Oracle Content & Experience Cloud
Technical Workshop

Lab – Going ecstatic

Contents

1. Introduction

In this Lab you will be creating a static website using content from Content & Experience Cloud.

The intent of the Lab is experience a bit of the art of the possible. It provides an alternative path of managing and deploying websites, using technologies like React and Kubernetes.

You will go through a series of steps whereby from a simple HelloWorld you'll go to building and deploying the homepage of the Café Supremo sample site and enhancing it with a conversational chat-like component.

In this Lab you will not be using the content creation and management features of Content & Experience Cloud. Assume thaty the content has been created and is managed by editors in Content & Experience Cloud. You will be using the the REST Delivery API to access the content to be used on the website.

The website used in the Lab is built with React.

Once finished creating the site, you will be using Docker and Kubernetes (Minikube) to deploy the site.

minikube

kubernetes

node

docker

SUPREMO
COFFEE & TEA

2. Pre-requisites

As the application is built with HTML5 and React, some familiarity with JavaScript, React, CSS3 and HTML5 are prerequisites.

Another prerequisite is a laptop with at least 16Gb of memory.

3. Prep work

This lab makes use of many tools. To get going with the steps to build the website you'll need node, yarn and git.

Install NodeJS

https://nodejs.org/en/download/package-manager/

Install Yarn

https://yarnpkg.com/lang/en/docs/install/

Install Git

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

4. Alternative learning path

This course is made up of 4 parts

1. Basic React and CEC content integration
2. Adding a conversational interface
3. Building and deploying to Docker
4. Deploying to Kubernetes/Minikube.

Depending on your own interests you can do part of the Basic React training, and then skip to step 8 to build the site. You can take that result forward and deploy to Docker and /or Kubernetes as described in the steps after 8.

In any case you should do step1 and step8 if you want to do the Docker/Kubernetes deployment.

5. Step 1: Hello World

The first step is to get you up and running with the simple HelloWorld. The sample has been prepared for you, to download from github: https://github.com/dolfdijkstra/cafe-supremo-react-front.

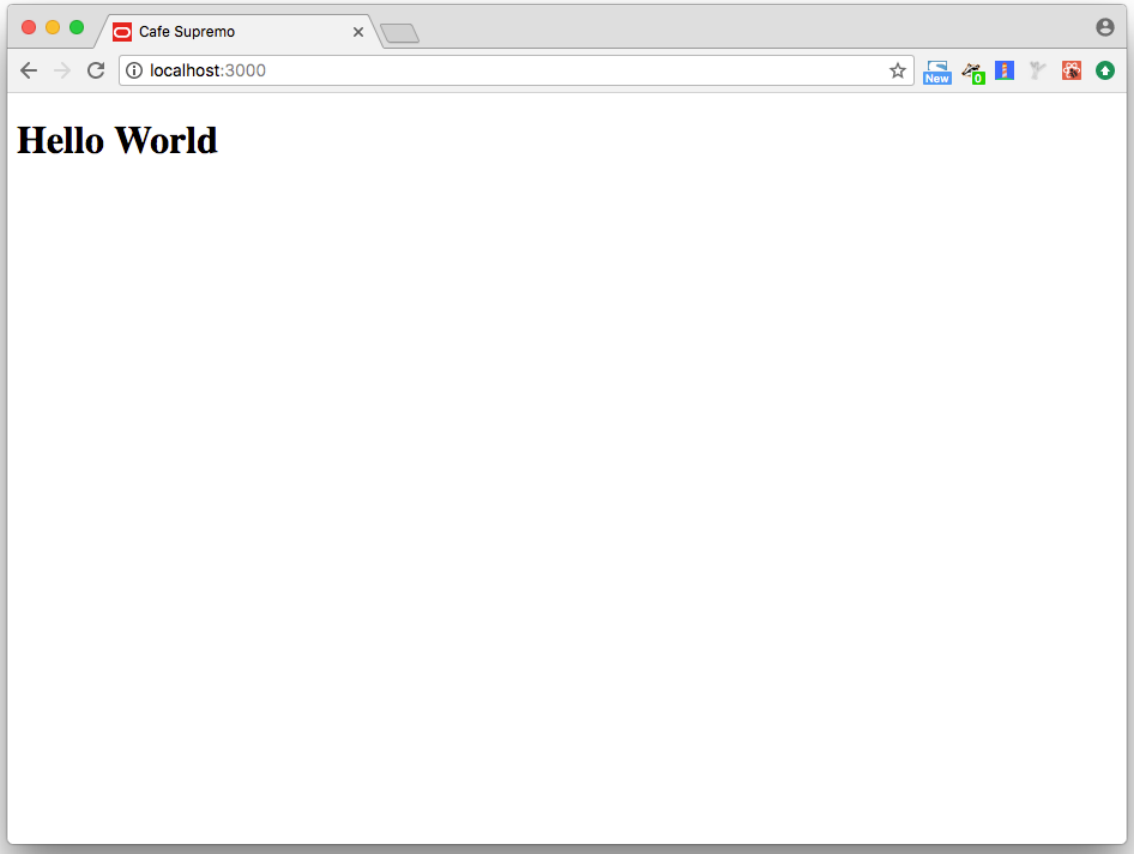
Open a terminal or command window, cd into a location where you want to work and execute the following commands to

```
> git clone https://github.com/dolfdijkstra/cafe-supremo-react-front.git
> cd cafe-supremo-react-front
> yarn install
> yarn start
```

Instead of cloning, you can also fork the repository. This allows you to maintain your own source history if want to keep a history of your changes.

This should download the code to work with, install the dependent NPM modules, and run the React application. It should also automatically open a browser to http://loalhost:3000.

Your browser should look like this:



The code for this HelloWorld sample is in src/step1. Please inspect the code. You'll need to open another terminal.

The React application is a front-end application. All code runs in the browser. The browser loads the index.html from the public folder, and the React development server (that you started with yarn start earlier) has enhanced this html file and injected index.js into it. There is technically more to it, but for now this explanation should keep you on track.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Cafe Supremo</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="author" content="">
    <link rel="shortcut icon" href="favicon.ico">
</head>
<body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
</body>
</html>
```

index.html loads src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App.jsx'

import registerServiceWorker from './registerServiceWorker'

ReactDOM.render(<App />, document.getElementById('root'))
registerServiceWorker()
```

index.js loads src/App.jsx. The JSX files mix JavaScript with HTML and is a very easy way to construct HTML pages. If you want to read more on JSX, please consult https://reactjs.org/docs/introducing-jsx.html.

```
import React, { Component } from 'react'
import Layout from './step1/Layout'

export default class App extends Component {
  render () {
    return (
      <React.Fragment>
        <Layout />
      </React.Fragment>
    )
  }
}
```

App.jsx loads and uses src/step1/Layout.jsx

```
import React, { Component } from 'react'

export default class Layout extends Component {
  render () {
    return (
      <div>
            <h1>Hello World</h1>
      </div>
    )
  }
}
```

To continue with the rest of the steps you will need to change src/App.jsx each time and change the location of the Layout to use, from step1 to the next steps.

If HelloWorld shows in your browser, this step is completed successfully.

6. Step 2: Render a List of items

The next exercise is to enhance the Layout.jsx file and display a list of content items read from a JSON file.

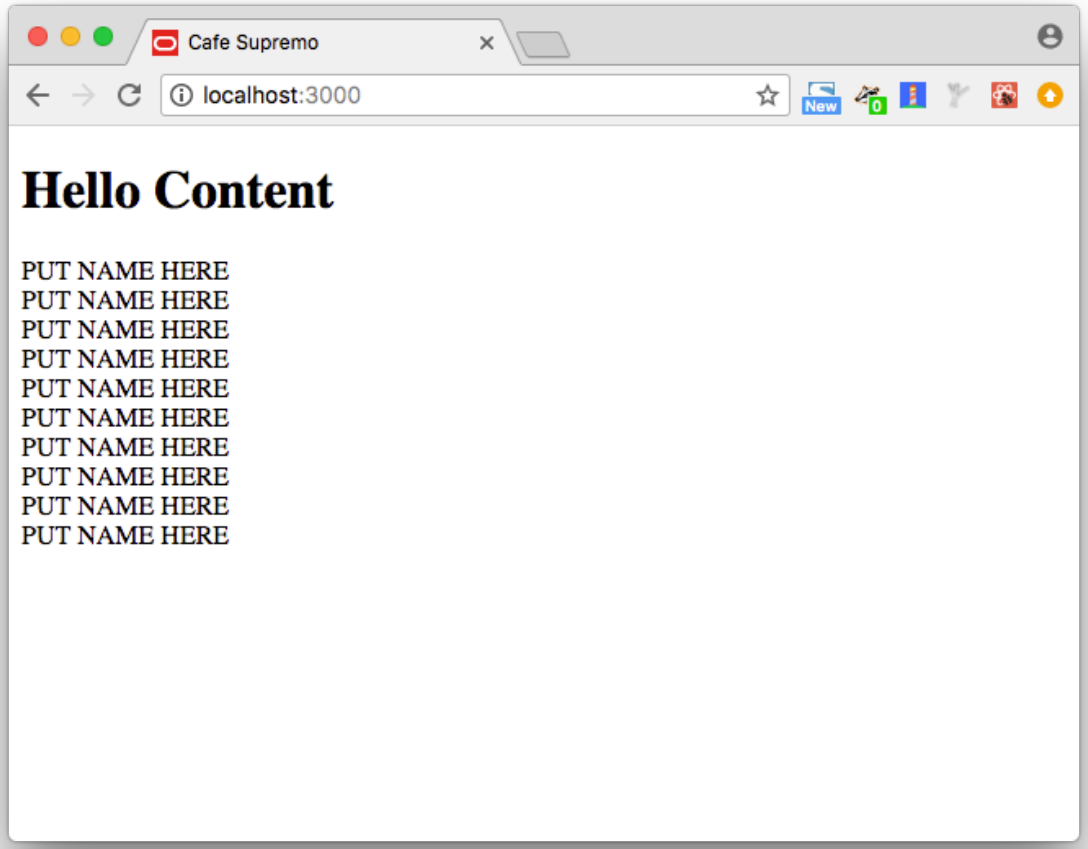
```
import React, { Component } from 'react'

import content from './content.json'
const allItems = content.ALL.data.items


const items = allItems.filter(item => item.type === 'Blog')

export default class Layout extends Component {
  render () {
    return (
      <div>
            <h1>Hello Content</h1>
            {items.map((item,index) => <div key={index}>PUT NAME HERE</div>)}
      </div>
    )
  }
}
```

