

CAB432 Cloud Computing

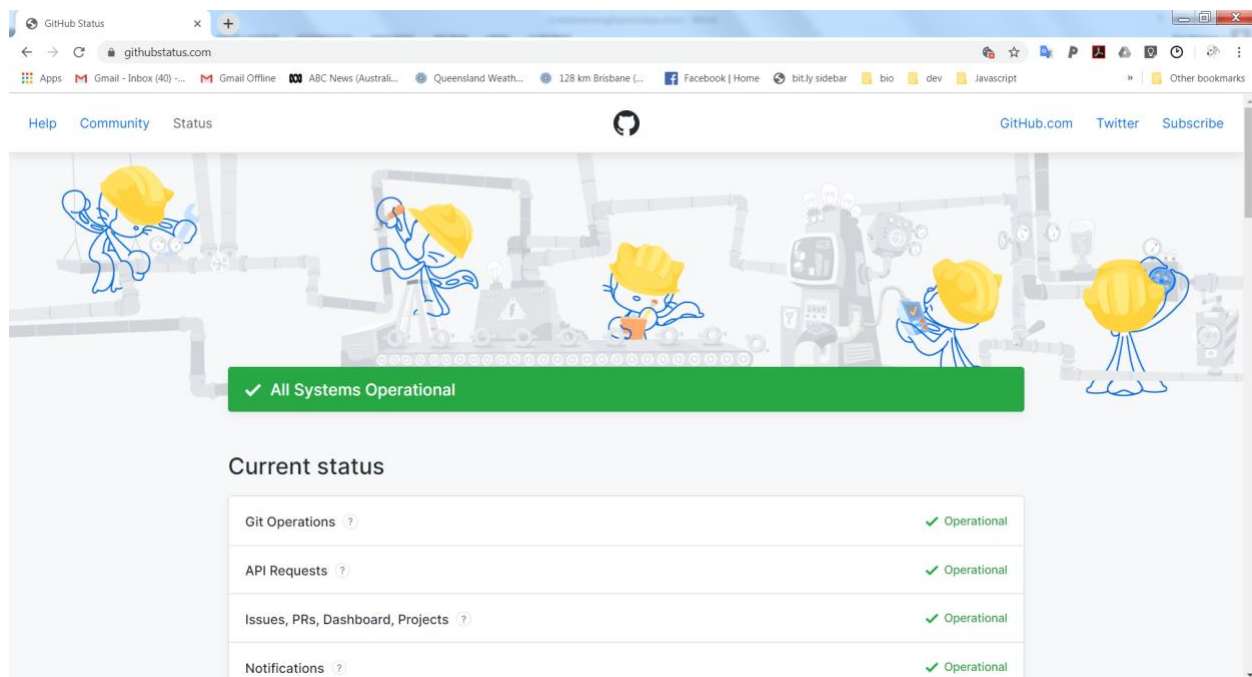
Containerising an Express Application

Overview

At this stage you've had some experience with Express and Docker separately. It's now time to put all of this together. Ultimately you can make a choice about how closely you wish to follow the instructions below. The goal is to help you further along the way to the Docker requirements for Assignment 1 – the app used here is to make that a bit easier, but if you are confident with the material, you can move more rapidly and apply the Docker instructions below to your assignment.

For now, however, we will begin with a small Express Application. We will then containerize it with Docker. We will develop a simple, one page application to retrieve the current status of GitHub and display it on a page for your users. There is nothing crucial about this app – it is an example.

GitHub provides basic status information here: <https://www.githubstatus.com/>



There is also a good basic API which is now in version 2 and is substantially different from version 1. You will find the docs here: <https://www.githubstatus.com/api> and we will focus on the status endpoint, which is documented here: <https://www.githubstatus.com/api#status>. As you will see, the endpoint returns JSON (<https://kctbh9vrtwdw.statuspage.io/api/v2/status.json>) and an example is given on the next page.

