Course Exercises Guide

Developing Node.js Applications on IBM Cloud

IBM MEA Skills Academy
Course code SANOD ERC 3.0

Ahmed Azraq
Mohamed Ahmed
Mohamed Ewies
Norhan Khaled
David Provest
Acknowledgments to members of **IBM Academy of Technology**

IBM Training

December 2019 edition

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Contents

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

Express®	IBM Cloud™	IBM Watson®
THINK®	Watson™	

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Exercises description

This course includes the following exercises

- Exercise 1. Developing a Hello World Node.js app on IBM Cloud
  In this exercise, you create a Node.js Cloud Foundry application on IBM Cloud. You will develop a Node.js-based server application (by using the Eclipse Orion Web IDE) that responds to web browser requests.
- Exercise 2. Understanding asynchronous callback
  This exercise shows how to use callback functions to call an external service. This exercise uses the IBM Watson Language Translator service in IBM Cloud. You create a Node.js module that contains the logic for these calls.
- Exercise 3. Creating your first Express application
  In this exercise, you create an application that uses the Express framework and the IBM Watson Natural Language Understanding service to extract the author name from articles that are published on the web. You provide the web address (URL) of the article to the application, and it outputs the name of the author (or multiple names if the article has multiple authors).
- Exercise 4. Building a rich UI application by using React and ES8
  This exercise guides you through building an interactive and rich client-side application by using React. You also explore the async/await feature of ECMAScript 2017, which is commonly known as ES8, and some features of ES8 through a server-side application by using Node.js

In the exercise instructions, you can check off the line before each step as you complete it to track your progress.

Most exercises include required sections, which should always be completed. It might be necessary to complete these sections before you can start later exercises.

Important

You must complete *Exercise 0. Setting up your hands-on environment* which is part of Module I (self-study) before starting the exercises in this course.

Information

Students in this course use an IBM Cloud Lite account to perform the exercises. This account will never expire, therefore students can continue working on the IBM Cloud environment after the class.

Note

It is recommended to use the Firefox browser to perform the exercises in this course.

Hint

Before performing the exercises, check the status of the services in the IBM Cloud Status page at https://cloud.ibm.com/status

Exercise 1. Developing a Hello World Node.js app on IBM Cloud

Estimated time

01:30

Overview

IBM SDK for Node.js provides a stand-alone JavaScript runtime and a server-side JavaScript solution for IBM platforms. It provides a high-performance, highly scalable, and event-driven environment with non-blocking I/O that is programmed with the familiar JavaScript programming language. IBM SDK for Node.js is based on the Node.js open source project.

To develop Node.js code, you use Eclipse Orion Web IDE, which is a web-based, integrated development environment (IDE) where you can create, edit, run, debug, and perform source-control tasks. You can seamlessly move from editing to running, submitting, and deploying.

In this exercise, you create a Node.js Cloud Foundry application on IBM Cloud. You do not need to install Node.js on your machine. You develop a Node.js-based server application (by using the Eclipse Orion Web IDE) that responds to web browser requests.

Objectives

Web developers write JavaScript applications to add interactivity to client-side web applications. As an interpreted scripting language, developers do not need to use compilers to write applications in JavaScript. The syntax of the programming language is simple enough for web developers with little programming experience to write simple applications.

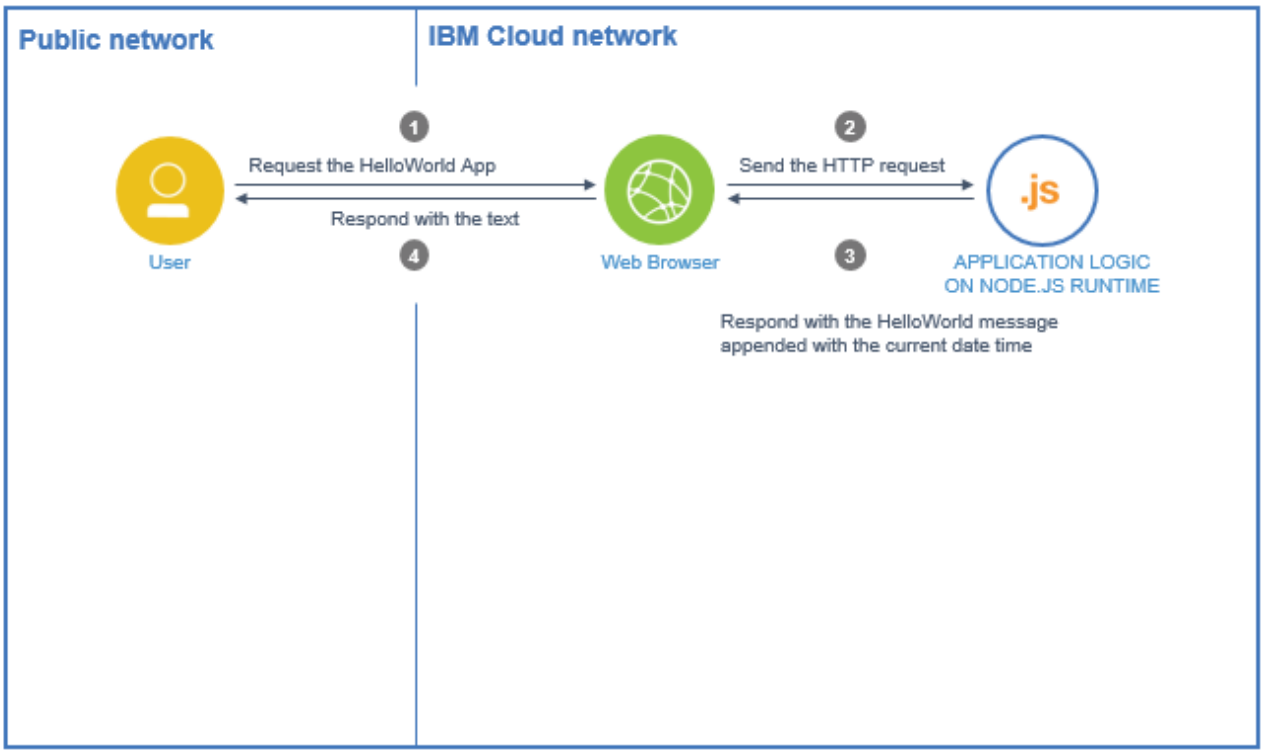
IBM SDK for Node.js uses the JavaScript programming language for server-side applications. Instead of running scripts in a web browser, the Node.js application interprets and runs JavaScript applications on a server. Node.js works on an event-driven model, which means it responds to events through callback functions that Node.js calls when an operation completes.

By the end of this exercise, you accomplish these objectives:

- Create an IBM SDK for Node.js application.
- Write your first Node.js application.
- Deploy an IBM SDK for Node.js application on an IBM Cloud account.
- Create a Node.js module and use it in your code.

Introduction

The architecture of the Node.js Hello World app is shown in the following figure.



The following steps explain the sequence of interactions between the components that are used in the exercise:

1. The user accesses the web application in a web browser through a URL that is provided by IBM Cloud.
2. The web browser sends the HTTP request to the deployed Node.js app in IBM Cloud.
3. The Node.js app listens to the incoming request and responds with a Hello World message that includes the current date and time.
4. The web browser shows the received message to the user.

Requirements

Before you start, be sure that you meet these prerequisites:

- An IBM Cloud account.
- A workstation that has these components:
  - Internet access
  - Web browser: Google Chrome or Mozilla Firefox
  - Operating system: Linux, Mac OS, or Microsoft Window

Exercise instructions

In this exercise, you complete the following tasks:

__ 1. Log in to your IBM Cloud account.

__ 2. Create the Node.js application on IBM Cloud.

__ 3. Enable continuous delivery by using toolchain.

__ 4. Create a Hello World Node.js server.

__ 5. Add a module to the Node.js application.

__ 6. Stop and delete the application.

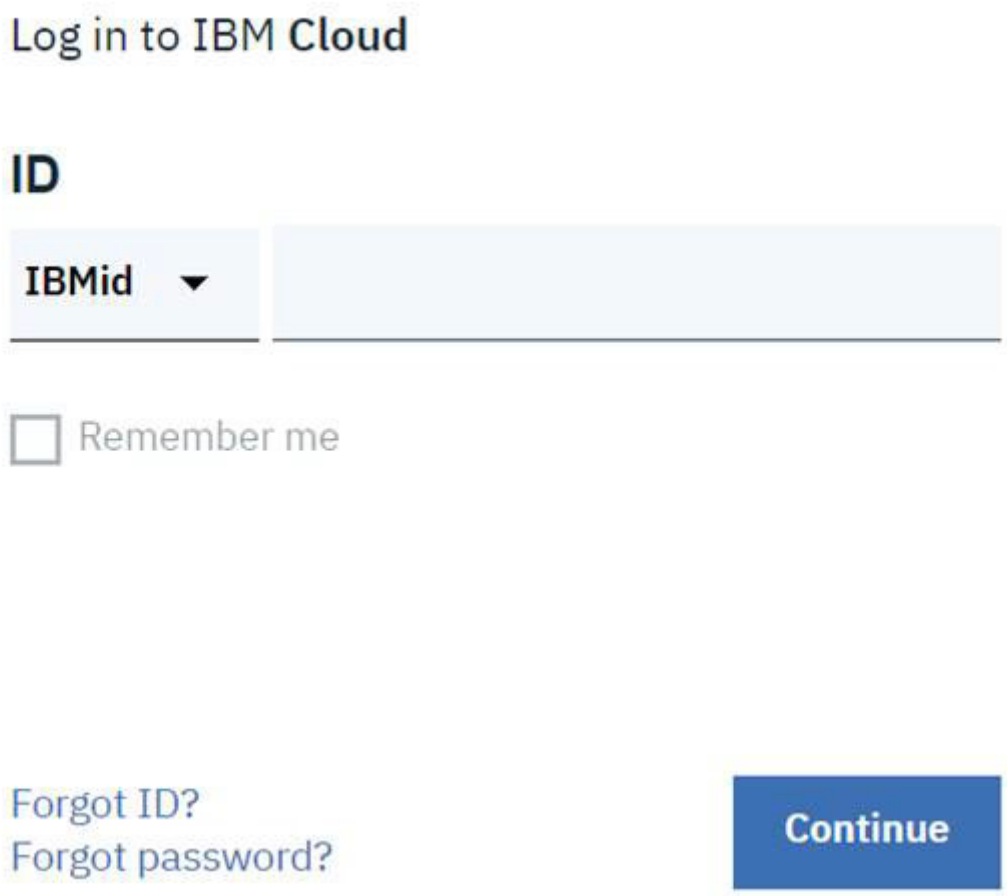
Part 1: Logging in to your IBM Cloud account

Log in to your IBM Cloud account by completing these steps:

__ 1. Open your web browser and enter the following web address:

`https://cloud.ibm.com`

__ 2. The IBM Cloud login page opens, as shown in the following figure. Enter your IBMid (which is the email that you signed up with). Click **Continue**.



__ 3. Enter your password and click **Log in**.

Part 2: Creating the Node.js application on IBM Cloud

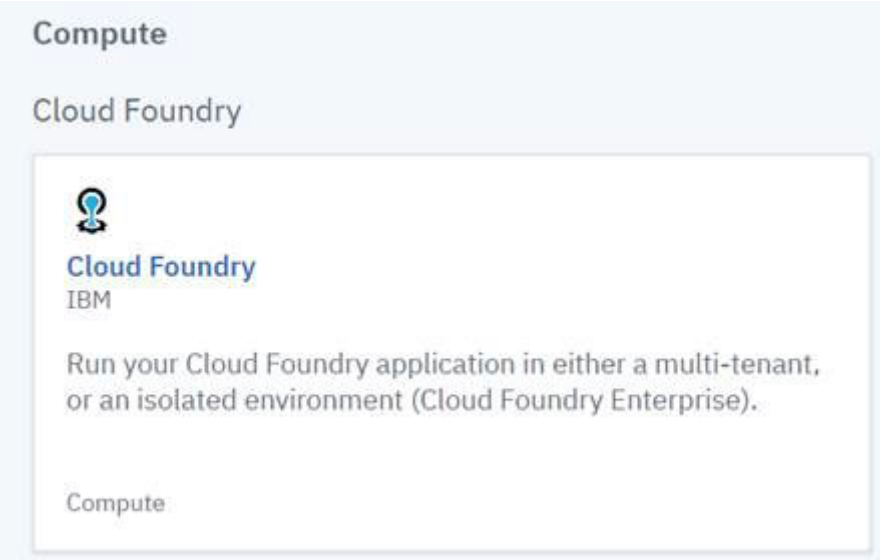
Create the Node.js app by using the SDK for Node.js runtime on IBM Cloud by completing these steps:

__ 1. In the IBM Cloud Dashboard, click **Create resource**, as shown in the following figure.

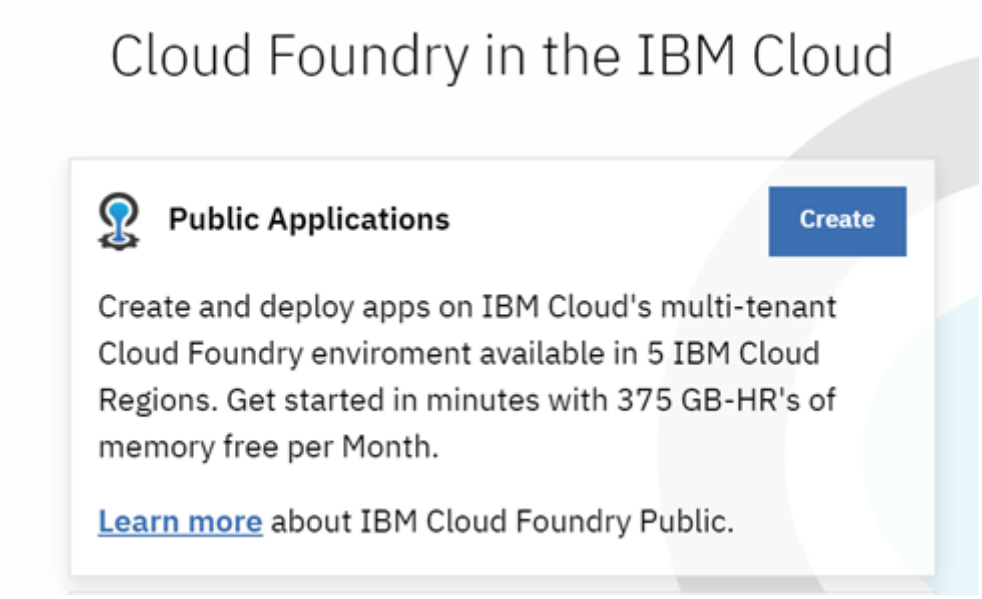
__ 2. The IBM Cloud Catalog page opens. It lists the infrastructure and platform resources that can be created in IBM Cloud. In the search field, enter **Cloud Foundry**.

__ 3. Select **Compute** from the left side menu as shown in the following figure.

__ 4. Select Cloud Foundry under Compute, as shown in the following figure.



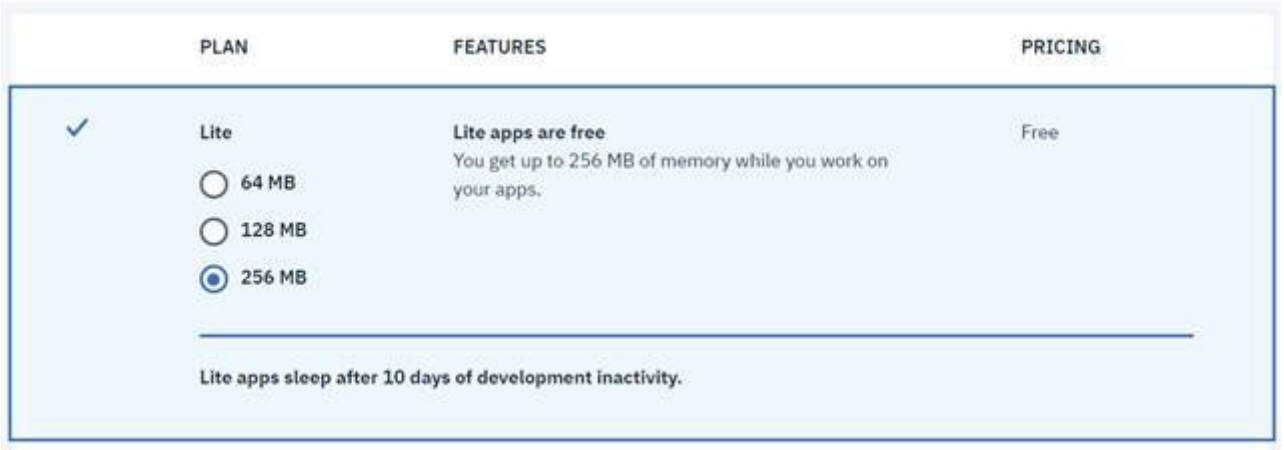
__ 5. In the Cloud Foundry page, click **Create** for Public Applications as shown in the following figure.



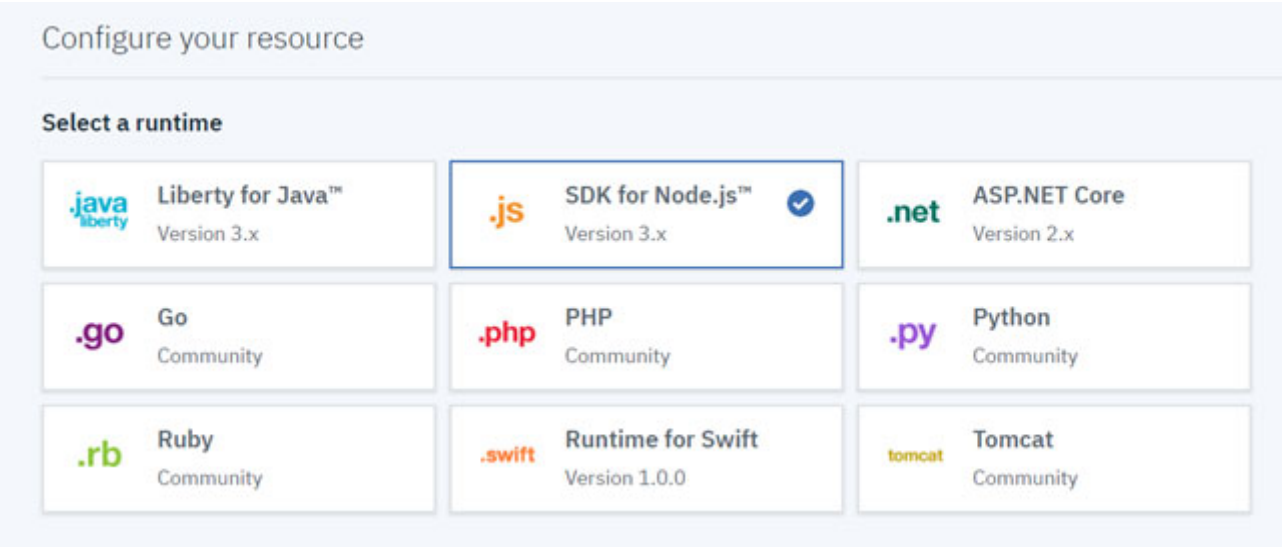
__ 6. Complete the application details to create the Cloud Foundry sample app:

__ a. Region is selected by default according to your location.

__ b. If you do *not* have a Lite account, skip this step. For Lite accounts, Pricing Plans show that the memory that is allocated to your app by default is 64 MB. For this exercise, select the maximum allocation of **256 MB**, as shown in the following figure.



__ c. Select **SDK for Node.js** from the provided runtimes as shown in the following figure.



__ d. In the App name field, enter `vy102-XXX-nodejs`. Replace XXX with three random characters, which become your unique key. You use this unique key in the naming convention of this exercise

__ e. The host name is set by default to the app name.

__ f. The domain is chosen according to your location.
Select a domain from the “Domain” drop-down list that has the subdomain `{region}.cf.appdomain.cloud`.

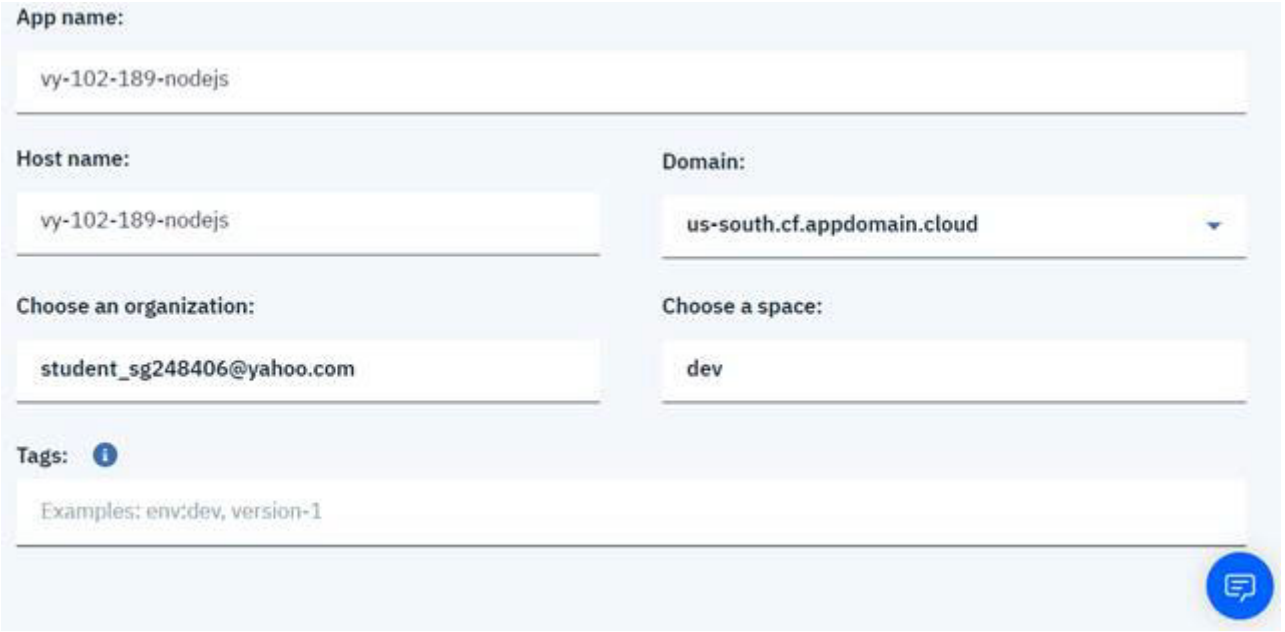
Note

Make sure that the domain selected by default has the format `{region}.cf.appdomain.cloud`. Do not choose “*`mybluemix.net`*” as a domain because it is deprecated.

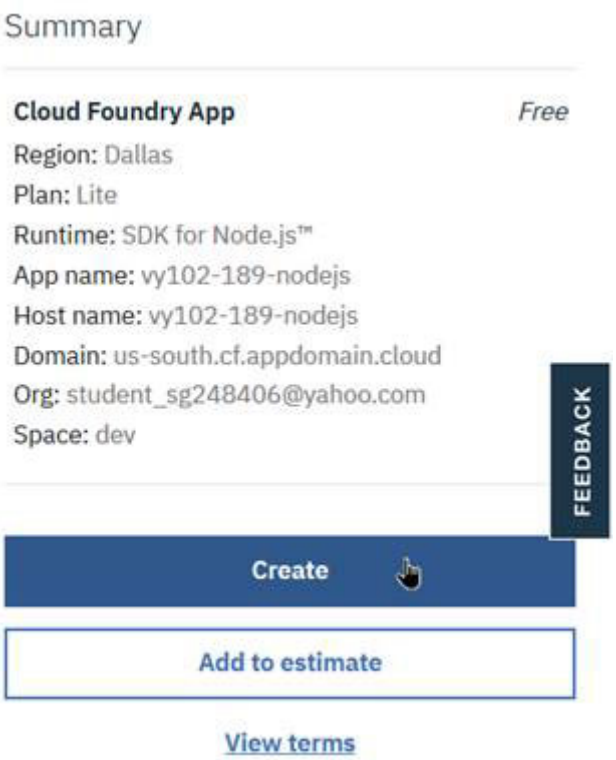
For example, if your region/location is *Dallas*, the domain is set to domain: `us-south.cf.appdomain.cloud`.

__ g. The organization is set by default to the email that you used to log in to IBM Cloud.

__ h. The space is set by default to dev, as shown in the following figure.

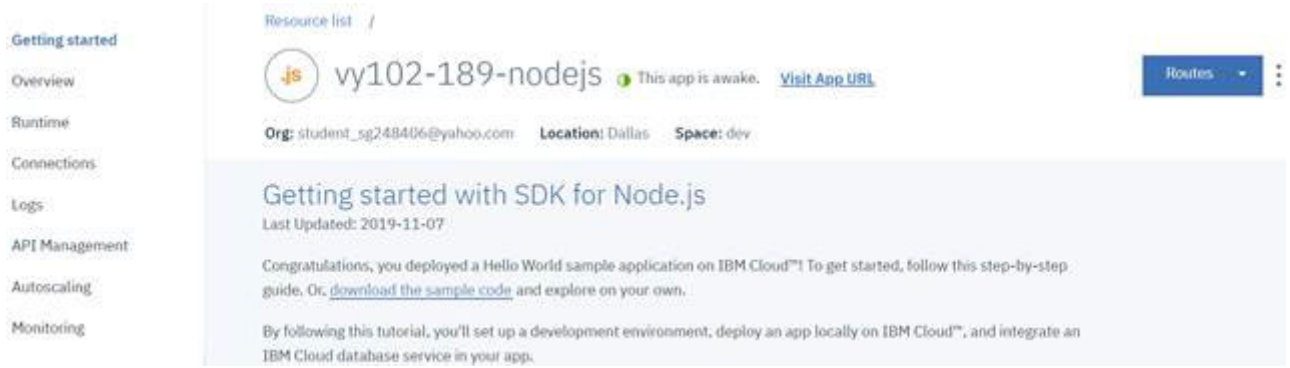


__ i. Click **Create**. IBM Cloud proceeds to deploy your application. Your application stages and deploys in a few minutes.

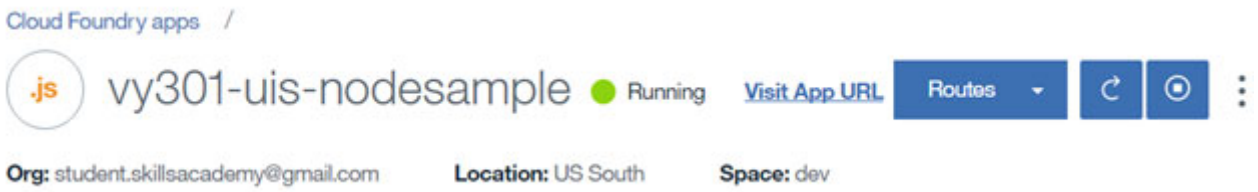


Stop

Wait until the application finishes staging and it is running in IBM Cloud before you proceed to the next step. For Lite accounts, wait for the application status “This app is awake”, as shown in the following figure.



If you do not have a Lite account, the application status should be “Running”, as shown in the following figure.

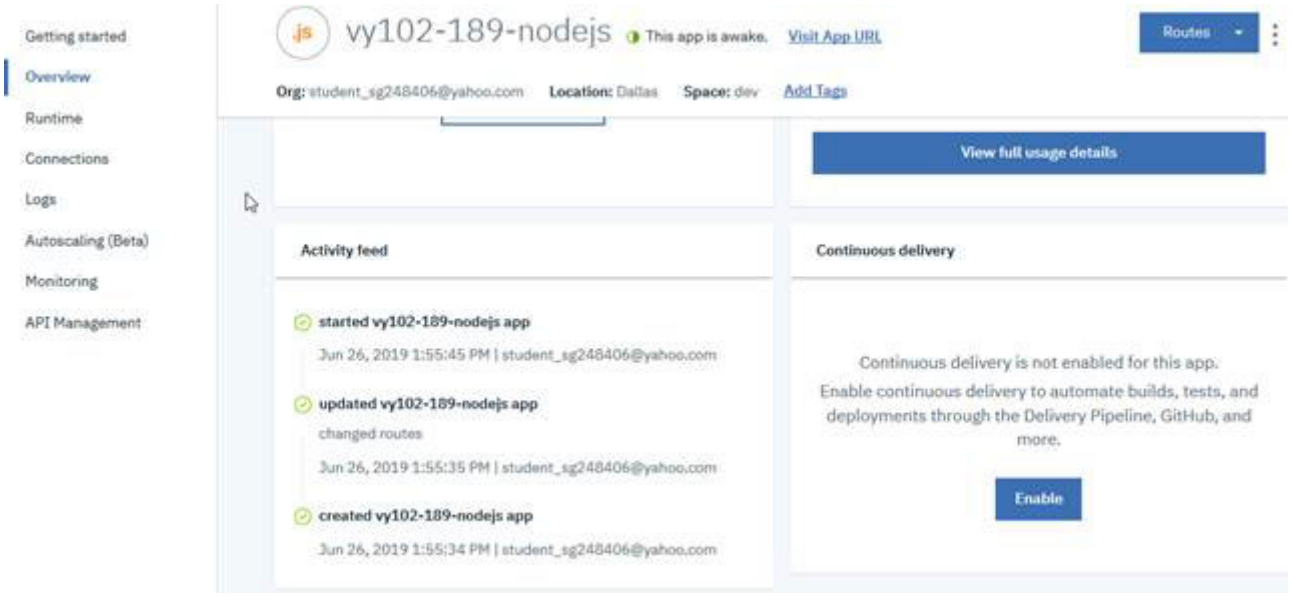


Part 3: Enabling continuous delivery by using the Toolchain

The Getting started page of your app shows instructions for accessing your app through the command-line interface (CLI). However, in this exercise, you use continuous delivery.

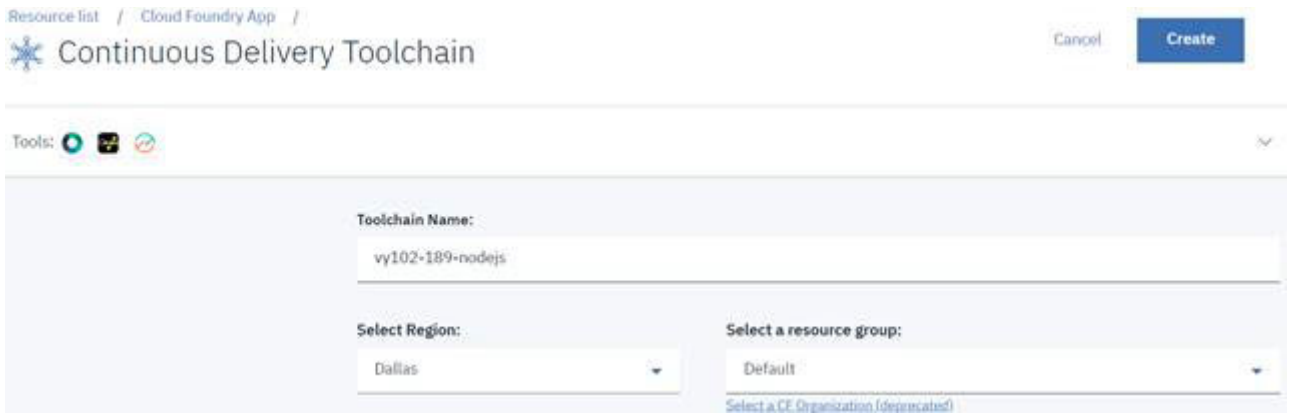
Enable continuous delivery for the Node.js app by completing these steps:

__ 1. Click **Overview** in the left pane, scroll down to the Continuous delivery tile, and then click **Enable**, as shown in the following figure.



__ 2. A new Continuous Delivery Toolchain tab opens. This toolchain includes tools to develop and deploy the application.

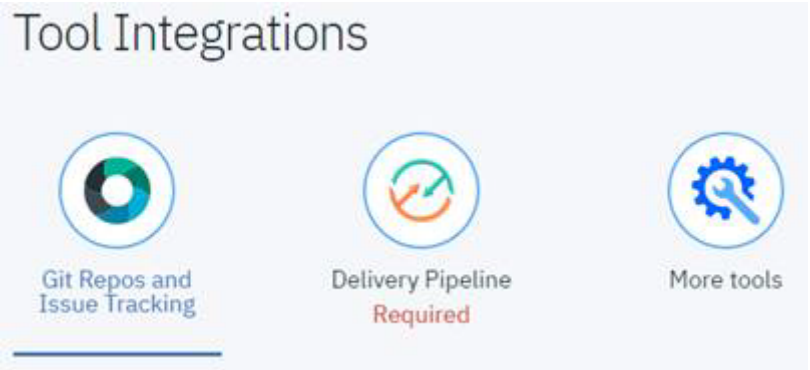
The Toolchain Name field is automatically populated. Keep the default values for both the **Select Region** and **Select a resource group** fields, as shown in the following figure.



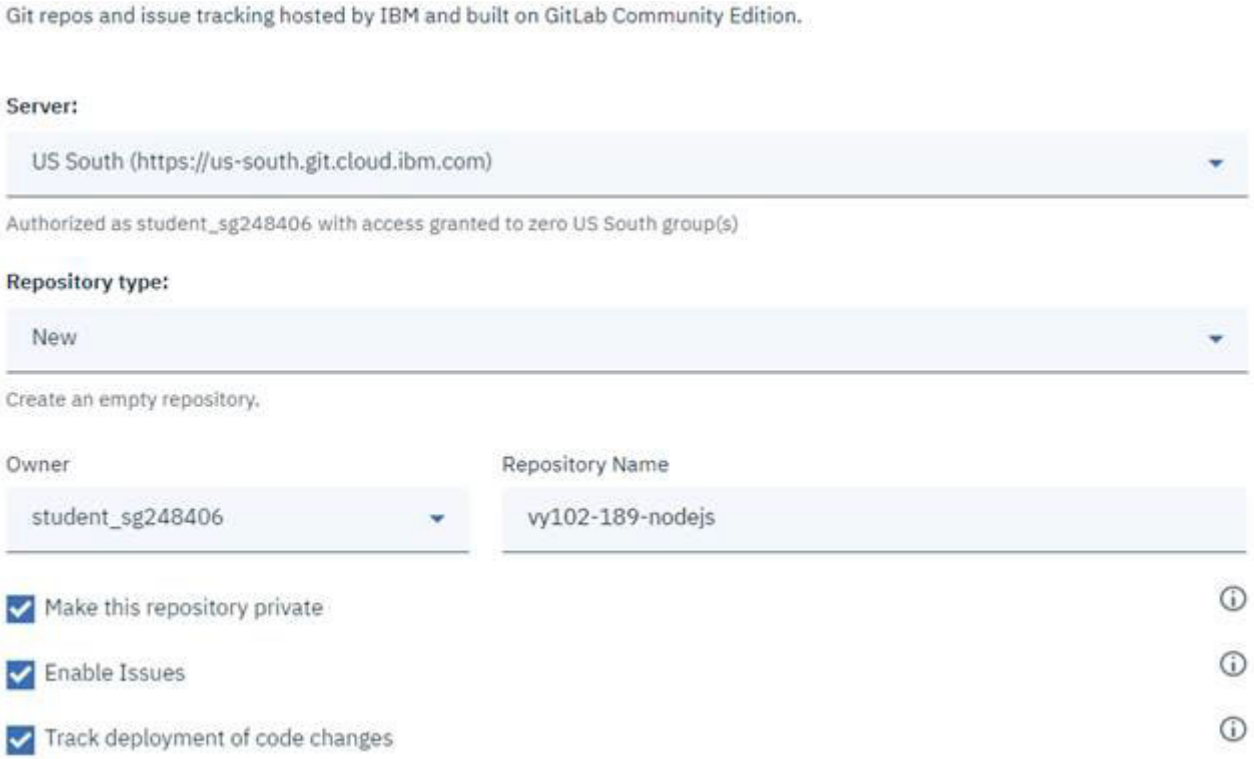
__ 3. Scroll down to see three main icons:

- Git Repos and Issue Tracking.
- Delivery Pipeline.
- More Tools.

The Git Repos and Issue Tracking icon is selected by default, as shown in the following figure.



The Git Repos and Issue Tracking section is shown as the following figure.



In the Repository type field menu you can select from the following options:

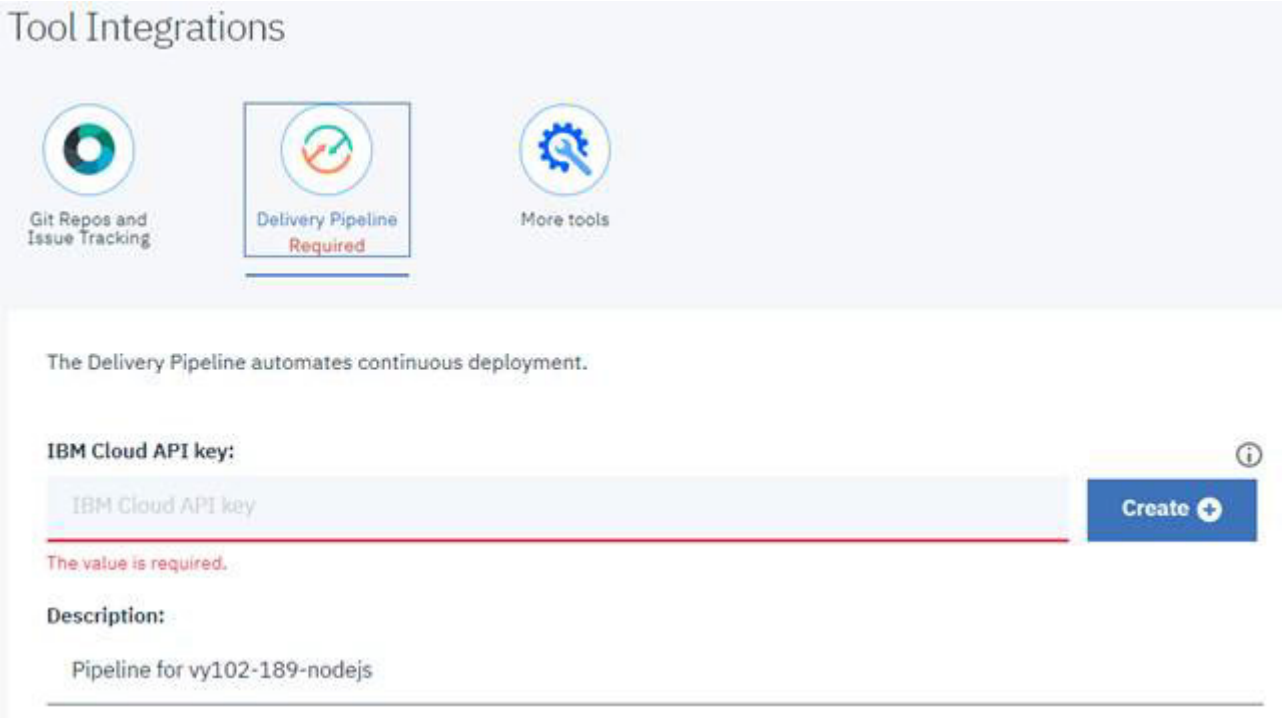
- New: Create an empty new repo.
- Fork: fork an existing GitLab repo (you specify its URL) so that you can contribute changes through merge requests
- Clone: Create a copy of an existing GitLab repo
- Existing: Link to an existing repository and continue working on it.

__ 4. The default selection is Clone. For this exercise, select **New** so that you can start developing your application from scratch.

__ 5. For the other fields, keep their default values and click the **Delivery Pipeline** tab.

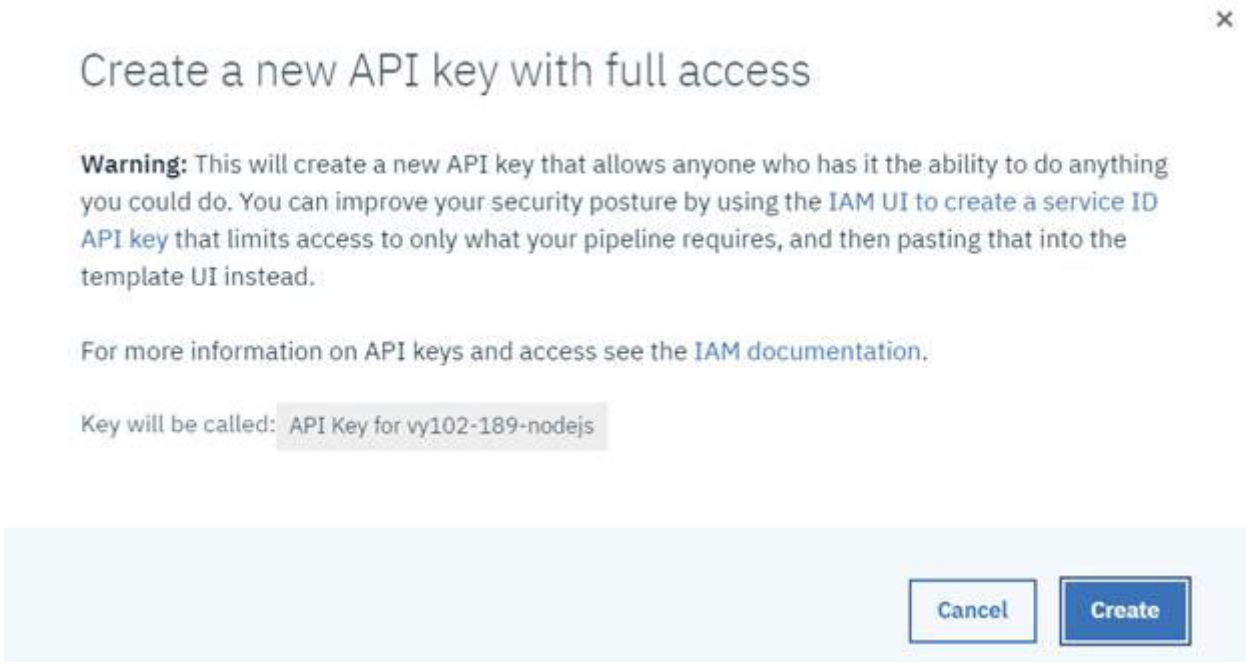
__ 6. In the Delivery Pipeline tab, which is shown in the following figure, two fields are shown:

- IBM Cloud API Key: Required to access IBM Cloud Runtime.
- Description: Describes the pipeline that is created in this step.

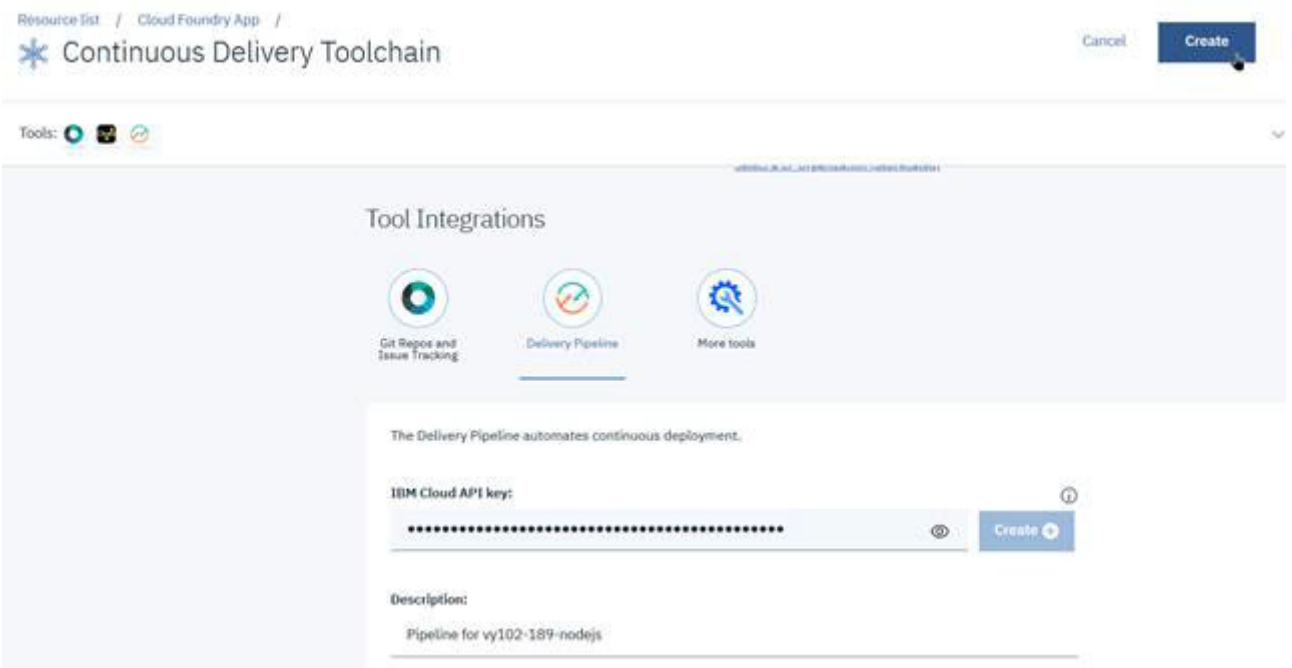


Click **Create +**.

__ 7. A window opens where you can create the API key, as shown in the following figure. Click **Create**.



__ 8. The window closes, and the main page shows the generated IBM Cloud API key, as shown in the following figure. Click **Create**.

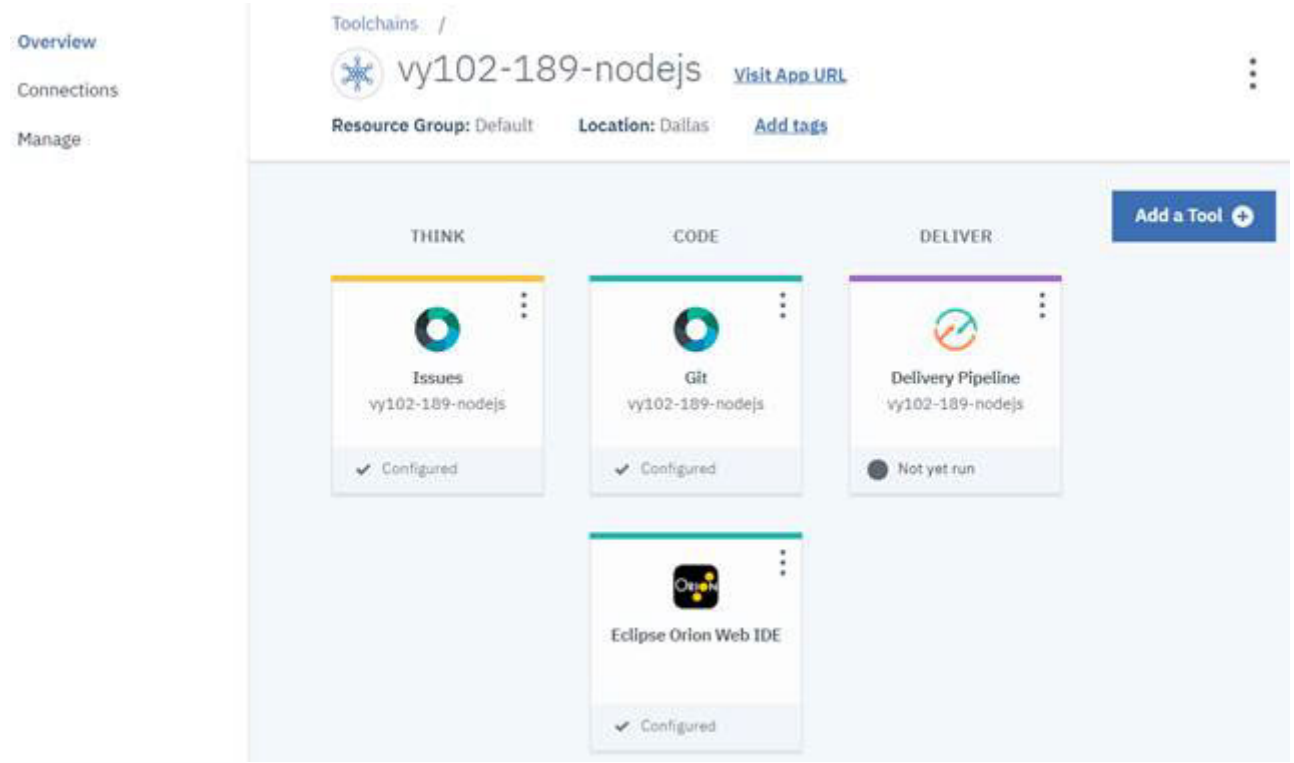


__ 9. A new page opens and shows the three main phases (THINK, CODE, and DELIVER). A toolchain is a set of tool integrations that support development, deployment, and operations tasks. The UI to create a new toolchain groups the tools into the following phases:

- **THINK**: In this phase, you plan the application by creating bugs, tasks, or ideas by using the Issue Tracker, which is part of the Git repository.

- **CODE**: In this phase, you implement the application by using a GIT repository as a source code management system, and by using a web IDE (Eclipse Orion) to edit your code online. In the repository, you can specify whether to clone a repository or start from scratch by selecting **New** in the repository type.
- **DELIVER**: In this phase, you configure the delivery pipeline. You can specify automatic build, deployment, and testing of your code after a developer pushes new code to the Git repository.

  The following figure shows these three phases.

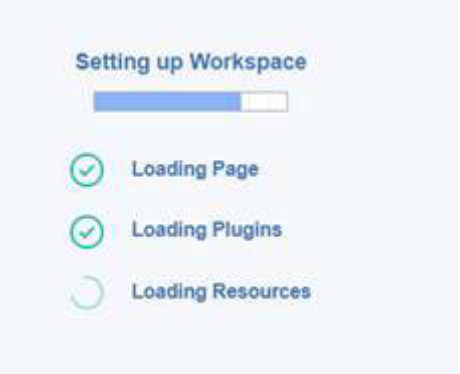


Part 4: Creating a Hello World Node.js server

The following steps describe how to write Node.js code in the Eclipse Orion Web IDE, and how to link the code to the Node.js app on IBM Cloud that you created in the previous sections.

Complete the following steps:

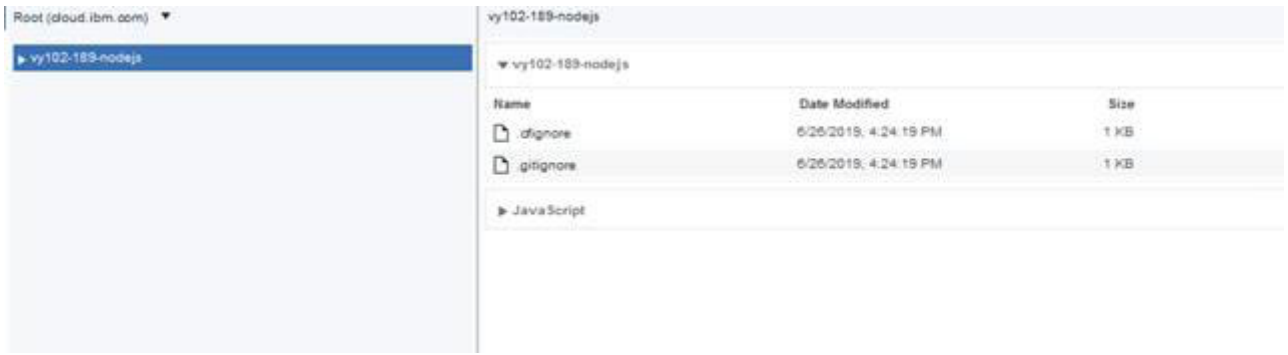
__ 1. On the Toolchains page, click the **Eclipse Orion Web IDE** icon. The page takes some time to set up the workspace, as shown in the following figure.



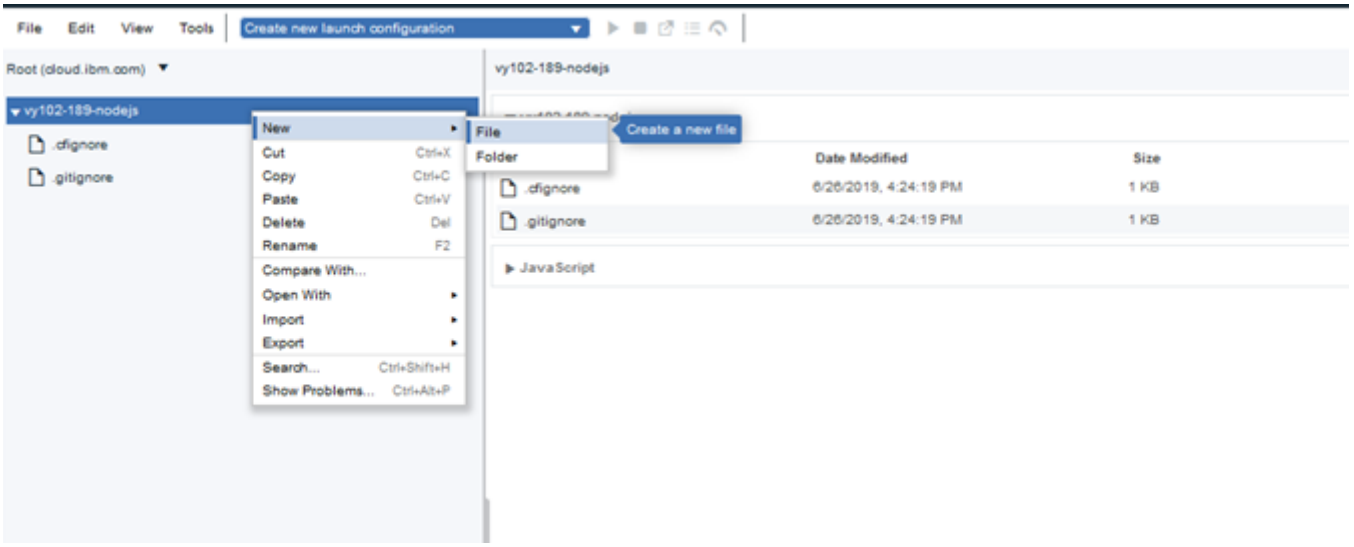
__ 2. The page now shows the generated Node.js project in the Eclipse Orion Web IDE.

The Eclipse Orion Web IDE is a browser-based development environment where you develop for the web. You can develop in JavaScript, HTML, and CSS with the help of content-assist, code-completion, and error-checking features.

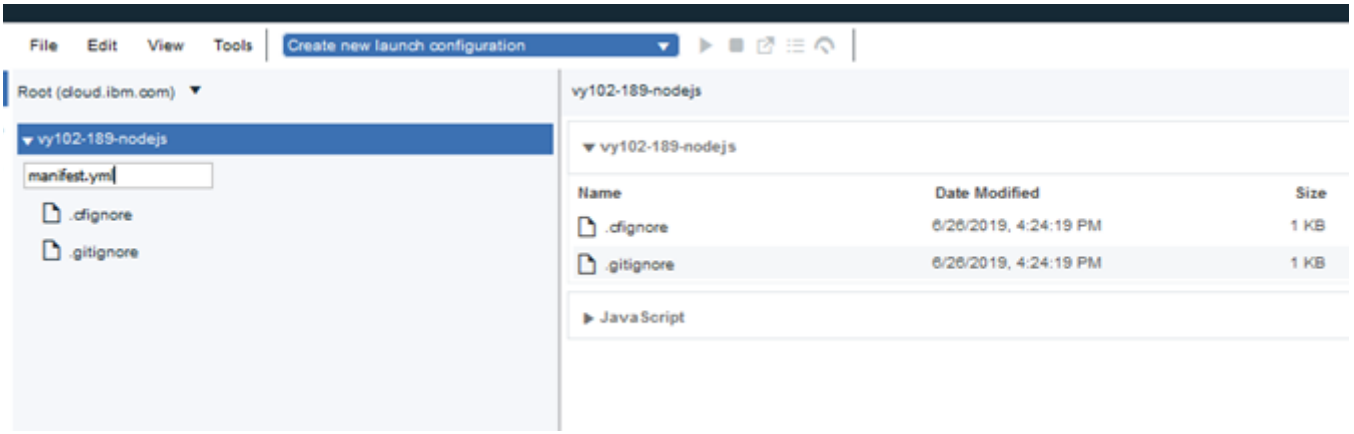
The left pane of the current page shows the project structure. Currently, no Node.js files are available. In the next steps, you create these files one by one.



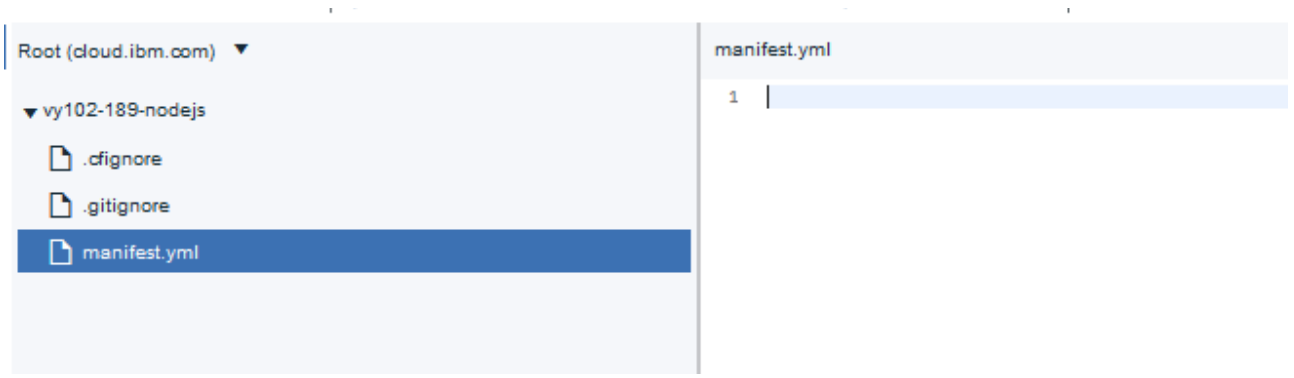
__ 3. Right-click the root of the project (named vy102-XXX-nodejs) from the project structure on the left, and then select **New** > **File**, as shown in the following figure.



__ 4. A text field is displayed. Enter `manifest.yml` and then press **Enter**, as shown in the following figure.



The `manifest.yml` file is now created, as shown in the following figure.



The `manifest.yml` file contains information about the deployment of the application to IBM Cloud.

__ 5. Add the following code snippet to `manifest.yml`.

```
applications:
- path: .
memory: 256M
instances: 1
name: vy102-XXX-nodejs
host: vy102-XXX-nodejs
disk_quota: 1024M
```

Note

Replace *XXX* with your unique key to give a unique name to your Node.js app and host.

The following table explains the meaning of each attribute in the `manifest.yml` file.

Attribute	Description
Path	Indicates to Cloud Foundry the directory where the application is located.
Memory	Specifies the memory limit for all instances of the application.
Instances	Specifies the number of app instances that are needed for the application. A value of one instance is enough for this exercise.
Name	The name of the application that you specified when you created the Cloud Foundry app in IBM Cloud.
Host	The subdomain where the application is available.
disk_quota	Specifies the allocation amount of disk space for the app instance.

__ 6. Save the file by clicking **File** > **Save**.

__ 7. Right-click the root of the project (named vy102-XXX-nodejs), select **New** > **File**, name the file package.json, and then press **Enter**.

The package.json file holds various metadata that are relevant to the project. For example, in the Code snippet for package.json file, the `start` field specifies `app.js`, which is the starting point (entry JavaScript file) for this application. The package.json file also specifies the dependencies on other Node.js modules.

The package.json file is used by the Node Package Manager (NPM), which is the default package manager for the JavaScript runtime environment in Node.js.

NPM provides two main functions:

- Online repositories for Node.js packages and modules, which are searchable at the Node.js website at `https://nodejs.org/en/`.
- A command-line utility to install Node.js packages and perform version management and dependency management of Node.js packages.

  NPM accesses the package.json file to perform tasks such as registering the application by using the name field in the package.json file, and making sure that the

dependencies in the package.json file are available in the Node.js online repository with the specified versions.

Insert the following code snippet in the package.json file.

```
{
       "name": "NodejsStarterApp",
        "version": "0.0.1",
        "description": "A Hello World NodeJS sample",
        "scripts": {
            "start": "node app.js"
      }
}
```

The main attributes of the package.json file are described in the following table.

Attribute	Description
Name	The name of your node application. The name and version together form an identifier that is assumed to be unique.
Version	The version of the current application. The version must always be in the form of *n.n.n* (for example, 1.0 is not an allowed value).
Description	A simple description about the application.
Scripts	A dictionary property containing script commands that are run at various times in the lifecycle of your package. The key is the lifecycle event, and the value is the command to run at that point. In this case, the start event is needed to refer to the node app.js command.

__ 8. Save the file by clicking **File** > **Save**.

__ 9. Create a file and name it app.js. This is the entry JavaScript file for your application.

__ 10. Insert the following line into the app.js file as shown in the following figure.

```
var http = require("http");
```

The `require` statement is used to import node modules that are managed by NPM.

The name of the built-in module, http, is managed by NPM. This module is used to allow Node.js to transfer data over the Hypertext Transfer Protocol (HTTP).

__ 11. Enter the following code snippet to create the server. This server is expected to handle the HTTP requests coming from the client.

```
// Read the port from the underlying environment.
// If it does not exist, use the default port: 8080
var port = process.env.VCAP_APP_PORT || 8080;

// Create the server and listen to requests on the specified port.
http.createServer(function (request, response) {

.listen(port);
```

Your `app.js` file, which now looks like the following figure, shows the **`createServer`** function.

As shown, the HTTP module has a **`createServer`** callback function, which is responsible for creating a server. This server provides a callback that receives two parameters:

- **`request`**: This object contains the HTTP request details from the client. You should be able to read parameters from this request to use in your application.
- **`response`**: This object is created by the HTTP module. You add the response details to it. Then, the HTTP module sends the response object back to the client as an HTTP response to the original HTTP request from the client.

The server listens to a port variable. The port variable is set by the following value:

`process.env.VCAP_APP_PORT`

The `process.env` property returns an object that contains the user's environment. In this case, it contains properties that are related to the deployed application and its underlying IBM Cloud environment.

The `process.env.VCAP_APP_PORT` port value is provided by IBM Cloud automatically when you create the project.

If `process.env.VCAP_APP_PORT` is null for some reason (this can happen if you run this application outside IBM Cloud), then the port variable is set to `8080` (port 8080 is the default port for Node.js).

__ 12. Inside the **createServer** function, add the following code snippet:

```
// Set the content type of the response
response.writeHead(200, {'Content-Type': 'text/plain'});
```

This line has the response's header details:

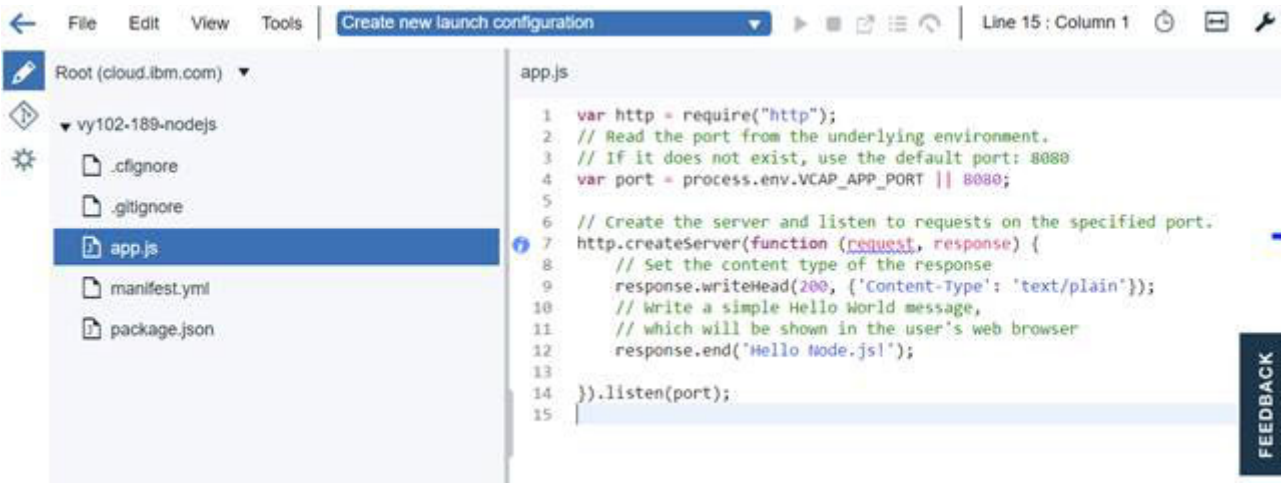
- Status code: Identifies whether the request was a success, a bad request, a forbidden request, and so on. The code for success is 200.
- Content type: Identifies the type of content that is returned in the body of the response. The content type can be text/plain, text/html, and other types. In this scenario, the content type is set to text/plain.

__ 13. Add the following code snippet to the end of the **createServer** function:

```
// Write a simple Hello World message,
// which will be shown in the user's web browser
response.end('Hello Node.js!');
```

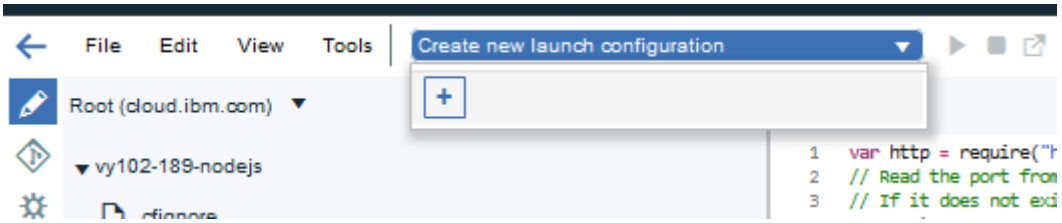
The end function is used to add the text that is passed as an argument to the body of the response.

The `app.js` file now looks like the one in the following figure, which shows the added response.



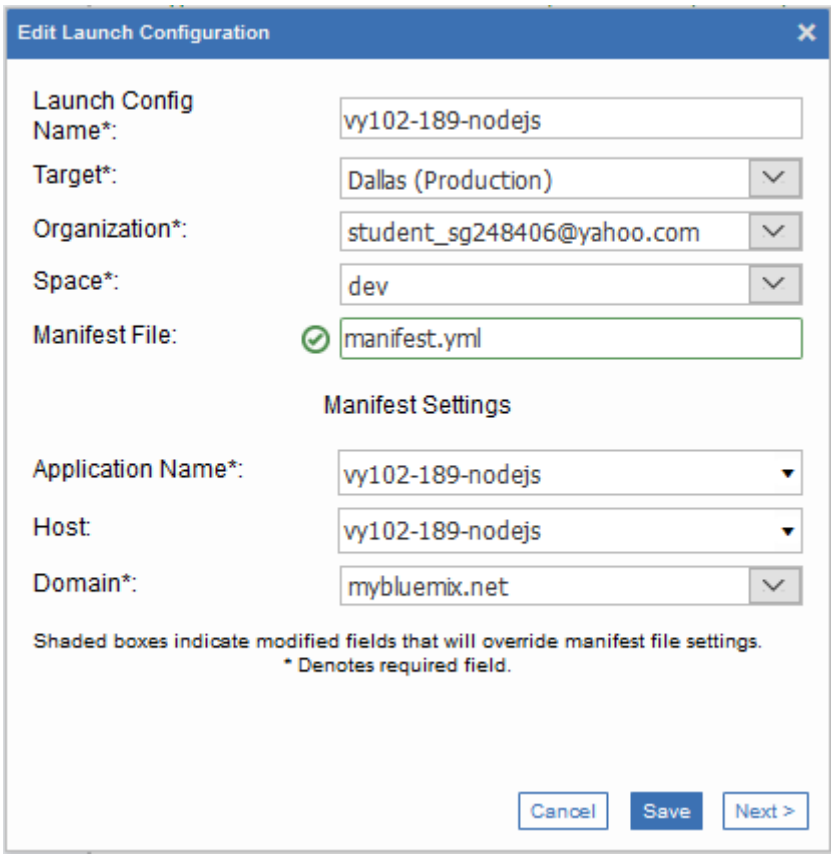
__ 14. Save the file by clicking **File** > **Save**.

__ 15. In the server toolbar, select **Create new launch configuration** from the drop-down list and click the **+** button to show the Edit Launch Configuration window, as shown in the following figure. If you do not have the **Create new launch configuration** option, skip this step.



__ 16. In the Edit Launch Configuration window, confirm that the values that are selected by default for the following fields are correct:

- Target is set to the location for your account.
- Organization is set to your email address.
- Space is set to **dev**.

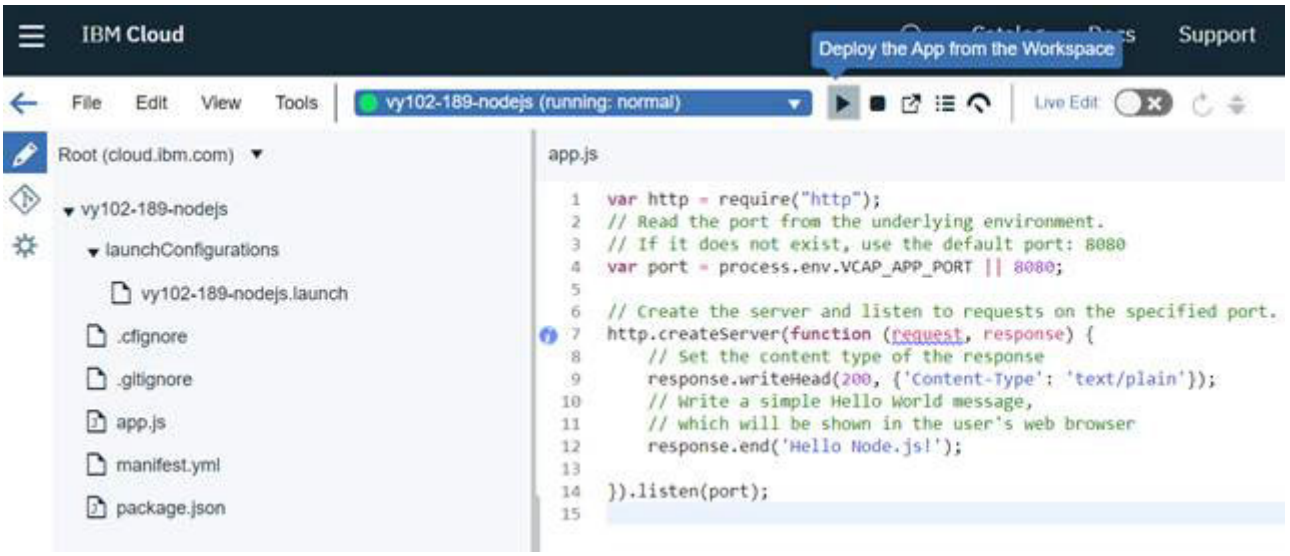


Note

If the message "**`Loading deployment settings...`**" is displayed in the Edit Launch Configuration window for a long time, change the default-selected Target field to the region that matches your account, for example, Dallas in this example. After you select the correct target, the other fields load successfully.

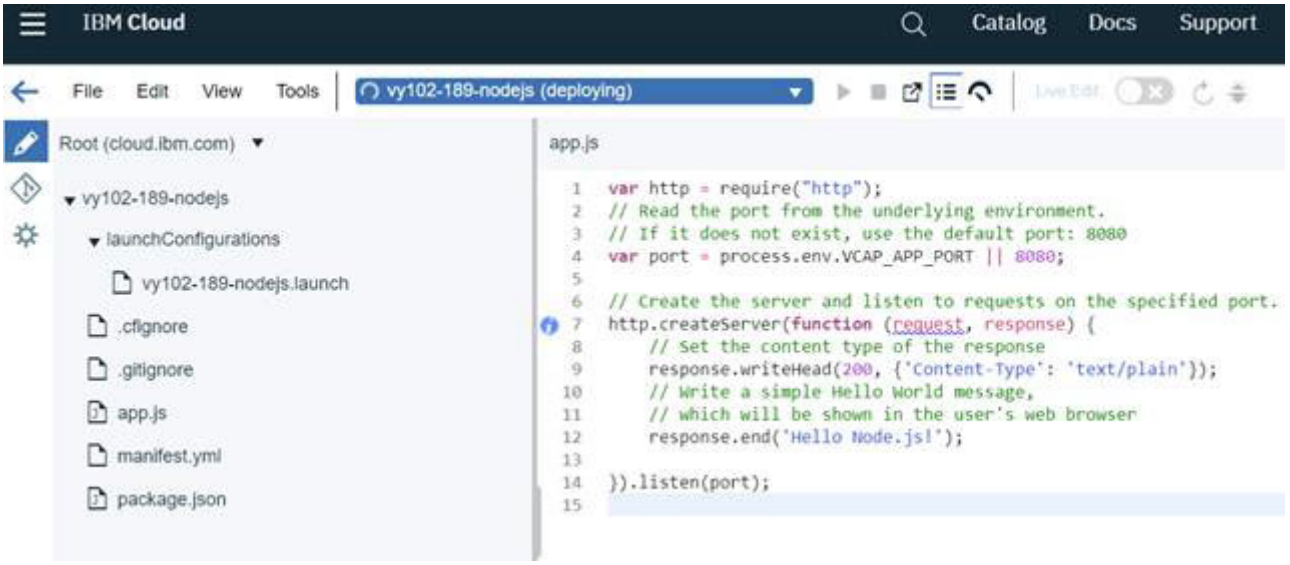
__ 17. Click **Save**.

__ 18. Now, you can click the play button (**Deploy the App from the Workspace**), as shown in the following figure. Clicking this button deploys the application to IBM Cloud.



__ 19. You might receive a notification, which warns you that your application will be redeployed. Click **OK** to confirm.

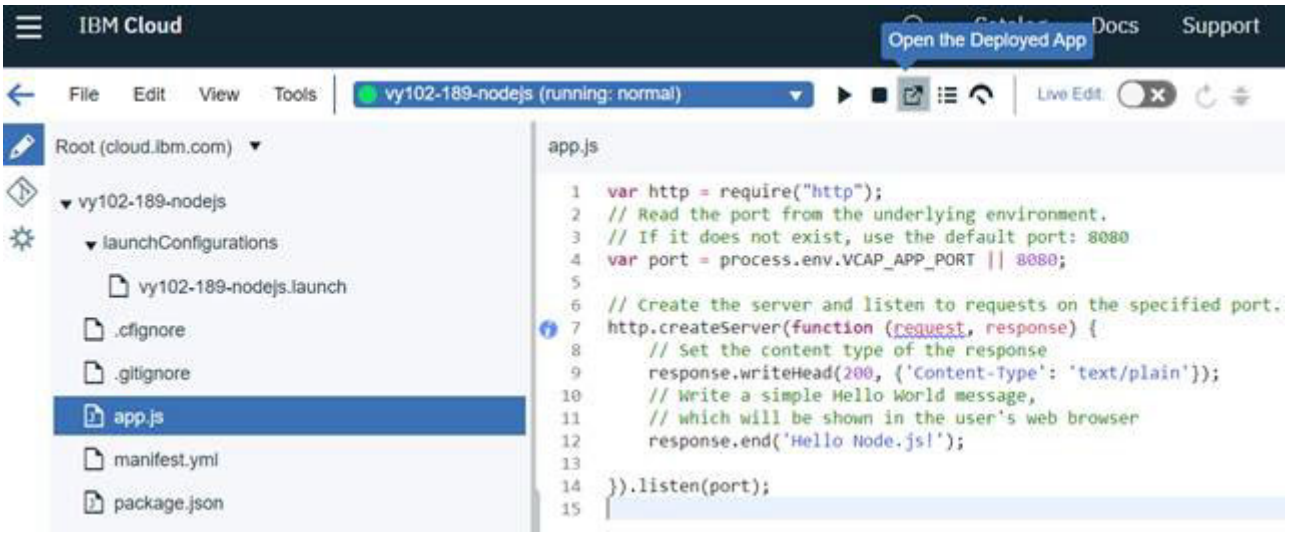
The deployment status indicates that deployment is in progress, as shown in the following figure.



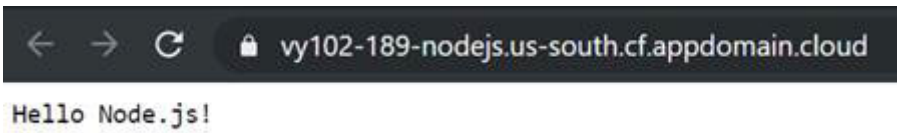
__ 20. After the deployment is complete, click **Open the Deployed App**, as shown in the following figure.

Note

If you receive an error message about a timeout, refresh the page.



A new tab opens and shows the Hello Node.js! message as shown in the following figure.



Troubleshooting

If you receive an error message or you still see the default message from the sample app “Hello World!”, refresh the page and deploy the app again.

Part 5: Adding a module to the Node.js application

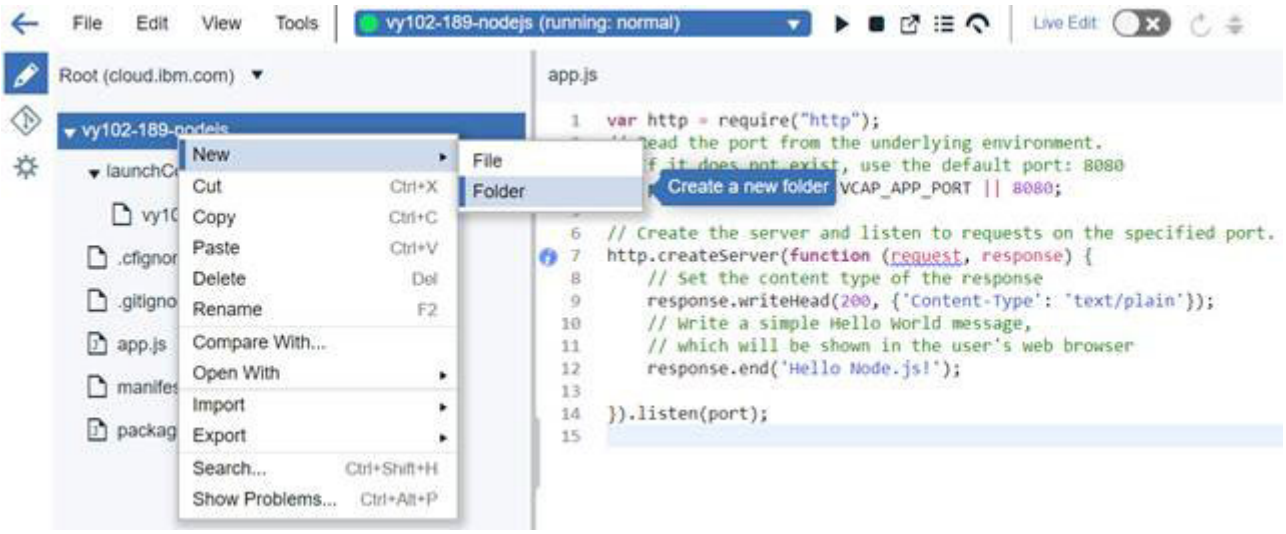
In this part, you add a Node.js module to your application.

A module encapsulates related code into a single unit. Creating a module means moving the related functions into one file. This point is illustrated by the following example, which involves an application that is built with Node.js.

Several modules are built in Node.js. For example, the HTTP module that is used in the previous section is a built-in module. The following steps show how to create your own custom module:

__ 1. Close the **Running Application** tab and return to the Eclipse Orion Web IDE.

__ 2. Right-click the **vy102-XXX-nodejs** folder, and then select **New** > **Folder**, as shown in the following figure.



__ 3. Name the new folder `currentDate`.

__ 4. In the `currentDate` folder, create a file that is named package.json.

__ 5. In this package.json file, add the following code snippet.

```
{
    "name": "currentDate",
    "main": "./lib/currentDate"
}
```

The results are shown in the following figure.

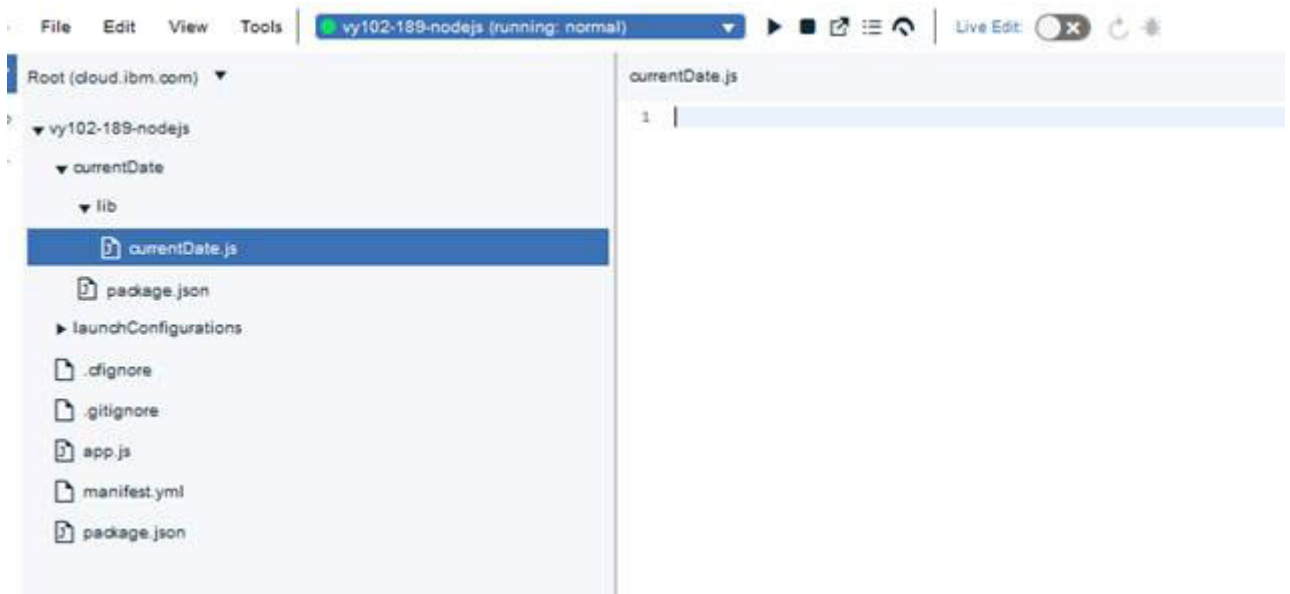
The figure shows that the new package.json file represents the new module:

- It has the usual name attribute and another attribute that is named main.
- This main attribute has the path of the JavaScript file that contains the code of this module.
- The value of the main attribute is ./lib/currentDate. You must create a currentDate.js file inside the lib folder, which you do in the next steps.