As you can see there are some enhancements compared to step1. First a content object is loaded from the content.json file. You can inspect this file to understand the structure of the JSON document.

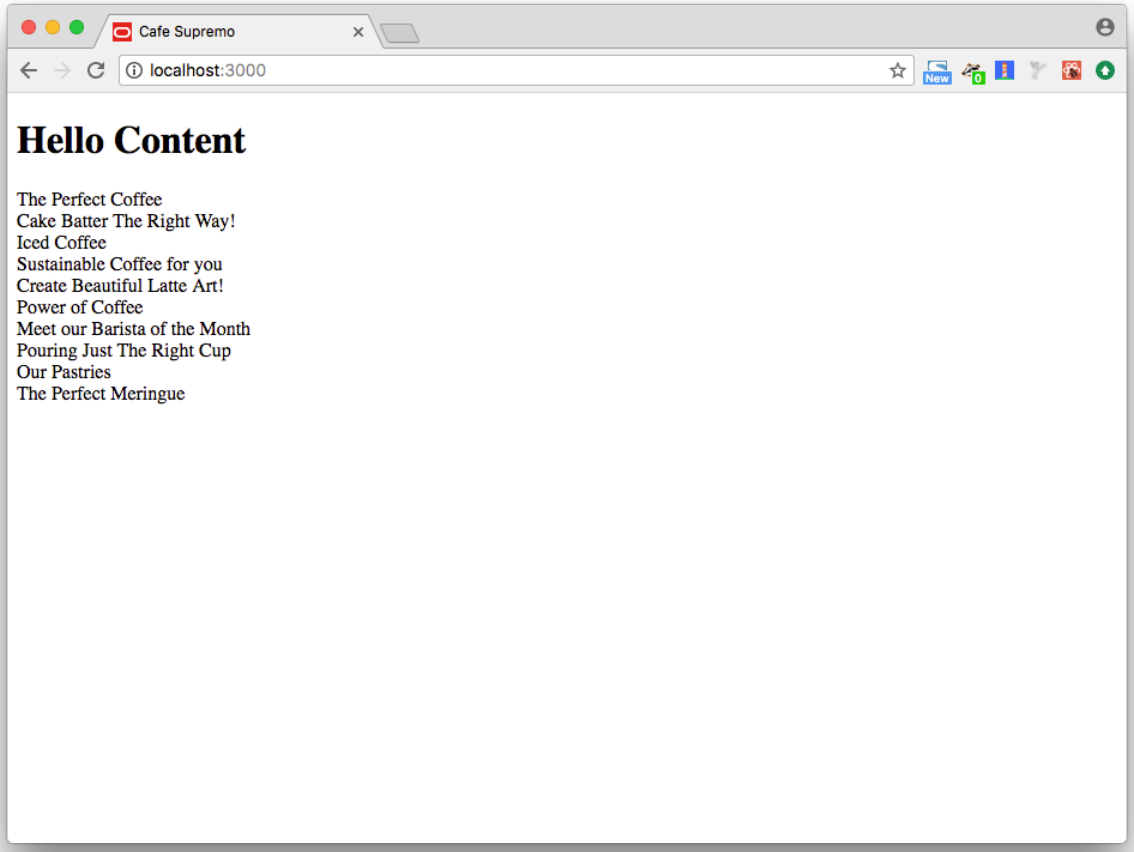
The the content is filtered for only the 'Blog' items. In the next exercises we will only use the Blog items.

Lastly, the HTML is extended to render a number of child div elements, one for each Blog iterm.

To load this file, change App.jsx and change step1 to step2 to load the step2 Layout. If you save this file, your browser should automatically reload.



It is you task to change the 'PUT NAME HERE' in the Layout.jsx so the actual namers of the content items get displayed. You can read at https://reactjs.org/docs/introducing-jsx.html to understand the JSX syntax. A hint: you'll need to swap PUT NAME HERE with {item.name}. Save and check your browser window.


Cafe Supremo
localhost:3000
Hello Content
The Perfect Coffee
Cake Batter The Right Way!
Iced Coffee
Sustainable Coffee for you
Create Beautiful Latte Art!
Power of Coffee
Meet our Barista of the Month
Pouring Just The Right Cup
Our Pastries
The Perfect Meringue

7. Step 3: Extracting the Item Component

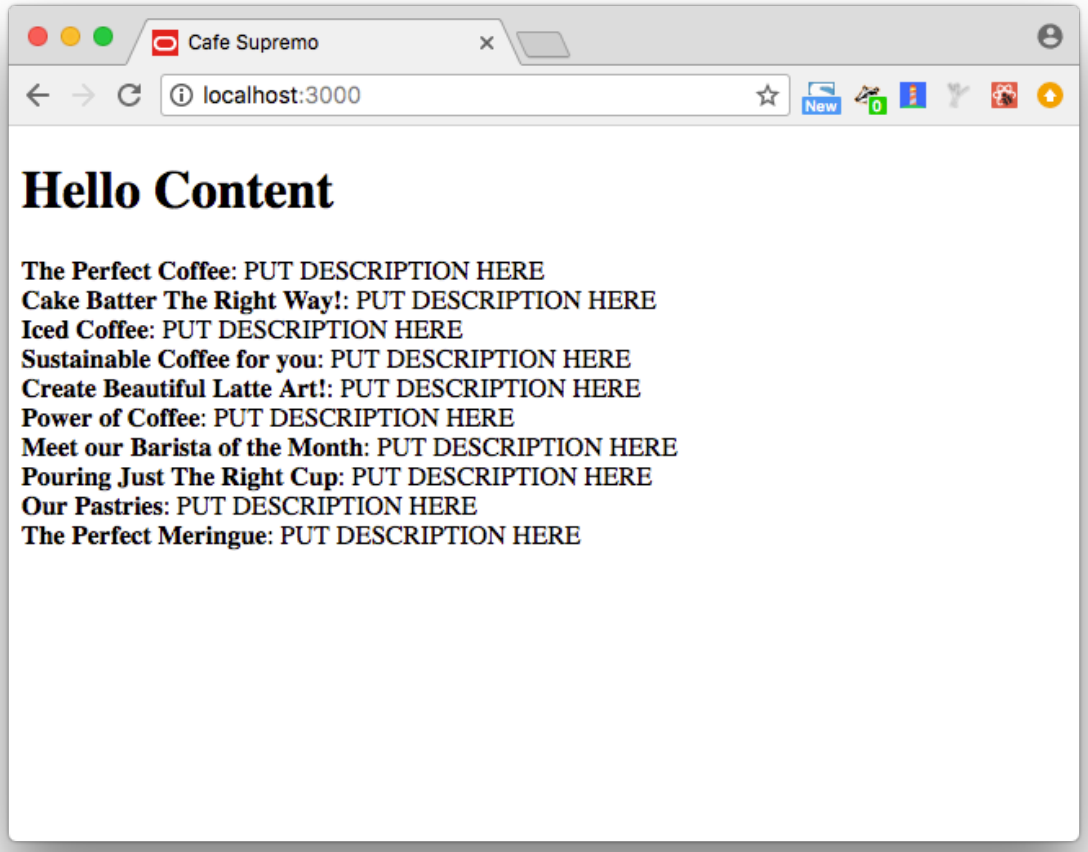
In the previous step we have enhanced the Layout to render a list of Blog items. You'll now further enhance the Layout to render a named Component for the Blog item.

```
import React, { Component } from 'react'

import content from './content.json'
const allItems = content.ALL.data.items

const items = allItems.filter(item => item.type === 'Blog')

export default class Layout extends Component {
  render () {
    return (
      <div>
            <h1>Hello Content</h1>
            {items.map((item,index) => <Blog key={index} item={item}/>)}
      </div>
    )
  }
}

const Blog = ({item}) => <div><strong>{item.name}</strong>: PUT DESCRIPTION
HERE</div>
```

