Locally Debug a Serverless Application from an IDE

Using cloud9 , Lambda, API gateway.

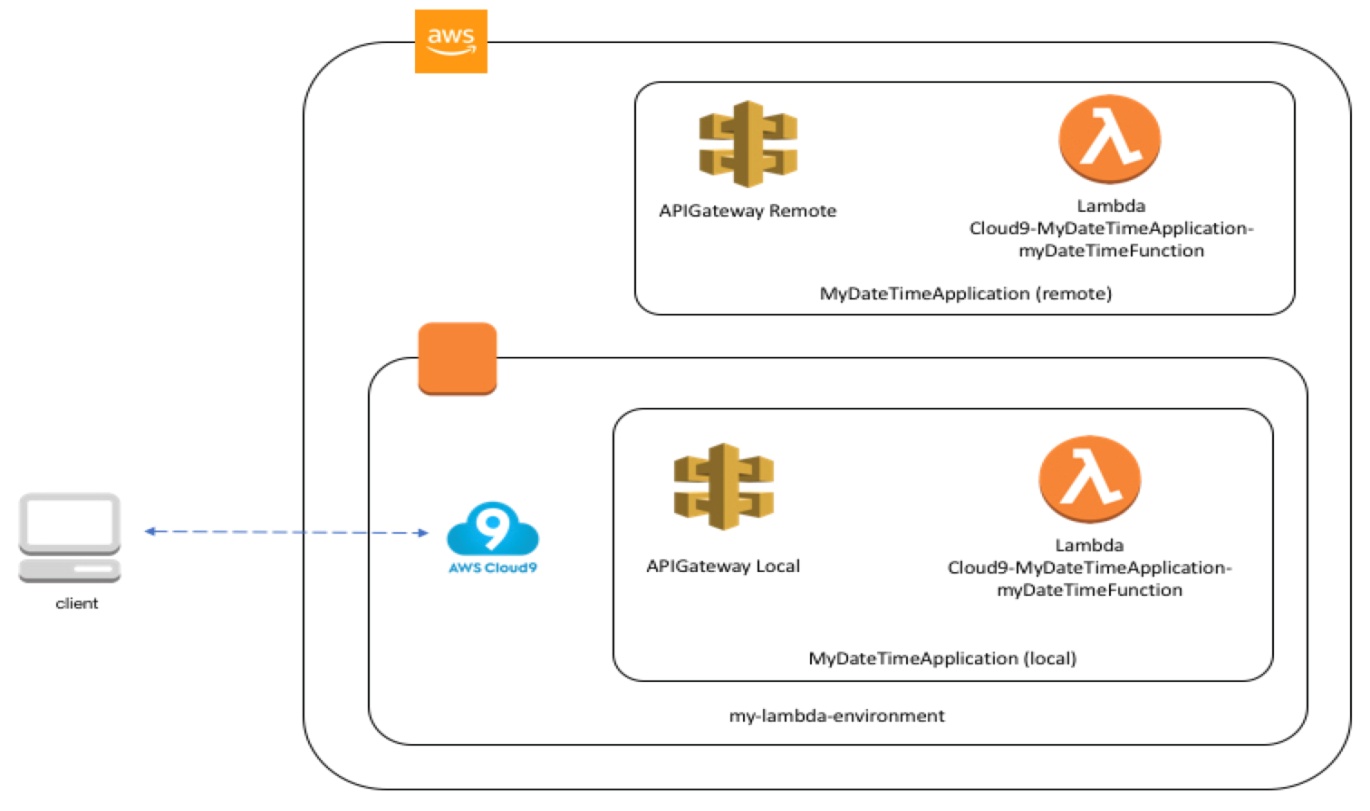
***SUBMITTED BY:***

Maniprasad Mishra

Regd. No.- 170301120036

Application Architecture

This application development project we use [AWS Cloud9](https://aws.amazon.com/cloud9/), [AWS Lambda](https://aws.amazon.com/lambda/), [Amazon API Gateway](https://aws.amazon.com/apigateway/) as pictured below:



# Step 1: Create and Open the Environment

In this step, you use the AWS Cloud9 console to create and then open an AWS Cloud9 development environment.

1. Sign in to the AWS Cloud9 console as follows:
   * If you're the only individual using your AWS account or you are an IAM user in a single AWS account, go to <https://console.aws.amazon.com/cloud9/>.
   * If your organization uses AWS Single Sign-On (AWS SSO), see your AWS account administrator for sign-in instructions.
   * If you're using an AWS Educate Starter Account, see [Step 2: Sign In to the AWS Cloud9 Console](https://docs.aws.amazon.com/cloud9/latest/user-guide/setup-student.html#setup-student-sign-in-ide) in Individual Student Signup.
   * If you're a student in a classroom, see your instructor for sign-in instructions.
2. After you sign in to the AWS Cloud9 console, in the top navigation bar, choose an AWS Region to create the environment in. For a list of available AWS Regions, see [AWS Cloud9](https://docs.aws.amazon.com/general/latest/gr/rande.html#cloud9_region) in the AWS General Reference.

