Name: A.P.Raja Lakshmi

College code: 9530

College name: st.Mother Theresa Engineering College

Team id:

Naan Mudhalvan id:au953021104037

Project Name: Image Recognition with IBM cloud

Problem Solution:

Image recognition with IBM cloud visual recognition

Step 1: Creating the Node.js starter app

To start developing your application locally, begin by logging in to the IBM Cloud Platform directly from the command line, as shown in the example. You can specify optional parameters, such as your organization with option -o and the space with option -s. If you're using a federated account use –sso.

Ibm cloud login

After logging in, when you are asked if you want to install any extensions, you may see an announcement regarding the Cloud Foundry plugin. Type the command as shown in order to download and install the CLI extension used in this tutorial.

Ibmcloud cf install

When you log in you might be asked to choose a region. For this exercise, select us-south as the region, as that same option is used to build a CD Toolchain later in this tutorial.

Next, set the endpoint (if it isn't set already). Other endpoints are possible, and might be preferable for production use. For now, use the code as shown, if appropriate for your account.

Ibmcloud api cloud.ibm.com

Target the Cloud Foundry (cf) aspect of IBM Cloud Platform by using the target command and the –cf option. The cf API is no longer embedded within the CLI Developer Tools and will have to be downloaded separately.

```
Ibmcloud target –cf
```

And now, time to create a web application. The dev space is a default option for your organization, but you might prefer to create others for isolating different efforts. For example, keeping 'finance' separate from 'development'.

```
Ibmcloud dev create
```

With that command, you're asked a series of questions. You can go back at many points in the process, so if you feel lost you can start over by deleting the existing directory and creating a new directory. Even when you create your application on the command line, you'll still see the results in your IBM Cloud console.

Note the option for creating a 'Web App'. That's the one you want.

```
Select an application type:

 1. Backend Service / Web App
 2. Mobile App
 0. Exit

? Enter selection number:> 1
```

A number of options are provided, but we want 'Node'. Type '4' and press enter.

```
Select a language:

 1. Go
```

```
2. Java – MicroProfile / Java EE

3. Java – Spring

4. Node

5. Python – Django

6. Python – Flask

7. Swift
--------------------------------------------------------------
0. Return to the previous selection

==============================================================
? Enter selection number:> 4
```

After you make your selection for the programming language and framework, the next selection will have so many options, it might scroll past your wanted service. As you can see in the example, we wish to use a simple Node.js Web App with Express.js. Type ‘3’ and press enter.

```
Select a Starter Kit:

APPSERVICE
--------------------------------------------------------------

1.  Node-RED – A starter to run the Node-RED open-source project on

  IBM Cloud.


2.  Node.js + Cloudant – A web application with Node.js and Cloudant


3.  Node.js Express App – Start building your next Node.js Express

  App on IBM Cloud.


WATSON
--------------------------------------------------------------

4.  Natural Language Understanding Node.js App – Use Watson Natural
```

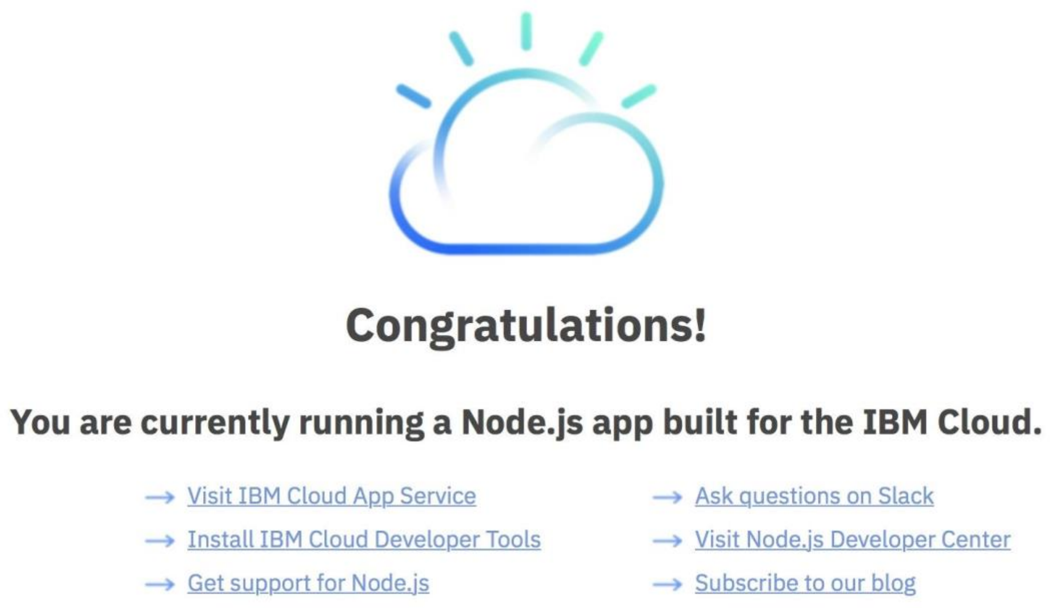
```
   Language Understanding to analyze text to help you understand its
   Concepts, entities, keywords, sentiment, and more.

5. Speech to Text Node.js App – React app using the Watson Speech to
   Text service to transform voice audio into written text.

6. Text to Speech Node.js App – React app using the Watson Text to
   Speech service to transform text into audio.

7. Visual Recognition Node.js App – React app using the Watson
   Visual Recognition service to analyze images for scenes, objects, text,
   And other subjects.
------------------------------------------------------------------------
0. Return to the previous selection
========================================================================
? Enter selection number:> 3
```