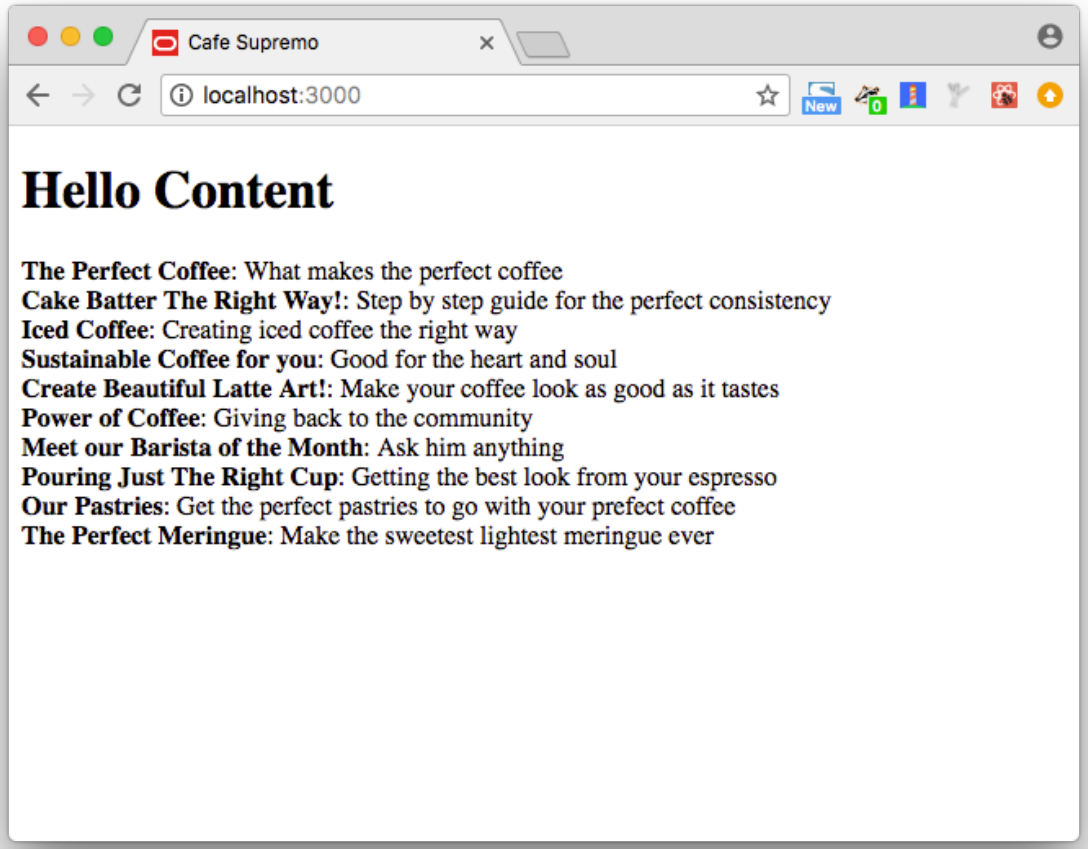
As you can see, the line with items.map changed from a to a <Blog />. This tell React to render the Blog component. The Blog component is defined in the same file.

When you look at the source, what will you need to change to display beside the name also the description? Please make the required changes.

To use this Layout, change step2 to step3 in App.jsx.



Outcome should be


Cafe Supremo
localhost:3000
Hello Content
The Perfect Coffee: What makes the perfect coffee
Cake Batter The Right Way!: Step by step guide for the perfect consistency
Iced Coffee: Creating iced coffee the right way
Sustainable Coffee for you: Good for the heart and soul
Create Beautiful Latte Art!: Make your coffee look as good as it tastes
Power of Coffee: Giving back to the community
Meet our Barista of the Month: Ask him anything
Pouring Just The Right Cup: Getting the best look from your espresso
Our Pastries: Get the perfect pastries to go with your prefect coffee
The Perfect Meringue: Make the sweetest lightest meringue ever

8. Step 4: Enhancing the Item Component

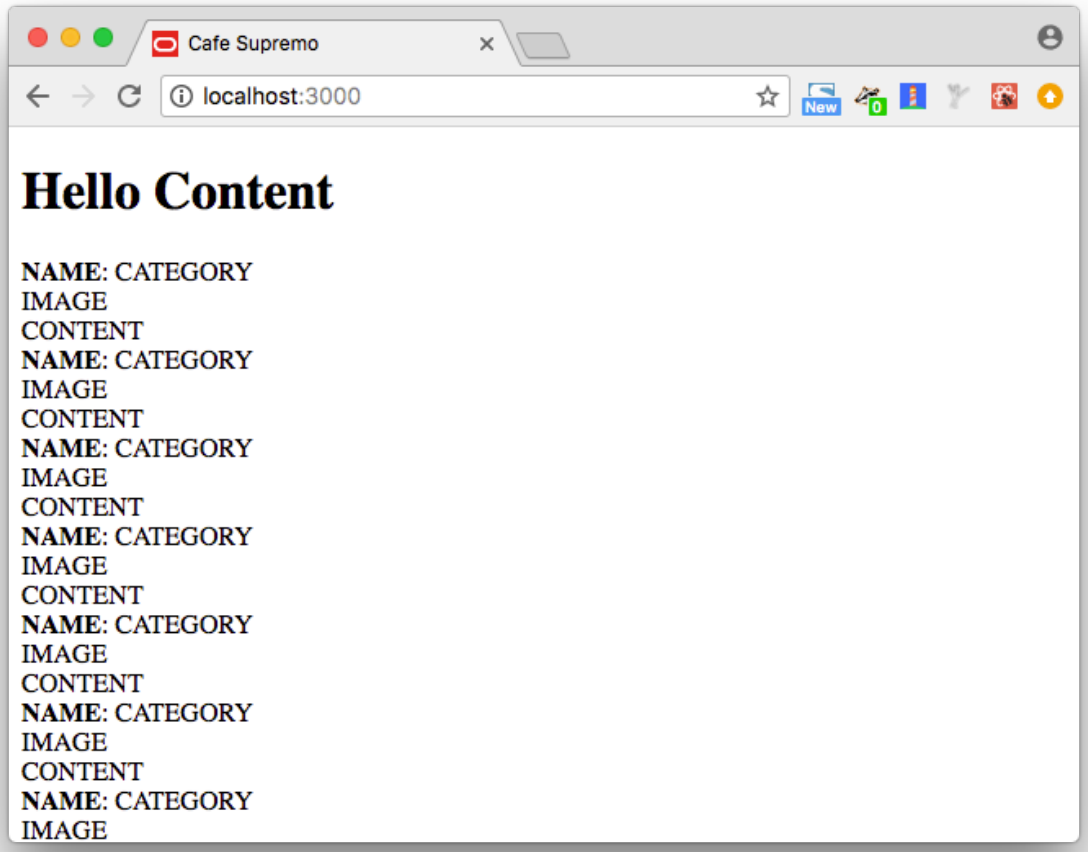
To display just name and description is not that visually appealing. To make things engaging, you'll need images. In this step you'll enhance the Layout and Blog components further to display images and the body of the Blog items.