Fig:1

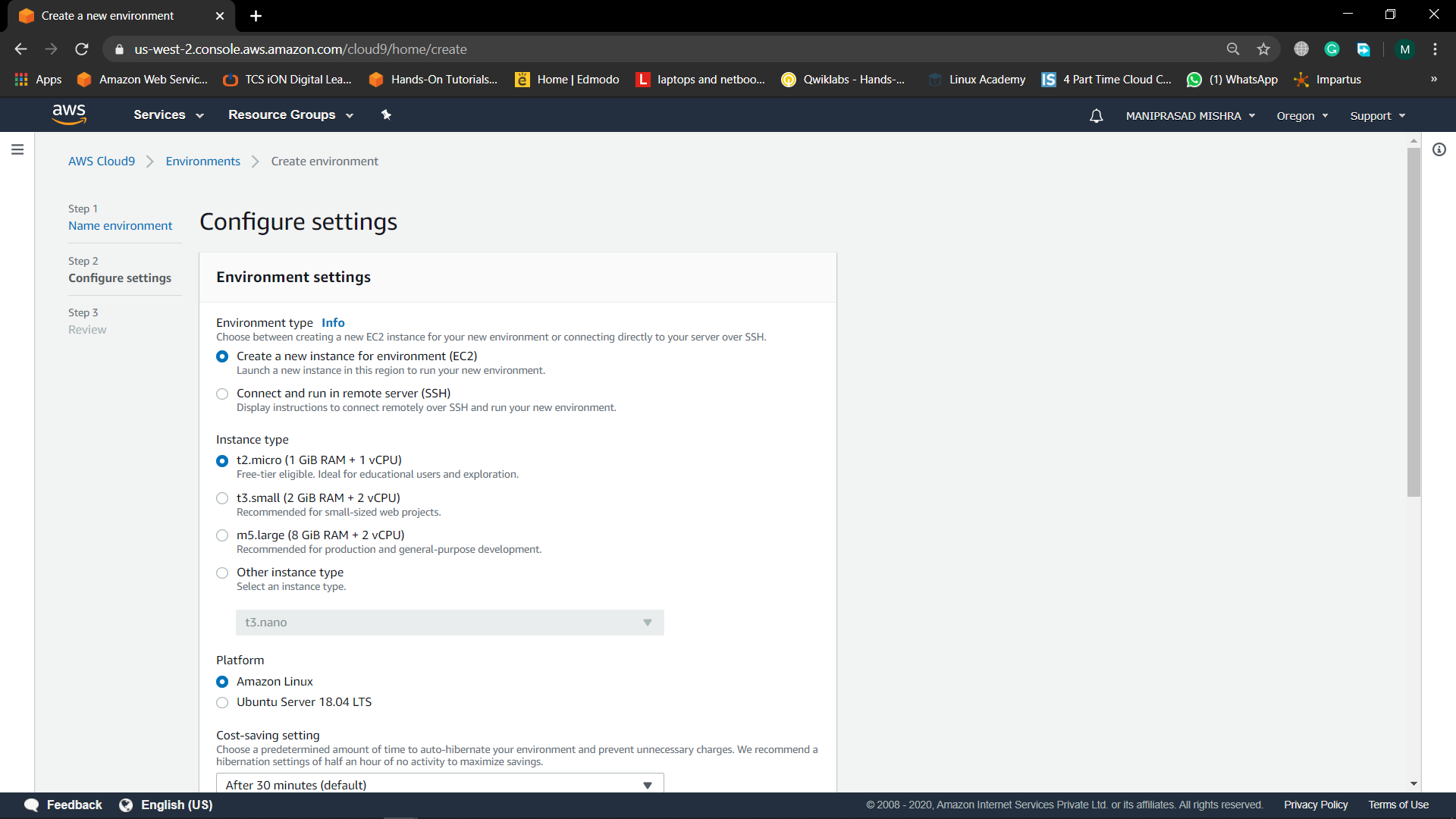


Fig:2

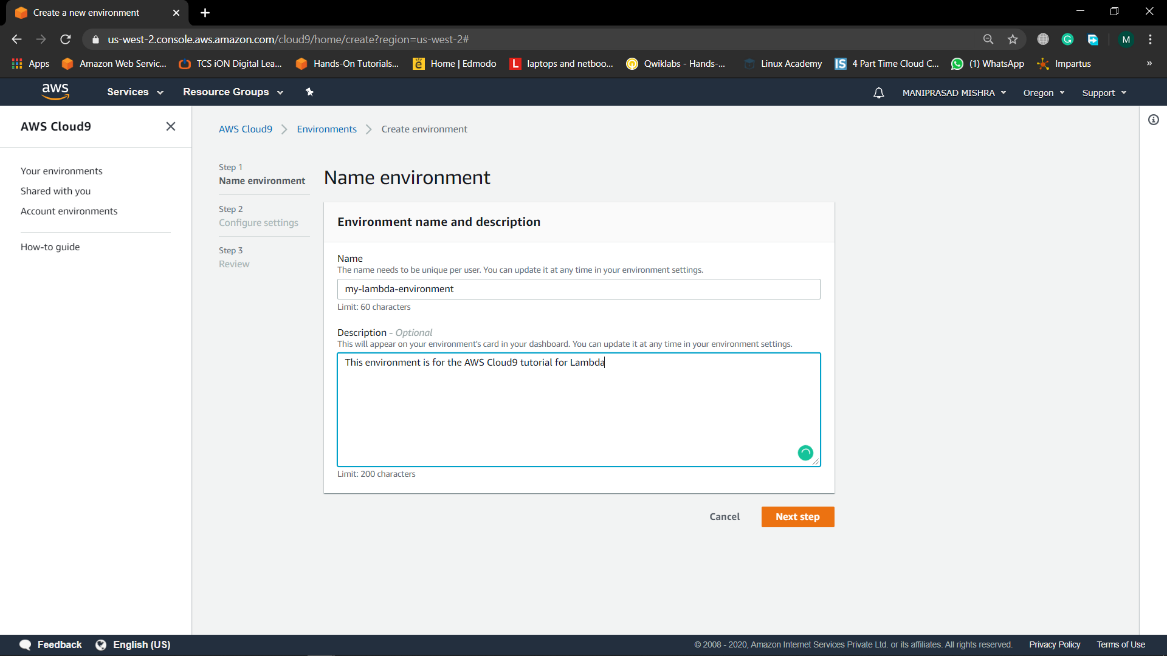
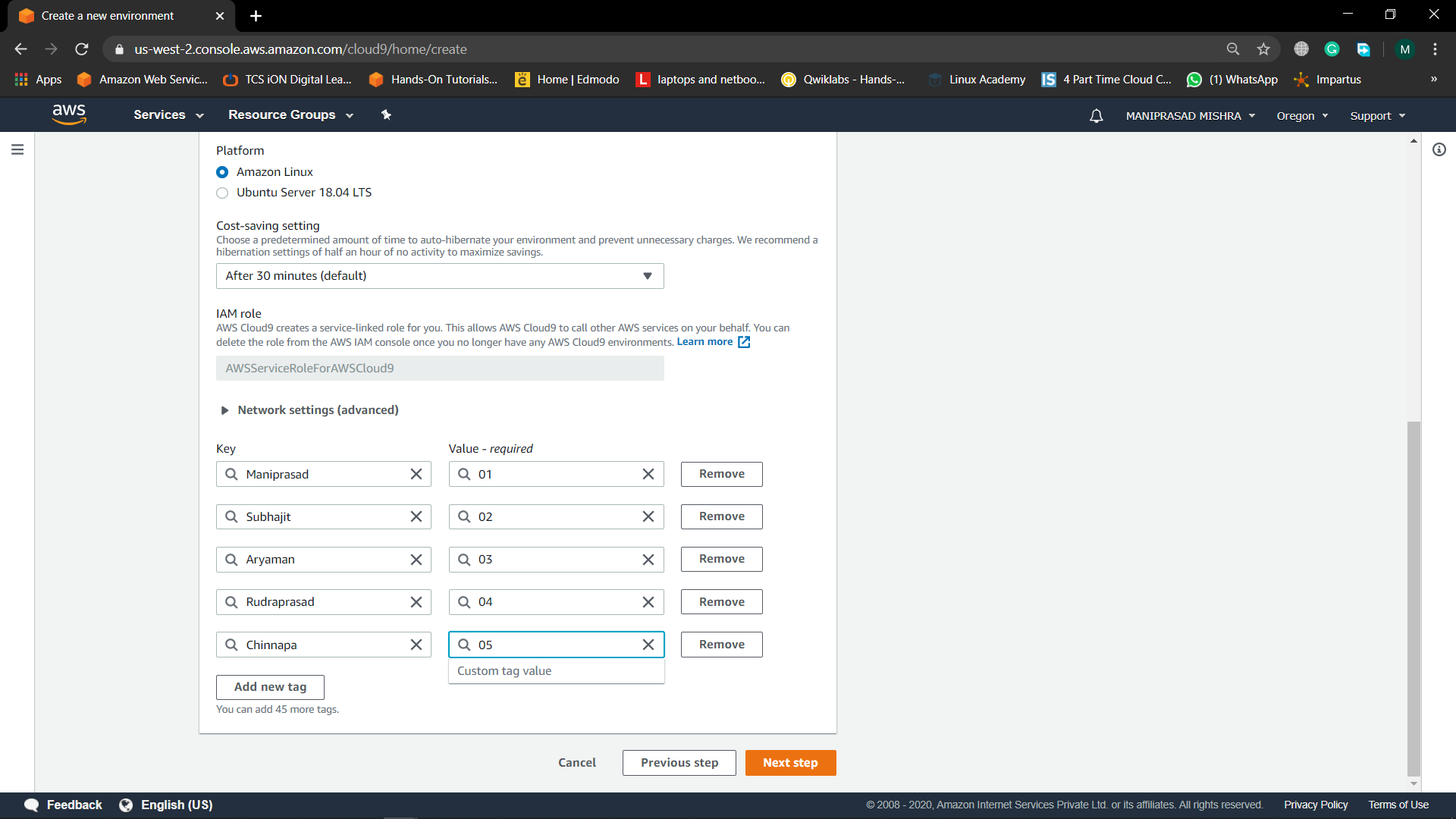


Fig:3



Choose the large **Create environment** button in one of the locations shown below.

If you have no AWS Cloud9 environments yet, the button is shown on a welcome page.

Fig:4


                Welcome page in the AWS Cloud9 console
            

If you already have AWS Cloud9 environments, the button is shown as follows

Fig:5


                Create environment button in the AWS Cloud9 console
            

1. On the **Name environment** page, for **Name**, enter a name for your environment. For this tutorial, use my-lambda-environment.
2. For **Description**, enter something about your environment. For this tutorial, use This environment is for the AWS Cloud9 tutorial for Lambda.
3. Choose **Next step**.
4. On the **Configure settings** page, for **Environment type**, choose **Create a new instance for environment (EC2)**.

**Warning**

Choosing **Create a new instance for environment (EC2)** might result in possible charges to your AWS account for Amazon EC2.

1. For **Instance type**, leave the default choice. This choice has relatively low RAM and vCPUs, which is sufficient for this tutorial.

**Warning**

Choosing instance types with more RAM and vCPUs might result in additional charges to your AWS account for Amazon EC2.

1. For **Platform**, choose the type of Amazon EC2 instance that you want: **Amazon Linux** or **Ubuntu**. AWS Cloud9 creates the instance and then connects the environment to it.
2. Choose a value for **Cost-saving setting**. When all web browser instances that are connected to the IDE for the environment are closed, AWS Cloud9 waits this amount of time and then shuts down the Amazon EC2 instance for the environment.

**Warning**

Choosing a longer time period might result in more charges to your AWS account.

1. Add up to 50 tags by supplying a **Key** and a **Value** for each tag. The tags will be attached to the AWS Cloud9 environment as resource tags, and are propagated to the following underlying resources: the AWS CloudFormation stack, the Amazon EC2 instance, and Amazon EC2 security groups. You can find information about tags in [Control Access Using AWS Resource Tags](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_tags.html) in the [*IAM User Guide*](https://docs.aws.amazon.com/IAM/latest/UserGuide/). Also see the [advanced information](https://docs.aws.amazon.com/cloud9/latest/user-guide/tags.html) about tags.

**Warning**

If you update these tags after you create them, the changes are NOT automatically propagated to the underlying resources. For more information, see [Propagating Tag Updates to Underlying Resources](https://docs.aws.amazon.com/cloud9/latest/user-guide/tags.html#tags-propagate) in the advanced information about [tags](https://docs.aws.amazon.com/cloud9/latest/user-guide/tags.html).

1. Choose **Next step**.
2. On the **Review** page, choose **Create environment**. Wait while AWS Cloud9 creates your environment. This can take several minutes.

# Step 2: Create the Lambda Function and API

In this step, you use the AWS Cloud9 IDE to create the Lambda function and its accompanying API at the same time. AWS Cloud9 stores the new function on the Amazon EC2 instance and deploys a copy of the function to Lambda. AWS Cloud9 also stores the new accompanying API on the instance and deploys a copy of the API to API Gateway.

The Lambda function returns information about the day or time you specify, for example, the day, month, and year, or the current hour, minute, and second.

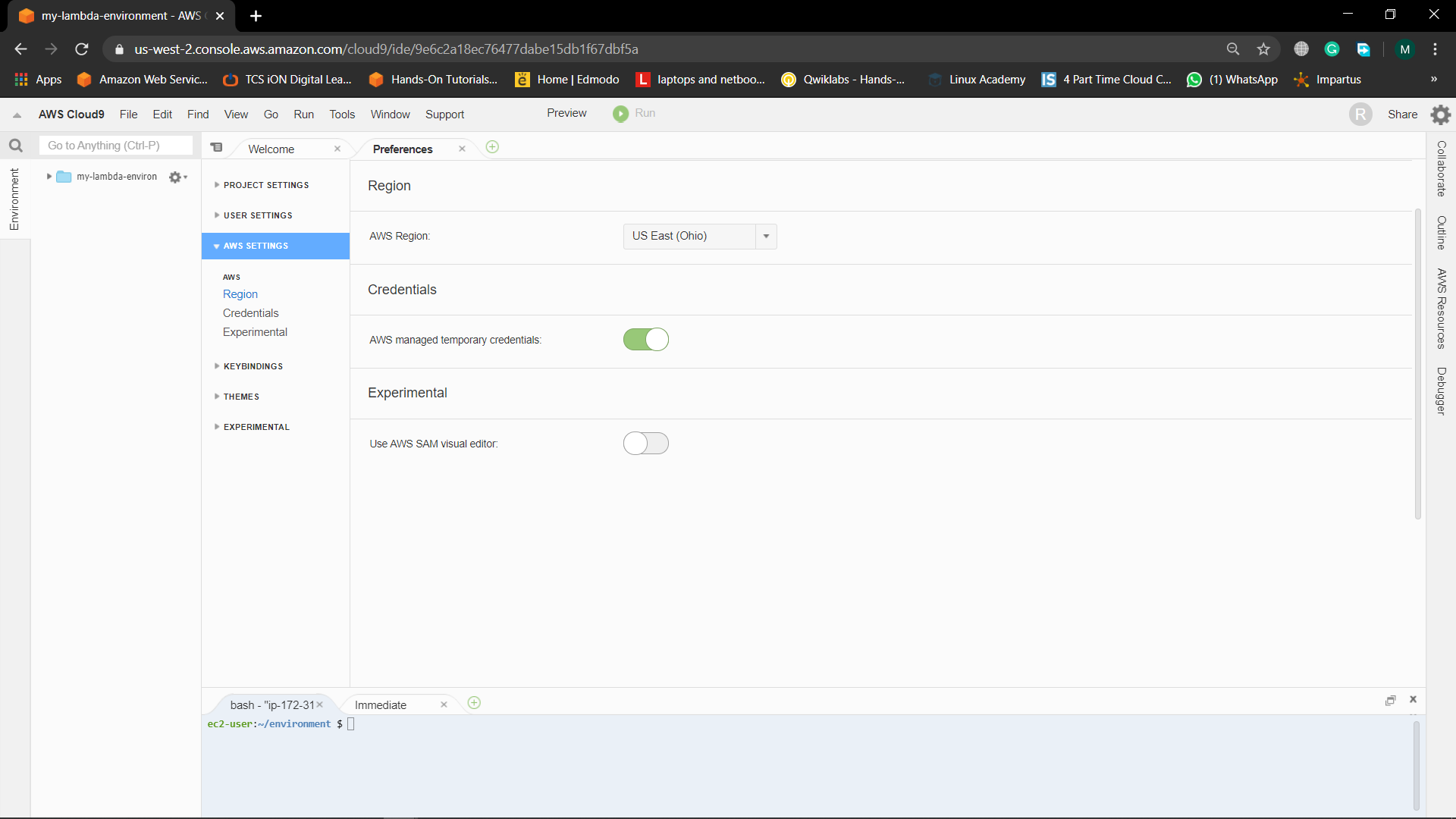
Currently, you can use the IDE to automatically create functions that use only Node.js or Python. This function uses Node.js.