The hardest option for developers everywhere is still required: naming your app. Follow the example and type webapplication, then press enter.

```
? Enter a name for your application> webapplication
```

Later, you can add as many services, like data stores or compute functions, as needed or wanted through the web console. However, type ‘n’ for no when asked if you want to add services now. Also, if you haven’t already set a resource group, you may be prompted at this time. You may skip this by typing ‘n’ at this prompt.

```
Using the resource group Default (default) of your account

? Do you want to select a service to add to this application? [Y/n]> n
```



Congratulations!

You are currently running a Node.js app built for the IBM Cloud.

→ Visit IBM Cloud App Service
→ Install IBM Cloud Developer Tools
→ Get support for Node.js
→ Ask questions on Slack
→ Visit Node.js Developer Center
→ Subscribe to our blog

Step 2: Creating the Web Gallery app

Let's recall the prerequisites that you needed for developing a Node.js app on IBM Cloud Platform. You already created your IBM Cloud Platform account as well as installed the Developer Tools, which installed Docker. Then, you installed Node.js. The last item listed as a prerequisite for this tutorial was Git, which we dive into now.

We're going to start the specifics of working on the image gallery in Node.js. For now, we use GitHub Desktop for this scenario, but you might also use the Git command-line client to complete the same tasks. To get started, let's clone a starter template for your new web application.

Follow this process:

Download the sample here: download. Download the template for your app to your local development environment using your browser. Rather than cloning the sample app from IBM Cloud Platform, use the command in the example to obtain the starter template for the IBM Cloud Object Storage Web Gallery app. After cloning the repo you will find the starter app in the COS-WebGalleryStart directory. Open a Git CMD window and change to a directory where you want to clone Github repo. Once there, use the command shown in the first example of this tutorial to start adding your new files.

```
Curl images/image-gallery-tutorial.zip -o image-gallery-tutorial.zip
```

Run the app locally. Open your terminal and change your working directory to the COS-WebGalleryStart directory. Note the Node.js dependencies that are listed in the package.json file. Download them into place by using the command shown next.

```
Npm install
```

Run the app by using the command shown.

```
Npm start
```

Open a browser and view your app on the address and port that is output to the console, http://localhost:3000.

```
Applications:
   -   Path: .
  Memory: 256M
  Instances: 1
  Domain: us-south.cf.appdomain.cloud
  Name: webapplication
  Host: webapplication
  Disk_quota: 1024M
  Random-route: true
```

Deploy the app to IBM Cloud Platform.

To get the starter app with your changes to IBM Cloud Platform, deploy it using the Developer Tools by repeating the same steps that we performed earlier.

If you haven't already, or if you restarted or logged out, log in to IBM Cloud Platform by using the login command.

```
Ibmcloud login
```

Set the API Endpoint for your region by using the api command.

```
Ibmcloud api cloud.ibm.com
```

Target the Cloud Foundry aspect of IBM Cloud Platform by using the target command and the –cf option.

```
Ibmcloud target –cf
```

Build the app for delivery that application with the build command (as in the example).

```
Ibmcloud dev build
```

Let's go ahead and test the application locally. This allows you to run the same code locally with the run command.

```
Ibmcloud dev run
```

Deploy the app to IBM Cloud Platform with the deploy command.

```
Ibmcloud dev deploy
```

The code shows the sequence of commands that are used in this example to build, test, and deploy the initial web application.

