SPA Development & Deployment

Using React + NodeJS + Cassandra + Docker + k8s

https://github.com/dagobah19/ReactSPA-API-Cassandra-k8s

David Orlowski & Chris Sporski

M&T Tech

Goals

Understanding SPA, benefits & drawbacks

Building the application foundation

Designing the API

Connecting it together

Deploying the SPA in containers

What is SPA?

Single Page Application

Becoming more common

Usually feature a very clean UI, smooth transition between “pages”, very fast

Examples include Facebook, Gmail

Drawbacks? A more complicated code base, not SEO friendly (This is due to how pages are usually rendered by javascript on the client side)

SPA is well suited for apps that don’t have to be indexed by a search engine

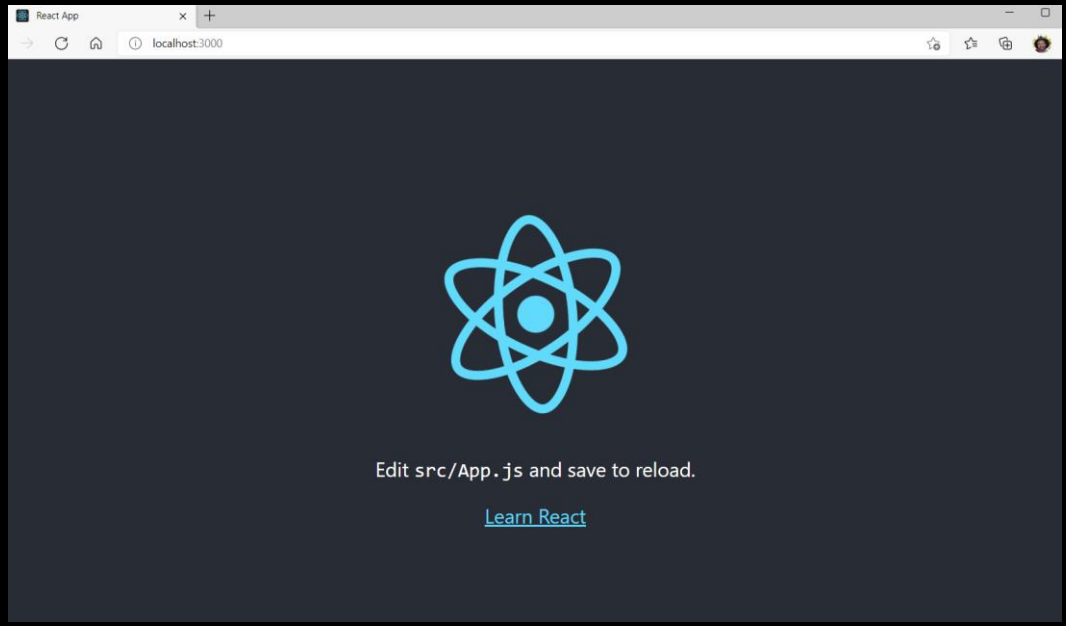
Let's build a SPA...

- We are going to use React (ReactJS)
  - Developed by Facebook
  - Increasingly popular
  - Features fast rendering and a virtual DOM
  - Leverages javascript and JSX
- There are other frameworks out there, such as Angular
- We are going to build a simple dashboard for IoT sensor data

What you'll want

- NodeJS
  - Will be used to build our API and SPA
- Docker CLI / Docker Desktop with Kubernetes enabled (k8s)
  - Build & test our completed application
- Code editor of your choice
  - VS Code, Sublime, etc

Creating the SPA using React



Create application template:
npx create-react-app spa-demo

NPX will execute a package and download its dependencies, whereas NPM is a package manager and will only download

NPX is bundled with NPM v 5.2+

Test the install
npm start – reads the package.json and starts the development server

Creating the dashboard

- In your application src directory, create two new folders:
  - Layouts
  - Components
- In layouts, create a new file called “main.js” with this code:

spa-demo > src > layouts > JS main.js > default

```
1  function Main() {
2      return (
3          <div>
4              <div id="dataDisplay">This is main content of the dashboard.</div>
5          </div>
6      )
7  }
8  export default Main;
```

A React function can ONLY return a single root element – but you CAN next within that element!