__ 6. Save the file by clicking **File** > **Save**.

__ 7. In the currentDate folder, create a folder that is named lib.

__ 8. In the lib folder, create a file that is named currentDate.js. The result is shown in the following figure.



In the `currentDate.js` file, enter the following code snippet:

```
exports.currentDateTime = function() {
    return Date();
};
```

This new **`currentDateTime`** function returns the current date by using **`Date()`**.

This function is added to a variable that is named *`exports`*, which is used for exposing the function so it can be used by other modules in the system. It is a special object, which by default is included in every JS file in the Node.js application.

Your file now looks like the one that is shown in the following figure.

__ 9. Save the file by clicking **File** > **Save**.

__ 10. Open the app.js file and find the following line:

```
var http = require("http");
```

Below that line, add the following line:

```
var dateModule = require('./currentDate');
```

This line imports `currentDate`, which you created previously, to your `app.js` file.

Note

In the line, the period and forward slash (`./`) characters indicate that the `currentDate` module exists in the same folder as the folder that contains this `app.js` file.

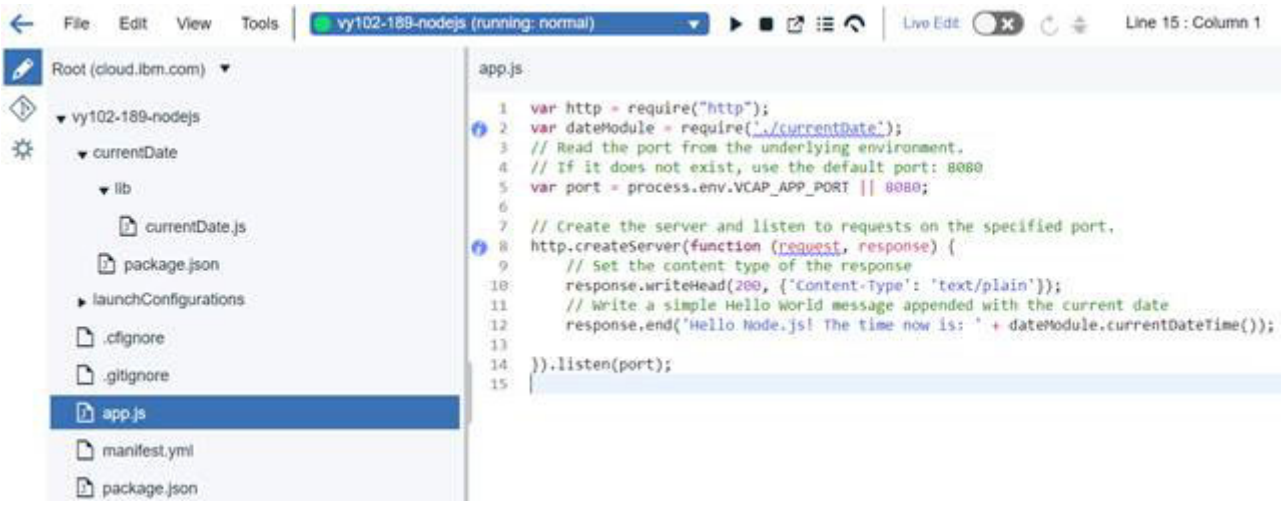
__ 11. Find the following lines:

```
// Write a simple Hello World message,
// which will be shown in the user's web browser
response.end('Hello Node.js!');
```

__ 12. Replace those lines with the following lines:

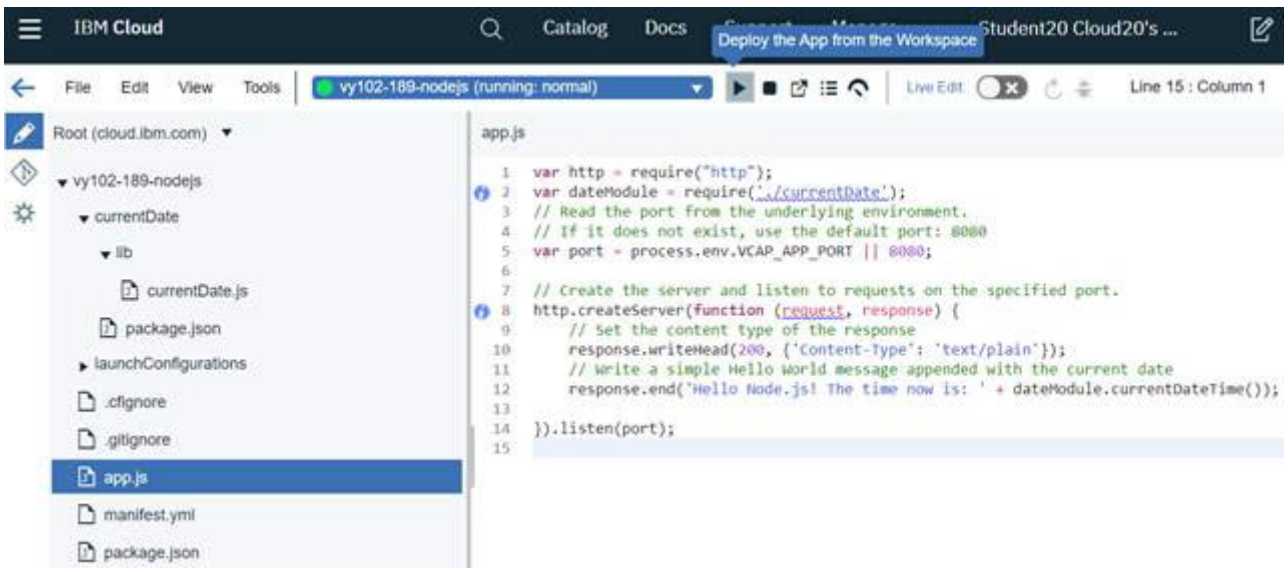
```
// Write a simple Hello World message appended with the current date
response.end('Hello Node.js! The time now is: ' +
dateModule.currentDateTime());
```

The result is shown in the following figure.

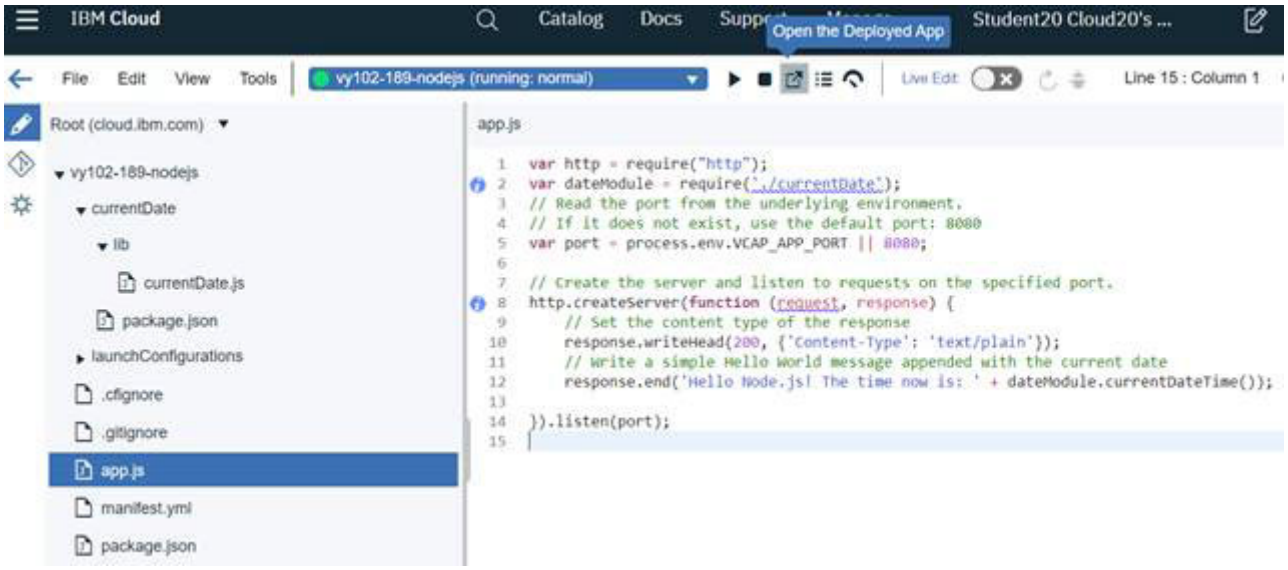


All changes are automatically saved in Eclipse Orion. If the changes are not automatically saved or if you want to force a save, select **File** > **Save**.

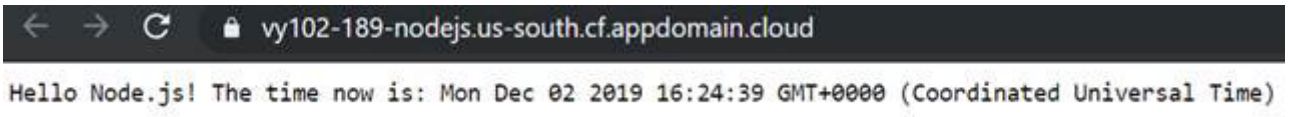
To deploy, click the play button (**Deploy the App from the Workspace**), as shown in the following figure. You might receive a notification that warns you that your application will be redeployed. Click **OK** to confirm.



__ 13. After deployment is complete, open the deployed application by clicking **Open the Deployed App**, as shown in the following figure.



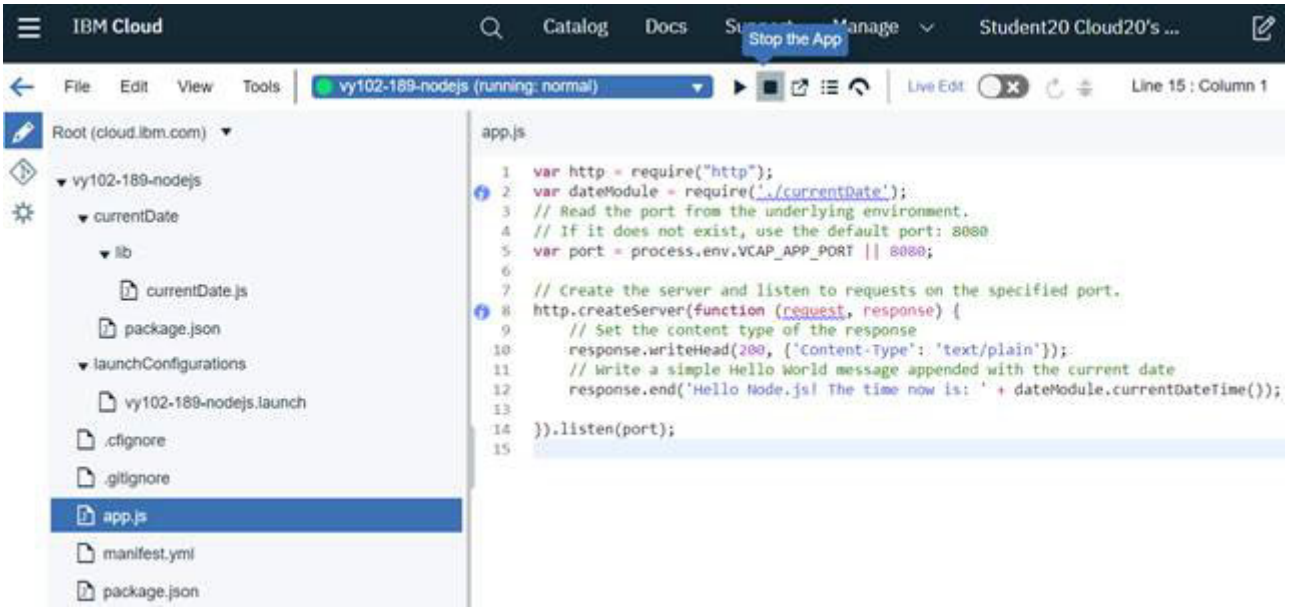
Your web browser opens and shows the result, as shown in the following figure.



Part 6: Stopping and deleting the application

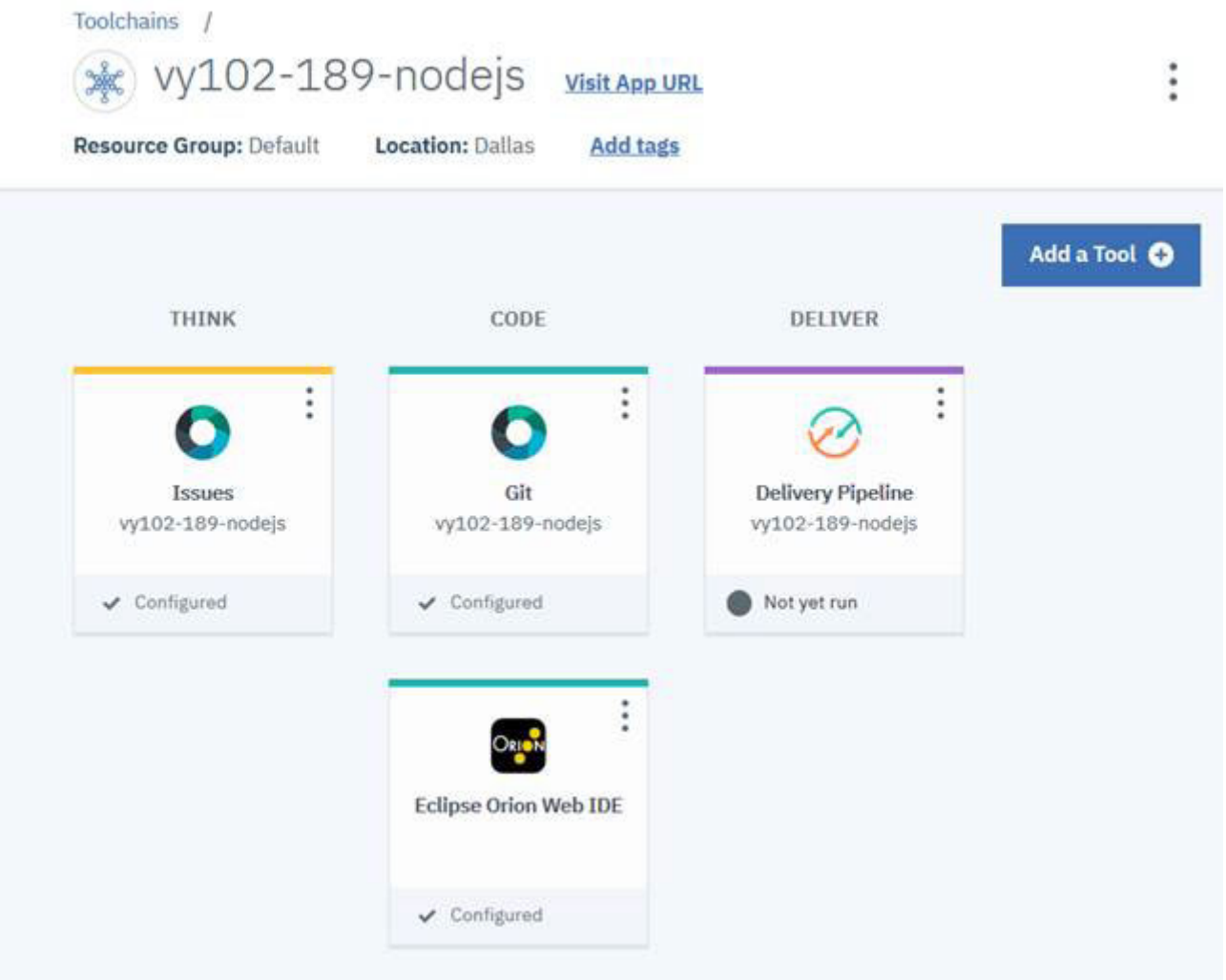
To free the resources that are assigned to your application, you can either stop your application or delete it:

__ 1. To stop your application, click the **Stop the App** icon, as shown in the following figure.

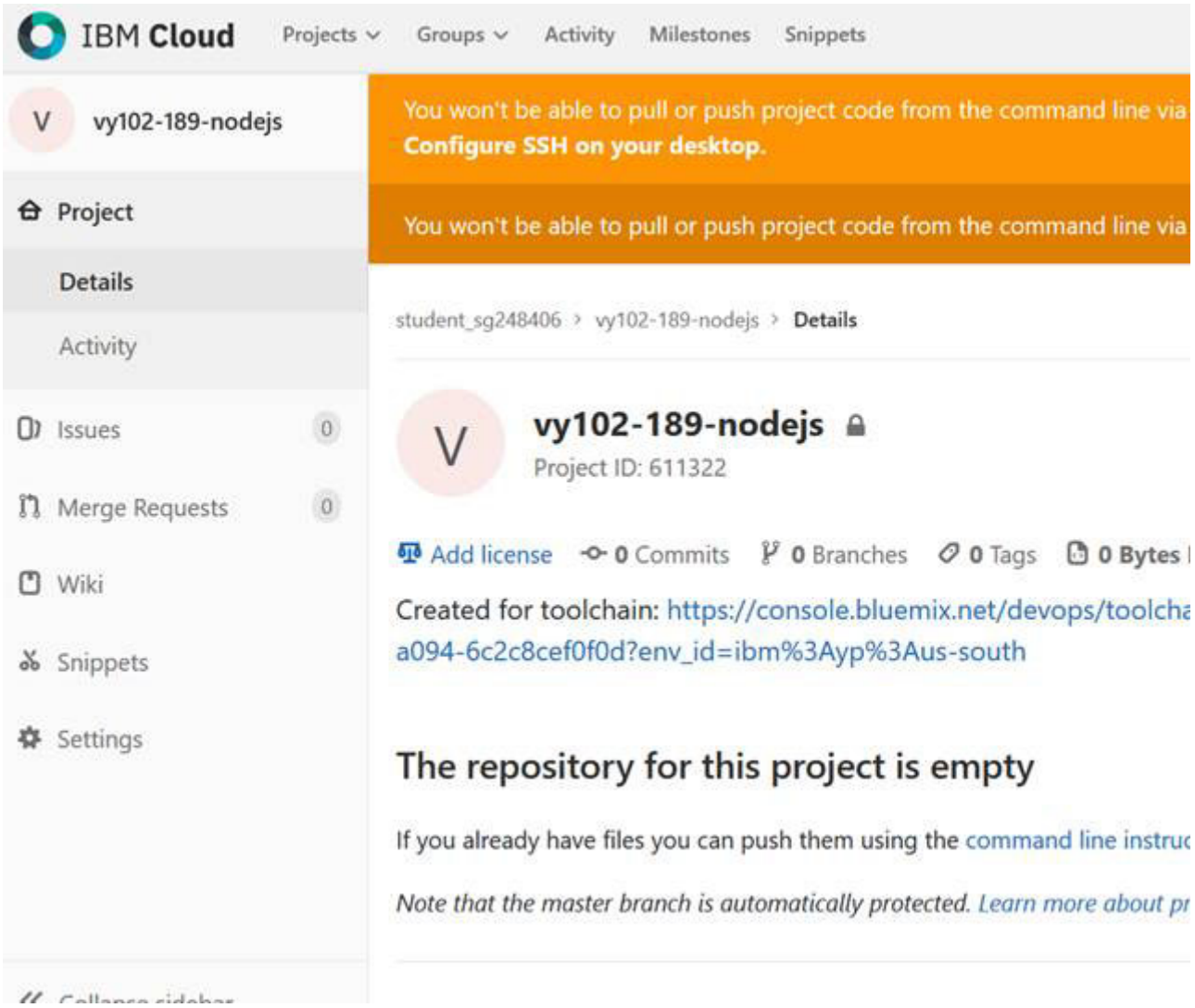


To delete the application, complete the following steps:

__ 1. Click the arrow on the upper left of the page; you are redirected to the page that is shown in the following figure.

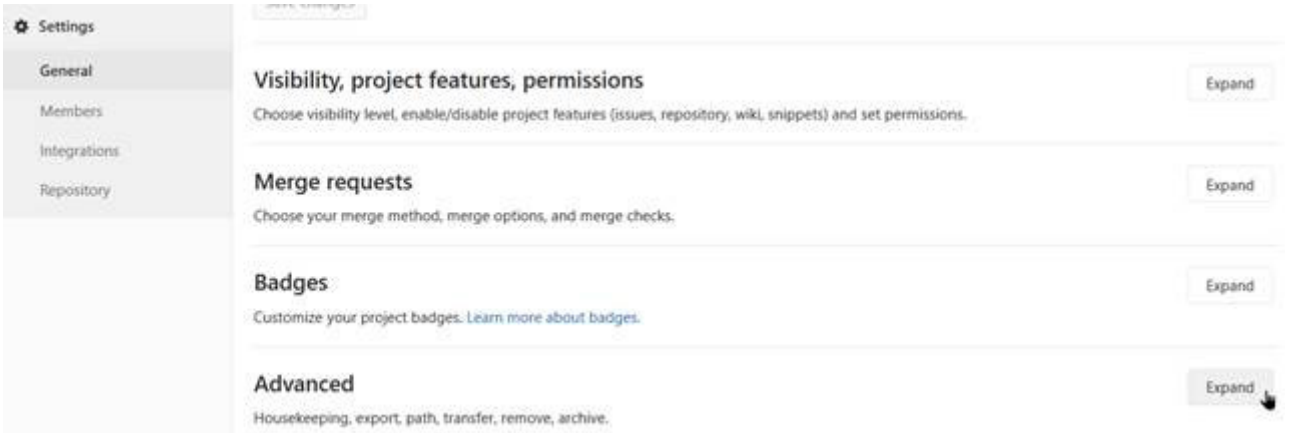


__ 2. Click **Git.** You are redirected to the project repository as shown in the following figure.



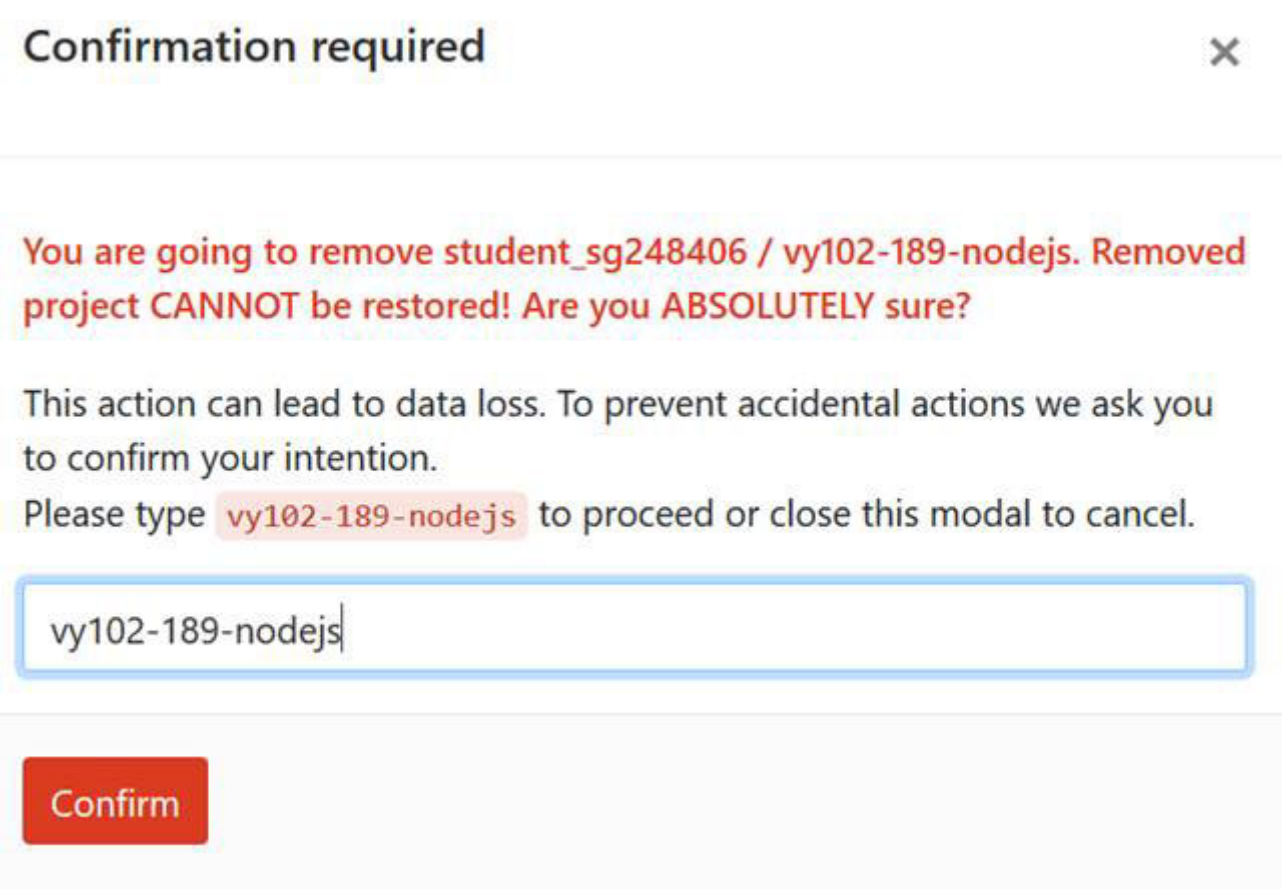
__ 3. Click **Settings** in the left bar.

__ 4. Scroll down and click **Expand** in the **Advanced** section**.**

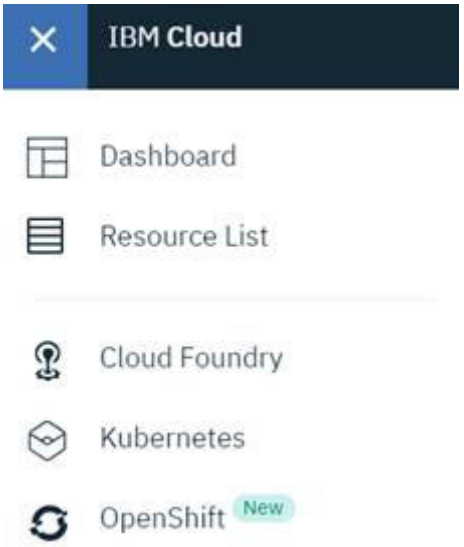


__ 5. Scroll-down and click **Remove project** as shown in the figure.

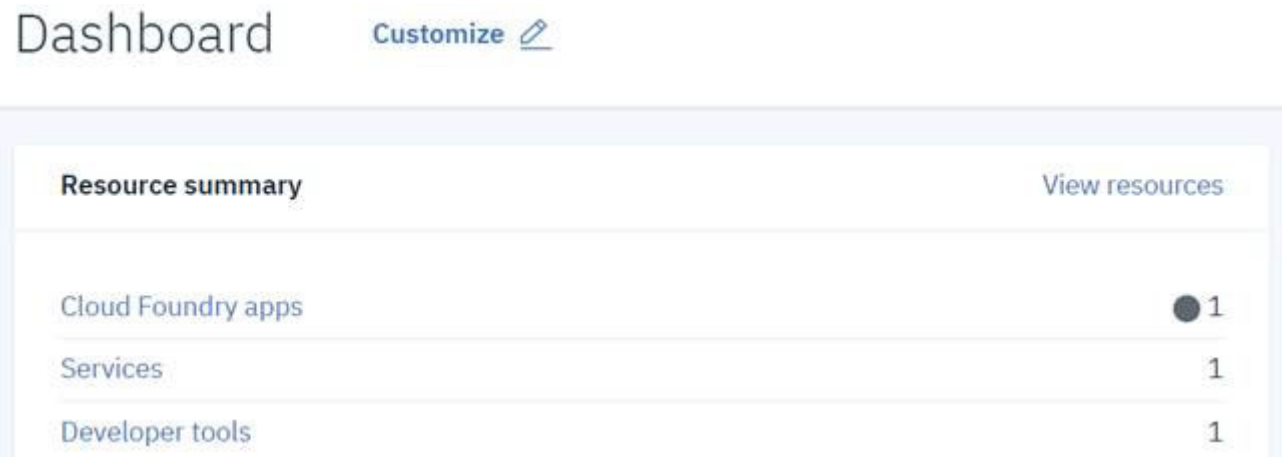
__ 6. In the confirmation window, type your application name and click **Confirm** to delete the application repository



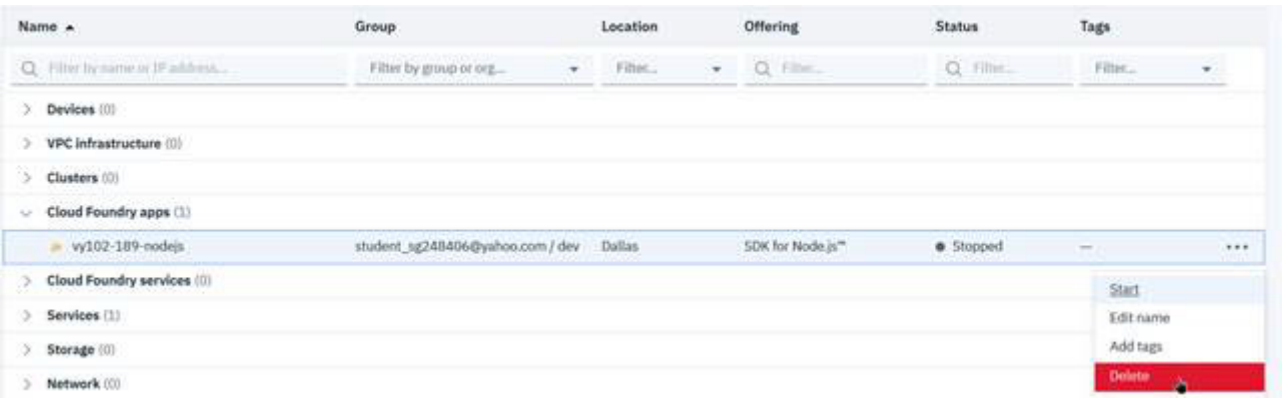
__ 7. Click the icon on the upper left of the screen, and then click Dashboard as shown in the following figure:



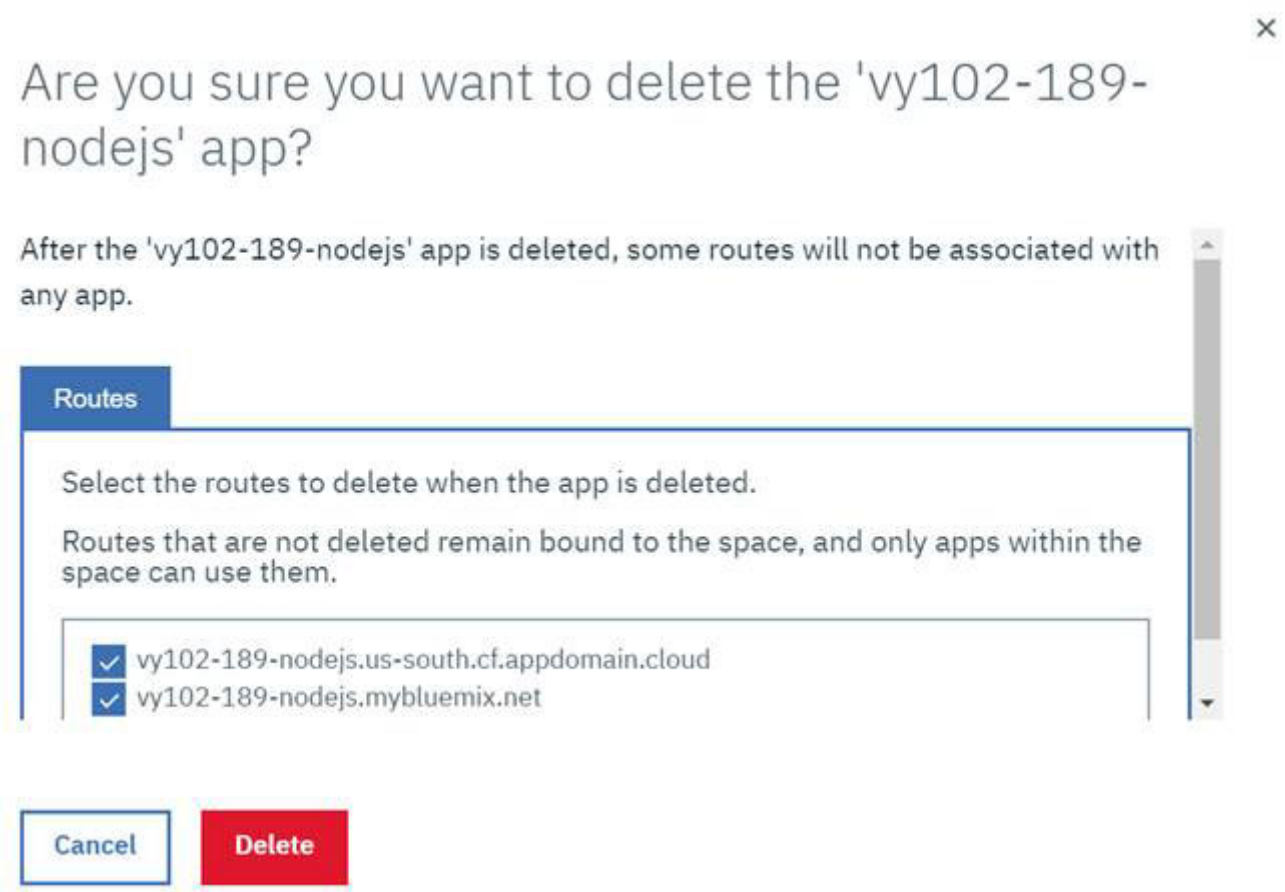
__ 8. Click **Cloud Foundry apps**, as shown in the following figure.



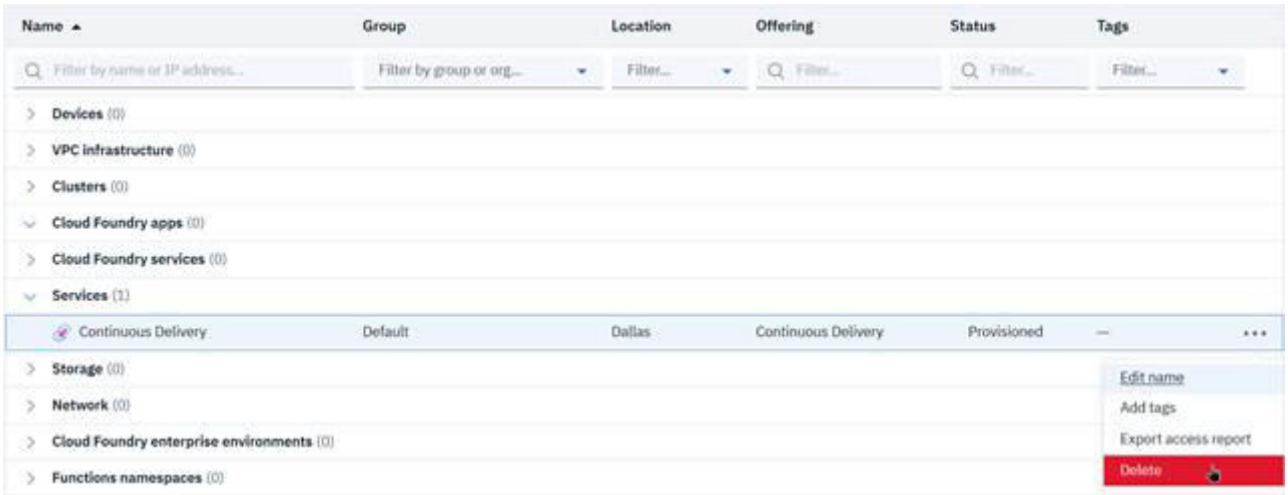
__ 9. Expand the **Cloud Foundry apps** section and click **Actions** (three dots) to the right of your Node.js app to list the available actions. Click **Delete**, as shown in the following figure.



__ 10. In the confirmation window, select the routes, and then click **Delete**, as shown in the following figure.



__ 11. Expand the **Services** section and delete the **Continuous Delivery** service that was associated with your app. In the list of actions, click **Delete**, as shown in the following figure.



__ 12. In the confirmation window, click **Delete** to confirm.

End of exercise

Exercise review and wrap-up

In this exercise, you accomplished the following goals:

- Created a Node.js App on the IBM Cloud environment.
- Used the Continuous Delivery Toolchain to develop and deploy the Node.js app.
- Wrote your first Node.js “Hello Node.js!” application.
- Created a module in Node.js to wrap and use functions from other JavaScript files.

Troubleshooting

For troubleshooting any issues, refer to the full code of this exercise at the following link:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp

The following link leads to the code that is explained in Part 4. “Creating a Hello World Node.js server”:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp/Ex1/1-HelloWorld-Node.js-Server

The following URL leads to the code explained in Part 5. “Adding a module to the Node.js application”:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp/Ex1/2-Add-Module

Exercise 2. Understanding asynchronous callback

Estimated time

01:30

Overview

This exercise describes two important concepts: Asynchronous method invocation and callbacks.

- Asynchronous method invocation

  This is a design pattern where an invoker method is not blocked while it is waiting for the invoked method to finish processing and return a result. Instead, the invoked method is run in a separate thread and the invoker is notified when the result is ready.

- Callback function

  This is a function that is passed to another function as a parameter, and the callback function is called and run within this other function.

An asynchronous callback has considerations that differ from a synchronous callback. This exercise describes those considerations.

This exercise shows how to use callback functions to call an external service. This exercise uses the IBM Watson Language Translator service in IBM Cloud. You create a Node.js module that contains the logic for these calls.

Objectives

By the end of this exercise, you should understand asynchronous callbacks and be able to write the code in a Node.js application.

Introduction

In this exercise, you use asynchronous callback functions.

A callback function is a function that is passed as a parameter to another function. When this other function runs, the callback function is run somewhere inside this other function.

Consider the following example:

```
setTimeout(function() {
    console.log("A");
}, 3000);

setTimeout(function() {
    console.log("B");
}, 2000);

setTimeout(function() {
    console.log("C");
}, 4000);

setTimeout(function() {
    console.log("D");
}, 1000);
```

The **setTimeout** function in this example is a function that receives the following parameters:

- A callback function that must run after a certain time interval.
- The time interval (in milliseconds) that must elapse before the callback function can run.

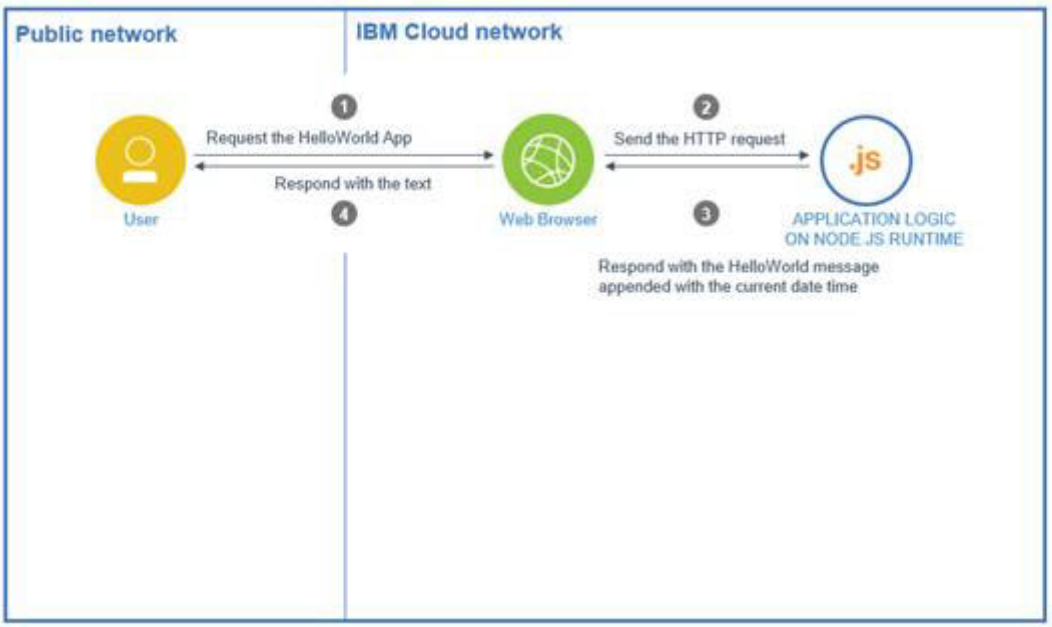
When the **setTimeout** function is called, Node.js does not wait for the processing to finish. Instead, Node.js registers the callback function for it to be run after the response is returned.

When the code in the example runs, the following output is returned in the console:

```
D
B
A
C
```

The result depends on the sequence in which the **setTimeout** functions finish their execution and not on the sequence of their declaration in the JavaScript file.

The architecture of the asynchronous callback in this exercise is shown in the following figure.



This exercise shows how to use callback functions in your Node.js application. In this scenario, the Node.js app accesses the Watson Language Translator service in IBM Cloud. The flow that is shown in the architecture diagram is as follows:

__ 1. The user enters the application's URL in the web browser.

__ 2. The web browser sends the HTTP request to the Node.js app that is deployed in IBM Cloud.

__ 3. The Node.js app asynchronously calls the Watson Language Translator service and registers a callback function that is called when the Watson Language Translator service sends the response.

__ 4. The Node.js app sends an HTTP request to the Watson Language Translator service on IBM Cloud that is exposed as a REST API.

__ 5. The Watson Language Translator service responds to the HTTP request with the requested data (translated text).

__ 6. The callback function (from step 3) now runs. This function responds to the HTTP request (from step 2).

__ 7. The Node.js app sends the data to the web browser in the HTTP response.

__ 8. The web browser displays a web page that shows the data to the user.

Requirements

Before you start, be sure that you meet these prerequisites:

- Basic knowledge of JavaScript
- An IBM Cloud account.
- A workstation that has these components:
  - Internet access
  - Web browser: Google Chrome or Mozilla Firefox
  - Operating system: Linux, Mac OS, or Microsoft Windows

Exercise instructions

In this exercise, you complete the following tasks:

__ 1. Log in to your IBM Cloud account.

__ 2. Create the Node.js application on IBM Cloud.

__ 3. Enable continuous delivery.

__ 4. Integrate the Node.js app with the Watson Language Translator service.

__ 5. Access the Watson Language Translator service from the Node.js app.

__ 6. Access the Watson Language Translator service through a Node.js module.

__ 7. Stop and delete the application.

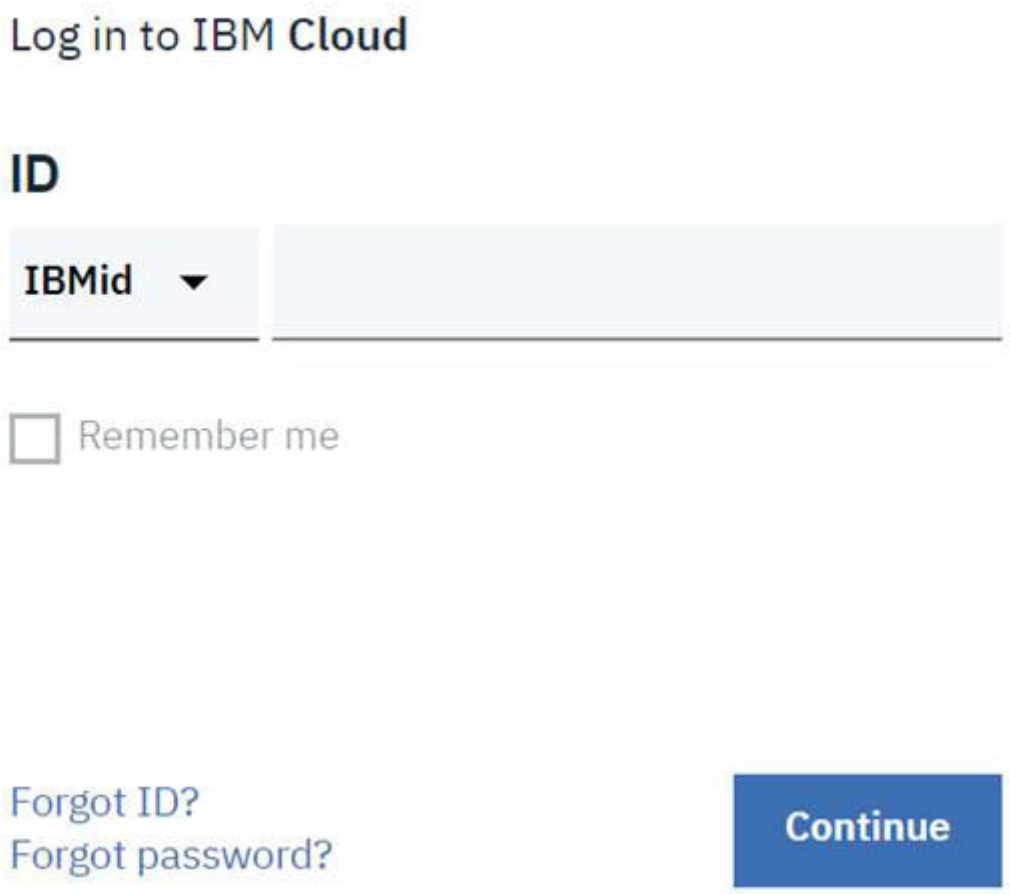
Part 1: Logging in to your IBM Cloud account

To log in to IBM Cloud, complete the following steps:

__ 1. Open your web browser and enter the following web address:

https://cloud.ibm.com

__ 2. The IBM Cloud login page opens (IBM Cloud login), as shown in the following figure. Enter your IBMid (which is the email with which you signed up). Click **Continue**.

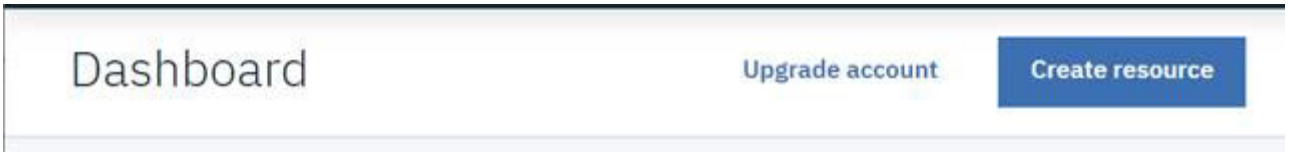


__ 3. Enter your password, and then click **Log in**.

Part 2: Creating the Node.js application on IBM Cloud

Create the Node.js app by using the SDK for Node.js runtime on IBM Cloud:

__ 1. In the IBM Cloud dashboard, click **Create resource**, as shown in the following figure.

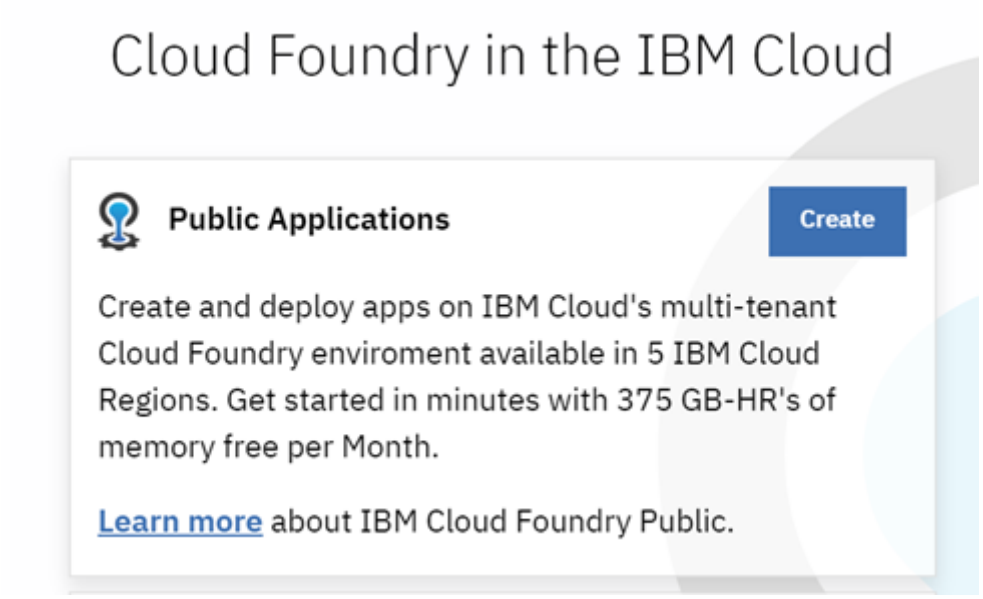


__ 2. The IBM Cloud Catalog page opens. It lists the infrastructure and platform resources that can be created in IBM Cloud. In the search field, enter **Cloud Foundry**.

__ 3. Select **Compute** from the left side menu as shown in the following figure.

__ 4. Select Cloud Foundry under Compute, as shown in the following figure.

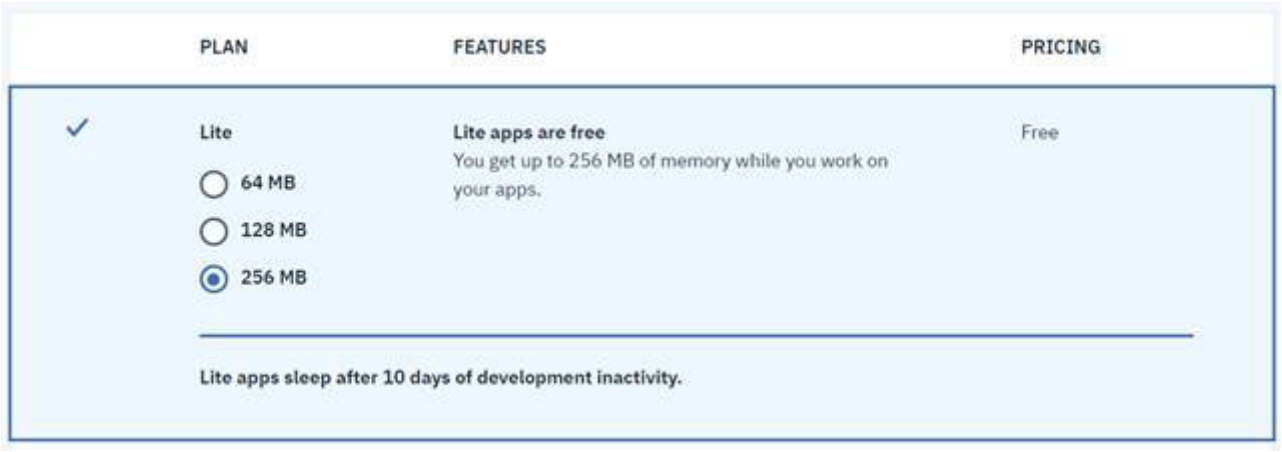
__ 5. From the Cloud Foundry overview page, click **Create** for Public Applications as shown in the following figure.



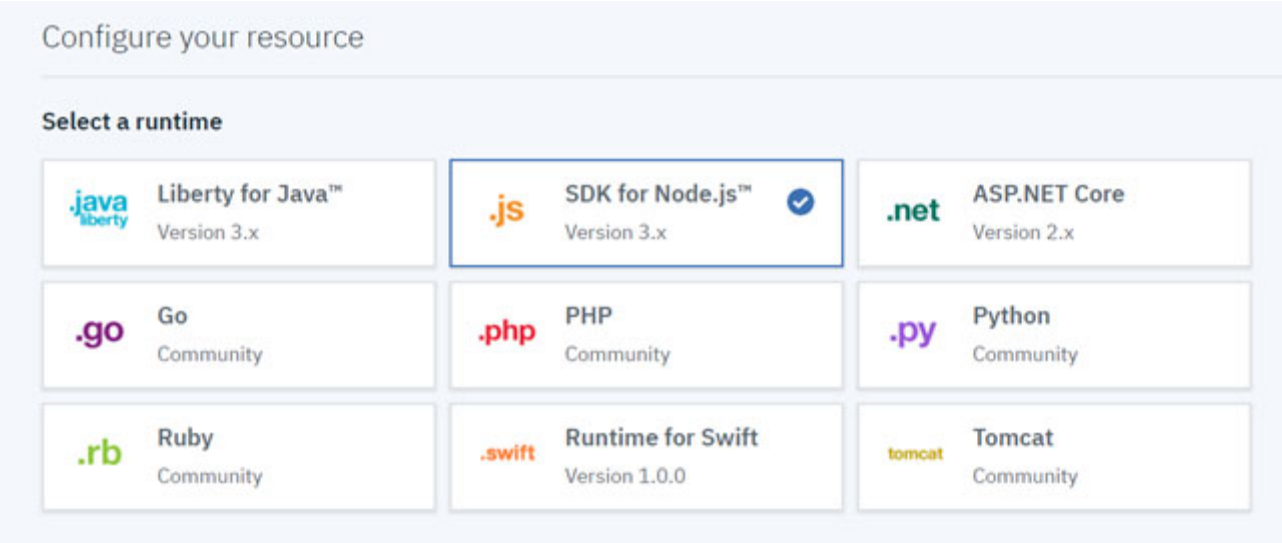
__ 6. Complete the application details to create the Cloud Foundry sample app:

__ a. The region is selected by default according to your location.

__ b. If you do *not* have a Lite account, skip this step. For Lite accounts, Pricing Plans show that the memory that is allocated to your app by default is 64 MB. For this exercise, select the maximum allocation of **256 MB**, as shown in the following figure.



__ c. Select **SDK for Node.js** from the provided runtimes as shown in the following figure.



__ d. In the App name field, enter `vy102-`*`XXX`*`-nodejs`. Replace XXX with three random characters to make your application name unique. You use this unique key in the naming convention for this exercise.

__ e. The host name is set by default to the app name.

__ f. The domain is chosen according to your location.
Select a domain from the “Domain” drop-down list that has the subdomain `{region}.cf.appdomain.cloud`.

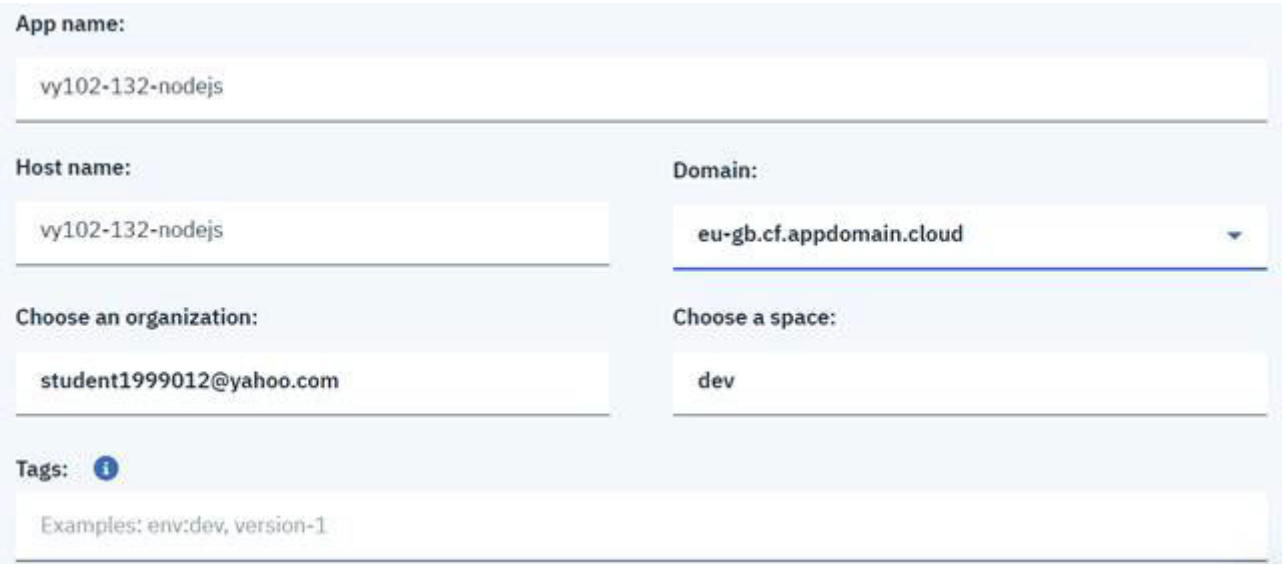
Note

Make sure that the domain that is selected by default has the format `{region}.cf.appdomain.cloud`. Do not choose “*`mybluemix.net`*” as a domain because it is deprecated.

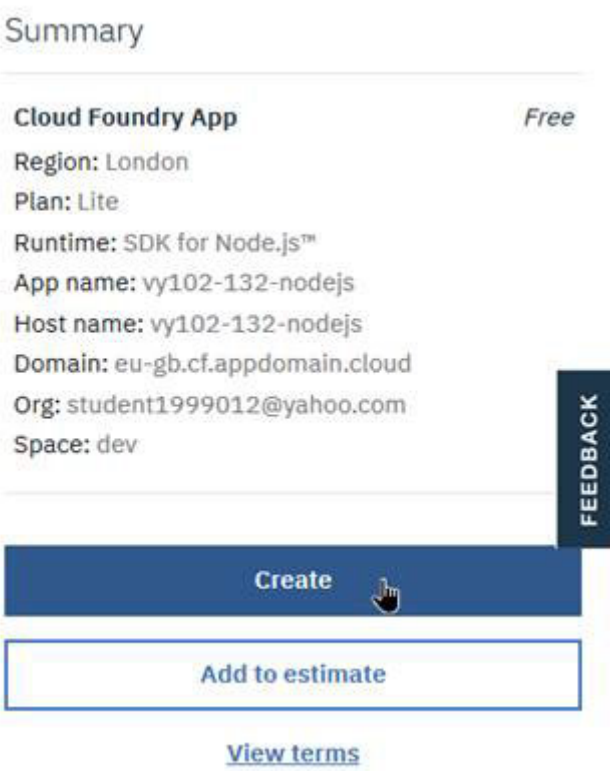
For example, if your region is *London*, the domain is set to London's domain: `eu-gb.cf.appdomain.cloud`.

__ g. The organization is set by default to the email that you use to log in to IBM Cloud.

__ h. The space is set by default to dev, as shown in the following figure.

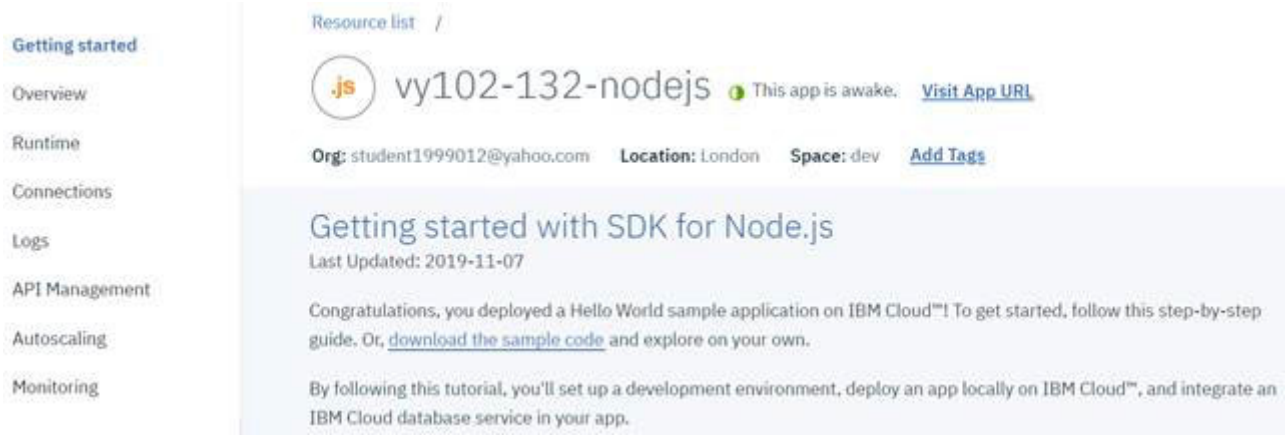


__ i. Click **Create**. IBM Cloud proceeds to deploy your application. Your application stages and deploys in a few minutes.

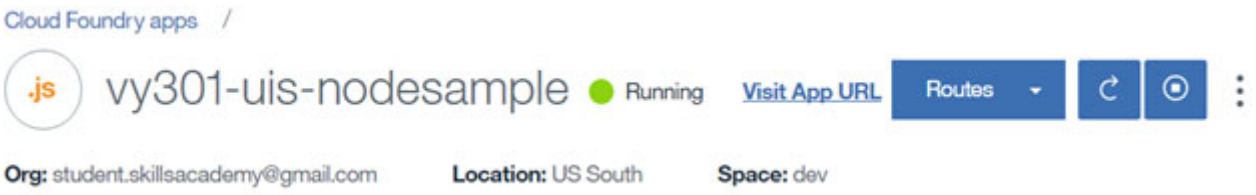


Stop

Wait until the application finishes staging and it is running in IBM Cloud before you proceed to the next step. For Lite accounts, wait for the application status “This app is awake”, as shown in the following figure.



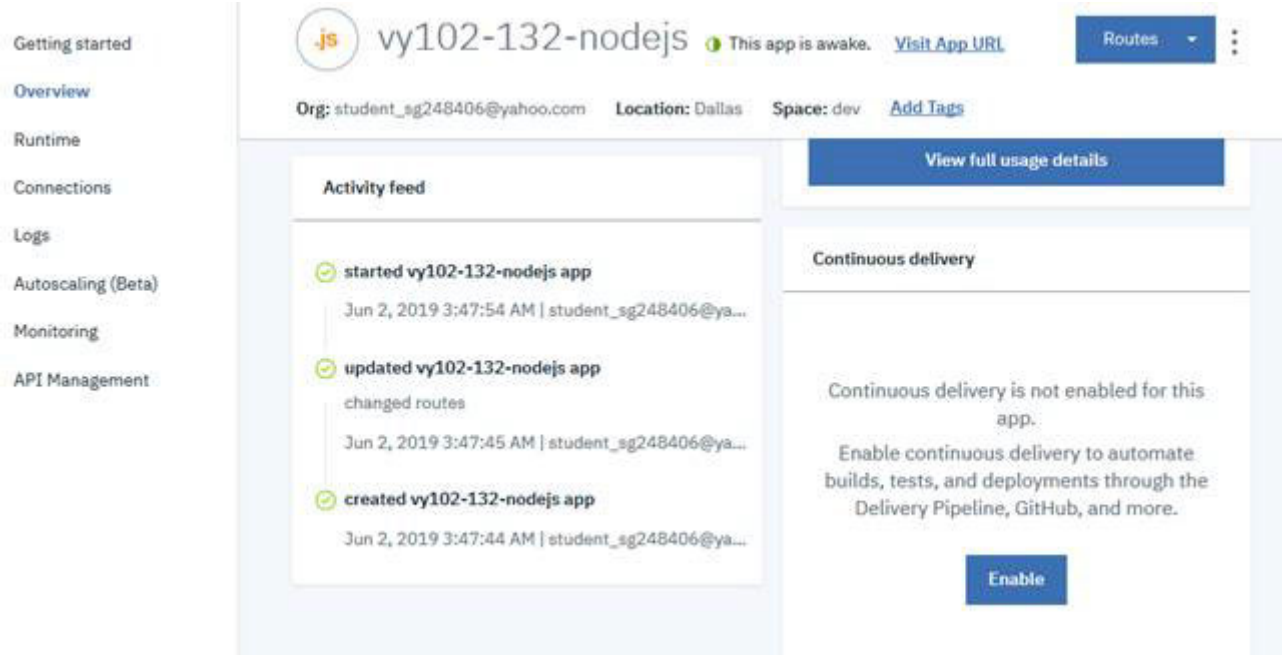
If you do not have a Lite account, the application status should be “Running”, as shown in the following figure.



Part 3: Enabling continuous delivery

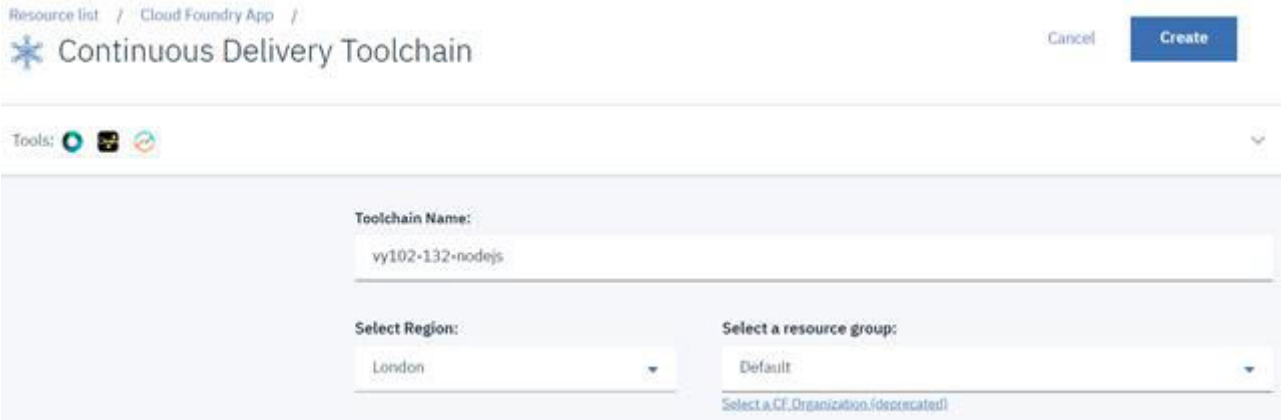
Enable continuous delivery for the Node.js app by completing these steps:

__ 1. Click **Overview** in the left pane, scroll down to the **Continuous delivery** tile, and then click **Enable**, as shown in the following figure.



__ 2. A new Continuous Delivery Toolchain tab opens. This toolchain includes tools to develop and deploy the application.

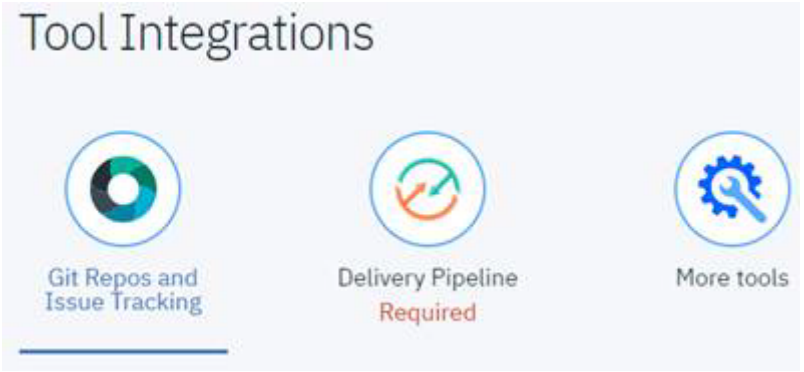
The Toolchain Name field is automatically populated. Keep the default values for both the **Select Region** and **Select a resource group** fields, as shown in the following figure.



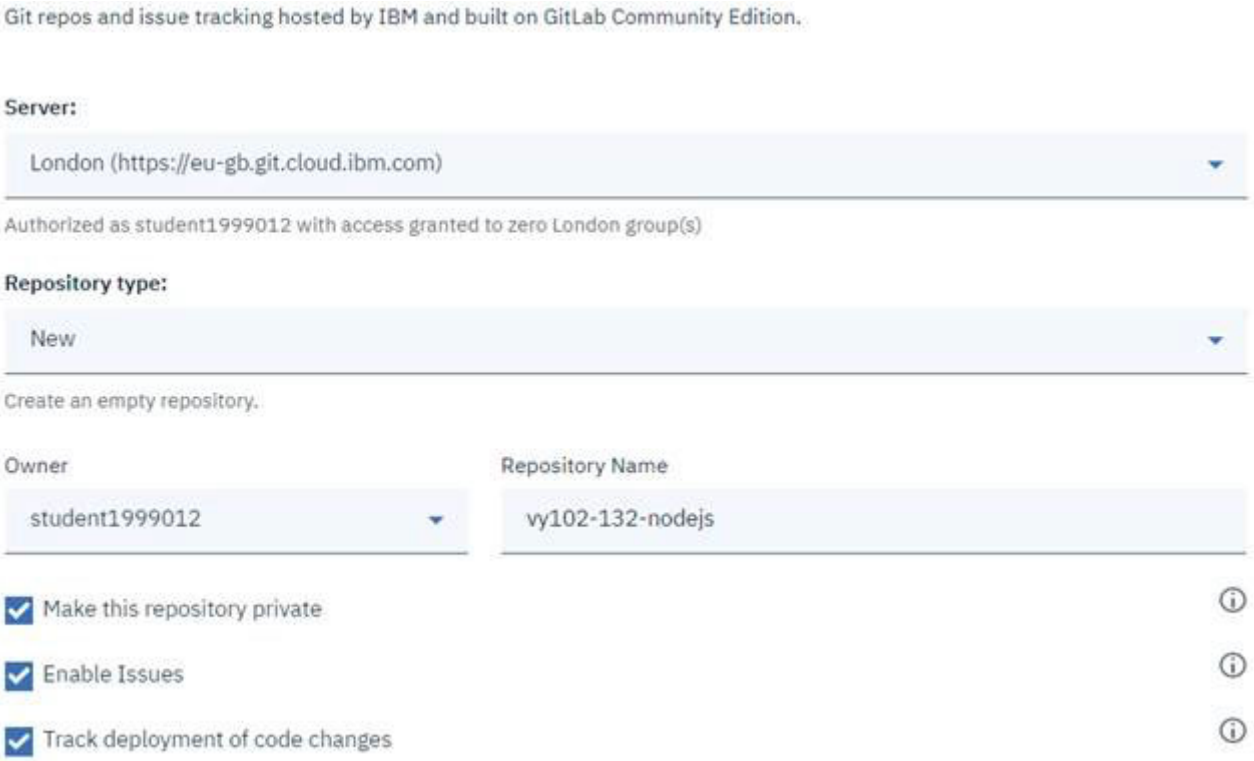
__ 3. Scroll down to see the three main tabs under Tool Integrations:

- Git Repos and Issue Tracking.
- Delivery Pipeline.
- More Tools.

The Git Repos and Issue Tracking icon is selected by default, as shown in the following figure.



The Git Repos and Issue Tracking section is shown as the following figure.



In the Repository type field menu you can select from the following options:

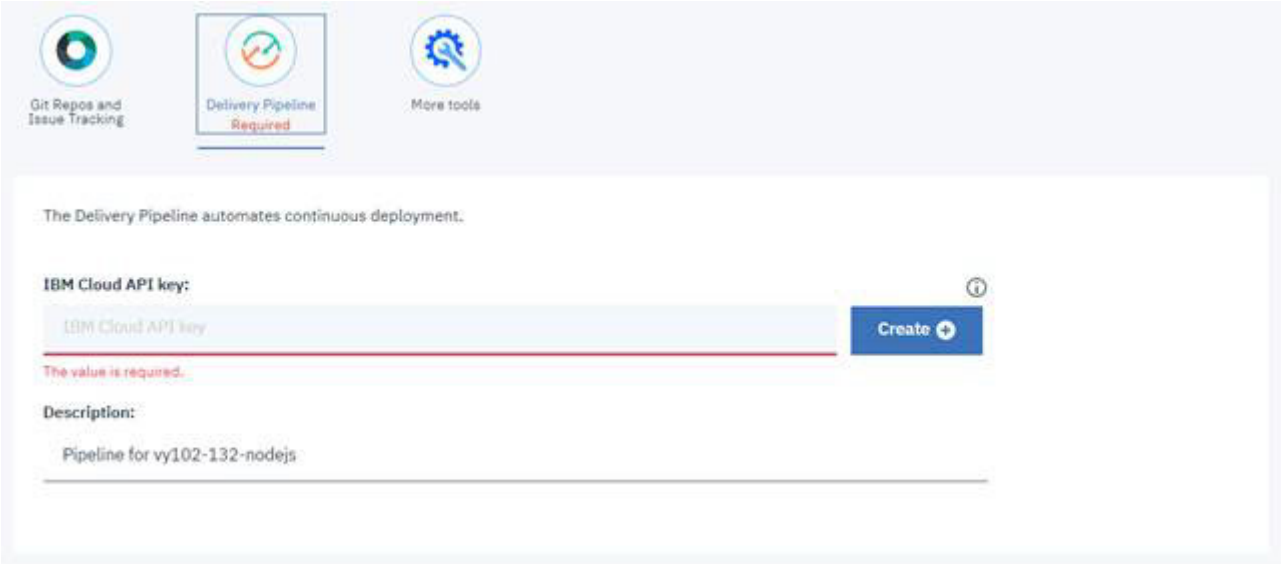
- New: Create an empty new repo.
- Fork: fork an existing GitLab repo (you specify its URL) so that you can contribute changes through merge requests
- Clone: Create a copy of an existing GitLab repo
- Existing: Link to an existing repository and continue working on it.

__ 4. The default selection is Clone. For this exercise, select **New** so that you can start developing your application from scratch.

__ 5. For the other fields, keep their default values and click the **Delivery Pipeline** tab.

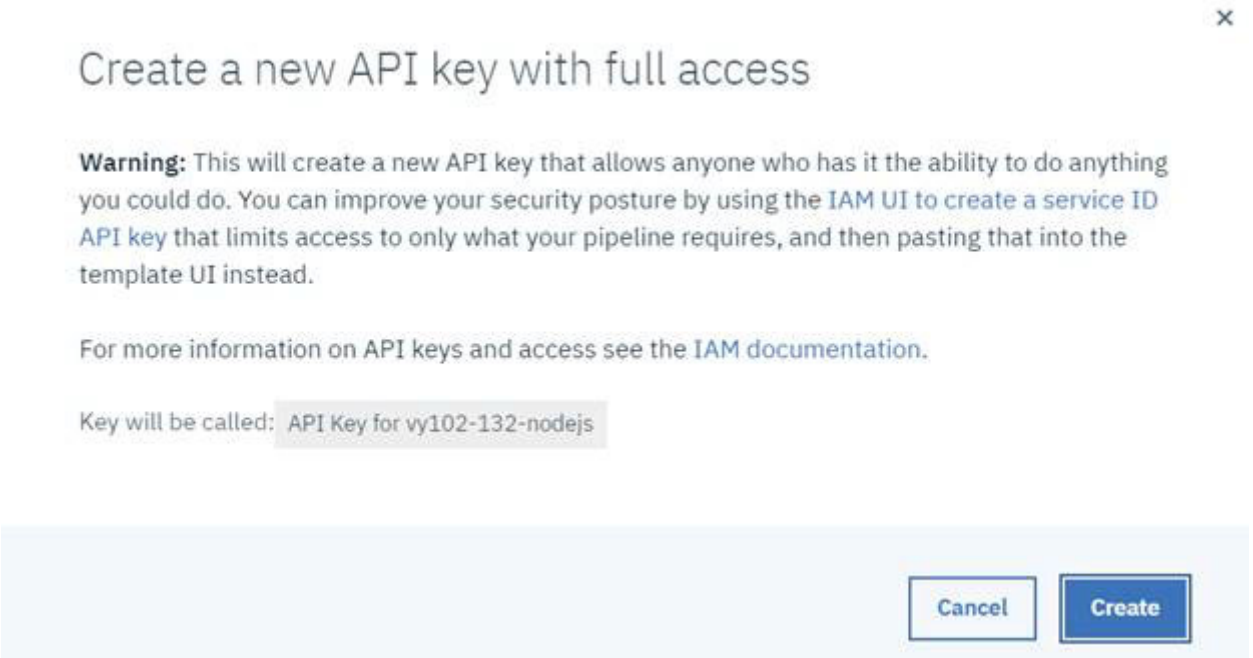
__ 6. In the Delivery Pipeline tab, which is shown in the following figure, two fields are shown:

- IBM Cloud API Key: Required to access IBM Cloud runtime.
- Description: Describes the pipeline that is created in this step.

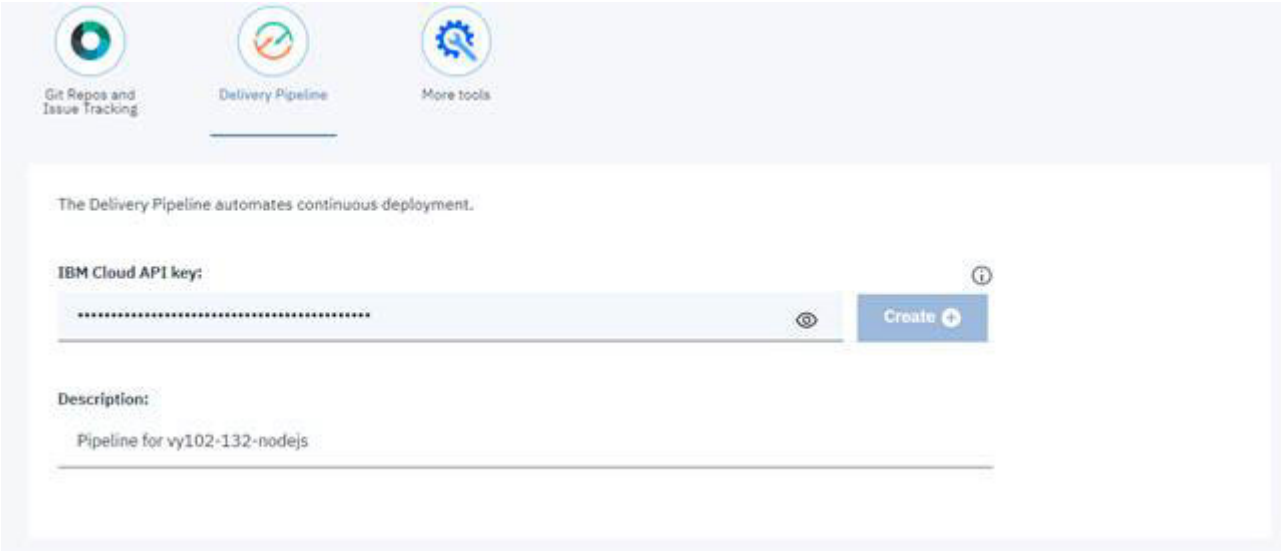


Click **Create +**.

__ 7. A window opens where you can create the API key, as shown in the following figure. Click **Create**.



__ 8. The window closes, and the main page shows the generated IBM Cloud API key, as shown in the following figure.

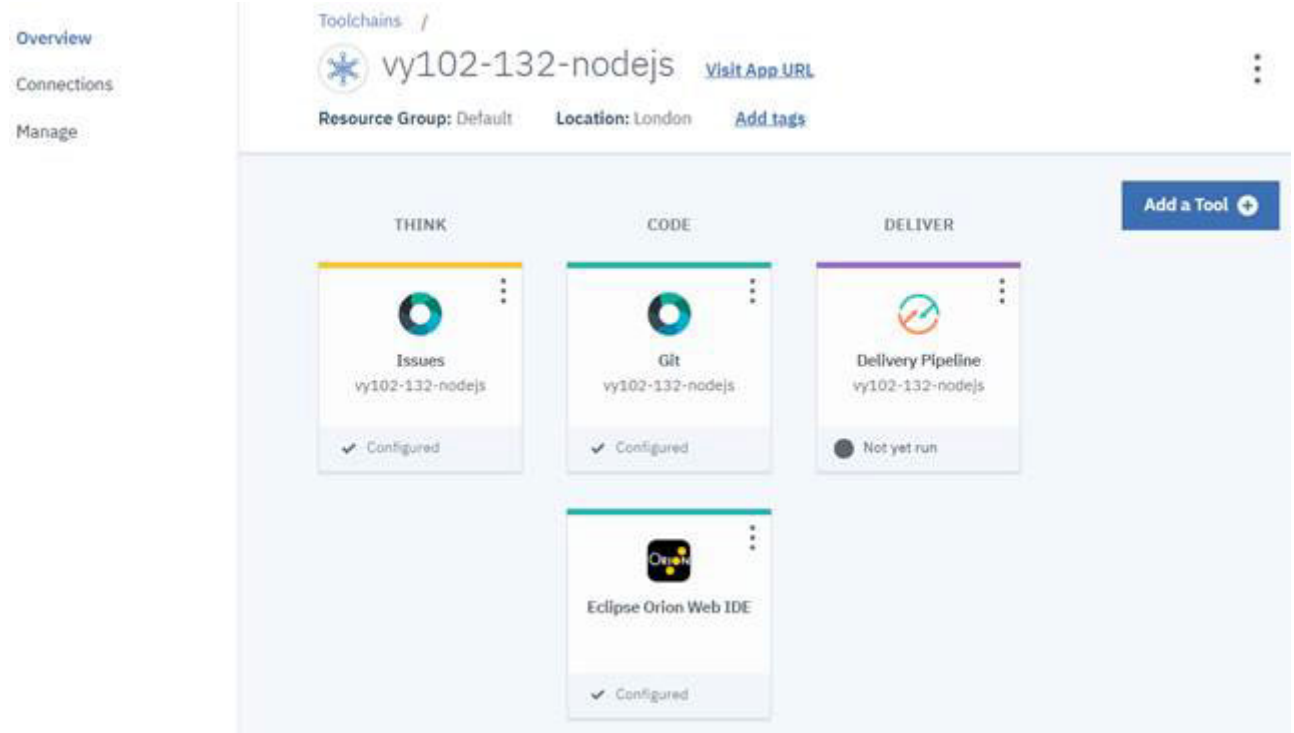


__ 9. Click **Create** at the upper right of the page

__ 10. A new page opens and shows the three main phases (THINK, CODE, and DELIVER). A toolchain is a set of tool integrations that support development, deployment, and operations tasks. The UI to create a new toolchain groups the tools into the following phases:

- **THINK**: In this phase, you plan the application by creating bugs, tasks, or ideas by using the Issue Tracker, which is part of the Git repository.
- **CODE**: In this phase, you implement the application by using a GIT repository as a source code management system, and by using a web IDE (Eclipse Orion) to edit your code online. In the repository, you can specify whether to clone a repository or start from scratch by selecting **New** in the repository type.
- **DELIVER**: In this phase, you configure the delivery pipeline. You can specify automatic build, deployment, and testing of your code after a developer pushes new code to the Git repository.