Ibmcloud login –sso

Ibmcloud api cloud.ibm.com

Ibmcloud target –cf

Ibmcloud dev enable

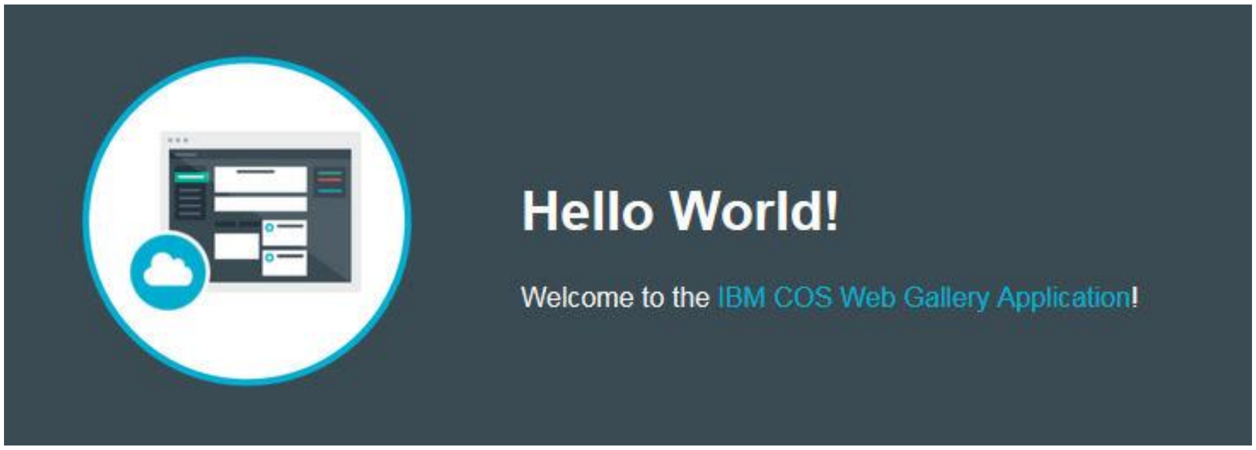
Ibmcloud dev build

Ibmcloud dev run

Ibmcloud dev deploy

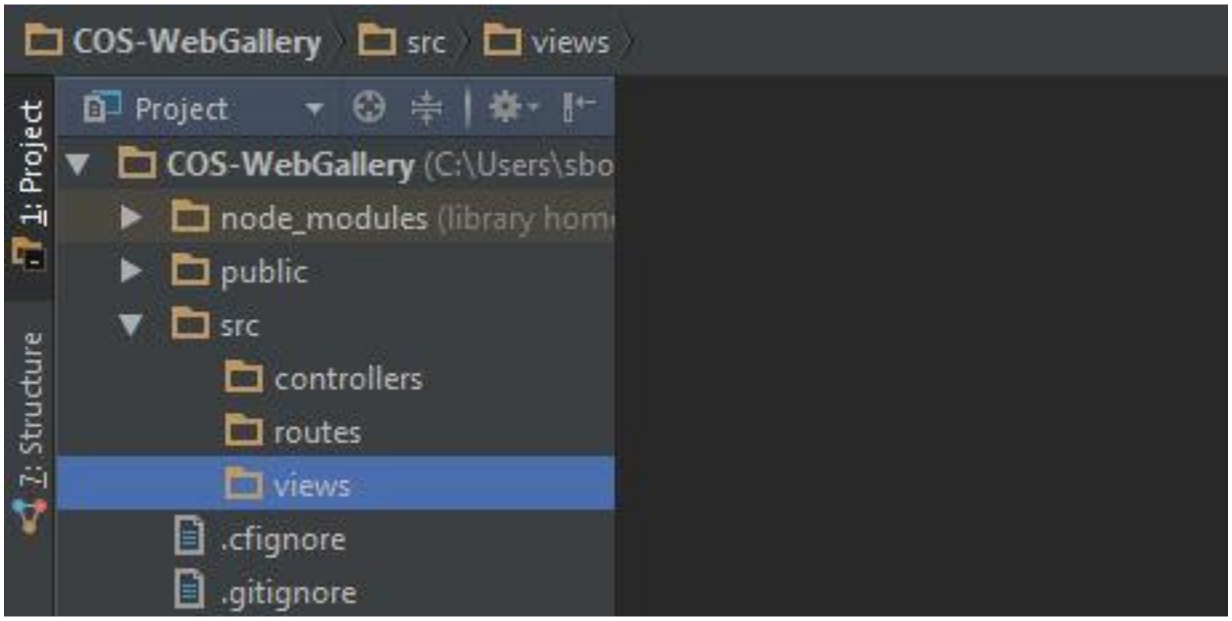
When the process finishes, the IBM Cloud Platform reports that the app was uploaded, successfully deployed, and started. If you're also logged in to the IBM Cloud Platform web console, you're notified there also of the status of your app. But, most importantly, you can verify that the app was deployed by visiting the app URL reported by IBM Cloud Platform with a browser, or from the web console by clicking View App button.