1. Make sure you have completed the prerequisites for work with Lambda functions in the AWS Cloud9 IDE for the environment. This includes checking to see whether you must:
   * Set up your IAM group with required access permissions.
   * Set up your environment with AWS access credentials.
   * Create an execution rule for your Lambda functions.

For more information, see [Prepare to Work with Lambda Functions](https://docs.aws.amazon.com/cloud9/latest/user-guide/lambda-functions.html#lambda-functions-prepare).

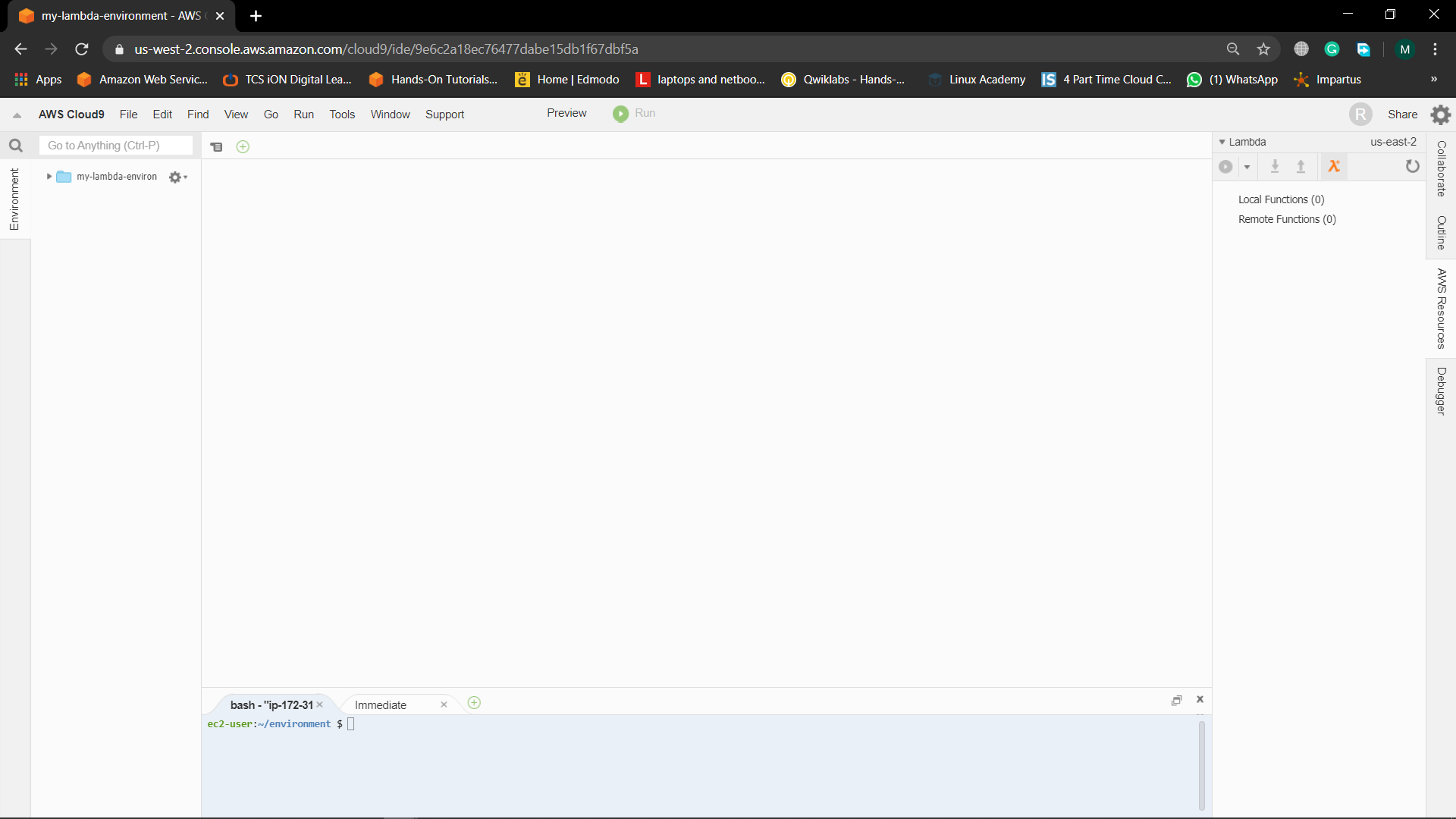
1. With the IDE open from the previous step, set the AWS Region that you want to create the function in. To do this, on the menu bar, choose **AWS Cloud9, Preferences**.
2. In the navigation pane of the **Preferences** tab, choose **AWS Settings**.
3. For **AWS Region**, select the AWS Region you want to create the function in.

Fig:6



1. On the edge of the IDE, choose **AWS Resources**.

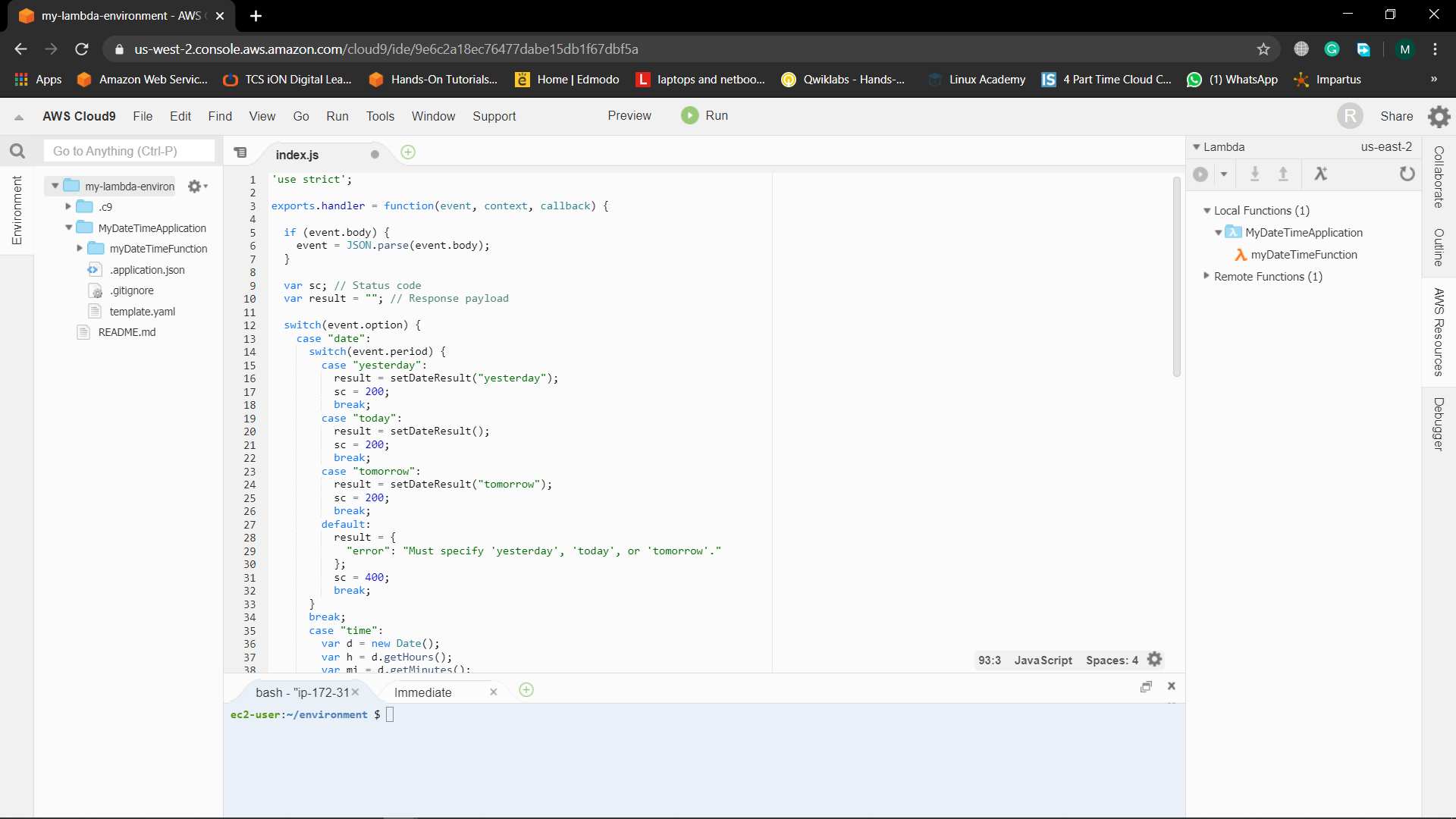
Fig:7



1. Expand the **Lambda**, if it isn't already expanded. On the toolbar, choose the **Create a new Lambda function** icon.
2. In the **Create serverless application** dialog box, for **Function name**, type a name for the function (for example, myDateTimeFunction).
3. For **Application name**, type a name for the function's containing serverless application (for example, MyDateTimeApplication).
4. Choose **Next**.
5. For **Select runtime**, choose **Node.js 6.10**.
6. For **Select blueprint**, choose **empty-nodejs**. This creates some starter code that you work with in the next step.
7. Choose **Next**.
8. For **Function trigger**, choose **API Gateway**. This creates an API in API Gateway that you use to run and debug the function in a later step.
9. For **Resource Path**, type / (a forward slash). The **Resource Path** defines a portion of the URL that you use to run and debug the API in a later step. For more information, see [Set up API Methods in API Gateway](https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-method-settings.html) in the *API Gateway Developer Guide*.
10. For **Security**, choose **NONE**, and then choose **Next**. This specifies that you don't need any special credentials to run and debug this API.
11. For **Memory (MB)**, leave the default value of **128 MB**, which is sufficient for this tutorial.
12. For **Role**, leave the default value of **Automatically generate role**, and then choose **Next**. This specifies the access permissions that the function needs to work properly.
13. Review your choices, and then choose **Finish**.