  The following figure shows these three phases.



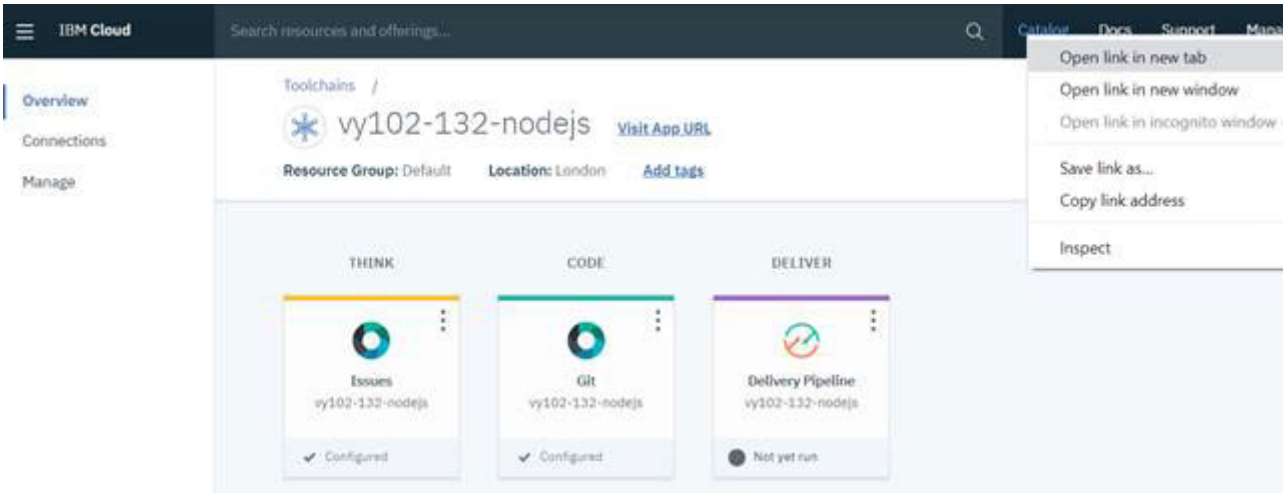
Part 4: Integrating the Node.js app with the Watson Language Translator service

An IBM Cloud service is a ready-for-use function that is hosted on IBM Cloud and can be accessed from your application over the internet. Examples of services are databases, message queues, and many others. In this exercise, you use the Watson Language Translator service.

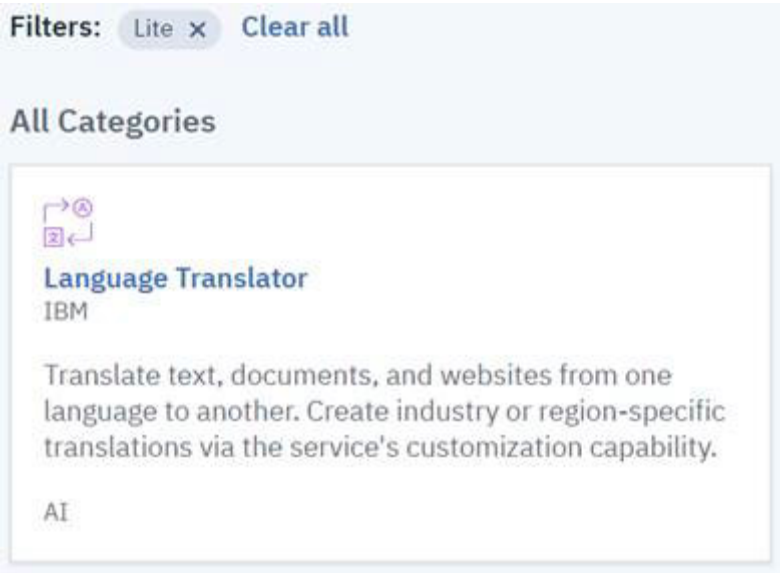
Creating a Watson Language Translator service instance

In the following steps, you create an instance of the Watson Language Translator service that is hosted on IBM Cloud and access this service from your Node.js application:

__ 1. Right-click **Catalog** and select **Open link in new tab**, as shown in the following figure. A Catalog tab opens.

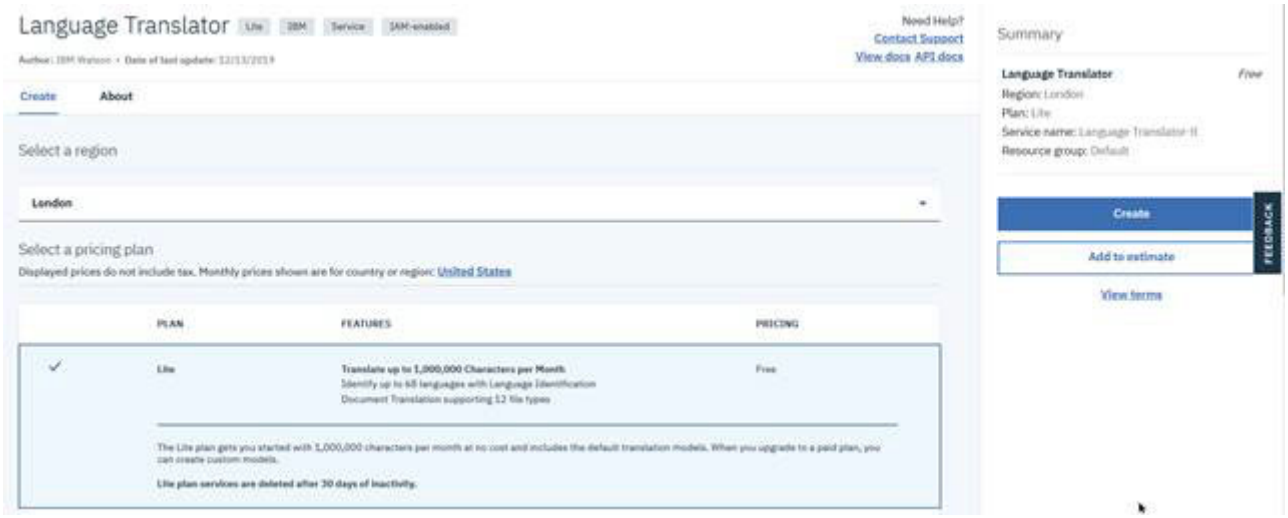


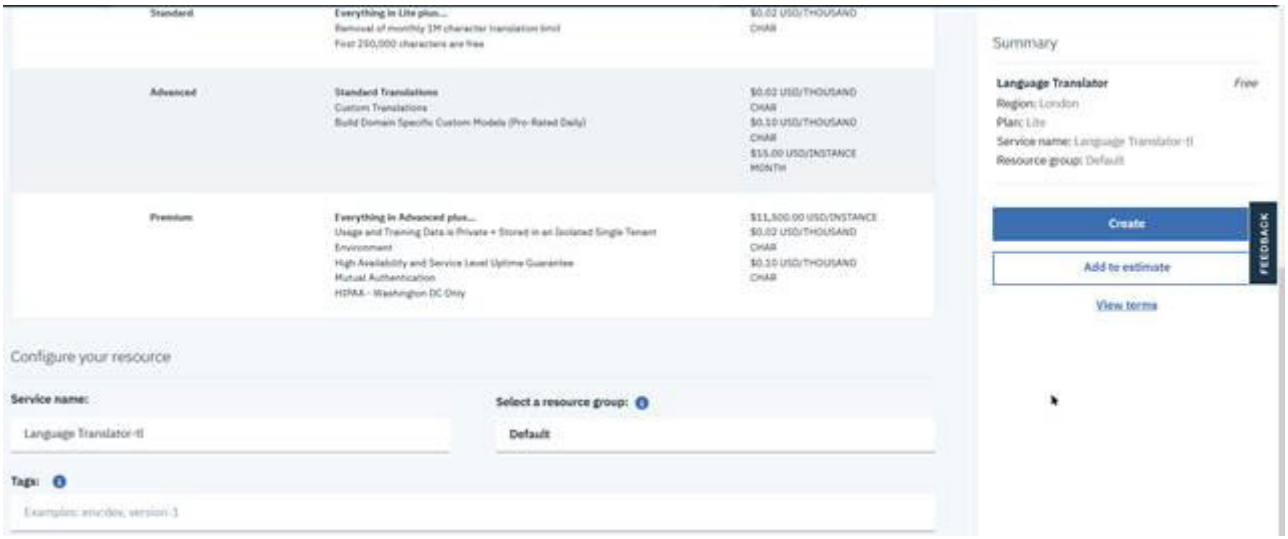
__ 2. Search for the Language Translator service and click it, as shown in the following figure.



Use this service to translate text from one language to another.

__ 3. The Language Translator service page opens, as shown in the following figure. Keep the default values and click **Create**.



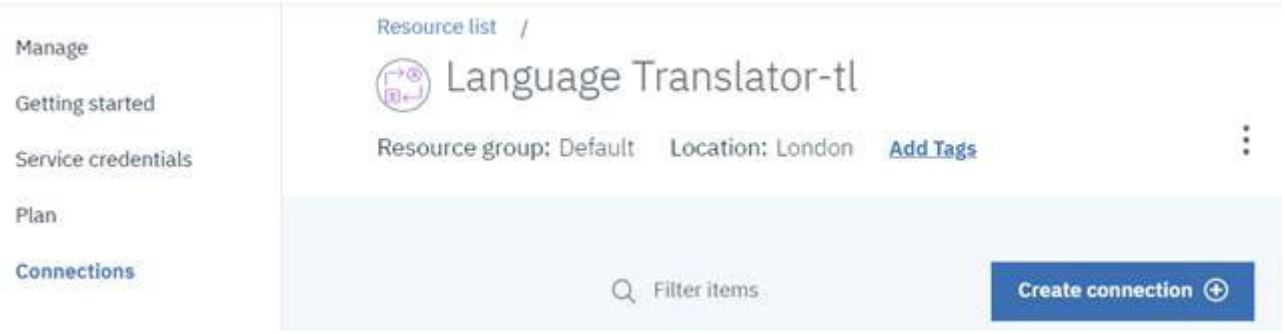


Connecting the Node.js app to the Watson Language Translator service

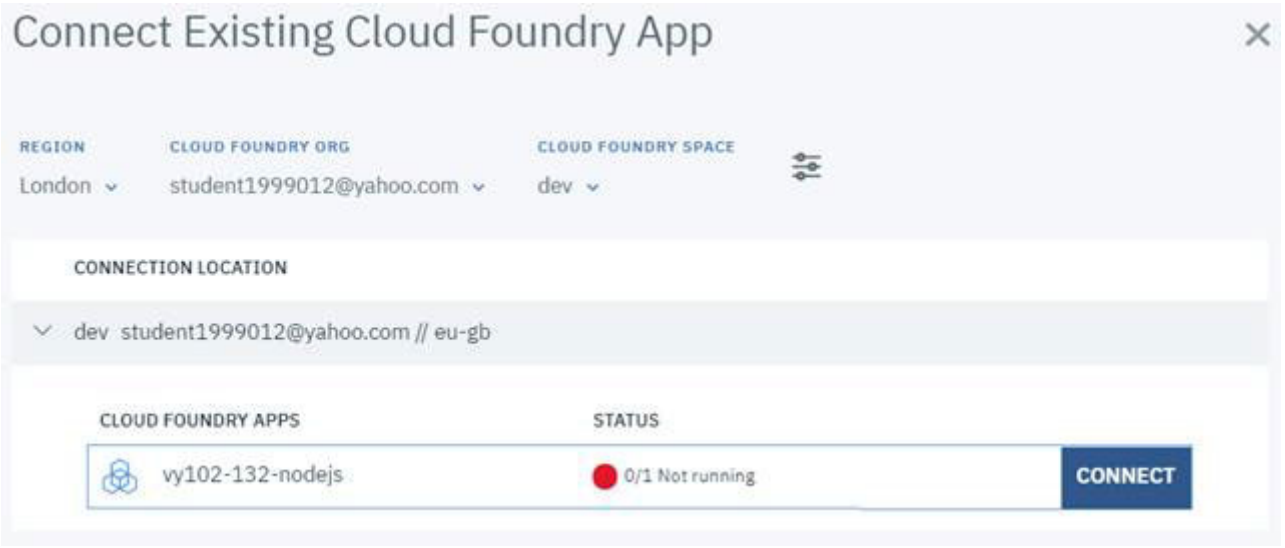
When the Node.js app is connected to the Watson Language Translator service, the connection details of the Watson Language Translator service are added in the environment variables of the Node.js application. Hence, you can read the connection details in the app without hardcoding them in your code.

To connect your Node.js app to the Watson Language Translator service, complete these steps:

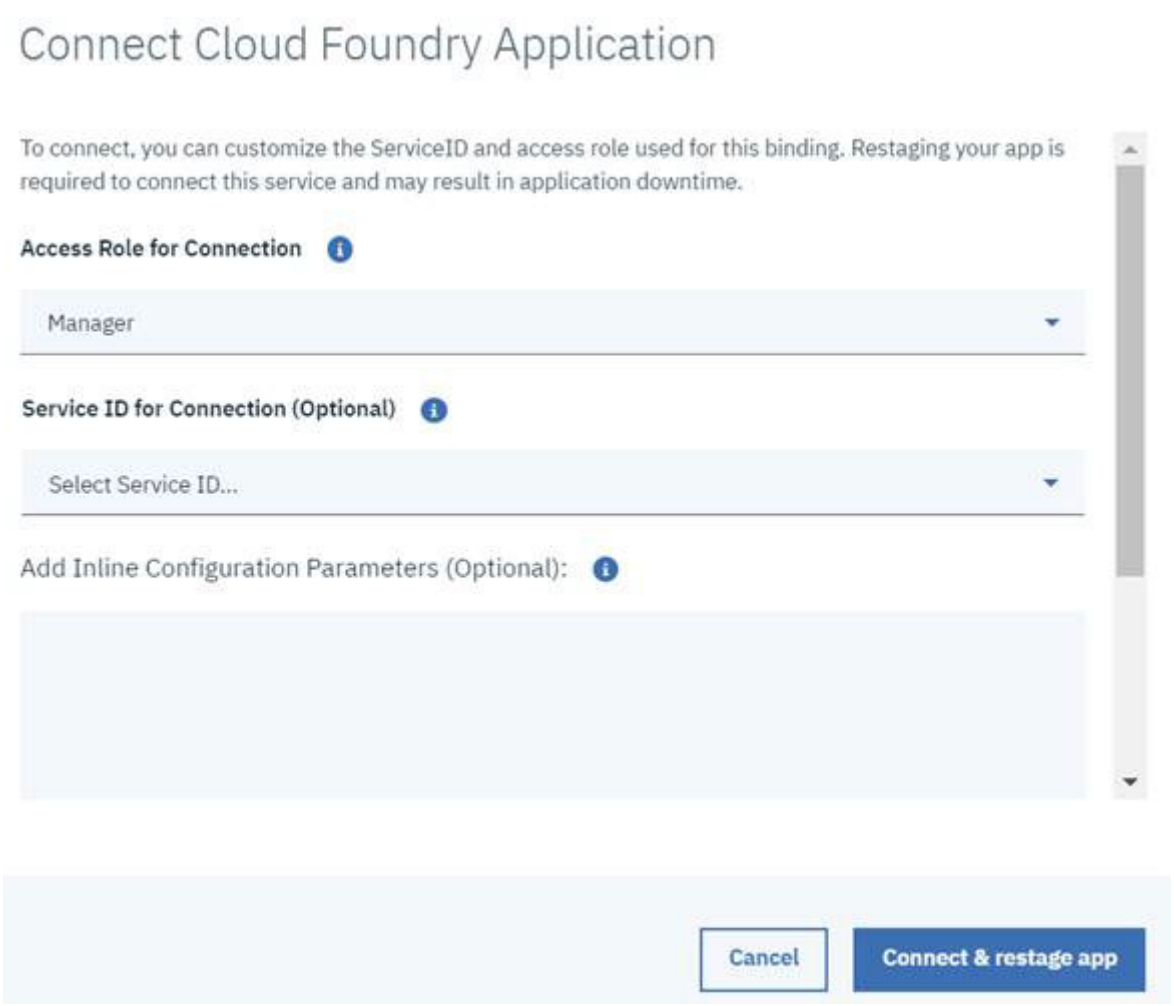
__ 1. Click **Connections** on the left navigation bar, and then click **Create Connection**.



__ 2. From the list, select the Node.js application that you created in Part 2 “Creating the Node.js application on IBM Cloud”, and click **Connect**.



__ 3. The Connect Cloud Foundry Application window opens, as shown in the following figure.

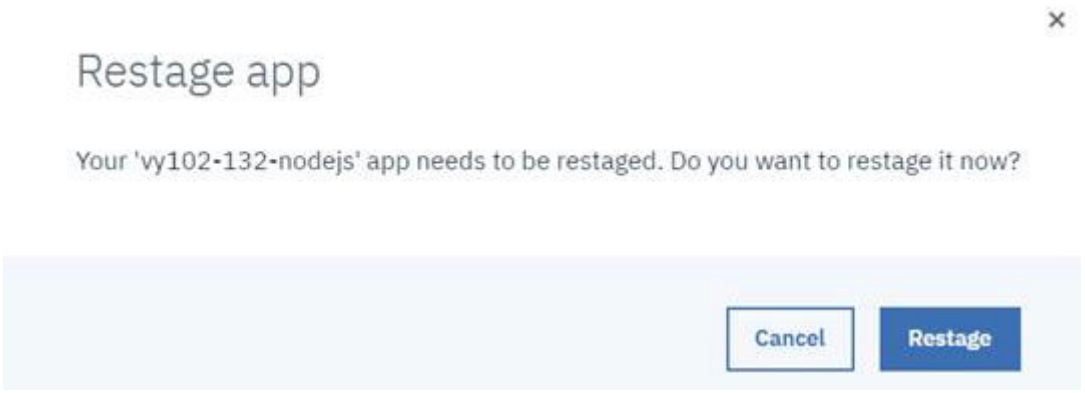


In the Connect Cloud Foundry Application window, you can configure the connection between the Watson Language Translator service and the Node.js application. The following fields are shown:

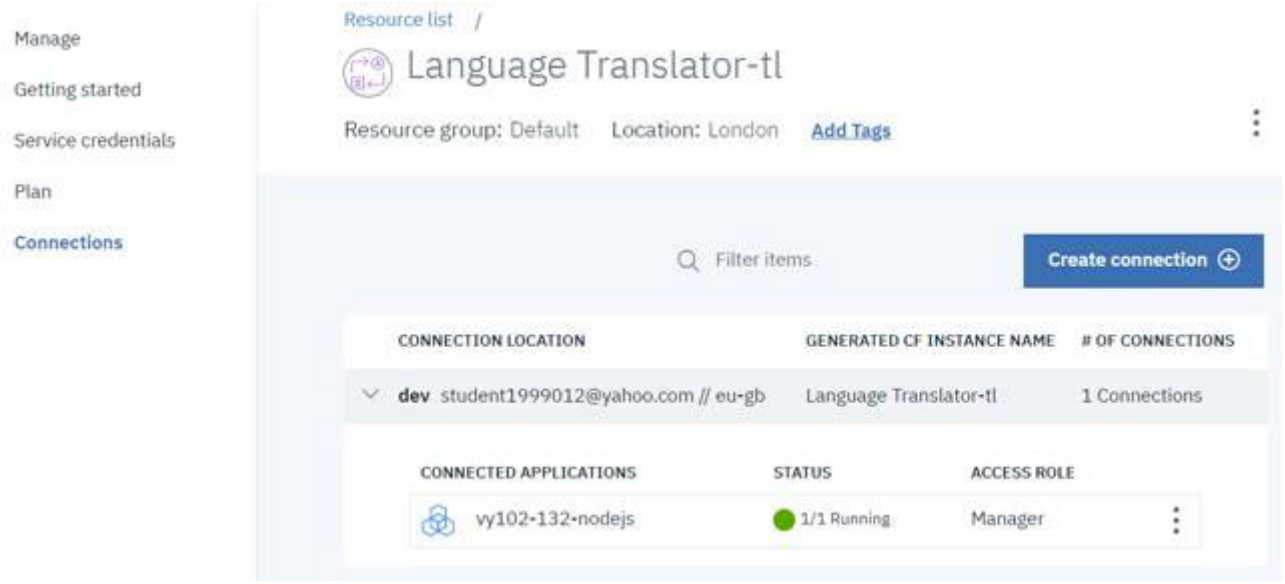
- **Access Role for Connection:** Defines the actions that are permitted when the service is called. The available options are Manager, Writer, and Reader.
- **Service ID for Connection:** Identifies a service or application similar to how a user ID identifies a user.
- **Add Inline Configuration Parameters:** Used when you need to provide service-specific configuration parameters. These parameters are set in a valid JSON format.

Keep the default values and click **Connect & restage app**.

__ 4. A message prompts you to restage the application so that it can use the Watson Language Translator service, as shown in the following figure. Click **Restage**.



__ 5. After the application is restaged, a window similar to the following figure is displayed.

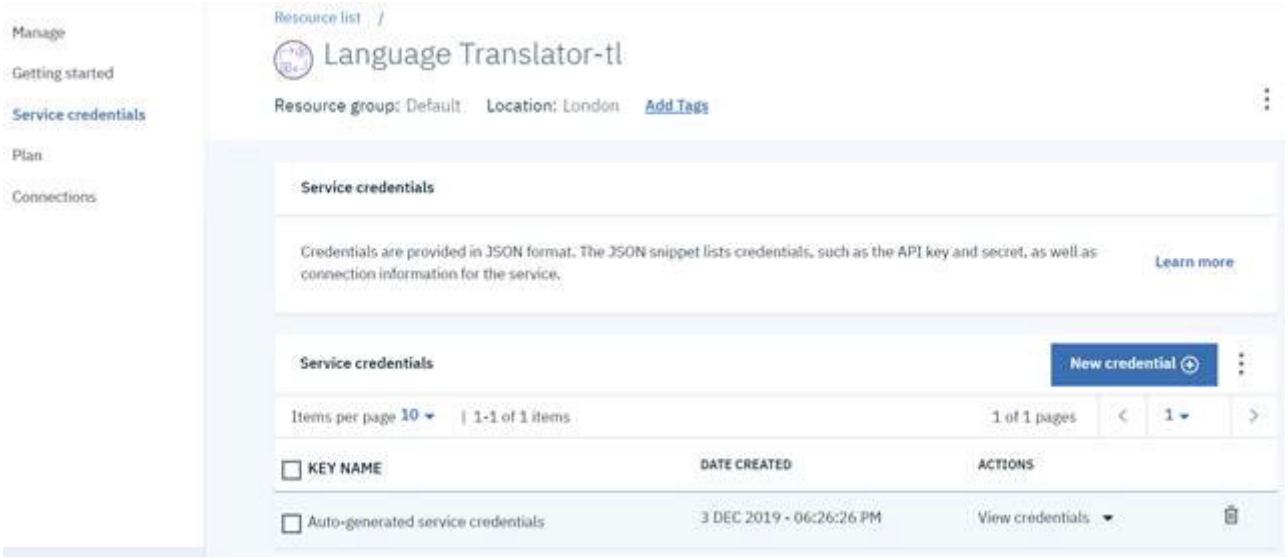


Obtaining the Watson Language Translator service credentials

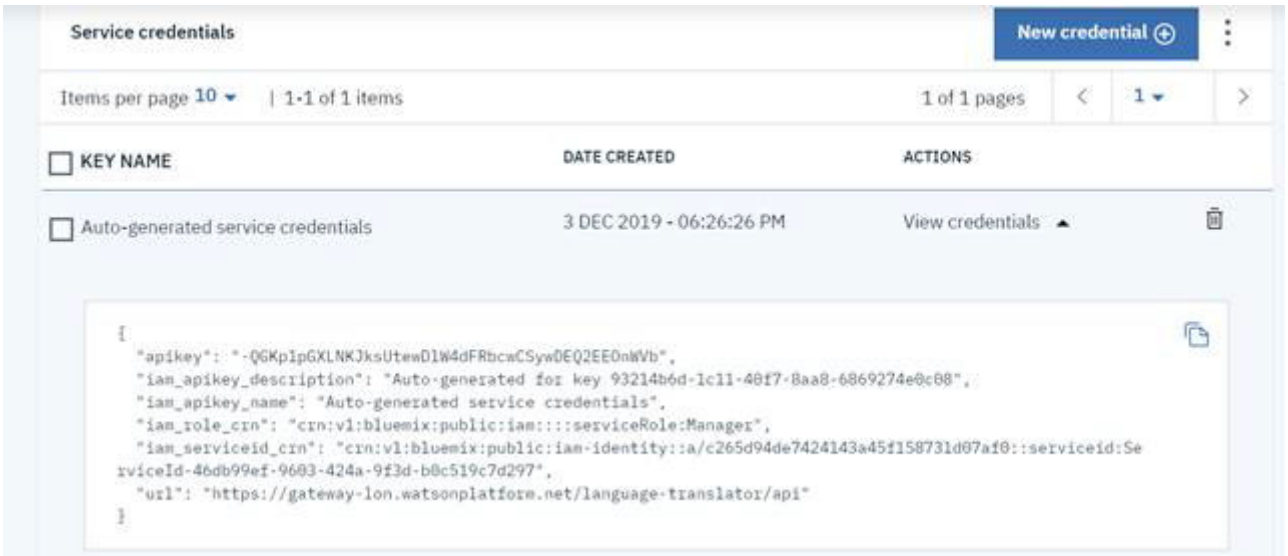
The service credentials are required to access the service. To create and obtain the credentials for your Watson Language Translator service instance, complete these steps:

__ 1. In the Language Translator service details page, click **Service credentials** on the left navigation bar.

__ 2. The Service Credentials screen opens with the “Auto-generated service credentials” section, as shown in the following figure.



__ 3. Click **View credentials**. The following figure shows a JSON object that contains the service credentials. These credentials are needed to access the Watson Language Translator service instance. The fields of the JSON object are used later in the exercise when the Node.js application is implemented to access the Watson Language Translator service.



__ 4. Close the Service Details browser tab.

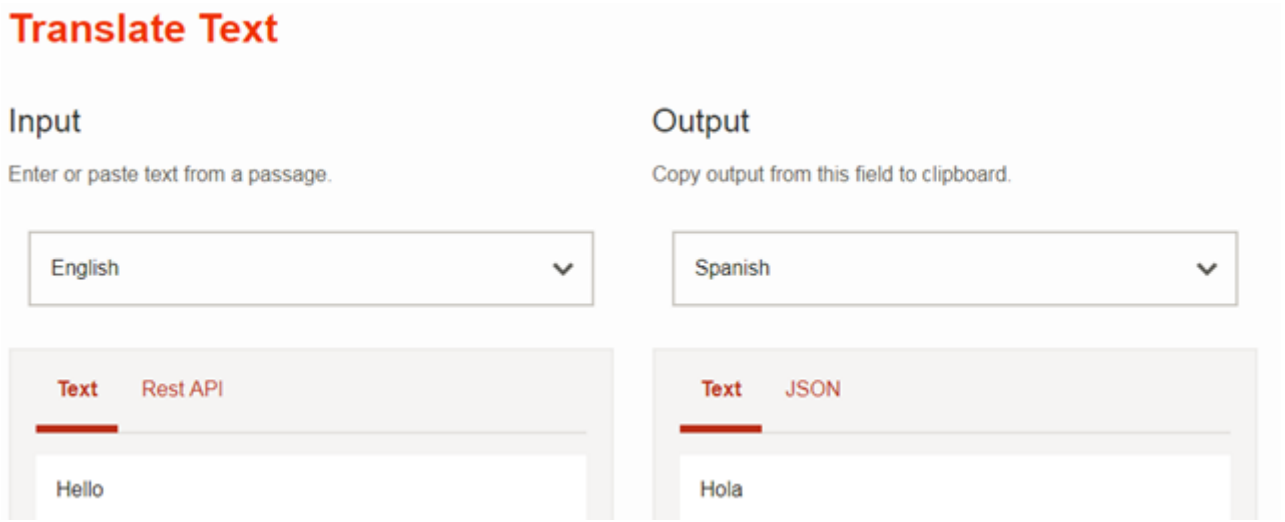
Understanding the Watson Language Translator REST API

The Watson Language Translator service provides REST APIs, which can be called by the applications that are linked to it. The following steps walk you through a demonstration that shows you how the Watson Language Translator REST API works:

__ 1. Open a new web browser tab. Use the following web address to open the Watson Language Translator for IBM Watson Developer Cloud API:

https://language-translator-demo.ng.bluemix.net/

__ 2. Scroll down to the Translate Text section. Select **English** for the input language, and **Spanish** for the Output language. Then, write `Hello` in the Text input field. The Text output field shows the translation as `Hola,` as shown in the following figure.



__ 3. In the Output section, click **JSON**. The JSON object that is retrieved from the API call is shown in the following code snippet:

```
{
{
  "translations": [
    {
      "translation": "Hola"
    }
  ],
  "word_count": 1,
  "character_count": 5
}
```

Note

The `translations.translation` field contains the translated text result. You learn how to read the content of this field in a Node.js app in Part 5 “Accessing the Watson Language Translator service from the Node.js app”.

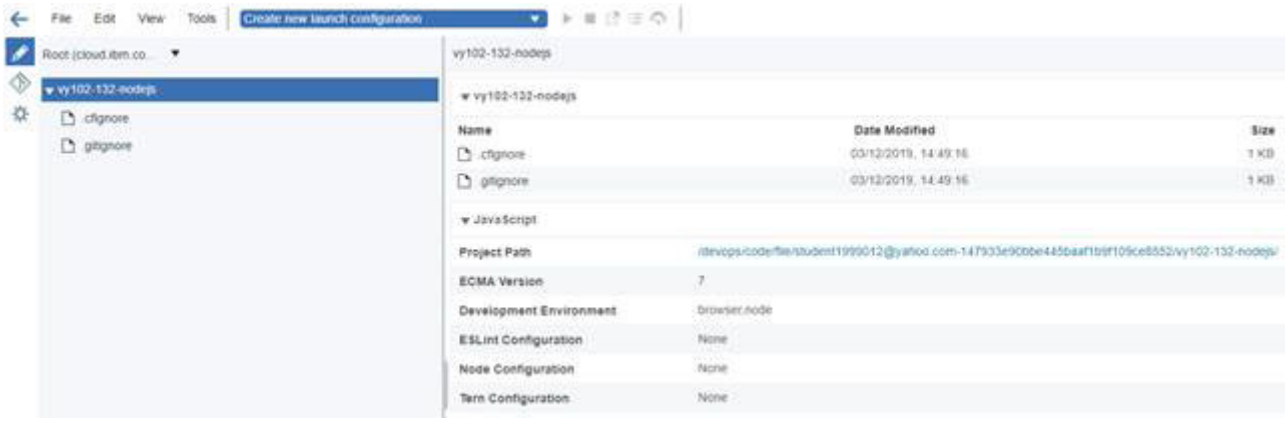
__ 4. Close the opened tab of the web browser.

Part 5: Accessing the Watson Language Translator service from the Node.js app

This section describes the steps to access the Watson Language Translator service from a Node.js application.

__ 1. On the View toolchain page, click the **Eclipse Orion Web IDE** icon.

The browser shows the generated Node.js project in the Eclipse Orion Web IDE, as shown in the following figure.



__ 2. Right-click the root of the project (named vy102-XXX-nodejs), and then select **New** > **File**. A text field appears. Type `manifest.yml` and then press **Enter**. The `manifest.yml` file is now created.

__ 3. Add the following code snippet to the `manifest.yml` file. Replace the XXX with your unique key (the three random characters that you used for the Node.js application name).

```
applications:
- path: .
memory: 128M
instances: 1
name: vy102-XXX-nodejs
host: vy102-XXX-nodejs
disk_quota: 1024M
```

__ 4. Create a file and name it package.json. Insert the following code snippet into the package.json file.

```
{
    "name": "NodejsStarterApp",
    "version": "0.0.1",
    "description": "App for understanding Callback",
    "scripts": {
        "start": "node app.js"
    }
}
```

Click **File**, and then **Save**.

__ 5. Create a file and name it app.js. In the app.js file, insert the following code:

```
const http = require('http');
```

The http module is used for HTTP functions. It is used in this example for listening to requests from the user.

__ 6. Below the http line, add another line for https:

```
const https = require('https');
```

The https module is used for HTTPS functions. It is used in this example for requesting the Watson Language Translator service, as shown in the upcoming steps.

__ 7. Next, insert the following code snippet:

```
var portNumber = process.env.VCAP_APP_PORT || 8080;
const server = http.createServer(handleRequests);
server.listen(portNumber, function() {
;
```

This code creates a server to receive the HTTP requests from the user. The function **handleRequests** is called whenever a request is received.

__ 8. The next step is to implement the **handleRequests** function. Insert the following code snippet at the end of the app.js file. The results are shown in the following figure

```
function handleRequests(userRequest, userResponse) {

}
```

The **handleRequests** function has two parameters:

- The request (named userRequest in this example).
- The response (named userResponse in this example).

.In the **handleRequests** function, add the following line for setting the header data:

```
userResponse.writeHead(200, {'Content-Type': 'text/plain'});
```

__ 9. Below the userResponse.writeHead line, enter the following code snippet.

```
const inputData = JSON.stringify({
        "model_id": "en-es",
        "text": "Hello"
    });
```

The inputData declaration contains the JSON input that is to be sent in the body of the request to the call to the Watson Language Translator service.

__ 10. After the `inputData` declaration, add the following line:

```
var outputData = '';
```

The `outputData` declaration contains the output of the Watson Language Translator service. Initially, the `outputData` value is empty.

__ 11. After the previous line, enter the following line to refer to the properties of **VCAP_SERVICES**:

```
var vcap_services = JSON.parse(process.env.VCAP_SERVICES);
```

VCAP_SERVICES is the environment variable that is added by IBM Cloud to contain any configurations that are dynamically added when the Node.js app is connected to a service, for example.

__ 12. After the `vcap_services` line, you must add the user name and password that will be used to access the Watson Language Translator service. Add the following lines:

```
const username = 'apikey';
      const password =
      vcap_services.language_translator[0].credentials.apikey;
```

The `username` must have the value `apikey`. The password is retrieved from the value of the environment variable inside **VCAP_SERVICES** that is called `language_translator[0].credentials.apikey`.

When the Node.js application is connected to the Watson Language Translator service, the environment variables are added and accessible to the Node.js application.

These environment variables are mentioned in the section “Obtaining the Watson Language Translator service credentials“.

After the `username` and `password` declarations, add the following lines to the options field:

```
const options = {
        hostname: 'gateway-lon.watsonplatform.net',
        path:
'/language-translator/api/v3/translate?version=2018-05-01',
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Content-Length': inputData.length,
            'Authorization': 'Basic ' + Buffer.from(username + ':' +
password).toString('base64')
       }
    };
```

The `options` field is a JSON object that contains the information that is required to access the Watson Language Translator service. The JSON properties are as follows:

`hostname`: The host name of the Watson Language Translator service.

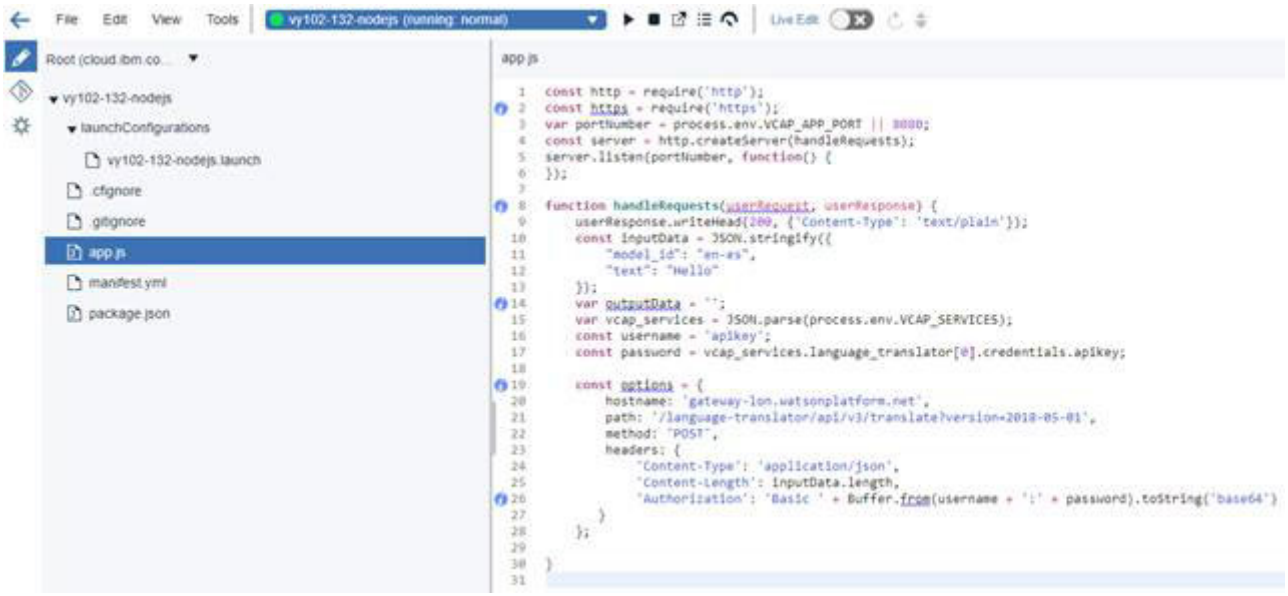
`path`: The route that is used by the exposed Watson Language Translator service. The version parameter is added to detect the version of the service.<br clear=all style='page-break-before:always'\

`method`: The type of the request to be used. To call the service, use the **POST** method.

headers: Contains the metadata to be sent in the request. You set the following parameters:

- o Content-Type: Sets the expected content type of the body of the request. The request body contains the JSON object, so the value is application/json.
- o Authorization: Sets the credentials of the Watson Language Translator service. The credentials are set by using the user name (apikey) and the password (which comes from the value of the apikey environment variable).

The code now looks as the following figure.



__ 13. After the options section, add the following code:

```
const languageTranslatorRequest = https.request(options,
function(languageTranslatorResponse) {
       });
```

The https.request function is used to call HTTPS services. The parameters that are used in the example for the https.request function are as follows:

- options: Contains the required metadata to connect to the Watson Language Translator service.
- A *callback function*, which is run after the response comes back from the call. This callback function has a parameter for the response that comes back from the called system. The variable name that is used here is languageTranslatorResponse.

__ 14. Inside the callback function, add the following code:

```
languageTranslatorResponse.on('data', function(d) {
           outputData+=d;
       });
```

The on function is an event listener function that contains the following parameters:

- The type of the event. In this case, it is data. Whenever parts of the response are received from the Watson Language Translator service, the data event is triggered.

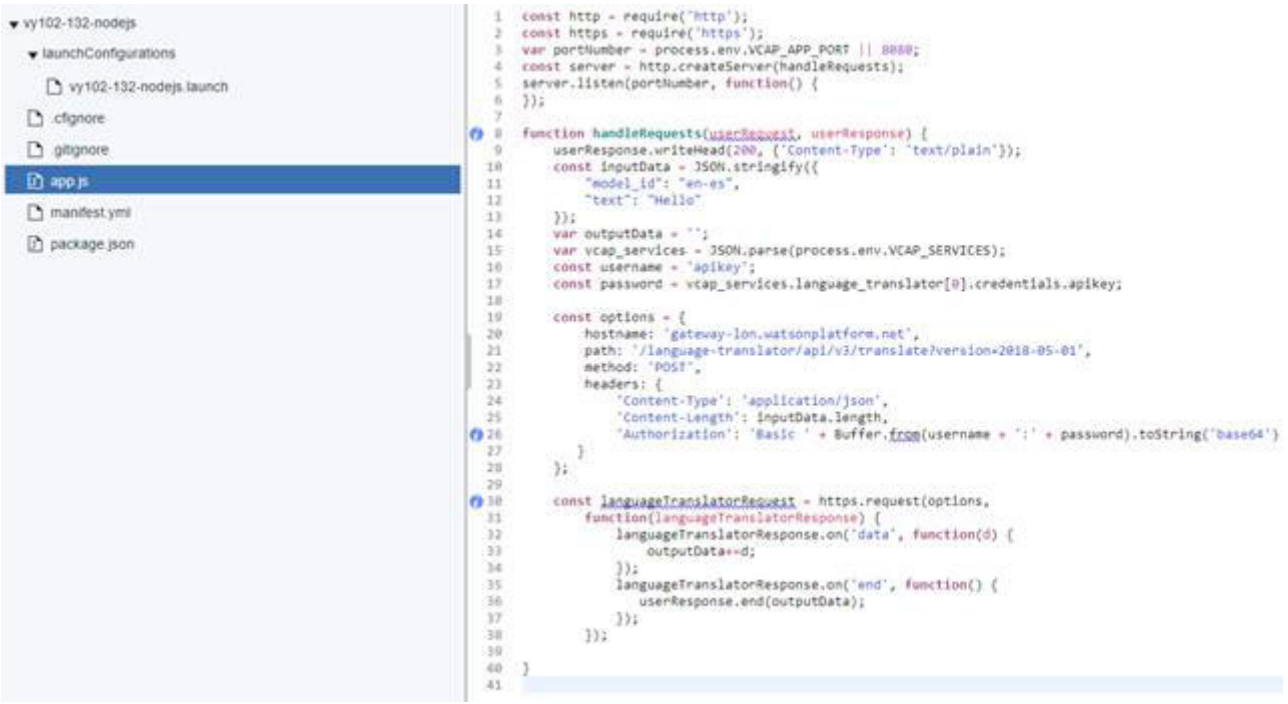
- The callback function that is run when the `data` event is triggered.
- The listener is added to `languageTranslatorResponse`, and whenever parts of the response are received, they are appended to the `outputData` variable.

__ 15. After the `languageTranslatorResponse.on` section, add the following code:

```
languageTranslatorResponse.on('end', function() {
        userResponse.end(outputData);
    });
```

This is another listener that is added to listen to the end event, which is triggered when `languageTranslatorResponse` is ready. Then, the `userResponse` (which is the response that replies to the user) can send the accumulated `outputData` to the caller.

The code for the listener looks like the following figure.



__ 16. The request in line 30 in the figure for `languageTranslatorRequest` is just the declaration and assignment of the variable. Part of the assignment is to add the callback function that is to be called when the response is ready, but this callback function has not run yet.

Below the `languageTranslatorRequest` section (line 38), add the following lines:

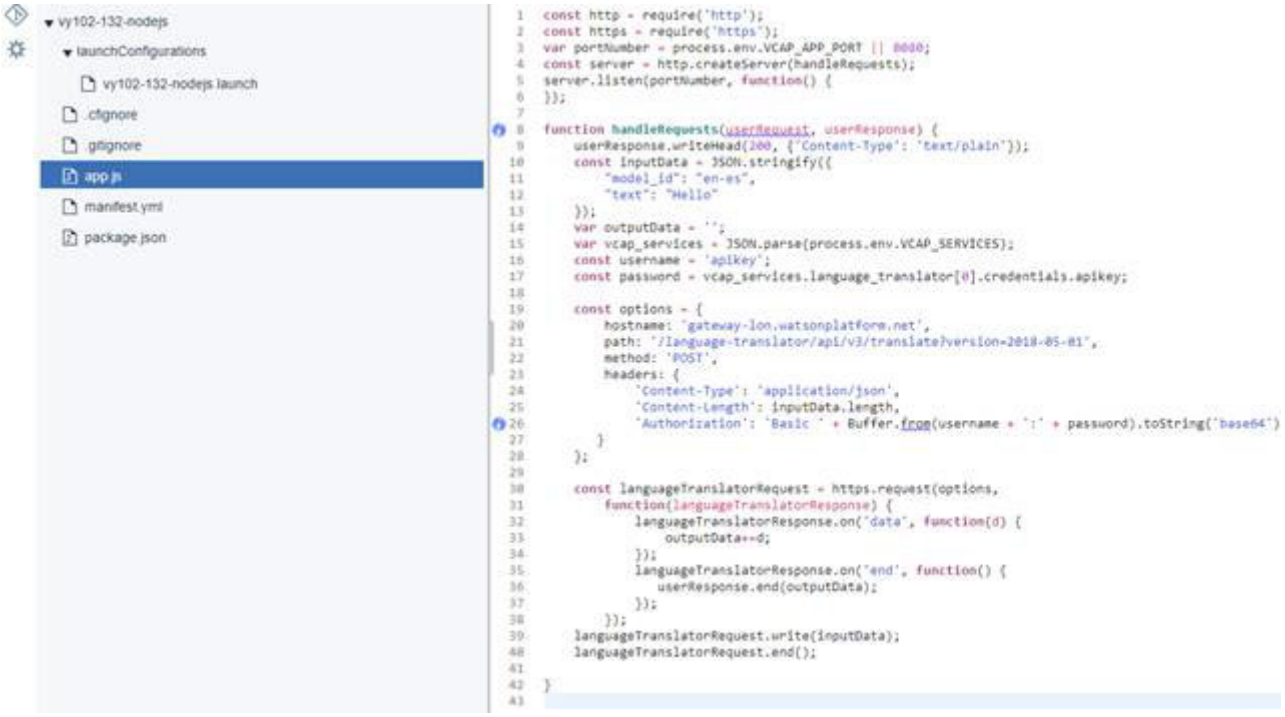
```
languageTranslatorRequest.write(inputData);
languageTranslatorRequest.end();
```

The `languageTranslatorRequest.write` line is used to fill the body of the request that is sent to the Watson Language Translator service.

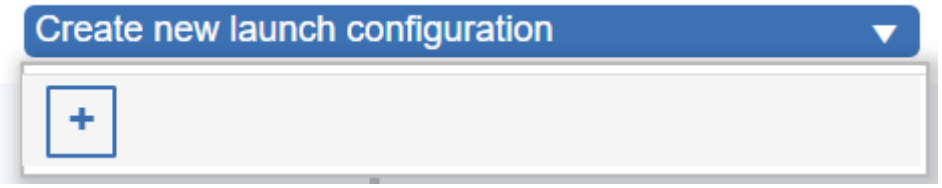
The **`write`** function can be used several times in the code. In this example, you need to use it only once.

After the required **write** calls, the `http` module expects you to call the **end** function to report that the request is ready to be sent and no more data is expected to be added to the request. Hence, the `languageTranslatorRequest.end()`call is used.

The code now looks like the following figure.

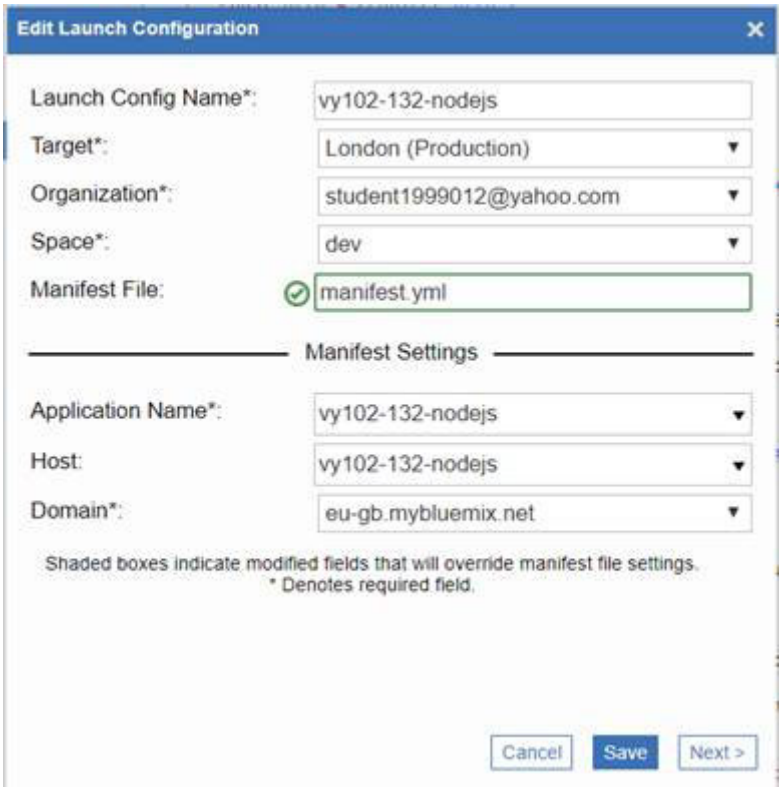


__ 17. Next, click **Create new launch configuration** and the "**+**" sign in the drop-down list, as shown in the following figure.



If you do not have the **Create new launch configuration** option, skip this step.

In the Edit Launch Configuration window, which is shown in the following figure, ensure that **Target** is set to your location (London in this example), **Organization** is set to your email address, and **Space** is set to dev, and click **Save**.



Note

If the message "**`Loading deployment settings...`**" is displayed in the Edit Launch Configuration window for a long time, change the default-selected Target field to the region that matches your account, for example, London in this example. After you select the correct target, the other fields load successfully.

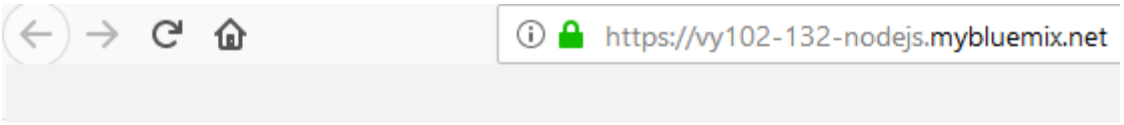
Then, click **Save**.

__ 18. Click the play icon (**Deploy the App from the Workspace**) to deploy the app.

__ 19. You might receive a notification that warns you that your application will be redeployed. Click **OK** to confirm.

__ 20. After the deployment is complete, click the **Open the deployed app** icon.

You see the output in your browser as a JSON response from the Watson Language Translator, as shown in the following figure. The output shows the translation of the *Hello* text from English to Spanish.




```
{
  "translations" : [ {
    "translation" : "Hola"
  } ],
  "word_count" : 1,
  "character_count" : 5
}
```


Part 6: Accessing the Watson Language Translator service through a Node.js module

In Exercise 1, you learned how to create a Node.js module. In this section, you learn how to return a callback function in a module.

The following steps create a Node.js module, called `translator` that contains the logic for accessing the Watson Language Translator service:

__ 1. In the root path, create a folder and name it `translator`.

__ 2. In the `translator` folder, create a file and name it package`.json`.

__ 3. Enter the following code snippet into the package`.json` file. The `main` field contains the path of the JavaScript file that has the Watson Language Translator service code.

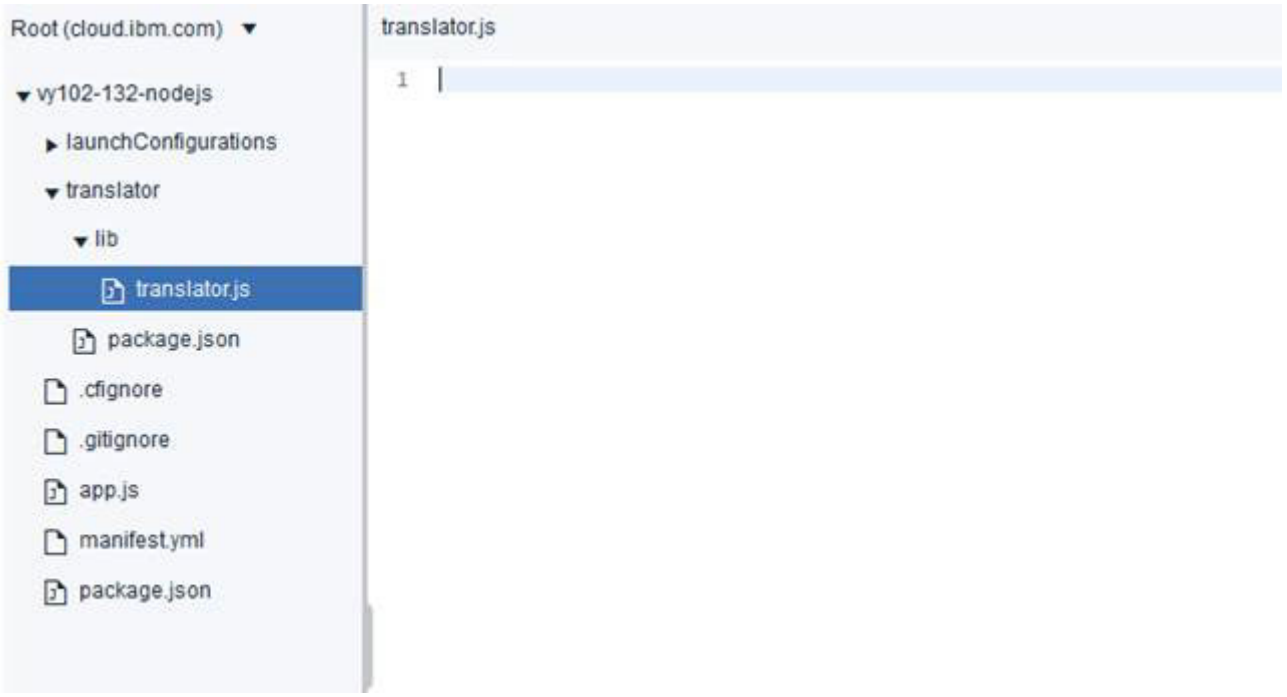
```
{
    "name": "translator",
    "main": "./lib/translator"
}
```

The package`.json` file in this code snippet shows the metadata of the `translator` module. The `name` attribute shows the name of the module. The `main` attribute shows the entry file to be referred to when the `translator` module is called.

__ 4. In the `translator` folder, create a folder that is named `lib`.

__ 5. In the `lib` folder, create the `translator.js` file.

The `translator` folder and the `translator.js` file now look like the following figure.



Leave the `translator.js` file empty for now.

__ 6. Open the `app.js` file.

__ 7. Cut the `require('https')` line, as shown in the following figure.

app.js

```
1  const http = require('http');
2  const https = require('https');
3
```

__ 8. Open the `translator.js` file and paste the `require ('https')` line there.

__ 9. Go back to the `app.js` file.

__ 10. After the `http` declaration, add the following line:

```
const translatorModule = require('./translator');
```

This line imports the `translator` module that you call in the upcoming steps.

__ 11. Inside the **`handleRequests`** callback function, after the `inputData` declaration, enter the following code snippet:

```
translatorModule.getTranslation(inputData, callback);
```

The code line in this code snippet calls the **`getTranslation`** function that must be defined in the Watson Language Translator module.

This function has following parameters:

- `inputData`

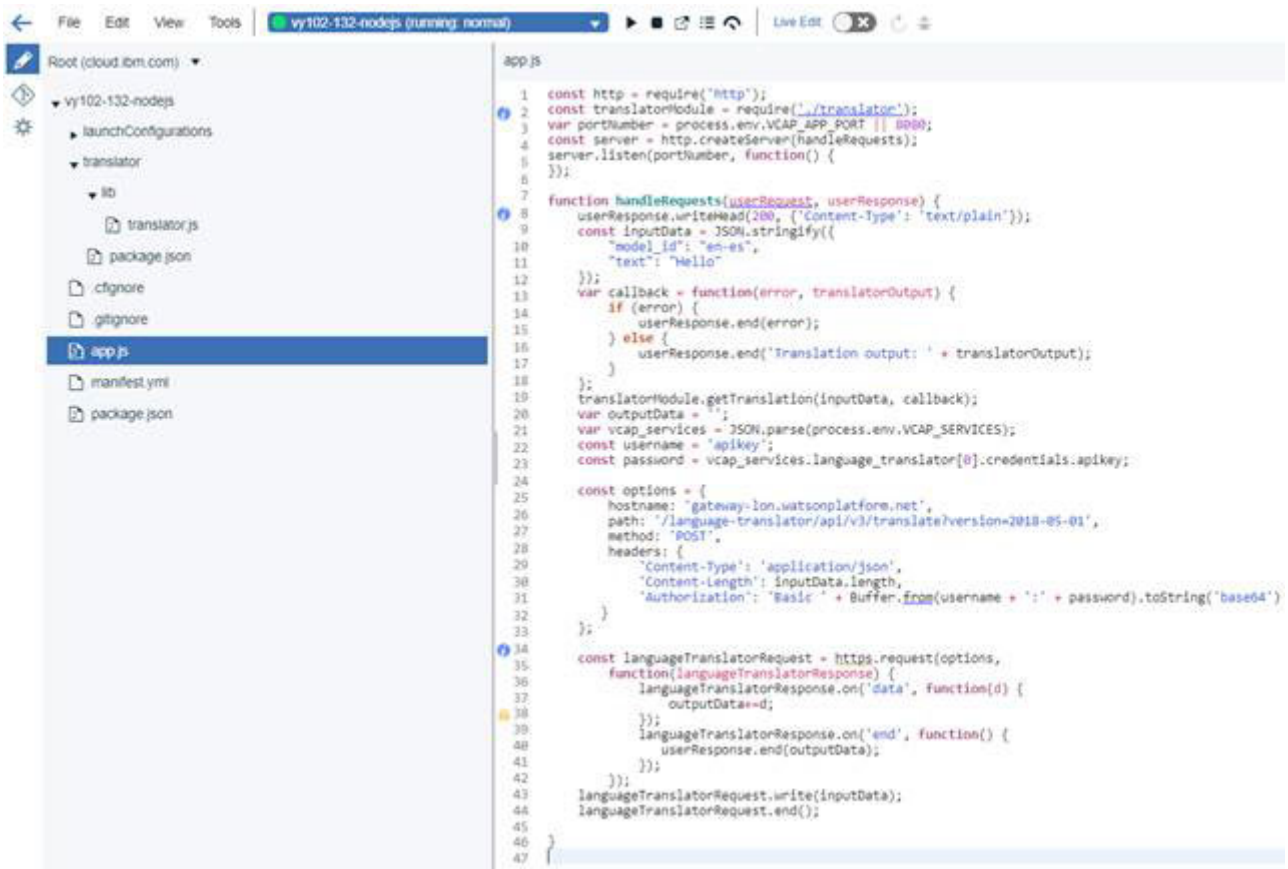
- The callback function, which is called somewhere inside the **`getTranslation`** function, as shown later.

  Before the `translatorModule.getTranslation()` line you just added, insert the following code snippet for the **`callback`** function that is mentioned in the previous step:

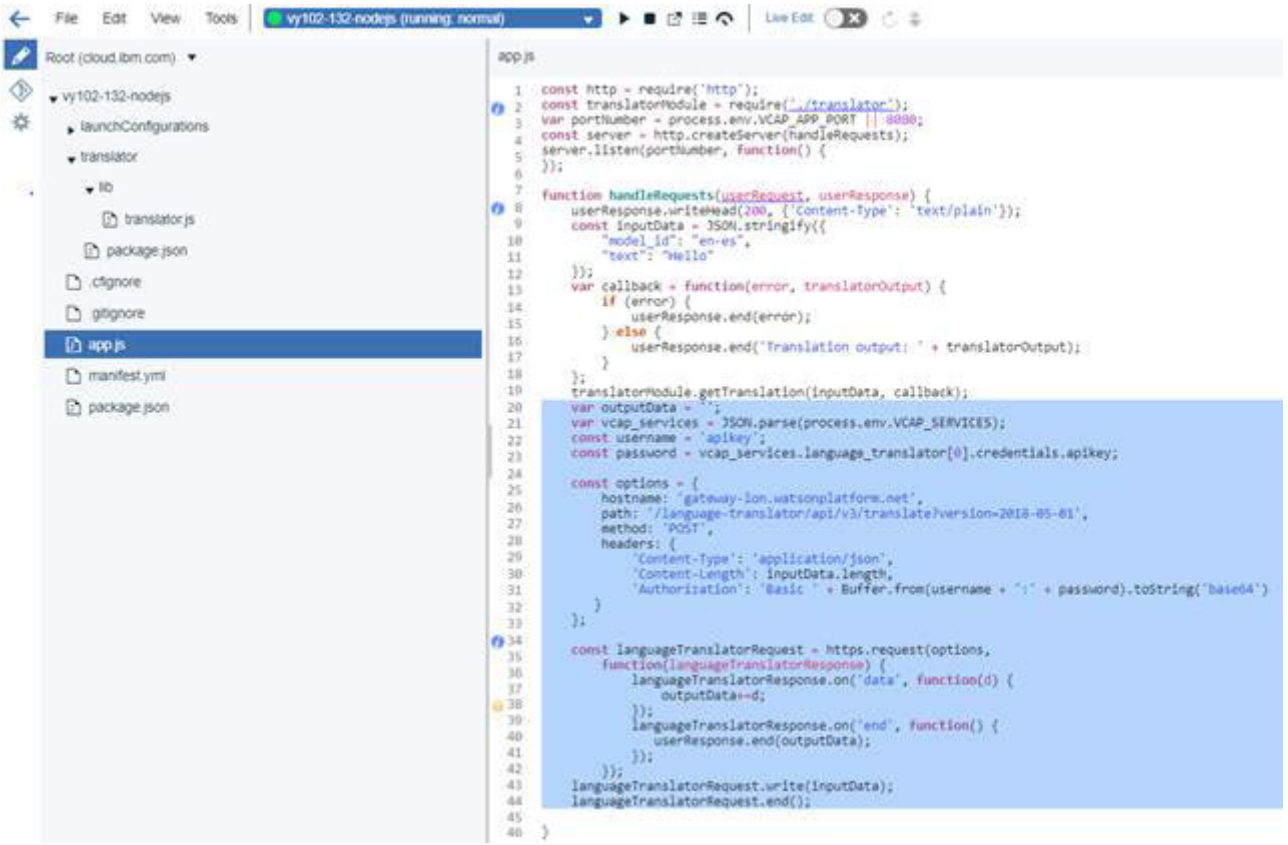
```
var callback = function(error, translatorOutput) {
        if (error) {
              userResponse.end(error);
        } else {
              userResponse.end('Translation output: ' +
translatorOutput);
        }
   };
```

The callback function is expected to have an **`error`** parameter that holds a value if errors occur. The other parameter is for `translatorOutput`. If the **`error`** object is not null, the HTTP response replies to the user with this error. If no error occurs, the message that contains the `translatorOutput` is sent in the HTTP response back to the user.

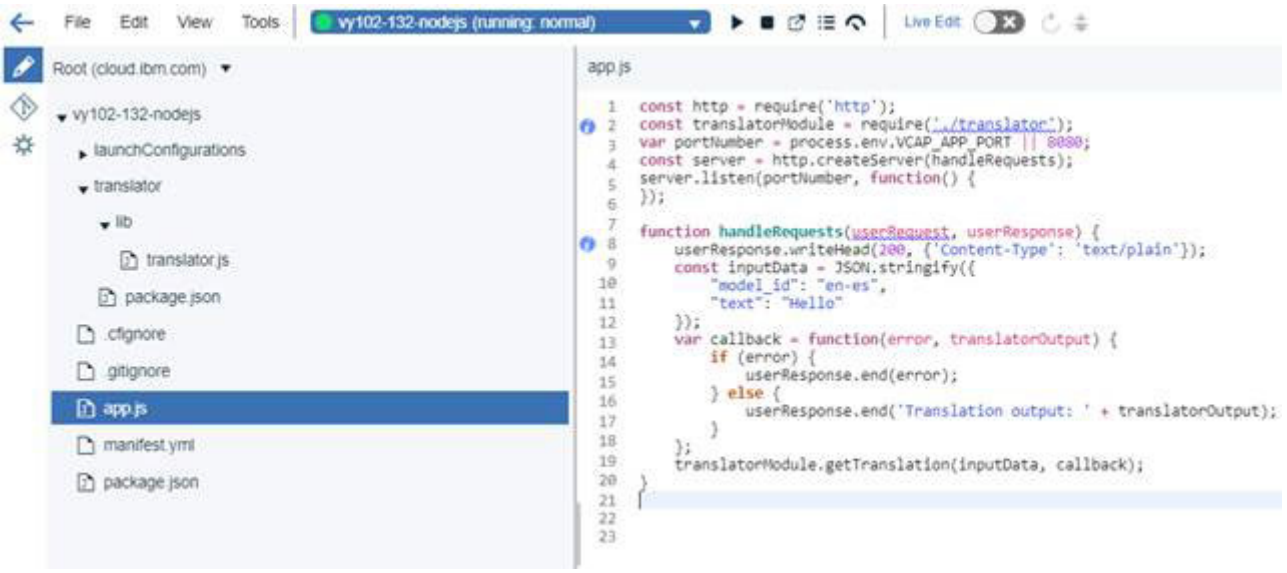
The `app.js` file now looks like the file in the following figure.



__ 12. Inside the app.js file, select the code below the `translatorModule.getTranslation` line, as shown in the following figure.



__ 13. Cut the selected code, and then paste it into a text editor somewhere outside Eclipse Orion. You use this code in the upcoming steps. The app.js file now looks like the following figure.



__ 14. Click **File**, and then **Save**.

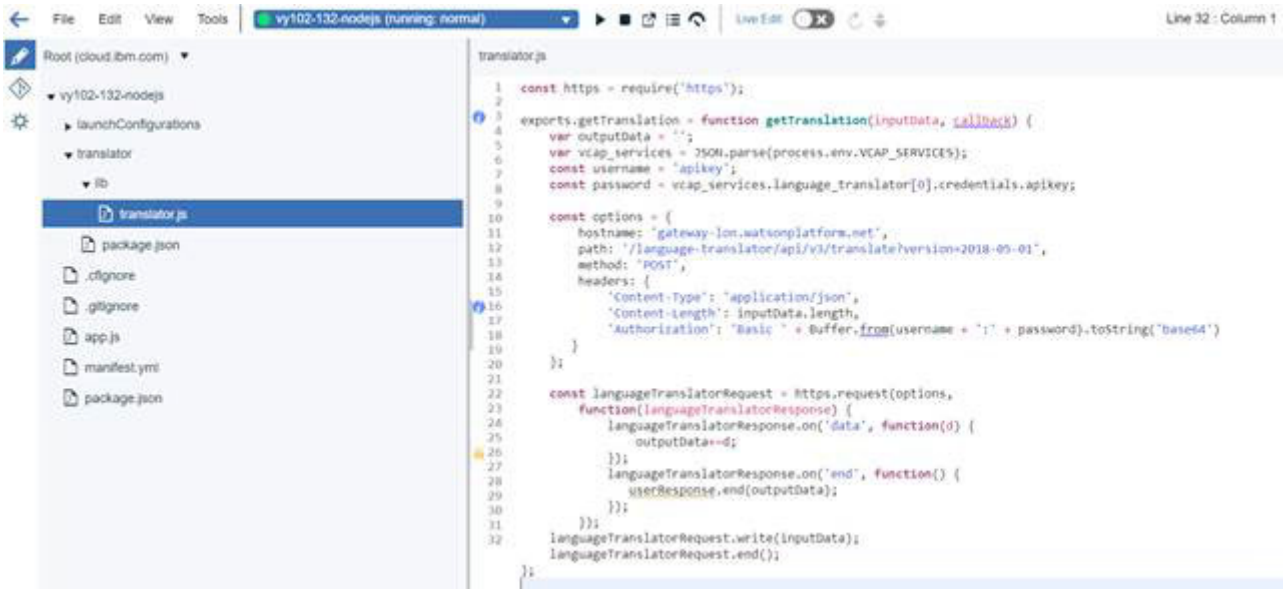
__ 15. Open the `translator.js` file. It contains only one line for requiring the `https` module, as shown in the following figure

__ 16. Next, define and export the **`getTranslation`** data, which is used in the app.`js` file. The initial code looks like the following code snippet.

```
exports.getTranslation = function getTranslation(inputData, callback) {

};
```

This function receives the fields `inputData` and the **`callback`** function.

__ 17. Inside the **`getTranslation`** function, enter the code that you recently copied outside the Eclipse Orion editor. The `translator.js` file now looks like the following figure.



__ 18. Inside the **`callback`** function replace the following lines:

```
languageTranslatorResponse.on('end', function() {
        userResponse.end(outputData);
    });
```

with the following lines:

```
languageTranslatorResponse.on('end', function() {
        callback(null, outputData);
    });
```

The ‘end’ listener is triggered when the response is received from the Watson Language Translator service so that you can run the callback function that is coming from the caller of the module.

The callback function (declared in the app.js file) is expecting to receive two parameters:

- `error`: If there is an exception when the service is called.
- `translatorOutput`: Expected to have the output that comes from the service.

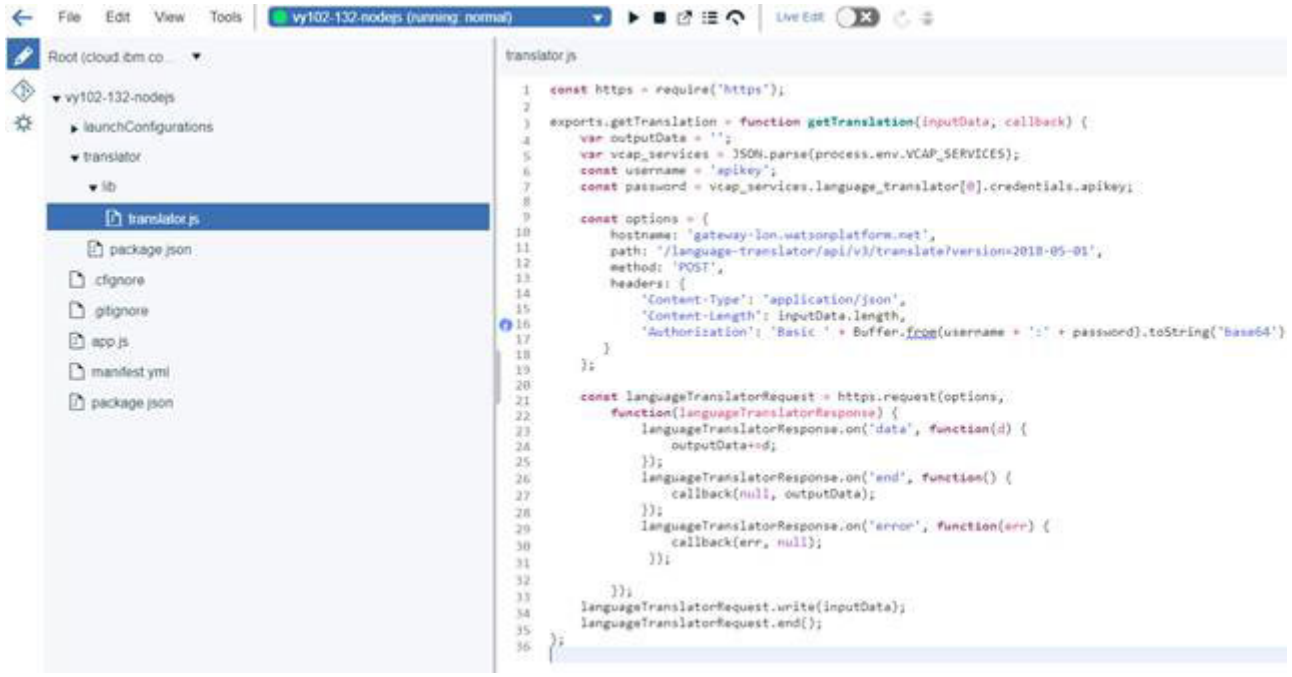
Because in this step it is expected that the call to the Watson Language Translator service succeeded, then the **`error`** parameter can be assigned to null, while the **`translatorOutput`** parameter contains the response data that comes from the Watson Language Translator service call.

__ 19. After the listener of the ‘end’ event, add the following code:

```
languageTranslatorResponse.on('error', function(err) {
        callback(err, null);
});
```

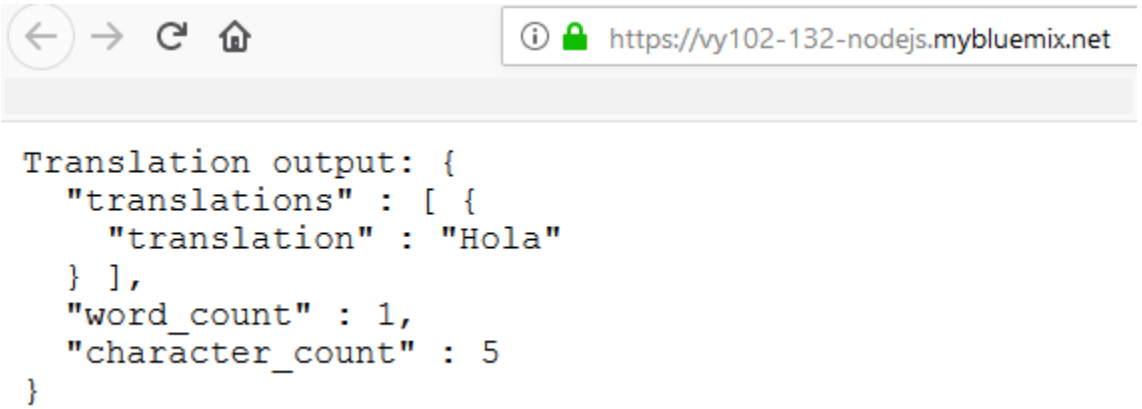
The ‘error’ listener is triggered if there is an error when the Watson Language Translator service is called. If an error occurs, the callback function is called and the error parameter is assigned to the error response that comes from the Watson Language Translator service. The `translatorOutput` is set to null.

The `translator.js` file now looks like the following figure.



__ 20. Click the play icon (**Deploy the App from the Workspace**) to deploy the app.

__ 21. After the deployment is complete, click the **Open the deployed app** icon. You see the output in your browser, as shown in the following figure.



Part 7: Stopping and deleting the application

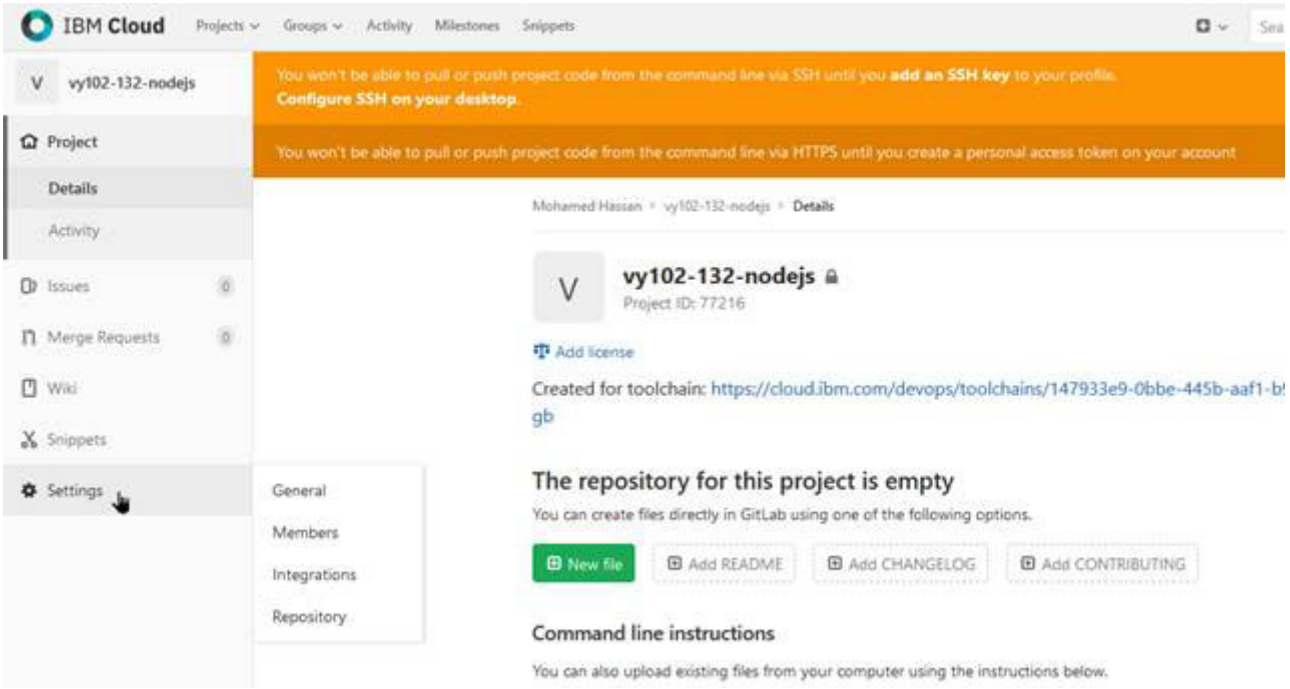
An IBM Cloud Lite account provides 256 MB of application memory for Cloud Foundry apps and 100 Cloud Foundry services.

To free the resources that are assigned to your application, you can either stop your application or delete it:

__ 1. To delete Git repository, go to Toolchain dashboard.

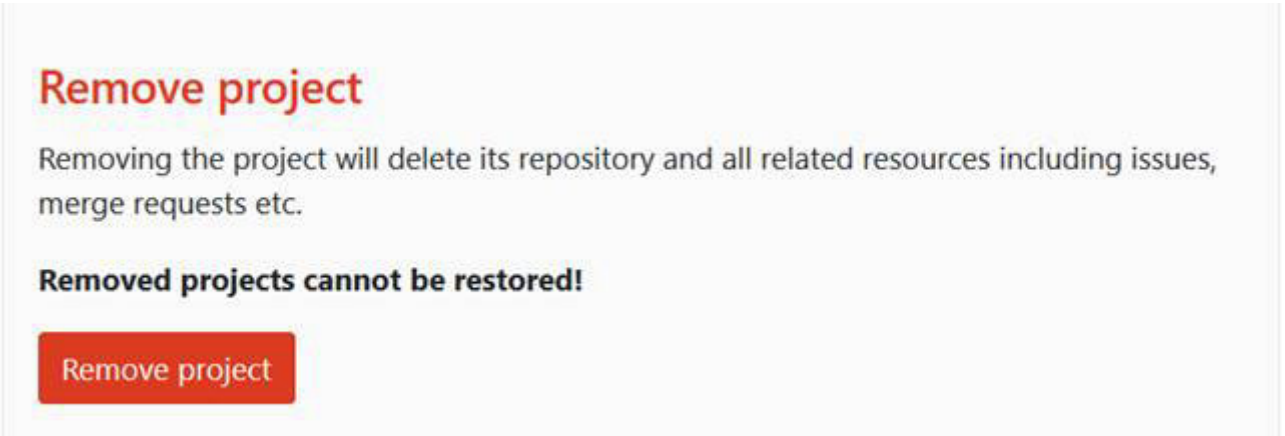
__ 2. Click the **Git** tile.