We will need to work with this JSON to grab the elements of the status object. We will display the results based on whether the GitHub services are working well or not. The `indicator` field tells us the severity of the problem. If this is set to the value `'none'`, then we don't have a problem, and we can tell our users that everything is fine. We will use the `description` field supplied by GitHub. If everything is fine, then we will see the message "All Systems Operational".

An example JSON object retrieved from the API is shown below:

```
{
  "page": {
    "id": "kctbh9vrtwd",
    "name": "GitHub",
    "url": "https://www.githubstatus.com",
    "time_zone": "Etc/UTC",
    "updated_at": "2019-08-28T08:17:45.586Z",
    "status": {
      "indicator": "none",
      "description": "All Systems Operational"
    }
  }
}
```

Express Generator

Express Generator is a command line tool that is used to generate a solid boilerplate for an express application. You have seen it used in the lectures and you can view the [documentation](#) on the `expressjs` website, under the *getting started* section. For this application we will pass through the optional `pug` flag to ensure that we get our preferred view engine.

To get started, install express generator globally by running `npm install express-generator -g` followed by `express --pug github-status`. You can now navigate into your newly created `github-status` folder and look at the generated files and application structure. Take particular note of the folder structure, the start script in `package.json` and the contents of `app.js`.

Building the Application

Before doing anything further, run `npm install` in your project directory to install all the modules your project is going to use. We will again use `axios`, and the code will be similar to that which you have seen in the earlier server-side Flickr prac. After jumping into the `index.js` file in the `routes` folder, we will add the logic needed to retrieve the current GitHub status. We will then render a template, with the output depending on the result.

The Route

Overall, the route logic is going to be reasonably straightforward. We're going to make a single request to the GitHub Status API and pass the result through to a template to be rendered. Edit your `routes/index.js` file to match the code fragment shown on the next page:

```

const express = require('express');
const axios = require('axios');
const router = express.Router();

/* GET home page. */
router.get('/', (req, res) => {

  const url = 'https://kctbh9vrtwd.statuspage.io/api/v2/status.json';

  axios.get(url)
    .then( (response) => {
      const rsp = response.data;
      res.render('index',{ rsp });
    })
    .catch((error) => {
      res.render('error',{ error });
    })
});

module.exports = router;

```

The necessary information from the response will be made available to the index template.

The View

Rendering a view is going to be done with two components: *layout.pug* and *index.pug*. Both are in the *views* folder. Note that the pug template makes use of [inheritance](#). We're rendering index, but index extends layout, so layout is rendered first, followed by the index content block as indicated.

Inheritance can be used to create extremely complicated and dynamic views, but in this example, we will just use layout to import our CSS library and semantic-ui, and to create a simple navigation menu. The index template will consume our data and display the relevant status result.

Before we get started on the actual templates, make a small edit to the *style.css* file located in: */public/stylesheets*. Note the change to line 2 and the addition of the logo class selector below.

```

body {
  padding: 75px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}

a {
  color: #00B7FF;
}

```

```
.logo {
  width: 100px;
}
```

We now move on to the layout and index templates. If this is the first time you've seen pug you can read all about it on their [website](#), but it is a convenient way of generating HTML without some of the syntactic complications. We first step through the *layout.pug* file:

[illegible]

The first change is to import the semantic-ui css framework. From lines 8 to 12 we have the navigation menu and line 13 pulls in the content block from the index template which you will see shortly. Note that Pug is sensitive to indentation; if the next line is indented further than the current one, the new element will be nested inside the one above it.

Finally, we look at the classes we are using to construct the menu. [Semantic UI](#) is designed to be readily comprehensible to a human reader and is a great starting point for your first CSS framework. Here we will use some of its simple elements without really explaining them. The main site has accessible documentation and tutorials that you can work with. But here we will keep things simple, and work toward our docker container.

The last step in our new express application is the `index.pug` template. Replicate the edits to the file below. Make note of the if statement in the template, along with the corresponding else below.

Here we work directly with the fields provided by the API. We will place the whole file at the top of the next page so that you can view it more clearly.