AWS Cloud9 creates the function and its related API on the instance. Then it deploys a copy of the function and API to Lambda and API Gateway. The serverless application and function are displayed in the **Local Functions** and **Remote Functions** lists in the **Lambda** pane of the **AWS Resources** window. The serverless application's and function's component files are displayed in the **Environment** window. The editor opens the function's code file, index.js.

Fig:8



**Note**

In addition to the function's code file, AWS Cloud9 creates the following helper files.

* .application.json: A hidden file that contains JSON-formatted settings that are specific to the serverless application. AWS Cloud9 uses these settings for its internal use. Do not edit this file.
* .gitignore: A hidden file that contains a list of files Git ignores, if you want to use Git to manage your source code for this function.
* template.yaml: An AWS Serverless Application Model (AWS SAM) template file that contains information about the Lambda function, the related API in API Gateway, and any other related, supported AWS resources. Whenever you update the local version of your function and then upload it to Lambda, AWS Cloud9 calls AWS SAM to use this template file to do the upload. For more information, see [Using the AWS Serverless Application Model (AWS SAM)](https://docs.aws.amazon.com/lambda/latest/dg/deploying-lambda-apps.html#serverless_app) in the *AWS Lambda Developer Guide*.

The .application.json and .gitignore files are hidden. To show hidden files or hide them if they're shown, in the **Environment** window, choose the gear icon. Then choose **Show Hidden Files**.

Fig:9

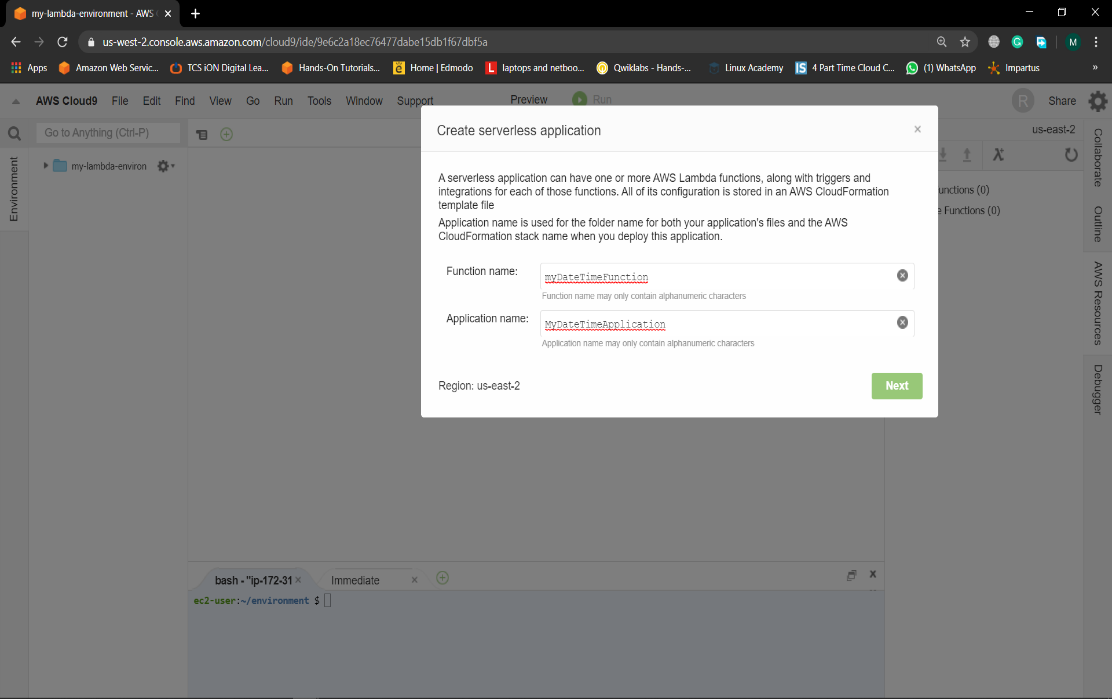


Fig:10

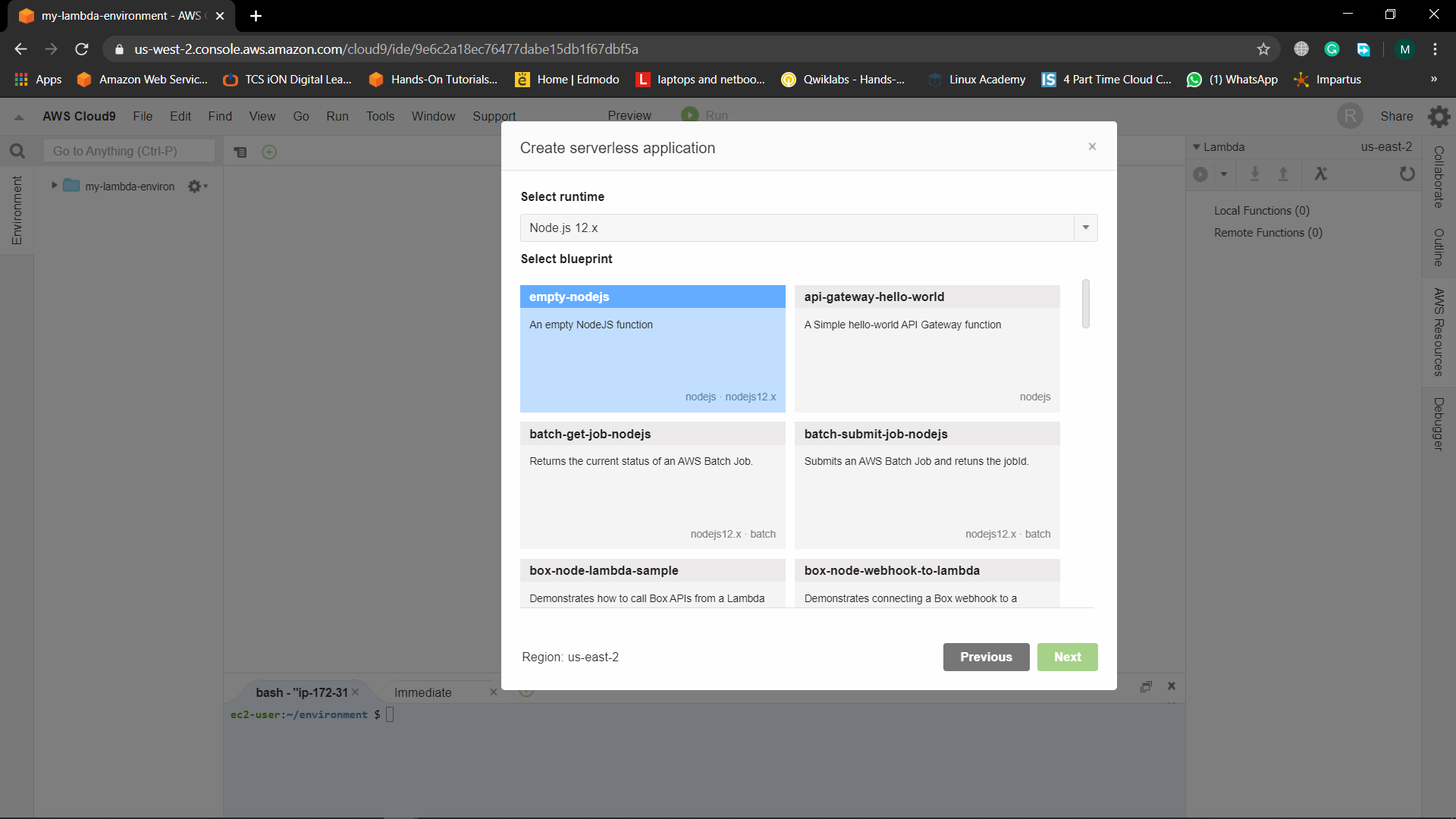
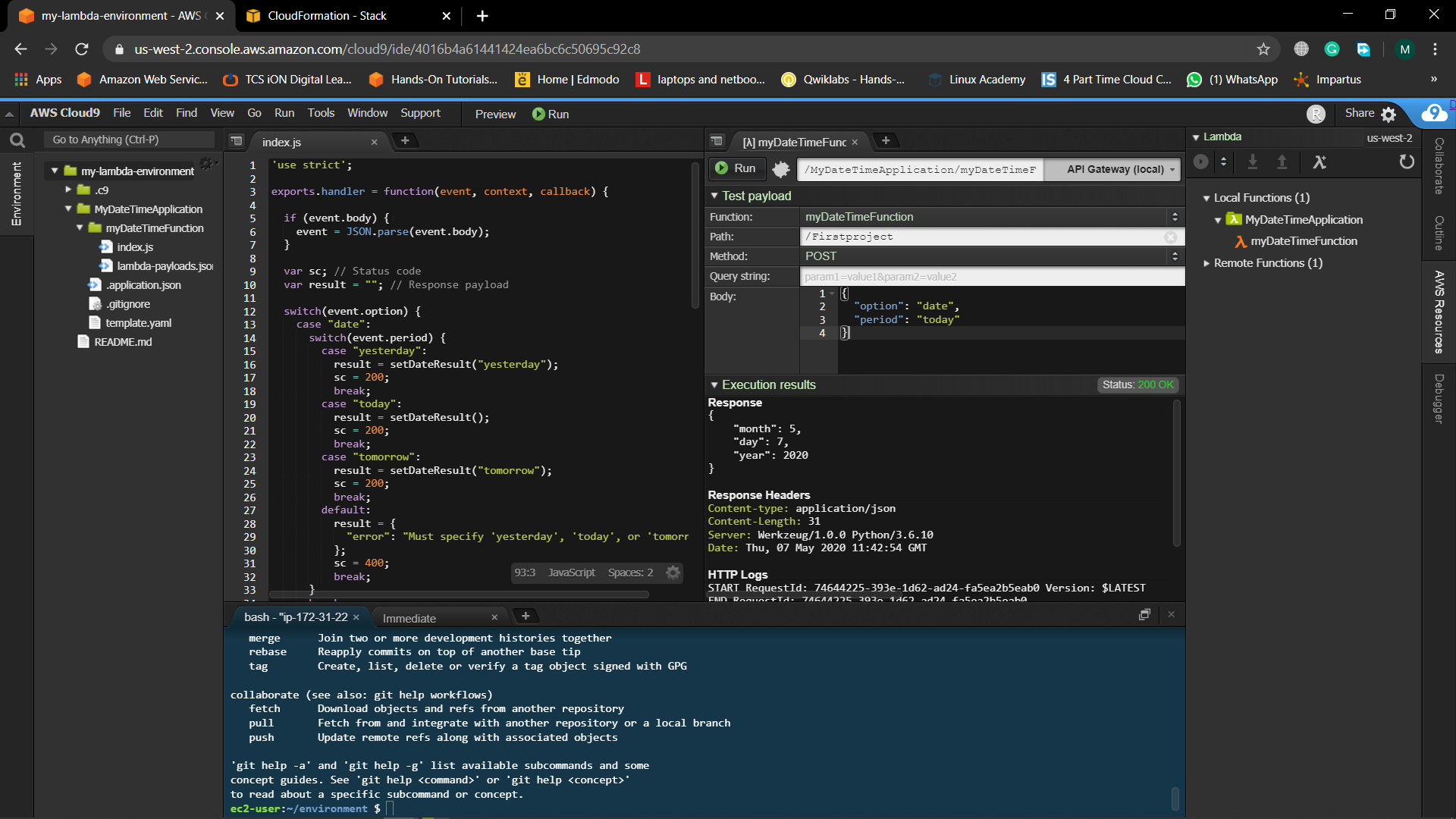