If you inspect src/step4/Layout.jsx, you'll notice

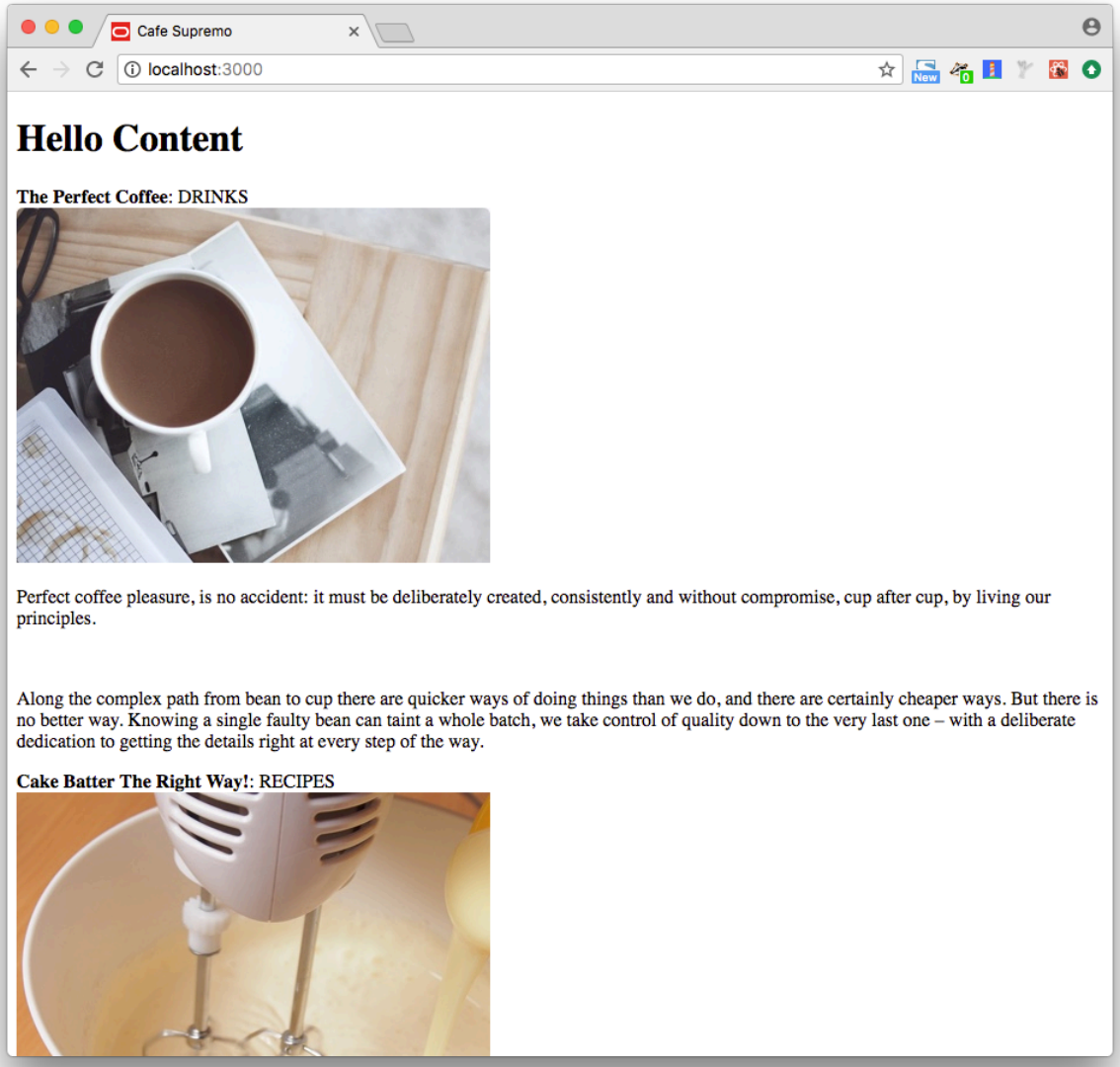
```
import React, { Component } from 'react'

import content from './content.json'

const allItems = content.ALL.data.items

const items = allItems.filter(item => item.type === 'Blog')

const fullItem = item =>content[item.id].data

const toHref = link => link.href.replace('/items/', '/digital-assets/')
+'/default'

const fields = ["blog_category", "blog_textposition", "blog_textcolor",
"blog_content", "blog_image_ad", "blog_image_thumbnail", "blog_image_header",
"blog_image_ad_small", "blog_author"]

export default class Layout extends Component {
  render () {
    return (
      <div>
            <h1>Hello Content</h1>
            {items.map(fullItem).map((item,index) => <Blog key={index}
item={item}/>)}
      </div>
    )
  }
}

const Blog = ({item}) => {
    const {data} = item
    const { blog_category, blog_textposition, blog_textcolor, blog_content,
blog_image_ad, blog_image_thumbnail, blog_image_header, blog_image_ad_small,
blog_author} = data
    const content = {__html:blog_content} // to be used with
dangerouslySetInnerHTML
    const image = toHref(blog_image_thumbnail.link)

    return (<div>
            <div><strong>NAME</strong>: CATEGORY</div>
            <div>IMAGE</div>
            <div>CONTENT</div>
        </div>)
}
```

The code has again been enhanced:

- There is now a new function call fullItem that retrieves all the content for an item for the content store.
- There is a function toHREF to build a href for an image to be used in a <img> tag.
- There is a fields value that holds a list of field names. This is for reference only.
- The Blog component is enhanced. The const {data} = item and the next line are JavaScript constructs to extract values from an object.



The task is to make it look like this:



The task is similar to the previous exercises, though there is likely a caveat with the rendering of the blog_content. Can you figure out how to fix display this content as HTML and not as escaped HTML? Have a look at https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml.

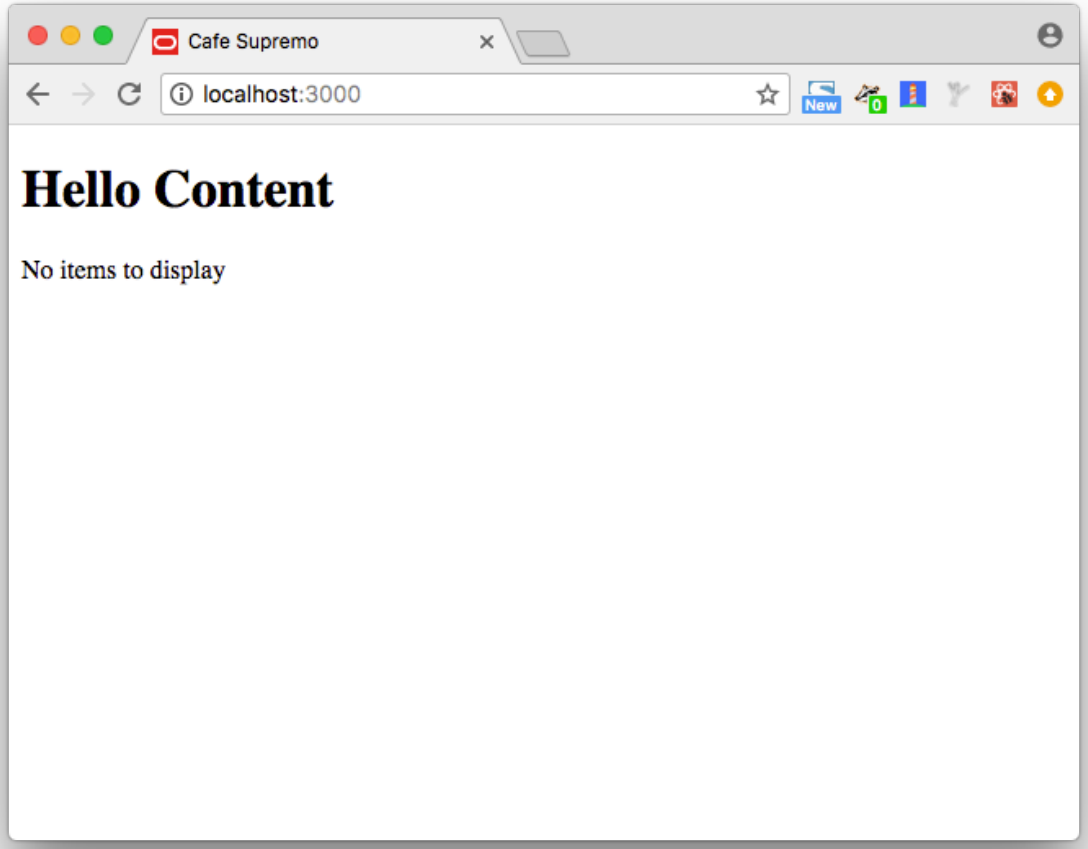
9. Step 5: Fetching some fresh content

So far all the content has been provided to you in a JSON file. What happens if the editorial team publishes a new Blog article, how do you get that displayed on your site?

There are two ways to handle that: you could load the content at runtime by the browser though an Ajax call, or, as we will do here, retrieve the content at build time.

Again: change App.jsx to use the Layout from step5.

The page should now show




```
import React, { Component } from 'react'

import content from './content.json'

const allItems = content.ALL.data.items

const items = allItems.filter(item => item.type === 'Blog')
```

```
const fullItem = item =>content[item.id].data

const toHref = link => link.href.replace('/items/', '/digital-assets/')
+'/default'

const fields = ["blog_category", "blog_textposition", "blog_textcolor",
"blog_content", "blog_image_ad", "blog_image_thumbnail", "blog_image_header",
"blog_image_ad_small", "blog_author"]

export default class Layout extends Component {
  render () {
    return (
      <div>
            <h1>Hello Content</h1>
            {items.length == 0 ? <NoItems/>:<List/>}
      </div>
    )
  }
}
const NoItems = () => <div>No items to display</div>
const List = () => <div>{items.map(fullItem).map((item,index) => <Blog
key={index} item={item}/>)}</div>


const Blog = ({item}) => {
    const {data} = item
    const { blog_category, blog_textposition, blog_textcolor, blog_content,
blog_image_ad, blog_image_thumbnail, blog_image_header, blog_image_ad_small,
blog_author} = data
    const content = {__html:blog_content}
    const image = toHref(blog_image_thumbnail.link)

    return (<div>
            <div><strong>{item.name}</strong>: {blog_category}</div>
            <div><img src={image}/></div>
            <div dangerouslySetInnerHTML={content}/>
        </div>)
}
```

The Layout.jsx has been slightly changed from step4. In essence the case where no content is found is handled gracefully.

What are there no items to display? Have a look at content.json.