Test the app. The visible change from the default app template that was deployed at creation to the starter app shown in the following proved that deploying the app to IBM Cloud Platform was successful.



STep 3: Customize your Node.js IBM Cloud Object Storage Image Gallery web Application

Because this example uses an MVC architecture, adjusting the directory structure within your project to reflect this architecture is a convenience as well as a best practice. The directory structure has a views directory to contain the EJS view templates, a routes directory to contain the express routes, and a controllers directory as the place to put the controller logic. Place these items under a parent source directory



Designing the app

These are the two main tasks that a user should be able to do with the simple image gallery web application:

- Upload images from a web browser to the Object Storage bucket.
- View the images in the Object Storage bucket in a web browser.

The next steps focus on how to accomplish these two demonstration functions rather than building a fully developed, production-grade app. Deploying this tutorial and leaving it exposed and running means that anyone who finds the app can perform the same actions: upload files to your IBM Cloud Object Storage bucket and view any JPEG images already there in their browser.

Developing the app

In the package.json file, inside the scripts object, you see how “start” is defined. This file is what IBM Cloud Platform uses to tell node to run app.js each time the app starts. Also, use it when testing the app locally. Look at the main application file, which is called app.js. This is the code that we have told Node.js to process first when you start your app with the npm start command (or nodemon).

```
{
   "scripts": {
     "start": "node app.js"
   }
}
```

Our app.js file uses node to load modules that are needed to get started. The Express framework creates the app as a singleton simply called app. The example ends (leaving out most of the code for now) telling the app to listen on the port that is assigned and an environment property, or 3000 by default. When successfully starting at the start, it prints a message with the server URL to the console.

```
Var express = require('express');
Var cfenv = require('cfenv');
Var bodyParser = require('body-parser');
Var app = express();
//…

// start server on the specified port and binding host
Var port = process.env.PORT || 3000;
App.listen(port, function() {
   Console.log("To view your app, open this link in your browser: http://localhost:" + port);
});
//…
```

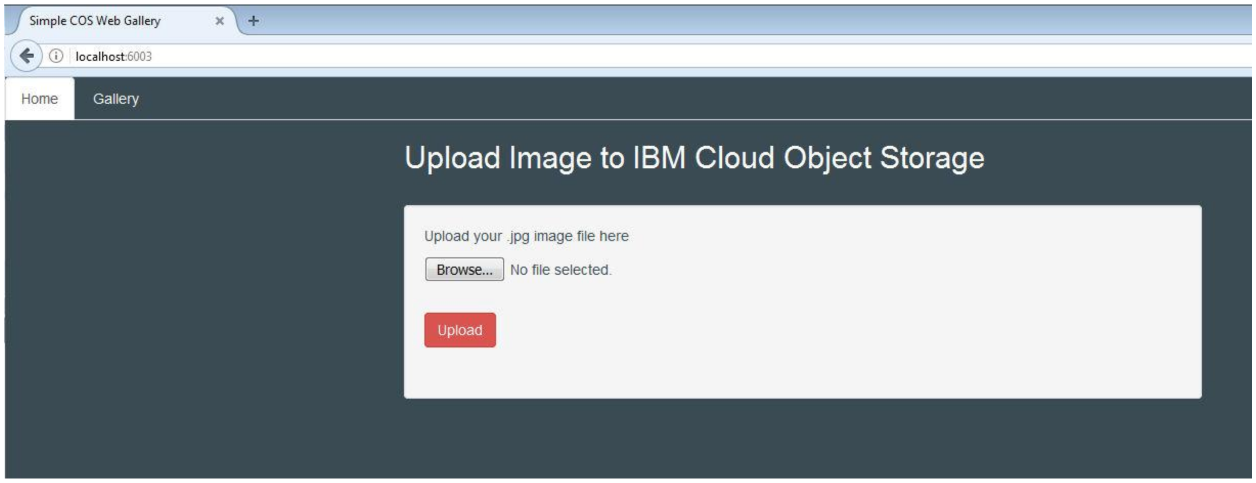
Show more

Let's see how to define a path and views. The first line of code tells the Express framework to use the public directory to serve our static files, which include any static images and stylesheets we use. The lines that follow tell the app where to find the templates for our views in the src/views directory, and set our view engine to be EJS. In addition, the framework uses the body-parser middleware to expose incoming request data to the app as JSON. In the closing lines of the example, the express app responds to all incoming GET requests to our app URL by rendering the index.ejs view template.

```
//…
// serve the files out of ./public as our main files
App.use(express.static('public'));
App.set('views', './src/views');
App.set('view engine', 'ejs');
App.use(bodyParser.json());

Var title = 'COS Image Gallery Web Application';
// Serve index.ejs
App.get('/', function (req, res) {
  Res.render('index', {status: '', title: title});
});

//…
```

Show more

The following figure shows what the index view template when rendered and sent to the browser. If you are using ,nodemon you might have noticed that your browser refreshed when you saved your changes.