__ 3. From the left bar, click **Settings** as shown in the following figure.

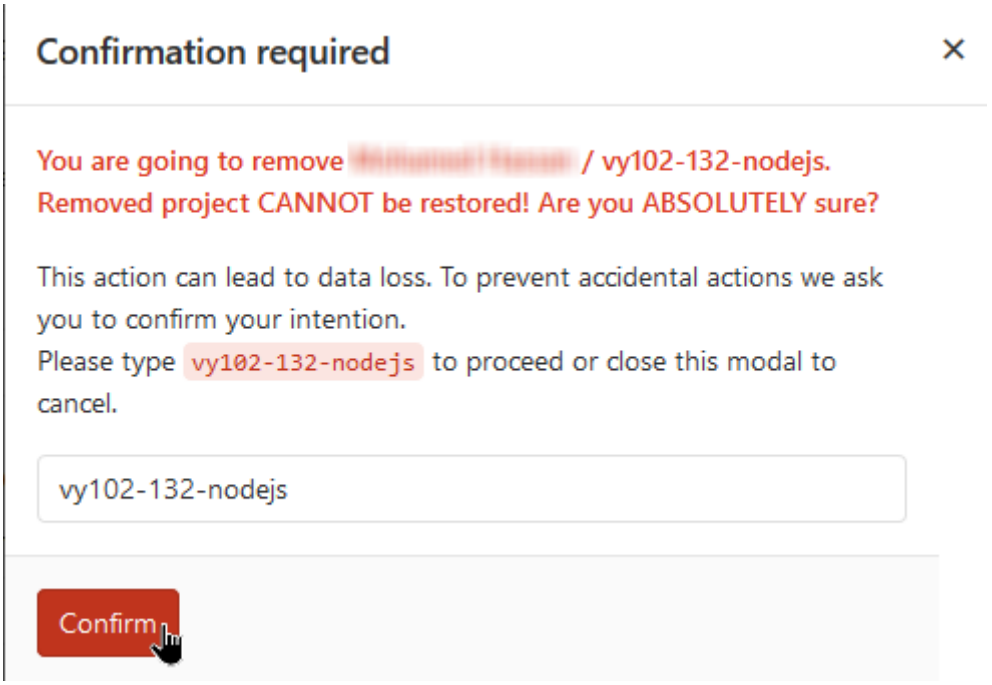


__ 4. Scroll down and click **Expand** in the **Advanced** section.

__ 5. Scroll down to **Remove project** and click **Remove project**.

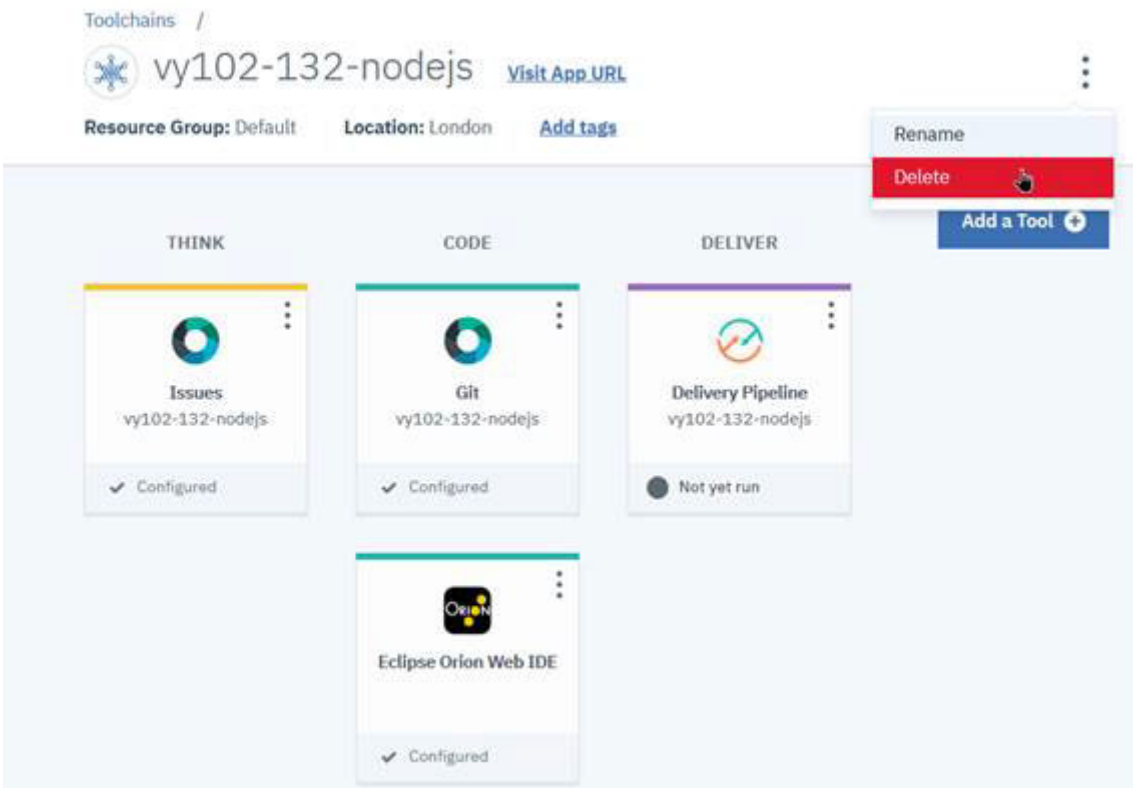


__ 6. At the confirmation window, type the name of your application and click **Confirm**.



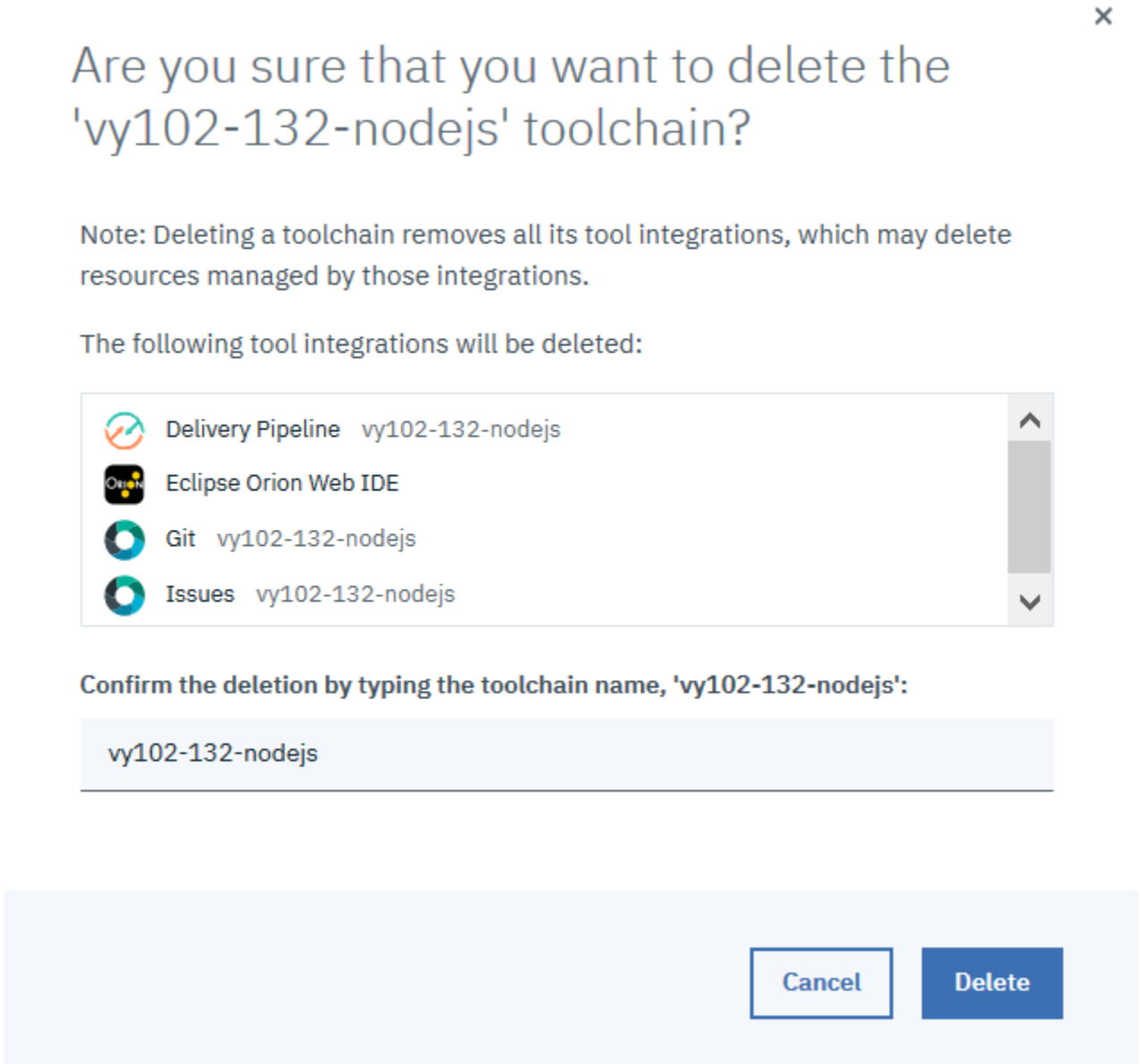
__ 7. To delete the toolchain that you created, go to Toolchain dashboard.

__ 8. Click the three dots next to the toolchain name.



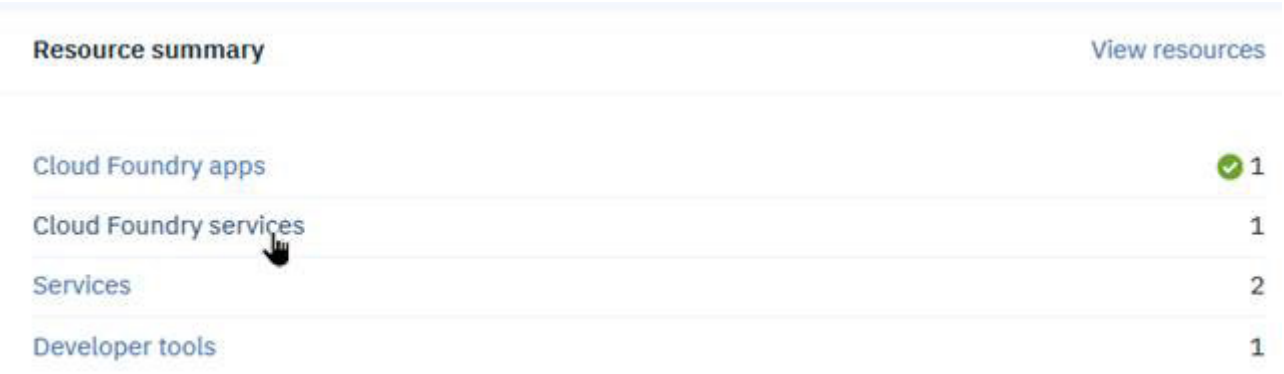
__ 9. Click **Delete**.

__ 10. At the confirmation window, type the name of your toolchain and click **Delete**

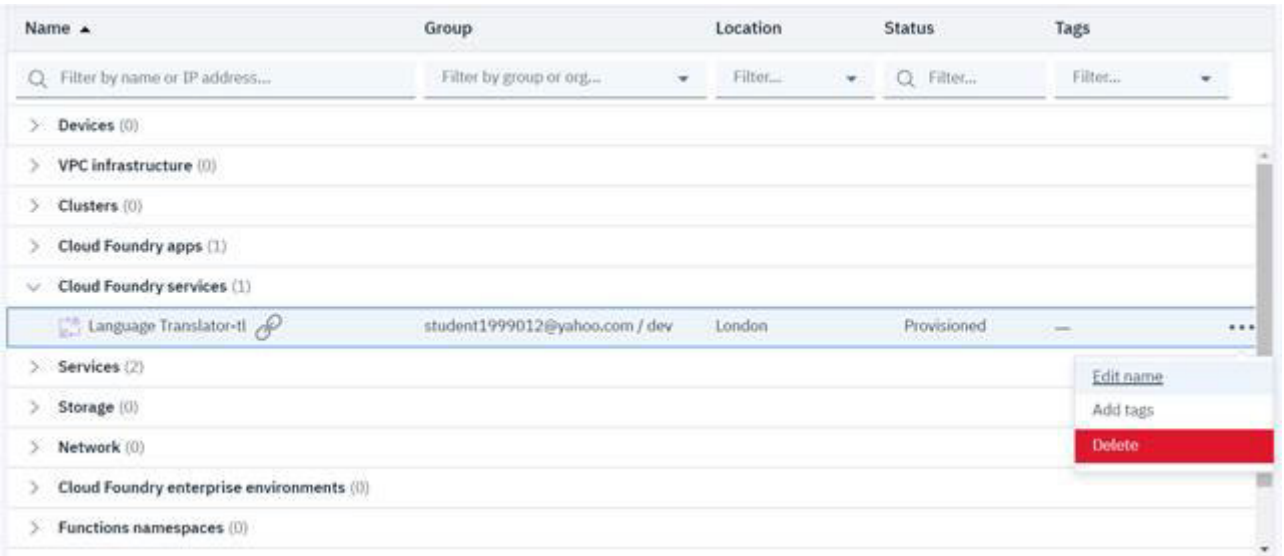


To delete the Language Translator service and the Node.js application complete the following steps:

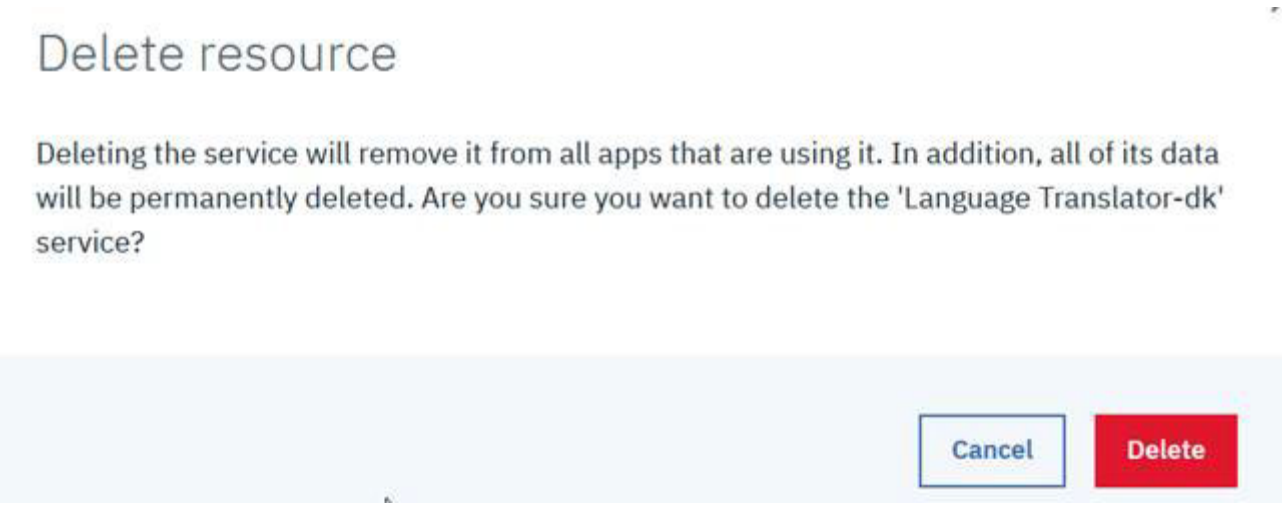
__ 1. Open the IBM Cloud dashboard and click **Cloud Foundry services**, as shown in the following figure.



__ 2. Click the **Actions** menu (three dots) for your Language Translator service and select **Delete**.

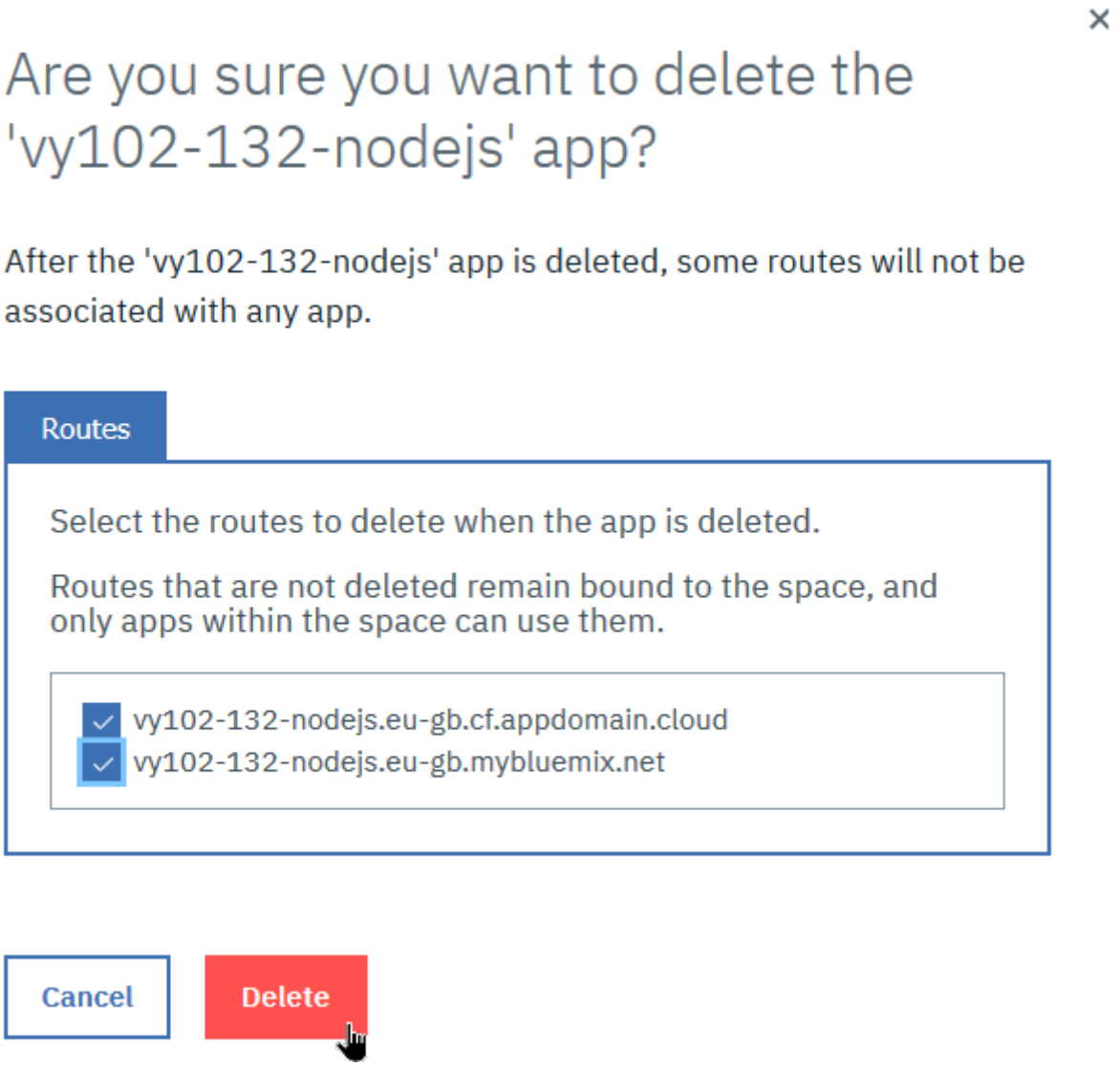


__ 3. At the Delete resource window, click **Delete**.



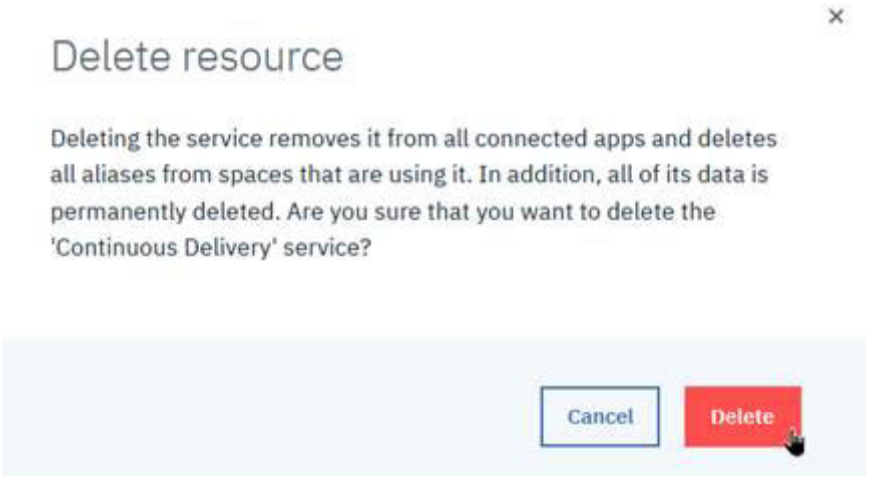
__ 4. Expand the **Cloud Foundry Apps** section and click Actions (three dots) to the right of your Node.js app to list the available actions. Click **Delete**, as shown in the following figure.

__ 5. In the confirmation window, select the routes and then click **Delete**, as shown in the following figure.



__ 6. Expand the **Services** section and delete the Continuous Delivery service that was associated with your app and the Language Translator service. In the list of actions, click **Delete**, as shown in the following figure.

__ 7. In the Delete resource confirmation window, click **Delete** to confirm.



End of exercise

Troubleshooting

- For troubleshooting any issues, see the full code of this exercise at the following link:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp

- The following link contains the code that is explained in Part 5 “Accessing the Watson Language Translator service from the Node.js app”:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp/Ex2/1-callback

- The following link contains the code that is explained in Part 6 “Accessing the Watson Language Translator service through a Node.js module”:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp/Ex2/2-callback-with-module

Exercise review and wrap-up

In this exercise, you accomplished the following goals:

- Created the Watson Language Translator service in IBM Cloud and connected it to your Node.js app.
- Used asynchronous callback functions in your Node.js app and learned how the callback function is run.
- Created a module in Node.js to call the Watson Language Translator service and used it from other JavaScript files.

Exercise 3. Creating your first Express application

Estimated time

1:30

Overview

Express is a Node.js web framework for rapid development of web applications. It provides an easy way to handle routing of an application by exposing REST APIs.

In this exercise, you create an application that uses the Express framework and the IBM Watson Natural Language Understanding service to extract the author name from articles that are published on the web. You provide the web address (URL) of the article to the application, and it outputs the name of the author (or multiple names if the article has multiple authors).

Objectives

By the end of this exercise, you accomplish these objectives:

- Create a Hello World Express application.
- Create a simple HTML view for your application.
- Explain Express routing.
- Use third-party modules in Node.js.
- Use the Watson Natural Language Understanding service in your application.
- Use a Git repository in DevOps on IBM Cloud.

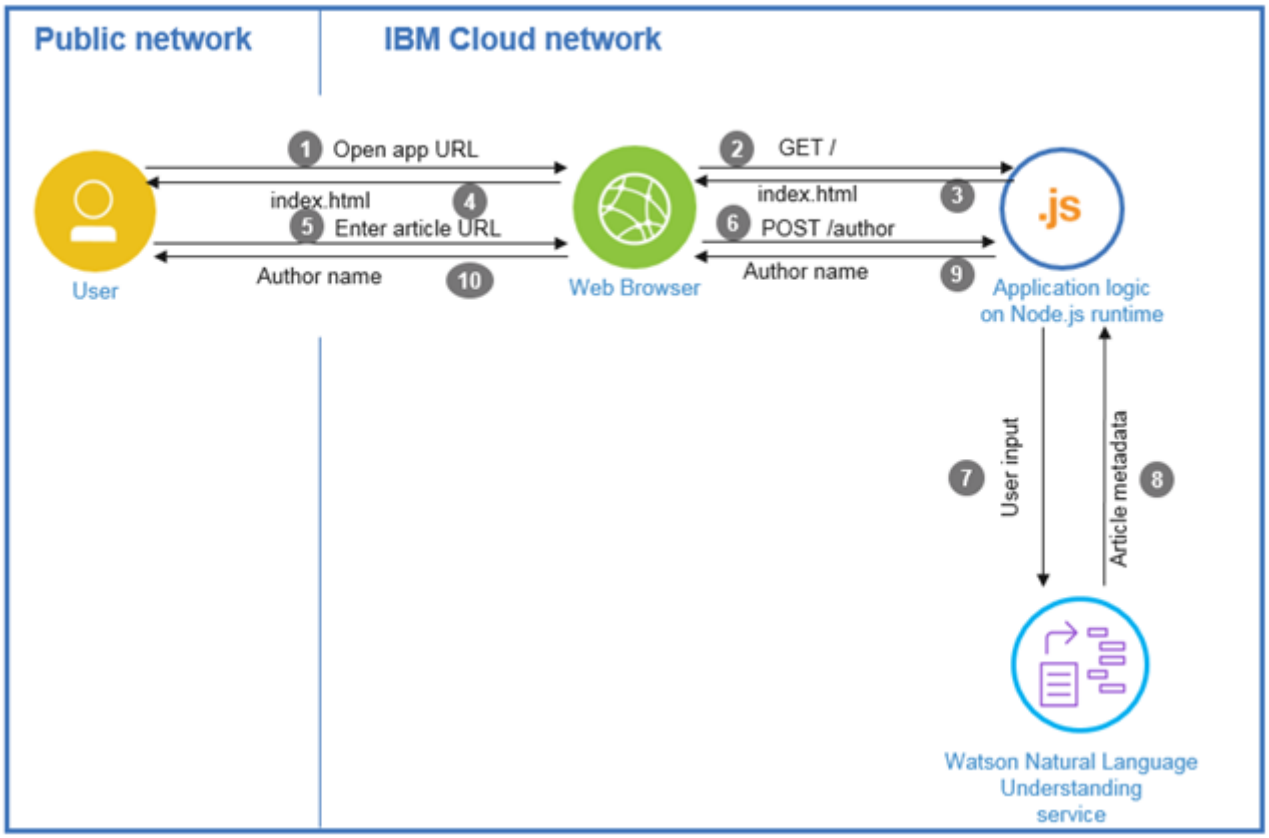
Requirements

Before you start, be sure that you meet these prerequisites:

- Basic JavaScript skills
- Basic HTML skills
- An IBM account.
- A workstation that has these components:
  - Internet access
  - Web browser: Google Chrome or Mozilla Firefox
  - Postman
  - Operating system: Linux, Mac OS, or Microsoft Windows

Introduction

The following figure shows the components and runtime flow of the application.



The following steps explain the sequence of interactions between the various components in the exercise:

__ 1. In a web browser, go to the application URL of this exercise:

`http://vy102-XXX-express.mybluemix.net`

__ 2. The web browser sends a **GET** request to the Node.js server. As mentioned in step 1, the application URL for this example `is http://vy102-`*`XXX`*`-express.mybluemix.net`. The path that follows this URL is a *route*, and there should be a handler for this route in the Node.js application.

For example, if the user sends the following request, there should be a route that is called `'GET /sample'` in the Node.js application:

`http://vy102-`*`XXX`*`-express.mybluemix.net/sample`

This route can return a resource (for example, an HTML page or an image), call a back-end service, or both.

In step 1, the user requested the home page of the application for this exercise. The browser sends a **GET** request to `http://vy102-`*`XXX`*`-express.mybluemix.net/`, which means that the `'GET /'` (root) route is called.

__ 3. The Express framework in Node.js returns the `index.html` file in response to the `'GET /'` route request.

__ 4. The web browser shows the `index.html` page to the user. The `index.html` page contains a form that has one text box, and a **Submit** button. In the text box, the user can enter the URL of the article.

__ 5. The user enters the article's URL, and then clicks **Submit**.

__ 6. The web browser sends a **POST** request to the `/author` route with the article URL passed in the body.

__ 7. The Express framework in Node.js passes the article URL to Watson Natural Language Understanding service. Also, it requests that the metadata is returned.

Note

The Watson Natural Language Understanding service uses natural language processing (NLP) to analyze semantic features of any text. The Watson Natural Language Understanding service has many features, such as concepts, categories, emotion, entities, keywords, metadata, and sentiment.

The feature that you use in this exercise is metadata. The feature has document metadata, including author name, title, RSS and Atom feeds, prominent page image, and publication date.

__ 8. The metadata of the article is returned by the Watson Natural Language Understanding service.

__ 9. Node.js filters the metadata to return only the author name (or names).

__ 10. The author name (or names) of the article are returned to the user on the web browser.

Exercise instructions

In this exercise, you complete the following tasks:

__ 1. Log in to your IBM Cloud account.

__ 2. Create the Node.js application on IBM Cloud.

__ 3. Create the Hello World Express application.

__ 4. Create a simple HTML view and organize the code.

__ 5. Integrate with the Watson Natural Language Understanding service.

__ 6. Deploy the application and run it.

__ 7. Clean up the environment

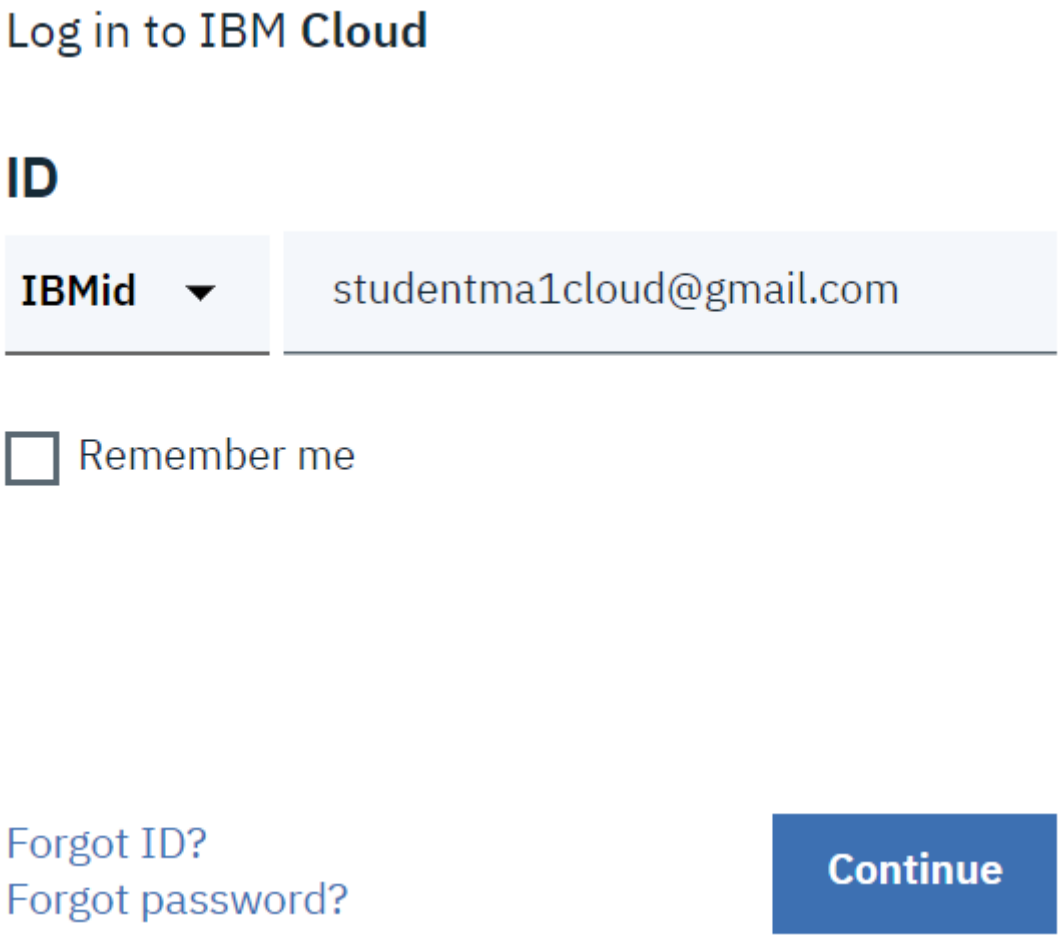
Part 1: Logging in to your IBM Cloud account

Log in to IBM Cloud by completing these steps:

__ 1. Open your web browser and enter the following web address:

`https://cloud.ibm.com/`

The IBM Cloud login page opens, as shown in the following figure.



__ 2. Enter your IBMid and click **Continue**.

__ 3. Enter your password and click **Login**.

Part 2: Creating the Node.js application on IBM Cloud

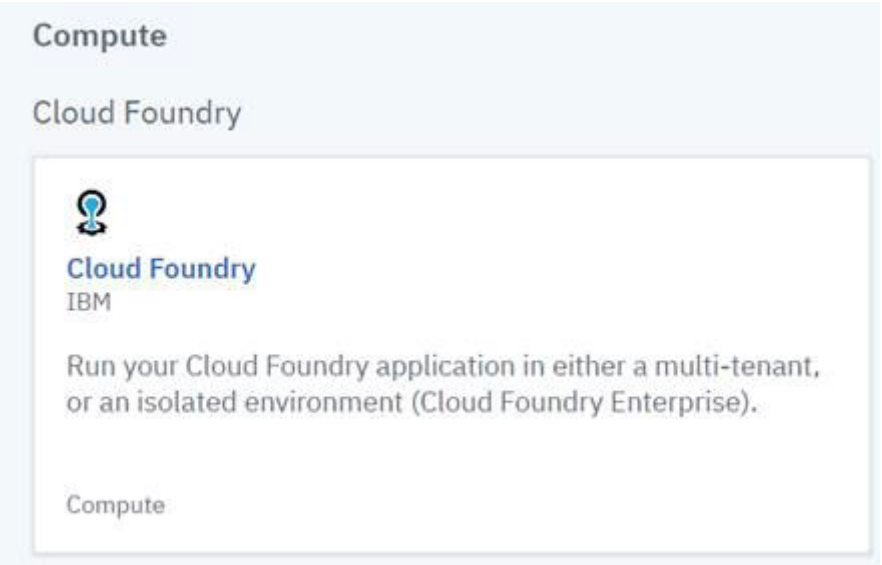
Complete these steps to create the Node.js app by using the IBM SDK for Node.js runtime on IBM Cloud and to enable continuous delivery for the application:

__ 1. In the IBM Cloud Dashboard, click **Create resource**, as shown in the following figure.

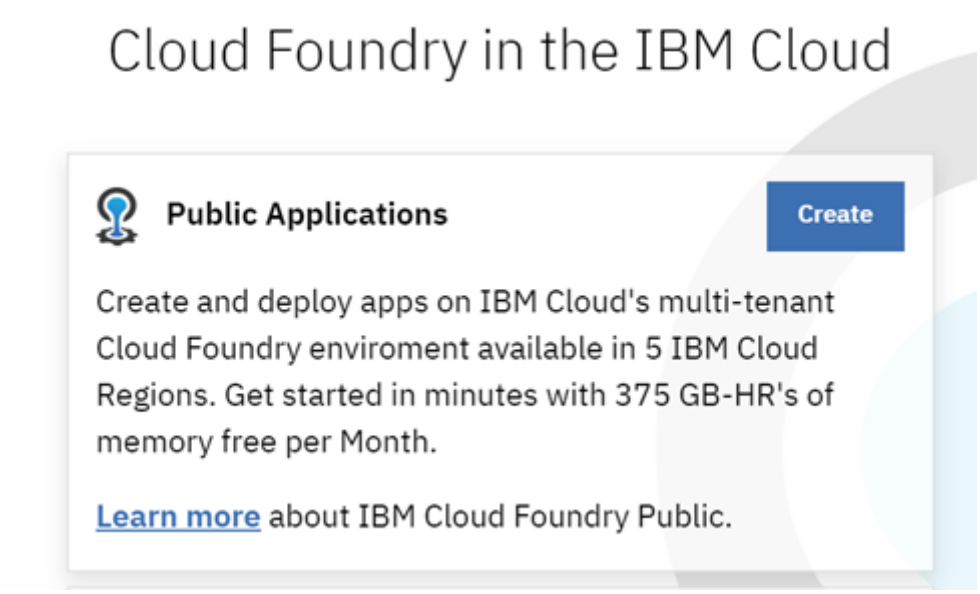
__ 2. The IBM Cloud Catalog page opens. It lists the infrastructure and platform resources that can be created in IBM Cloud. In the search field, enter **Cloud Foundry**.

__ 3. Select **Compute** from the menu on the left as shown in the following figure.

__ 4. Select Cloud Foundry under Compute, as shown in the following figure.



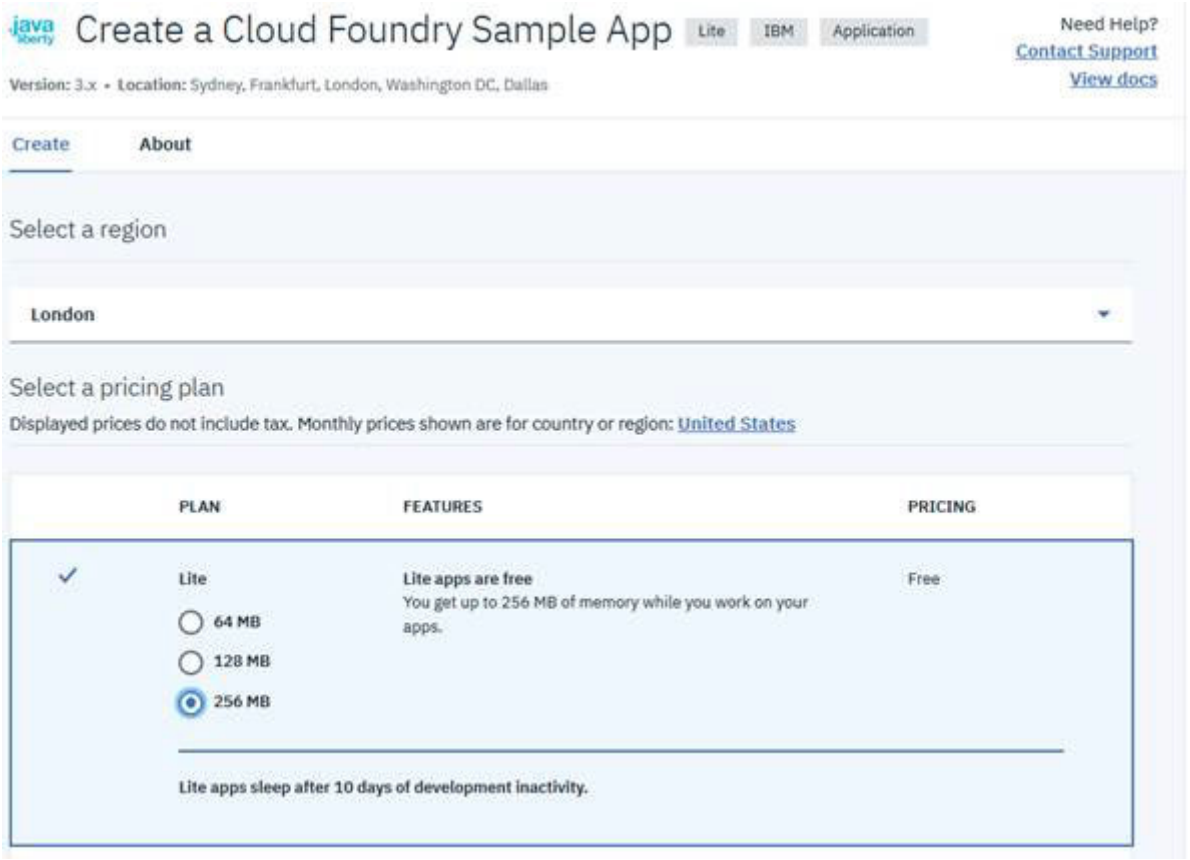
__ 5. From Cloud Foundry overview page, click **Create** for Public Applications as shown in the following figure.



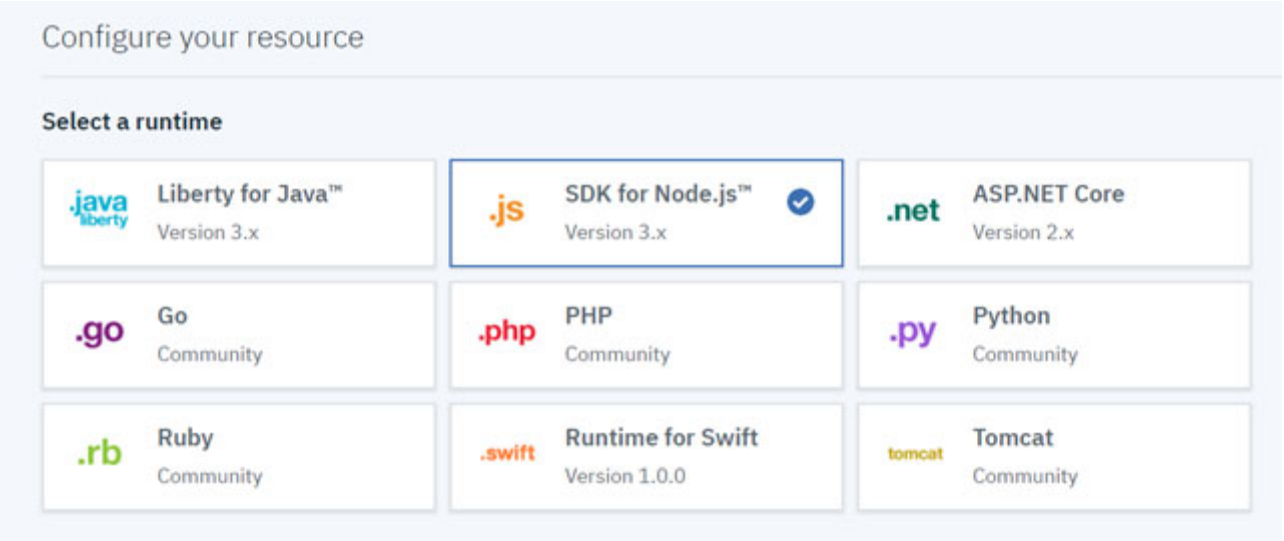
__ 6. Complete the application details for creating the Cloud Foundry Sample App:

__ a. Region is selected by default according to your location.

__ b. If you do *not* have a Lite account, skip this step. For Lite accounts, Pricing Plans show that the memory that is allocated to your app by default is 64 MB. For this exercise, select the maximum allocation of **256 MB**, as shown in the following figure.



__ c. Select **SDK for Node.js** from the provided runtimes as shown in the following figure.



__ d. In the **App name** field, type `vy102-`*`XXX`*`-express`, as shown in the following figure. Replace XXX with three random characters, which become your unique key. You use this unique key in the naming convention of this exercise

__ e. The host name is set by default to the app name.

__ f. The domain is chosen according to your location.
Select a domain from the “Domain” drop-down list that has the subdomain `{region}.cf.appdomain.cloud`.

Note

Make sure that the domain that is selected by default has the format `{region}.cf.appdomain.cloud`. Do not choose “*mybluemix.net*” as a domain because it is deprecated.

For example, if the region/location is set to *Dallas*, and domain is set to `us-south.cf.appdomain.cloud`.

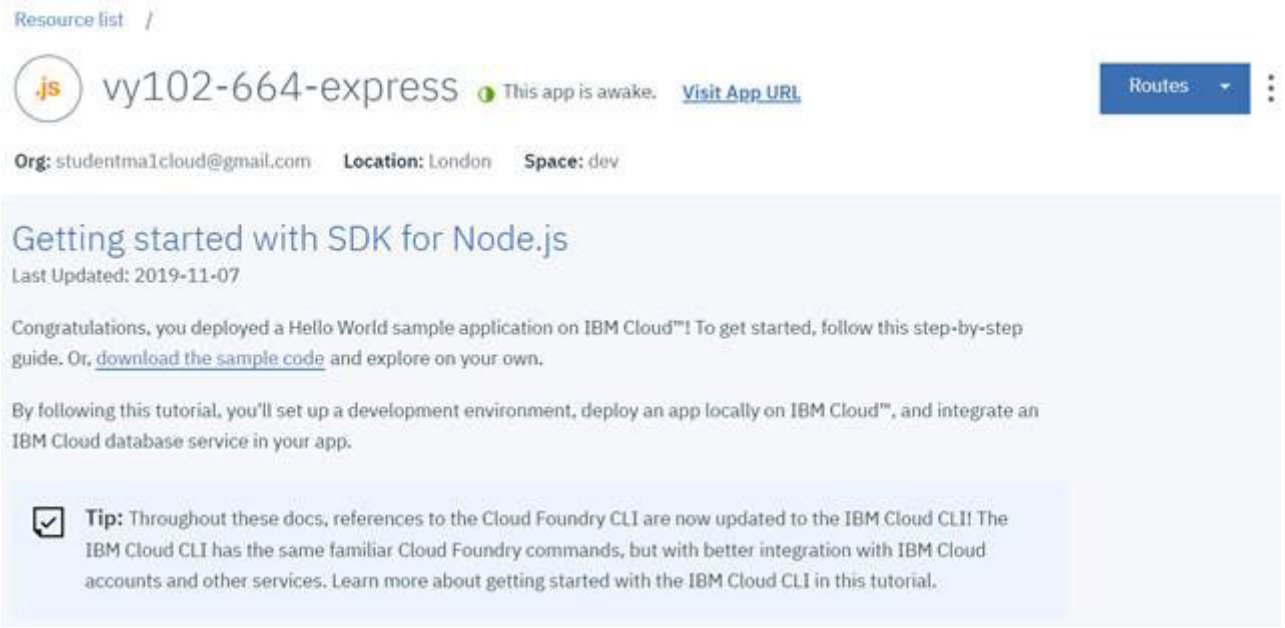
__ g. The organization is set by default to the email that you use to log in to IBM Cloud.

__ h. The space is set by default to dev, as shown in the following figure.

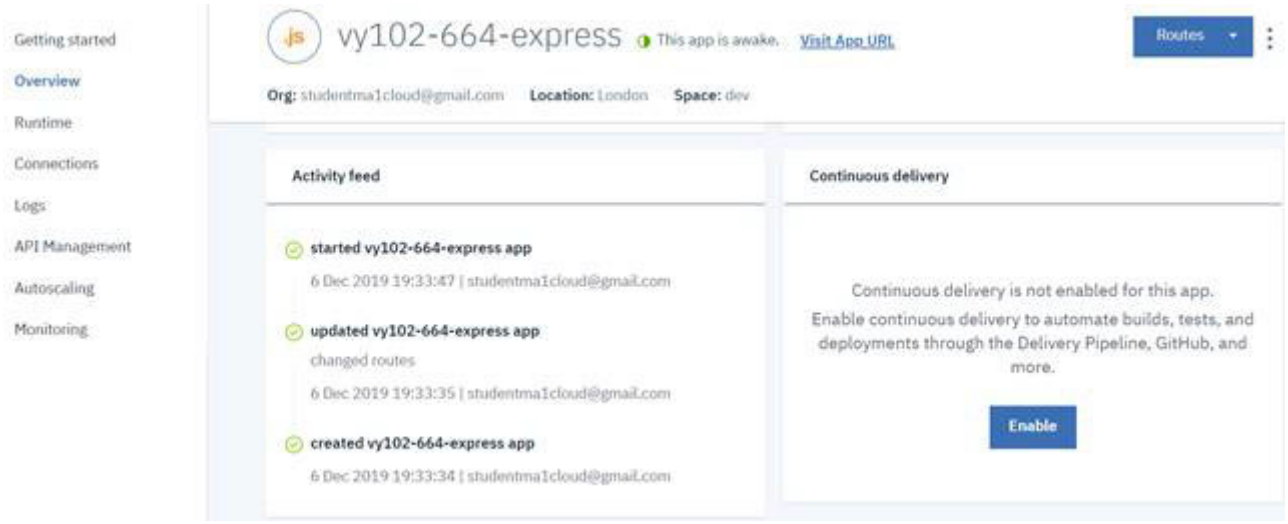
__ 7. Click **Create**.

In the next steps, you enable continuous delivery for this application.

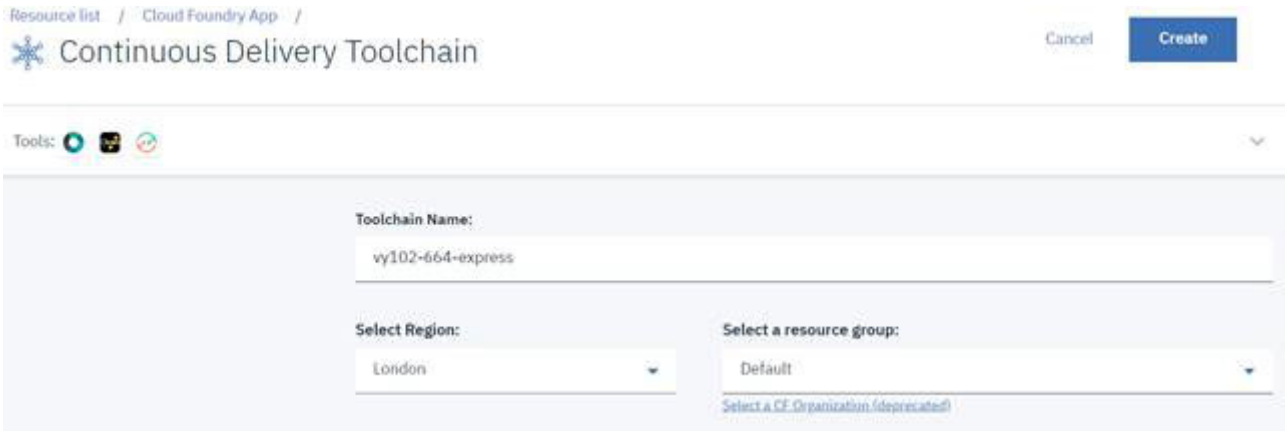
__ 8. Application Details shows the Getting started page, as shown in the following figure. Wait until the application is started and then click **Overview**.



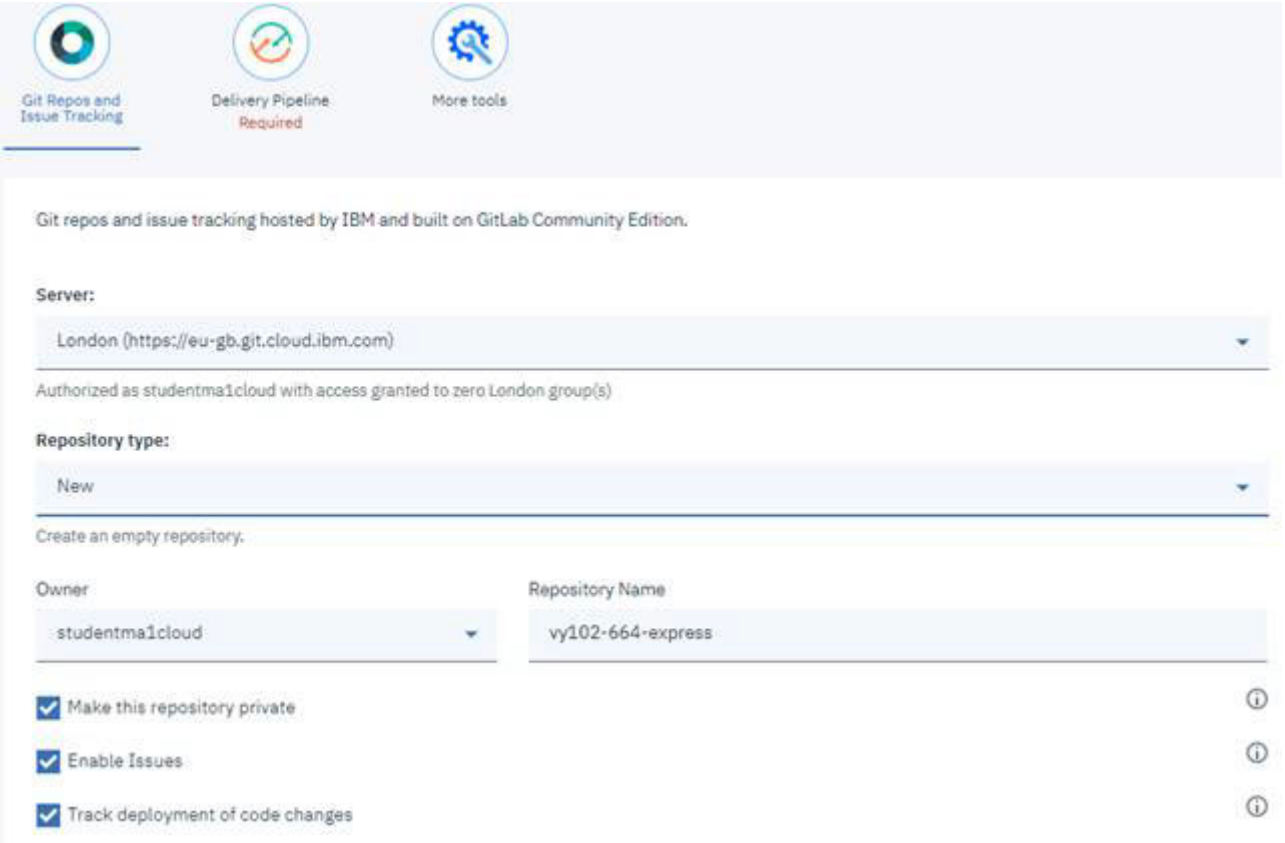
__ 9. In the Continuous delivery tile of the Overview page, click **Enable**, as shown in the following figure.



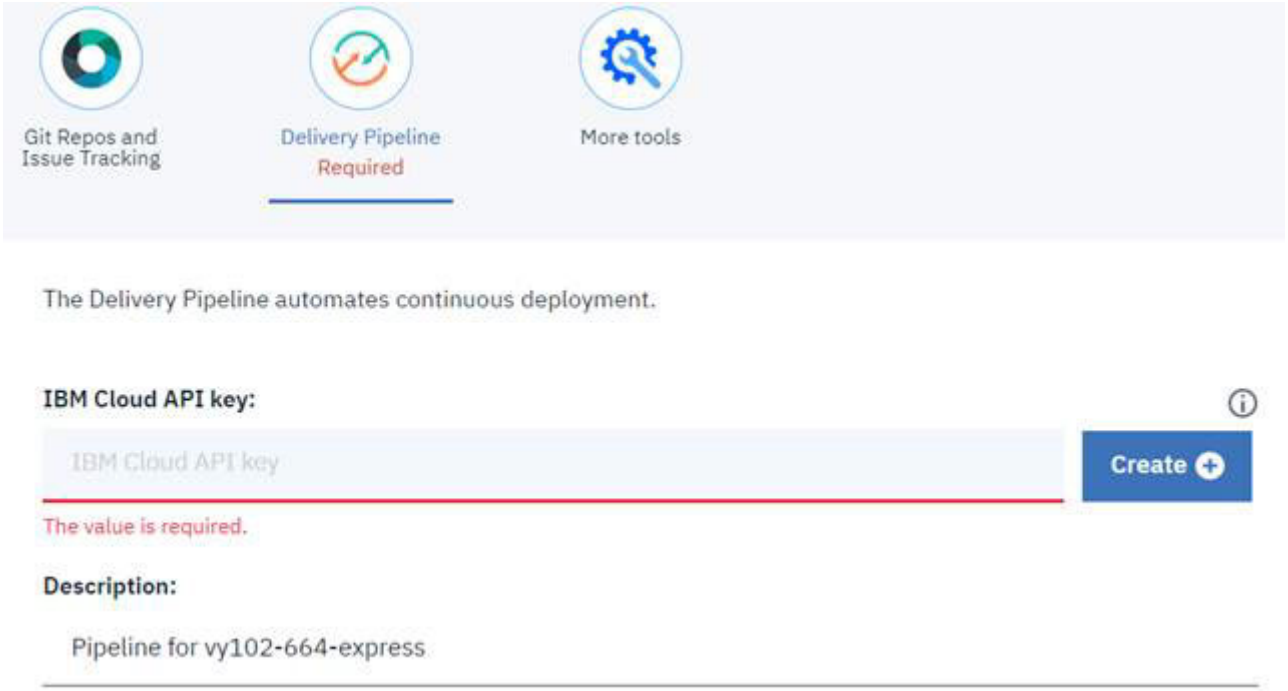
__ 10. A new Continuous Delivery Toolchain tab opens. This toolchain includes tools to develop and deploy the application. The **Toolchain Name** field is automatically populated. Keep the default values for the **Select Region** and **Select a resource group** fields, as shown in the following figure.



__ 11. Scroll down, and in the Repository type field, select **New** to create an empty repository, as shown in the following figure.

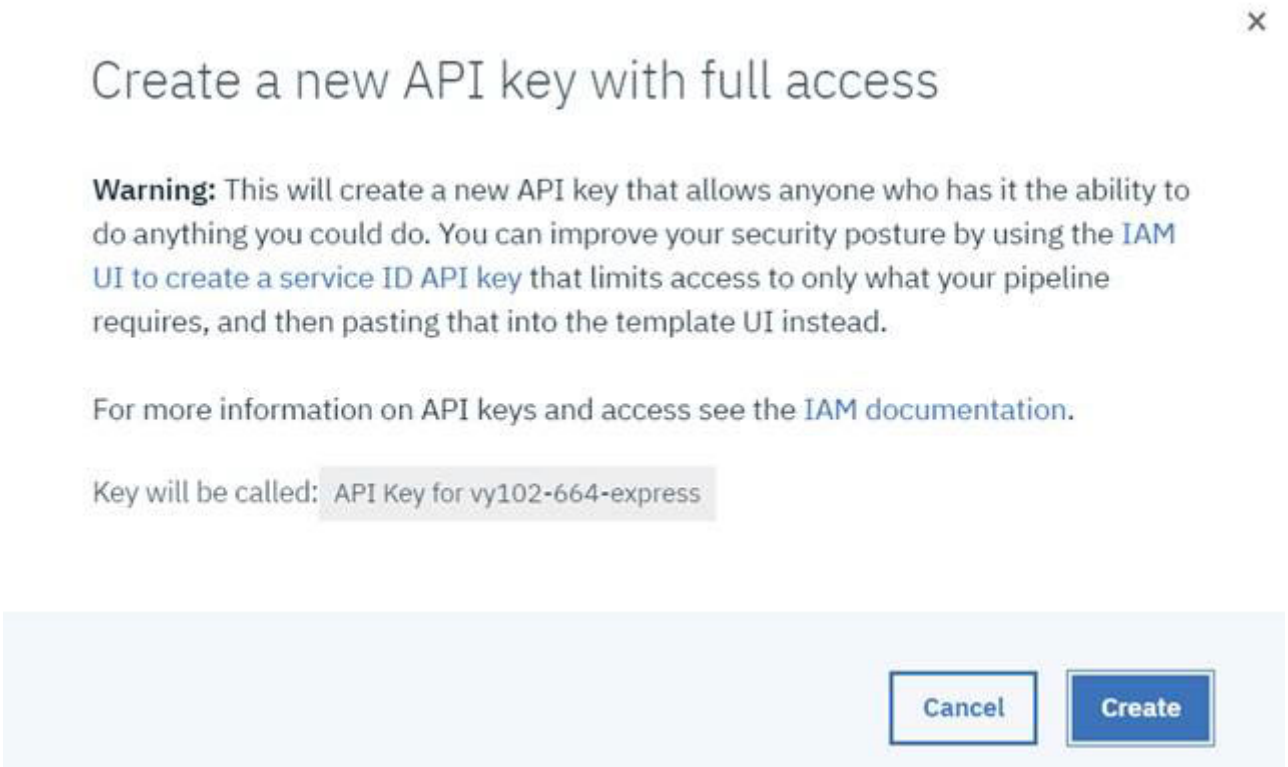


__ 12. Click **Delivery Pipeline**, as shown in the following figure.

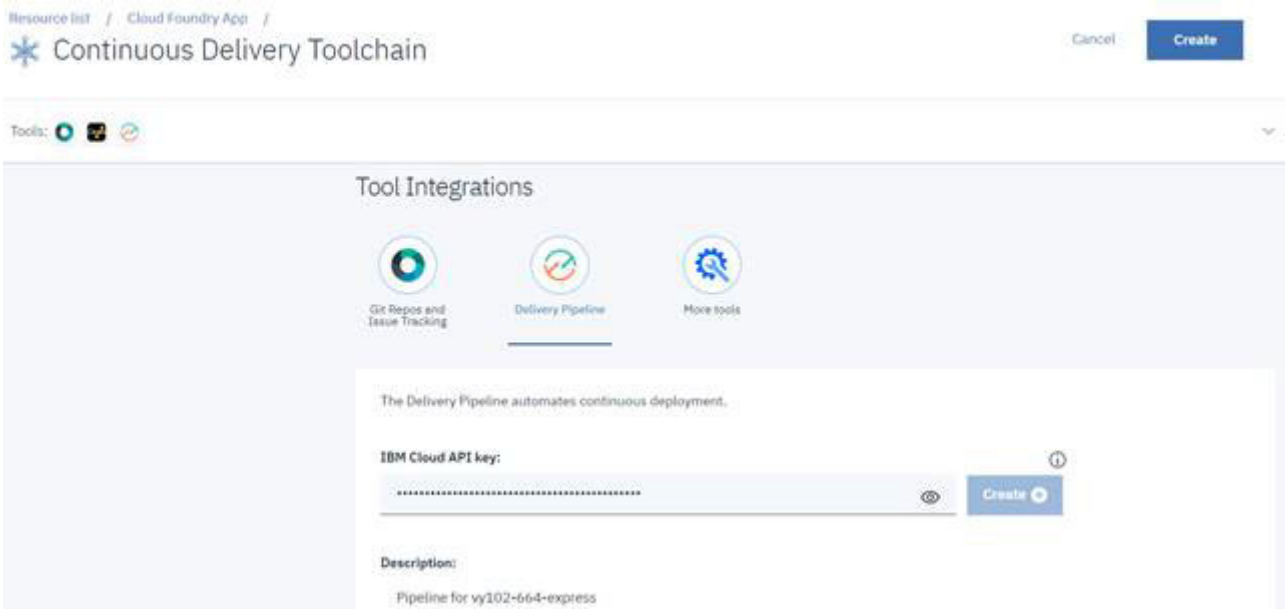


__ 13. The IBM Cloud API Key is required to access IBM Cloud Runtime. Click **Create +**.

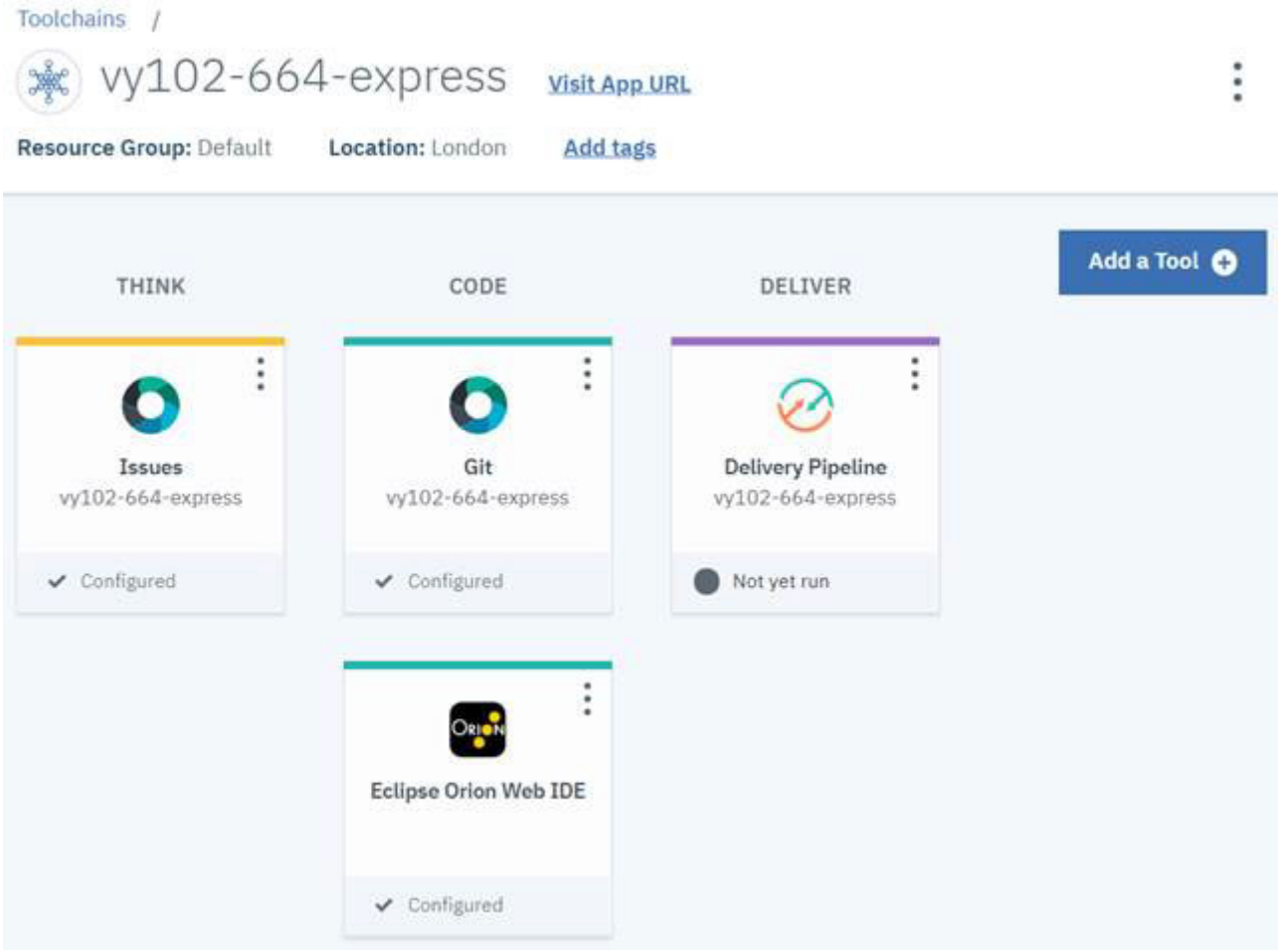
__ 14. In the Create API key window, click **Create**, as shown in the following figure.



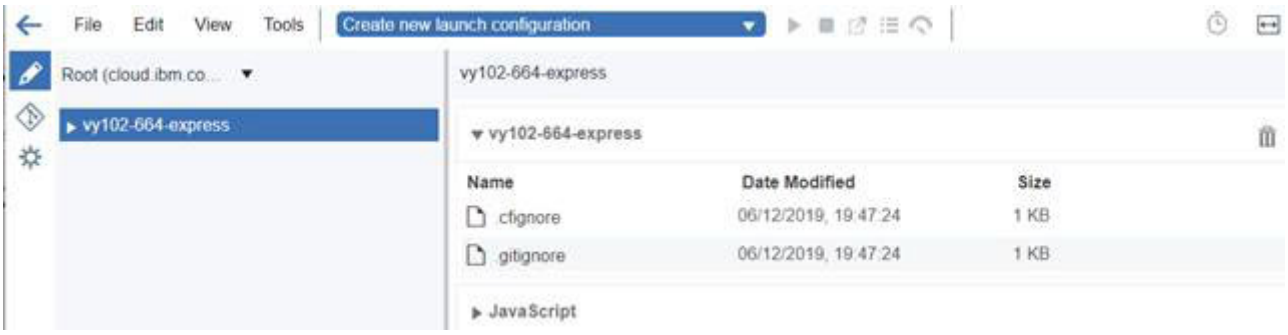
__ 15. The window closes and now the API key is generated. Click **Create** to create the pipeline, as shown in the following figure.



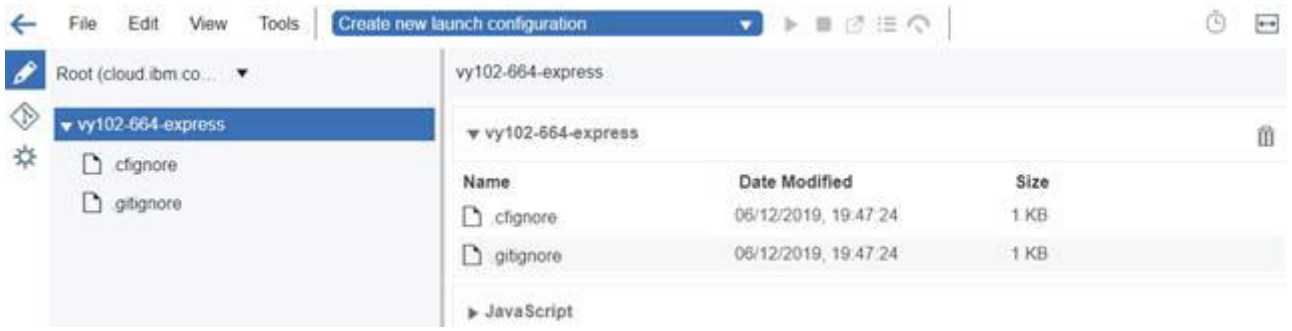
__ 16. A new toolchain page opens. Click **Eclipse Orion Web IDE** to start editing the code, as shown in the following figure.



__ 17. On the left, expand **vy102-xxx-express**, as shown in the following figure.



Notice that an empty project is created, as shown in the following figure.



Part 3: Creating the Hello World Express application

Express is a Node.js framework. It is used to simplify the creation of web applications on Node.js. The core component of Express is the *route*.

Route refers to the definition of application endpoints (Uniform Resource Identifiers (URIs)) and how they respond to client requests. Express supports the routing methods that correspond to the HTTP methods **GET**, **POST**, **PUT**, **HEAD**, **DELETE**, **OPTIONS**, and **TRACE**.

In the following steps, you create the Hello World Express application that returns the words “Hello Express!” in response to the `GET /` route request. The root (/) route is called whenever a user accesses the URL of the application. Also, you create another route, `POST /author`, after the user sends a **POST** request to the `/author` resource. This route returns “Author Name” to the user.

__ 1. Create the package.json file and add Express framework as one of its dependencies:

__ a. Right-click vy102-*xxx*-express in the left navigation bar and select **New > File**.

__ b. Name the file package.json, as shown in the following figure, and then press **Enter**.

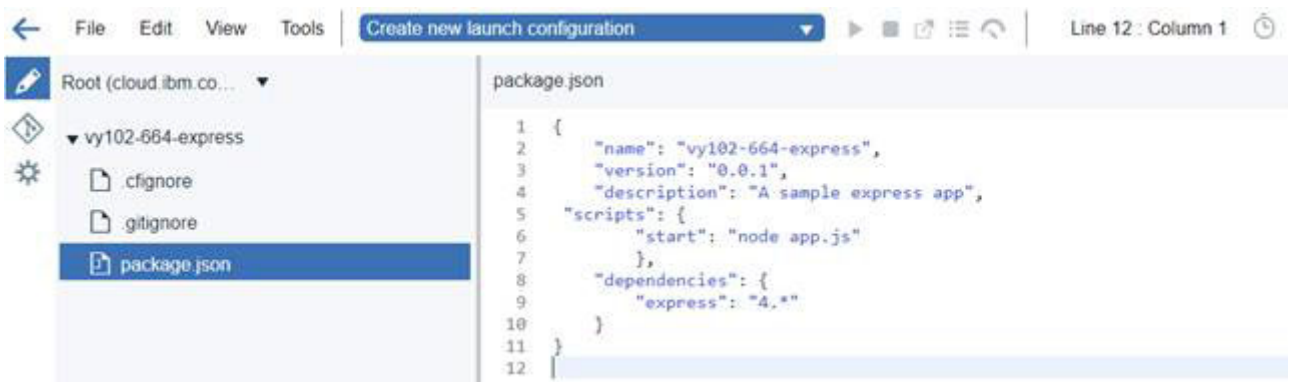
__ c. Copy the following code snippet to the package.json file:

```
{
    "name": "vy102-XXX-express",
    "version": "0.0.1",
    "description": "A sample express app",
 "scripts": {
           "start": "node app.js"
           },
    "dependencies": {
        "express": "4.*"
    }
}
```

Note

This code snippet is a JSON object that contains the metadata of the application, such as its name, version, and any dependencies that are needed to run the application.

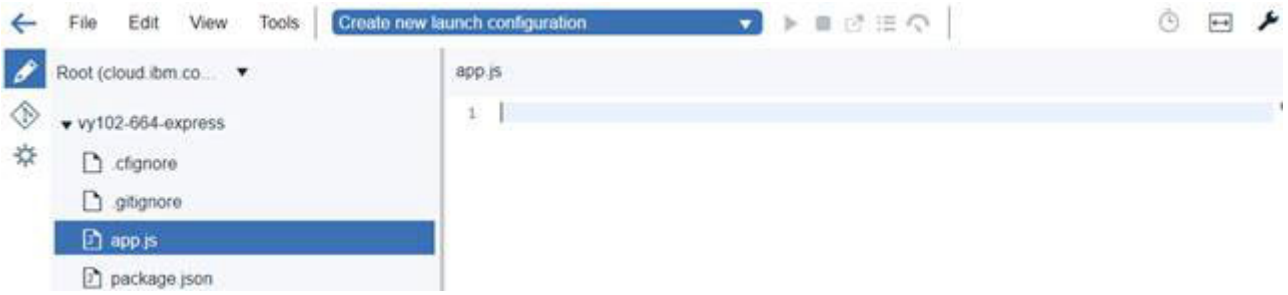
__ d. Notice that Express version 4.x is added as a dependency. In the name, replace *XXX* with the three characters that you assigned as part of your application name, as shown in the following figure.



__ 2. Create the app.js file. In app.js; you create an instance of Express and it is the starting point of your application:

__ a. In the navigation bar, right-click vy102-*XXX*-express, and then select **New > File**.

__ b. Name the file app.js and press **Enter**. Now, app.js is empty, as shown in the following figure.



__ c. Copy the following code snippet into the app.js file:

```
var port = process.env.VCAP_APP_PORT || 8080;

//Express Web Framework, and create a new express server
let express = require('express');
let app = express();

// In case the caller calls GET to the root '/', return 'Hello Express!'.
app.get('/', function(req, res) {
    res.send('Hello Express!');
;

// In case the caller calls POST to /author, return 'Author name'
app.post('/author', function(req, res) {
    res.send('Author name');
;

// start server on the specified port and binding host
app.listen(port);
```

The code instantiates the Express framework, listens to the default port of IBM Cloud, and exposes two routes (see the following figure):

- `GET /`: Returns “Hello Express!” to the caller when the caller requests the root of the application.
- `POST /author`: When the caller issues a post request to `/author`, “Author name” is returned.

app.js

```
1  var port = process.env.VCAP_APP_PORT || 8080;
2
3  //Express Web Framework, and create a new express server
4  let express = require('express');
5  let app = express();
6
7  // In case the caller calls GET to the root '/', return 'Hello Express!'.
8  app.get('/', function(req, res) {
9      res.send('Hello Express!');
10 });
11
12 // In case the caller calls POST to /author, return 'Author name'
13 app.post('/author', function(req, res) {
14     res.send('Author name');
15 });
16
17 // start server on the specified port and binding host
18 app.listen(port);
19
```

Note

In this exercise, you use `let` and `var`, but there are slight differences between them.

The main difference between them is that instead of being function-scoped, `let` is block-scoped, which means that a variable that is created with the `let` keyword is available inside the *block* in which it was created and also in any nested blocks. The term “block” refers to lines of code that are surrounded by curly braces {}, such as a `for` loop or an `if` statement.

Both let and var are global if outside a block.

Hint

You can press Shift+Alt+F to format the code.

__ 3. Add the `manifest.yml` file with the domain and host, and configure it to run `app.js` after the Node.js server starts:

__ a. Right-click **vy102-*xxx*-express** in the left navigation bar, and then select **New** > **File**.

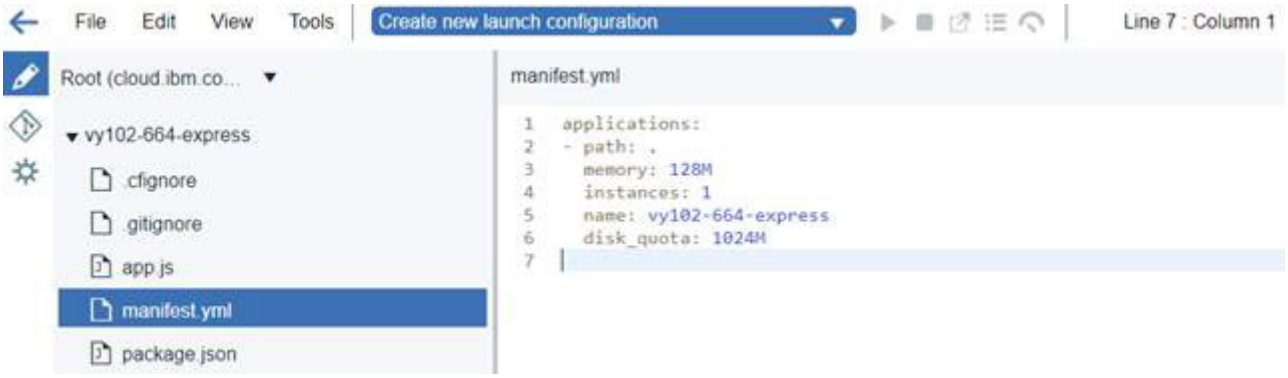
__ b. Name the file `manifest.yml` and press **Enter**.

__ c. Copy the following code snippet to the `manifest.yml` file. This code configures the domain, name, and memory of the application.

In the `manifest.yml` file, replace *XXX* with the three characters that you chose as part of your application and host names.

```
applications:
- path: .
  memory: 128M
  instances: 1
  name: vy102-XXX-express
  disk_quota: 1024M
```

Your `manifest.yml` now looks like the file in the following figure.

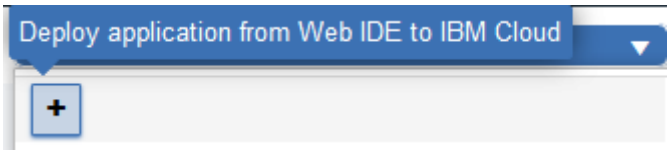


The file is saved automatically. You can force save it by clicking **File** > **Save**.

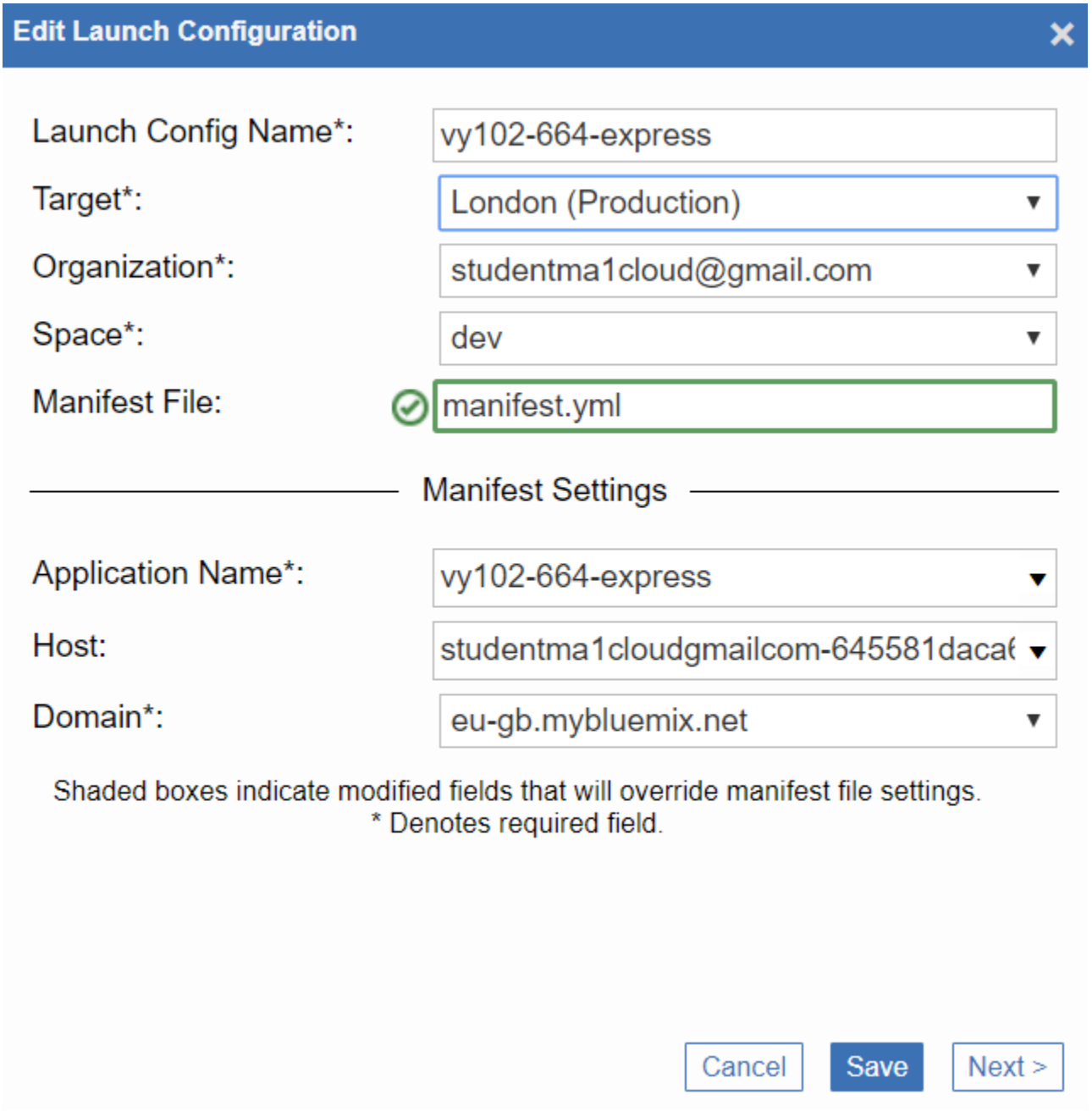
__ 4. Deploy the application from the workspace by completing these steps:

__ a. In the server toolbar, select **Create new launch configuration** from the drop-down list. If you do not have the Create new launch configuration option, skip this step

__ b. Click the **+** button to open the Edit Launch Configuration window (see the following figure).



__ c. In the Edit Launch Configuration window, ensure that **Target** is set to the region where your app was created, wait until the fields load and then ensure that the **Organization** is set to your email address, and **Space** is set to **dev** (see the following figure).



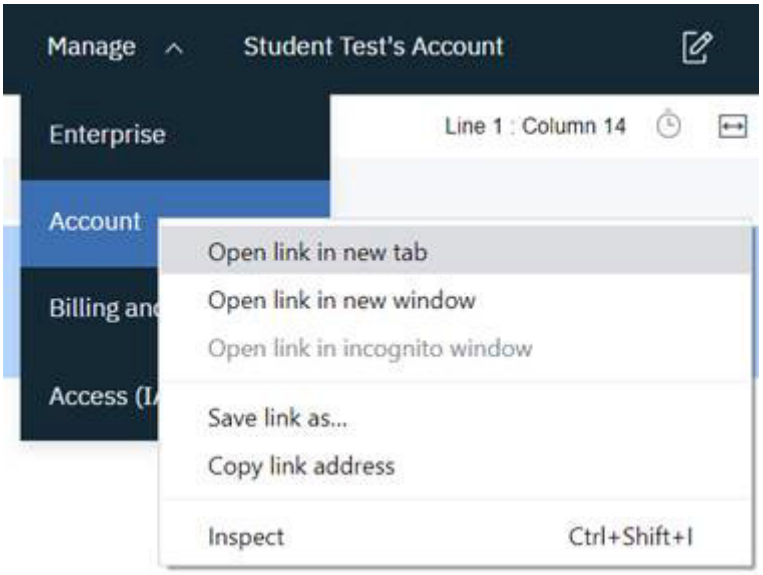
Note

If the message "**`Loading deployment settings...`**" is displayed in the Edit Launch Configuration window for a long time, change the default-selected Target field to the region that matches your account, for example, London. After you select the correct target, the other fields load successfully.

