ELECTRICAL AND INFORMATION ENGINEERING

University of the Witwatersrand, Johannesburg

Software Development III

Laboratory 4 — Testing, Deployment and Local Storage

In this lab, we will implement testing for the class list app from Lab 3 and deploy it to Azure using Travis which is a Continuous Integration (CI) service.

1 Testing

It is important to test our code regularly in order to ensure that we don't break or regress features as we add new code. There are many testing frameworks for JavaScript and Node. We will be using the Jest framework for testing.

1.1 An Introduction to Jest

Jest is a powerful and popular testing framework which has been developed, and is maintained, by Facebook. Jest is easy to use and runs on Node. Besides offering the ability to write and run tests, it is also capable of generating code coverage reports and mocking objects.

In C++, using doctest, you became familiar with tests in the form of:

```
TEST_CASE("Adding to a vector increases its size") {
        std::vector<int> v(5); // creates a vector with a size of 5

        v.push_back(1);

        CHECK(v.size() == 6); // assertion
}
```

Jest provides a similar expression syntax.

```
function sum (a, b) {
  return a + b
}

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3) // assertion
})
```

In Jest, assertions are made through the use of `expect`, and `toBe` tests for exact equality. `toBe` is known at a "matcher".

1.2 Jest Tutorial

Exercise 1

There is a very good beginner tutorial on Jest, by Valentino Gagliardi. Work through this tutorial by setting up Jest as per the instructions given, and then run the code samples. The tutorial also demonstrates Jest's code coverage capabilities. You can ignore the small section on React.

Before starting the tutorial it is worth noting the following:

1. Jest is installed as a development dependency so that it will only be installed in the development environment and not in production:

   ```
   npm i jest --save-dev
   ```

2. Jest expects to find tests either in a directory called `__tests__` *or* in files which have `.test.` or `.spec.` in the filename. Often tests are put in a `test` directory and follow the convention of having `.test.` in the name of the test file.

3. The Jest testing framework makes use of a number of global functions, including `test()` and `describe()`. If you are using StandardJS for linting (as advised) then these globals produce linting warnings. In order to remove these warnings you need to inform the linter that you are using Jest. To do this, put the following line at the top of each test file:

   ```
   /* eslint-env jest */
   ```

1.3 Jest Extension for Visual Studio Code

The Jest extension for VS Code makes Jest very easy to use. The test files, and the code being tested, are constantly watched and the tests are run automatically whenever there are saved changes. Passing and failing tests are highlighted inline and failing tests can be debugged using the built-in debugger.

Code coverage reports can also be generated by turning on the code coverage overlay. Note, if you are running your tests via this extension (instead of npm), then code coverage reports will not be generated unless this overlay is turned on, irrespective of your settings in `package.json`.

Try the next exercise using the Jest extension.

1.4 Testing the classList Module

Okay, back to the example from the previous lab. It's good practice to abstract your data storage from the rest of your code with an interface so that your business logic does not become dependant on your specific database or version. We are going to do this with our class list app.

Exercise 2

At the moment, the class list is just stored in an array at the top of the router file. We are going to create a module that presents an interface for adding, deleting, editing and viewing students from the class list.

Create a file called `classList.js` with the following in it:

```
// Private
let list = []

// Public
module.exports = {
  add: function (student) {
    list.push(student)
  },
  edit: function (student, index) {
    list[index] = student
  },
  get: function (index) {
    return list[index]
  },
  delete: function (index) {
    list.splice(index, 1) // remove one element starting from index
  }
}
```

Now edit `classRoutes.js` to require and use your new module instead of the array. Add functions to `classList.js` if necessary.

Using the documentation for Jest, implement some real tests for your newly-created module. The Jest cheatsheet summarises key functionality.

In order to obtain the bonus mark for this course, you need to submit your solution on GitHub for this exercise. You must submit only the following four files:

1. `classList.js`
2. `classRoutes.js`
3. Your file containing the tests
4. A README file containing your name

Make your submission by typing the following in the VS Code terminal:

1. Create a local repo using `git init` in the directory containing your work.
2. Add and commit the README file to *master* using `git add README`, followed by `git commit -m "Initial commit"`
3. Create a *solutions* branch and switch to it: `git checkout -b solutions`
4. Add `classList.js`, `classRoutes.js` and your test file to staging.
5. Commit these to the *solutions* branch: `git commit -m "Exercise 2"`
6. Link to your GitHub remote repo:
   `git remote add origin <your Lab-4-Exercise-2 repo's SSH URL>`

7. Push all your branches to GitHub: `git push --all origin -u`
8. Lastly, make a pull request on GitHub from *solutions* into *master*; the pull request can be named "Exercise 2".