Fig:11



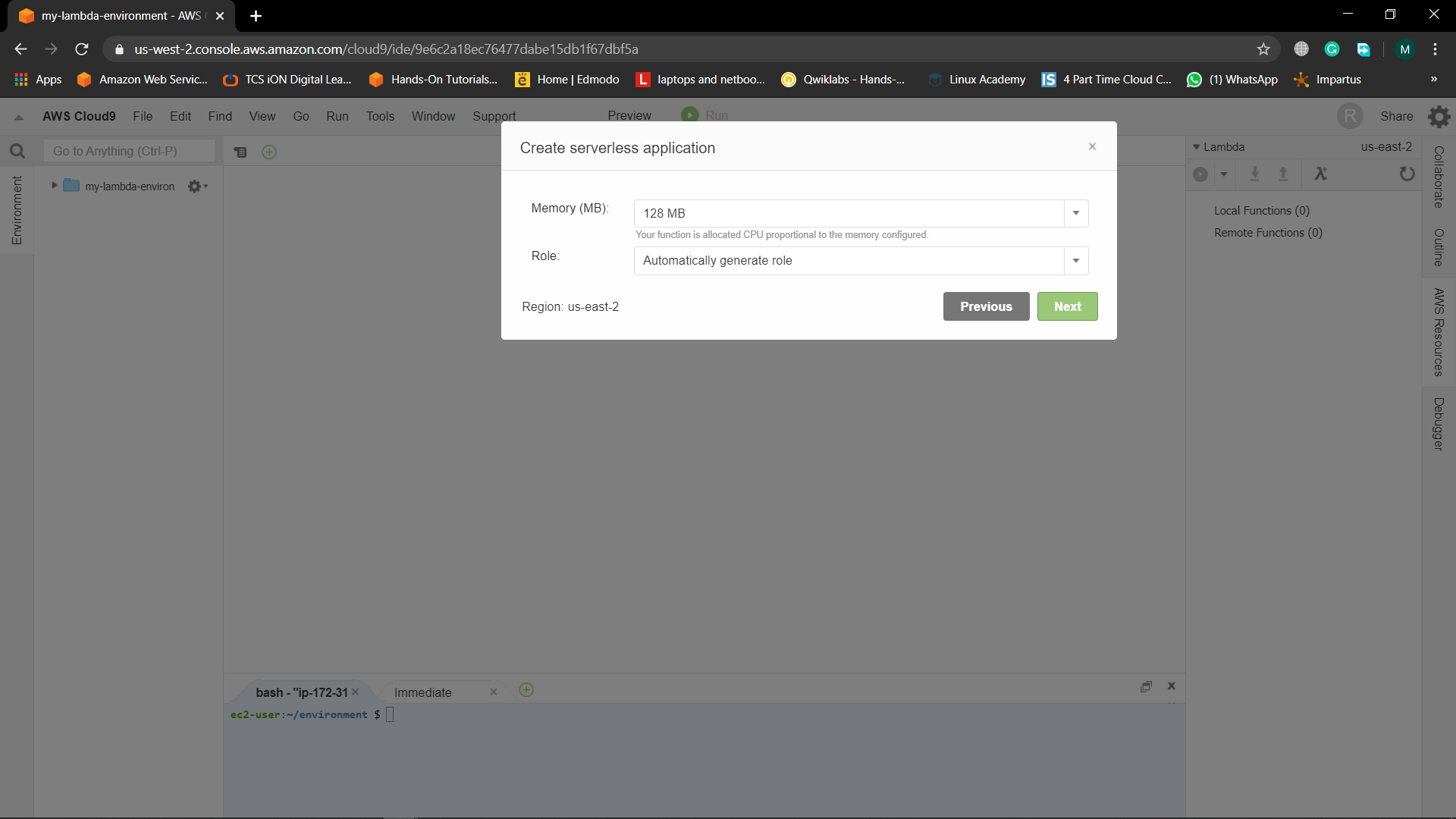


Fig:12

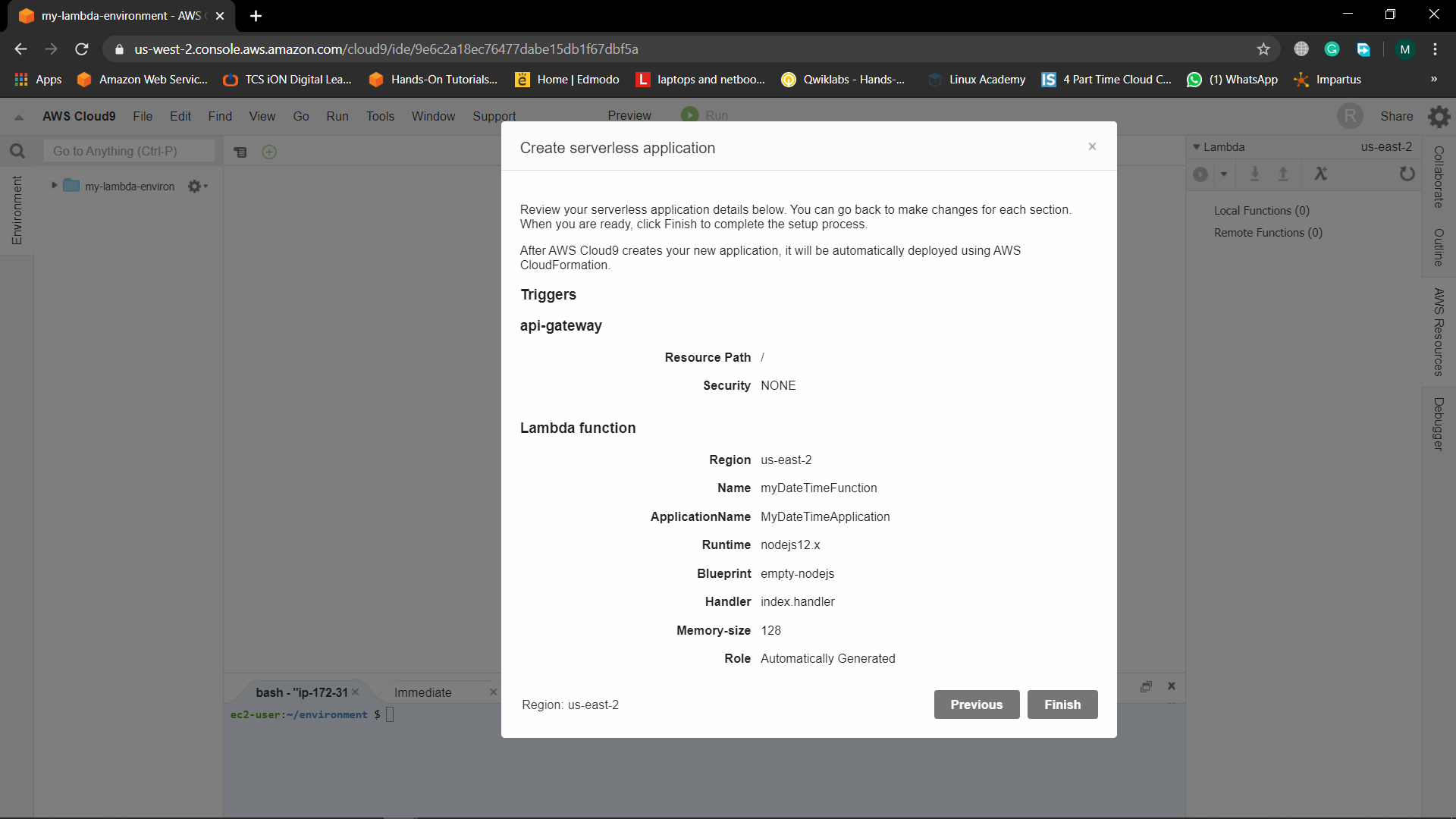


Fig:13

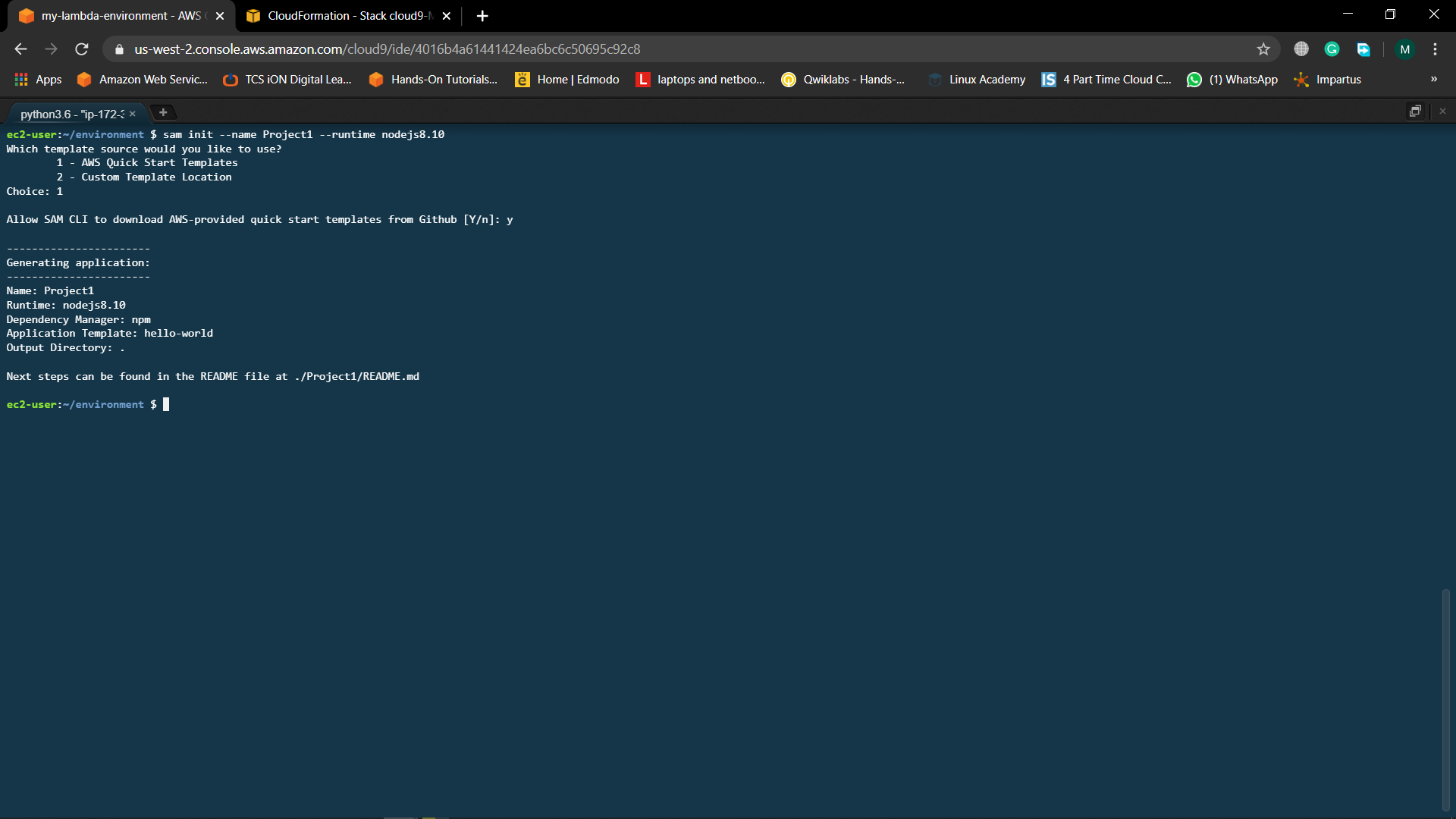


Fig:14


               Showing the hidden Lambda files
            

# Step 3: Add Code to the Function

In this step, you replace the starter code for the generated function with specific code that returns information about the day or time you specify.

With the index.js file already open in the editor, completely replace the file's contents with the following code, and then save the file.

'use strict';

exports.handler = function(event, context, callback) {

if (event.body) {

event = JSON.parse(event.body);

}

var sc; *// Status code*

var result = ""; *// Response payload*

switch(event.option) {

case "date":

switch(event.period) {

case "yesterday":

result = setDateResult("yesterday");

sc = 200;

break;

case "today":

result = setDateResult();

sc = 200;

break;

case "tomorrow":

result = setDateResult("tomorrow");

sc = 200;

break;

default:

result = {

"error": "Must specify 'yesterday', 'today', or 'tomorrow'."

};

sc = 400;

break;

}

break;

case "time":

var d = new Date();

var h = d.getHours();

var mi = d.getMinutes();

var s = d.getSeconds();

result = {

"hour": h,

"minute": mi,

"second": s

};

sc = 200;

break;

default:

result = {

"error": "Must specify 'date' or 'time'."

};

sc = 400;