Upload Image to IBM Cloud Object Storage

Upload your .jpg image file here

No file selected.

Consider these two specifications for our app:

We set our form method to POST and the form-data encoding type as multipart/form-data on line 24. For the form action, we send the data from our form to the app to the app route “/”. Later, we do extra work in our router logic to handle POST requests to that route.

We want to display feedback about the status of the attempted file upload to the user. This feedback is passed to our view in a variable named “status”, and is displayed after the upload form.

```
<!DOCTYPE html>

<html>


<head>

  <%- include(‘head-inc’); %>

</head>


<body>

<ul class=”nav nav-tabs”>

  <li role=”presentation” class=”active”><a href=”/”>Home</a></li>

  <li role=”presentation”><a href=”/gallery”>Gallery</a></li>

</ul>

<div class=”container”>

  <h2>Upload Image to IBM Cloud Object Storage</h2>

  <div class=”row”>

    <div class=”col-md-12”>

      <div class=”container” style=”margin-top: 20px;”>

        <div class=”row”>


          <div class=”col-lg-8 col-md-8 well”>


            <p class=”wellText”>Upload your JPG image file here</p>
```

```
                    <form method=”post” enctype=”multipart/form-data” action=”/”>
                        <p><input class=”wellText” type=”file” size=”100px” name=”img-file” /></p>
                        <br/>
                        <p><input class=”btn btn-danger” type=”submit” value=”Upload” /></p>
                    </form>

                    <br/>
                    <span class=”notice”><%=status%></span>
                </div>
            </div>
        </div>
    </div>
  </div>
</div>
</body>

</html>
```

Show more

Let’s take a moment to return to app.js. The example sets up Express routes to handle extra requests that are made to our app. The code for these routing methods are in two files under the ./src/routes directory in your project:

imageUploadRoutes.js: This file handles what happens when the user selects an image and clicks Upload.

galleryRoutes.js: This file handles requests when the user clicks the Gallery tab to request the imageGallery view.

```
//…
```

```
Var imageUploadRoutes = require('./src/routes/imageUploadRoutes')(title);

Var galleryRouter = require('./src/routes/galleryRoutes')(title);


App.use('/gallery', galleryRouter);

App.use('/', imageUploadRoutes);


//…
```

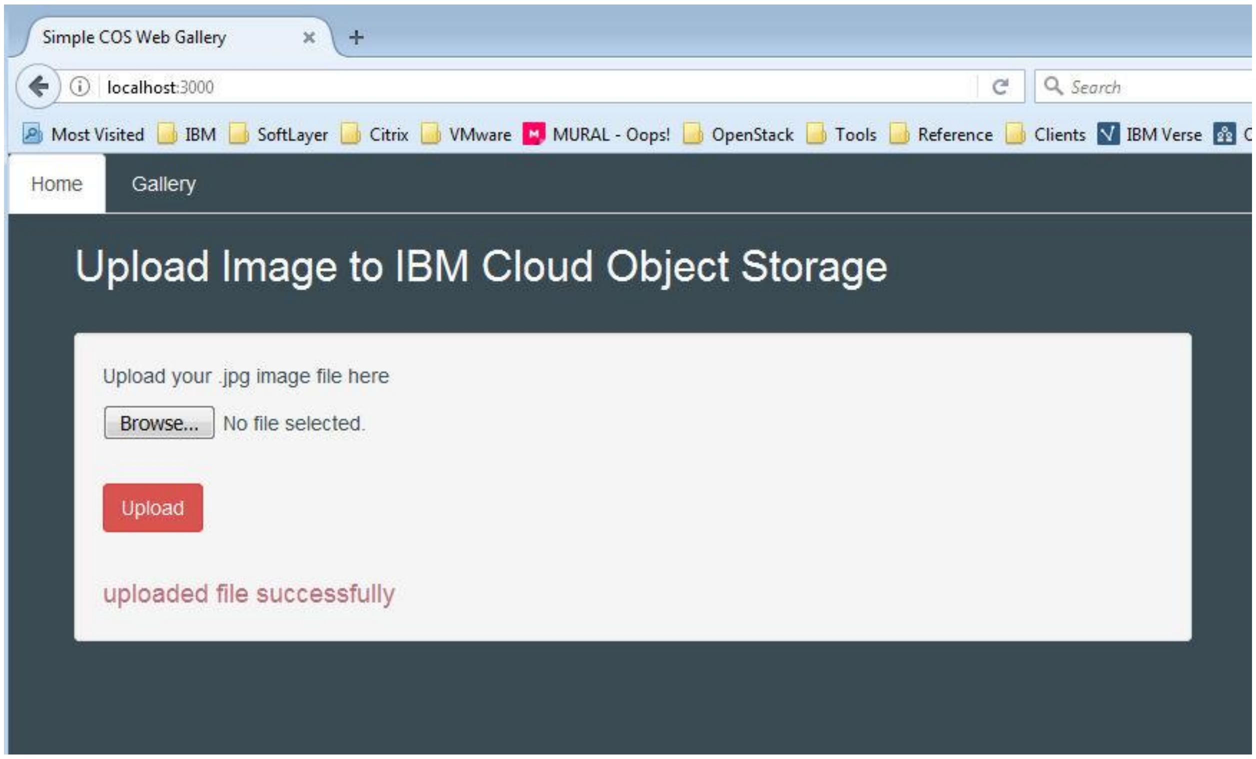
See the code from imageUploadRoutes.js. We must create an instance of a new express router and name it imageUploadRoutes at the start. Later, we create a function that returns imageUploadRoutes, and assign it to a variable called router. When completed, the function must be exported as a module to make it accessible to the framework and our main code in app.js. Separating our routing logic from the upload logic requires a controller file named galleryController.js. Because that logic is dedicated to processing the incoming request and providing the appropriate response, we put that logic in that function and save it in the ./src/controllers directory.