2 Deployment

Manually watching your git repository for changes, running tests and then deploying to your production server is time-consuming. Instead most companies opt to have an automated deployment pipeline.

We are going to set up our own automated deployment platform using Travis CI and Azure.

Firstly, ensure that you have a student account with both Travis CI and Azure. The best method for getting these is using the GitHub Student Pack (https://education.github.com/pack).

2.1 Travis CI

Navigate to https://travis-ci.com and register using your GitHub account. You should be redirected to Github.com where you should follow the prompts.

Now add a file called `.travis.yml` with the following to the root of your git repository where you replace version with the version of node you are running :

```
language: node_js
node_js:
  - "version"
```

The version can be found by running `node -v`. Commit and push the file to your repository.

If you then open Travis CI, you should see your project being built:

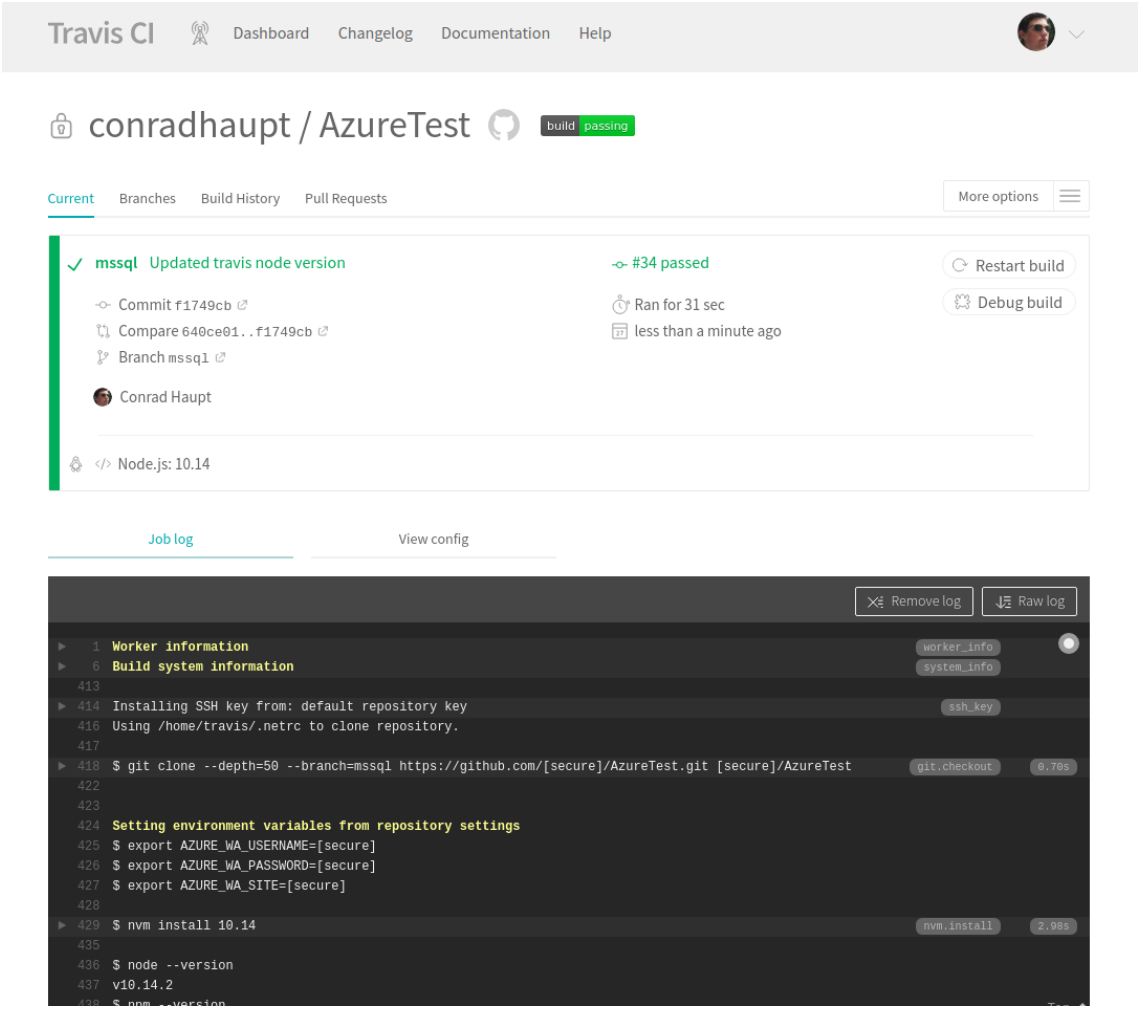

Figure 1: A successful build in Travis CI

2.2 Azure Deployment from Travis CI

Right now your Azure setup will deploy any new commit on the master branch while Travis CI checks all tests independently. What you really want is for deployments to occur IF AND ONLY IF the most recent commit has passed all tests successfully. To do this we need to move the automatic deployment functionality from Azure to Travis CI.