The task is to retrieve the content from CEC by means of a small node module.

```
const fetch = require('node-fetch')
const path = require('path')

const fs = require('fs')

const token = '11ad271cc1ae61bd03249332e8445c96'
const host = 'https://demo-gse00009991.sites.us2.oraclecloud.com'

const itemsURL = ({ maxResults, sortOrder }) =>
  `${host}/content/published/api/v1/items?contentType=published&orderBy=${esc(
    sortOrder
```

```
  )}&limit=${maxResults}&access-token=${token}`

const write = data => {
    fs.writeFileSync(path.join(__dirname , '/content.json'),
JSON.stringify(data,null,2))
    return data
}

const esc = encodeURIComponent

const noDigitalAssets = e => e.type !== 'DigitalAsset'

const fetchItem = link => {
  return fetch(`${link.href}?access-token=${token}`)
    .then(r => r.json())
    .then(data => ({ [link.id]: { data: data } }))
}
const fetchItems = data => {
  const links = data.ALL.data.items
    .filter(noDigitalAssets)
    .map(e => ({ id: e.id, href: e.link.href, rel: e.link.rel }))
    .sort((a, b) => a.href.localeCompare(b.href))
    .filter((e, i, a) => a.indexOf(e) === i)
  const itemFetches = links.map(fetchItem)
  return Promise.all(itemFetches)
    .then(a => a.reduce((a, e) => Object.assign(a, e), {}))
    .then(r => Object.assign(data, r))
}
const fetches =
    fetch(itemsURL({ maxResults: 500, sortOrder: 'updateddate:desc' }))
      .then(response => response.json())
      .then(data => ({ ALL: { data: data} }))


fetches
  .then(fetchItems)
  .then(write)
```

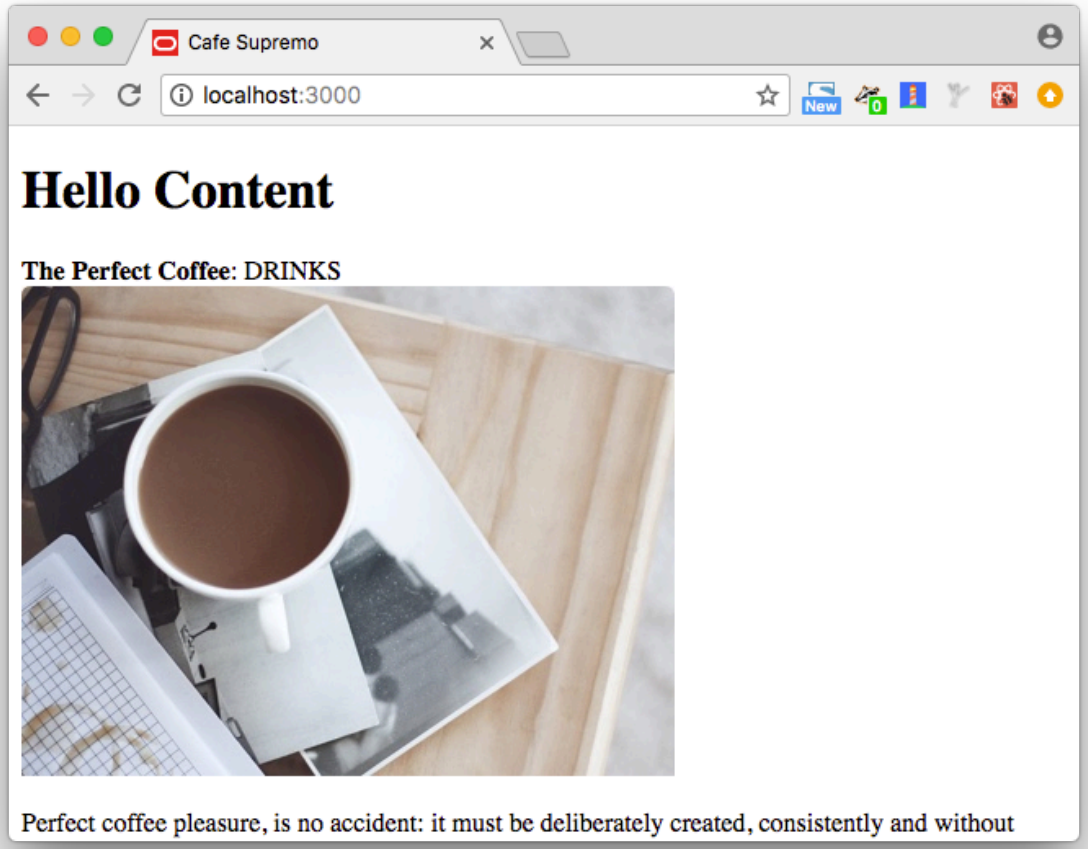
This module fetches a maximum of 500 items from CEC, ordered by upddateddate.

It writes a JSON file in the same directory as the module with the name content.json. That is the file we have used so far.

To fetch the content items from CEC, you'll need to issue

```
> node src/step5/fetchContentFromCEC.js
```

If you watch your browser window, you'll see it immediately update as the React development tooling detects that the referenced JSON file is updated.

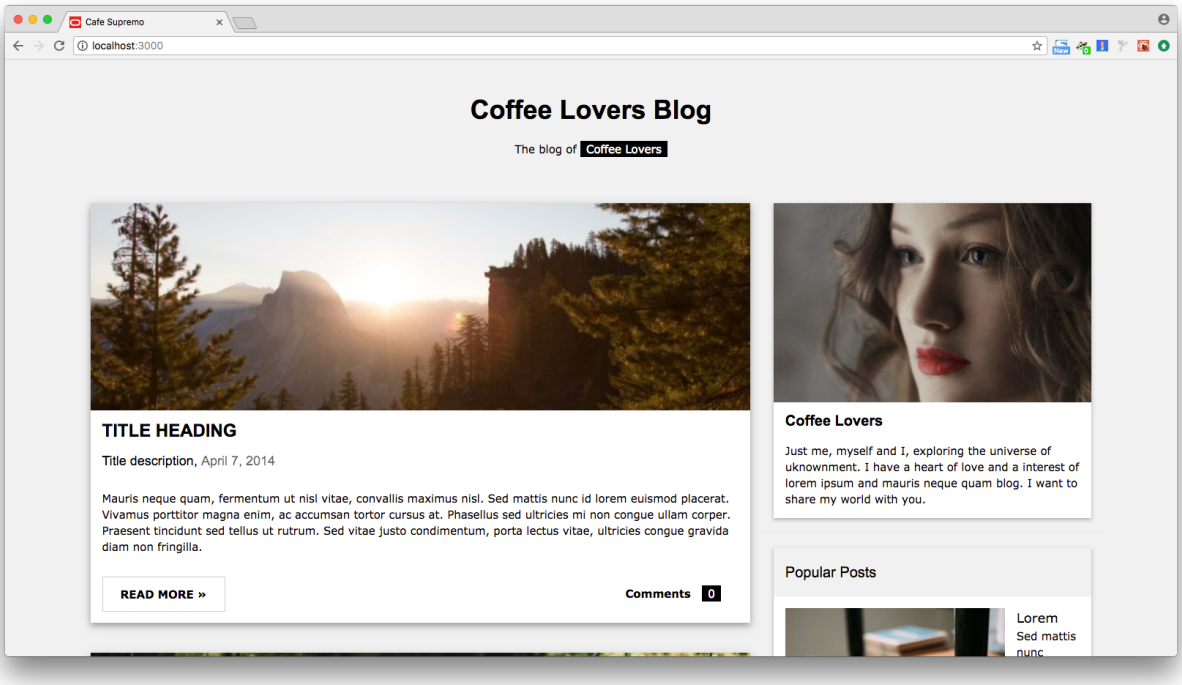


If you closely look at the fetch and Blog component code, you'll see that the images are retrieved of the CEC server at runtime. I'll leave it up the the reader to implement the code to retrieve the images also at build time and serve them of the static server like the index.html, css and JavaScript file.

10. Step 6: Putting it on a Page

The page we have built so far is pretty naked. It is a title with a list of blog articles. Next step is to put this list of Blog articles into a sample page that was downloaded from https://www.w3schools.com/w3css/w3css_templates.asp. The sample Blog page has been transformed for you to React by the course developer.