The instance of the Router from the express framework is where our imageUploadRoutes is designed to route requests for the root app route ("/") when the HTTP POST method is used. Inside the post method of our imageUploadRoutes, we use middleware from the multer and multer-s3 modules that is exposed by the galleryController as upload. The middleware takes the data and file from our upload form POST, processes it, and runs a callback function. In the callback function we check that we get an HTTP status code of 200, and that we had at least one file in our request object to upload. Based on those conditions, we set the feedback in our status variable and render the index view template with the new status.

```
Var express = require('express');

Var imageUploadRoutes = express.Router();

Var status = '';


Var router = function(title) {


  Var galleryController =

    Require('../controllers/galleryController')(title);
```

```
   imageUploadRoutes.route('/')
          .post(
                   galleryController.upload.array('img-file', 1), function (req, res, next) {
           if (res.statusCode === 200 && req.files.length > 0) {
              status = 'uploaded file successfully';
           }
           Else {
              Status = 'upload failed';
           }
           Res.render('index', {status: status, title: title});
         });

   Return imageUploadRoutes;
};

Module.exports = router;
```

In comparison, the code for the galleryRouter is a model of simplicity. We follow the same pattern that we did with imageUploadRouter and require galleryController on the first line of the function, then set up our route. The main difference is we are routing HTTP GET requests rather than POST, and sending all the output in the response from getGalleryImages, which is exposed by the galleryController on the last line of the example.



Upload Image to IBM Cloud Object Storage

Upload your .jpg image file here

No file selected.

uploaded file successfully

Image retrieval and display

Remember back in app.js, the line of code app.use('/gallery', galleryRouter); tells the express framework to use that router when the /gallery route is requested. That router, if you recall, uses galleryController.js , we define the getGalleryImages function, the signature of which we have seen previously. Using the same s3 object that we set up for our image upload function, we call the function that is named listObjectsV2. This function returns the index data defining each of the objects in our bucket. To display images within HTML, we need an image URL for each JPEG image in our web-images bucket to display in our view template. The closure with the data object returned by listObjectsV2 contains metadata about each object in our bucket.

The code loops through the bucketContents searching for any object key ending in ".jpg," and create a parameter to pass to the S3 getSignedUrl function. This function returns a signed URL for any object when we provide the object's bucket name and key. In the callback function, we save each URL in an array, and pass it to the HTTP Server response method res.render as the value to a property named imageUrls.

```
//...
```

```
  Var getGalleryImages = function (req, res) {
    Var params = {Bucket: myBucket};
    Var imageUrlList = [];

    S3.listObjectsV2(params, function (err, data) {
      If (data) {
        Var bucketContents = data.Contents;
        For (var I = 0; I < bucketContents.length; i++) {
          If (bucketContents[i].Key.search(/.jpg/i) > -1) {
            Var urlParams = {Bucket: myBucket, Key: bucketContents[i].Key};
            S3.getSignedUrl(‘getObject’, urlParams, function (err, url) {
              imageUrlList.push(url);
            });
          }
        }
      }
      Res.render(‘galleryView’, {
        Title: title,
        imageUrls: imageUrlList
      });
    });
  };

//…
```