Fortunately Travis CI has a built in provider for Azure deployment. To configure and enable it, first you must remove the old GitHub Deployment Option on your Azure instance. Navigate to your website's configuration page and then to the `Deployment Centre` sub-menu. `Disconnect` your current config and click `Setup`. In the new window select `Local Git Repository` as the new `source` then click `OK` until the deployment change is complete. Once you're back at the main portal window for your website, select `Deployment Credentials`. Select `User Credentials` at the top of the sidebar and fill in `UNIQUE` details for the username and password.

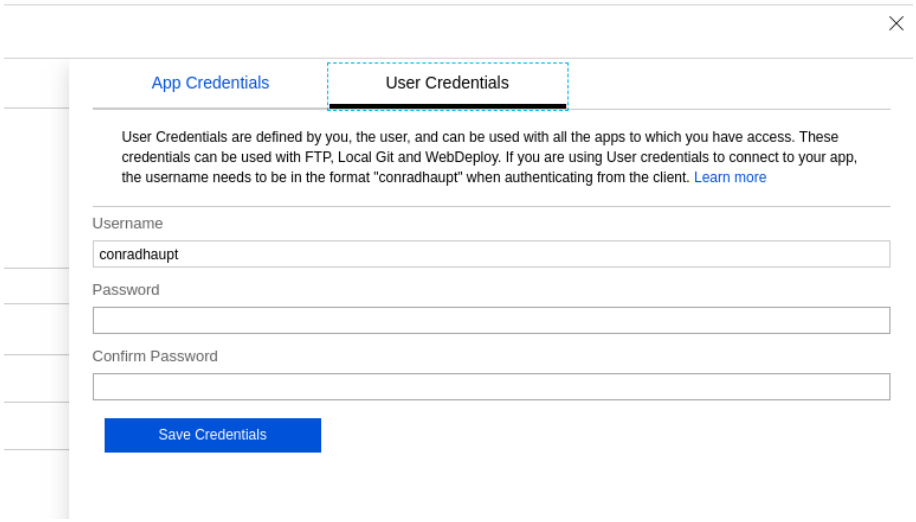

Figure 2: Example App Credentials for Azure Deployment

Storing authentication information in Git and GitHub is very insecure. Travis CI allows us to store environment variables in the build information to ensure we don't have any security risks with our code. Navigate to your Travis CI page for your GitHub project and click on `More Options` and then `Settings`. In the window that appears, scroll down to the section titled `Environment Variables`. Add the environment variables listed in the table below to Travis CI. Ensure that the variable names are in all-caps.

Table 1: Travis CI Environment Variables necessary for Azure Deployment

Variable Name	Variable Value
`AZURE_WA_USERNAME`	The username you setup in the deployment credentials page
`AZURE_WA_PASSWORD`	The password you setup in the deployment credentials page
`AZURE_WA_SITE`	The name of your Azure project

After adding those three environment variables, your settings page should look like the figure below.

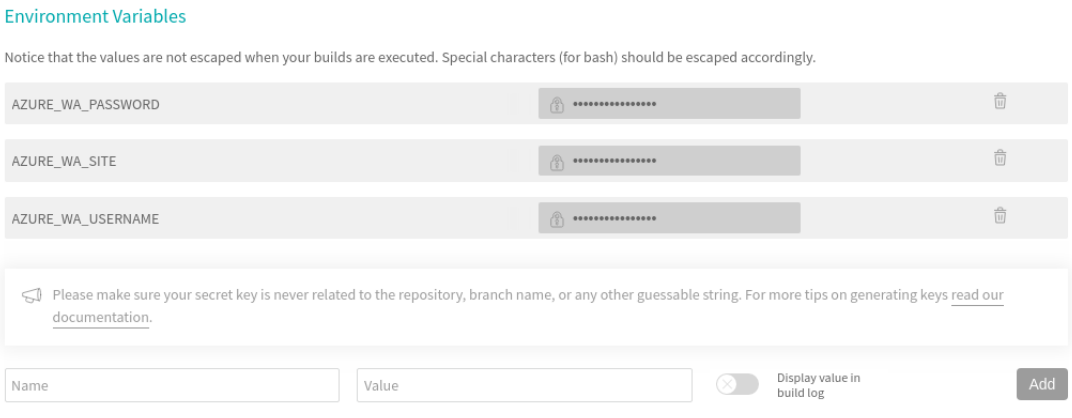

Figure 3: Travis CI Azure Deployment Environment Variables settings section

The final change to make is to add the following code to the end of your `.travis.yml`