If you change App.jsx to step6, you'll see:



Layout.jsx has much more code:

```
import React, { Component } from 'react'
import './w3.css'
import './raleway.css'

import content from './content.json'

const allItems = content.ALL.data.items

const items = allItems.filter(item => item.type === 'Blog')

const fullItem = item => content[item.id].data

const toHref = link =>
  link.href.replace('/items/', '/digital-assets/') + '/default'


export default class Layout extends Component {
  render () {
    return (
      <div>
        <Body/>
```

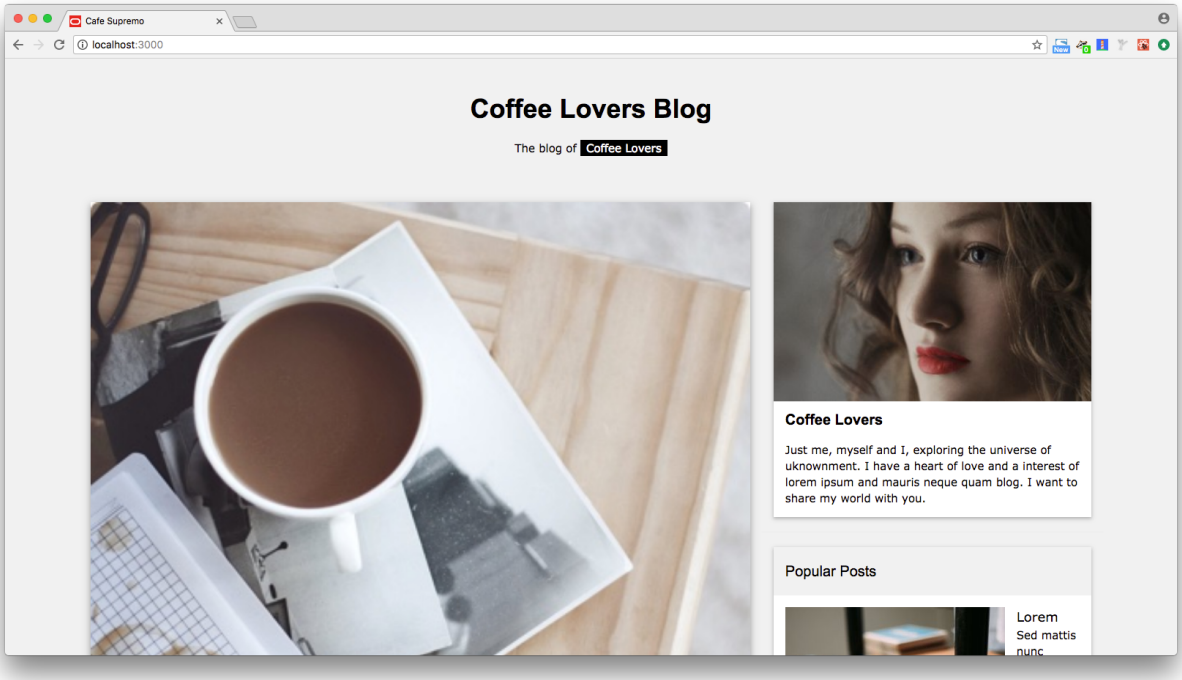
```
      </div>
    )
  }
}
const w100pct = { width: '100%' }
const w50px = { width: '50px' }

const Body = () =>(
  <div className='w3-light-grey'>
    <div className='w3-content' style={{maxWidth:'1400px'})}>
      <Header />
      <div className='w3-row'>
        <div className='w3-col l8 s12'>
          <Blogs />
        </div>
        <div className='w3-col l4'>
          <AboutCard />
          <hr />
          <PopularPosts />
        </div>
      </div>
      <br />
    </div>
    <Footer />
  </div>
)

const Header = () => (
  <header className='w3-container w3-center w3-padding-32'>
    <h1>
      <b>Coffee Lovers Blog</b>
    </h1>
    <p>
      The blog of <span className='w3-tag'>Coffee Lovers</span>
    </p>
  </header>
)

const Footer = () => (
  <footer className='w3-container w3-dark-grey w3-padding-32 w3-margin-top'>
    <p>
      Powered by{' '}
      <a href='https://www.w3schools.com/w3css/default.asp' target='_blank'>
        w3.css
      </a>
    </p>
  </footer>
)

const AboutCard = () => (
  // removed for readability
)


const PopularPosts = () => (
  // removed for readability
)

const Blogs = () => (
```

```
  // removed for readability

)

const NoItems = () => <div>No items to display</div>
const List = () => (
  <div>
    {items.map(fullItem).map((item, index) => <Blog key={index} item={item}
/>)}
  </div>
)
const  dateToMDY =(date) =>  new Date(date.value).toLocaleDateString('en-US',
{ year: 'numeric', month: 'long', day: 'numeric' })


const Blog = ({ item }) => {
  // removed for readability

}
```

The Layout has now more div blocks and components. The render method calls the Body component, and the Body call the Header, Footer, Blogs, AboutCard and PopularPost Components. Here you see React and it's components help in describing the structure of the page.

When you run the page you can see that the blog articles are the articles from the sample. You are challenged to implement the code to render the Blog content items from content.json.

The outcome should be like this:



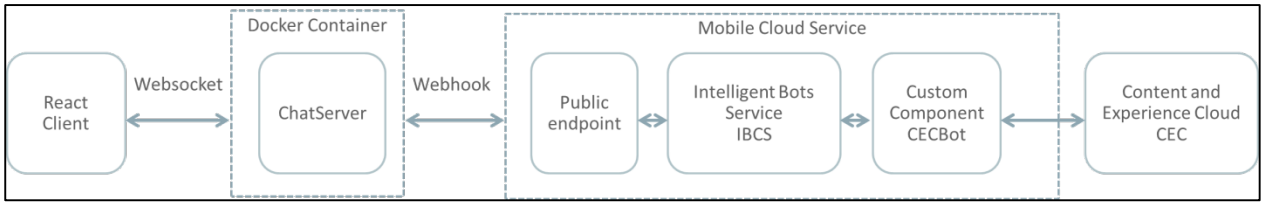
11. Step 7: Adding a conversational interface

In this step we will look at adding a new React component to our application. The ContentSearch React component encapsulates the code required to process and render searches made to a conversational Oracle Intelligent chat bot. This component can be used in a standalone manner or across your application.

The component we are going to add makes use of a chatbot custom component called “CECBot”. CECBot allows a client (web, mobile, etc) to query CEC content in a conversational manner. The chatbot uses AI framework to respond with the most relevant content.

The creation of Custom Component and IBCS are beyond the scope of this course. Further reading about Oracle Intelligent Chat Bots can be found here. A great article on the CECbot can be found here.

Access to the CECbot is done through a service called ChatServer. The ChatServer is a sample application that comes with IBCS. There are a range of sample applications which provide developers to quickly create clients which will talk to custom components running in IBCS. For this exercise we will be using our own client which makes use of an exposed websocket. Below is a high level diagram of the systems involved in this system.



For this exercise we will firstly deploy the ContentSearch skeleton component and build functionality in, in a few simple steps.

- Add/deploy the skeleton code, into Default. This adds the ContentSearch Component
- Add the search input field
- Add the code to handle the websocket connection and messages.
- View and examine the response JSON
- Add the code to render the content.

Now proceed with the actual steps:

1. Edit App.jsxlike in the previous examples to point to step7. Firstly, adding the import to your component.

```
import Layout from './step7/Layout'
```

Save the file and head back to your browser, refresh. You should see a yellow div. This is your SearchContent deployed.

2. First we would like to add an input field to our page. Open your editor the Component (in the step7/SearchContent.jsx). Copy in the code from the inputfield.txt.

Save the file, refresh your browser you should see the input field appear.

You will see an error message

This is because the input field you added has an initial which is coming from the value this.state.value.

We need to define this in the constructor code, after the super(props) call. Note: Class components should always call the base constructor with props. Further reading here.

```
this.state = {
  count: 0,
  message: '',
  messages: []
}
```

Save and refresh browser and you will see the new input field with default value “”.

3. Now let's add the code to handle the websocket connection, send and receive messages. To do this we are using a library called reconnecting-websocket. This library is a wrapper around a websocket which handles the reconnection complexities.

Firstly add the import in for this library

```
import ReconnectingWebSocket from 'reconnecting-websocket'
```

Note: this library has already been installed into the node modules directory.

In the SearchContent.jsx Component you will see the constructor method. Copy the code from the helper file in to the constructors where you see //copy here, under the state variables.

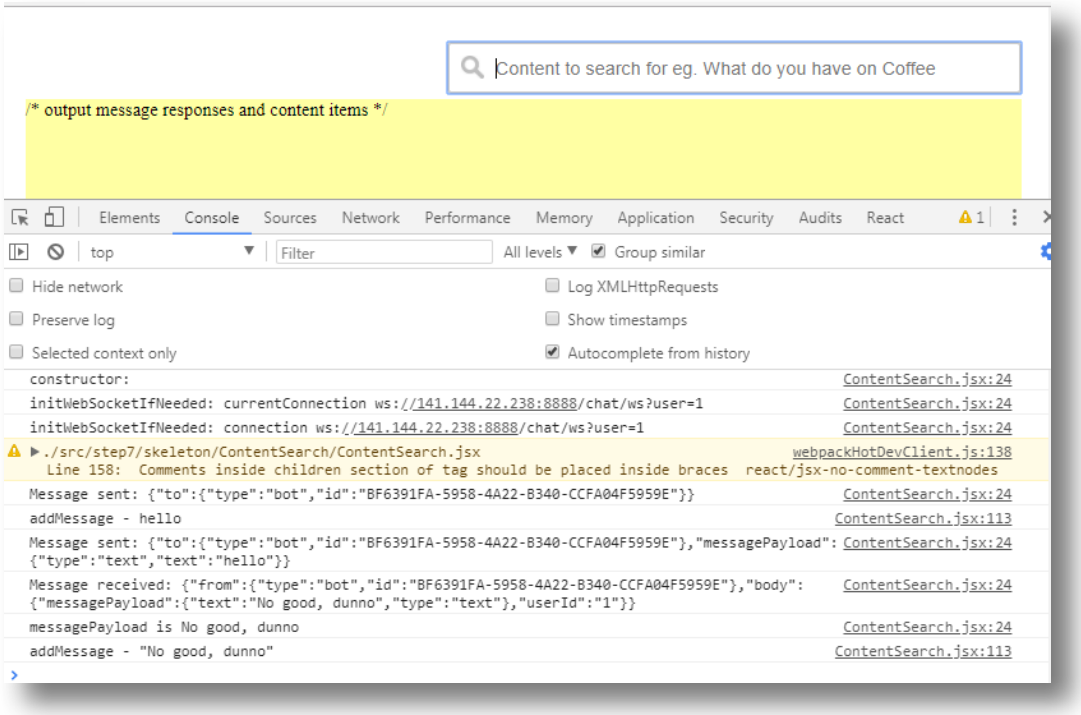
Review the code you added. There are some important methods : sendMessage, sendToBot, on-message, waitForSocketConnection

Again, save and refresh the page.