Show more

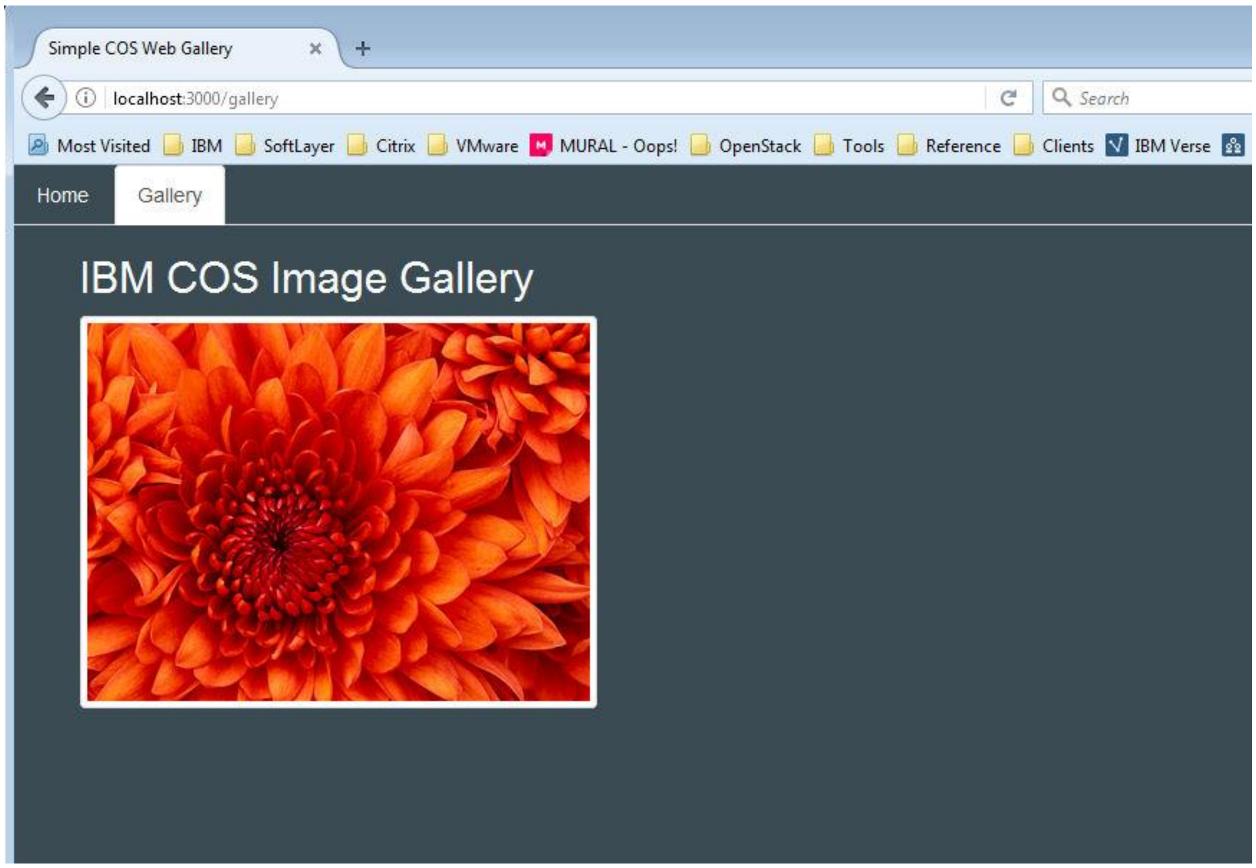
The last code example shows the body of the galleryView template with the code that is needed to display our images. We get the imageUrls array from the res.render() method and iterate over a pair of nested <div>…</div> tags. Each sends a GET request for the image when the /gallery route is requested.

```
<!DOCTYPE html>
<html>

<head>
  <%- include(‘head-inc’); %>
</head>

<body>
  <ul class=”nav nav-tabs”>
    <li role=”presentation”><a href=”/”>Home</a></li>
    <li role=”presentation” class=”active”><a href=”/gallery”>Gallery</a></li>
  </ul>
  <div class=”container”>
    <h2>IBM COS Image Gallery</h2>

    <div class=”row”>
      <% for (var i=0; I < imageUrls.length; i++) { %>
        <div class=”col-md-4”>
          <div class=”thumbnail”>
              <img src=”<%=imageUrls[i]%>” alt=”Lights” style=”width:100%”>
          </div>
        </div>
      <% } %>
    </div>
```

```
   </div>
</body>


</html>
```