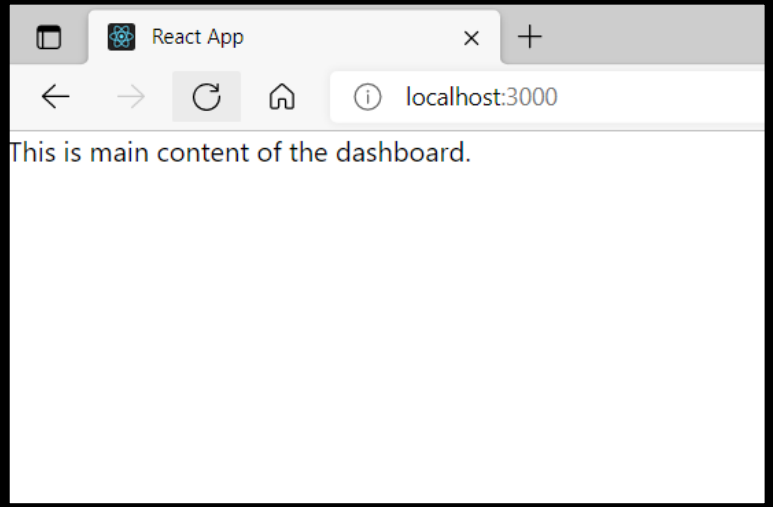
Adding the main layout to our app

- Add the import the layout in app.js
- Replace the return in App function to return the <Main></Main> layout

main.js U | App.js M

spa-demo > src > App.js > ...

```
1
2   import './App.css';
3   import Main from './layouts/main';
4
5   function App() {
6     return (
7       <Main></Main>
8     );
9   }
10
11  export default App;
12
```



M&T Tech

Spicing up the dashboard

- Dashboards should be visually appealing...so let's put a chart on it
- Stop the server by pressing ctrl-c and answering "y" to terminate the server
- Let's add some new packages:
  - Npm install react-router-dom --save
  - Npm install react-chartjs-2 chart.js axios--save
- We will use the charts in our new component, we'll need the router later

The Dashboard Component

① Add code into home.js

spa-demo > src > components > home > JS home.js > Home

```
import React from 'react';
import { Chart as ChartJS, ArcElement, Tooltip, Legend } from 'chart.js';
import { Pie } from 'react-chartjs-2';

ChartJS.register(ArcElement, Tooltip, Legend);

export const data = {
    labels: ['Motion Detected', 'No Motion'],
    datasets: [
      {
        label: 'Motion Sensor',
        data: [12, 19],
        backgroundColor: [
          'rgba(255, 99, 132, 0.2)',
          'rgba(54, 162, 235, 0.2)',
        ],
        borderColor: [
          'rgba(255, 99, 132, 1)',
          'rgba(54, 162, 235, 1)',
        ],
        borderWidth: 1,
      },
    ],
  };

export default function Home() {
    return (
        <Pie data={data} />
    )
}
```

② And to main.js:

main.js U | index.css M | App.js M | home.js U

spa-demo > src > layouts > main.js > Main

```
import Home from '../components/home/home'

function Main() {
    return (
        <div>
            <div id="dataDisplay" class="chart-container">
                <h1>Motion Detector</h1>
                <Home/>
            </div>
        </div>
    )
}
export default Main;
```

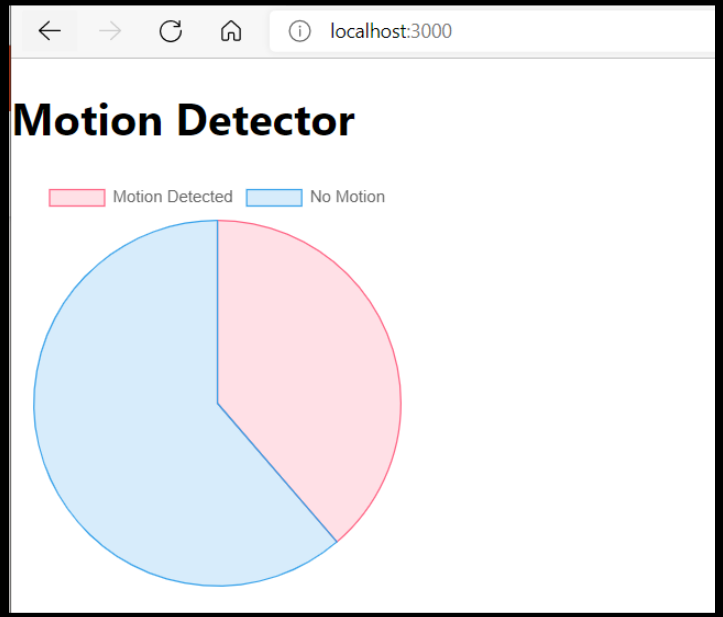
③ Some styling to index.css:

```
.chart-container {
    width:300px;
    height: 300px;
}
```