You will see no changes to the last time

d) Now we will view an examine what's going on in the Component. In your browser open the developer view. Enter the view console.

Enter "hello" in your search bar and view the output. Next, try "what do you have on coffee" and view the different json being returned. You should see 4 objects. You can use the React plugin if you are using Chrome.

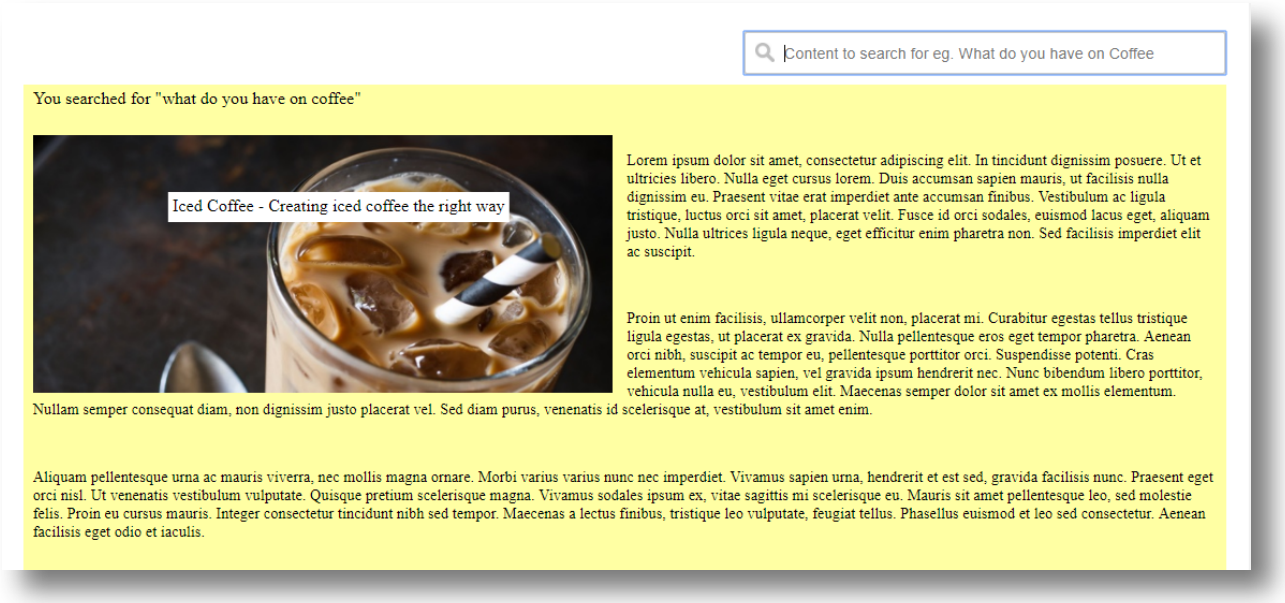


This is the output which is being returned from the CECbot.

Next we will add the html/jsx code to render the blog articles.

e) From the renderCode.txt file copy in the div into the

Save and refresh your page.



Bonus Exercise:

If you have finished this exercise, you can now add this new component inside the previous example 6.

- Inside step6/Layout.jsx add your import of ContentSearch Component
- In the same file, add your ContentSearch Component in the div around line 47

Save your file and you should see your new component inside the previous example.

12. Step 8: Building the Site with react-scripts

So far you have been using the development server from React to inspect your changes. Let's assume you are satisfied with your work and want to build and deploy the site.

Step 8 is there to guide you through the process.

The React tooling comes with a build command that takes your work and optimizes it for production delivery. It combines all the JavaScript files into one file, minimizes the JavaScript and CSS files etc. It also automatically versions the build file so they work well with upstream caches.

To build the site you have to issue:

```
> yarn build
```

As you can see in the console output, React gives a hint on how to inspect the build site.

```
> yarn global add serve
> serve -s build
```

Go with your browser to http://localhost:5000 and the browser window should look the same as your development browser window. Once you have inspected the page, please close the *serve* program, with Ctrl-C for instance.

Note: on windows if the serve command is not recognized please add this to your environment variables. For example "…;C:\Users\name\AppData\Local\Yarn\bin"

13. Step 9: Building the docker image

In order to run a docker container you'll need to build a docker image with your website first.

For this you need to docker toolchain installed locally.

Install Docker Community Edition

https://docs.docker.com/install/

After you have installed, check if you have access to docker from your command line:

```
> docker --version
```

This line should output the docker version number you are using. Please make sure you are using a recent version of docker. We have tested with Docker version 17.12.0-ce.

To build the image issue:

```
> ./build-docker-image.sh
```

This makes use of this Dockerfile

```
FROM nginx
EXPOSE 80
COPY build/ /usr/share/nginx/html
```

As you can see: nginx is used and the local build directory is copied to the root folder of the webserver.

For Windows 7, to run Docker, you need Hyper-V support which isn't included in Windows 7. You can see an official workaround Install Docker Box on Windows

Once running, navigate to the root directory of the cafe-supremo-react-front. You can then continue with above sh command

If you should close your window run Docker QuickStart Terminal

14. Step 10: Deploying on Docker

To run the Docker container, issue

```
> docker run --name cafesupremo-front --rm -d -p 9000:80 cafesupremo:1.0.0
```

Start a browser and navigate to http://localhost:9000. The same familiar page should come up.

Please stop the container with

```
> docker kill cafesupremo-front
```

For windows 7, to find the ip you can use:

```
> docker-machine env
```

15. Step 11: Deploying on Kubernetes

Like we have deployed to Docker, we can deploy also to Kubernetes. For this exercise we'll use Minikube, as VirtualBox that has Kubernetes installed on it, for us ready to use.

Install Minikube

Follow the guide from https://kubernetes.io/docs/tasks/tools/install-minikube/.

Install Kubectl

Follow the guide from https://kubernetes.io/docs/tasks/tools/install-kubectl/.

Check the installation with

```
> kubectl version
```

```
Client Version: version.Info{Major:"1", Minor:"9", GitVer-
sion:"v1.9.3", GitCommit:"d2835416544f298c919e2ead3be3d0864b52323b",
GitTreeState:"clean", BuildDate:"2018-02-09T21:50:44Z", GoVer-
sion:"go1.9.4", Compiler:"gc", Platform:"darwin/amd64"}
```

Create a Docker Hub Account

Go to https://hub.docker.com, and create an account if you don't have one.

Login to Docker

For reference: https://docs.docker.com/docker-cloud/builds/push-images/

```
> export DOCKER_ID_USER="username"
> docker login
```

Deployment steps

After all the preparation work it is now time to

Next tag and push the build image to Docker Hub, for Minihube to pull from when it deploys the containers.

```
> docker tag cafesupremo:1.0.0 $DOCKER_ID_USER/cafesupremo:1.0.0
> docker push $DOCKER_ID_USER/cafesupremo:1.0.0
```

You can verify this is successful by looking under Repositories in your hub.docker.com

For reference: https://kubernetes.io/docs/getting-started-guides/minikube/ .

On windows, restart the DockerBox to pick up the new minikube path

If Minikube is not running, please start it with

```
> minikube start
```

You can check if you can connect with kubectl.

```
> kubectl version
```

Client Version: version.Info{Major:"1", Minor:"9", GitVersion:"v1.9.3", GitCommit:"d2835416544f298c919e2ead3be3d0864b52323b", GitTreeState:"clean", BuildDate:"2018-02-09T21:50:44Z", GoVersion:"go1.9.4", Compiler:"gc", Platform:"darwin/amd64"}

Server Version: version.Info{Major:"1", Minor:"8", GitVersion:"v1.8.0", GitCommit:"0b9efaeb34a2fc51ff8e4d34ad9bc6375459c4a4", GitTreeState:"clean", BuildDate:"2017-11-29T22:43:34Z", GoVersion:"go1.9.1", Compiler:"gc", Platform:"linux/amd64"}

You should see that a Server Version is returned.