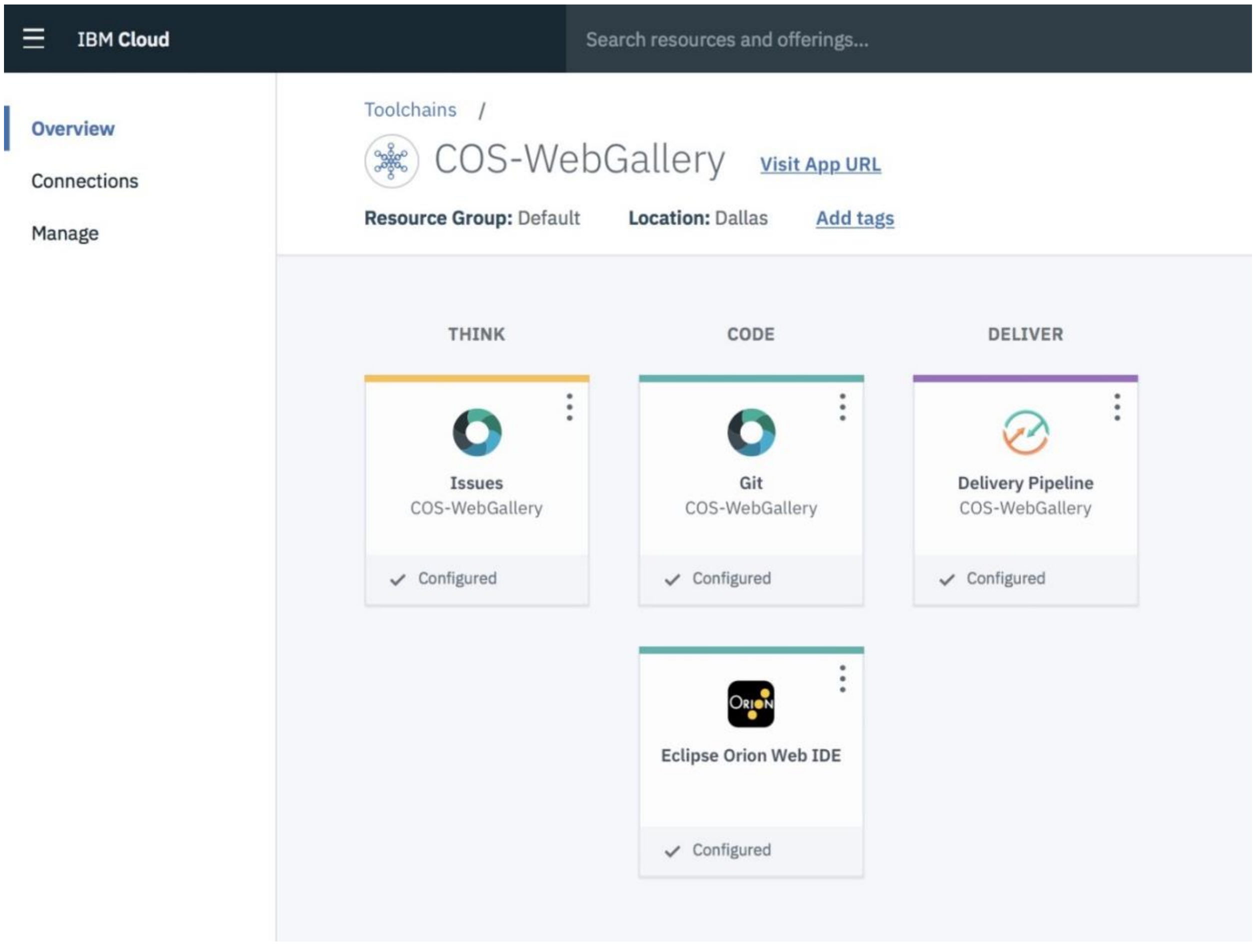
Show more

We test the app locally from http://localhost:3000/gallery and see our image.



Committing to Git

Now that the basic features of the app are working, we commit our code to our local repo, and then push it to GitHub. Using GitHub Desktop, we click Changes (see Figure 11), type a summary of the changes in the Summary field, and then click Commit to Local-dev.



When we click sync, our commit is sent to the remote local-dev branch. This action starts the Build and Deploy Stages in our Delivery Pipeline.

We went from beginning to end and built a basic web application image gallery by using the IBM Cloud Platform. Each of the concepts we've covered in this basic introduction can be explored further at IBM Cloud Object Storage.