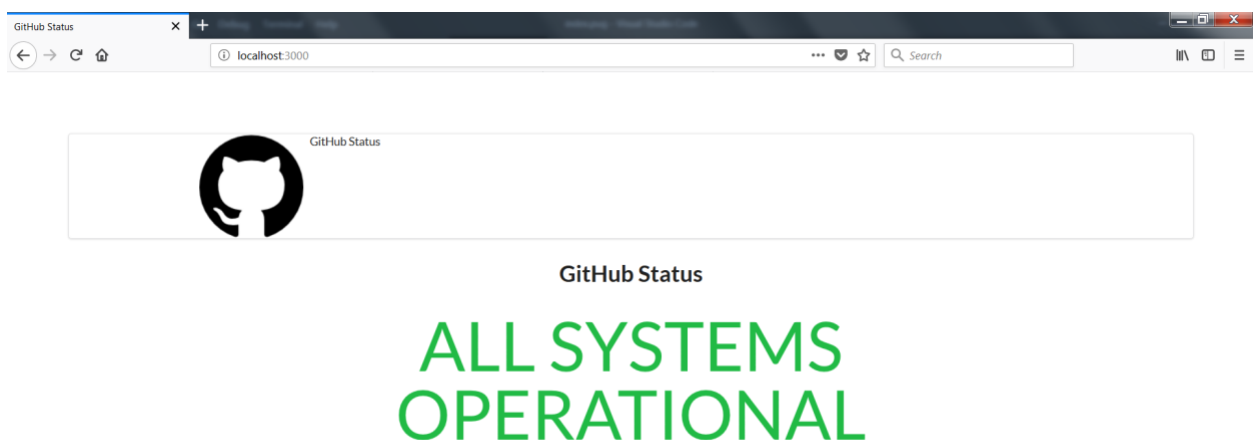
```

extends layout

block content
  .ui.container
    .ui.basic.center.aligned.segment
      h1.ui.header GitHub Status
      if rsp.status.indicator === "none"
        .ui.green.huge.statistic
          .value #{rsp.status.description}
      else
        .ui.red.huge.statistic
          .value Severity is #{rsp.status.indicator}
        .ui.red.huge.statistic
          .value #{rsp.status.description}

```

If you haven't already, you can start the express application by running `npm start` and look at `localhost:3000`. You will generally see something like this:

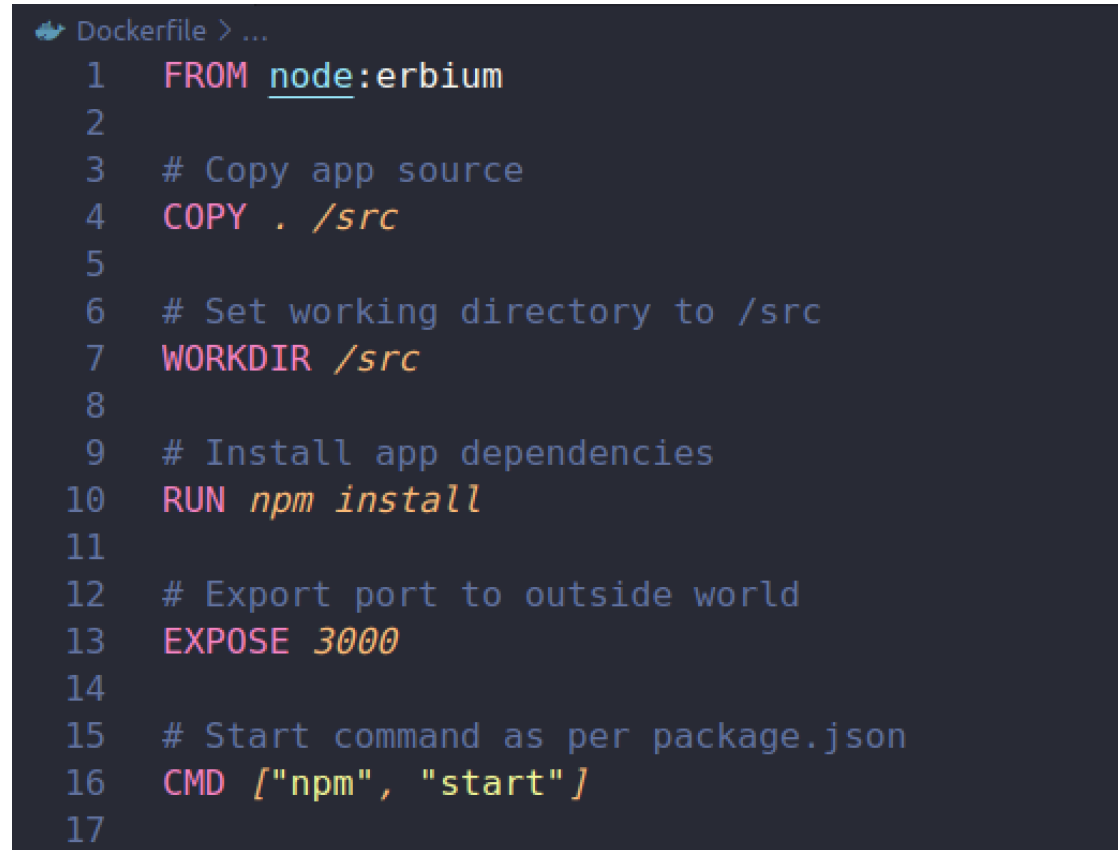


In the remainder of this document, we will show you how to containerise the application. We will begin with the traditional approach, following on from the lectures and the prac from Week 2 and Week 3. Subsequently, you will see a short section on Docker compose for those who are interested.

In the example below, the image is created on top of the standard node images for Docker, though you may equally work from the ground up based on an Ubuntu Linux 18.04 LTS. The only difference is the additional command to install node and npm.

Traditional Docker Image

Containerizing an express application with Docker is a very straightforward process. Create the *Dockerfile* as shown below and take note of the comments which explain the purpose of each of the commands.

A screenshot of a code editor showing a Dockerfile. The editor has a dark background with light blue and yellow text. The Dockerfile contains 17 lines of code, each preceded by a line number from 1 to 17. The code is as follows:

```
1 FROM node:erbium
2
3 # Copy app source
4 COPY . /src
5
6 # Set working directory to /src
7 WORKDIR /src
8
9 # Install app dependencies
10 RUN npm install
11
12 # Export port to outside world
13 EXPOSE 3000
14
15 # Start command as per package.json
16 CMD ["npm", "start"]
17
```

Now that we have created our *Dockerfile* we can build and tag the image. This is very similar to what we have seen in earlier exercises. Run the command below to build the image and apply the tag *github-status*. Note the '.' at the end of the command, indicating the current directory:

```
$ docker build -t github-status .
```

With the image built, the final step is to launch the container. Note that we have exposed port 3000 from the container but we still need to bind that port to the host machine. In this case we will use port 80. Run the command below to launch your newly built image.

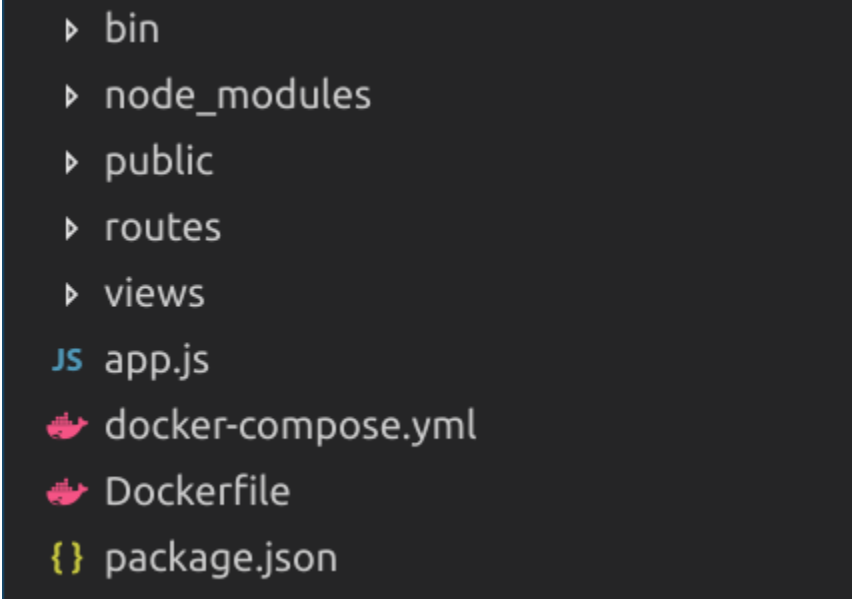
```
$ docker run -p 80:3000 github-status
```