M&T Tech

Testing the component

- If we refresh our page, we should now see a chart being rendered:



M&T Tech

Let's build the API

- Let's make the dashboard dynamic
- In a new command window, we will install the pre-requisites using npm:
  - Create an API folder and cd into it
  - Npm init (initializes the package.json)
  - Npm install express cors --save
- Create a new file called index.js (or whatever your entrypoint was when running npm init)

Creating the API server

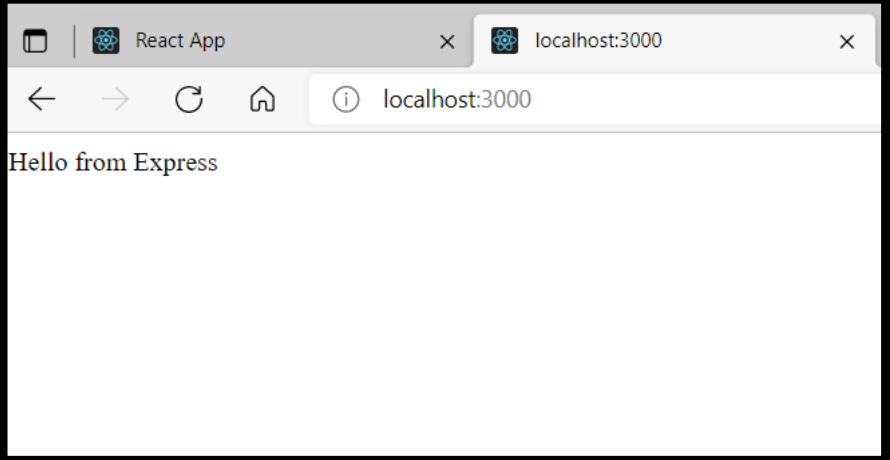
- Initialize express and create default endpoint:

```
var express = require('express');
var app = express();
const port=3000

app.get('/', function (req, res) {
    res.send('Hello from Express');
});

// start server on port
app.listen({port}, function () {
    console.log('Example app listening on port 3000.');
});
```

- Type node index.js and start the server



M&T Tech

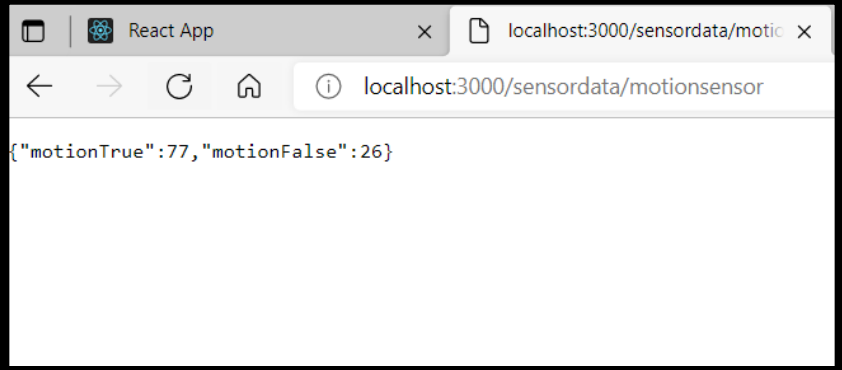
Creating an API endpoint

- Let's add a sensordata endpoint along with a 404 route, remove the default route:

```
app.get('/', function (req, res) {
    res.send('Not implemented');
});

app.get('/sensordata/:sensor', function(req,res) {
    console.log(req.params);
    var sensorName=req.params.sensor;
    res.json([77,26])
});

app.use(function(req, res, next) {
    res.status(404).send("Not Found");
});
```



M&T Tech

Oh oh....we are using the same port!