break;

}

const response = {

statusCode: sc,

headers: { "Content-type": "application/json" },

body: JSON.stringify( result )

};

callback(null, response);

function setDateResult(option) {

var d = new Date(); *// Today*

var mo; *// Month*

var da; *// Day*

var y; *// Year*

switch(option) {

case "yesterday":

d.setDate(d.getDate() - 1);

break;

case "tomorrow":

d.setDate(d.getDate() + 1);

default:

break;

}

mo = d.getMonth() + 1;

da = d.getDate();

y = d.getFullYear();

result = {

"month": mo,

"day": da,

"year": y

};

return result;

}

};

This function takes an incoming payload with an option value of date or time. If date is specified, you must also specify a period value of yesterday, today, or tomorrow. The function then returns the corresponding month, day, and year. If, however, an option value of time is specified, the function returns the current hour, minute, and second.

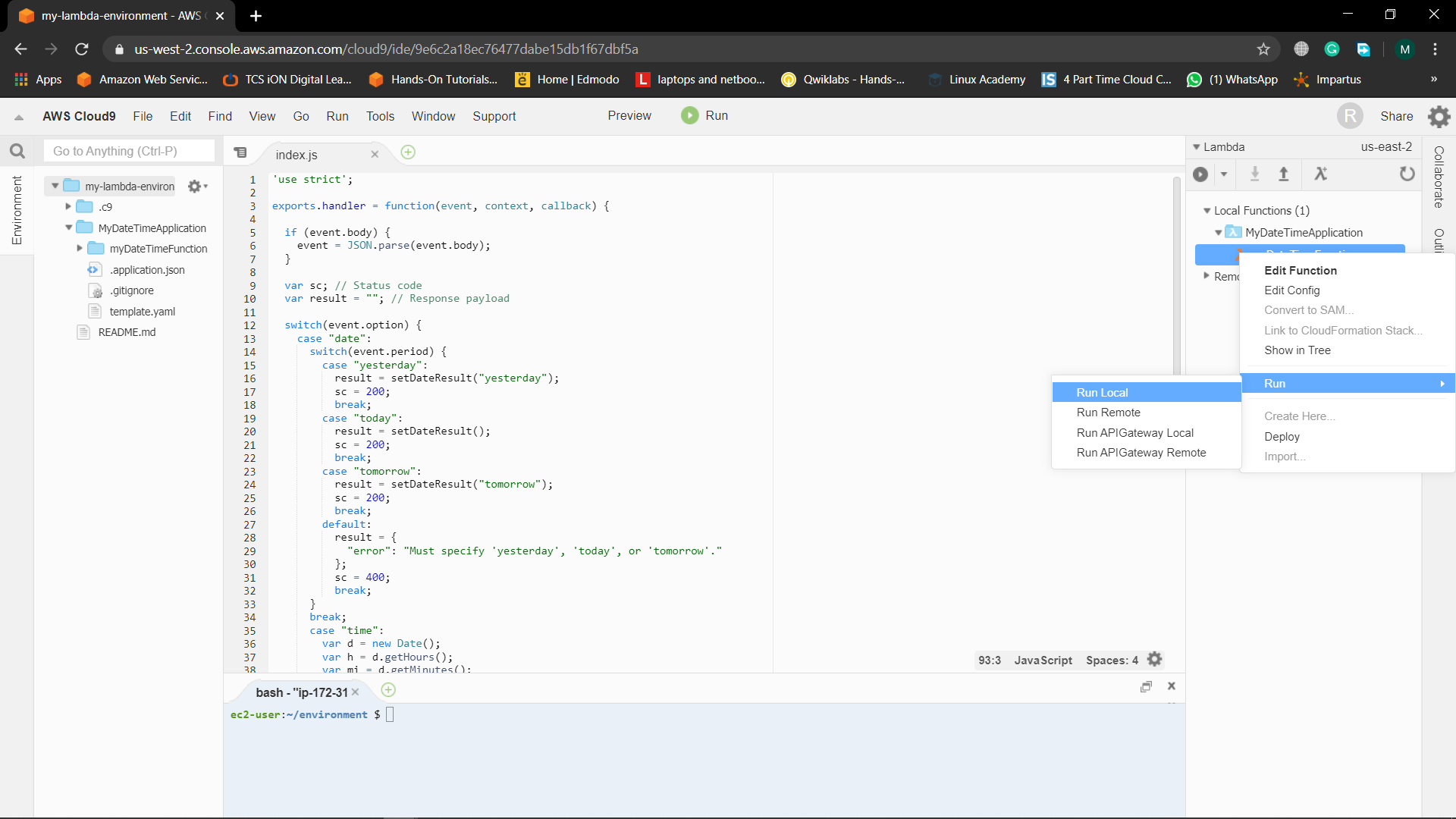
# Step 4: Run or Debug the Function Locally

**Step 4.1: Run the Function Locally**

In this step, you use the IDE to run the function on the instance. Currently, you can use the IDE to run functions that use only Node.js or Python.

1. In the **Lambda** pane of the **AWS Resources** window, expand the **MyDateTimeApplication** Lambda folder, and then right-click the **myDateTimeFunction** Lambda icon.
2. Choose **Run, Run Local**.