Open your browser of choice and navigate to *localhost* to view your freshly containerised express application. In the section below we will outline an alternative approach which some of you may find more convenient, and likely more extensible than the usual approach discussed above.

Appendix: Docker Compose

[Docker Compose](#) allows an alternative approach to creating docker containers from the one shown in earlier lectures and pracs. Docker Compose is especially convenient when applications extend beyond a single container or require that we deploy multiple instances of the same container behind another load-balancing container. Compose allows you to define each service in your application stack, and to automatically build and connect them for you.

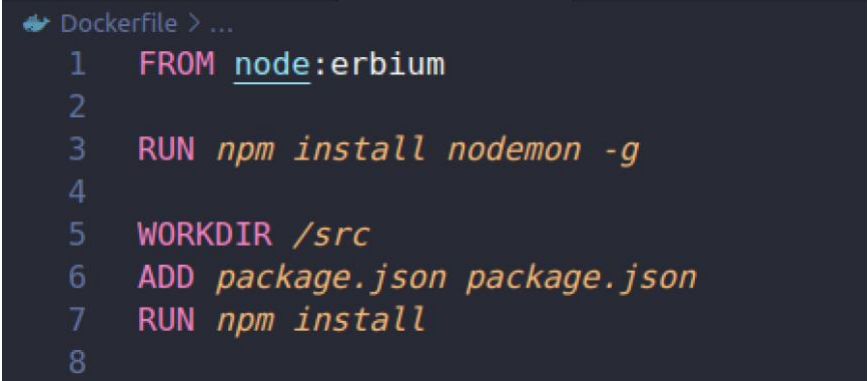
Assuming you already have Docker installed, you can follow the simple [instructions](#) on the compose install guide to get started. Here, we create two new files, *Dockerfile* and *docker-compose.yml*. Your file structure should now be as shown:



```

  ▶ bin
  ▶ node_modules
  ▶ public
  ▶ routes
  ▶ views
  JS app.js
  🚢 docker-compose.yml
  🚢 Dockerfile
  {} package.json
```

You are of course already familiar with docker and using a Dockerfile, but this time things are going to look a little bit different because we're going to get Docker Compose do some of the heavy lifting for us. Specifically, we're not going to expose any ports, copy over our application, or start express. Our *Dockerfile* is going to look something like the image below.