- At this point when you start and stop one of the servers you will notice that we are running the frontend and our API on port 3000
- To get around this, let's change our API to run on port 3001:

```
var express = require('express');
var app = express();
const port=3001
```

```
// start server on port
app.listen(port, function () {
    console.log('Example app listening on port ' + port);
});
```

Connecting React to the API

- We added axios to connect React to the API previously.
- Next we will implement our 'get'

- In the API, we need to implement CORS:

```
var express = require('express');
var cors = require('cors')
var app = express();
const port=3001

//allow CORS for all requests
app.use(cors())

```

```
import React, { useEffect, useState } from 'react';
import { Chart as ChartJS, ArcElement, Tooltip, Legend } from 'chart.js';
import { Pie } from 'react-chartjs-2';
import axios from 'axios'

ChartJS.register(ArcElement, Tooltip, Legend);

const baseURL = "http://localhost:3001/sensordata/motionsensor";
export default function Home() {


  const [sensorVals, setSensorVals] = useState({})
  useEffect(() => {
    axios.get(baseURL).then((response) => {

      setSensorVals(response.data)
    });
  }, []);

    return (
      <Pie data={{
        labels: ['Motion Detected', 'No Motion'],
        datasets: [
          {
            label: 'Motion Sensor',
            data: sensorVals,
            backgroundColor: [
              'rgba(255, 99, 132, 0.2)',
              'rgba(54, 162, 235, 0.2)',
            ],
            borderColor: [
              'rgba(255, 99, 132, 1)',
              'rgba(54, 162, 235, 1)',
            ],
            borderWidth: 1,
          },
        ]
      }} />
    )
}
```


M&T Tech

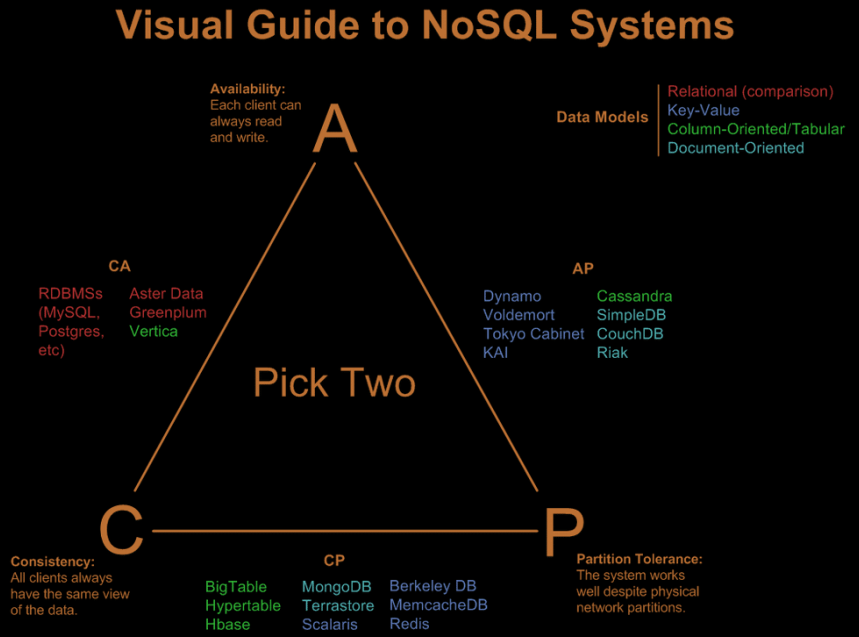
Let's create a database!

- Introducing Docker containers
  - Easy way to run without installing a server
  - Portable containers which can run on any platform
- Eventually, we will be putting everything in a container
- For now, we'll start with our database - Cassandra

Why Cassandra?

- P2P architecture
  - No single point of failure
- Very scalable
- High Availability
- Extremely fast writes
- Non Relational (loose schema)

This makes it well suited for “big data”



M&T Tech

Keyspace model

- A keyspace is a container for 1+ column families
  - A column family is a collection of rows
  - Each row contains data and columns

- Similar to "Tables" in a RDBMS except:
  - You can add columns any time you want
  - Cassandra does not force rows to have all the columns

- A column is the basic data structure, which has a name, value, and time stamp

- Relationships are represented with collections not joins or keys

Creating & starting the database server

- Docker run –p 9042:9042 --name Cassandra-sensor –v c:/mount/Cassandra:/var/lib/Cassandra –d cassandra:latest