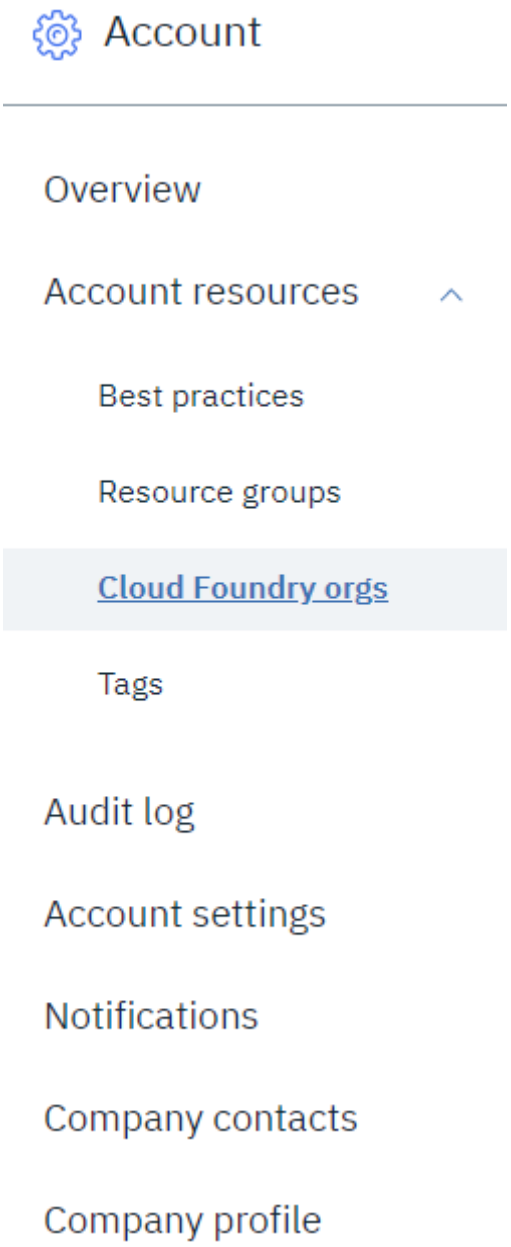
Note

To confirm your Target, follow these steps:

1. In the header, click **Manage** and then right click **Account** and click “Open link in a new tab” (see the following figure).

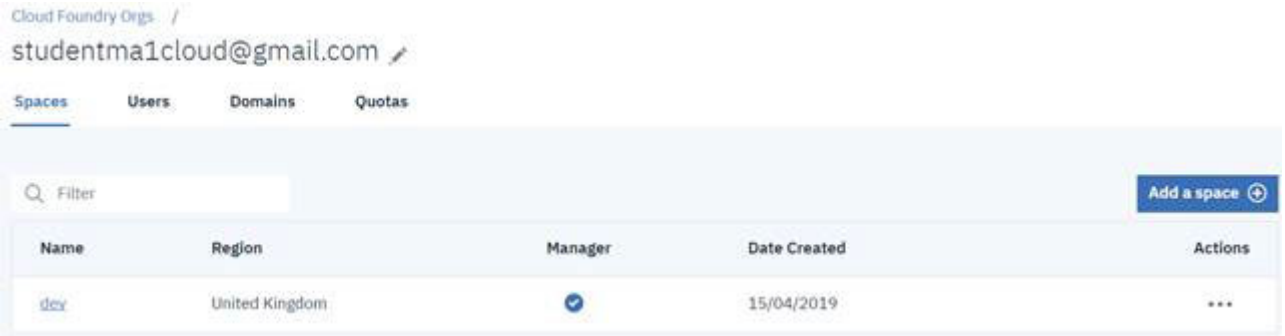


2. In the left menu, click **-Cloud Foundry orgs**, as shown in the following figure.



3. Click your organization's name, as shown in the following figure.

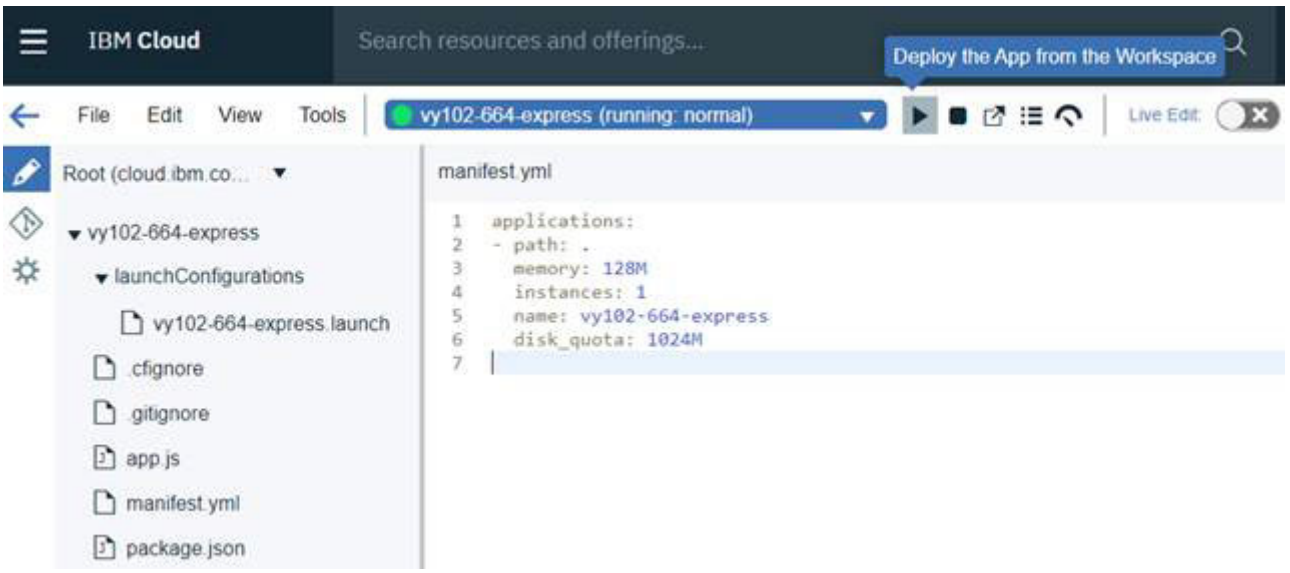
4. Your region and your space are shown, as show in the following figure.



5. If the Region is **United Kingdom**, then confirm that Target is **London (Production)**. If the Region is **US South,** then confirm that Target is **Dallas (Production).** If the Region is **Australia**, then confirm that Target is **Sydney (Production)**.
6. Close the current tab, and go back to the code tab.

__ d. In the “Edit Launch Configuration” window, select the correct target, and then click **Save**.

__ e. Click the play icon (**Deploy the App from the Workspace**), which is highlighted in the following figure. Then, if necessary, click **OK** in the window that opens to confirm the action.



__ f. If the **Stop or Redeploy?** prompt opens, click **OK** to redeploy the app.

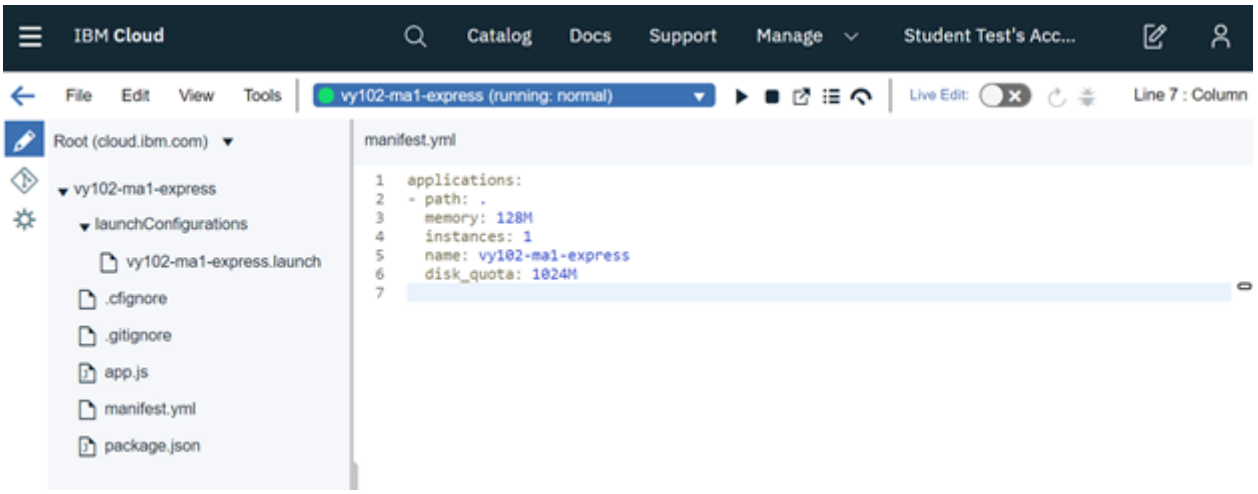
Note

The deployment of the application usually takes a few minutes. If the deployment status changed to “running: normal” in only a few seconds, then the application might not be deployed correctly. In this case, click the play button again to ensure that the deployment is successful.

__ g. The Live Edit and debug options are visible on the top toolbar after the deployment is completed, as shown in the following figure. The Live Edit and debug features are available only for the Node.js application. When IBM Cloud detects that you are creating a Node.js application (from the `manifest.yml` file and the launch configuration), the two features become available on the toolbar.

Note

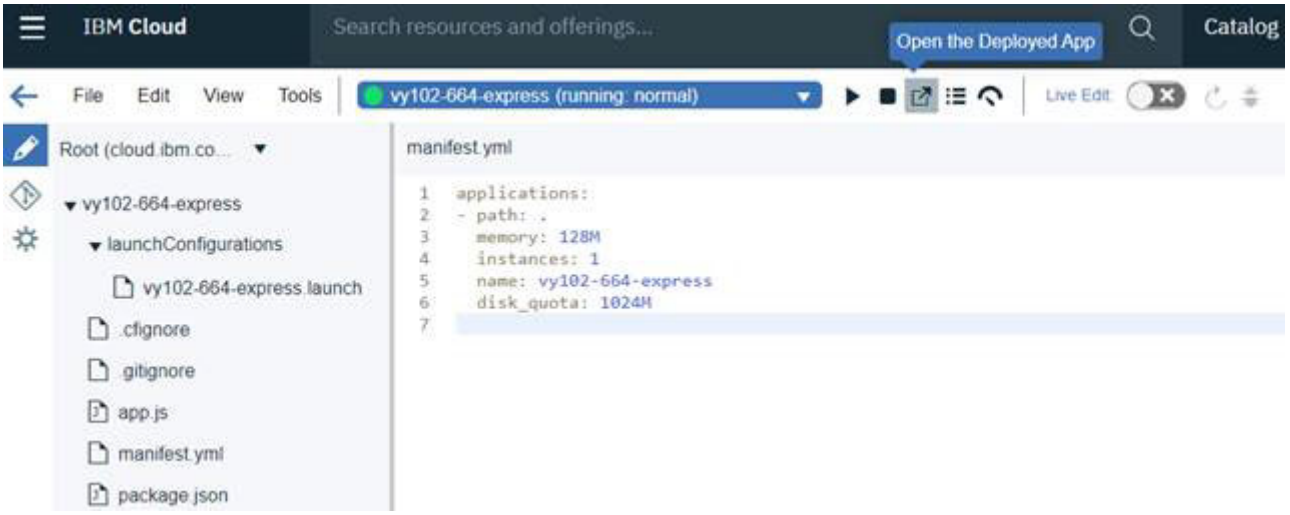
The IBM Cloud Lite account does not support the Live Edit feature. If you are using a Lite account, this option does not work. If you attempt to enable Live Edit, you receive the error `You have exceeded your organization's memory limit: app requested more memory than available`.



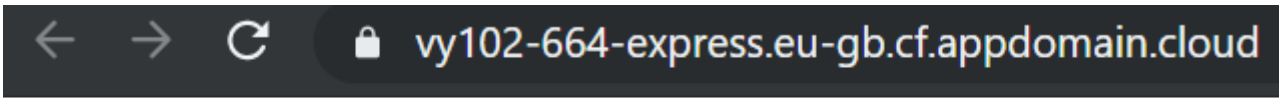
Note

If you receive an error at the bottom with the message “Upload failed”, click the play button again to retry the deployment.

__ 7. Run the application by clicking the **Open the Deployed App** icon, as shown in the following figure.

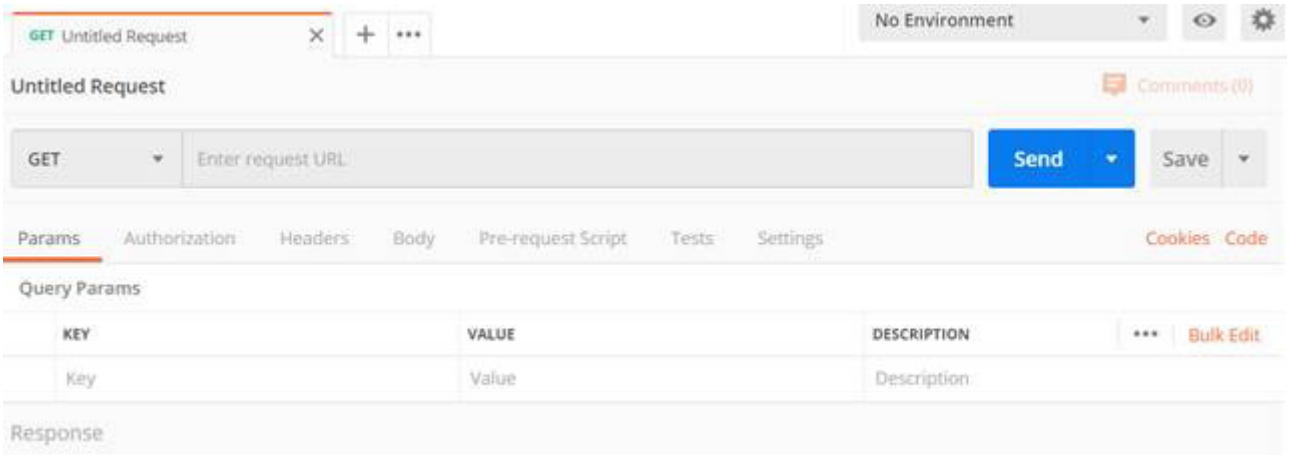


__ 8. A new tab opens with the application, and the “Hello Express!” text is returned, as shown in the following figure.

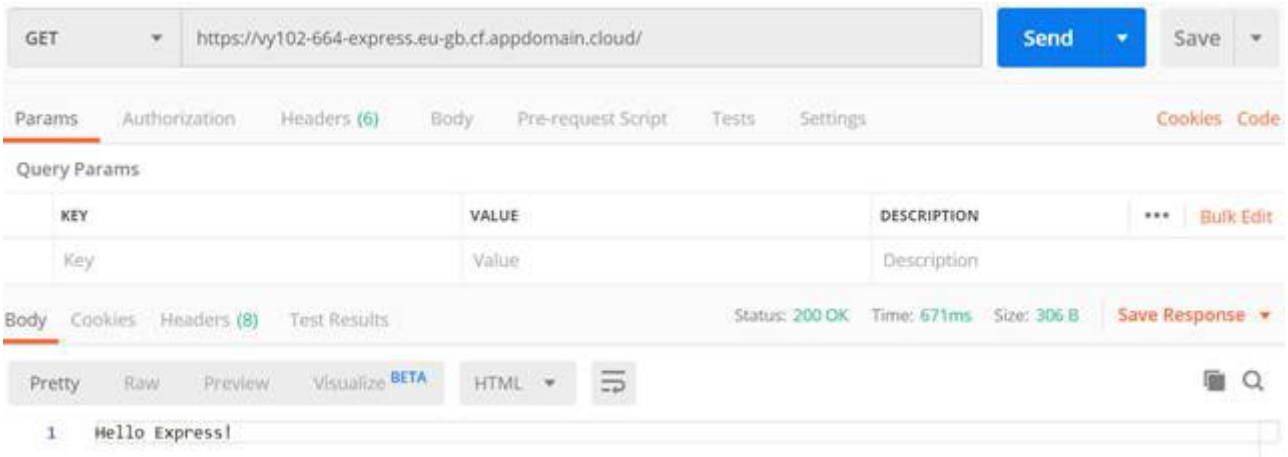


Hello Express!

__ 9. You use an HTTP API client (Postman) to interact with and test the server. Open **Postman** and select the **GET** method, as shown in the following figure.

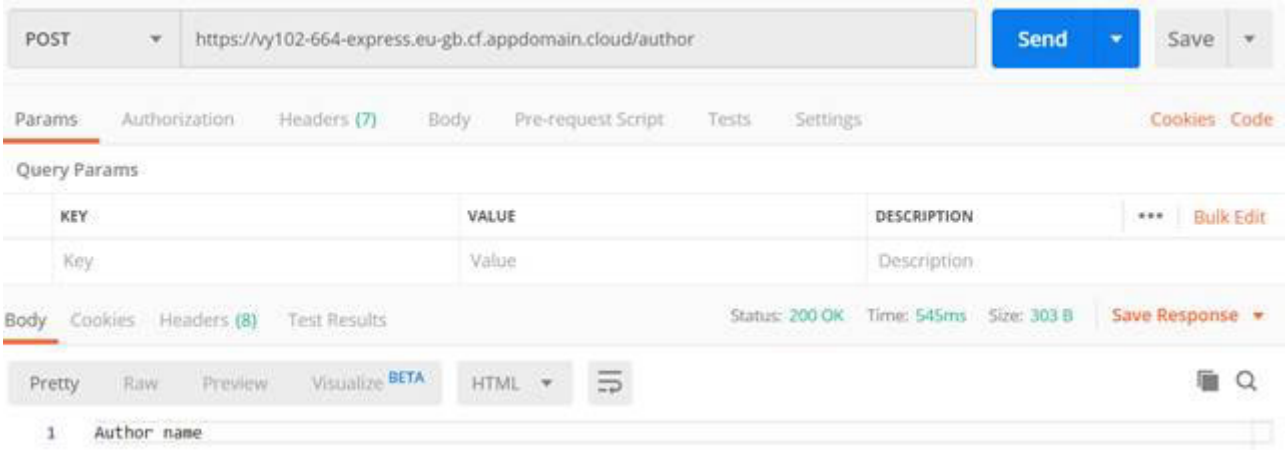


__ 10. Copy your app URL to test the **GET** route request. Paste the URL into the **Enter Request URL** field as shown in the following figure and click **Send**.



The **GET** response is displayed in the body section.

__ 11. To test the **POST** request, change the method to POST and change the URL to the author endpoint, as shown in the following figure. Click **Send**.



Part 4: Creating a simple HTML view and organizing the code

In the next steps, you create a simple HTML page that has a form where the user enters the URL of the article. When the user clicks **Submit**, the article URL is posted to `/author` in the request body.

You also organize the code by creating routing modules instead of handling routing in the app.`js` file.

Complete the following steps:

__ 1. Add `views/index.html` as the starting page, as shown in the following figure:

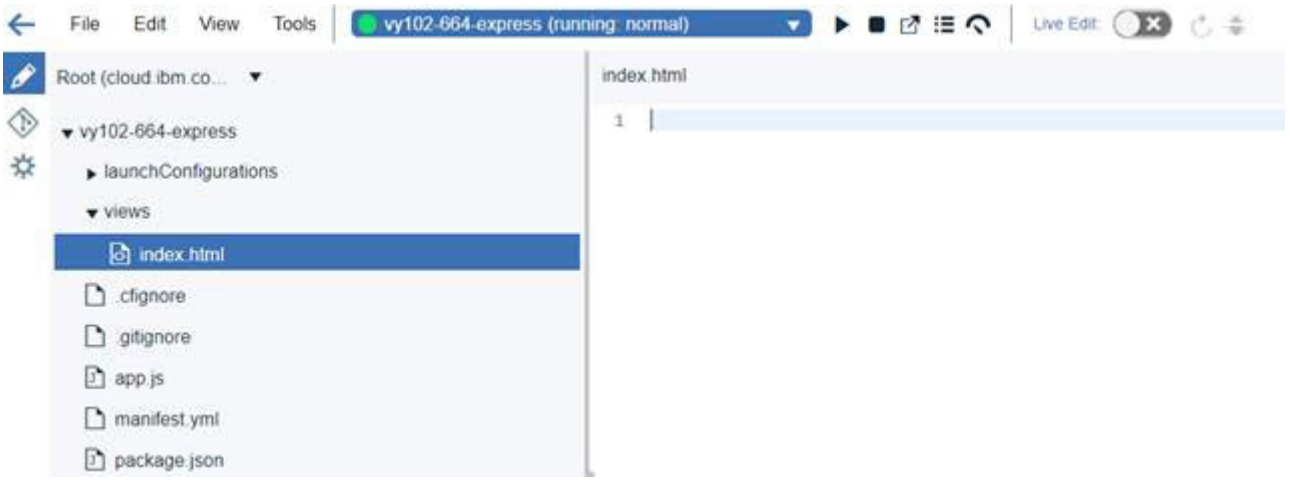
__ a. Close the browser tab where the application is running.

__ b. Right-click **vy102-xxx-express** on the left bar, and select **New** > **Folder**.

__ c. Name the folder `views`, and then press **Enter**.

__ d. Right-click the `views` folder on the left bar, and select **New > File**.

__ e. Name the file `index.html`, and then press **Enter**.



__ 2. Copy the following code snippet into the `index.html` file:

```
<html>
  <body>
    <h1 style="color:blue;">Watson Author Finder</h1>
    <p>To get information about the author of an article, enter the URL of
that article.</p>
    <form action="author" method="post">
        <input type="text" name="url" />
        <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

The HTML code indicates the following information:

- A heading that contains the words “Watson Author Finder” in blue.
- A paragraph that instructs the user what to do.
- A form that contains a text field and a Submit button. Upon submission, the parameters are submitted in the form of `x-www-form-urlencoded` in the body. In this code snippet, the only parameter is the URL of the article. The URL is submitted as a **POST** method to the `author` action, which triggers the **POST** `/author` route.

  The `index.html` file now looks like the one that is shown in the following figure.

index.html

```
<html>
  <body>
    <h1 style="color:blue;">Watson Author Finder</h1>
    <p>To get information about the author of an article, enter the URL of that article.</p>
    <form action="author" method="post">
        <input type="text" name="url" />
        <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

__ 3. In app.js, change the root route (GET /) to send the index.html page to the caller:

__ a. Open app.js from the left navigation bar.

__ b. Add the following code snippet after the line that contains `let app = express();`. This code snippet references the path module. The path module provides utilities for handling the directories, so it must point to the index.html file.

```
let path = require('path');
```

__ c. Update the callback function for the / route, which is the line after `app.get('/', function(req, res) {`. Change the code so that the callback function returns the index.html page instead of the words Hello Express!:

```
res.sendFile(path.join(__dirname, 'views/index.html'));
```

Also, change the comment before the line `app.get('/', function(req, res) {` to the following text:

```
// In case the caller calls GET to the root '/',
// return the content of index.html
```

The following figure shows the app.js file with your updates.

app.js

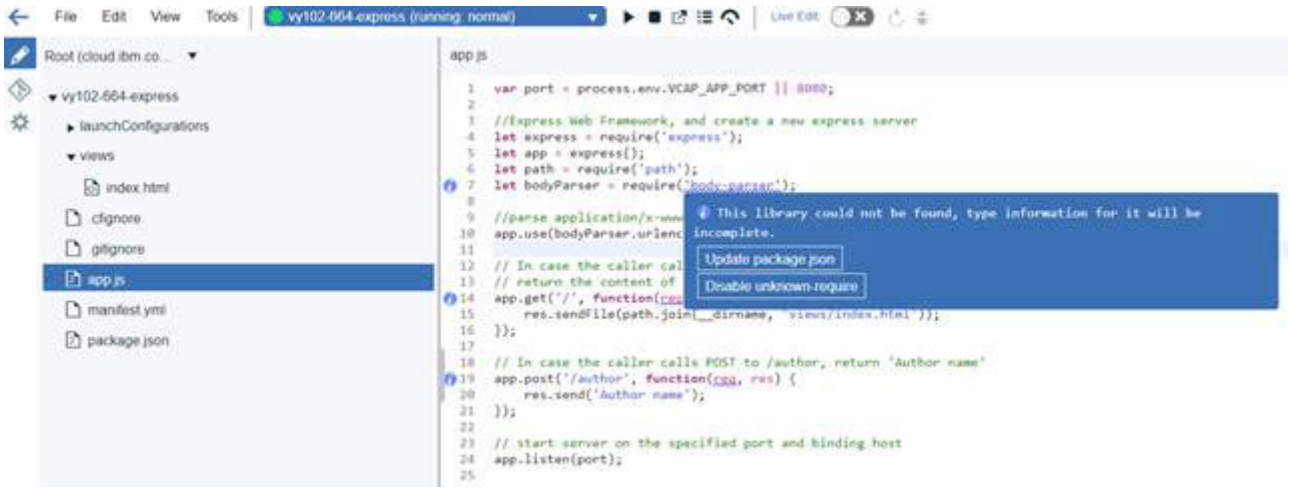
```
1  var port = process.env.VCAP_APP_PORT || 8080;
2
3  //Express Web Framework, and create a new express server
4  let express = require('express');
5  let app = express();
6  let path = require('path');
7  // In case the caller calls GET to the root '/',
8  // return the content of index.html
9  app.get('/', function(req, res) {
10     res.sendFile(path.join(__dirname, 'views/index.html'));
11 });
12
13 // In case the caller calls POST to /author, return 'Author name'
14 app.post('/author', function(req, res) {
15     res.send('Author name');
16 });
17
18 // start server on the specified port and binding host
19 app.listen(port);
20
```

__ 4. To receive the request parameter, you must add a module that is named body-parser. The body-parser middleware module parses the data and populates the request object with the data under the req.body module. Complete these steps:

__ a. *After* the line that contains `let path = require('path');`, add a reference to the third-party body-parser module:

```
let bodyParser = require('body-parser');

//parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }));
```

__ b. Hover the cursor over the information for body-parser and notice the warning message that is generated by Eclipse Orion Web IDE, as shown in the following figure. The message indicates that the body-parser module is not defined in package.json.



__ c. You can either update package.json manually to add dependencies to body-parser or click **Update package.json**, which updates the dependencies automatically.

Open the package.json file and update the dependencies, as shown in the following code.

Note

Make sure that you add a comma after the express dependency line, as shown in the example.

```
"dependencies": {
  "express": "4.*",
  "body-parser": "*"
}
```

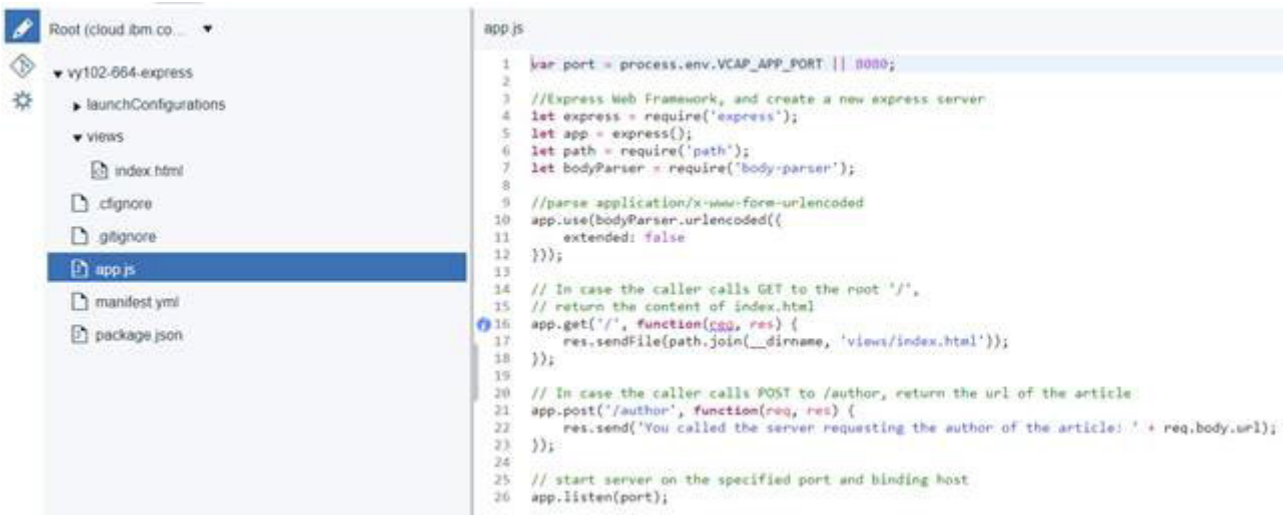
__ d. Open app.js and update the callback function for the POST /author route to send the URL to the user in response to post /author instead of sending “Author name”. Replace `res.send('Author name');` with the following code snippet:

```
res.send('You called the server requesting the author of the article: ' +
req.body.url);
```

Change the comment before the line `app.post('/author', function(req, res) {` to the following text:

```
// In case the caller calls POST to /author, return the url of the article
```

__ e. Press **Shift** + **Alt** + **F** to format the code. Your updated app.js is shown in the following figure.



__ 5. In this step, you organize the code by moving all the routing to the routes module:

__ a. Right-click **vy102-xxx-express** on the left navigation bar, and then select **New** > **Folder**. Name the folder `routes` and press **Enter**.

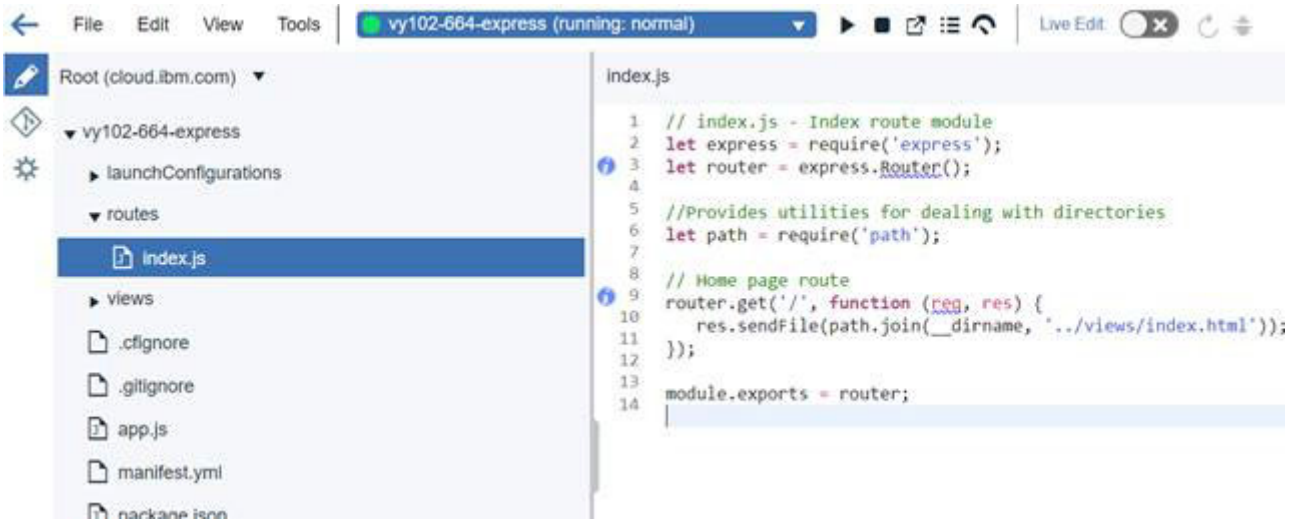
__ b. Create `index.js` to handle all the routing that is related to the root resource, right-click the `routes` folder, select **New** > **File**, name the file `index.js`, and then press **Enter**.

__ c. Copy the following code snippet and paste it into the `index.js` file:

```
// index.js - Index route module
let express = require('express');
let router = express.Router();

//Provides utilities for dealing with directories
let path = require('path');

// Home page route
router.get('/', function (req, res) {
   res.sendFile(path.join(__dirname, '../views/index.html'));
;

module.exports = router;
```

This code snippet uses `express.Router`, which is introduced in Express 4 and provides an isolated instance of routes. It is used here to define an endpoint (URI) that handles the routing when a user sends a **GET** request to the home page of the application for this exercise.

The updated `index.js` file is shown in the following figure.

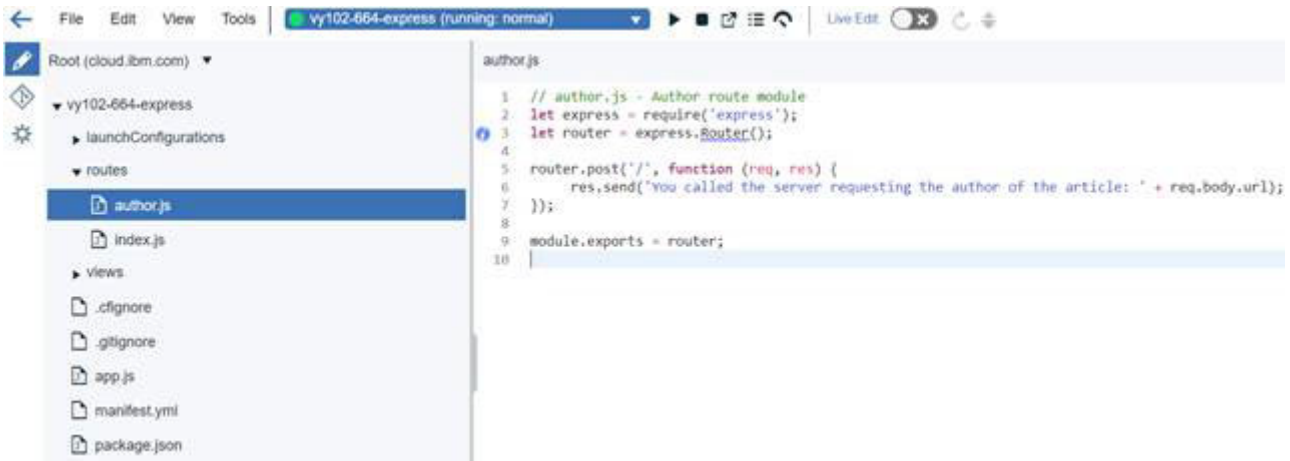


__ d. Create an `author.js` file to handle all the routing that is related to the `/author` resource. Right-click the `routes` folder, select **New** > **File**, name the file `author.js`, and then press **Enter**.

__ e. Copy the following code snippet into the `author.js` file:

```
// author.js - Author route module
let express = require('express');
let router = express.Router();

router.post('/', function (req, res) {
     res.send('You called the server requesting the author of the article:
' + req.body.url);
;

module.exports = router;
```

The following figure shows the updated `author.js` file.



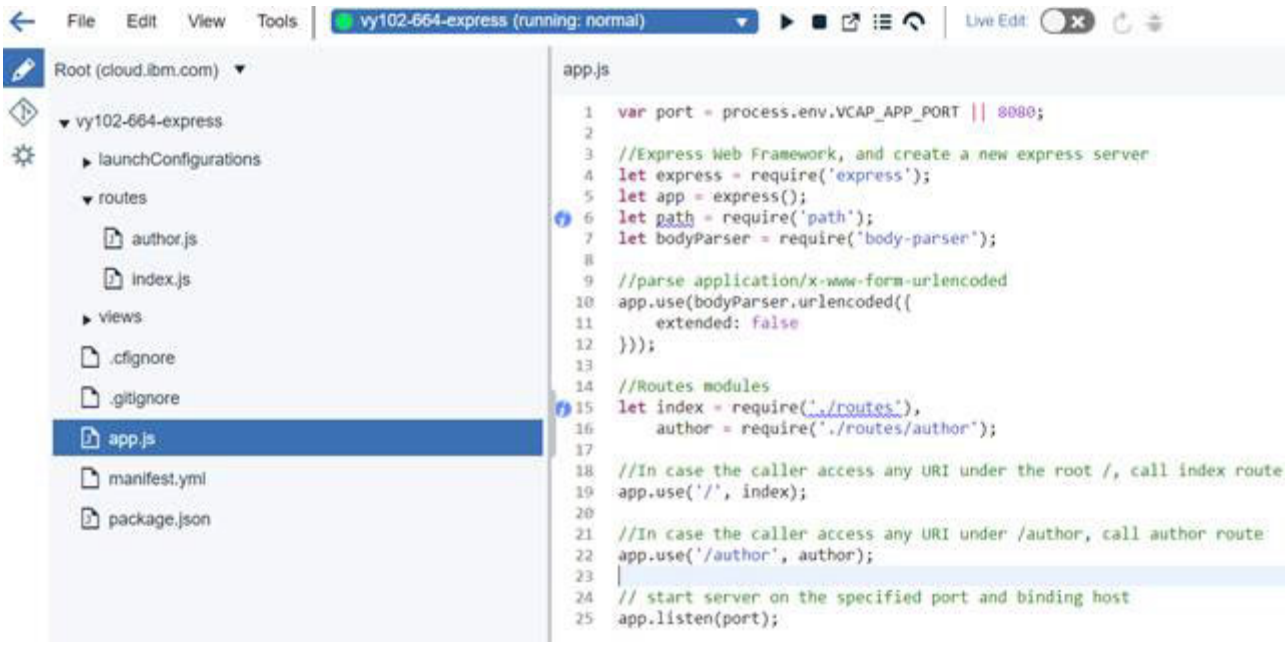
__ f. Edit `app.js` to configure Express to use the index and author route modules:

- Click app.js in the left navigation bar.
- Find the lines of code between the comments `// In case the caller calls GET` and `// start server on the specified port and binding host`.
- Replace those lines, including the comment `// In case the caller calls GET`, with the following code snippet. Remove any unintended line breaks, especially in the comments, after you paste the code.

```
//Routes modules
let index = require('./routes'),
    author = require('./routes/author');

//In case the caller access any URI under the root /, call index route
app.use('/', index);

//In case the caller access any URI under /author, call author route
app.use('/author', author);
```

This example uses app.js to bind the routing modules that you defined previously with specific paths. Routing of any URI under root (/) is handled by the index routing module, and routing of any URI under /author is handled by the author routing module.

Your updated app.js file is shown in the following figure.



__ 6. Deploy the application and run it by completing these steps:

__ a. Click the **Play** icon (**Deploy the App from the Workspace**) on the top toolbar to deploy the app. Confirm that you want to redeploy the app if prompted to do so.

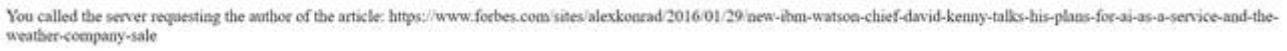
__ b. Wait for the deployment to complete.

__ c. Click the **Open the Deployed App** icon on the top toolbar to run the application.

The application opens in your browser, as shown in the following figure.

__ d. In the text box, enter the URL for an article of your choice and click **Submit**. The route `POST /author` is then called, as shown in the following figure. In this example, the following URL was used for testing:

`https://www.forbes.com/sites/alexkonrad/2016/01/29/new-ibm-watson-chief-david-kenny-talks-his-plans-for-ai-as-a-service-and-the-weather-company-sale`



Part 5: Integrating with the Watson Natural Language Understanding service

The default Node.js framework includes only a minimal set of features. However, a large community of developers add to the Node.js framework through third-party libraries.

In this section, you extract the author name by calling the Watson Natural Language Understanding service. IBM Watson services provide REST APIs that you can use to add AI capabilities to your applications.

Watson Natural Language Understanding uses natural language processing (NLP) to analyze semantic features of any text, which can be plain text, HTML, or a public URL. Watson Natural Language Understanding returns results for the features that you specify. The feature that you use in this exercise is metadata. It gets document metadata, including author name, title, RSS/Atom feeds, prominent page image, and publication date.

In Node.js, the Watson APIs are wrapped inside a third-party module that is named `watson-developer-cloud` that you use in this exercise. The `watson-developer-cloud` module is developed and maintained by IBM. It is a third-party module because it is not developed as part of the Node.js foundation, and not packed into Node.js by default. To include the `watson-developer-cloud` module, you must add it as a dependency in the `package.json` file.

Complete these steps:

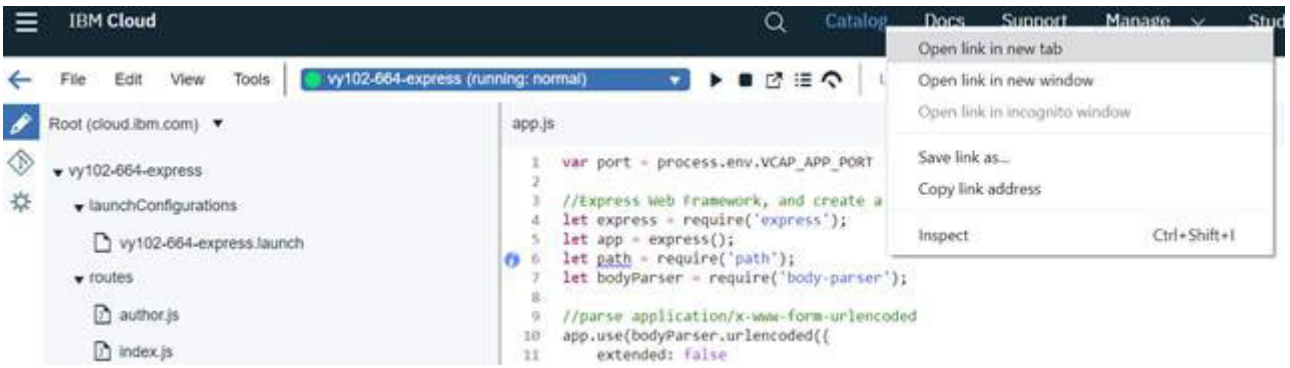
__ 1. In this step, you add the Watson Natural Language Understanding service and bind it to the application.

IBM Cloud services are a set of capabilities or functions that are delivered over the internet that IBM Cloud hosts and manages. You can add services from the IBM Cloud catalog to your IBM Cloud application. Services provide a predefined endpoint that you can access from your application. The infrastructure for services is managed by IBM Cloud, and your app must use only the provided endpoint.

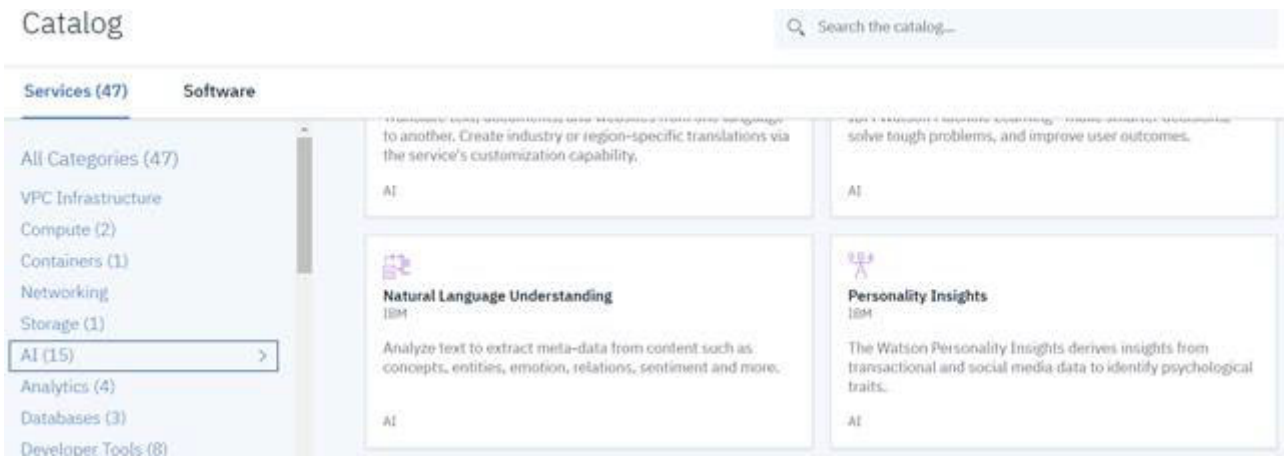
You bind the Watson Natural Language Understanding service to your application so that your application can use it by completing these steps:

__ a. Close the running application tab and go back to the code editor page.

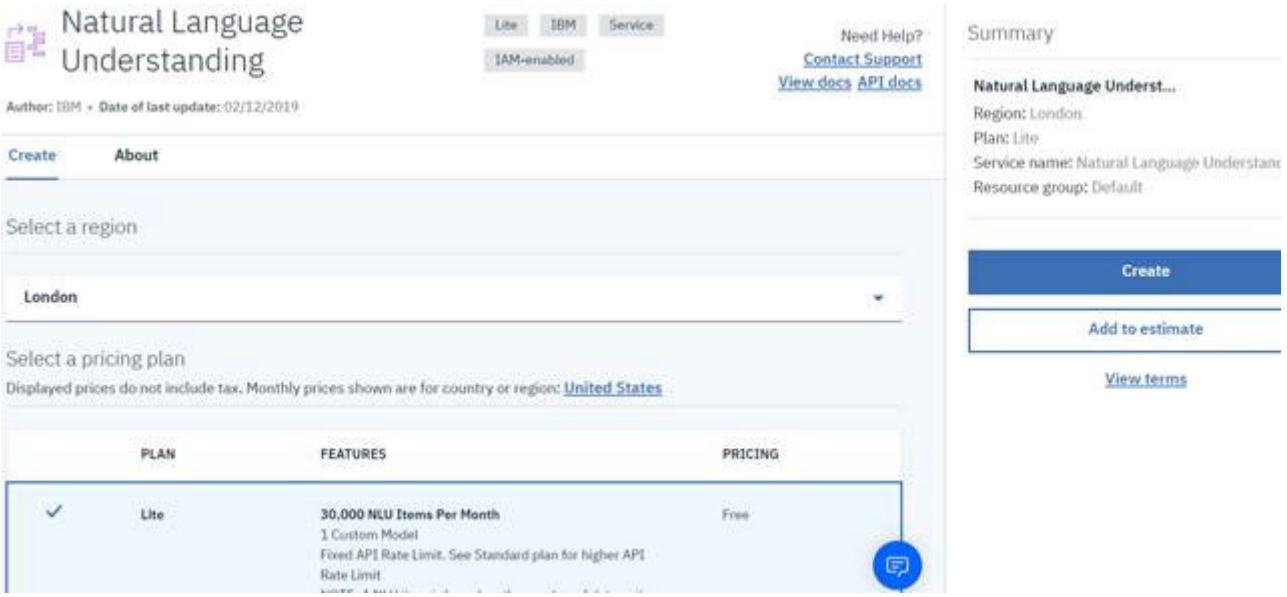
__ b. Right-click **Catalog** and select **Open link in new tab**, as shown in the following figure.



__ c. In the IBM Cloud Catalog, click **Natural Language Understanding** from the **AI** section, as shown in the following figure.



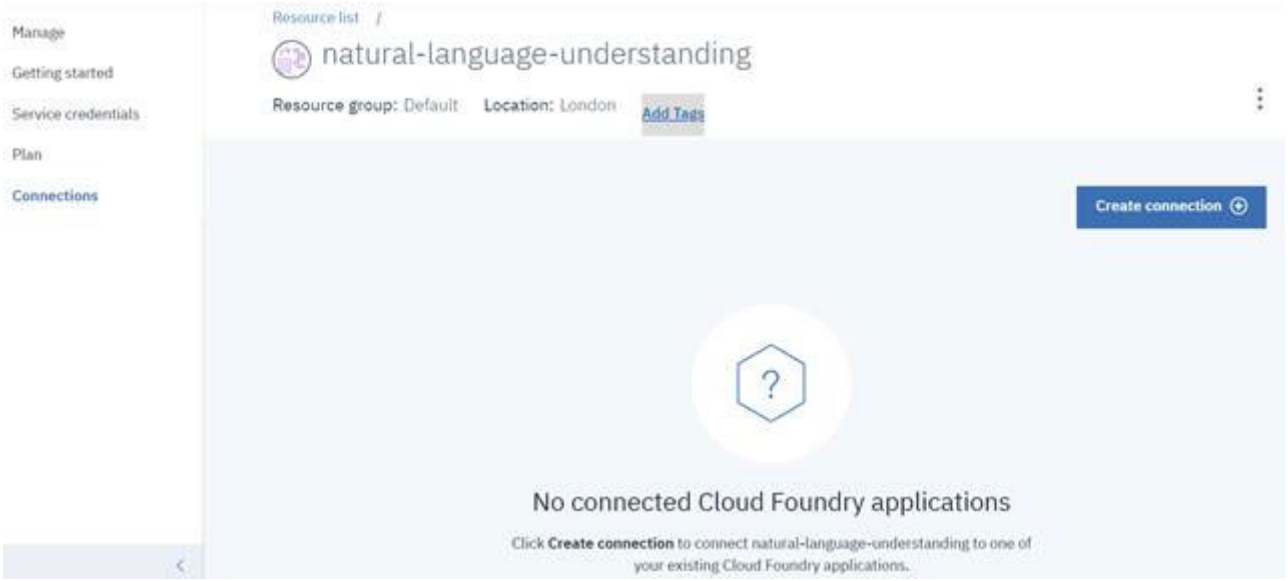
The Natural Language Understanding page opens, as shown in the following figure.



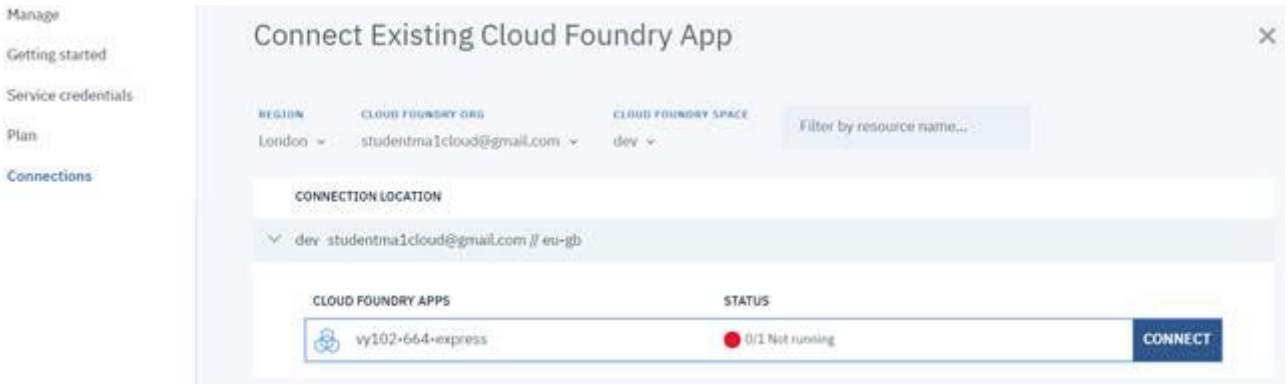
__ d. Keep the Lite plan selected, and then scroll down to the **Configure your resource** section. Change the **Service name** field to `natural-language-understanding`. This is the name of the service. The **Configure your resource** section looks as shown in the following figure.

__ e. Click **Create**.

__ f. Click **Connections** in the left pane and then click **Create connection**.



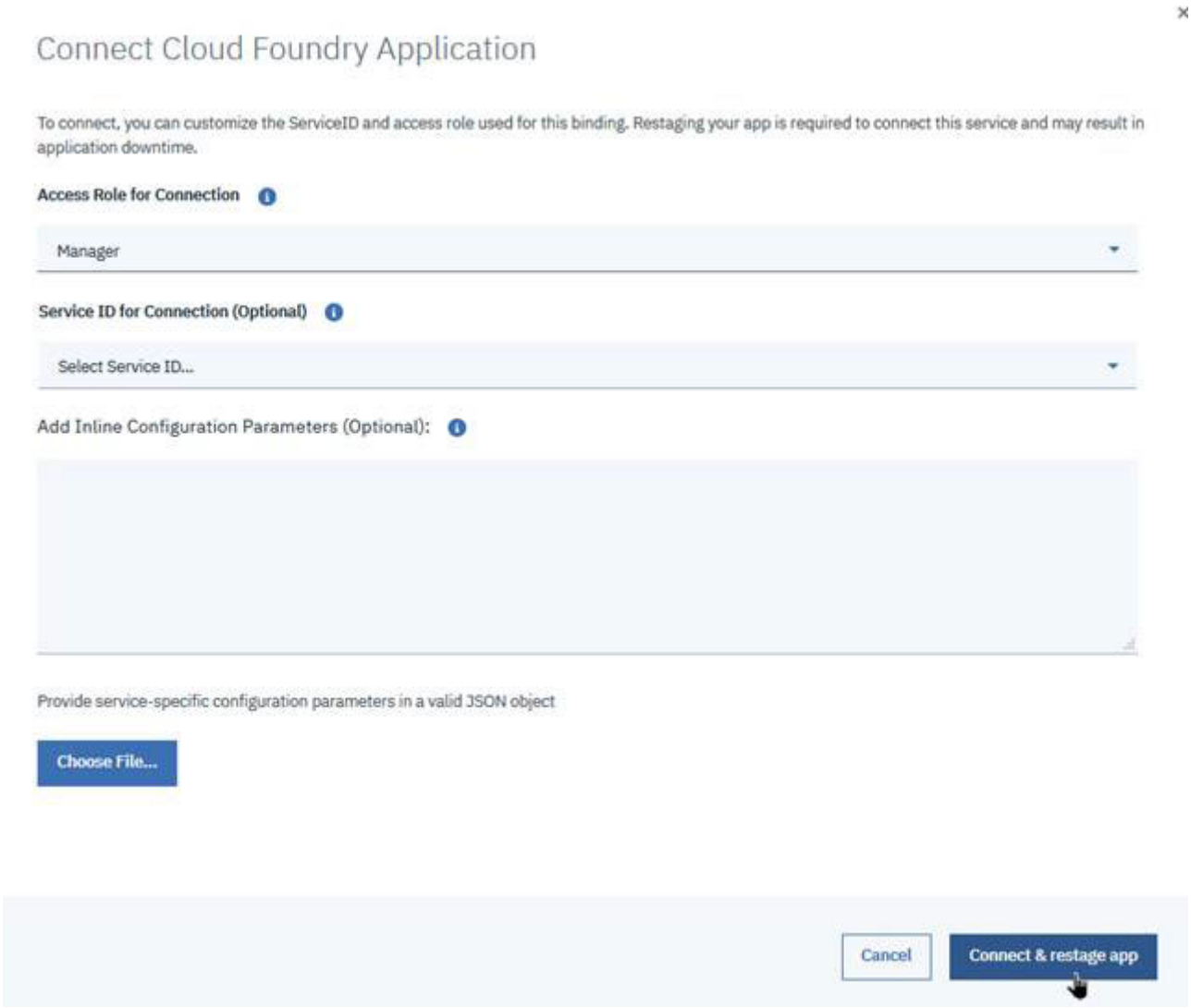
__ g. Select your application **vy102-XXX-express**, and click **Connect**, as shown in the following figure.



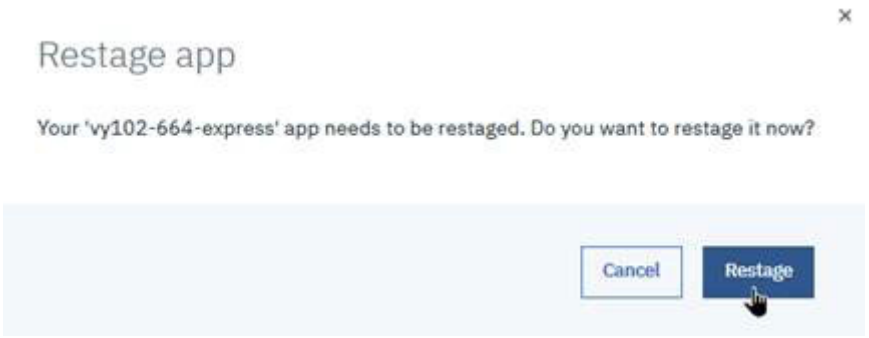
Important

You *must* perform this step because it binds the instance of the Watson Natural Language Understanding service to the application so that your application can use the service.

__ h. To make the service available for use, click **Connect & restage app** in the dialog window that opens, as shown in the following figure. Restaging the application means redeploying the application.



__ i. Click **Restage** in the Restage app dialog box.



__ j. Close the IBM Watson service page tab to return.

__ 2. For integration with Watson, you use a module that is named `watson-developer-cloud`. Add a dependency for it into the `package.json` file:

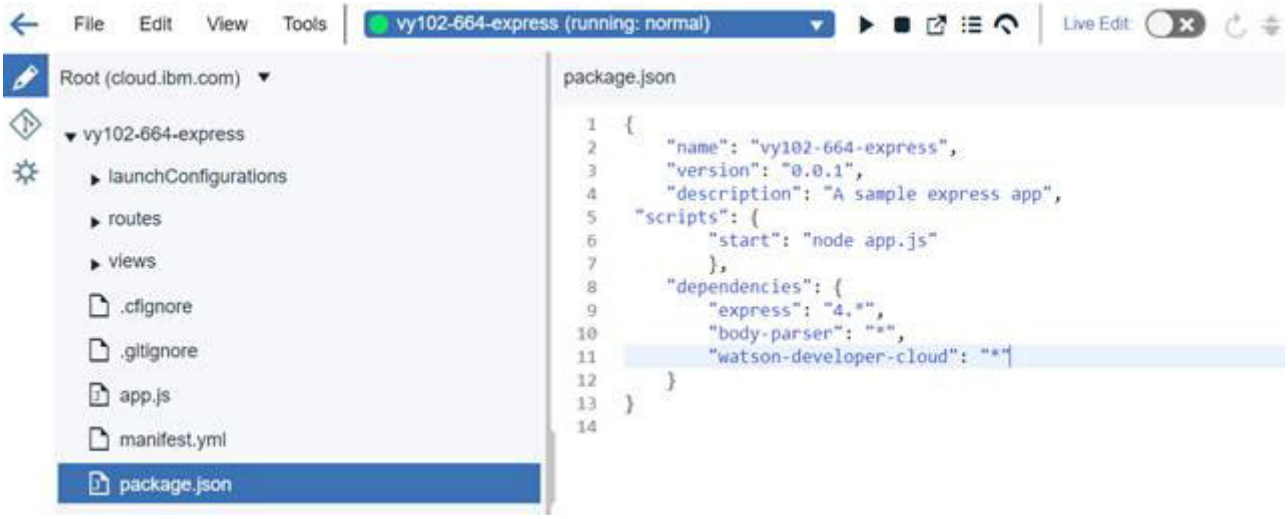
__ a. Open the `package.json` file.

__ b. Add `watson-developer-cloud` as a dependency in `package.json`:

- Find the line “`body-parser": “*”` (it can also be `"body-parser": "latest"` if it was added automatically before). Add a comma (,) at the end of the line and press **Enter**.
- In the new line, add the following code snippet:

  `"watson-developer-cloud": "*"`

The updated `package.json` is shown in the following figure.



__ 3. Create a module that is named `articleServices` to handle all the business logic that is related to the articles. This module has the function `extractArticleAuthorNames` that calls the Watson Natural Language Understanding service, passes the article URL to it, and returns the list of author names. Complete these steps:

__ a. Right-click **vy102-xxx-express** in the left navigation bar and then select **New** > **Folder**. Name the folder `services`, and then press **Enter**.

__ b. Create `articleServices.js`. Right-click the `services` folder, select **New** > **File**, name the file `articleServices.js`, and press **Enter**.

__ c. Create an instance of the third-party Node.js module `watson-developer-cloud` by copying the following code snippet to `articleServices.js`. You do not need to remember or save the IAM_apikey entry of the service because you can call it anytime by using **VCAP_SERVICES**.

```
//initializing env variable
   let env = JSON.parse(process.env.VCAP_SERVICES);
// Watson Natural Language Understanding third-party module
//Specify the release for the Natural Language Understanding service
let NaturalLanguageUnderstandingV1 =
require('watson-developer-cloud/natural-language-understanding/v1.js');
let natural_language_understanding = new NaturalLanguageUnderstandingV1({
  version: '2018-11-16',
iam_apikey:
env["natural-language-understanding"][0]["credentials"]["apikey"],
url:
'https://gateway-lon.watsonplatform.net/natural-language-understanding/ap
i'
;
```

Unknown dependency: Eclipse Orion might produce an unknown dependency warning message. You can ignore this warning because the parent module `watson-developer-cloud` was included previously. The `watson-developer-cloud` module contains the `natural-language-understanding` module.

Note

The VCAP_SERVICES environment variable contains information that you can use to interact with a service instance in IBM Cloud. The fields in this environment variable are set when you bind a service instance to an application.

__ d. Create a function that is named `extractArticleAuthorNames`, as shown in the following code snippet. This function calls the Watson Natural Language Understanding service to extract the authors' names. Add the following code snippet to the end of the `articleServices.js` file.

```
/*
 * Call Watson NLU Service to extract the list of author names for the
requested article URL
*/
exports.extractArticleAuthorNames = function(req){
};
```

__ 4. Prepare the parameters for calling the Watson Natural Language Understanding service by placing the following code snippet inside the `extractArticleAuthorNames` function:

```
// url is the parameter that is passed in the POST request to /author
// It contains the URL of the article
// The metadata feature returns the author, title, and publication date.
    var parameters = {
        'url': req.body.url,
        'features': {
            'metadata': {}
        }
    };
```

__ 5. Call the Watson Natural Language Understanding service and return to the caller a callback function that contains the author's name. Complete these steps:

__ a. Add `callback` as a parameter to the function:

```
exports.extractArticleAuthorNames = function(req, callback){
```

__ b. Call the **`analyze`** function from the `natural_language_understanding` third-party module. If the request succeeds, it returns a metadata object of the article. In this exercise, you are interested in the `authors` object in the metadata that returns the list of authors for the article.

Copy the following code snippet to the **`extractArticleAuthorNames`** function after the parameters' variable initialization:

```
// Call the Watson service and return the list of authors
    natural_language_understanding.analyze(parameters,  function(err,
response) {
        if (err)
            callback(err,null);
        else
            callback(null,response.metadata.authors);
    });
```

__ 6. If the URL that is passed is empty, the user receives an error message. Complete the following steps:

__ a. Add the following error message as a constant after the initialization of the `natural_language_understanding` variable:

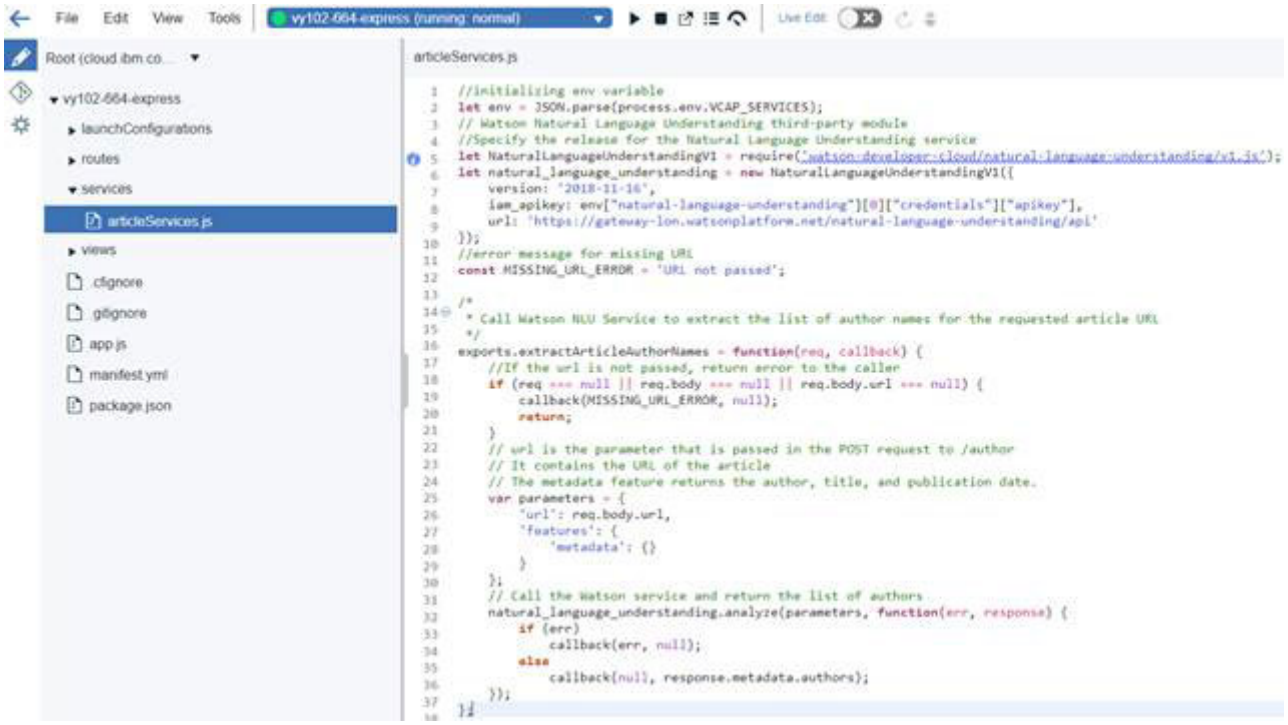
```
//error message for missing URL
const MISSING_URL_ERROR = 'URL not passed';
```

__ b. Check whether the URL is empty. At the beginning of the **`extractArticleAuthorNames`** function, (see the line `exports.extractArticleAuthorNames = function(req, callback){`), add the following code snippet to return the error message if the URL is not defined:

```
//If the url is not passed, return error to the caller
if(req===null||req.body===null||req.body.url===null){
      callback(MISSING_URL_ERROR,null);
      return;
}
```

__ c. Press **Alt** + **Shift** + **F** to format the code.

The complete `articleServices.js` file now looks like the file in the following figure.



__ 7. Edit the author route to call `authorServices.extractArticleAuthorNames` instead of just returning the article URL by completing these steps:

__ a. From the navigation bar, open the `routes` folder and select the `author.js` file.

__ b. Add a reference to the `authorServices` module after the router variable initialization:

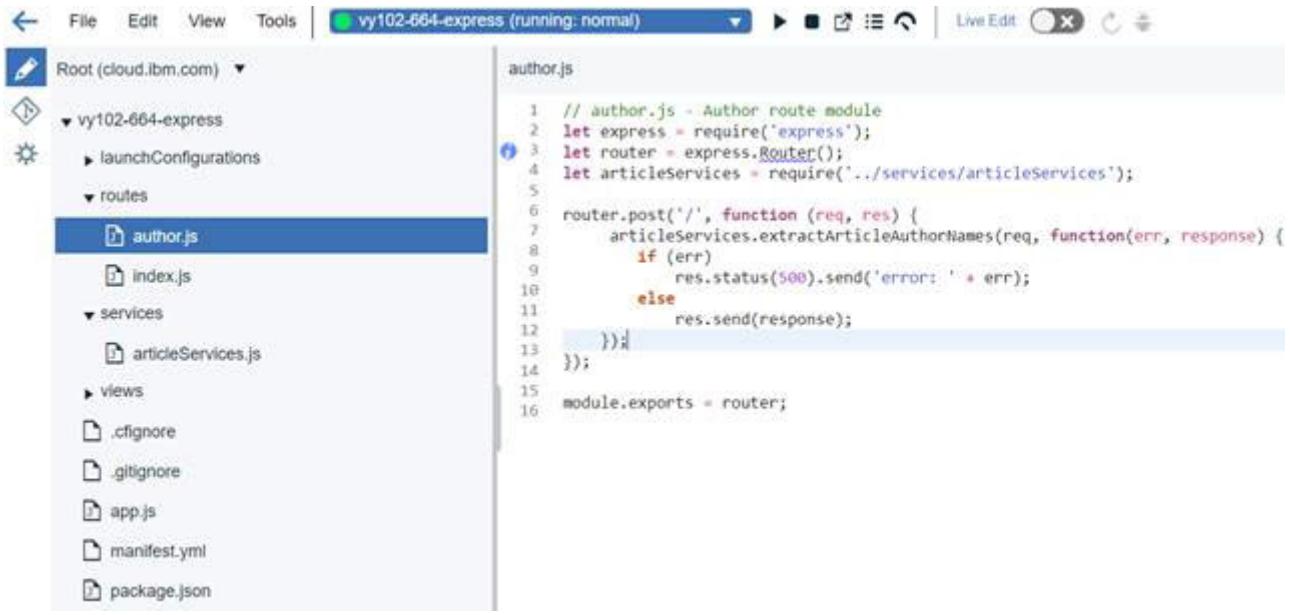
```
let articleServices = require('../services/articleServices');
```

__ c. Edit `/ route` to call the **`extractArticleAuthorNames`** function from the `articleServices` module. This process involves replacing the line `res.send('You called the server requesting the author of the article: ' + req.body.url);` with the following code snippet:

```
articleServices.extractArticleAuthorNames(req, function(err, response)
{
        if (err)
            res.status(500).send('error: ' + err);
        else
            res.send(response);
    });
```

The code indicates that if there is an error, a 500 status code is returned to the user, which means Internal Server Error.

The updated `author.js` file is shown in the following figure.



Part 6: Deploying the application and running it

In addition to deploying the application from Eclipse Orion Web IDE, there are other ways to deploy it. In the next steps, you push the code to the Git repository, and then the IBM Cloud Delivery Pipeline automatically builds and deploys the code.

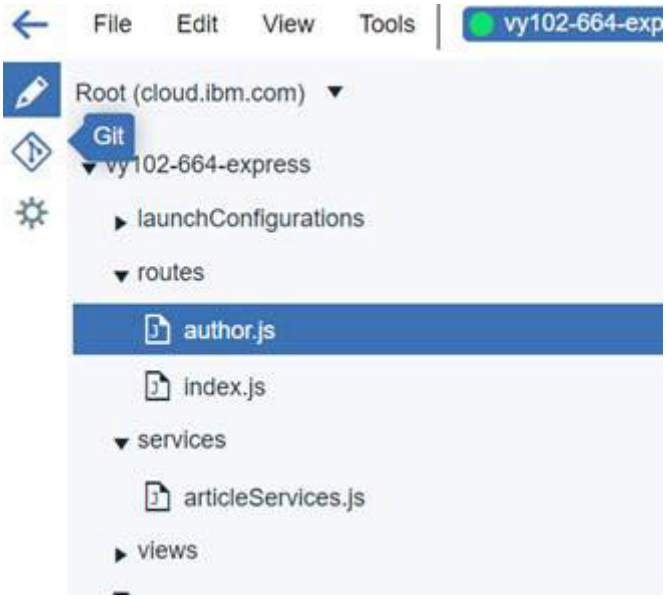
By default, enabling continuous delivery for a project creates a DevOps toolchain for your project. The toolchain includes a Git repository that is based on GitLab. *Git* is an open source change management system.

The Git repository perspective in IBM Cloud Continuous Delivery supports common Git commands to manage your code. You can also develop your application on your own workstation and commit your changes to the Git repository by using a standard Git client.

By default, IBM Cloud Delivery Pipeline services automatically run the build and deploy tasks when you commit changes to the Git repository.

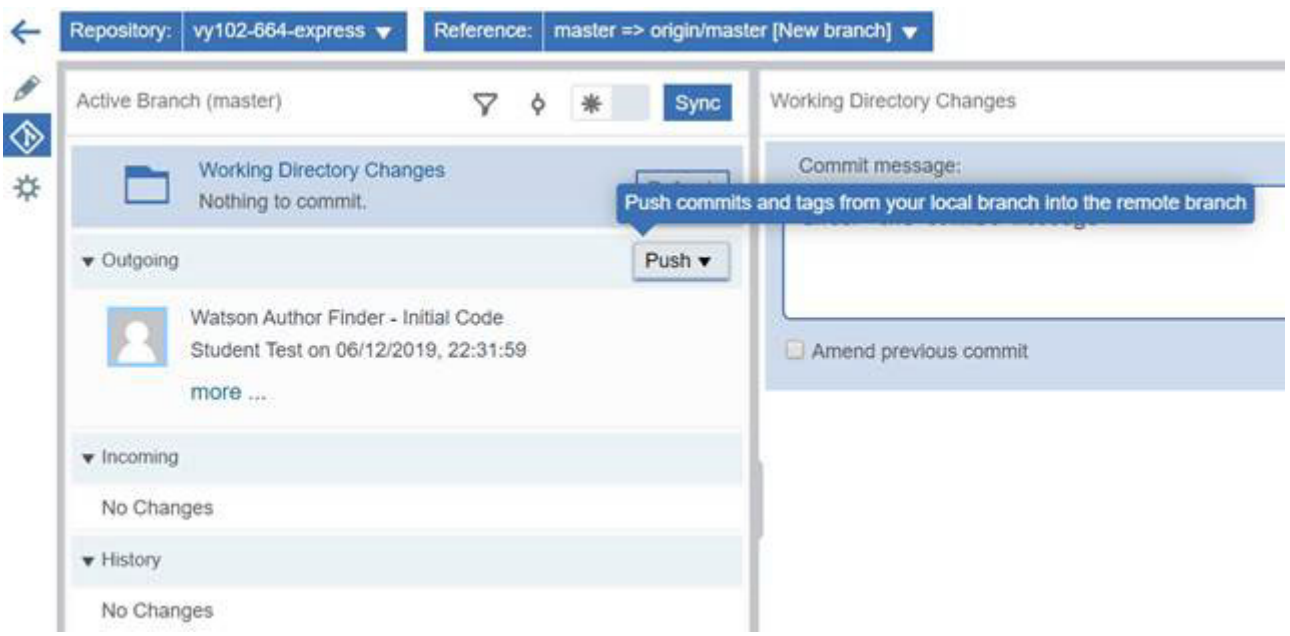
Complete these steps:

__ 1. Switch to the Git perspective by clicking the **Git** icon in the left toolbar (highlighted in the following figure).



__ 2. Notice that all the changed files are listed in the Working Directory Changes window, as shown in the following figure. Enter a descriptive commit message, for example, `Watson Author Finder - Initial Code`, and then click **Commit**.

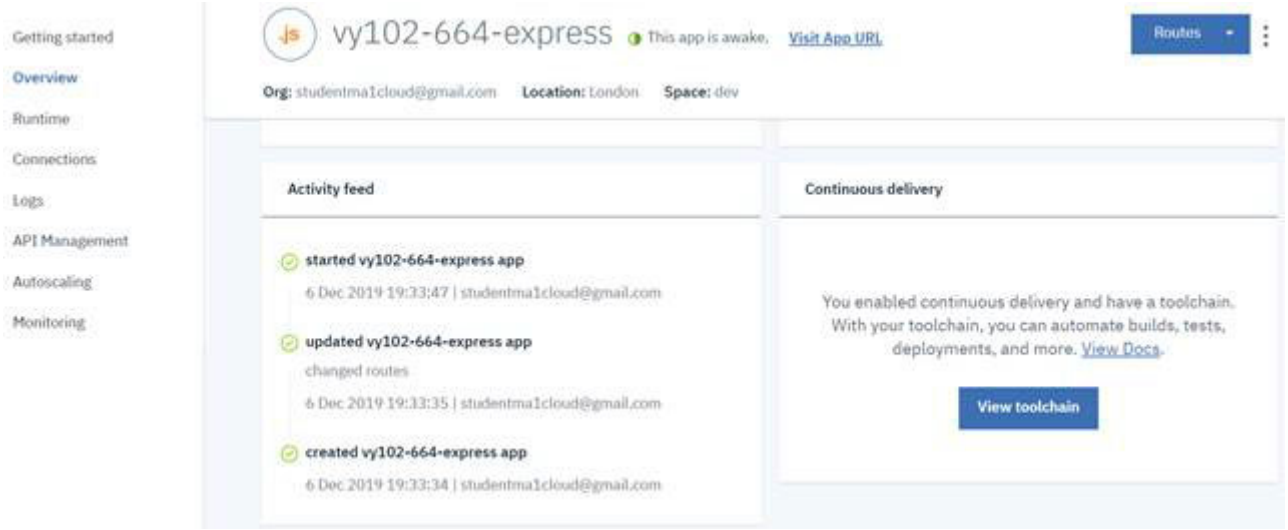
__ 3. Your change is shown in the outgoing window, as shown in the following figure. Click **Push**.



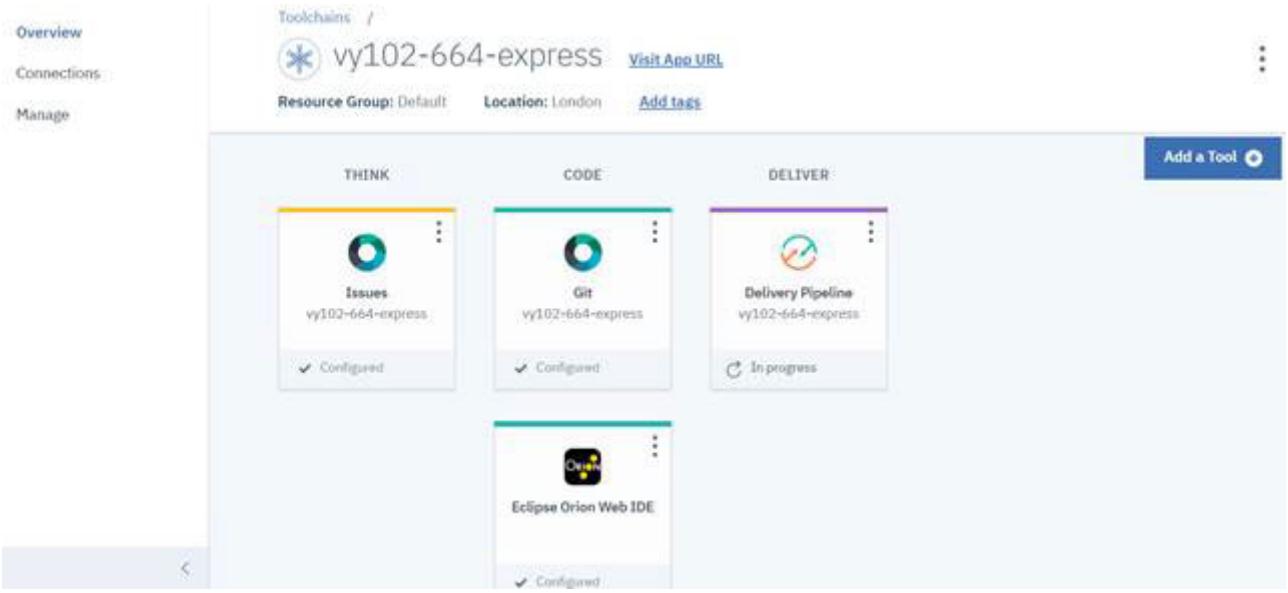
__ 4. After the application is pushed to Git, the delivery pipeline automatically builds and deploys the application. Complete these steps:

__ a. Close the Git tab and return to the Application Details window.

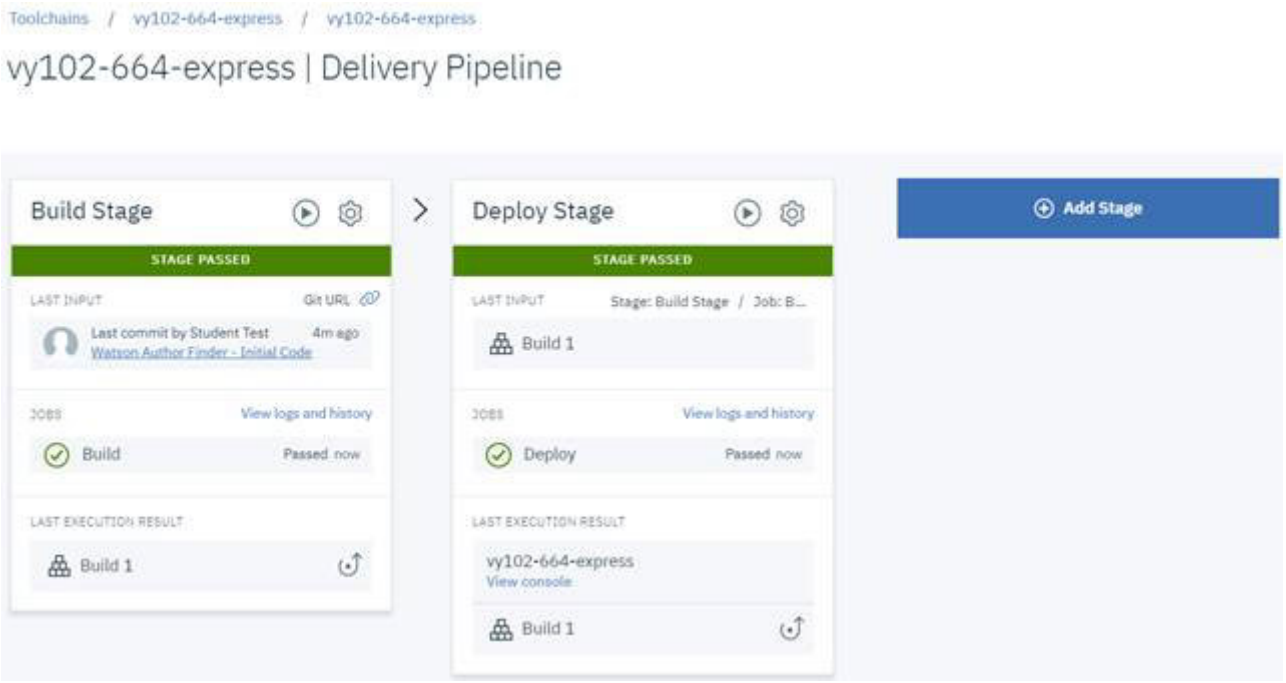
__ b. Scroll down to the Continuous delivery tile and then click **View toolchain**, as shown in the following figure.