Fig:15



1. In the **Payload** pane of the run tab, replace {} with the following, which sends this data as input to the function for processing.
2. {
3. "option": "date",
4. "period": "today"

}

1. Choose **Run**.
2. The **Response** pane displays the following.
3. {
4. "statusCode": 200,
5. "headers": {
6. "Content-type": "application/json"
7. },
8. "body": "{\"month\":4,\"day\":12,\"year\":2018}"

}

**Note**

If the **Response** pane displays null, be sure to save the index.js, and then choose **Run** again.

Compare your results to the following.

Fig:16


                     Running the Lambda function
                  

1. Run the function several times with the following payloads to see what the **Response** pane displays.
2. {
3. "option": "date",
4. "period": "yesterday"
5. }
6. {
7. "option": "date",
8. "period": "tomorrow"
9. }
10. {
11. "option": "time"
12. }
13. Displays an error. Must also specify a 'period' of 'yesterday', 'today', or 'tomorrow'.
14. {
15. "option": "date"
16. }
17. Displays an error. Must specify an 'option' of 'date' or 'time'.
18. {
19. "option": "dates"

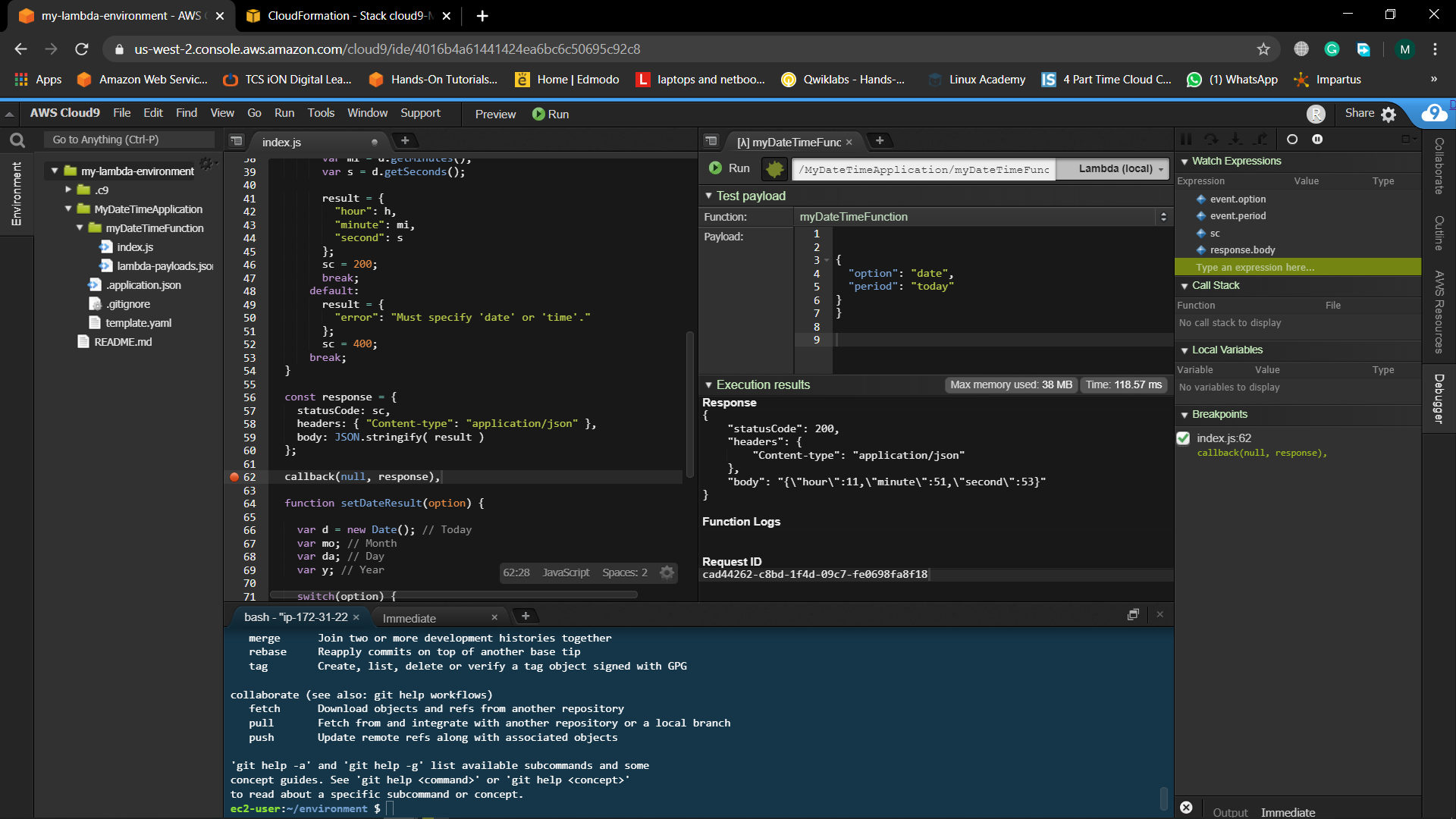
}

**Step 4.2: Debug the Function Locally**

In this step, you use the IDE to debug the function on the instance. Currently, you can use the IDE to debug functions that use only Node.js or Python. Also, you can use the IDE to debug functions locally only. You cannot use the IDE to debug functions in Lambda itself.

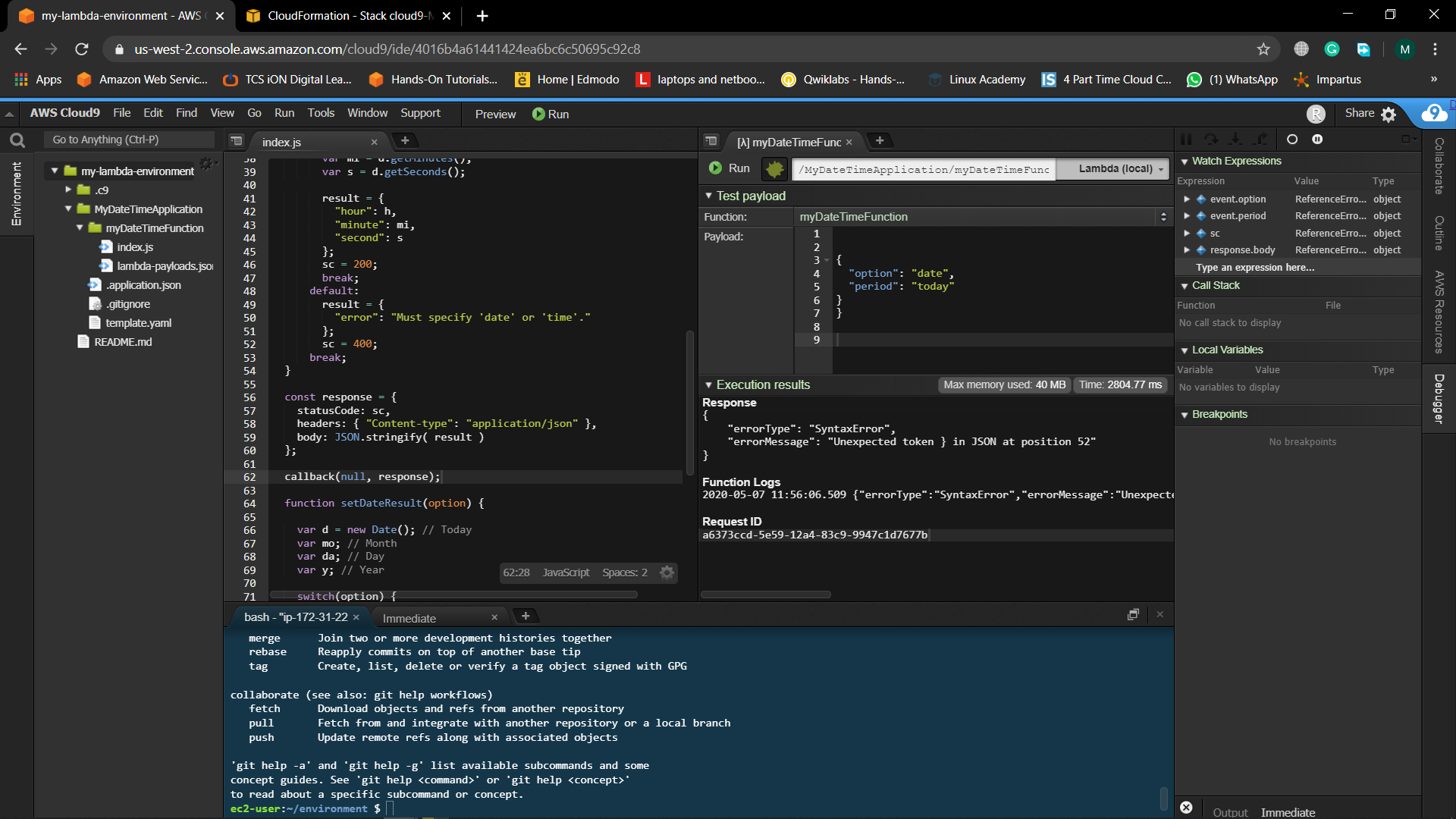
1. In the index.js file, create a breakpoint for the debugger. To do this, in the editor, next to the line of code callback(null, response), click the gutter just to the left of line 62. A red circle is displayed, representing the breakpoint.
2. On the right edge of the IDE, choose **Debugger**.
3. Add four expressions for the debugger to watch. To do this, in the **Watch Expressions** area, for **Type an expression here**, type event.option, and then press Enter. Do this three more times, typing event.period, sc, and response.body.

Fig:17



1. On the run tab that AWS Cloud9 opened in [Step 4.1: Run the Function Locally](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-or-debug-local-function.html#tutorial-lambda-run-local-function), choose the icon that looks like a bug. (It switches from grey to green.)

Fig:18




                     Turning on debugging.
                  