This command will download the latest image of cassandra and run it with a local volume (c:/mount/cassandra) mapping to the image data location

```
PS C:\SPA class> docker run --name cassandra-sensor -v c:/mount/cassandra:/var/lib/cassandra -d cassandra:latest
Unable to find image 'cassandra:latest' locally
latest: Pulling from library/cassandra
7b1a6ab2e44d: Pull complete
5c122ee837fa: Pull complete
060f1998a9ab: Pull complete
3d68b8b292ea: Pull complete
b5aa81e9d8a3: Pull complete
ab8a264b599d: Pull complete
1c8983c926a1: Pull complete
1802c7f27836: Pull complete
dfcd8d520983: Pull complete
Digest: sha256:c186d79dca1cebe1ce382aaba201cce08b0f415e850d4b9eb6c1317f02bea293
Status: Downloaded newer image for cassandra:latest
10a19c61cec918b7c1e610930d72723d2d6bf3ce348dd200d97e218f67565acf
PS C:\SPA class>
```

Connecting and init the database

①

- Connect to the containerized db using docker desktop, going to container list and selecting CLI from the running container

--OR—

- Docker exec –it <name> /bin/sh

②

```
CREATE KEYSPACE sensordata WITH replication
= {'class':'SimpleStrategy', 'replication_factor':1};
USE sensordata;
CREATE COLUMNFAMILY sensordata(
sensor_name text,
entry_date timestamp,
motion Boolean,
primary key (sensor_name,entry_date));
```

```
cqlsh:sensordata> INSERT INTO sensordata (sensor_name,entry_date,motion) VALUES ('demosensor',toTimeStamp(now()),false);
cqlsh:sensordata> INSERT INTO sensordata (sensor_name,entry_date,motion) VALUES ('demosensor',toTimeStamp(now()),false);
cqlsh:sensordata> INSERT INTO sensordata (sensor_name,entry_date,motion) VALUES ('demosensor',toTimeStamp(now()),false);
cqlsh:sensordata> INSERT INTO sensordata (sensor_name,entry_date,motion) VALUES ('demosensor',toTimeStamp(now()),true);
cqlsh:sensordata> INSERT INTO sensordata (sensor_name,entry_date,motion) VALUES ('demosensor',toTimeStamp(now()),true);
cqlsh:sensordata> INSERT INTO sensordata (sensor_name,entry_date,motion) VALUES ('demosensor',toTimeStamp(now()),false);
cqlsh:sensordata>
```


M&T Tech

Skipping ahead...

- In the interest of time, we are going to move to Routing & deployment

- The code repo has several improvements to our application:

    - Adding and using bootstrap (the React way!)
    - Adding the card UI for each dashboard component
    - Adding another component (sensor health)
    - Consuming API services in React

Introducing Routing...

- Routing adds more “pages” to our SPA
- Let’s start by putting the component on its own “page” & implement a navigation component:

spa-demo > src > components > nav > JS nav.js > Navbar

```
1  import React from 'react'
2  import {Nav} from 'react-bootstrap'
3
4  export default function Navbar(){
5
6      return(
7          <Nav variant="tabs" defaultActiveKey="/">
8              <Nav.Item>
9                  <Nav.Link href="/">Overview</Nav.Link>
10             </Nav.Item>
11             <Nav.Item>
12                 <Nav.Link>Sensor Health</Nav.Link>
13             </Nav.Item>
14             </Nav>
15     )
16 }
```

spa-demo > src > layouts > JS main.js > Main

```
1
2   import Home from '../components/home/home'
3   import Health from '../components/health/health';
4   import Navbar from '../components/nav/nav'
5   import {Container, Row} from 'react-bootstrap'
6
7   function Main() {
8       return (
9           <Container fluid>
10              <Row>
11                  <Navbar />
12              </Row>
13              <Row id="dataDisplay">
14                <h1>Welcome to the sensor console</h1>
15                <Home/>
16                <Health />
17              </Row>
18          </Container>
19      )
20  }
21  export default Main;
```

Router Configuration

Modify main js:

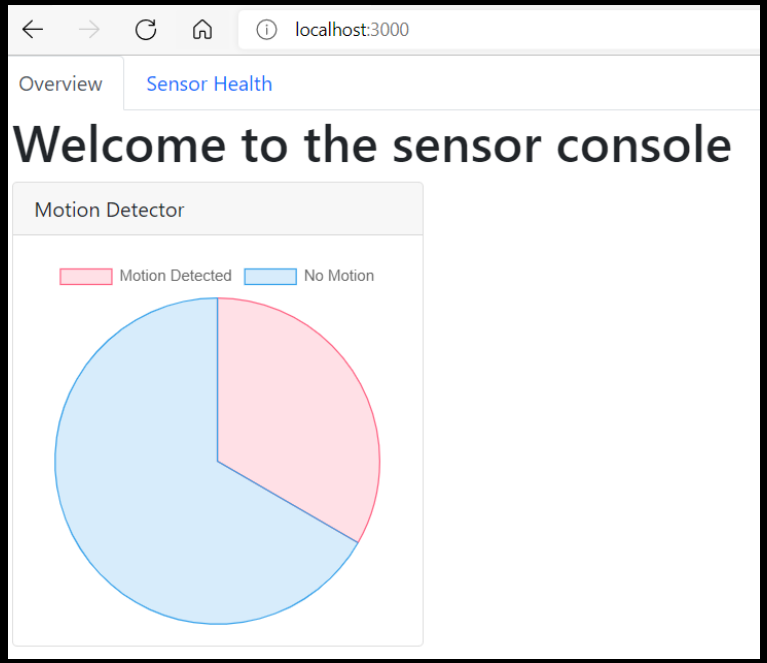
```
6   import { Route, Routes, BrowserRouter } from 'react-router-dom'
7
8   function Main() {
9       return (
10          <Container fluid>
11              <Row>
12                  <Navbar />
13              </Row>
14              <Row id="dataDisplay">
15                <BrowserRouter>
16                  <Routes>
17                      <Route path="/" exact element={<Home />}></Route>
18                      <Route path="/health" element={<Health />}></Route>
19                  </Routes>
20                </BrowserRouter>
21              </Row>
22          </Container>
23      )
24  }
```

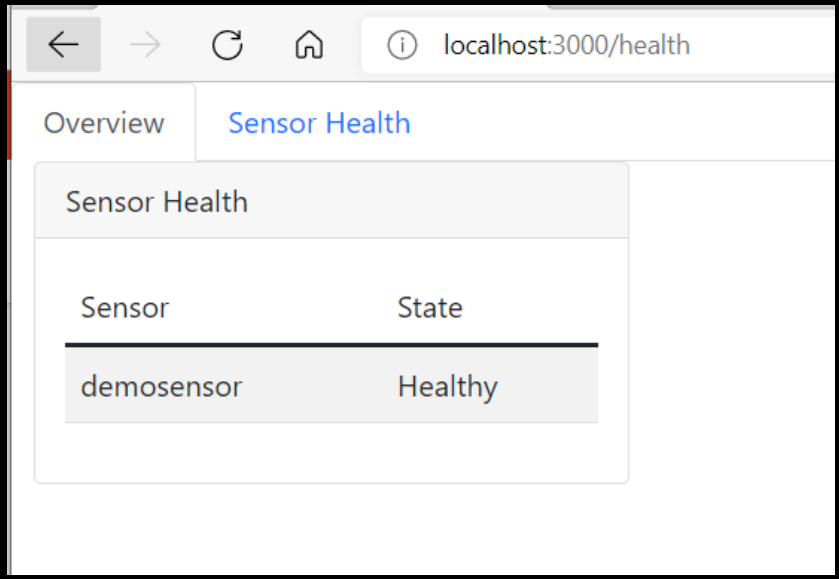
Modify nav:

```
5
6       return(
7           <Nav variant="tabs" defaultActiveKey="/">
8               <Nav.Item>
9                   <Nav.Link eventKey="/" href="/">Overview</Nav.Link>
10              </Nav.Item>
11              <Nav.Item>
12                  <Nav.Link eventKey="health" href="/health">Sensor Health</Nav.Link>
13              </Nav.Item>
14              </Nav>
15      )
```


M&T Tech

Test!


localhost:3000
Overview
Sensor Health
Welcome to the sensor console
Motion Detector
Motion Detected
No Motion


localhost:3000/health
Overview
Sensor Health
Sensor Health
Sensor
State
demosensor
Healthy

M&T Tech

Building & deploying images into containers

- Now that we have our app “developed” let’s make an image of it

- We will have 2 images:
  - React Application
  - API

- Create a Dockerfile in the root of the SPA application folder
- We will use a multistage dockerfile to reduce the image size
- We will use a .dockerignore to manage the image contents