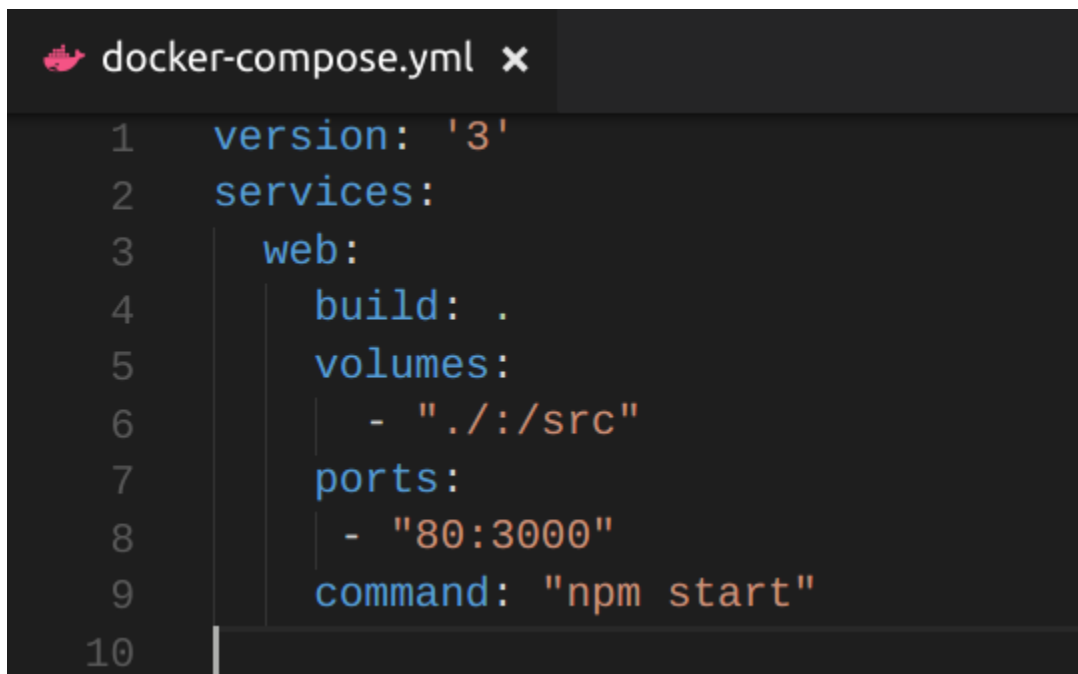
```

🚢 Dockerfile > ...
1  FROM node:erbium
2
3  RUN npm install nodemon -g
4
5  WORKDIR /src
6  ADD package.json package.json
7  RUN npm install
8
```

The package [nodemon](#) allows us to run the node application conveniently. There is one final change to be made to the package.json to allow its use. You should make a change to the start script in the package.json file to allow its use.

```
5     "scripts": {  
6       "start": "nodemon -L bin/www"  
7     },
```

The last step in this process is the new part, `docker-compose.yml`. YAML may be new to some of you, but it's very similar in principle to JSON, and you should be thoroughly familiar with JSON by now. YAML is a flexible, human readable file format that is ideal for storing object trees.



```
1  version: '3'  
2  services:  
3    web:  
4      build: .  
5      volumes:  
6        - './:/src'  
7      ports:  
8        - '80:3000'  
9      command: "npm start"  
10
```

A few things to note; Much like PUG from before, YAML is whitespace sensitive in that each nested key is a child of the parent above. In this example we're telling Docker Compose that we'd like to use *version 3* of their tool to create a single service called *web*. Things should start to look a little more familiar below that. For the web service it's going to build the *Dockerfile* in the current directory, mount our source code as a volume, expose the container port 3000 to the host port 80 (as we did in the first docker exercises) and run `npm start` to invoke the application.

Finally, we run `docker-compose up` to start your newly containerised application.


```
→ githubstatus sudo docker-compose up
Starting githubstatus_web_1 ...
Starting githubstatus_web_1 ... done
Attaching to githubstatus_web_1
web_1 | npm info it worked if it ends with ok
web_1 | npm info using npm@3.10.10
web_1 | npm info using node@v6.11.3
web_1 | npm info lifecycle githubstatus@0.0.0~prestart: githubstatus@0.0.0
web_1 | npm info lifecycle githubstatus@0.0.0~start: githubstatus@0.0.0
web_1 |
web_1 | > githubstatus@0.0.0 start /src
web_1 | > nodemon -L bin/www
web_1 |
web_1 | [nodemon] 1.12.0
web_1 | [nodemon] to restart at any time, enter `rs`
web_1 | [nodemon] watching: *.*
web_1 | [nodemon] starting `node bin/www`
```

Note that the combination of using nodemon and mounting our code as a volume rather than copying it in directly means we can now also develop in any environment with our code running directly inside the docker container that it's going to be deployed in.