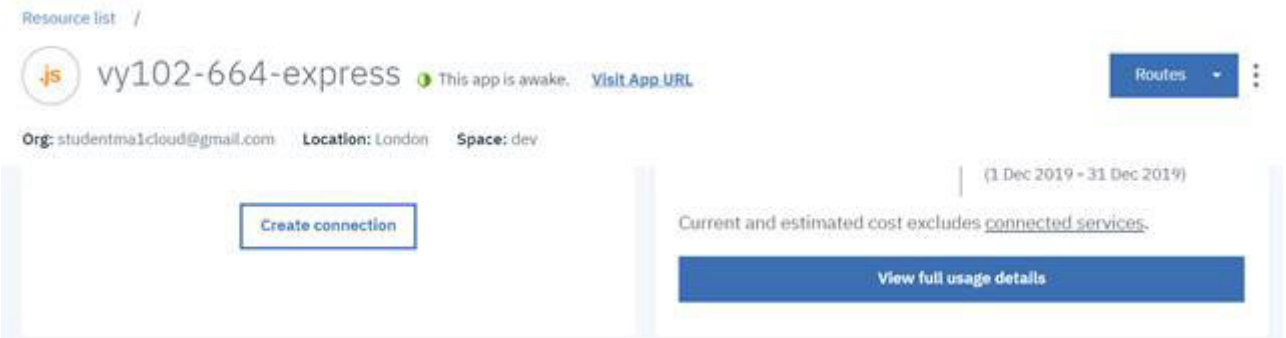
__ c. Click **Delivery Pipeline**, as shown in the following figure.



__ d. Wait until all the jobs in the Build Stage and Deploy Stage complete. The following figure shows that both stages are complete.



__ e. To run the application, go back to the app details and click **Visit App URL**, as shown in the following figure.

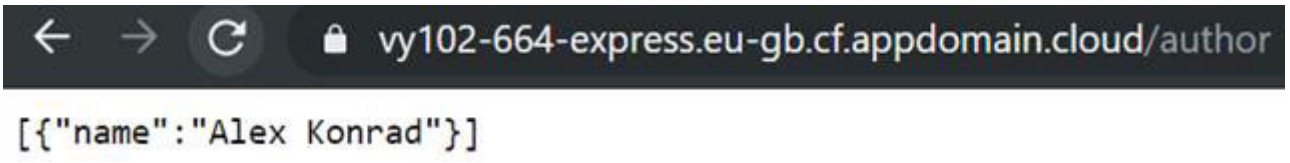


__ f. Run the application by entering the URL of any article and then clicking **Submit**, as shown in the following figure.

Here is an example of an article URL:

```
https://www.forbes.com/sites/alexkonrad/2016/01/29/new-ibm-watson-chief-david-kenny-talks-his-plans-for-ai-as-a-service-and-the-weather-company-sale
```

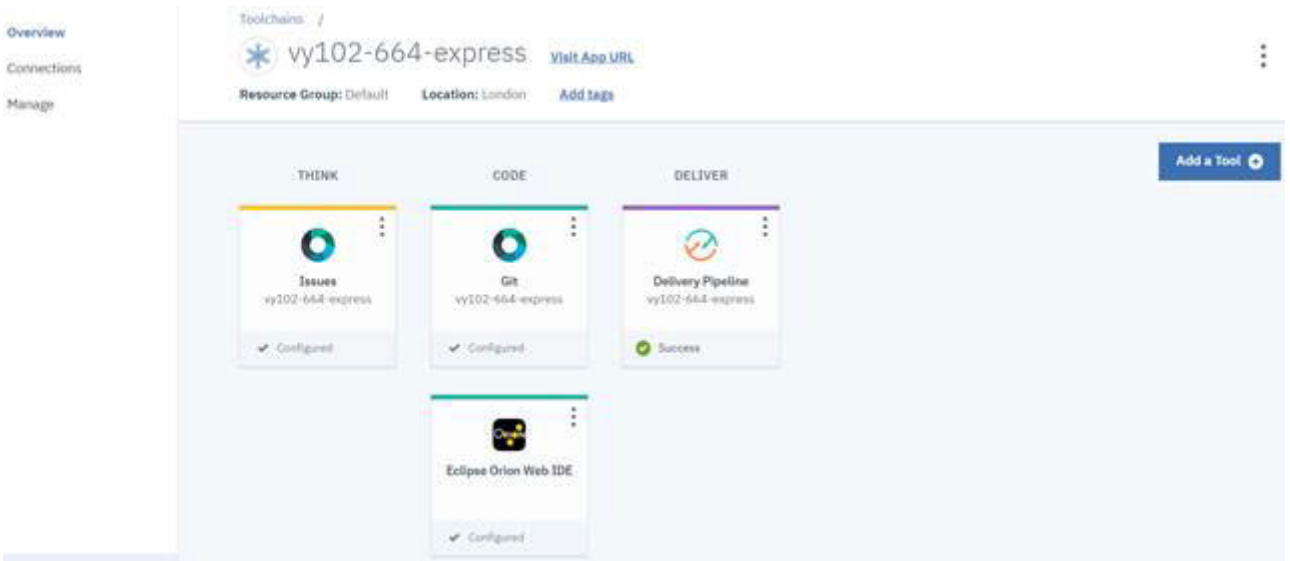
The author name is retrieved from the Watson Natural Language Understanding service (Watson Author Finder returned results), as shown in the following figure.



Part 7: Cleaning up the environment

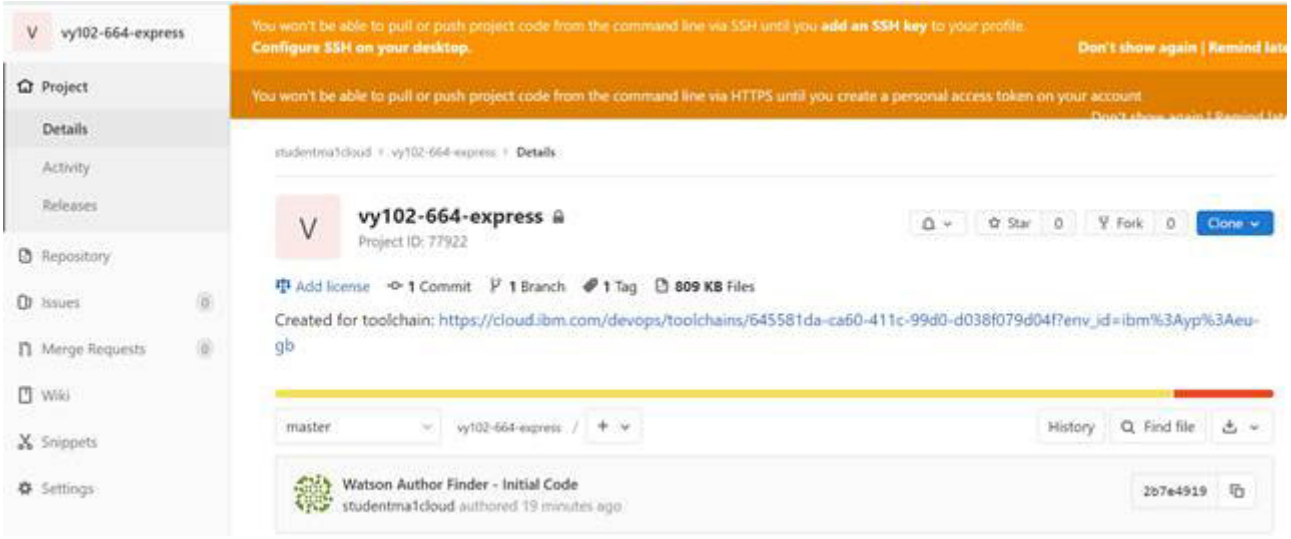
In this part, you delete the application to free resources for the next exercise:

__ 1. Open the toolchain dashboard.



__ 2. Click **Git**.

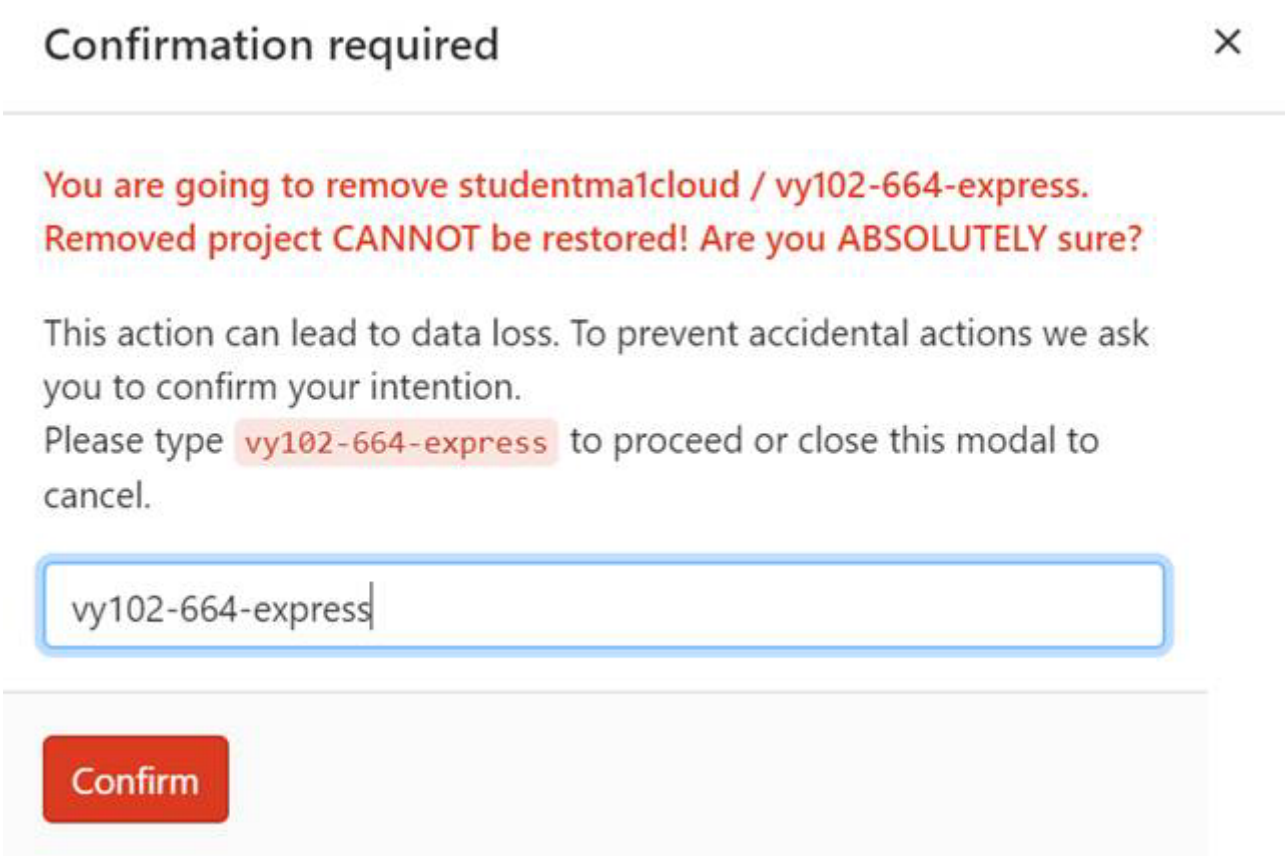
__ 3. From the left bar, click **Settings**.



__ 4. Scroll down and click **Expand** in the **Advanced** section.

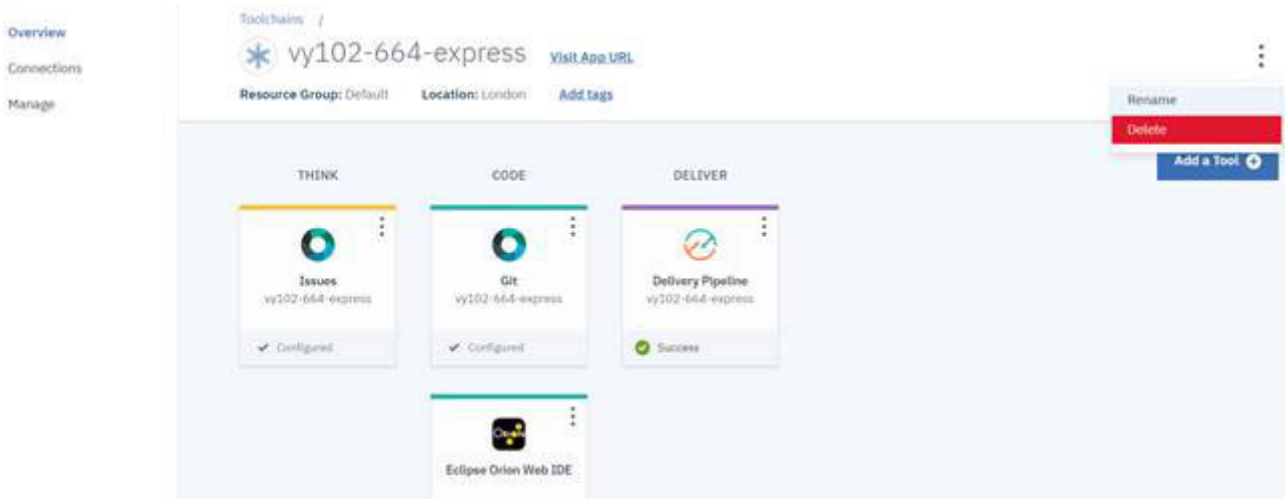
__ 5. Scroll down to Remove project and click **Remove project**.

__ 6. At the confirmation window, type the name of your project and click **Confirm**.

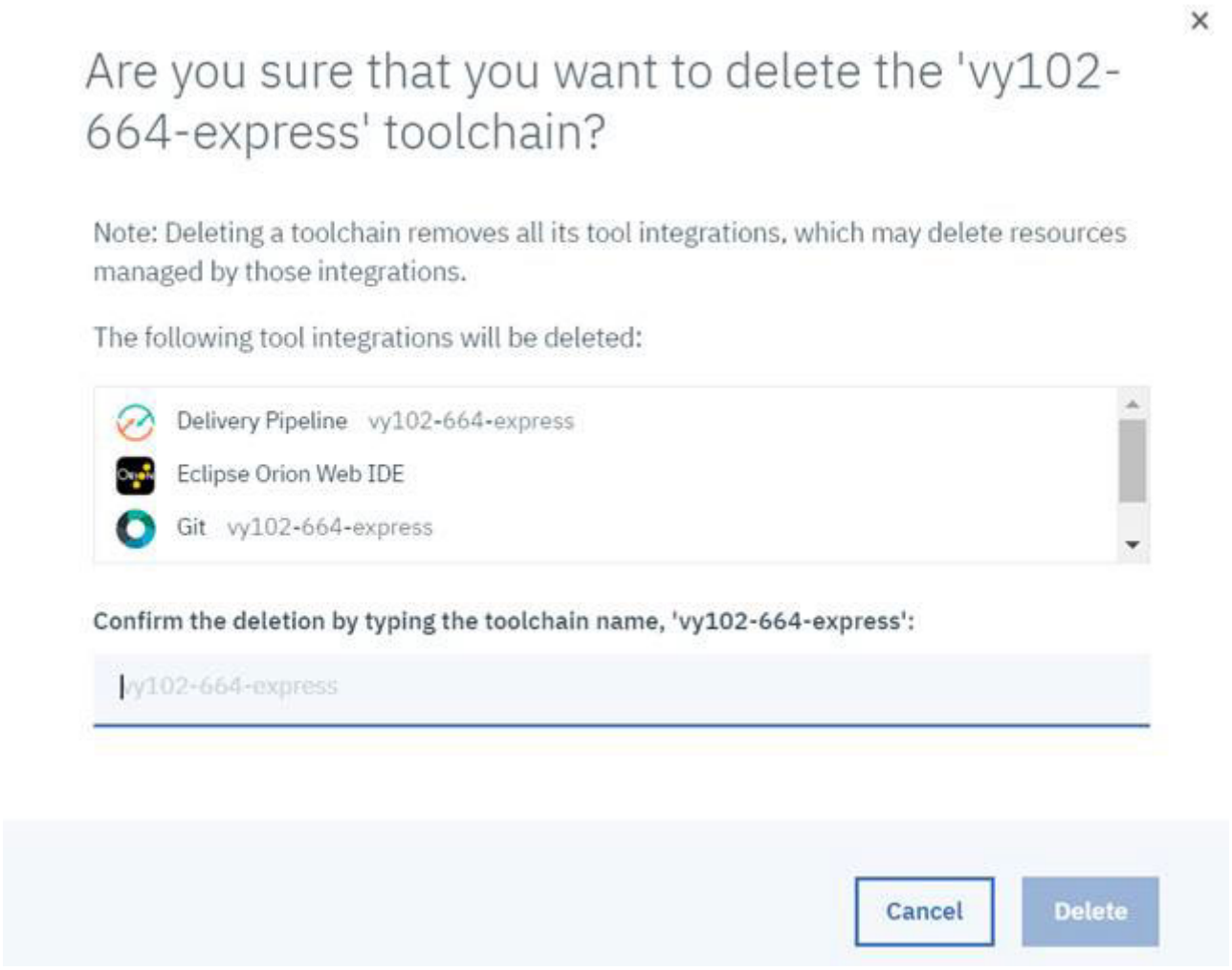


__ 7. To delete the toolchain, go to the toolchain dashboard.

__ 8. Click the three dots next to toolchain name and click **Delete**.



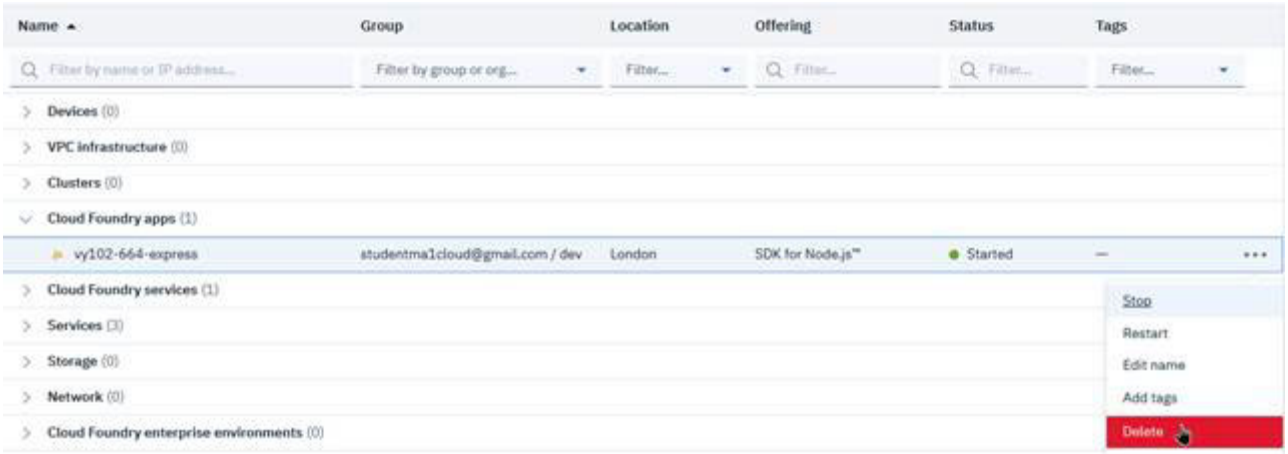
__ 9. At the confirmation window, type the name of your toolchain and click **Delete**.



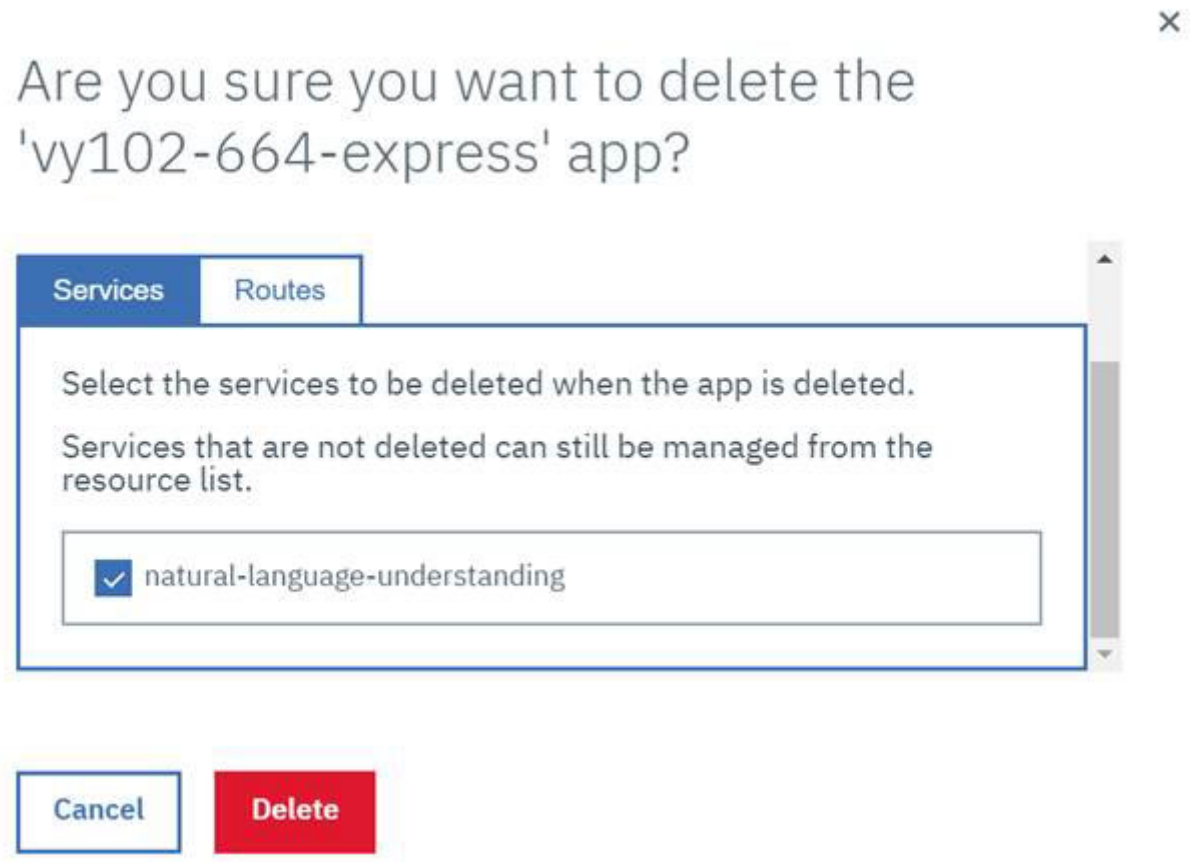
__ 10. To delete the Cloud Foundry application, open the IBM Cloud dashboard.

__ 11. Under Cloud Foundry Apps, click the **Actions** menu (the three dots) for your application.

__ 12. Select **Delete**, as shown in the following figure.



__ 13. Select the boxes for the services and routes that are associated with the app to be deleted, as shown in the following figure, and then click **Delete**.

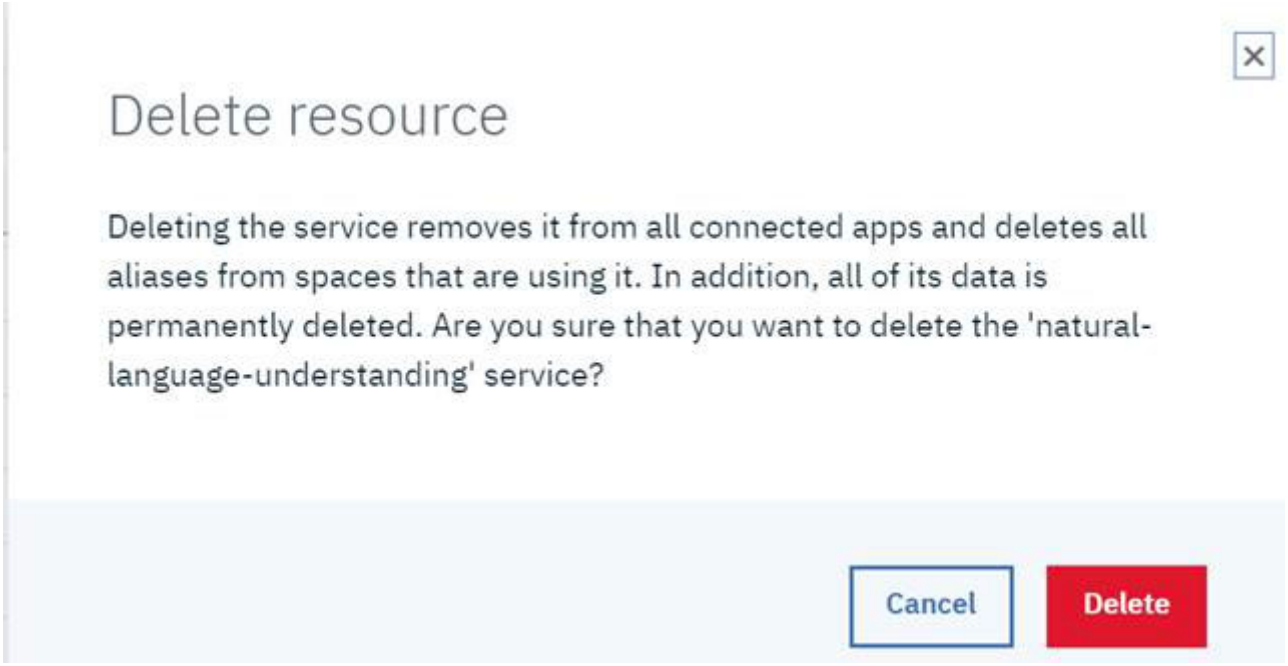


__ 14. Delete the Natural Language Understanding service. At the IBM Cloud dashboard expand **Services**.

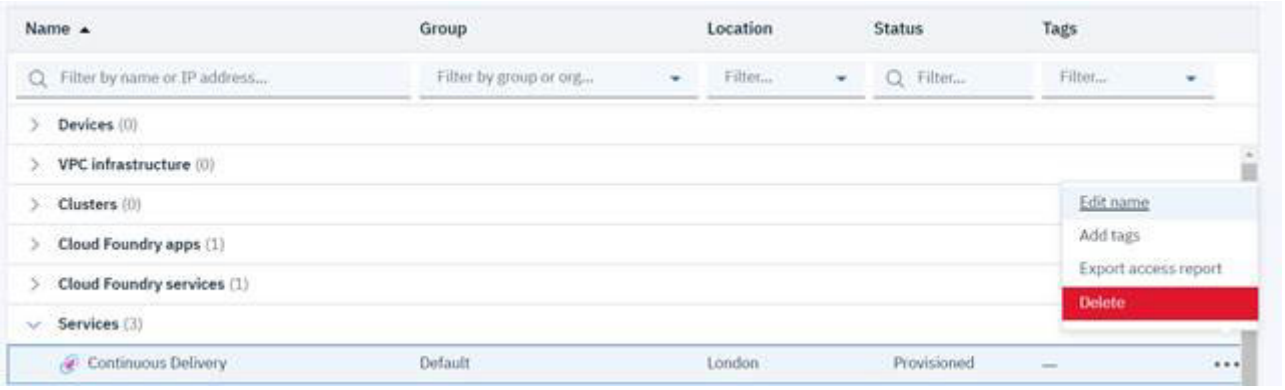
__ 15. Click the **Actions** menu (the three dots) next to your **Natural Language Understanding** service.

__ 16. Click **Delete**

__ 17. At the Delete resource window, click **Delete**



__ 18. Delete the Continuous Delivery service. Expand **Services**, click **Actions** (three dots) to the right of **Continuous Delivery** and click **Delete**.



__ 19. At the Delete resource window, click **Delete**.

Exercise review and wrap-up

During this exercise, you accomplished the following goals:

- Created a Hello World Express application that includes two routes that handle the URIs `GET /` and `POST /author`. You now understand the basics of the Express framework.
- Sent the `index.html` page to the caller of the **`GET`** URI. You now know how to use Express to send an HTML page to the user in response to a route.
- Integrated a Node.js application with Watson Natural Language Understanding service.
- Organized the code into routes, views, and services. By following the steps, you now know some of the best practices for organizing Express applications in Node.js.

Troubleshooting

For troubleshooting any issues, see the full code of this exercise at the following link:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp/Ex3

Exercise 4. Building a rich front-end application by using React and ES8

Estimated time

01:30

Overview

This exercise guides you through building an interactive and rich client-side application by using React. You also explore the async/await feature of ECMAScript 2017, which is commonly known as ES8, and some features of ES8 through a server-side application by using Node.js.

Objectives

By the end of this exercise, you should be able to accomplish these objectives:

- Deploy a React application on IBM Cloud.
- Deploy a Node.js application on IBM Cloud.
- Explore the structure of the React application.
- Explore the ES8 features of the Node.js application.

Prerequisites

Before you start, be sure that you meet these prerequisites:

- An IBM Cloud account.
- An understanding of Cloud DevOps basic concepts.
- An understanding of Git basic concepts.
- Access to a web browser: Google, Chrome, or Mozilla Firefox.

Introduction

In this exercise, you use two JavaScript frameworks, React and Express, to build the Tone Analyzer sample application, which takes input from the user on the client side, sends it to the Tone Analyzer service on the server side to be analyzed, and then sends the results in an HTTP response to be displayed on the client side.

To avoid confusion between the client-side files and the server-side files, keep the client files in a separate application from the server application.

React is a JavaScript-based front-end web application framework that you use to develop single-page applications. The single-page application is a web page that loads one HTML page and dynamically updates it as the user interacts with the app.

Express is a web application framework for Node.js. It is designed for building full-stack web applications and back-end only APIs.

ECMAScript/ES is a language specification that standardizes JavaScript. ECMAScript is used for client-side scripting on the web and for writing server applications and services by using Node.js. In this exercise, you use the async/await feature of ECMAScript 2017, which is commonly known as ES8. ES8 improves the asynchronous programming experience by providing an async function with an await fulfillment.

Exercise instructions

In this exercise, you complete the following tasks:

__ 1. Create a Tone Analyzer service.

__ 2. Create a Node.js application.

__ 3. Integrate the Node.js application and Tone Analyzer service.

__ 4. Clone the React application and Node.js application from GitHub by using the Delivery Pipeline.

__ 5. Integrate the React application and the Node.js application

__ 6. Explore the React application

__ 7. Explore the Node.js application code.

__ 8. Test your application.

__ 9. Clean up the environment.

Part 1: Creating a Tone Analyzer service

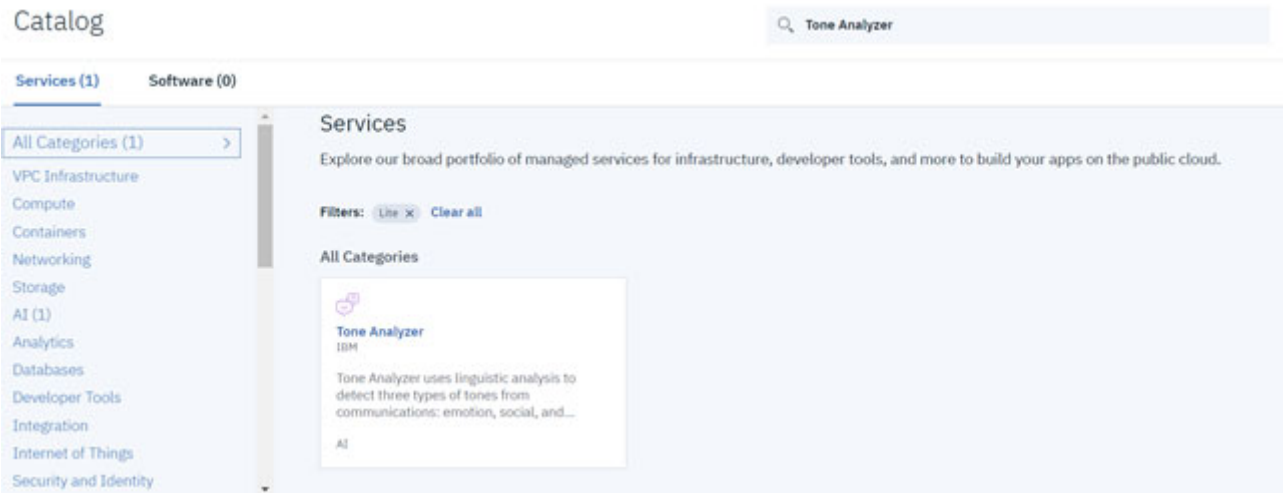
To create the Tone Analyzer service that is needed in this exercise, complete these steps:

__ 1. Log in to IBM Cloud at https://cloud.ibm.com/login.

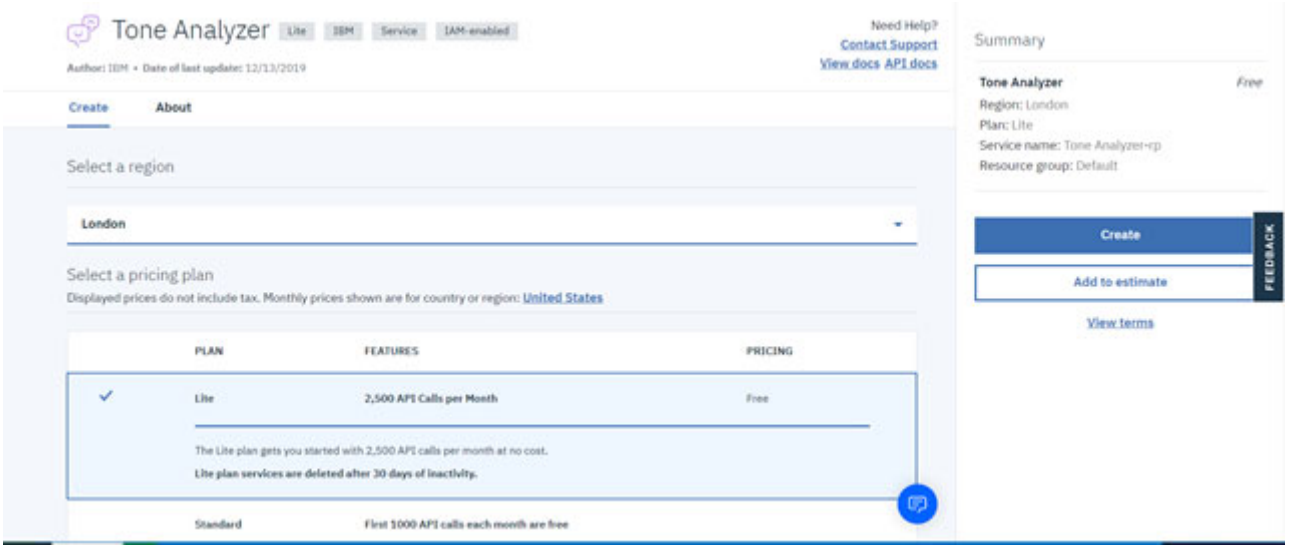
__ 2. Click **Catalog**.

__ 3. In the search field, enter “Tone Analyzer”.

__ 4. Choose the Tone Analyzer service, as shown in the following figure.



__ 5. Create a Tone Analyzer service by keeping the default values, as shown in the following figure.



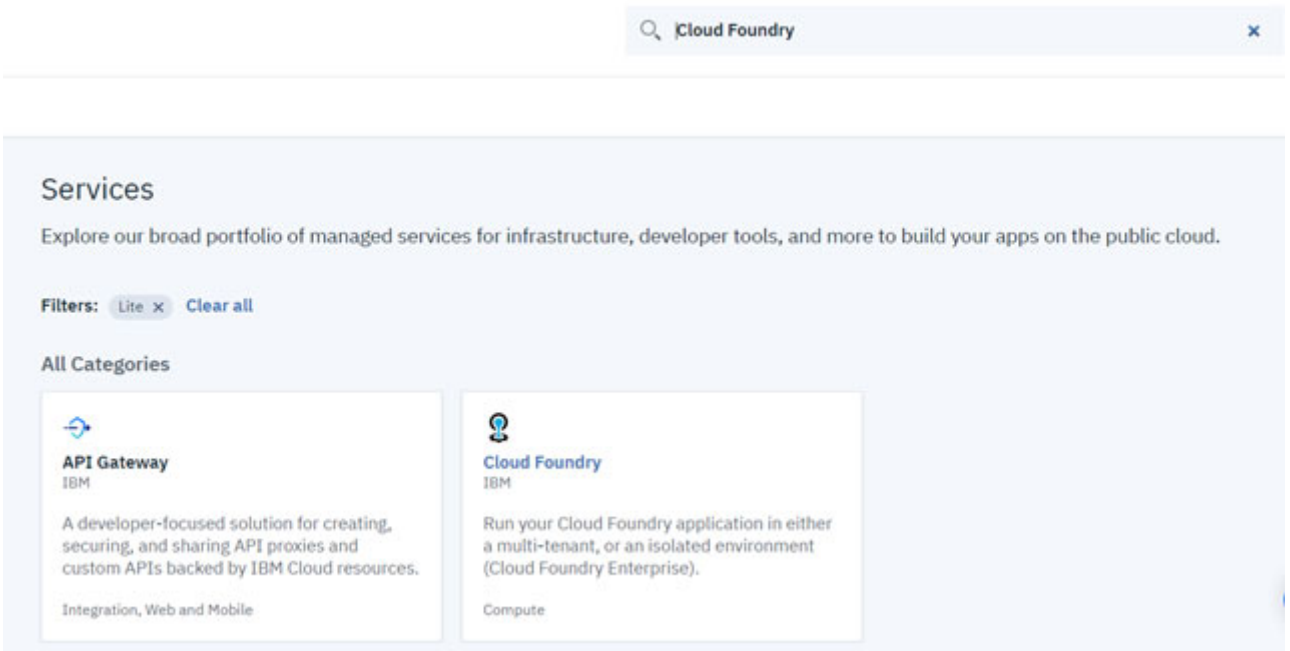
__ 6. Click **Create**.

Part 2: Creating a Node.js application

In this part, you create a Node.js application by completing these steps:

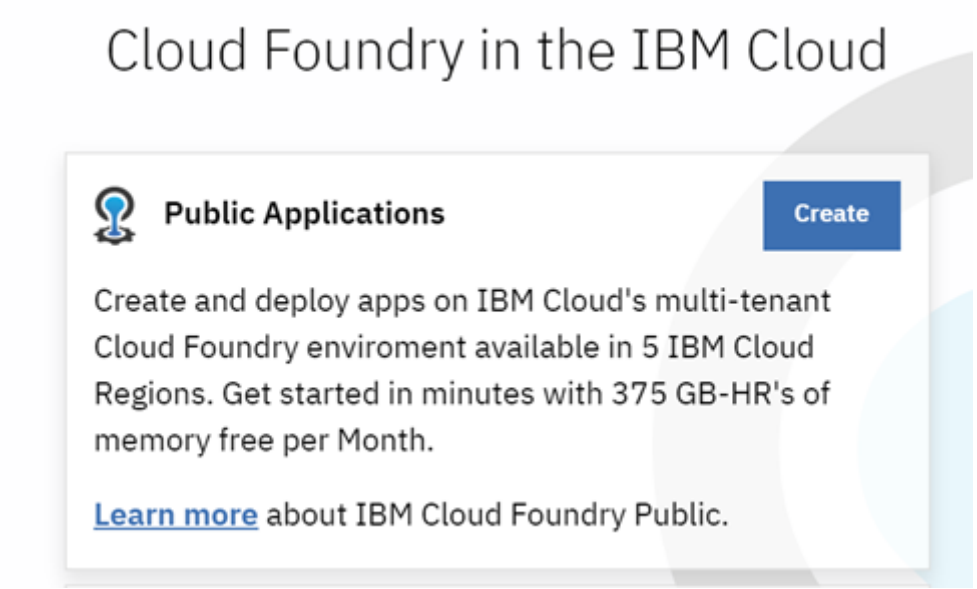
__ 1. Click **Catalog**.

__ 2. In the search field, enter **Cloud Foundry**, as shown in the following figure.

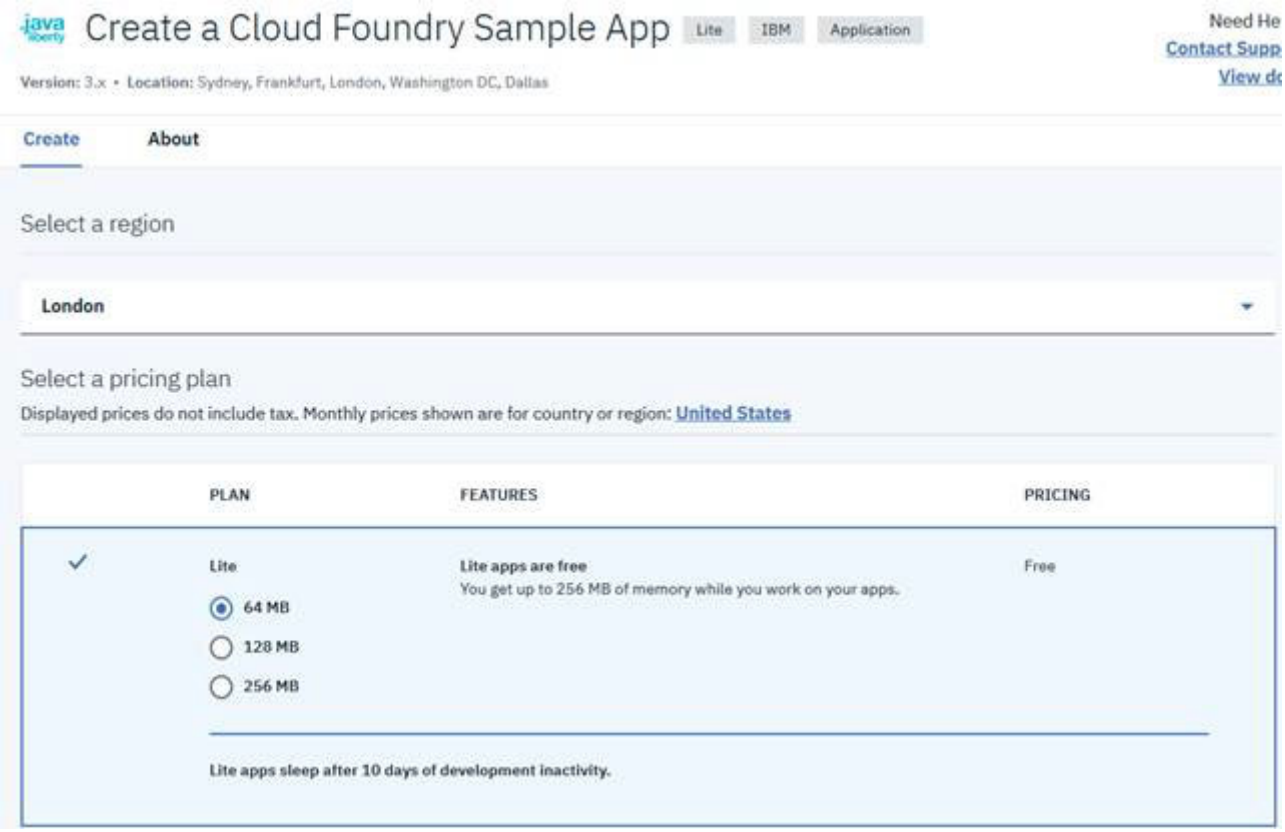


__ 3. Choose **Cloud Foundry** service.

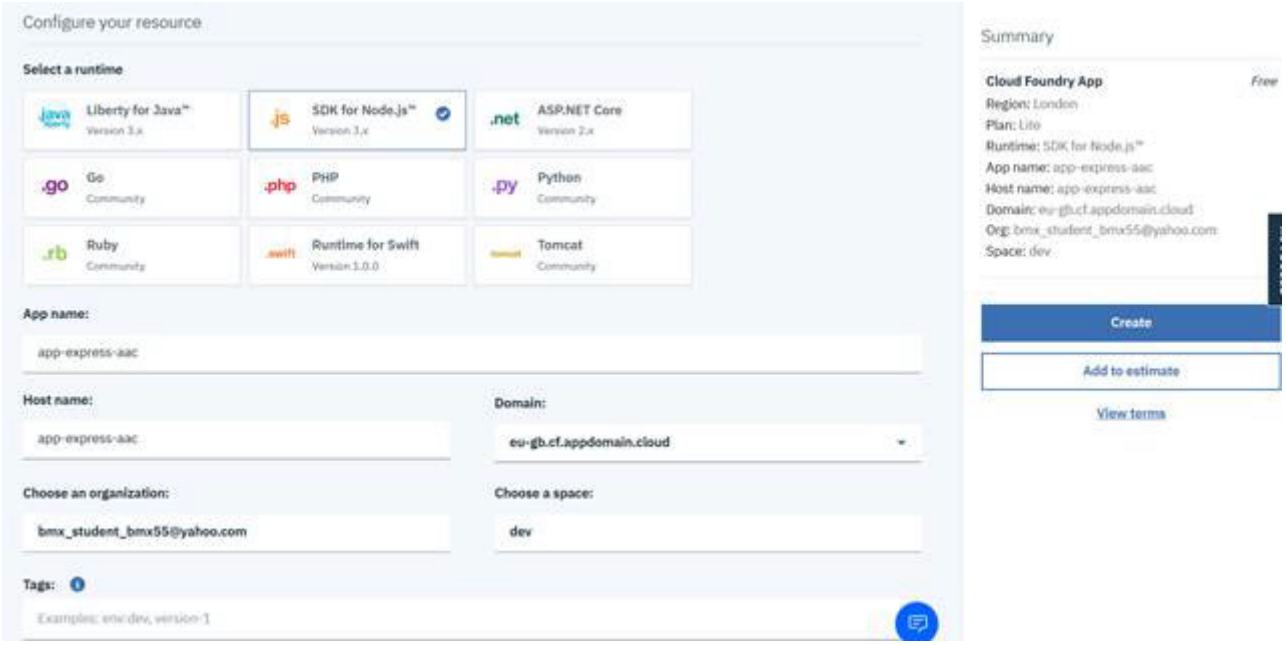
__ 4. From the Cloud Foundry overview page, click **Create** in **Public Applications** as shown in the following figure.



__ 5. Accept the default for region and pricing plan, as shown in the following figure.

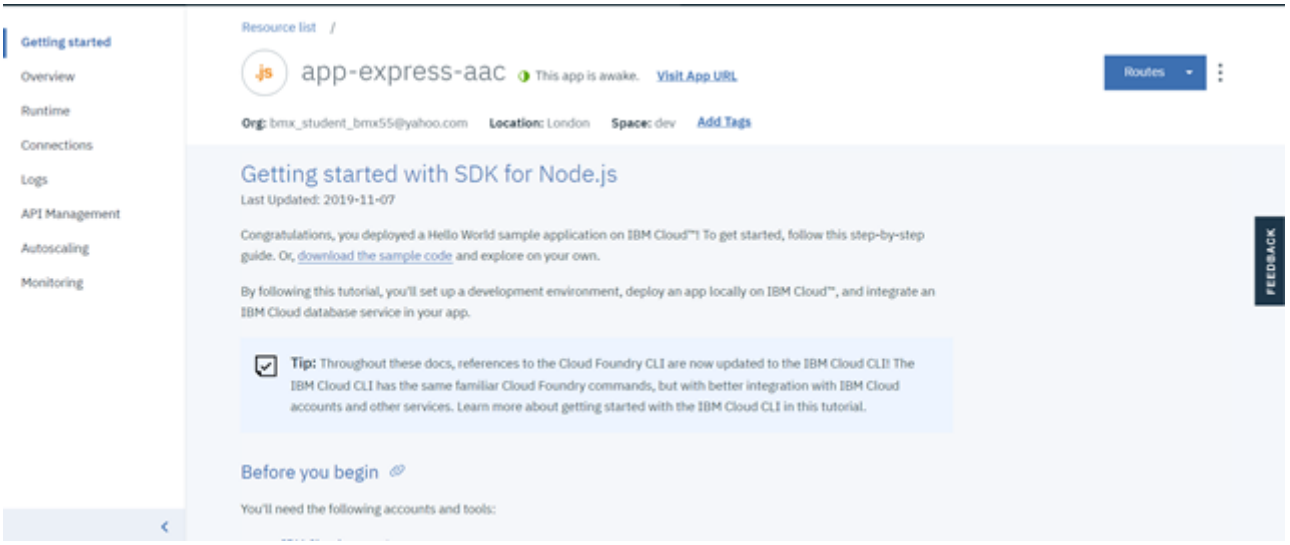


__ 6. Select **SDK for Node.js** as runtime and enter the app name as **app-express-*XXX***, where XXX are three unique characters. Accept the rest of the default values, as shown in the following figure.

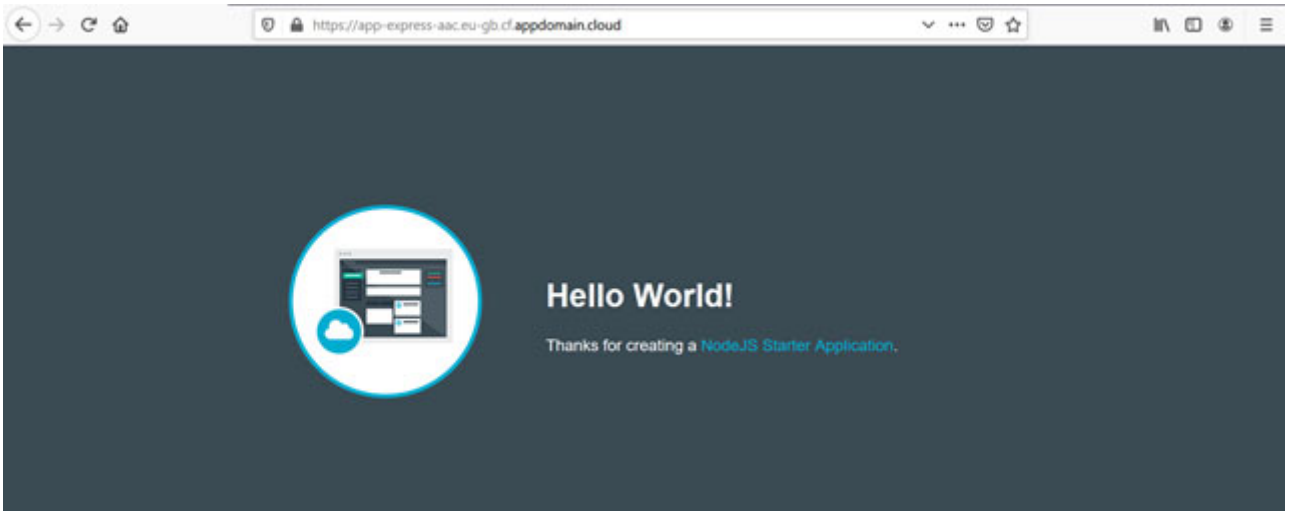


__ 7. Click **Create**.

__ 8. Wait until the status changes to **This app is awake** as shown in the following figure, then click **Visit App URL**.



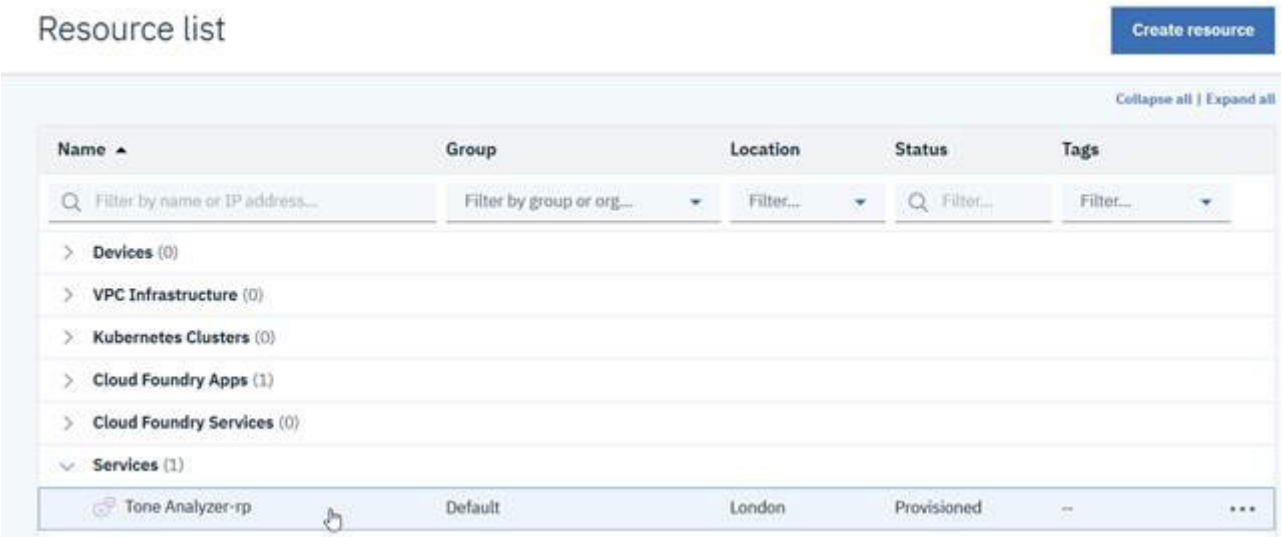
__ 9. A new tab opens as shown in the following figure. Copy the URL and paste it in a text editor for later use. This is the URL to the Node.js server application.



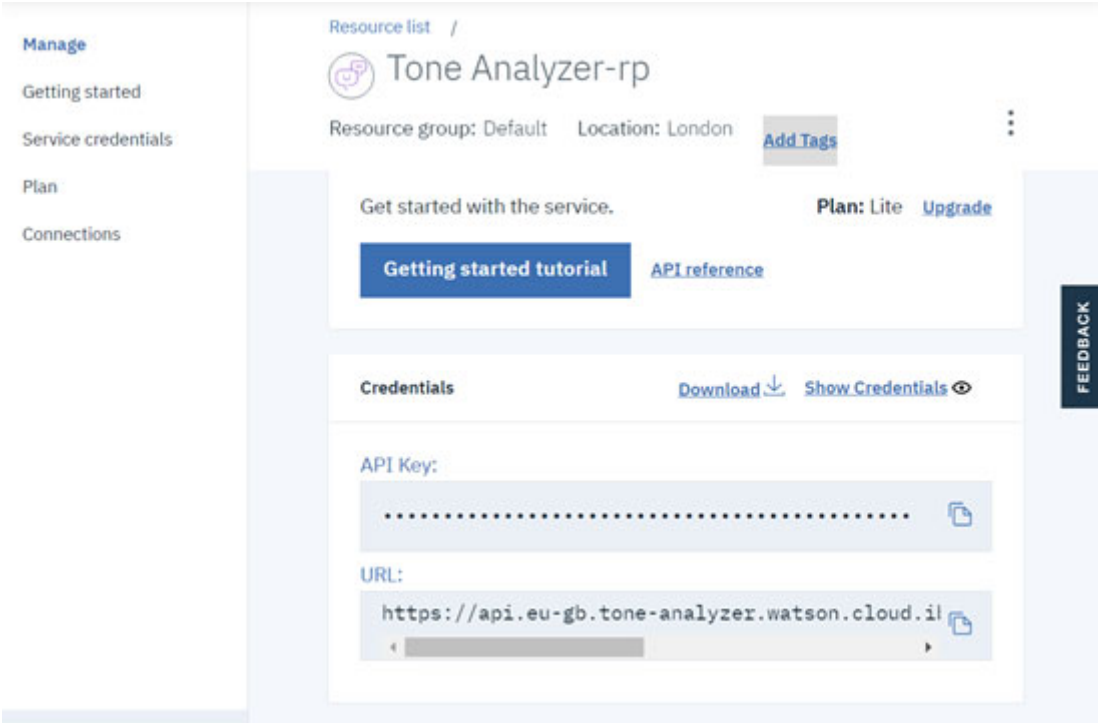
Part 3: Integrating the Node.js application and Tone Analyzer service

In this part, you bind the Tone Analyzer service to the Node.js application by completing the following steps:

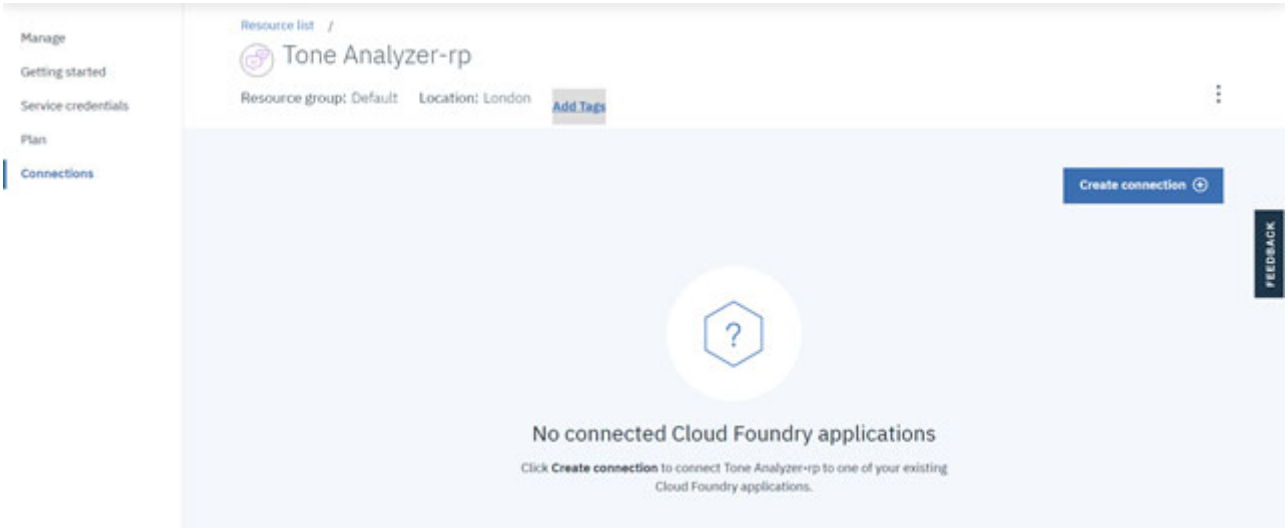
__ 1. In the IBM Cloud Dashboard, expand Services and click the Tone Analyzer service that you created in *Part 1. Creating a Tone Analyzer service*.



__ 2. The window that is shown in the following figure is displayed.

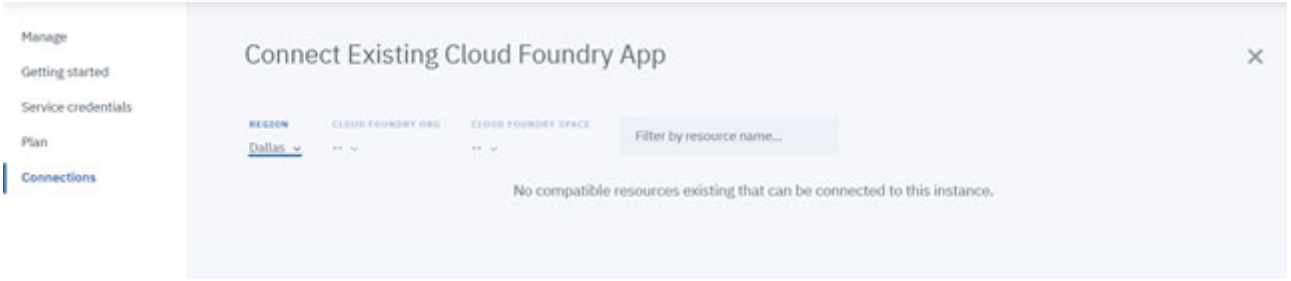


__ 3. Click **Connections** in the left bar.



__ 4. Click **Create Connection**.

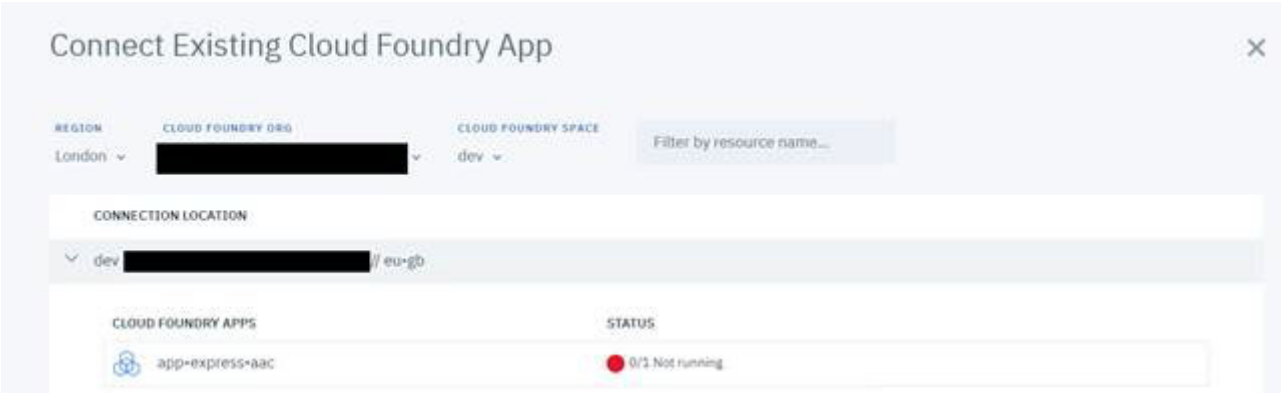
__ 5. Make sure that your region (the region where you created the Node.js app) is selected. If the correct region is not select by default, select the correct region now. If the wrong region is selected, you cannot view your Cloud Foundry applications, as shown in the following figure.



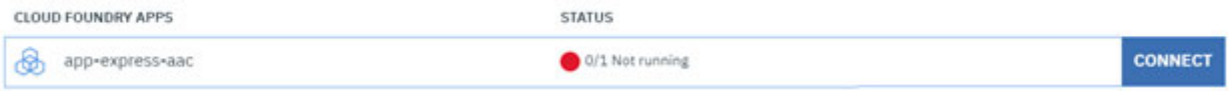
Note

In this example, the user's region is London, so no Cloud Foundry apps are displayed if Dallas is selected.

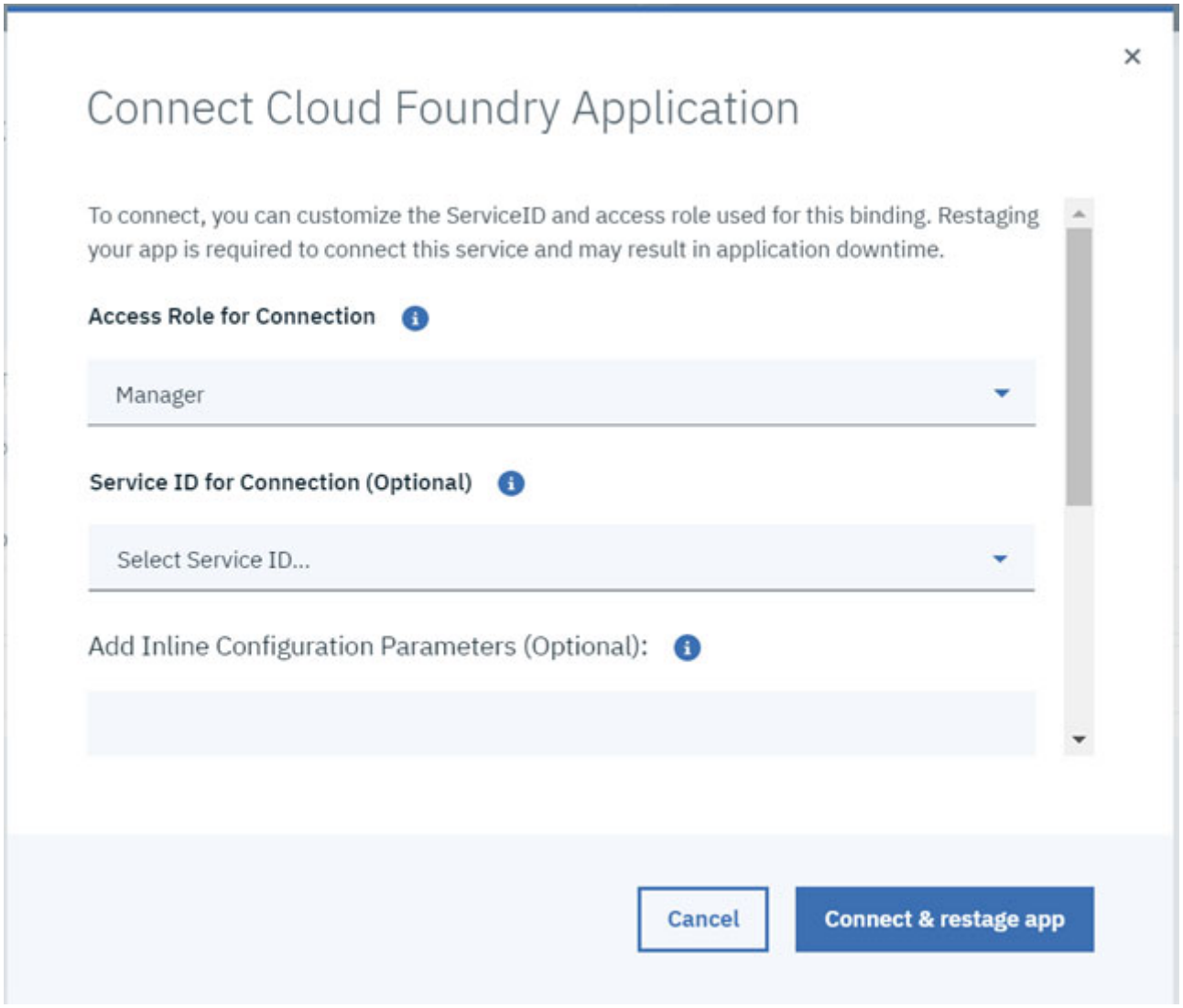
__ 6. Your Cloud Foundry apps are displayed when the correct region, Cloud Foundry org, and space are selected, as shown in the following figure.



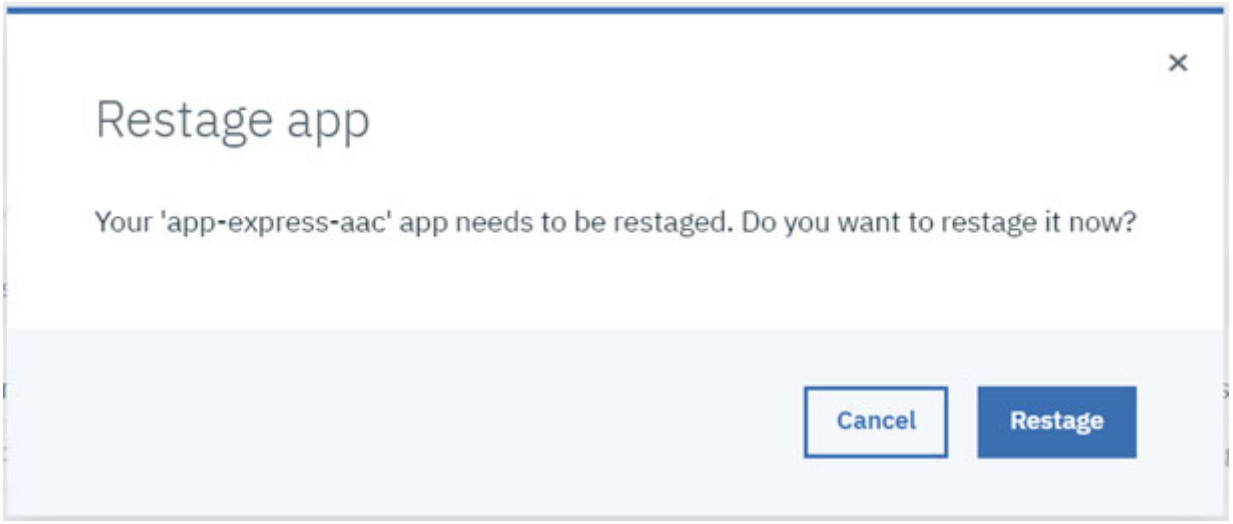
__ 7. Click your Cloud Foundry app to select it. The CONNECT button is activated, as shown in the following figure. Click **CONNECT**.



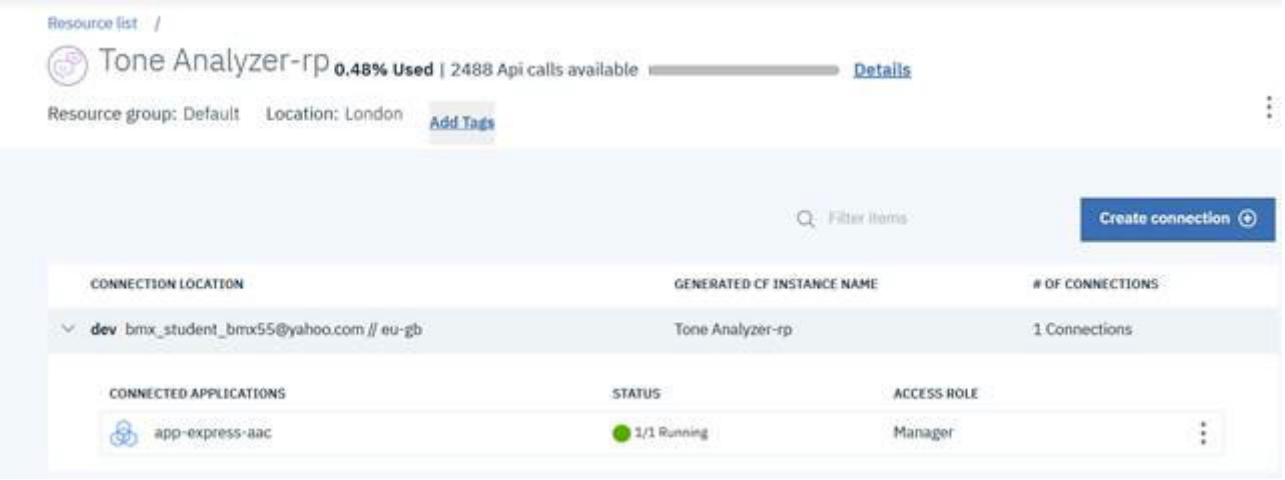
__ 8. Click **Connect & restage app**, as shown in the following figure.



__ 9. A window opens. Click **Restage**, as shown in the following figure.



__ 10. The Node.js app and the Tone Analyzer service are now connected, as shown in the following figure.



Part 4: Cloning the React and Node.js applications from GitHub by using the Delivery Pipeline

In this part, you clone the React application code and the server application code from GitHub and deploy it to IBM Cloud.

To deploy quickly the application, complete the following sections.

Cloning the React application

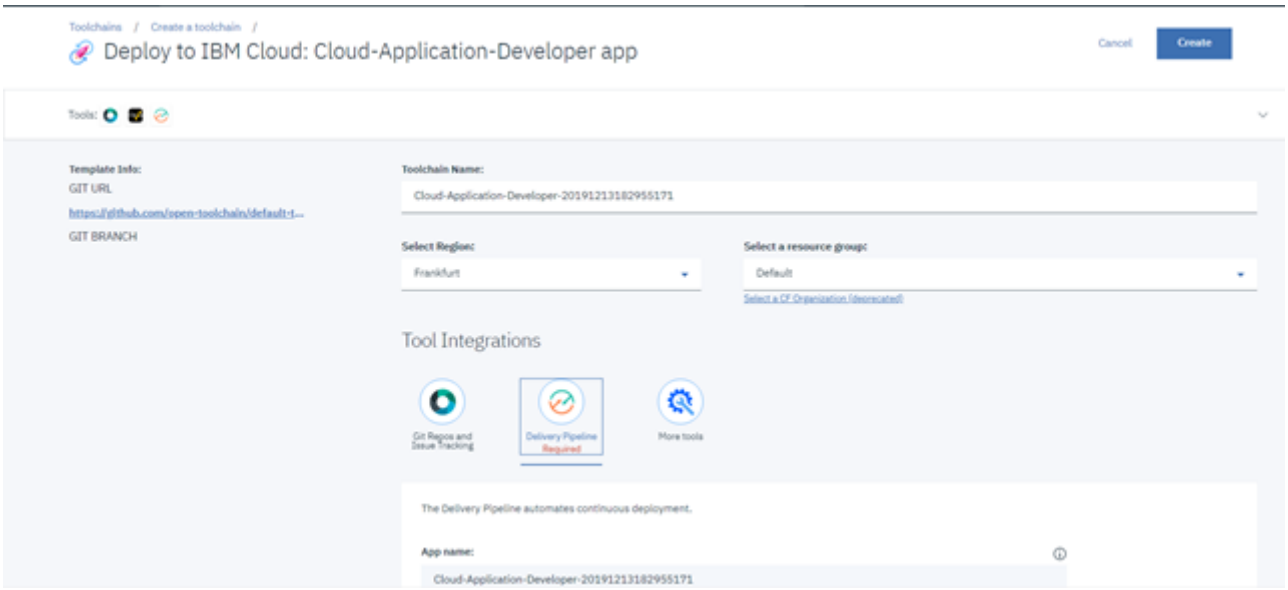
__ 1. Open the following link:

https://cloud.ibm.com/devops/setup/deploy?repository=https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer

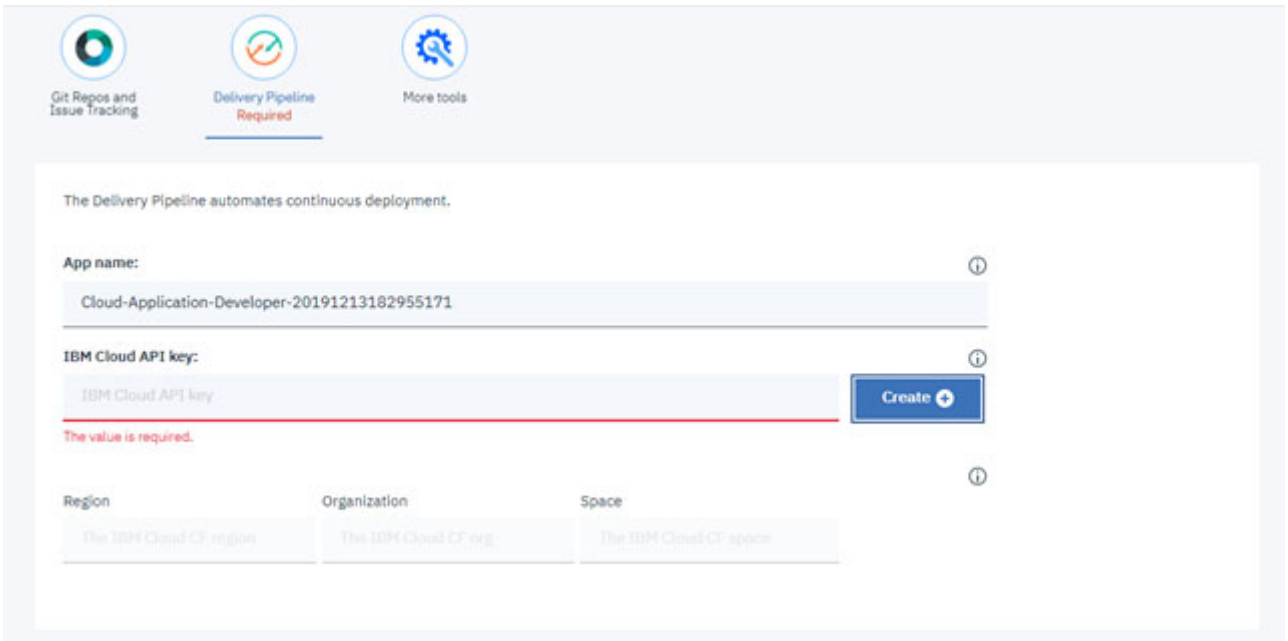
__ 2. On the page that opens, write down the selected region for the toolchain (**Frankfurt** in this example) and select **Delivery Pipeline** as shown in the following figure.

Information

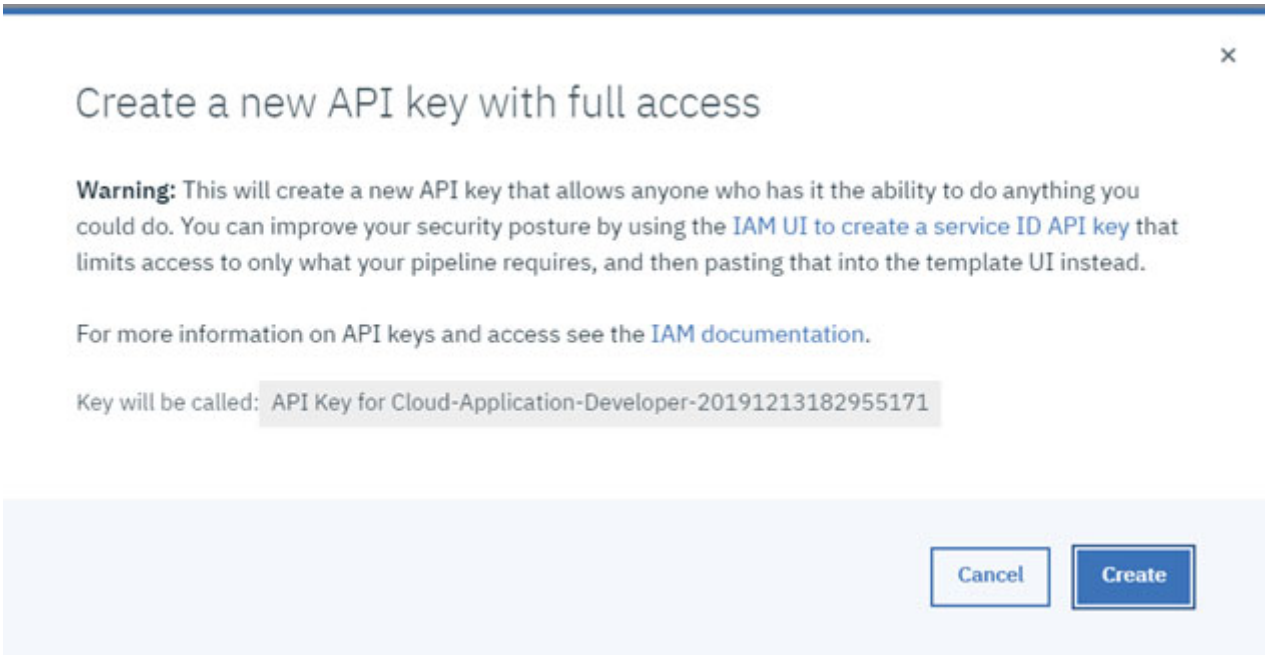
This is the region where the toolchain for the *React* application will be created.



__ 3. Scroll down to IBM Cloud API key and click **Create+**.

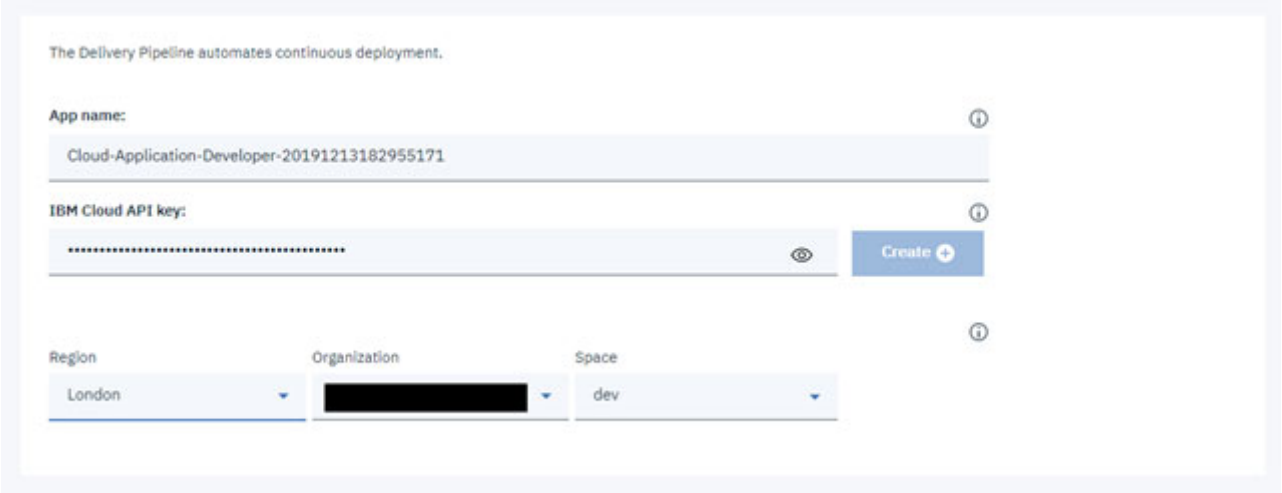


__ 4. At the Create API key window, click **Create** as shown in the following figure.



The IBM Cloud API key is generated.