Next is to check the local ip of your Kubernetes cluster

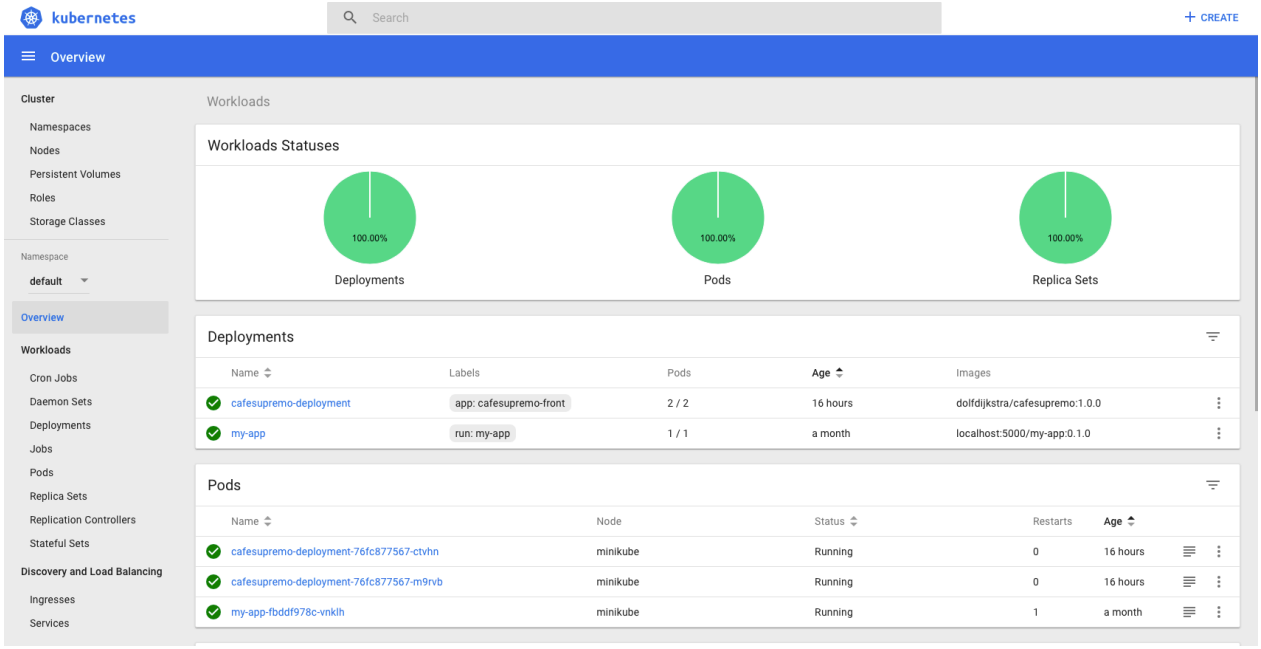
```
> minikube ip
```

192.168.99.100

To launch the Kubernetes Dashboard UI, issue

```
> minikube dashboard
```

A browser will start (http://192.168.99.100:30000/#!/overview?namespace=default).



To have Kubernetes download the image from DockerHub it needs to know the username and password for DockerHub as a Kubernetes secret . You can use kubectl to set the secret.

```
$ kubectl create secret docker-registry dockerhubreg --docker-
username=$DOCKER_ID_USER --docker-password="password" --docker-
email="your email from docker hub"
```

After the secret is set you can create the Kubernetes Pods and Deployments. The next two commands create the Kubernetes Pods and LoadBalancer:

```
$ kubectl run supremo-front --image=$DOCKER_ID_USER/cafesu-
premo:1.0.0 --replicas=1 --port=80
$ kubectl expose deployment supremo-front --target-port=80 --
type=LoadBalancer
```

After these two step you can check the Kubernetes Dashboard and find the Deployements and Pods. You can also use kubectl to inspect the cluster:

```
kubectl get services
```

```
NAME             TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
kubernetes       ClusterIP      10.96.0.1      <none>        443/TCP          2d
supremo-front    LoadBalancer   10.102.89.8    <pending>     80:31740/TCP     10m
```

```
$ kubectl get pods
```

```
NAME                              READY     STATUS    RESTARTS   AGE
supremo-front-6b6c6fb59-6vw95     1/1       Running   0          11m
```

```
$ kubectl get deployments
```

```
NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
supremo-front    1         1         1            1           11m
```

To see the deployed website, you need the external IP of the LoadBalancer. Minikube does not report this as part of the *kubectl get services* output. The minikube command can reveal it:

```
$ minikube service supremo-front --url
```

The response might be something like *http://192.168.99.100:31740*. Open a browser to this address and you should see the deployed website hosted on the Kubernetes cluster.

16. Step 12: Towards Café Supremo

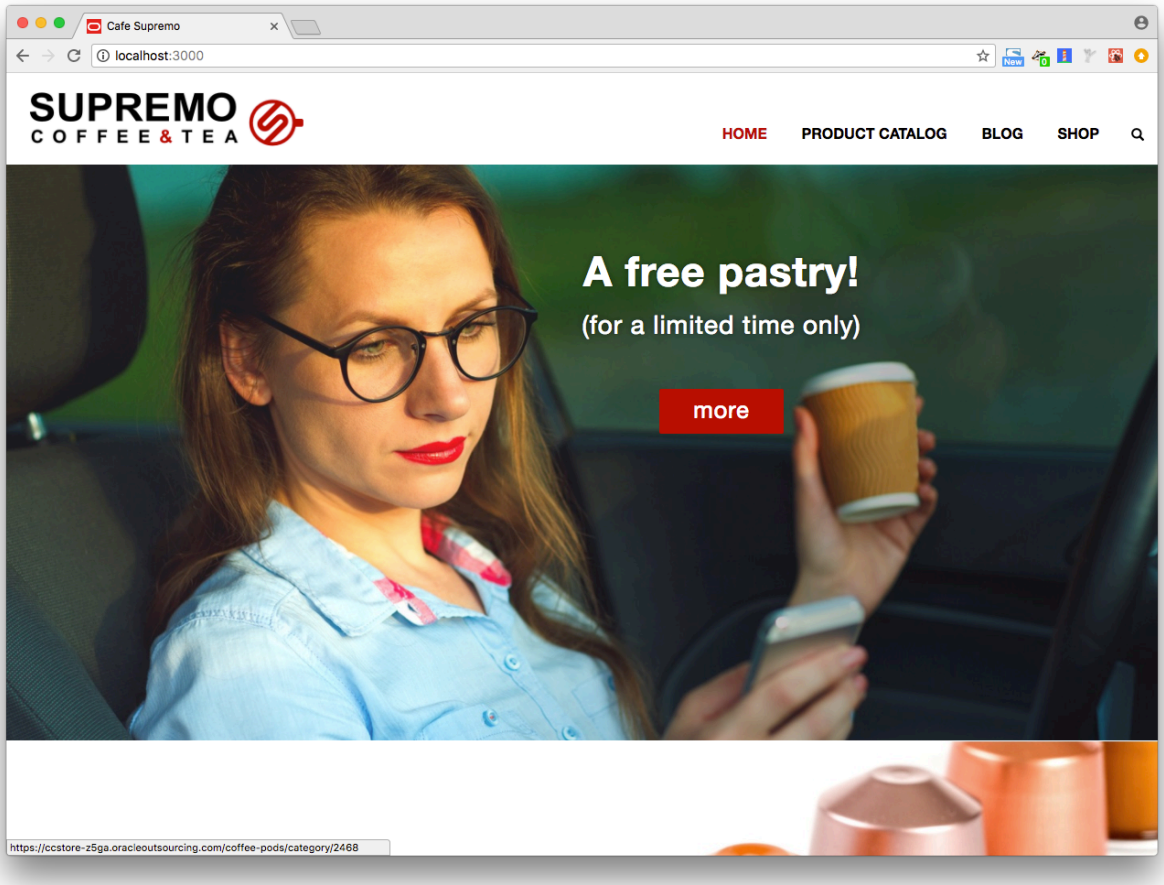
As the final step in the course you'll work towards the Café Supremo Homepage. The instructor has provided a not-yet-full-functional Homepage of the Café Supremo site. The task is to review the code and understand with what you have learned so far where are the pieced ended up

- How is the page decomposed in Components?
- How is content read, how can this be updated?
- How can you deploy this on Docker and Kubernetes as a static site?
- How is the content of the Slots (as configured by the editorial team) being used? Where is the configuration read from?
- How can you switch between Layouts? How are themes implemented?

To get the page visible, you have to change the React application in a similar way as you have done so far. Instead of changing App.jsx, we intercept earlier in the rendering process. Change index.js:

```
import App from './CafeSupremo.jsx'
```

After this change, you page should reload again, and the page from Café Supremo shows:



With what you have learned you should be able to understand structure, build and deploy to Kubernetes.

If you deploy to Kubernetes, does the version of the site appear? If not, which version of the docker image are you building and deploying.

Oracle Content&Experience Cloud Workshop
August, 2018
Author: Dolf Dijkstra, Emma Thomas

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com