **Before each test** set up test data so that tests are executed against a standard setup
3. Execute the test
4. **After each test** clean the data from the database
5. **At the end of the suite** delete the test database

Best practice

1. Integrate early, integrate often...
2. Don't test business logic with integration tests
3. Keep test suites separate
4. Log often
5. Follow a test plan
6. Automate whatever you can



Summary

Unit testing

- Test business logic
- Use assertions to confirm that the program state is as it should be
- Use mock objects to replicate parts of the system where necessary

Integration testing

- Ensure that the components of the system work together

Next

- Continuous integration & deployment