file in your repo, commit, and push to GitHub.

```
deploy:
    provider: azure_web_apps
```

After Travis CI has run the tests for your new commit, you should see something similar to the following output in the `Job Log`. If you navigate to your Azure Instance's URL you will be viewing an automatically deployed version of your repository that is controlled by Travis CI.

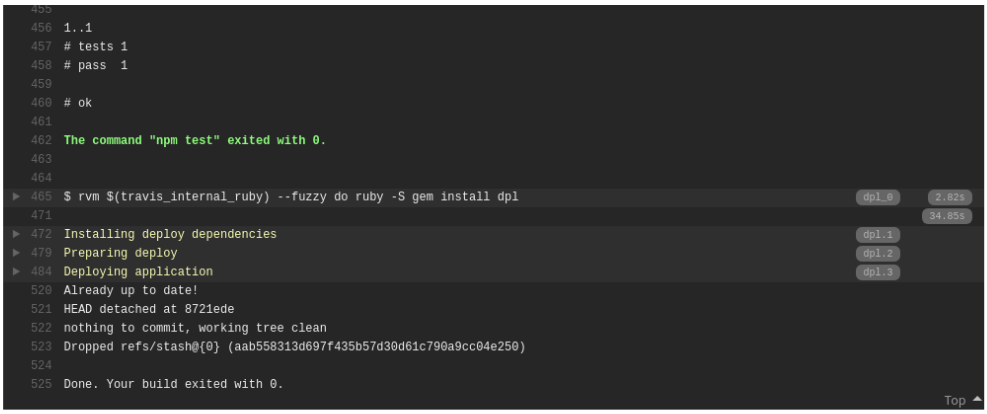

Figure 4: Travis CI Job Output for Azure Deployment

3 Browser LocalStorage

With new developments in browsers and the internet at large, websites can start to take advantage of device capabilities that weren't available 5 years ago. An example is `LocalStorage`. Browsers can now store information and data that is needed by the website without having to deal with cookies or server-side storage.

Lets say that your website needs to store a variable that describes the theme of your website for each user. You have three options:

- Require the user to change the theme every time they navigate to your website.
- Store a session cookie on the user's browser and the matching theming variable on a server-side database. The website would then be themed by the server using the cookie value.
- Store the theming variable in LocalStorage on the user's browser, requiring only JavaScript on the client-side.

The third option is the most modern and advised for new websites. Obviously, older browser version may not support this but most do. To check if a client has LocalStorage capabilities, you can run the code below.

```
if(typeof(Storage) !== "undefined"){
// LocalStorage is available
} else {
// LocalStorage is NOT available
}
```

If the client's browser supports it, then you can do fancy things like storing key-value pairs.

```
// The user would like the website to use the dark theme
window.localStorage.setItem("theme", "dark")

console.log(window.localStorage.getItem("theme"))
```

You can also use named member variables of localStorage.

```
window.localStorage.theme = "dark"

console.log(window.localStorage.theme)
```

You can delete entries in LocalStorage using `window.localStorage.removeItem("theme")`. The life of the variables stored in LocalStorage are dependent on the browser but they will last for longer than the tab is open.

If you want the variable to be deleted when the tab is closed, replace `localStorage` with `sessionStorage` in the above examples. This way, the variables will be deleted once the browser stops navigating to your website. You can verify if a variable exists by writing `if(window.localStorage.<variable name>)`. The condition will return **`true`** if the variable exists and false if it doesn't. It should be noted that localStorage is a client-side only feature so it will not work if the code is run in scripts that are server side only.

Exercise 3

Add code to your app to store the class list array on the user's computer. If done correctly, the class list should persist, with edits and all, through stopping and starting the app.

This is not something you would typically store on a user's computer. This is for the purpose of understanding the functionality of LocalStorage Hint: Use your `/cdn/class/index.js` to access the LocalStorage and use your already existing add and edit functionality to save it to local storage and send it to the server.