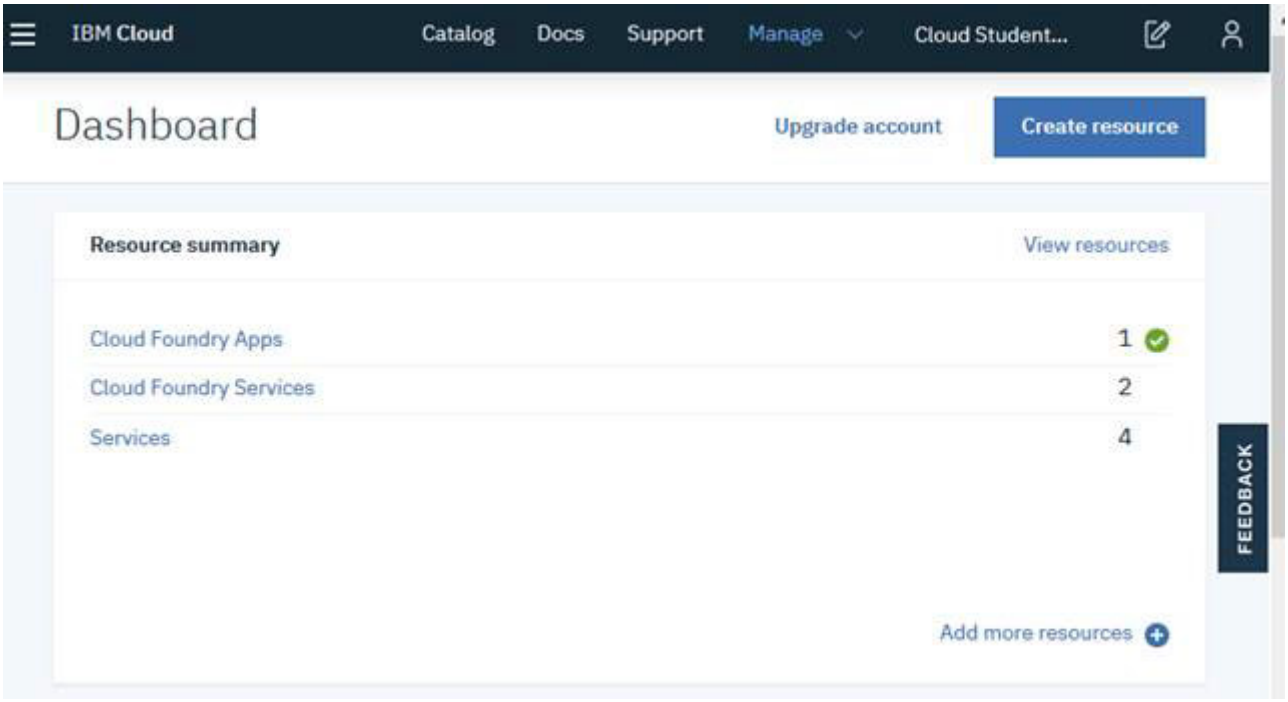
__ 5. Verify that the region where your Cloud Foundry app was created in *Part 2. Creating a Node.js application* is selected or select it now (London in this example), as shown in the following figure. The Organization and Space are automatically populated.



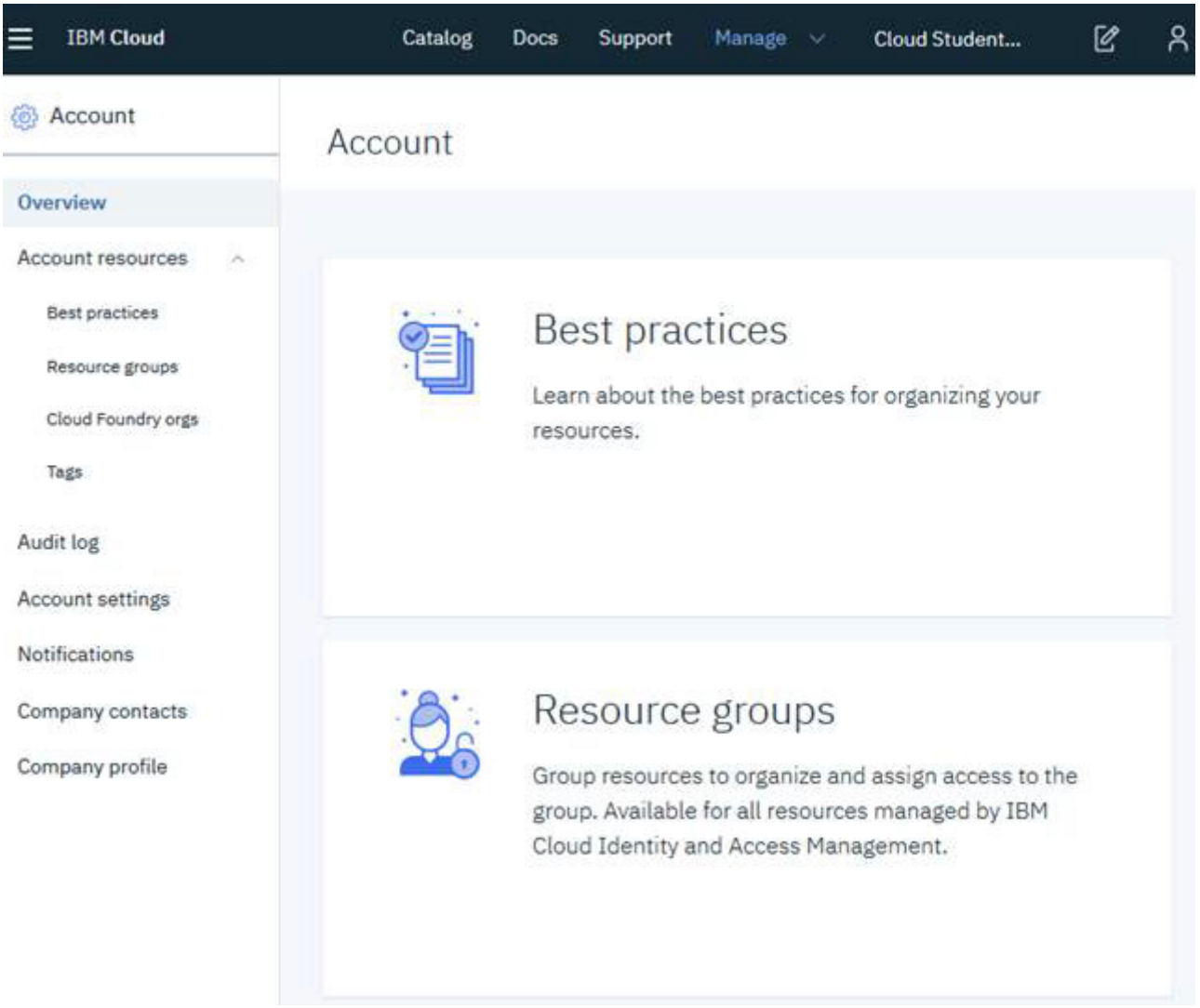
Note

To find your Region, check where the organization is by completing the following steps:

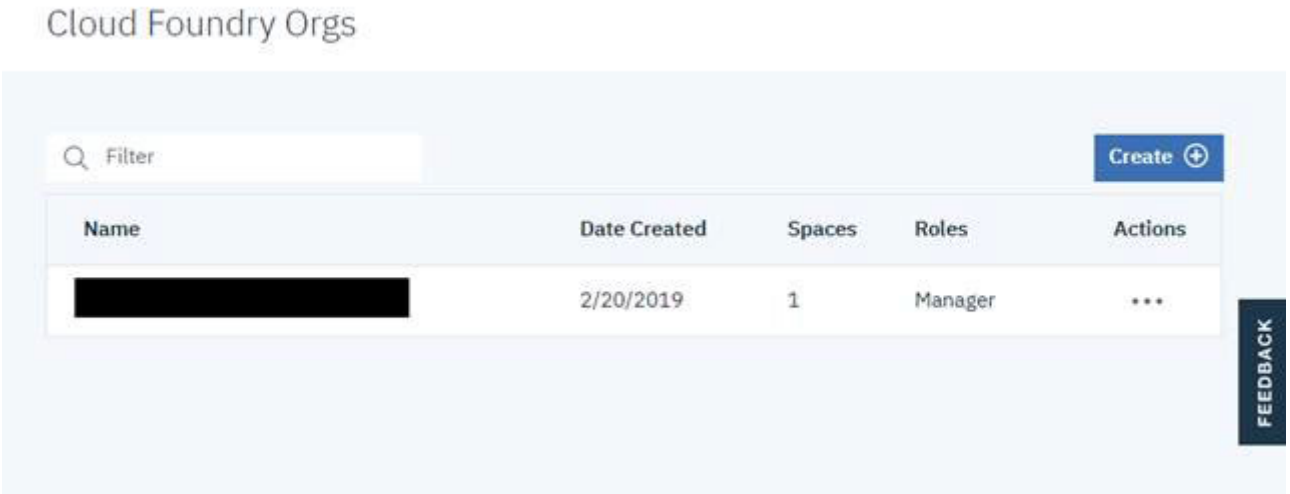
- In a new tab, go to the IBM Cloud dashboard and click **Manage**.



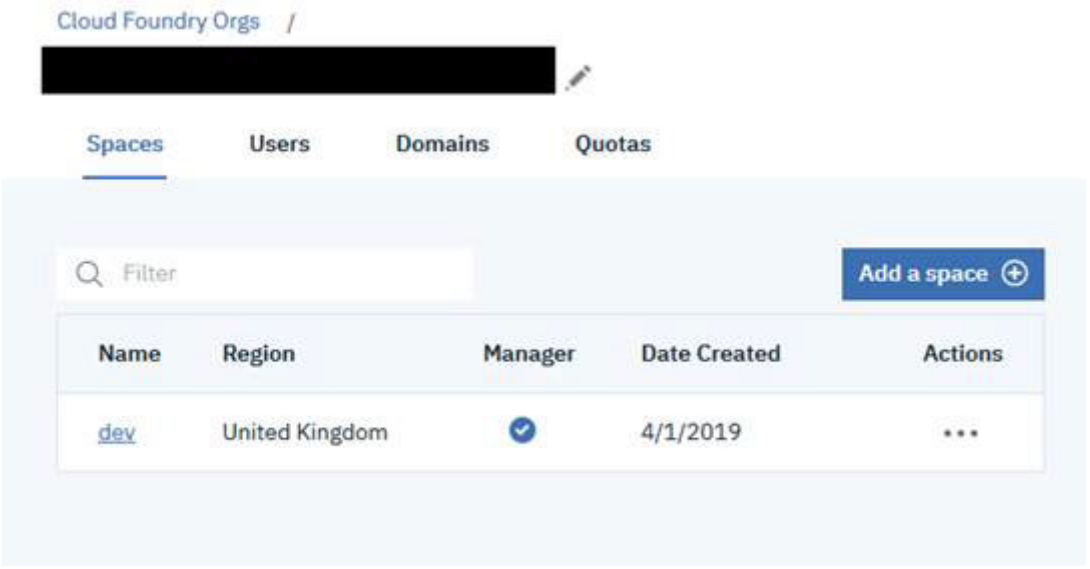
- Click **Account**. The window in the following figure opens.



- In the left pane, click **Cloud Foundry orgs**. The window that is shown in the following figure opens.



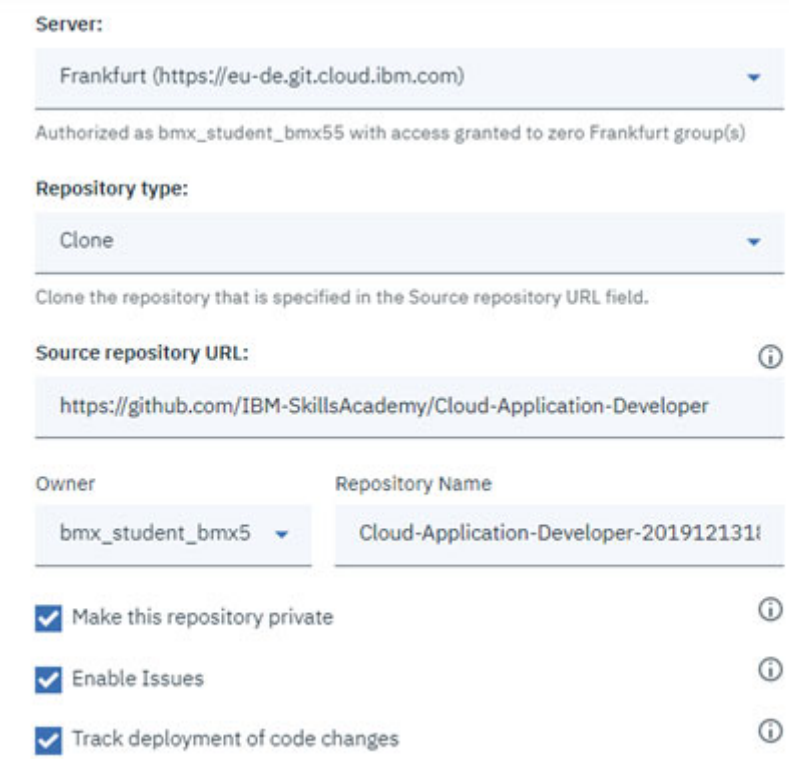
- Click the name of your org. You find your region and your space, as shown in the following figure.



- If the Region is United Kingdom, then choose **London (Production)**.
- If the Region is US South, then choose **Dallas (Production)**.
- If the Region is Australia, then choose **Sydney (Production)**.

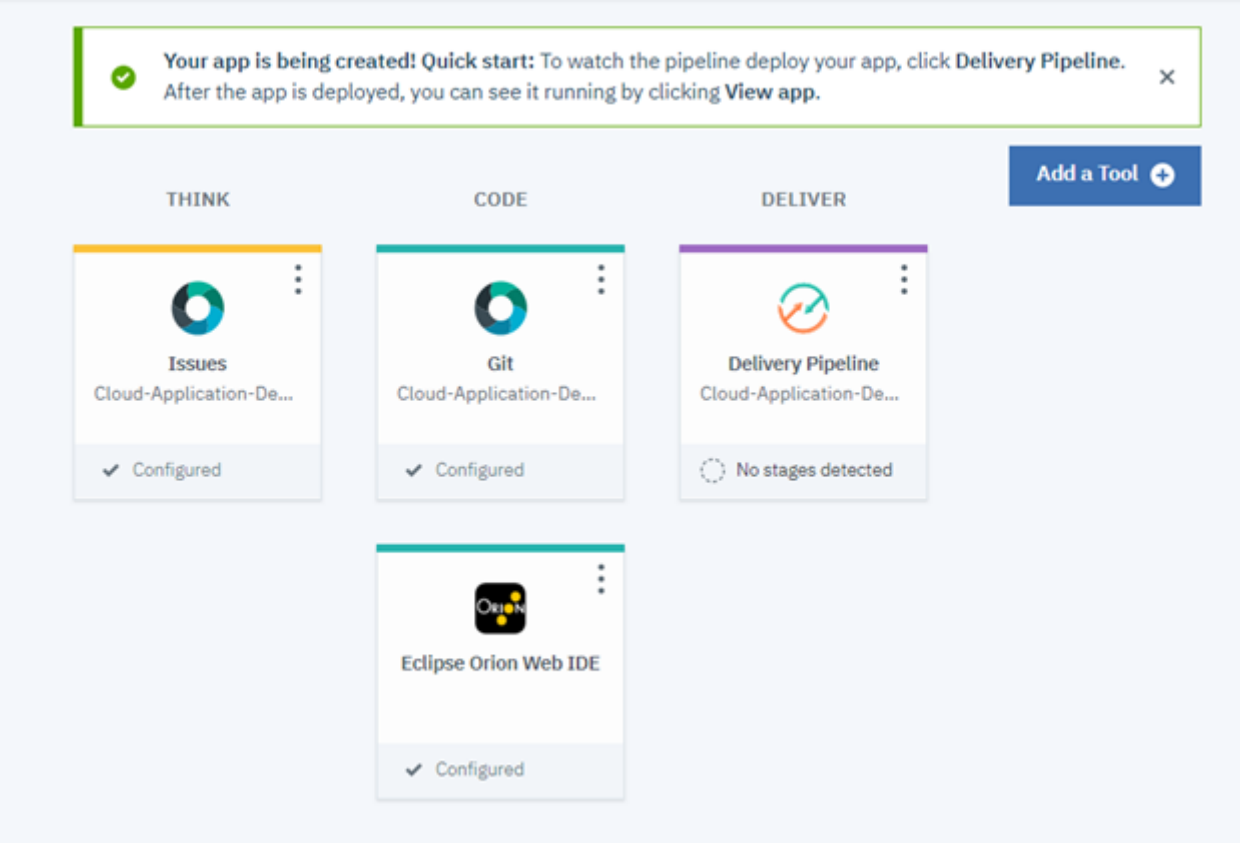
__ 6. Click **Git Repos and Issue Tracking** and enter or verify the following information, as shown in the following figure:

- Source repository URL: https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer
- Repository type: **Clone**



__ 7. Click **Create**.

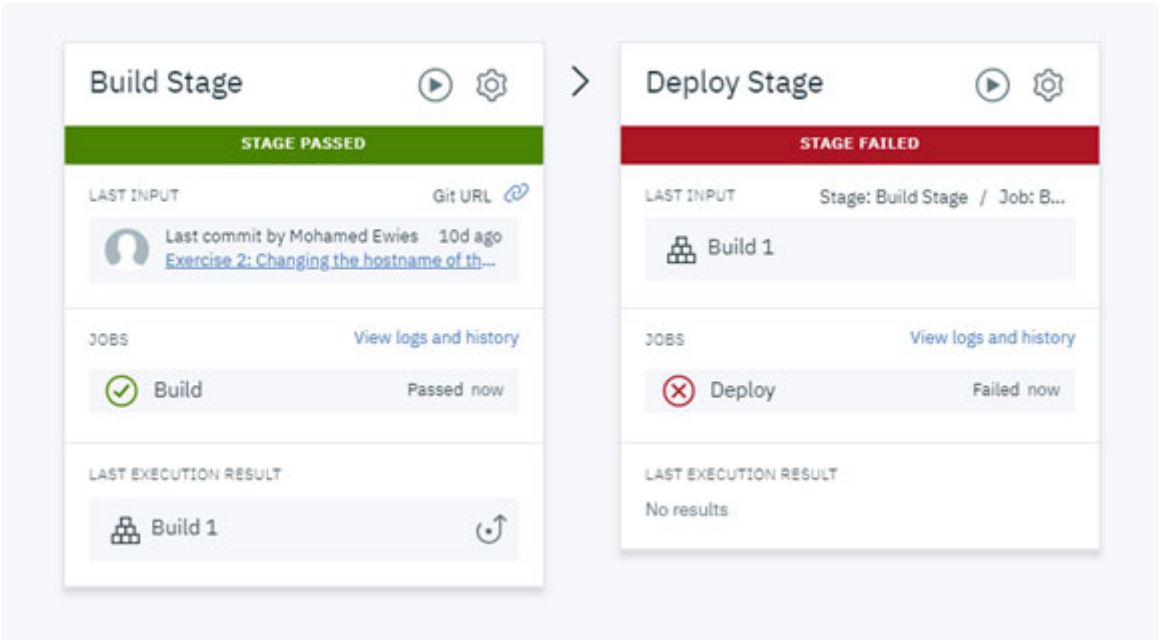
__ 8. A window opens (see the following figure). Click **Delivery Pipeline**.



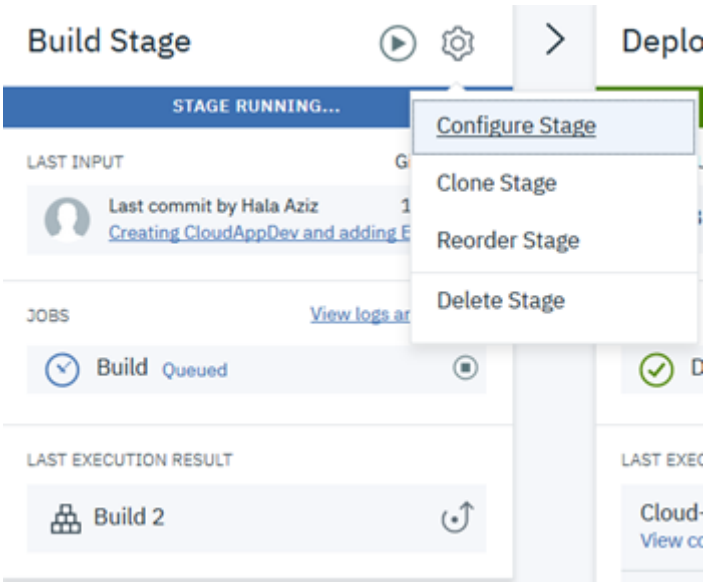
__ 9. You are redirected to the Stages page to build and deploy your application, as shown in the following figure. Click the **Stage Configuration** icon (the gear icon) in the Build Stage pane.

Note

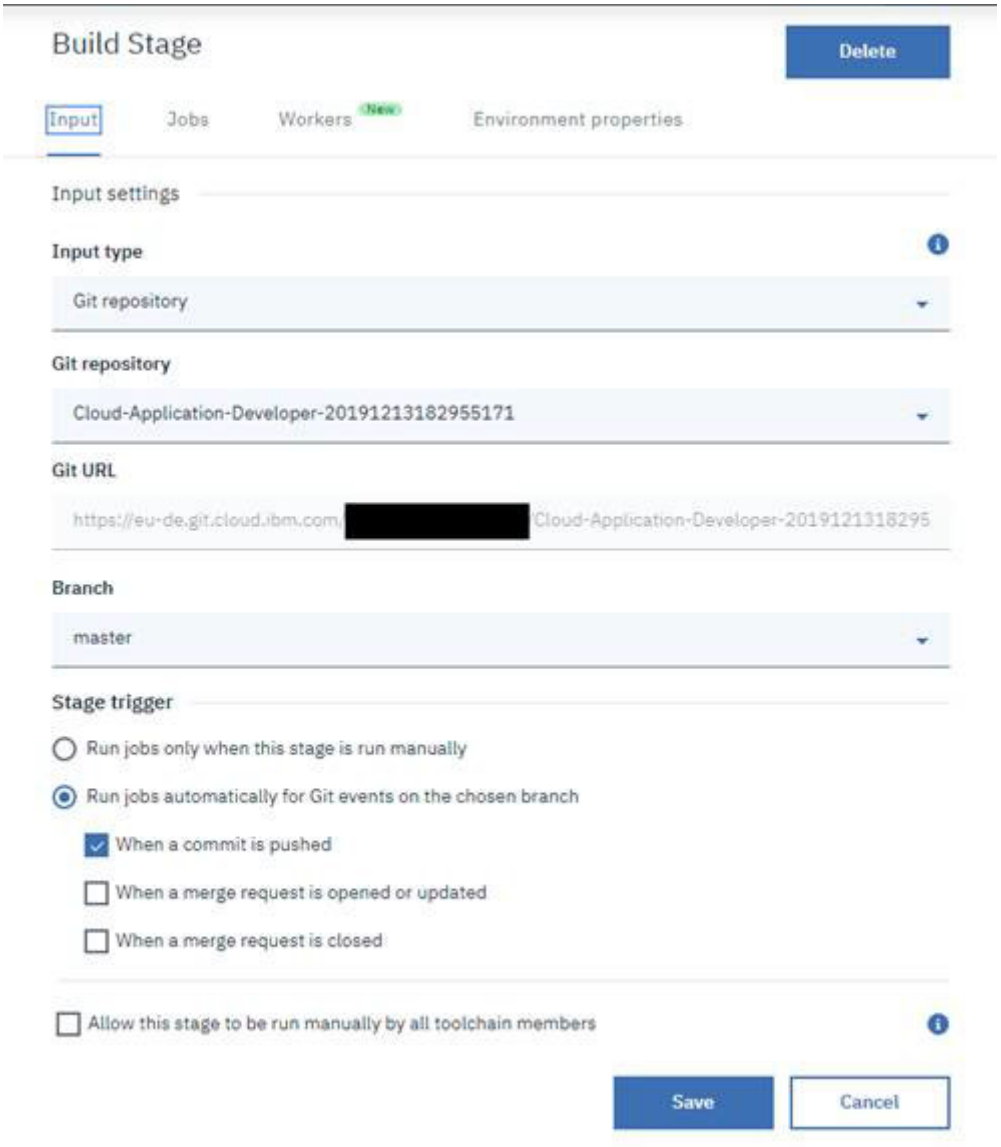
Do not worry about STAGE FAILED in the Deploy Stage pane because you are going to configure it in the next steps.



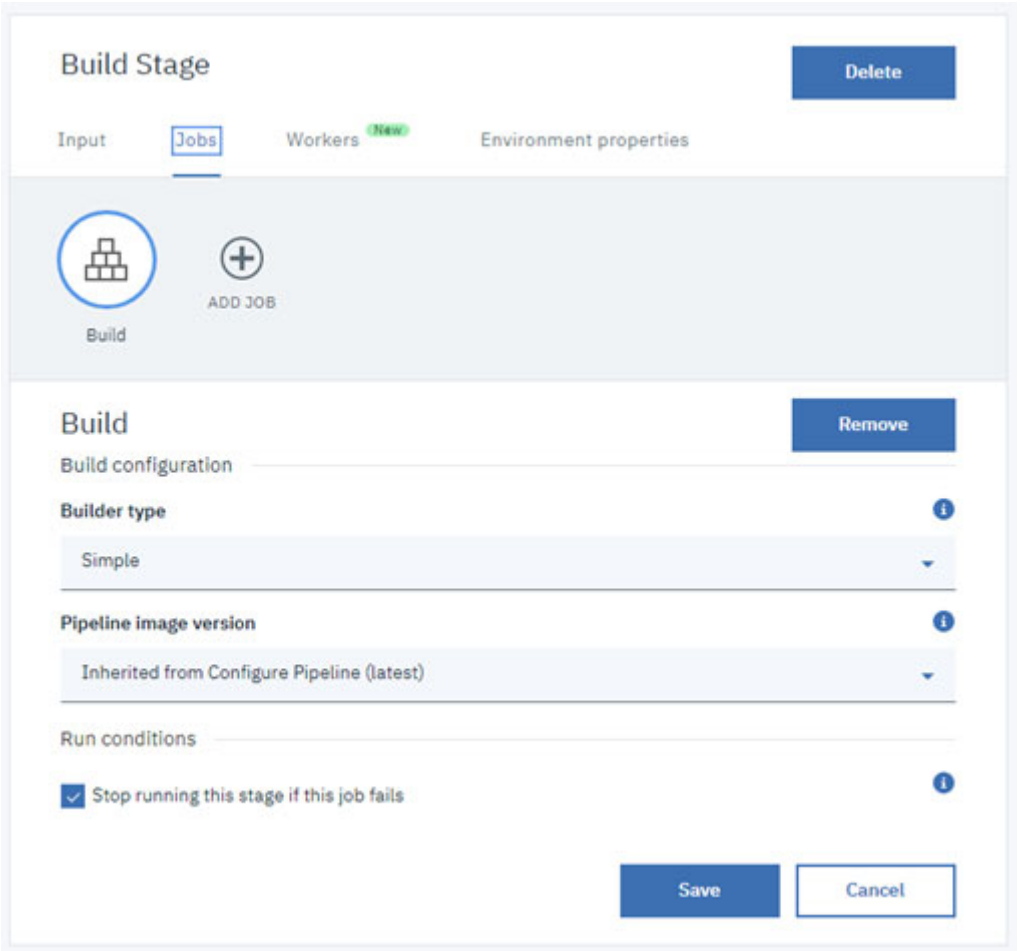
__ 10. Click **Configure Stage**, as shown in the following figure.



The window that is shown in the following figure opens. Make sure that the **Input** tab is selected.



__ 11. Select the **Jobs** tab. The window that is shown in the following figure opens.



__ 12. Select **Custom Docker Image** for Builder type. This selection opens another configuration window. Enter the following information:

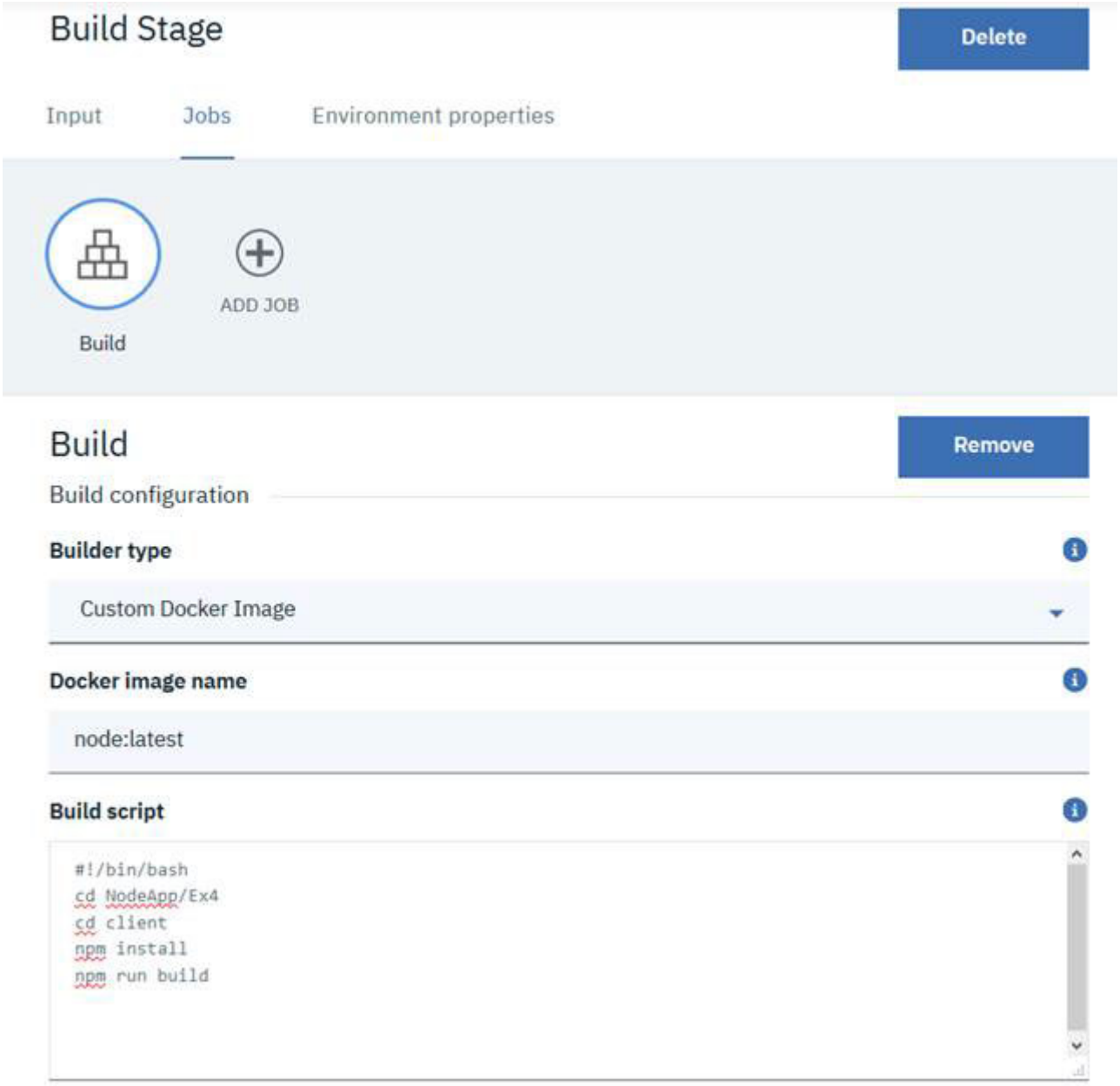
- Docker image name: **node:latest**
- Build Script:

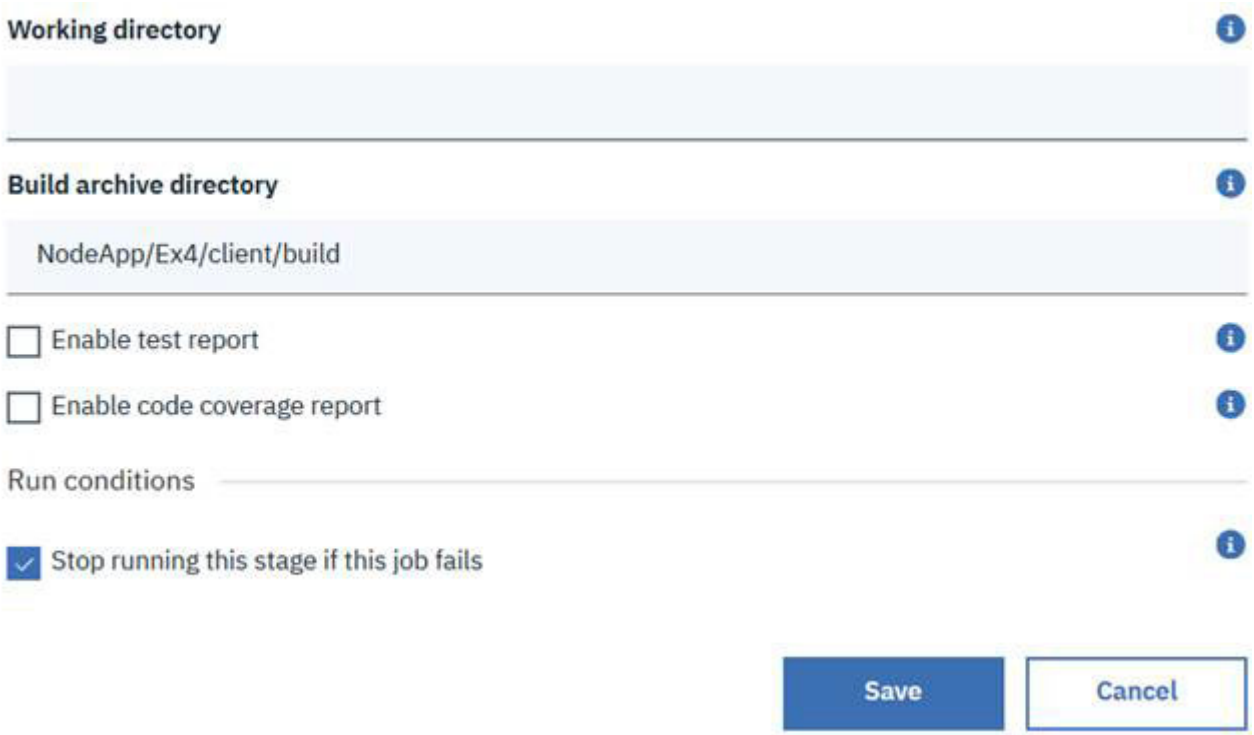
```
#!/bin/bash
cd NodeApp/Ex4
cd client
npm install
npm run build
```

- Build archive directory:

```
NodeApp/Ex4/client/build
```

Your build configuration should look like the following figures.





Information

You need a custom Docker image to build this application and the latest Node version. It is a best practice to have a newer version of Node.js to support ES8 if you are going to use it inside the code.

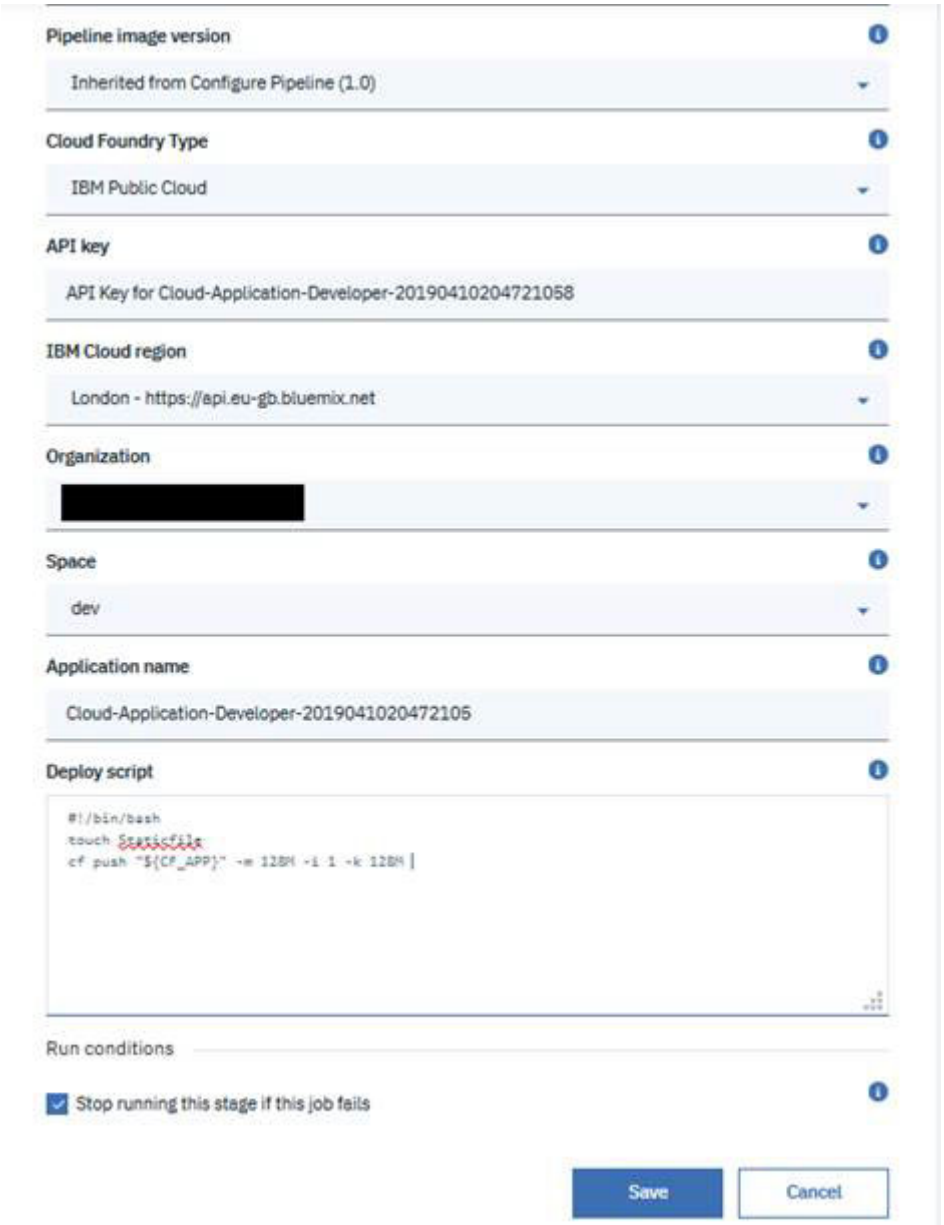
You build the application by running **`npm run build`**, which builds the app for production and places it in the build folder. It correctly bundles React in production mode and optimizes the build for the best performance.

__ 13. Click **Save**.

__ 14. Click the **Stage Configuration** icon (the gear icon) in the Deploy Stage pane, and then choose **Configure Stage**.

__ 15. In the Deploy Stage, Jobs tab, enter the following Deploy script, as shown in the following figure:

```
#!/bin/bash
touch Staticfile
cf push "${CF_APP}" -m 128M -i 1 -k 128M
```



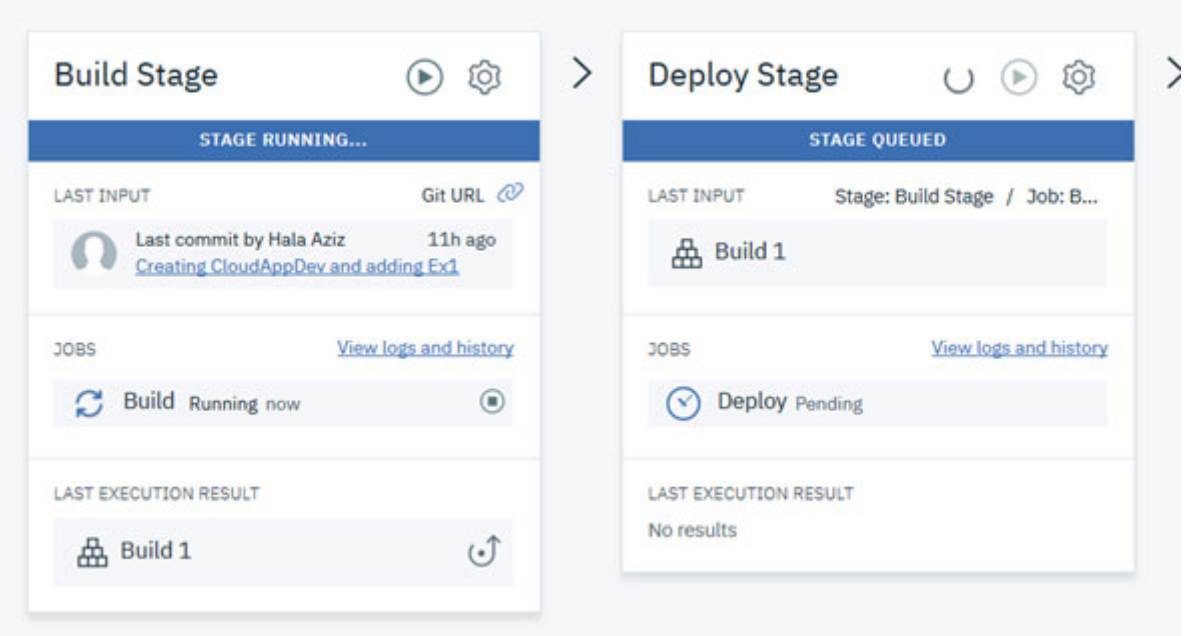
Note

If you create a file that is named `Staticfile` in the root directory of your app, Cloud Foundry automatically uses the `Staticfile` buildpack when you push your app.

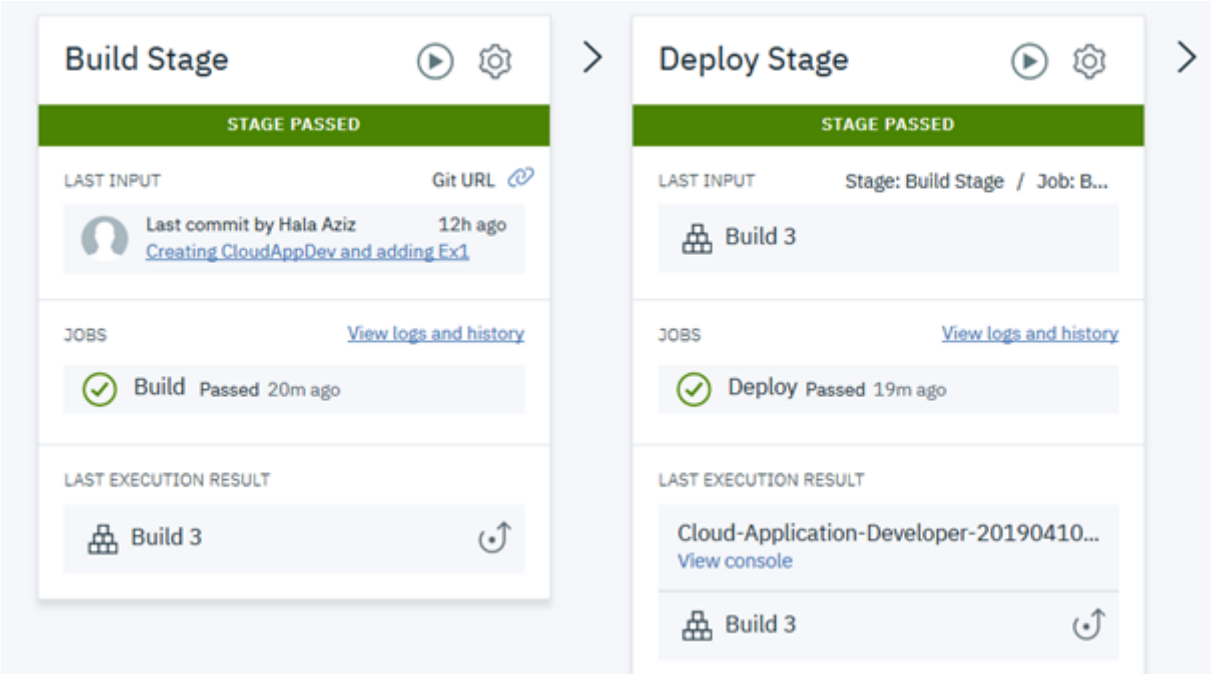
The `Staticfile` file can be an empty file, or it can contain configuration settings for your app. In this exercise, you leave it empty by using the command **`touch Staticfile.`**

__ 16. Click **Save**.

__ 17. Click the **Run Stage** icon (the arrow in the circle icon) in both the Build Stage and Deploy Stage panes, as shown in the following figure.



After both stages run, you see an output like the following figure.



The React application is now deployed on IBM Cloud.

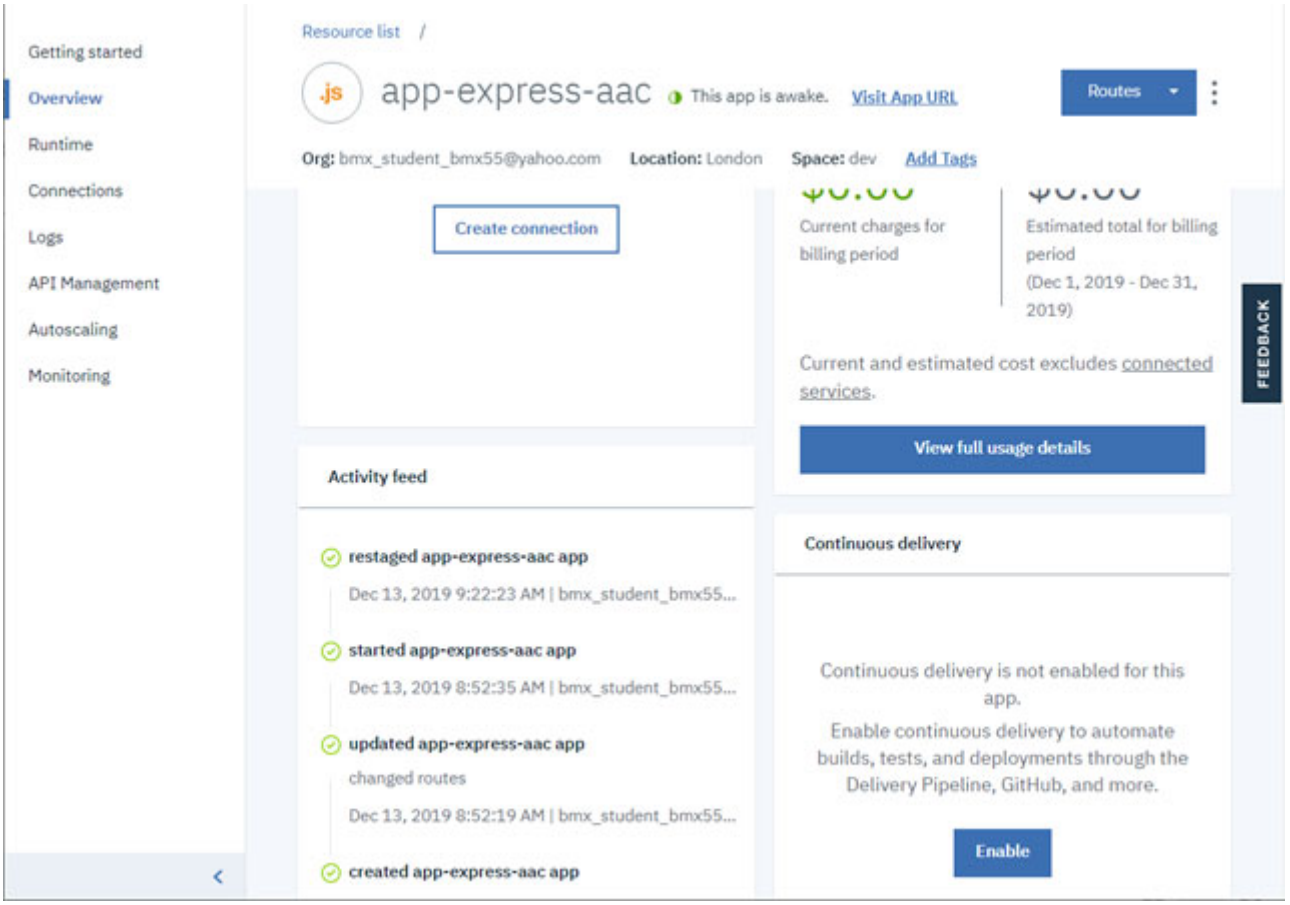
Cloning the Node.js Server application

In this part, you deploy the Node.js application, which is the server side that connects the client side with the Tone Analyzer service. Complete the following steps:

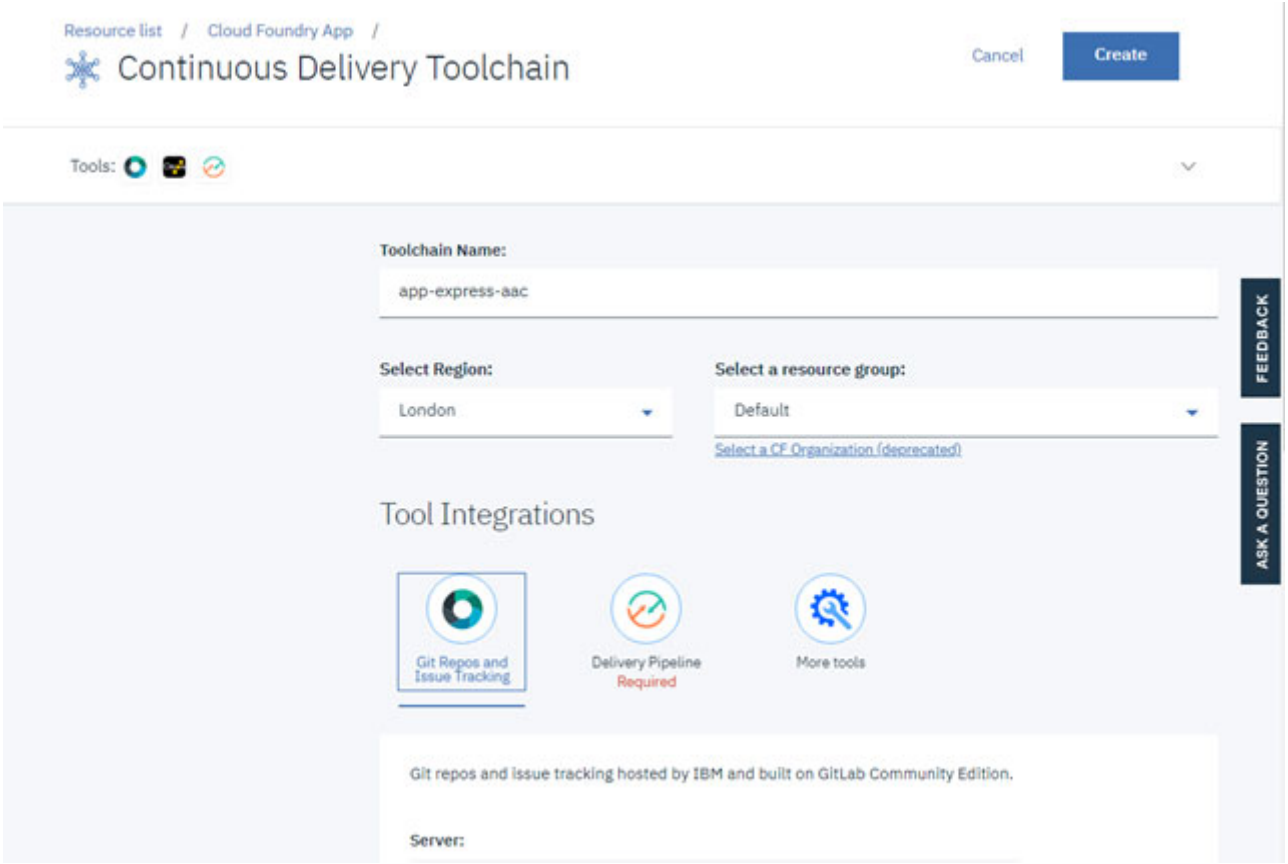
__ 1. Go to the IBM Cloud dashboard.

__ 2. Expand Coud Foundry Apps and open the details page for the Node.js application (app-express-xxx) that you created in Part 2.

__ 3. Click **Overview**. The window that is shown in the following figure opens.

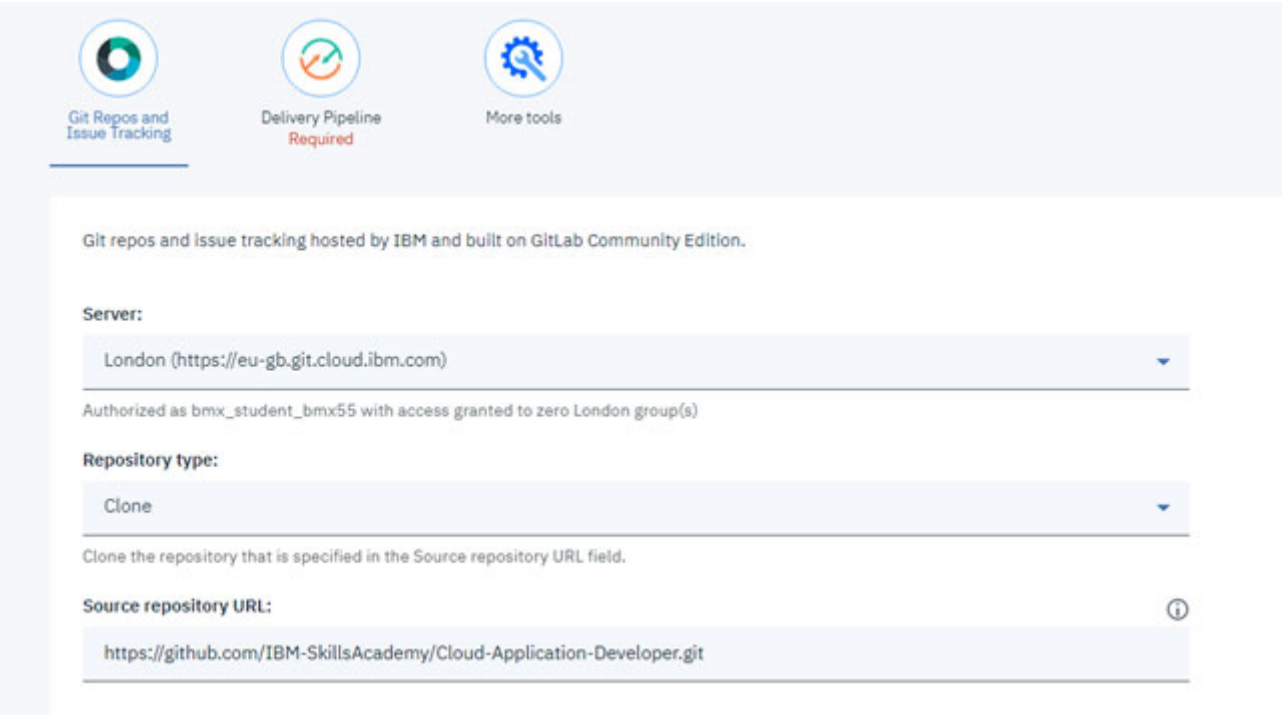


__ 4. Scroll down and click **Enable** in the “Continous delivery” pane. The window that is shown in the following figure opens.



__ 5. In the “Git Repos and Issue Tracking” pane, complete the following fields, as shown in the following figure:

- Repository type: **Clone**
- Source repository URL: https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer.git

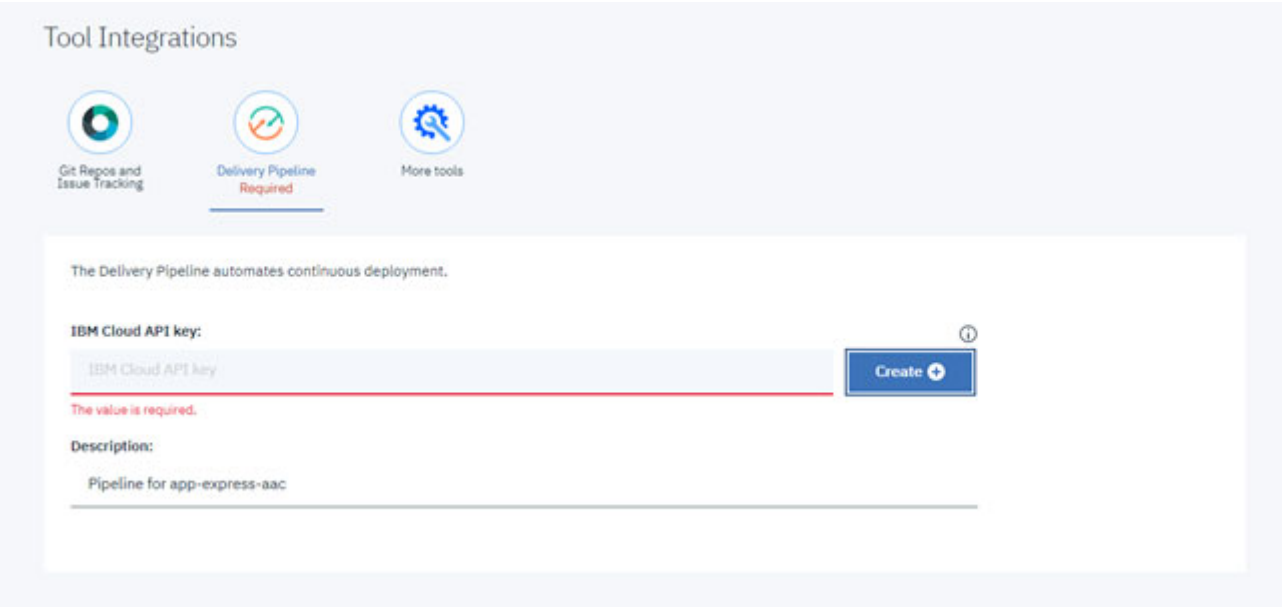


Troubleshooting

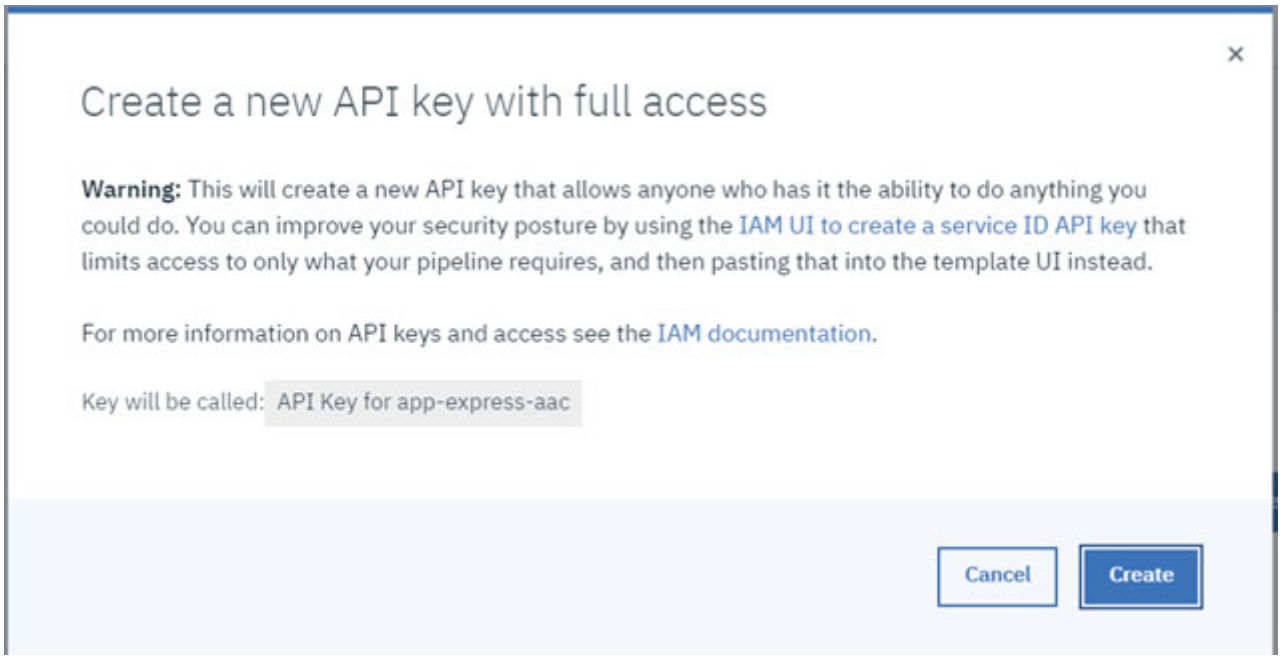
If the default Repository Name is already in use as shown in the following figure, change the name to make it unique.

__ 6. Click **Delivery pipeline**.

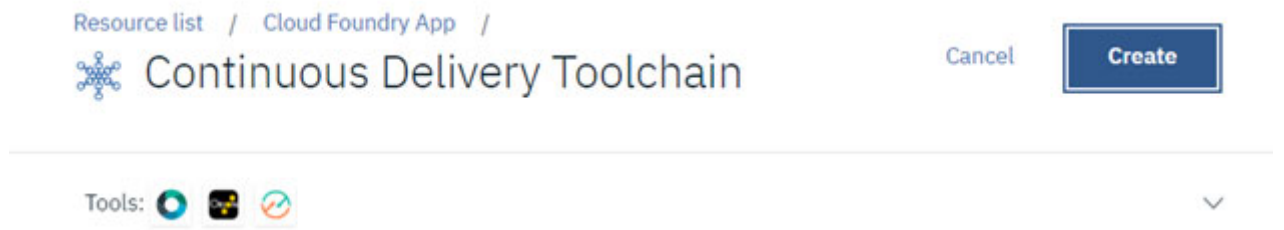
__ 7. Click **Create+** for IBM Cloud API Key, as shown in the following figure.



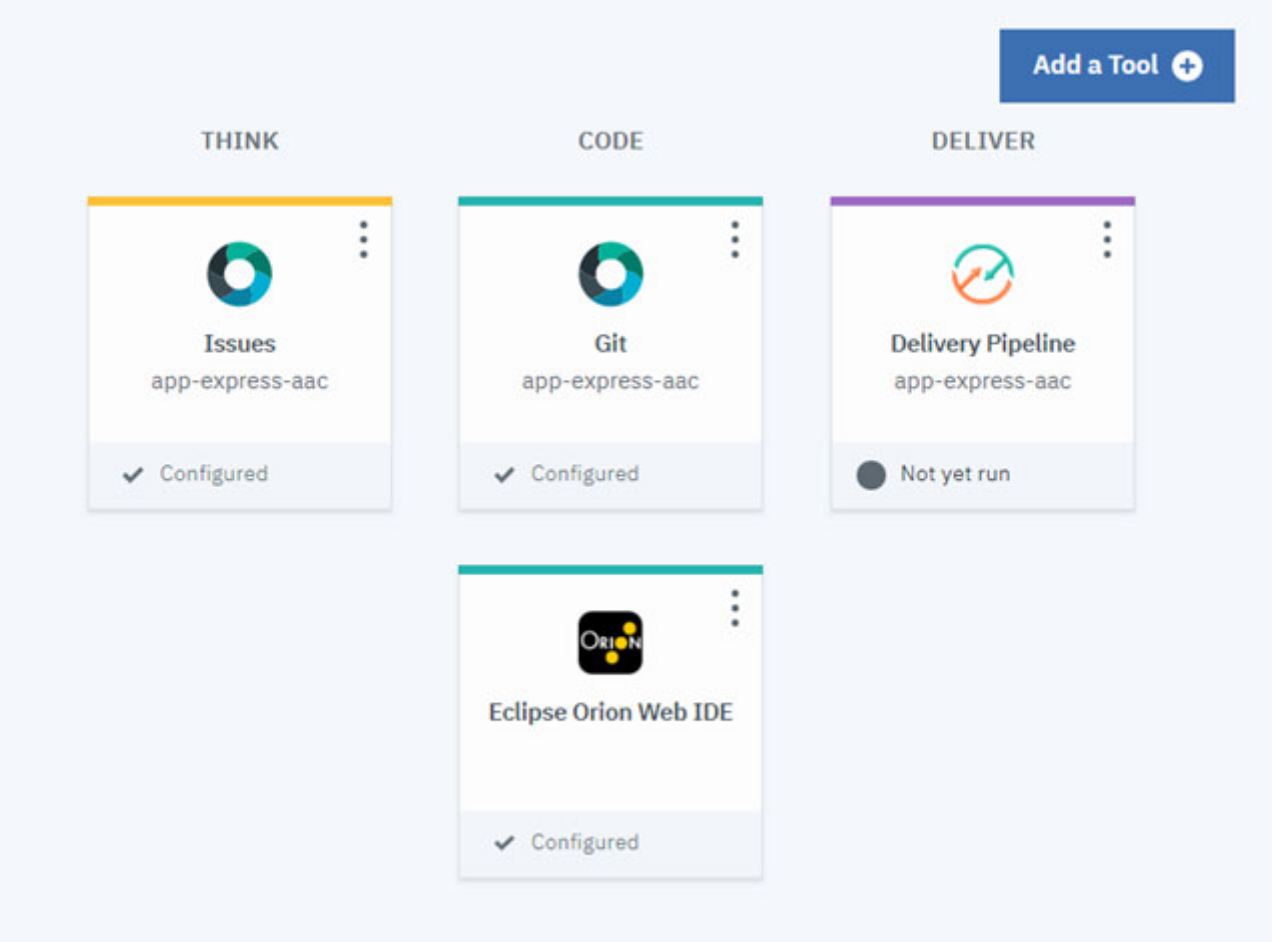
__ 8. In the Create API key window that opens, click **Create**, as shown in the following figure.



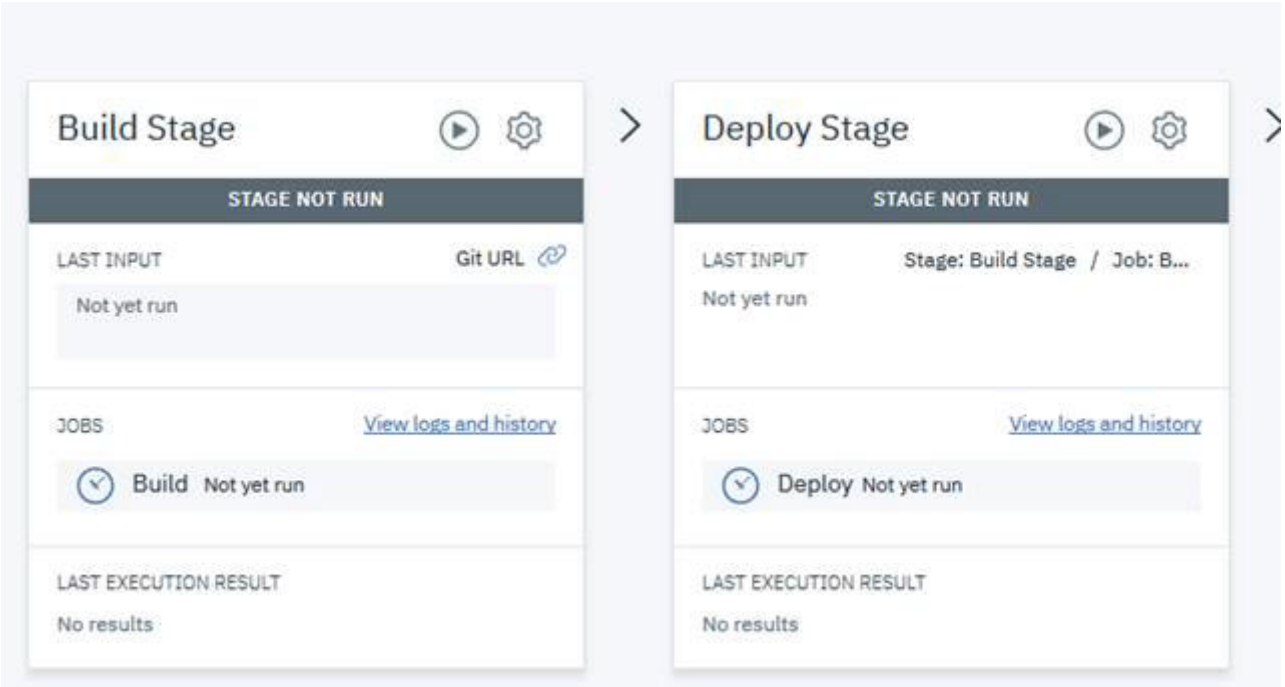
__ 9. Click **Create**, as shown in the following figure.



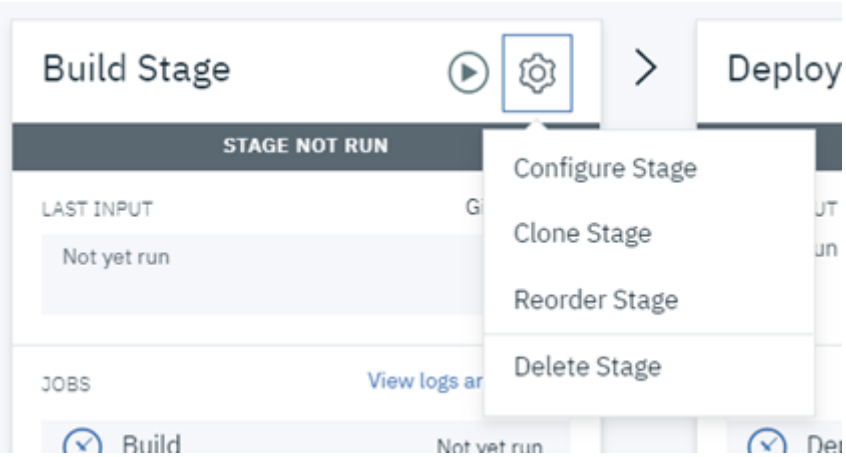
You are redirected to the window that is shown in the following figure.



__ 10. Click **Delivery Pipeline**.You are redirected to the Build and Deploy Stage window, as shown in the following figure.



__ 11. In the Build Stage pane, click the **Stage Configuration** icon (the gear icon) and then select **Configure Stage**, as shown in the following figures.

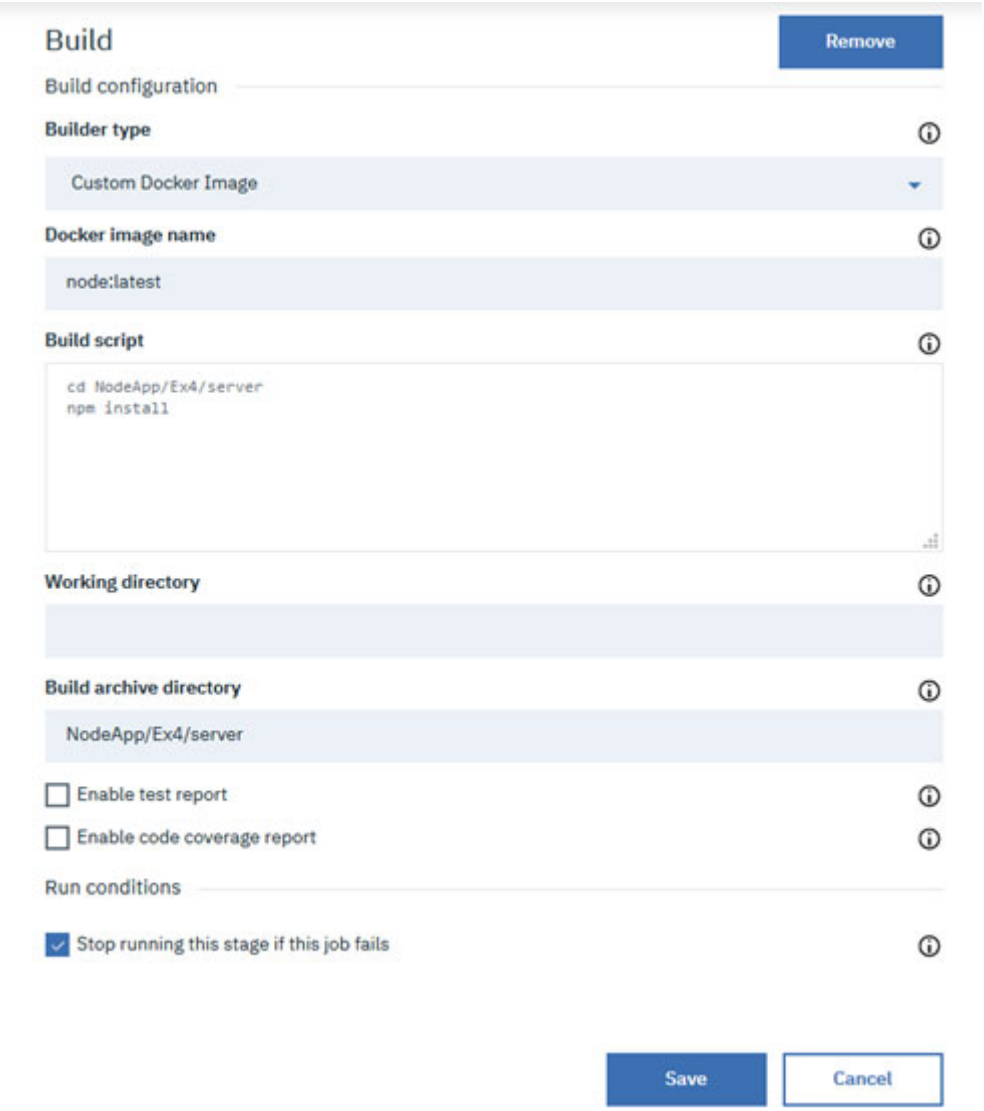


__ 12. Add the Build configuration options that will be used to build the Node.js application by completing the following fields, as shown in the following figure:

- Builder type: **Custom Docker Image**
- Docker image name: `node:latest`
- Build script:

```
cd NodeApp/Ex4/server
npm install
```

- Build archive directory: `NodeApp/Ex4/server`

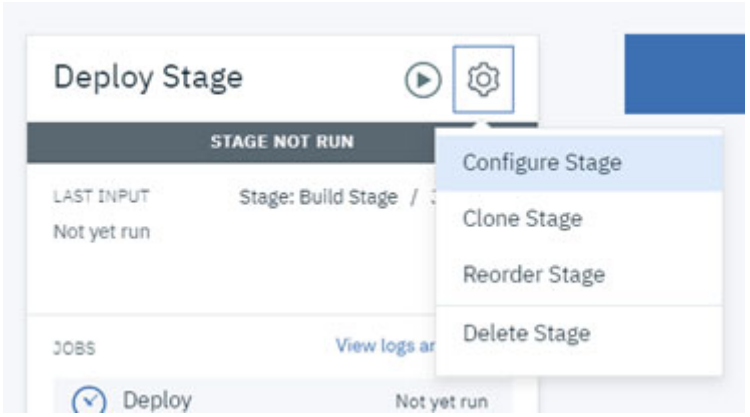


Note

You need a custom docker image to build this application because you need the latest node version (Node.js 7.6 is the official supported version for ES8).

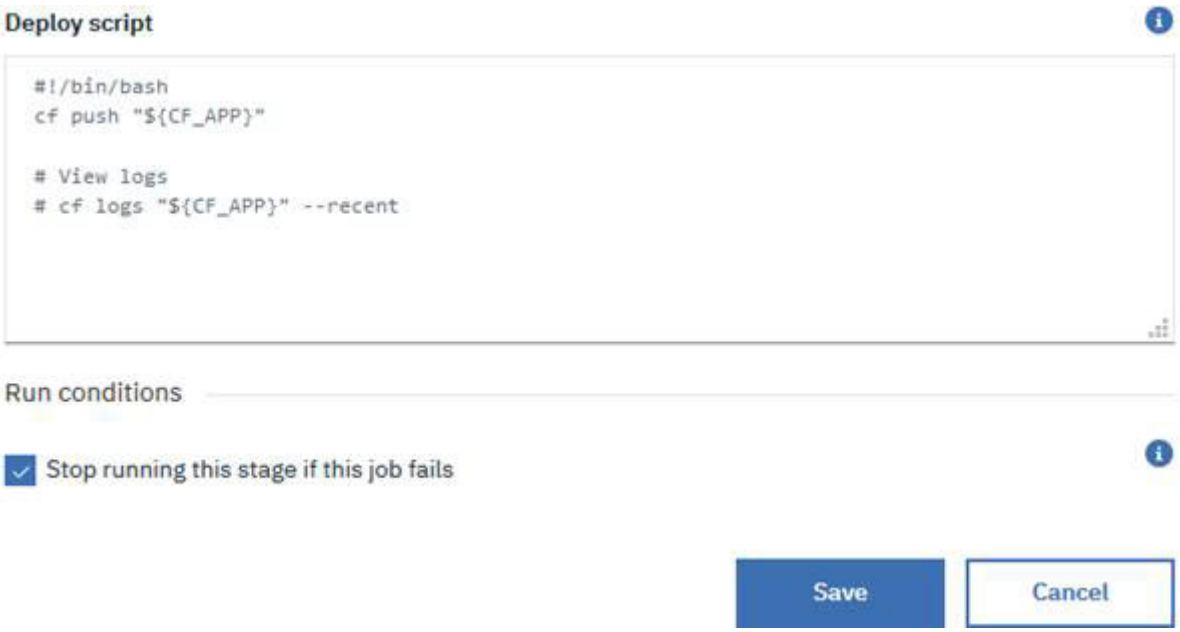
__ 13. Click **Save**.

__ 14. In the Deploy Stage pane, click the **Stage Configuration** icon (the gear icon) and select **Configure Stage**, as shown in the following figure.



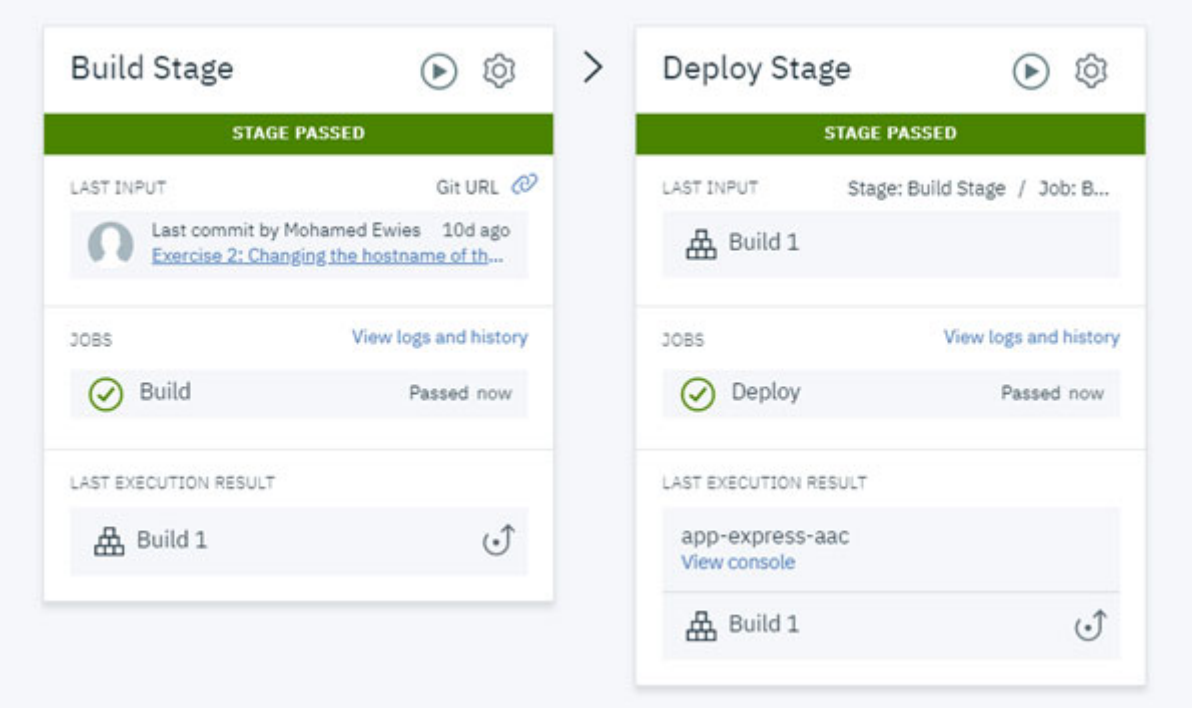
__ 15. The deployment script must look like the following code:

```
#!/bin/bash
cf push "${CF_APP}"
#view logs
#cf logs “${CF_APP}” --recent
```



__ 16. Click **Save**.

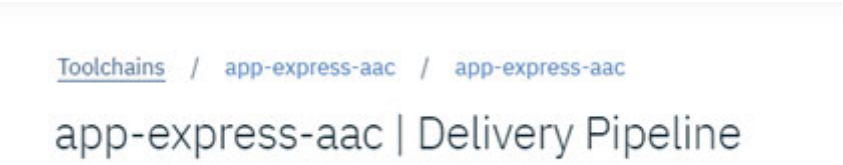
__ 17. Click the **run** icon (the arrow in a circle icon) in the Build Stage. The Deploy Stage runs automatically after the Build Stage ends successfully. The expected results for both stages is “STAGE PASSED”, shown in the following figure.