Our Dockerfiles

Dockerfile U

spa-demo > Dockerfile > ...

```
# STAGE 1: First grab the node base image
FROM node:14-alpine as build
# Ensure root user
USER 0
# Create a working directory
RUN mkdir -p /home/1001/app
# Switch to that directory & copy the package manifests
WORKDIR /home/1001/app
COPY --chown=1001 package*.json /home/1001/app/
# Install the packages required
RUN npm install
# Copy our source and supporting files
COPY --chown=1001 . .
# Run the build script
RUN npm run build

# STAGE 2: Grab the nginx web server image
FROM nginx:1.21.4-alpine
# Establish the port
ARG port=80
# Switch to the html default directory
WORKDIR /usr/share/nginx/html
# Copy the compiled app from previous stage
COPY --from=build /home/1001/app/build/ .
# Expose the port
EXPOSE ${port}
# Start the nginx server with daemon off
CMD ["nginx", "-g", "daemon off;"]
```

Dockerfile

api > Dockerfile > ...

```
# Grab the node base image
FROM node:17-alpine3.12
# Ensure root user
USER 0
# Create a working directory
RUN mkdir -p /home/1001/app
# Switch to that directory & copy the package manifests
WORKDIR /home/1001/app
COPY --chown=1001 package*.json /home/1001/app/
# Install the packages required
RUN npm install
# Copy our source
COPY --chown=1001 . .
# Set to a non-root built-in user `node`
USER 1001
# Bind to all network interfaces so that it can be mapped to the host OS
ENV HOST=0.0.0.0 PORT=3000
EXPOSE ${PORT}/tcp
CMD [ "node", "." ]
```

M&T Tech

Building the images

```
PS C:\SPA class\spa-demo> docker image build --force-rm .
[+] Building 40.1s (17/17) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 949B
 => [internal] load .dockerignore
 => => transferring context: 34B
 => [internal] load metadata for docker.io/library/nginx:1.21.4-alpine
 => [internal] load metadata for docker.io/library/node:14-alpine
 => [stage-1 1/4] FROM docker.io/library/nginx:1.21.4-alpine@sha256:12aa12ec4a8ca049537dd486044b966b0ba6cd8890c4c900ccb5e7e630e03df0
 => [build 1/7] FROM docker.io/library/node:14-alpine@sha256:7bcf853eeb97a25465cb385b015606c22e926f548cbd117f85b7196df8aa0d29
 => [internal] load build context
 => => transferring context: 1.77MB
 => CACHED [build 2/7] RUN mkdir -p /home/1001/app
 => CACHED [build 3/7] WORKDIR /home/1001/app
 => CACHED [build 4/7] COPY --chown=1001 package*.json /home/1001/app/
 => CACHED [build 5/7] RUN npm install
 => CACHED [stage-1 2/4] COPY nginx/nginx.conf /etc/nginx/nginx.conf
 => CACHED [stage-1 3/4] WORKDIR /usr/share/nginx/html
 => [build 6/7] COPY --chown=1001 . .
 => [build 7/7] RUN npm run build
 => [stage-1 4/4] COPY --from=build /home/1001/app/build/ .
 => exporting to image
 => => exporting layers
 => => writing image sha256:557df4404978298ccf11531f4c5631990b76478af1fe8386cbd085dfb17c73fb
```

Hmmmm….it would be nice if we could name the image something other than the SHA2 string

Docker image build –force-rm –tag reactapp:v1

M&T Tech

Deploy locally

You can use the command docker image ls to get the name of your image, or if you used tag, just run it:

docker run –p 80:80 –d reactapp:v1
docker run –p 3000:3000 –d

Note: If you don't run the command with the –d (detached mode) it will run in your console window.

```
PS C:\SPA class\api> docker run -p 3000:3000 -d nodeapi:v1
7462074354d53aee2fb2d5ffda02b1549f503af68c66d1106e327038209d32ad
PS C:\SPA class\api>
```

Adding Kubernetes (k8s)

We do this through yml files

We will combine the api and react app in the same deployment
    They are still separate images but will be in the same POD

The yml files are in the repo and are commented...but here they are:

Service / Deployment

k8s > ! services.yml

```yaml
# Here we define the services that will handle the routing from external systems
apiVersion: v1
kind: Service
# name the service
metadata:
  name: react-service
  labels:
    app: reactspa
spec:
  # define the load balancer as the type for the services
  type: LoadBalancer
  # we define a valid IP for our cluster in the allowed range so we can tell the LB where to push traffic to
  # if we comment this out, the cluster IP will be regenerated each time the cluster is rebuilt
  clusterIP: 10.96.47.163
  # if the load balancer has a set IP, set it here
  #loadBalancerIP: XXXXX
  # target the "reactspa" application, or whatever is set in deployment
  selector:
    app: reactspa
  # uncommenting the below will enable routing all traffic from clientip to the same pod after it reaches the
  #sessionAffinity: ClientIP
  # define each container with a service route, which will allow communication from the Load Balancer to a con
  # the name must be unique!
  ports:
    - name: ui-service
      protocol: TCP
      port: 80
      targetPort: 80
    - name: api-service
      protocol: TCP
      port: 3000
      targetPort: 3000
```

```yaml

# create the pods as a deployment
apiVersion: apps/v1
# setup as a deployment so we can scale
kind: Deployment
metadata:
  # this is the name of the pod
  name: reactapp-deployment
  labels:
    # label the app so the load balancer can see it
    app: reactspa
spec:
  # this is the number of identical pods we want to create
  replicas: 3
  # the selector.matchLabels.app should match the app name
  selector:
    matchLabels:
      app: reactspa
  template:
    metadata:
      labels:
        app: reactspa
    spec:
      # here we define the containers themselves
      containers:
      - name: ui
        # this is the image we want to build from, can be <imagename> or to target a specific tag, <imagename>:<tag>
        image: reactapp:v1
        # we specify Never, so it will not go to the docker.io registry to pull and build a container
        imagePullPolicy: Never
        # container port to expose
        ports:
        - containerPort: 80
      - name: api
        image: nodeapi:v1
        imagePullPolicy: Never
        ports:
        - containerPort: 3000
```

Starting it up!

```
PS C:\SPA class\k8s> kubectl apply -f services.yml
service/react-service created
PS C:\SPA class\k8s> kubectl apply -f deployment.yml
deployment.apps/reactapp-deployment created
PS C:\SPA class\k8s> kubectl rollout status -f deployment.yml
deployment "reactapp-deployment" successfully rolled out
PS C:\SPA class\k8s> kubectl get svc,deployment,pods -l app=reactspa
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                       AGE
service/react-service   LoadBalancer   10.96.47.163   <pending>     80:32444/TCP,3000:31676/TCP   68s

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/reactapp-deployment   3/3     3            3           51s

NAME                                       READY   STATUS    RESTARTS   AGE
pod/reactapp-deployment-78569d88c9-bvs55   2/2     Running   0          51s
pod/reactapp-deployment-78569d88c9-hxdwk   2/2     Running   0          51s
pod/reactapp-deployment-78569d88c9-m26th   2/2     Running   0          51s
PS C:\SPA class\k8s>
```

Testing:

- Delete or stop one or more containers
- A new one will be started, as we need to have 3 fully running pods

Stopping & cleaning up

```
PS C:\SPA class\k8s> kubectl delete -f deployment.yml
deployment.apps "reactapp-deployment" deleted
PS C:\SPA class\k8s> kubectl delete -f service.yml
error: the path "service.yml" does not exist
PS C:\SPA class\k8s> kubectl delete -f services.yml
service "react-service" deleted
PS C:\SPA class\k8s> kubectl get svc,deployment,pods -l app=reactspa
NAME                                        READY   STATUS        RESTARTS   AGE
pod/reactapp-deployment-78569d88c9-bvs55    2/2     Terminating   0          11m
pod/reactapp-deployment-78569d88c9-hxdwk    2/2     Terminating   3          11m
pod/reactapp-deployment-78569d88c9-m26th    2/2     Terminating   1          11m
PS C:\SPA class\k8s> kubectl get svc,deployment,pods -l app=reactspa
No resources found in default namespace.
PS C:\SPA class\k8s>
```

kubectl delete –f services.yml
kubectl delete –f deployment.yml

Now what?

- Adding security
- Adding more components
- With minor modifications, can deploy to a cloud environment such as AWS, etc
- Env files
- Potentially split into separate api specific container?

Thank you!

M&T Tech