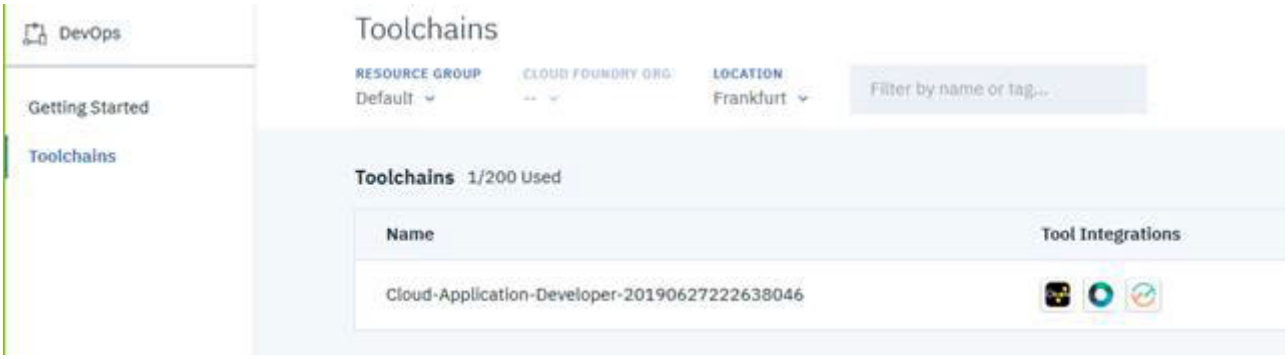
Part 5: Integrating the React application and the Node.js application

In this part, you update the URL in the React application to send the HTTP request to the Node.js application. Complete the following steps:

__ 1. Click **Toolchains**, as shown in the following figure.

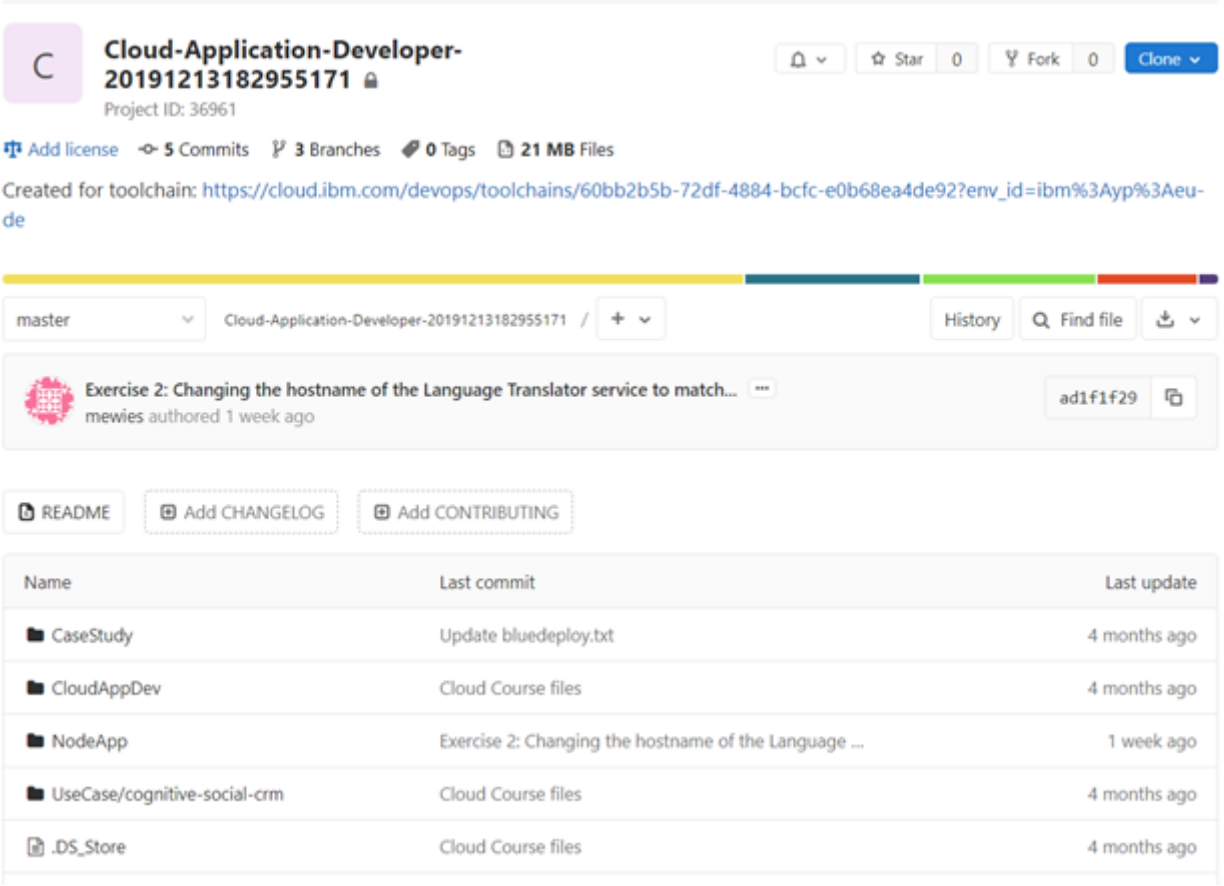


__ 2. Select the location that you wrote down in section *Cloning the* React application, step __ 2, which is Frankfurt in this example, as shown in the following figure. This is the toolchain for the React application.



__ 3. Under Tools Integrations, click the **Git** icon.

__ 4. The Git repo for the Cloud Application Developer course is displayed, as shown in the following figure. The folder NodeApp contains the code for the Node.js exercises.



__ 5. Open the folder **NodeApp**.

__ 6. Open the folder **Ex4**.

__ 7. Open the folder **client**.

__ 8. Open the folder **src**.

__ 9. In the `App.js` file, click **Edit** and change the baseUri (line 6) to the URL of the Node.js application that you saved in *Part 2.Creating a Node.js application* step __ 7. The following figure shows the baseURI *before* the change.

App.js 1.56 KB

Edit Replace Delete

```
import React, { Component } from 'react';
import './App.css';
import Result from './Result';

export default class App extends Component {
  baseUri = 'http://localhost:5001';

  constructor() {
    super();

    this.state = {
      text: 'Four score and seven years ago our fathers brought forth on this continent, a new nation,
      tones: []
    };
  }

  componentDidMount() {
    document.body.style.background = '#cafafe';
    document.getElementById('text-input').value = this.state.text;
    this.getTones();
  }
```

The following figure shows the baseUri *after* the change.

Edit file

Write Preview changes

master NodeApp/Ex4/client/src

Soft wrap

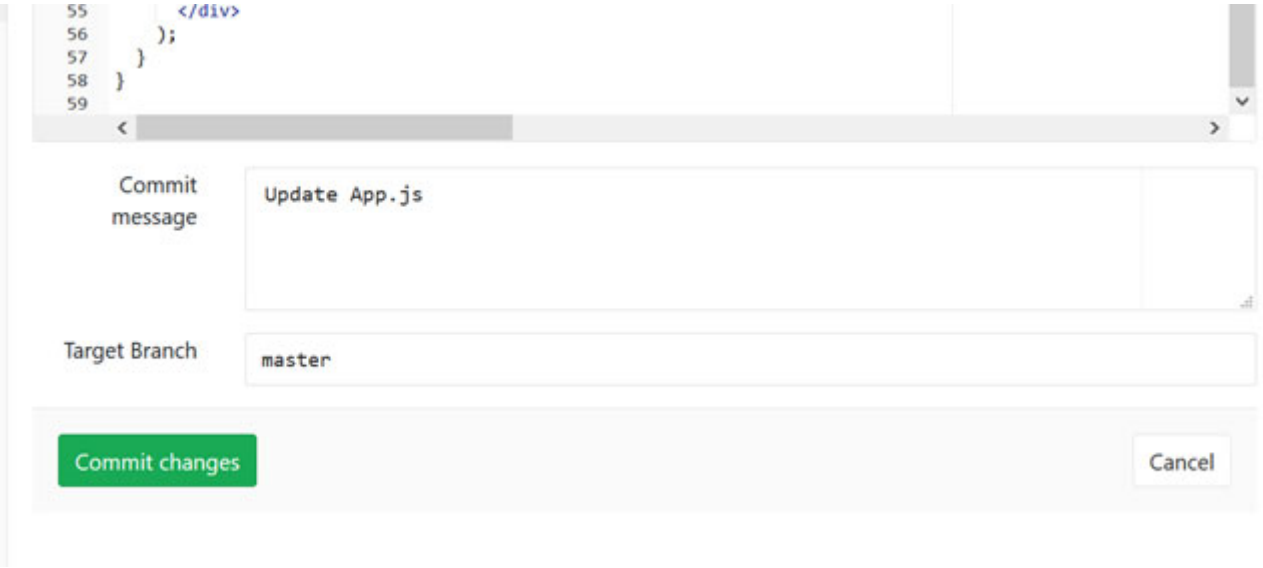
text

```
import React, { Component } from 'react';
import './App.css';
import Result from './Result';

export default class App extends Component {
  baseUri = 'https://app-express-aac.eu-gb.cf.appdomain.cloud';

  constructor() {
    super();

    this.state = {
      text: 'Four score and seven years ago our fathers brought forth on this continent, a new n
      tones: []
    };

    this.getTones = this.getTones.bind(this);
    this.updateText = this.updateText.bind(this);
  }
```

Attention

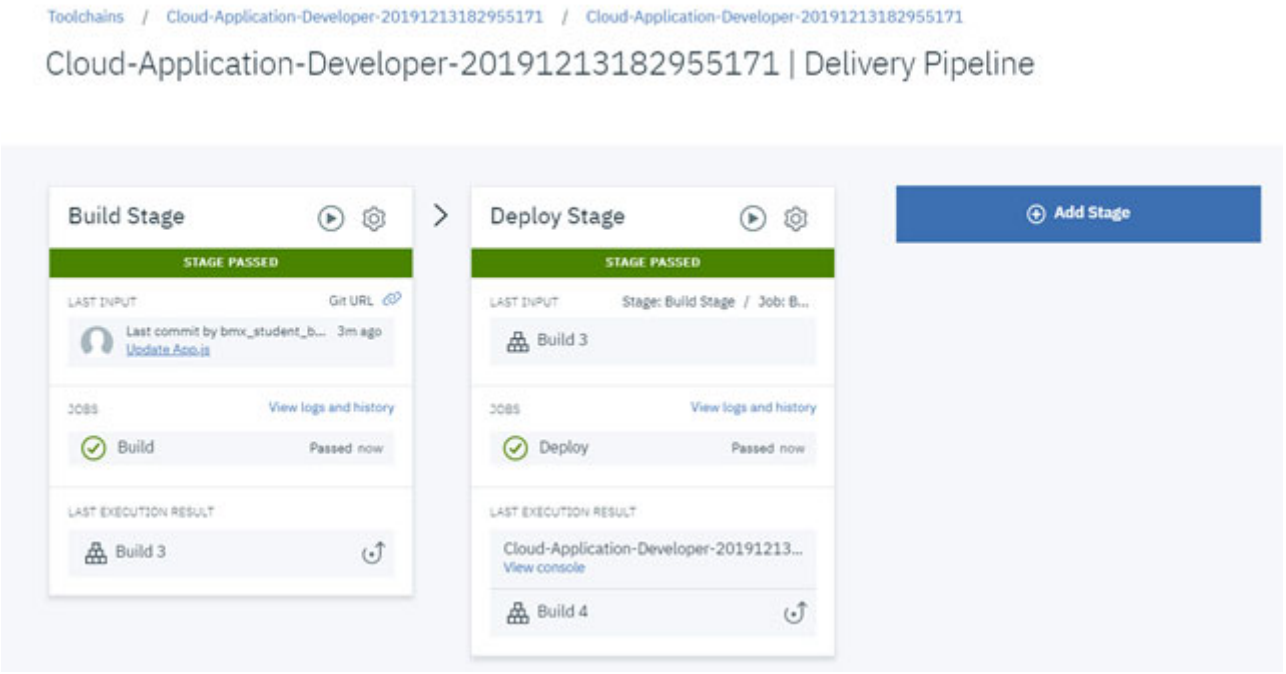
When you add the baseUri, make sure that it does not include a slash “/” at the end. The format is **`https://app-express-xxx.eu-gb.cf.appdomain.cloud`**.

__ 10. Scroll down and click **Commit changes**, as shown in the following figure.



__ 11. Go back to the Delivery Pipeline to monitor the progress of the deployment. You can access your Toochains from your logged in IBM Cloud session at https://cloud.ibm.com/devops/. Click **Delivery Pipeline**.

__ 12. Wait a few minutes until your application is deployed again. The results that are shown in the following screen are displayed when the application completes deployment.



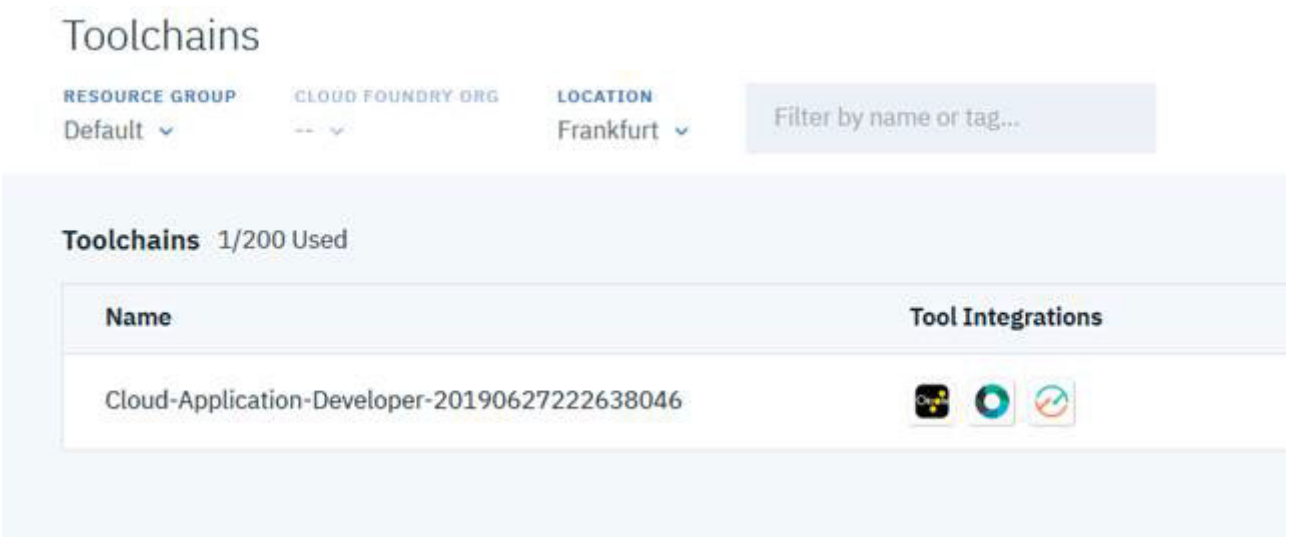
Hint

You can continue with the next part while the application is being deployed.

Part 6: Exploring the React application

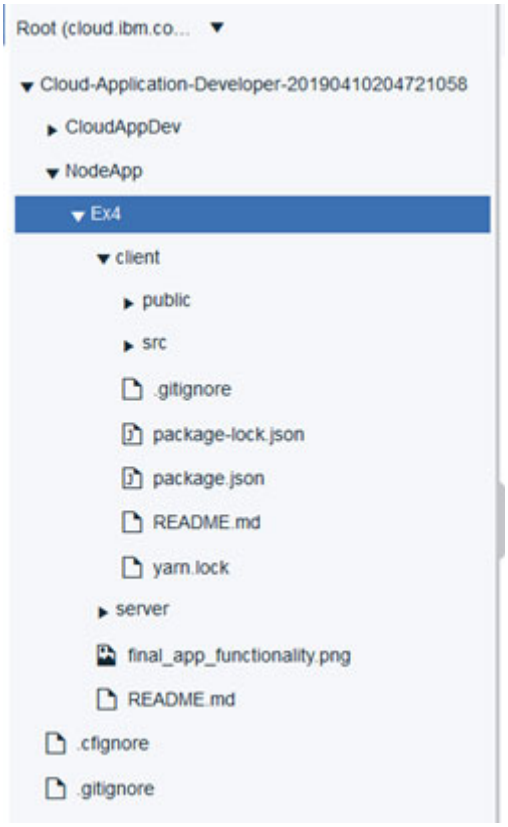
In this part, you review the React application code to learn more about the React framework. Complete the following steps:

__ 1. Click **Toolchains** and select the region where you created the toolchain for the React app. The window that is shown in the following figure opens.

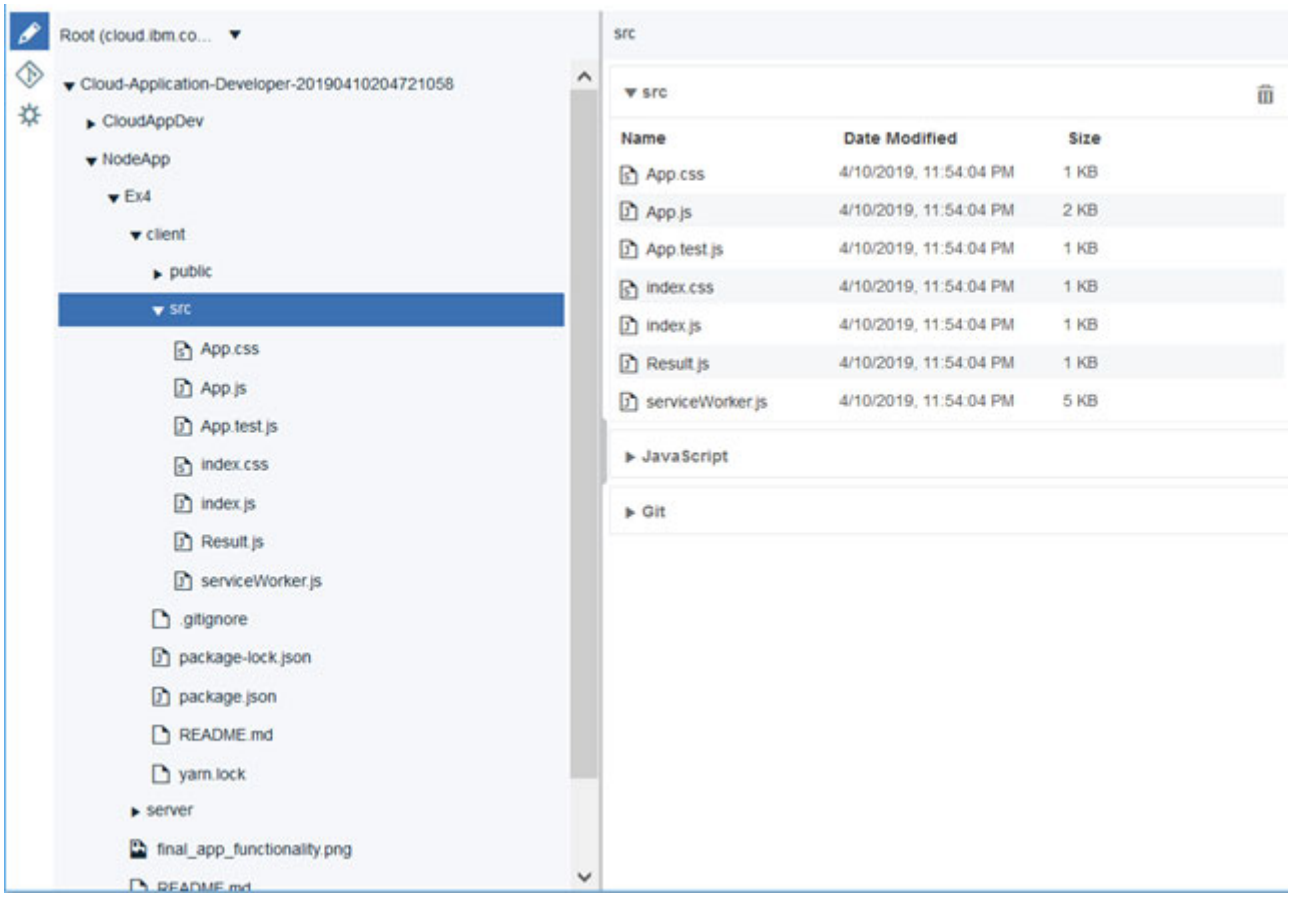


__ 2. Click **Eclipse Orion Web IDE**, as shown in the following figure.

__ 3. Select **NodeApp** > **Ex4**, as shown in the following figure.



You find two folders: **client** and **server**. The **client** folder contains the React application. To explore the client folder, click **client** > **src**, as shown in the following figure.



Information

The **src** folder contains JavaScript files and CSS files that you use in your application. Here are the files:

- `Index.js` is the JavaScript entry point.
- The basic folder structure in any React project is as follows:
  - package.`json`: This file contains the list of node dependencies that are needed.
  - `public/index.html`: When the application starts, this file is the first page that is loaded. It is the only `.html` file in the entire application because React is generally written in JavaScript XML (JSX). Also, this file has the line of code `<div id="root"></div>`, which is significant because all the application components are loaded into this `div`.
  - `src/index.js`: This is the JavaScript file that corresponds to `index.html`. This file has the following significant line of code:
  - ReactDOM.render(<App />, document.getElementById('root'));
  - `src/index.css`: The CSS file that corresponds to `index.js`. It carries all the styles.
  - `src/App.js`: This is the file for App Component, which is the main component in React that acts as a container for all the other components.

- `src/App.css`: This is the CSS file that corresponds to App Component, which carries all the styles.

__ 4. Click **App.js**. This file is where you can create your components, as shown in the following figure.

```
import React, { Component } from 'react';
import './App.css';
import Result from './Result';

export default class App extends Component {
  baseUrl = "http://localhost:5001";

  constructor() {
    super();

    this.state = {
      text: 'Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.
      tones: []
    };
  }

  componentDidMount() {
    document.body.style.background = "#cafafe";
    document.getElementById('text-input').value = this.state.text;
    this.getTones();
  }

  updateText(e) {
    this.setState({
      text: e.target.value
    });
  }

  async getTones() {
    try {
      const response = await fetch(`${this.baseUrl}/tone?text=${this.state.text}`);
      const toneResponse = await response.json();

      this.setState({
        tones: toneResponse.document_tone.tones
      });
    } catch (error) {
      this.setState({
        tones: []
      });
    }
  }

  render() {
    return (
      <div className="container">
        <h1>Tone Analyser</h1>
        <div>
          <div className="form-group">
            <textarea className="form-control" id="text-input" placeholder="Enter text here" onChange={e=>this.updateText(e)}></textarea>
          </div>
          <button className="btn btn-info" onClick={() => this.getTones()}>Retrieve Tones</button>
        </div>
        <Result text={this.state.text} tones={this.state.tones}/>
```

The previous figure shows what the top of the example `App.js` file looks like. When you create a component in a React application, consider the following points:

- Import React and any stylesheets from **react**. Then, use the same syntax to import all your individual components. In this exercise, you import the Result component that shows the results of the Tone Analyzer.
- The component name is App, which is denoted by the file name that is shown in the following statement:

  ```
  export default class App extends Component {}
  ```

- The *constructor* is where you initialize a state. If you do not initialize a state and bind methods, you do not need to implement a constructor for your React component. In this exercise, you initialized the text and the tones, as shown in the following lines:

```
constructor() {
super();
this.state = {
text: 'Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the proposition
that all men are created equal. Now we are engaged in a great civil war, testing
whether that nation...',
tones: []
};
}
```

- The **componentDidMount()** method is a lifecycle method that runs after the component output is rendered to the Document Object Model (DOM). It is run immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go into this method. If you need to load data from a remote endpoint, this method is a good place to instantiate the network request. In this app.js file, this method is used to fetch the text that is written and call the **getTones()** function, as shown in the following lines:

```
componentDidMount() {
    document.body.style.background = '#cafafe';
    document.getElementById('text-input').value = this.state.text;
    this.getTones();
}
```

- The **getTones()** function is an async function that sends the text to the Node.js application and waits for a response before the function sets the state (setState) of the tones. The function applies the **async** and **await** properties in ES8 so that the function can be called asynchronously, but it waits until it receives a response before it runs. The following lines show how the function is implemented:

```
async getTones() {
    try {
      const response = await
fetch(`${this.baseUri}/tone?text=${this.state.text}`);
      const toneResponse = await response.json();

      this.setState({
        tones: toneResponse.document_tone.tones
      });
    } catch (error) {
      this.setState({
        tones: []
      });
    }
}
```

- After the previous lines, you have the **render()** and **return** functions. The **return** value of this function is shown on the screen. Whenever the **render()** function is called, the screen is rerendered automatically by the application.

```
render() {
    return (
      <div className="container">
        <h1>Tone Analyser</h1>
        <div>
          <div className="form-group">
            <textarea className="form-control" id="text-input"
placeholder="Enter text here" onChange={e=>this.updateText(e)}></textarea>
          </div>
          <button className="btn btn-info" onClick={() =>
this.getTones()}>Retrieve Tones</button>
        </div>
        <Result text={this.state.text} tones={this.state.tones}/>
      </div>
    );
  }
}
```

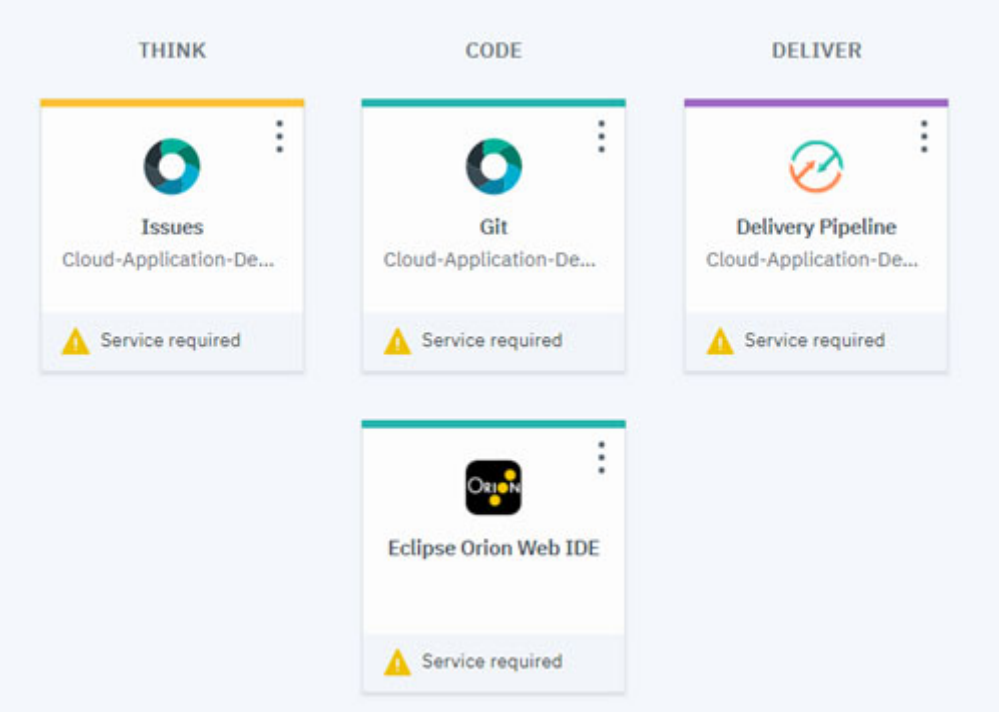
- The `state` feature is a private feature that belongs only to a single component. The `state` feature enables React components to change dynamically output over time in response to certain events. For example, `{this.state.text}` can be changed later by using the **setState()** function as follows:

```
this.setState({
        tones: toneResponse.document_tone.tones
;
```

Part 7: Exploring the Node.js application code

In this section, you explore the Node.js server application code.

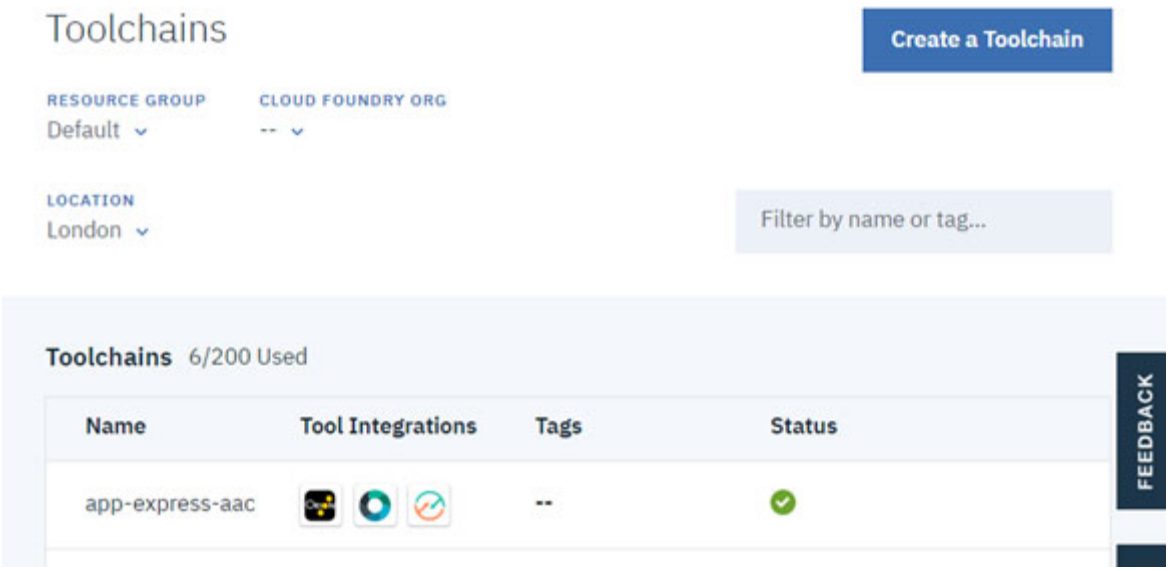
__ 1. To explore the Node.js application code, click the back arrow at the upper left in Eclipse Orion Web IDE. You are redirected to the window that is show in the following figure.



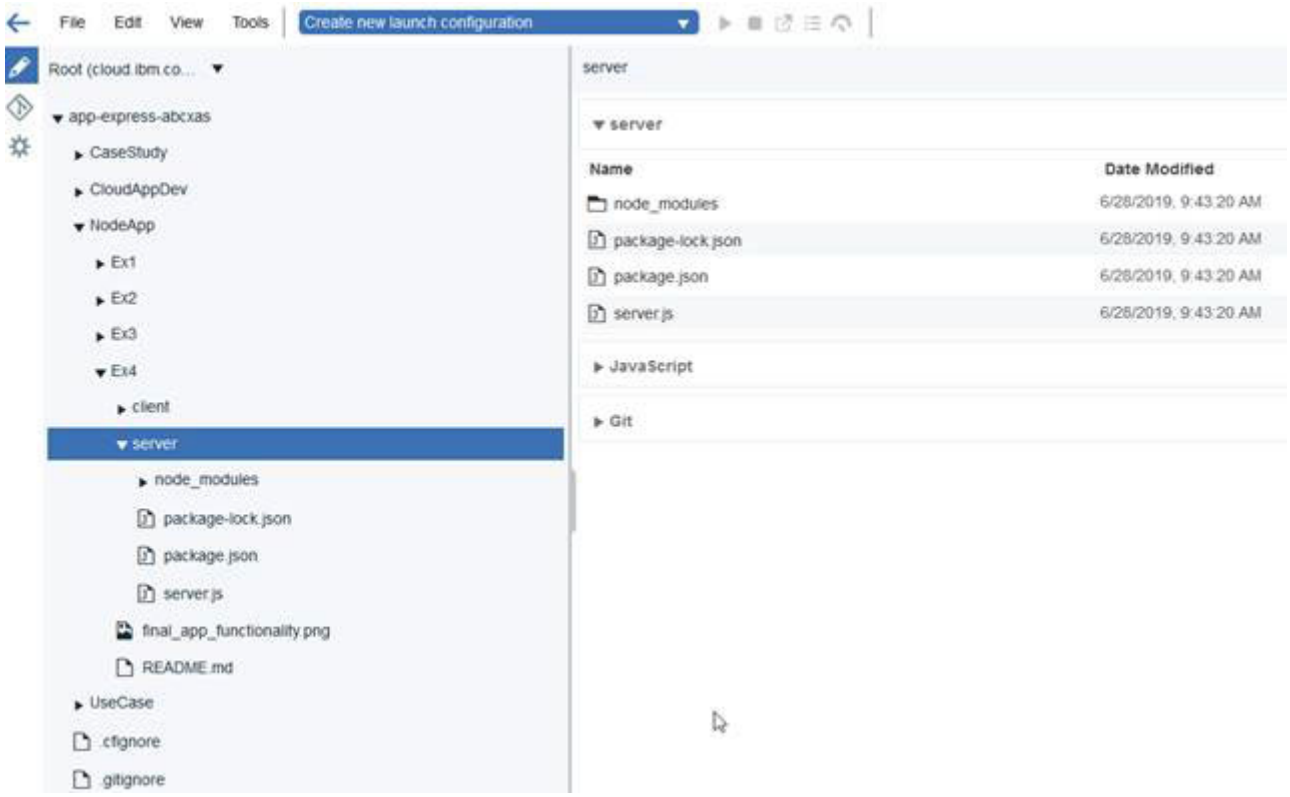
__ 2. Click **Toolchains**.

__ 3. Choose the Location where you created your toolchain for the Node.js application in section *Cloning* the Node.js Server application, step __ 5. In this example the location is London.

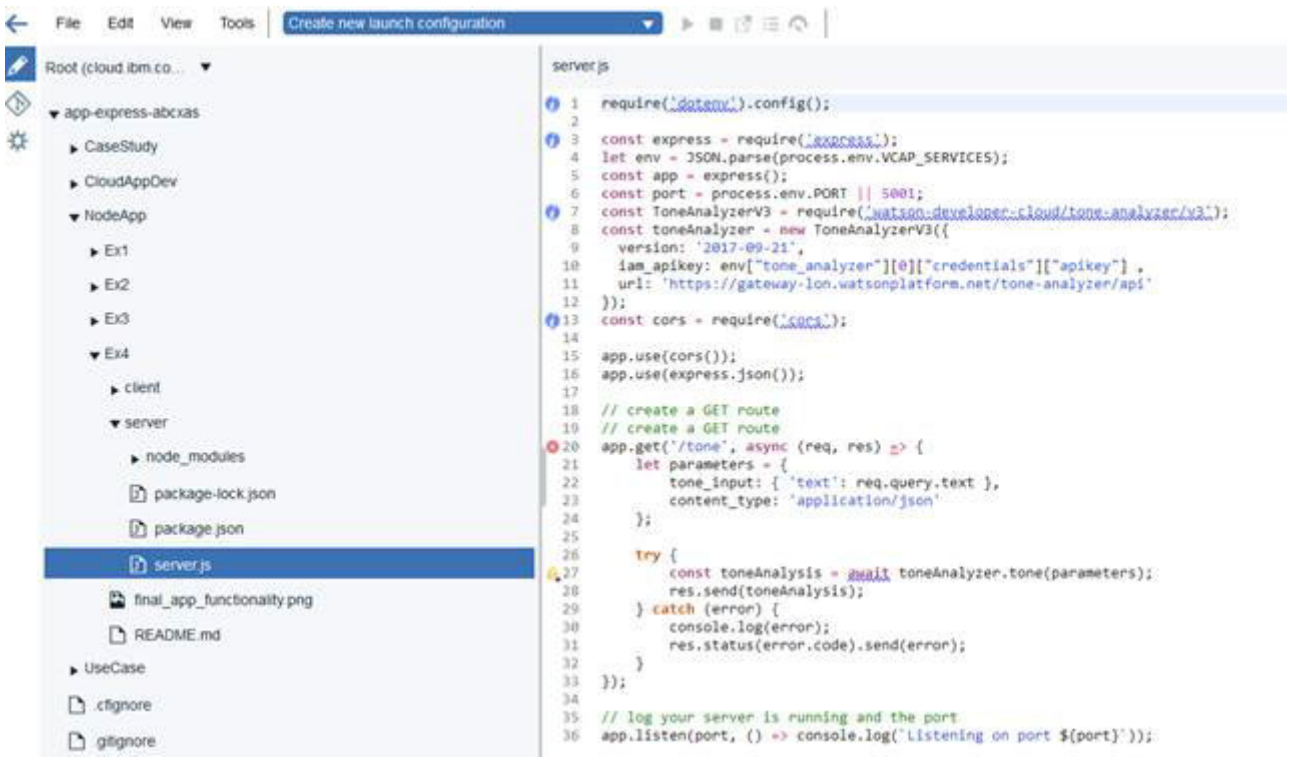
__ 4. Click the **Eclipse Orion Web IDE** icon under Tool Integrations.



__ 5. The window that is shown in the following figure opens.



__ 6. Click **NodeApp** > **Ex4** > **server**, and then select **server.js** to open the entry point of the server application, as shown in the following figure.



By using `server.js`, whenever the React application needs a result from Tone Analyzer, the server sends a **`GET HTTP`** request to the Node.js application when the Node.js application is running. The Node.js application sends the data to the Tone Analyzer service and returns the result in the HTTP response.

In this code, you find some of the ECMAScript features:

- The **`const`** and **`let`** keywords

```
const port = process.env.PORT || 5001;
let parameters = {
        tone_input: { 'text': req.query.text },
        content_type: 'application/json'
    };
```

  In ECMAScript 2015 (ES6), the two keywords **`let`** and **`const`** were added. The difference between **`const`** and **`let`** is that **`const`** is a signal that the identifier will not be reassigned. **`let`** is a signal that the variable might be reassigned, such as a counter in a loop or a value swap in an algorithm. **`let`** also signals that the variable is used only in the block in which it is defined**,** which is not always the entire containing function.

- The async/await function

  In ECMAScript 2017 (ES8), the async/await feature was added. The **`async`** function declaration defines an asynchronous function, which returns an Async Function object.

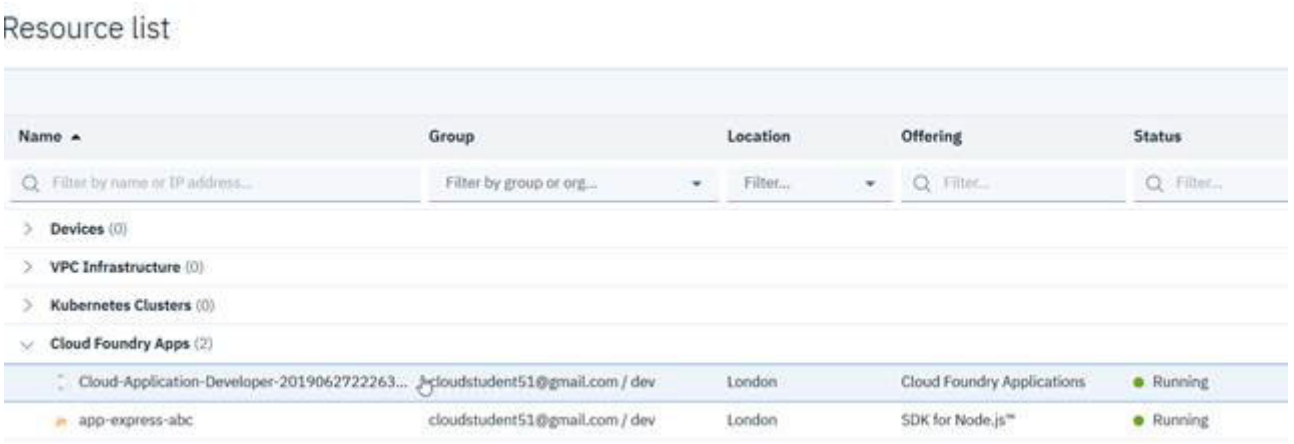
  An **`async`** function can contain an **`await`** expression, which pauses the execution of the **`async`** function and waits for the passed Promise's resolution and then resumes the **`async`** function's execution and returns the resolved value, as shown in this code snippet:

```
app.get('/tone', async (req, res) => {
    let parameters = {
        tone_input: { 'text': req.query.text },
        content_type: 'application/json'
    };

    try {
        const toneAnalysis = await toneAnalyzer.tone(parameters);
        res.send(toneAnalysis);
    } catch (error) {
        console.log(error);
        res.status(error.code).send(error);
    }
;
```

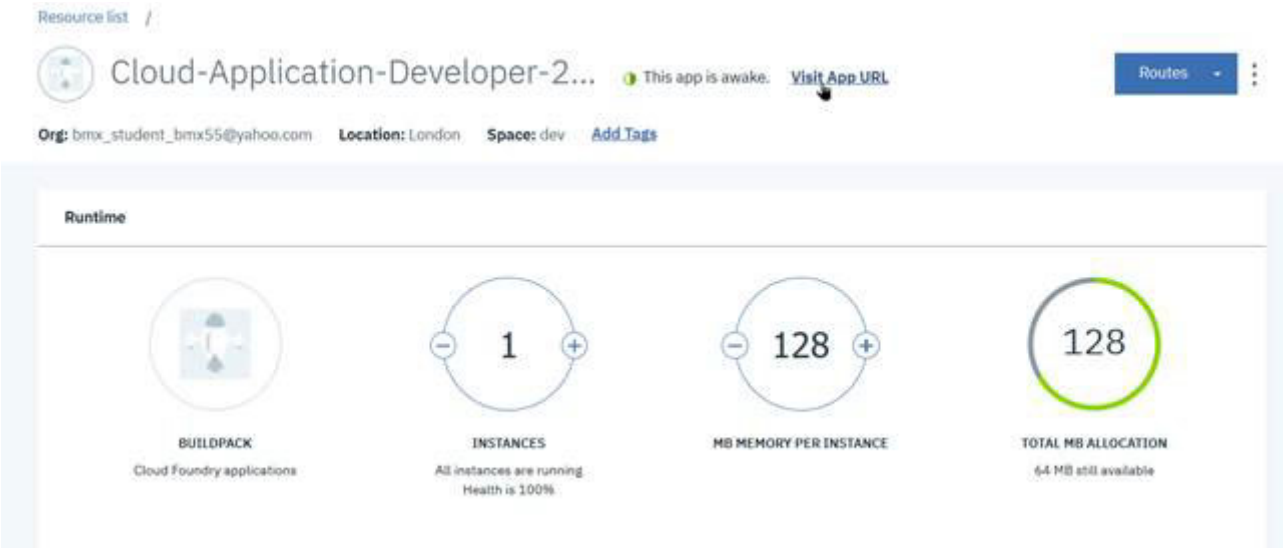
Part 8: Testing your application

In this part, you test your application to see the connection between the React application and the Node.js application. Complete the following steps:

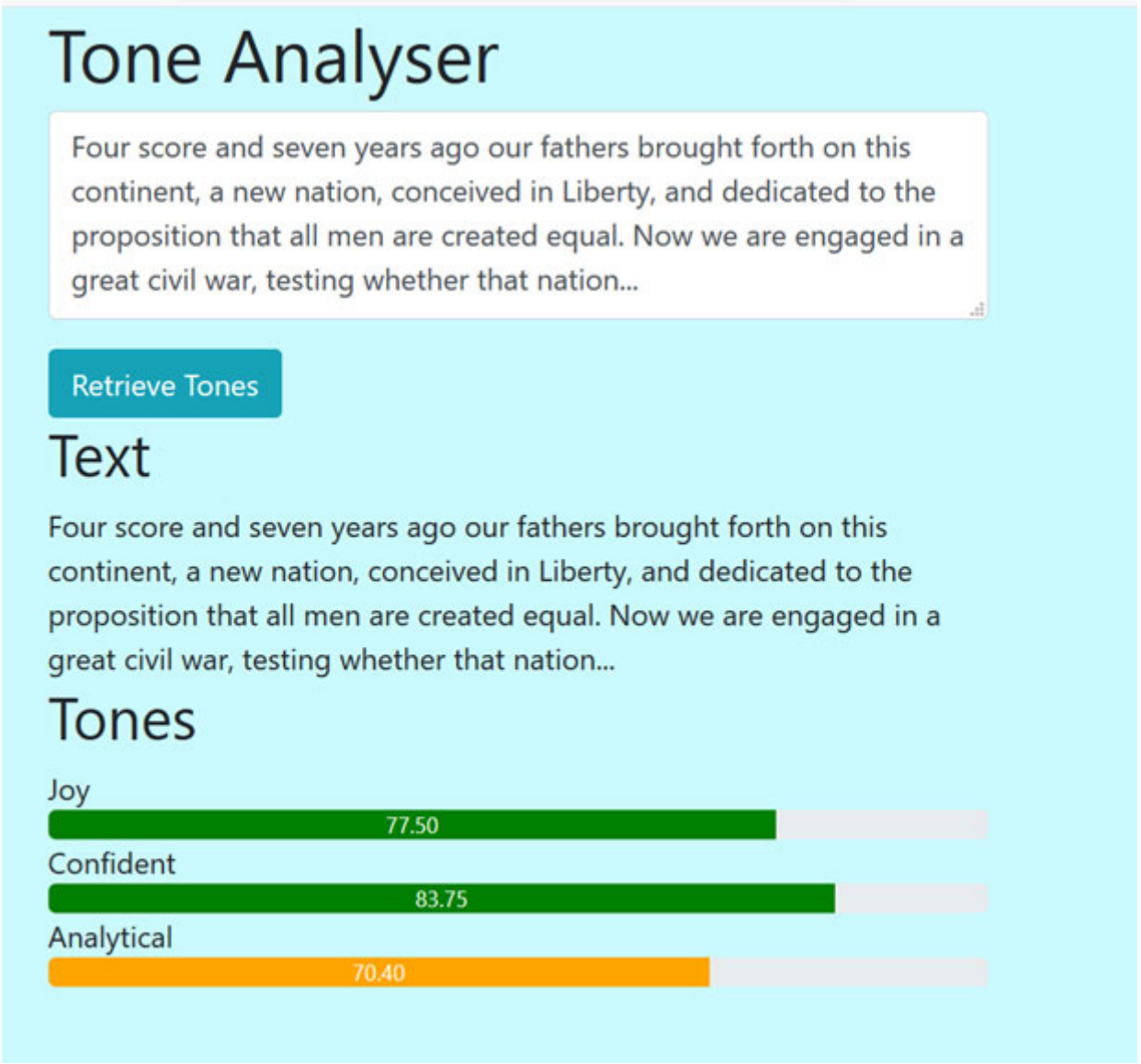
__ 1. From the IBM Cloud dashboard, click your React application’s name as shown in the following figure.



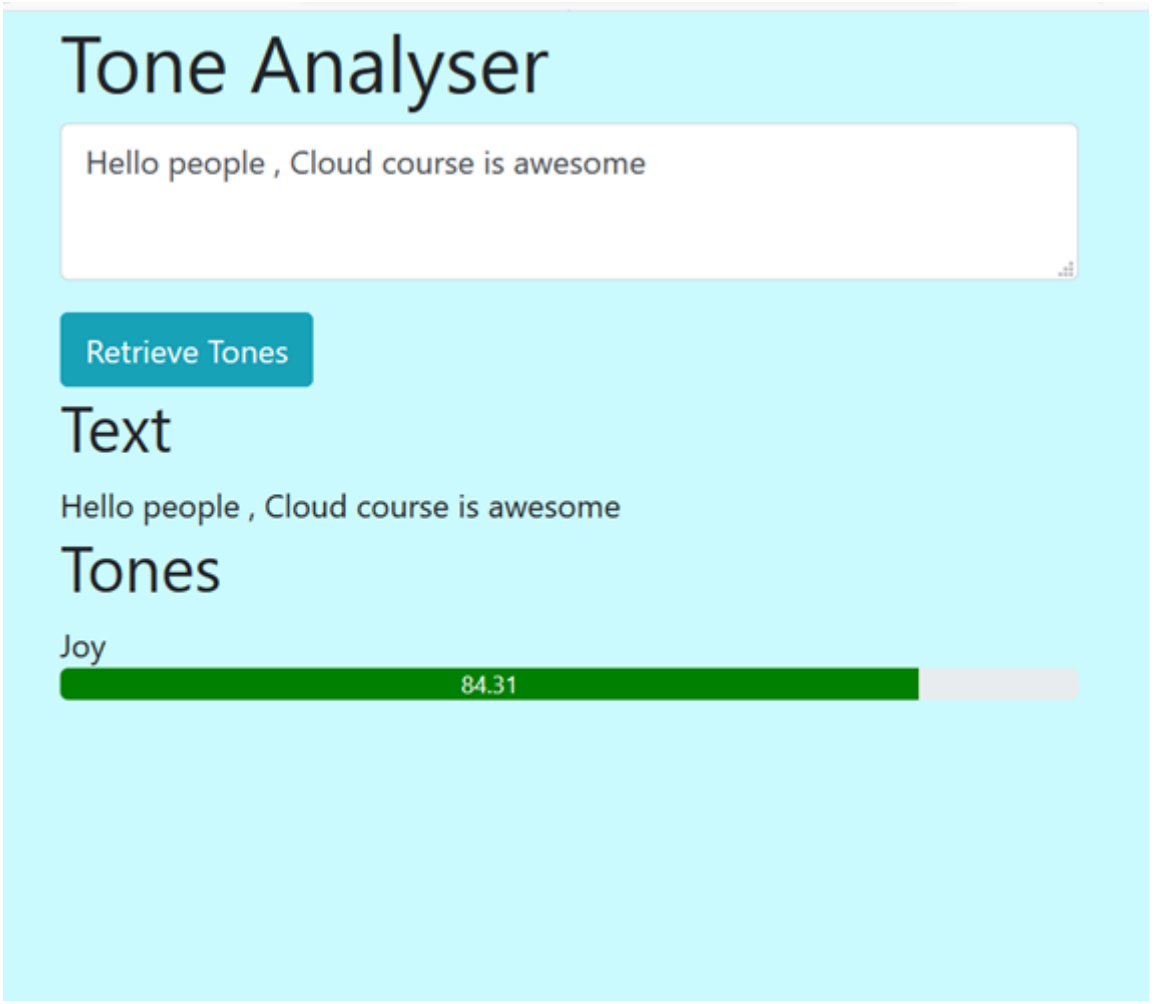
__ 2. Click **Visit App URL**, as shown in the following figure.



__ 3. Your front-end application opens in another browser tab as shown in the following figure.



__ 4. Enter text into the Tone Analyzer text box and click **Retrieve Tones**, as shown in the following figure.



Part 9: Cleaning up the environment

In this part, you clean up the environment.

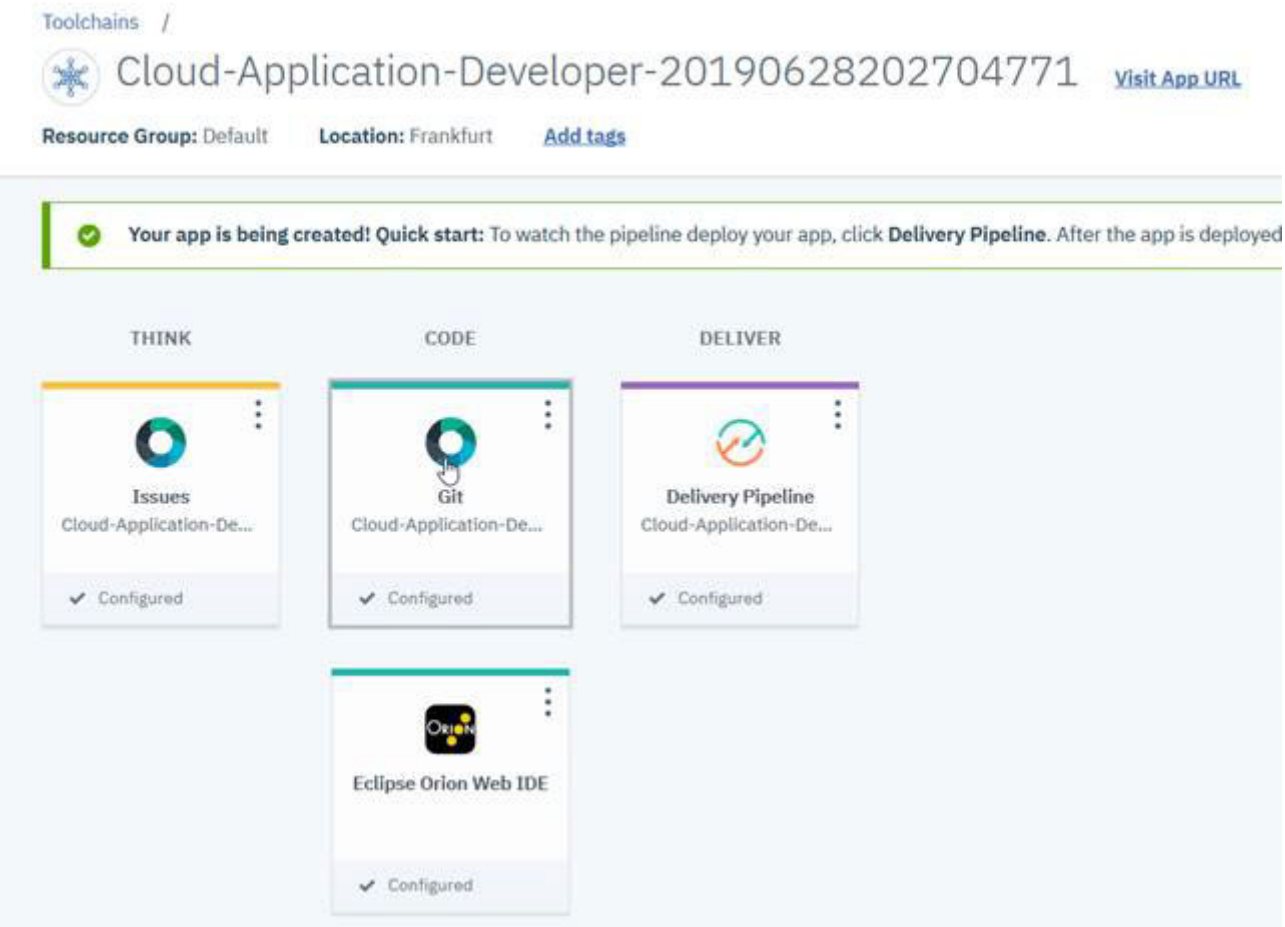
Removing the Git repos

In this section, you delete the Git repos for the React and Node.js application.

Complete the following steps:

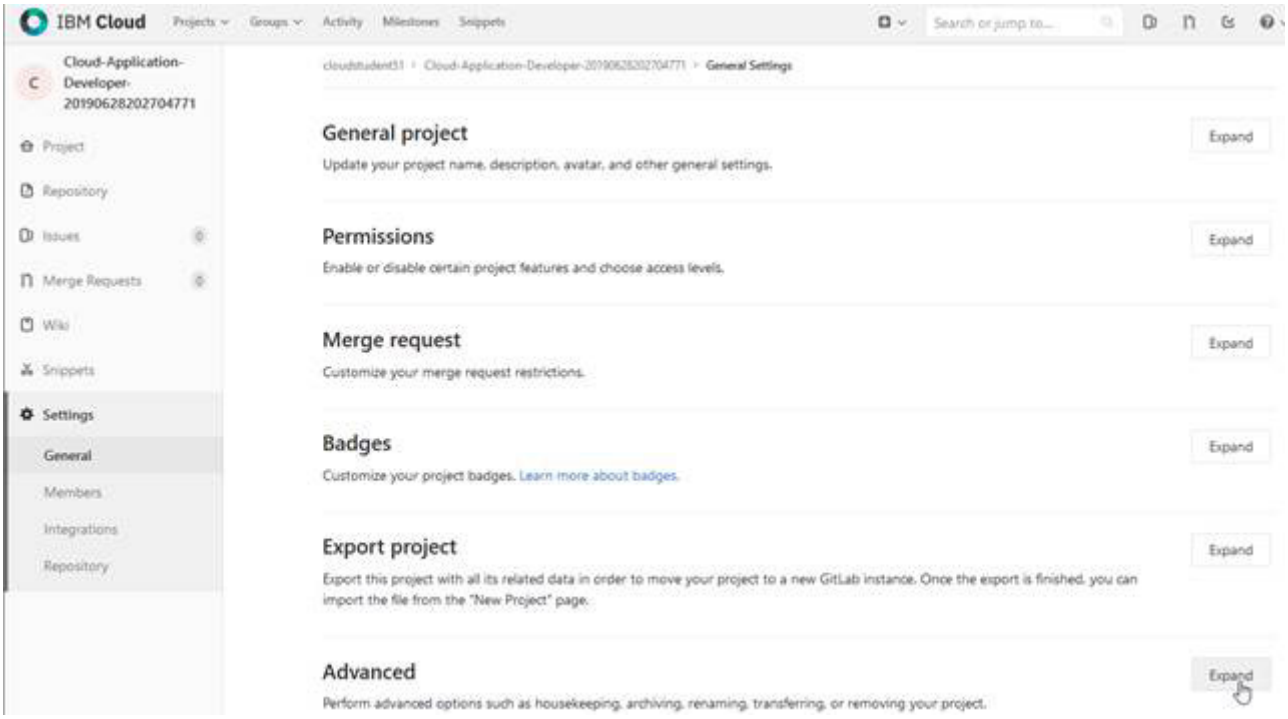
__ 1. Open the toolchain dashboard for the React app by selecting the location where you created it (Frankfurt in this example).

__ 2. Click **Git**, as shown in the following figure.



__ 3. From the left bar, click **Settings**.

__ 4. Click **Expand** for the **Advanced** section.



__ 5. Scroll down and click **Remove project**.

__ 6. Type the project name and click **Confirm.**

__ 7. Return to Toolchains (https://cloud.ibm.com/devops/) and open the toolchain dashboard for the Node.js app by selecting the location where you created it (London in this example)

__ 8. Repeat the previous steps to remove the Git repo for the Node.js application.

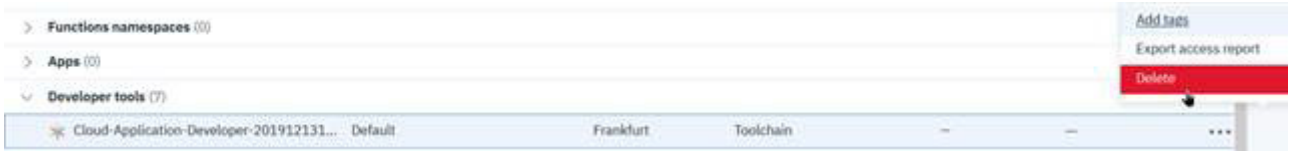
Deleting the toolchains

In this section, you delete the toolchains for the React and Node.js applications.

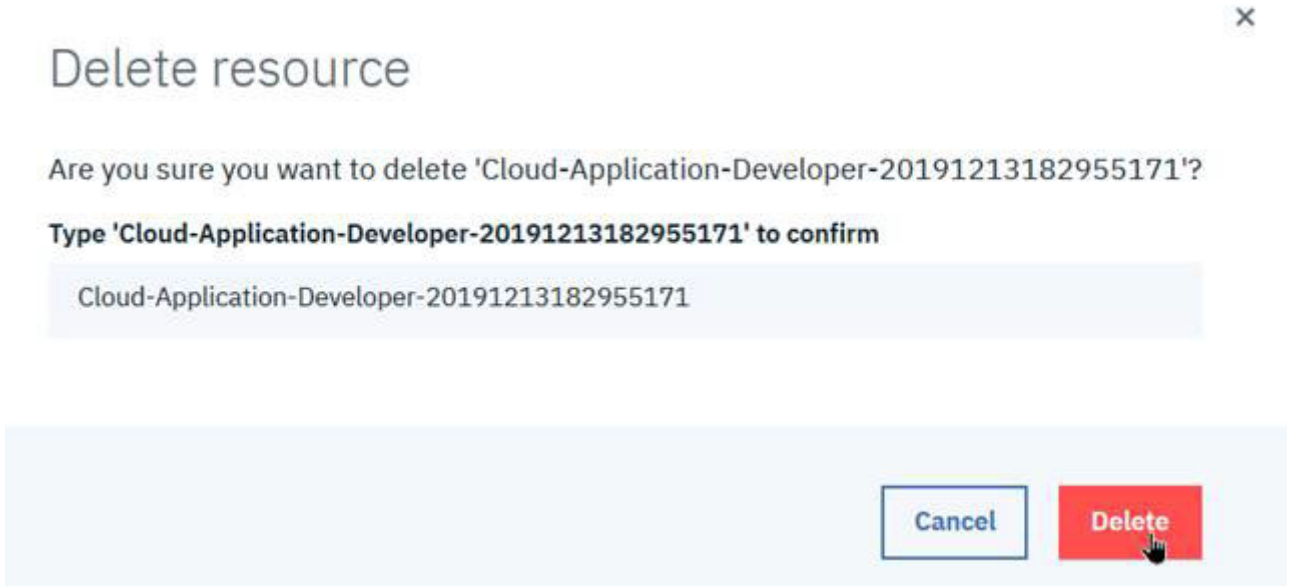
Complete the following steps:

__ 1. Select the toolchain for the React app.

__ 2. Click **Actions** (three dots) to the right of your toolchain and select **Delete**, as shown in the following figure.

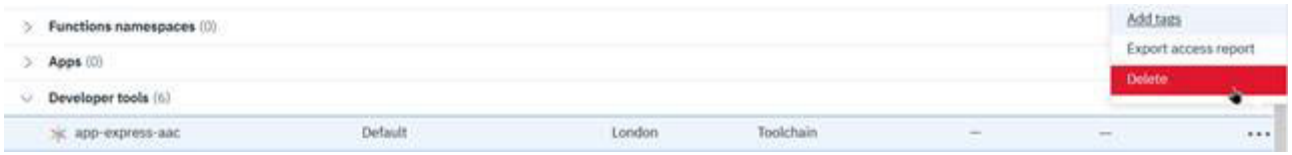


__ 3. Type the toolchain name to confirm deletion and click **Delete**, as shown in the following figure.

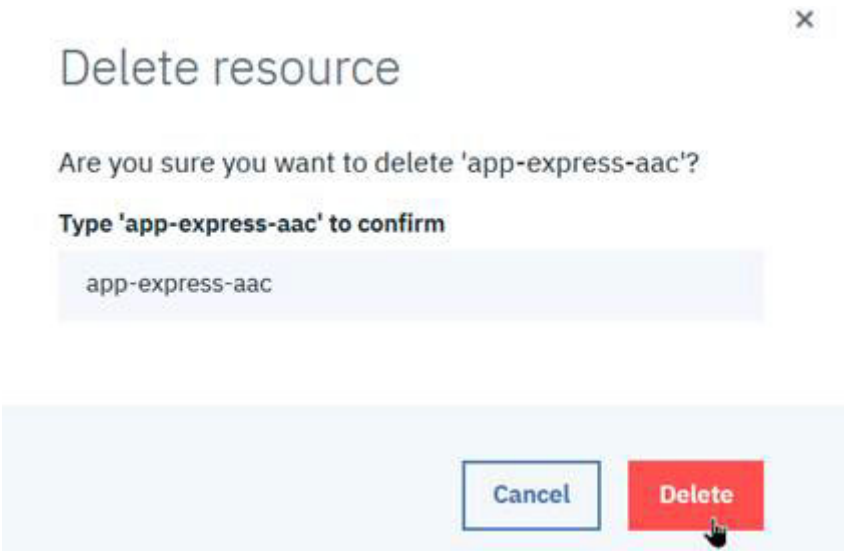


__ 4. Select the toolchain for the Node.js application.

__ 5. Click **Actions** (three dots) to the right of your toolchain and select **Delete**, as shown in the following figure.



__ 6. Type the toolchain name to confirm deletion and click **Delete**, as shown in the following figure.



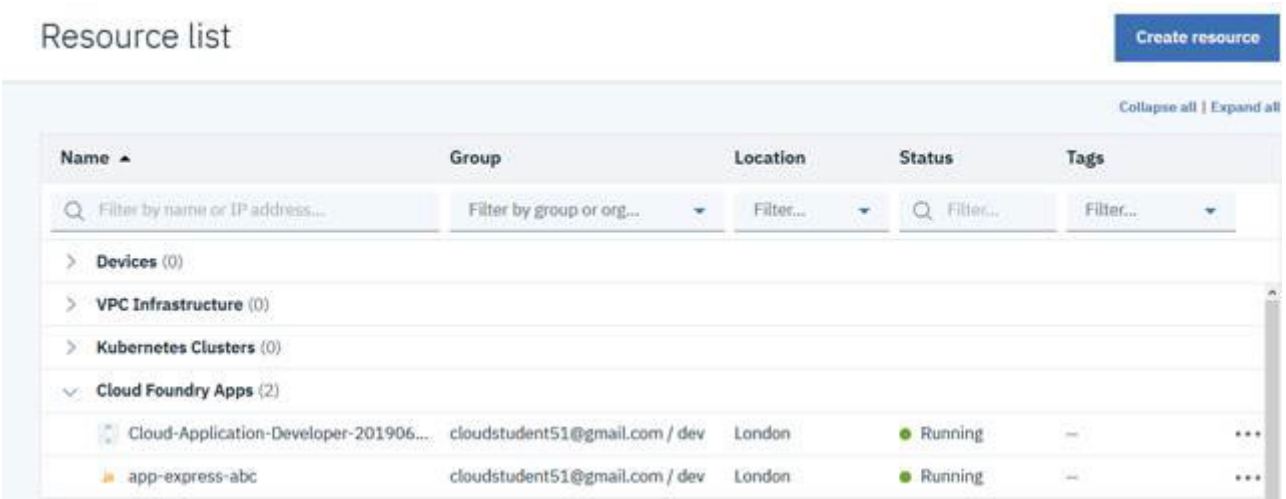
Deleting the React and Node.js applications

In this section, you delete the React and Node.js applications.

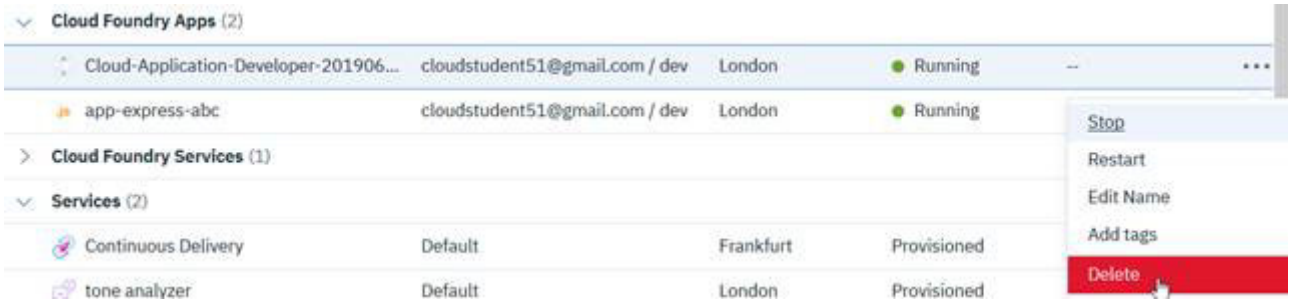
Complete the following steps:

__ 1. Open the IBM Cloud dashboard.

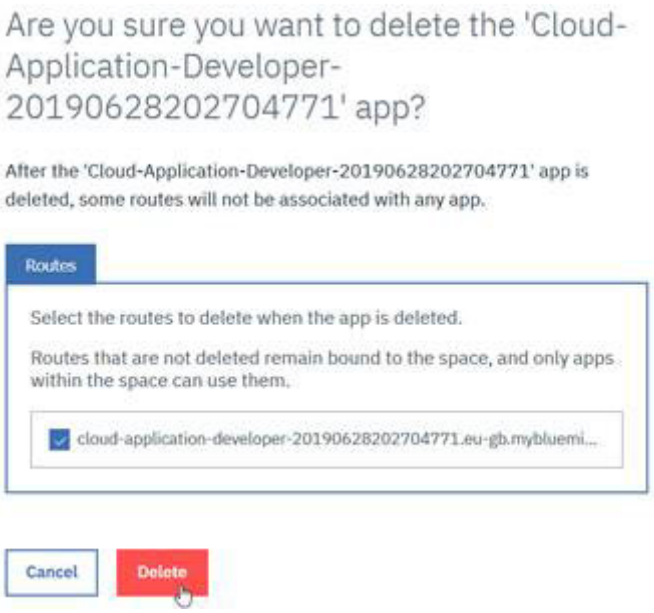
__ 2. Expand **Cloud Foundry Apps**, as shown in the following figure.



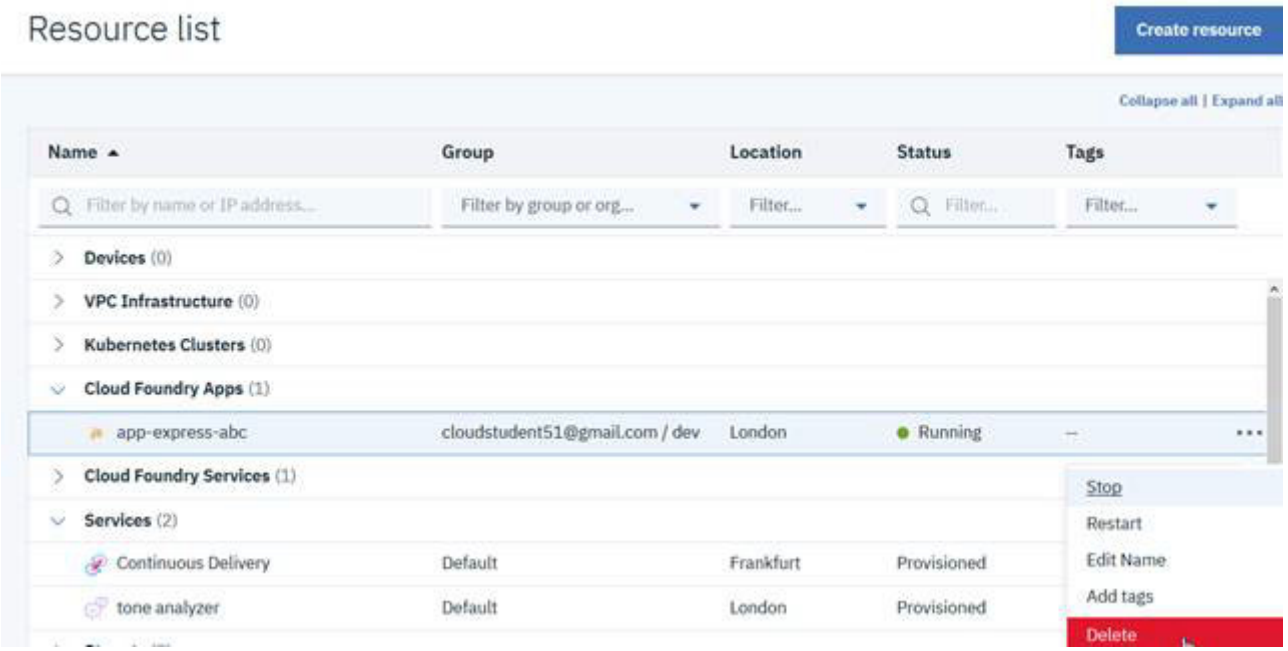
__ 3. Click **Actions** (the three dots icon) for your React application and select **Delete**, as shown in the following figure.



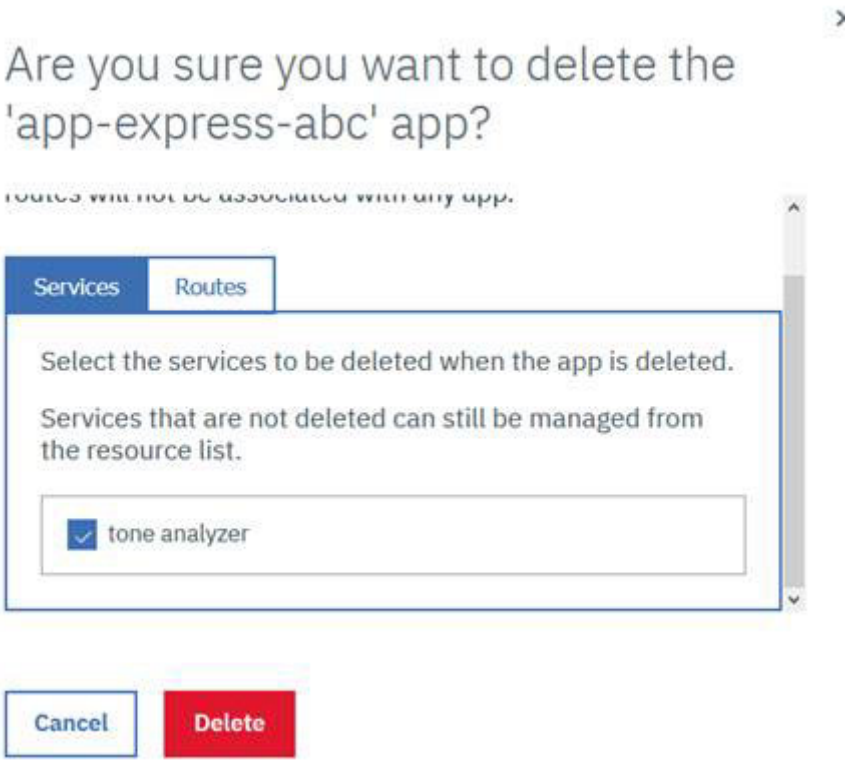
__ 4. Select the routes to delete and click **Delete**, as shown in the following figure.



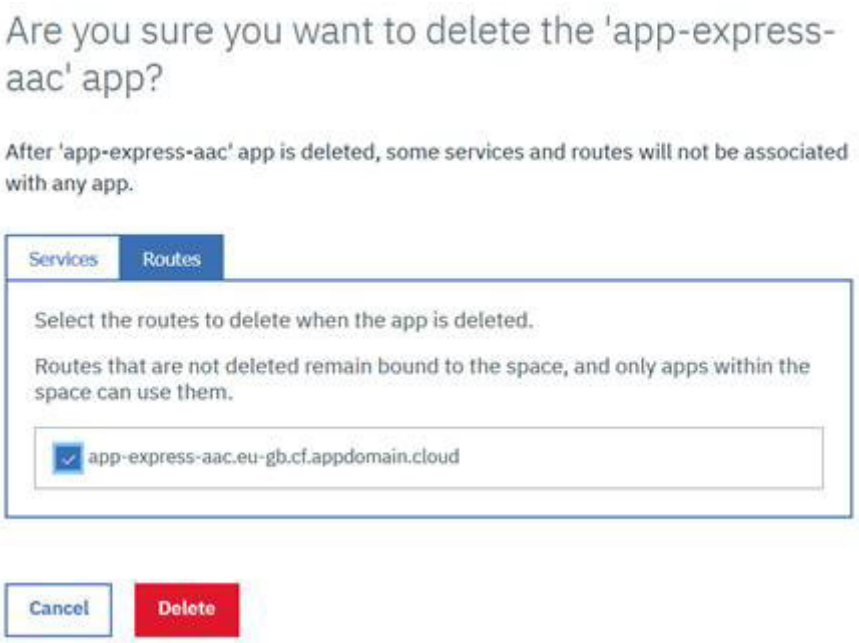
__ 5. Repeat the previous steps to delete your Node.js application.



__ 6. Select **tone analyzer** in the Services tab.



__ 7. Select the route to delete in the **Routes** tab.



__ 8. Click **Delete**.

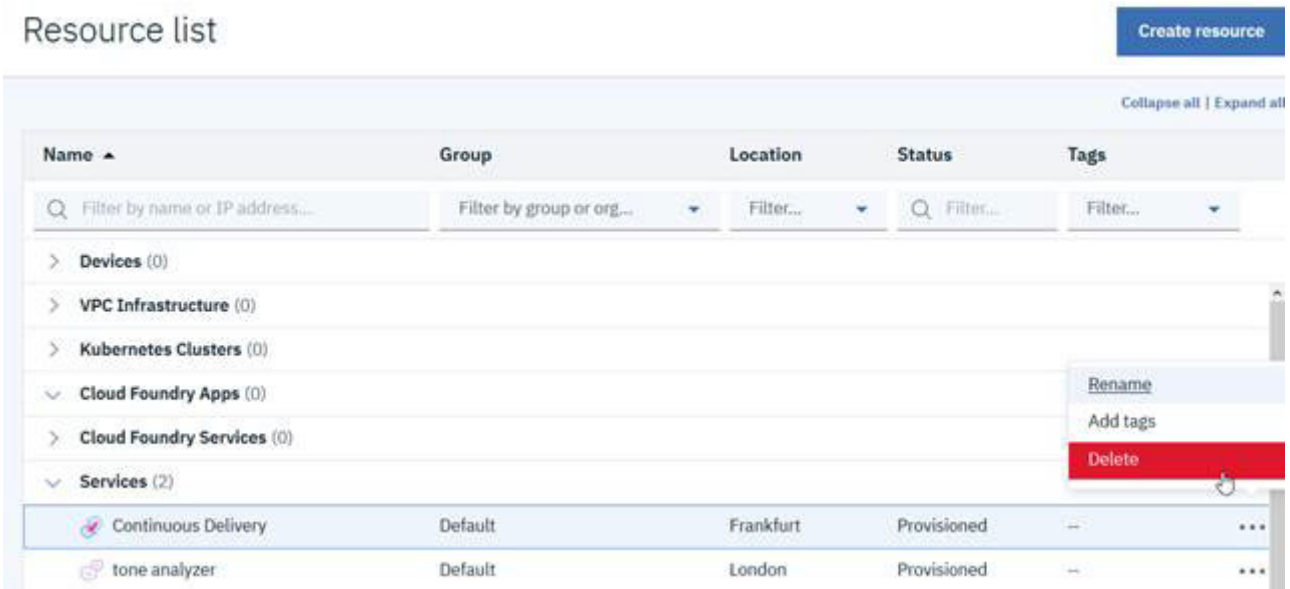
Deleting the Continuous Delivery and Tone Analyzer services

In this section, you delete the Continuous Delivery and Tone Analyzer services.

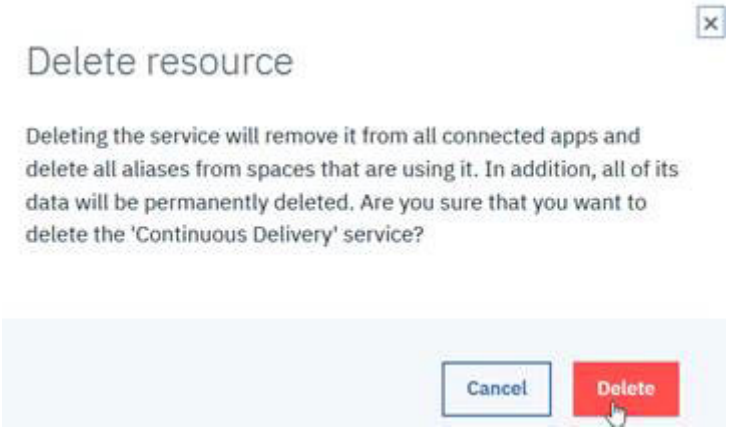
Complete the following steps:

__ 1. Expand **Services**.

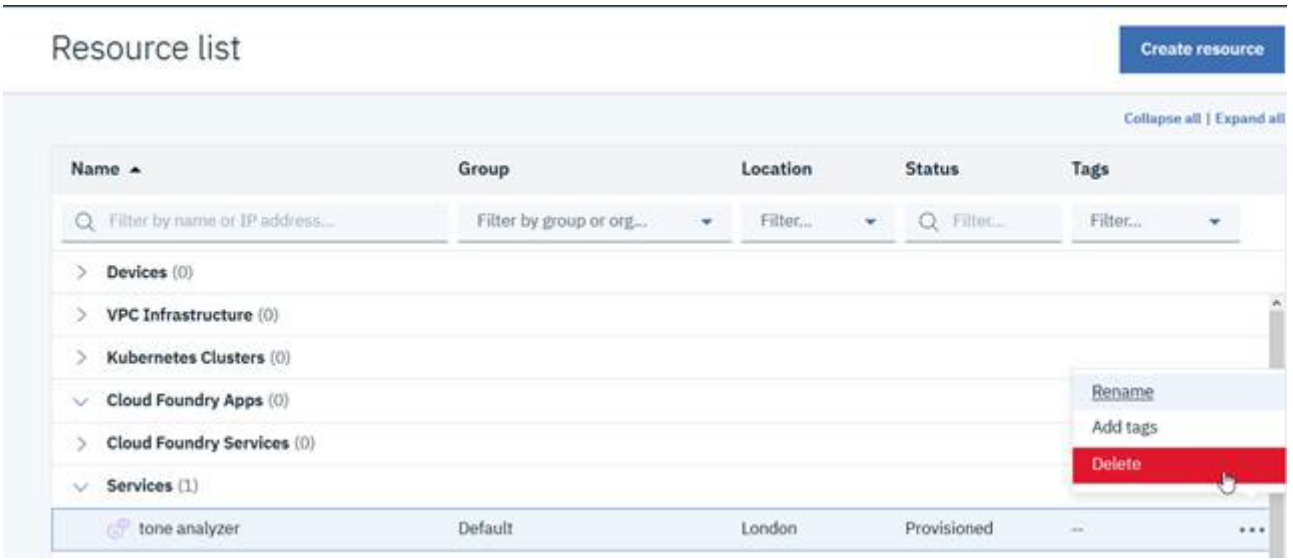
__ 2. Click **Actions** (the three dots icon) for your Continuous Delivery service and select **Delete**, as shown in the following figure.

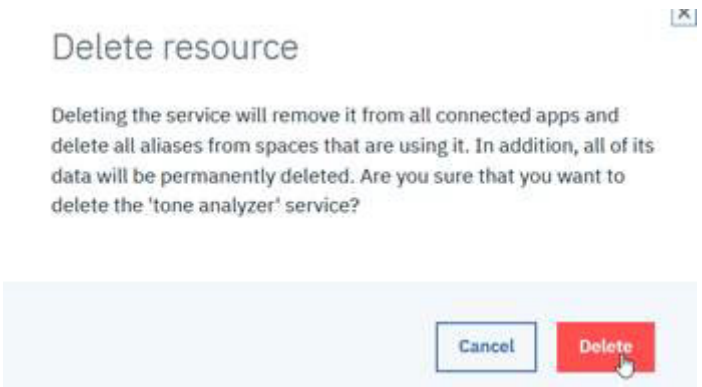


__ 3. At the Delete resource window, click **Delete**.



__ 4. Repeat the steps to delete the **tone analyzer** service.





Exercise review and wrap-up

During this exercise, you achieved the following goals:

- Learned the basics of the React framework when you explored a basic React application that uses React components to interact with the user.
- Learned the basics of ES8 when you explored some features of ES8 in the Node.js application.
- Learned how to serve static files to serve the front-end files and deploy a React application.
- Used Git to clone an existing project. You used Git to clone the source code from GitHub and used it as the basis for this exercise.

References

https://reactjs.org/docs/getting-started.html

https://www.ecma-international.org/publications/standards/Ecma-262.htm

Troubleshooting

The following path contains all the code for this exercise:

https://github.com/IBM-SkillsAcademy/Cloud-Application-Developer/tree/master/NodeApp/Ex4

IBM Training