1. In the **Payload** pane on the run tab, be sure the following payload is still visible.
2. {
3. "option": "date",
4. "period": "today"

}

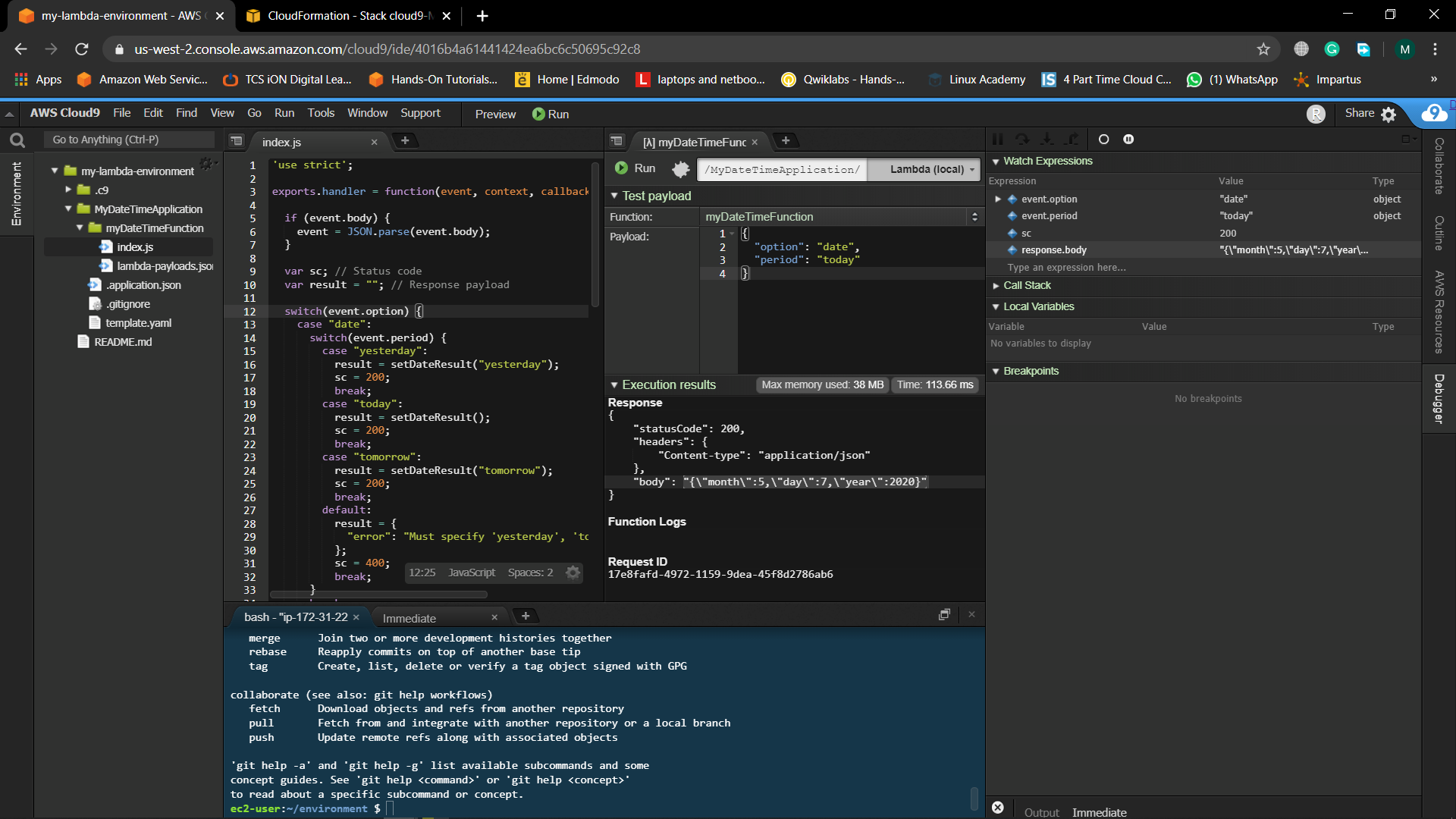
1. Choose **Run**.

Code execution pauses at the breakpoint. The current values of event.option, event.period, sc, and response.body are displayed in the **Watch Expressions** area of the **Debugger** window.

You can also see these values in the code by hovering your mouse over option in line 12 in the code, period in line 14, sc in line 21, and response in line 56. For response in line 56, expand **Object {}**, and then see the **body** value.

Compare your results to the following.

Fig:19



1. In the Debugger window, choose the blue **Resume** arrow to finish running the code. (It looks like a triangular play button.)

# Step 5: Run or Debug the API Locally

**Step 5.1: Run the API Locally**

In this step, you use the IDE to have API Gateway run the Lambda function on the instance.

1. In the **Lambda** pane of the **AWS Resources** window, right-click the **myDateTimeFunction** Lambda icon, and then choose **Run, Run APIGateway Local**.

**Note**

You can also do this by choosing the **Lambda (local)** list on the run tab from the previous step, and then choosing **API Gateway (local)**.

1. On the run tab, for **Path**, type / (a forward slash).
2. For **Method**, choose **POST**.
3. For **Body**, replace {} with the following, which sends this data as input to the API for processing.
4. {
5. "option": "date",
6. "period": "today"

}

1. If the bug icon is green, choose it to turn it off. (It switches back to grey.)


                     Turning off debugging
                  

1. Choose **Run**.
2. The **Response** pane displays the following response.
3. {
4. "month": 4,
5. "day": 12,
6. "year": 2018

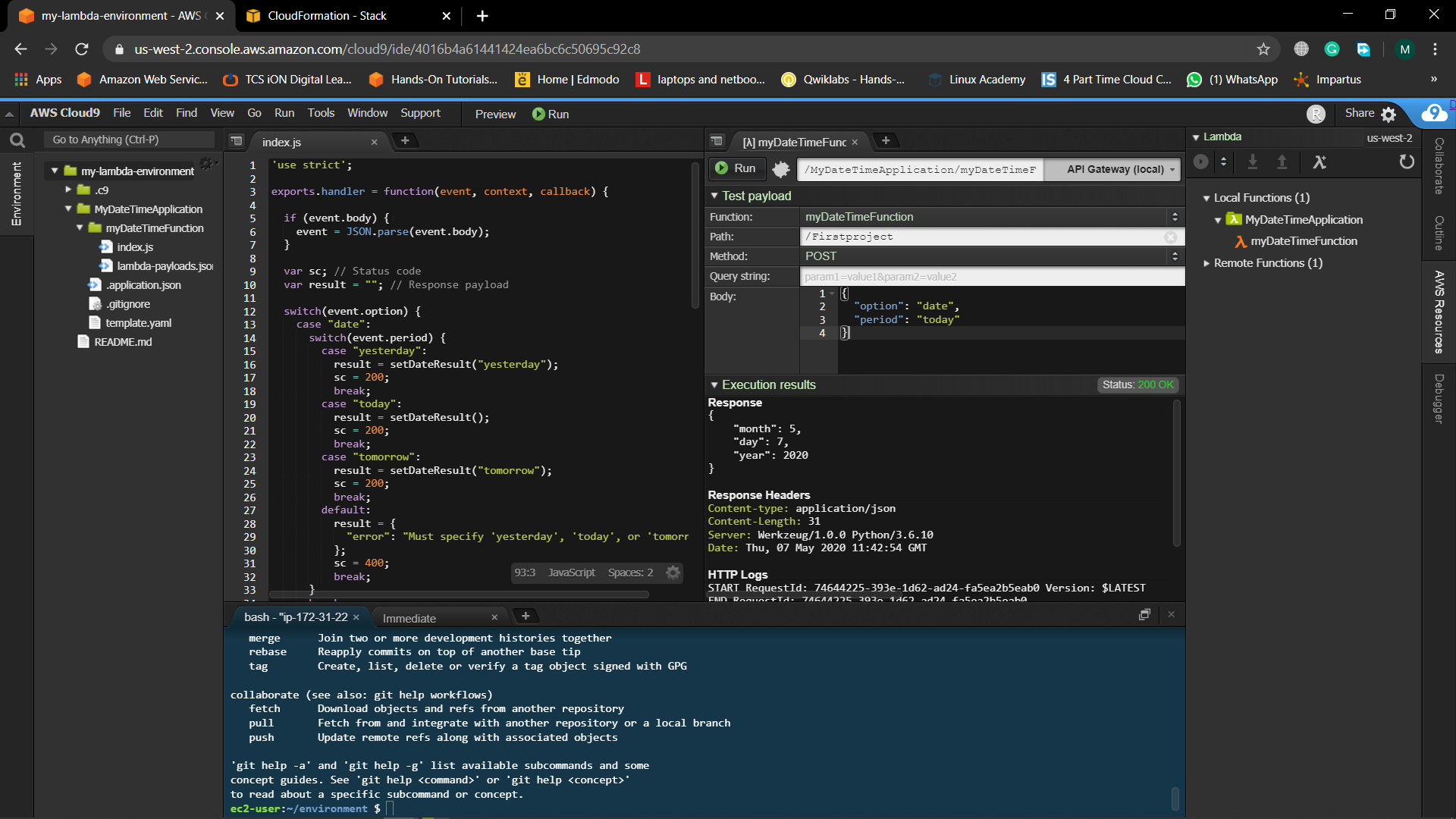
}

1. Run the function several times with the following payloads to see what the **Response** pane displays.
2. {
3. "option": "date",
4. "period": "yesterday"
5. }
6. {
7. "option": "date",
8. "period": "tomorrow"
9. }
10. {
11. "option":"time"
12. }
13. Displays an error. Must also specify a 'period' of 'yesterday', 'today', or 'tomorrow'.
14. {
15. "option":"date"
16. }
17. Displays an error. Must specify an 'option' of 'date' or 'time'.
18. {
19. "option":"dates"

}

Compare your results to the following.

Fig:20



**Step 5.2: Debug the API Locally**

In this step, you use the IDE to have API Gateway debug the Lambda function on the instance.

1. Ensure that the index.js file still has a breakpoint set on the line of code callback(null, response).
2. Ensure that the **Watch Expressions** area of the **Debugger** window is still watching event.option, event.period, sc, and response.body.
3. On the run tab from the previous step, choose the icon that looks like a bug. (It switches from grey to green.)
4. In the **Body** pane on the run tab, ensure the following is still visible.
5. {
6. "option": "date",
7. "period": "today"

}

1. Choose **Run**.

Code execution pauses at the breakpoint, and the current values of event.option, event.period, sc, and response.body are displayed in the **Watch Expressions** area of the **Debugger** window.

You can also see these values in the code by hovering your mouse over option, period, sc, and response in the code. For response, expand **Object {}**, and then see the **body** value.

1. In the Debugger window, choose the blue **Resume** arrow to finish running the code.

**Debug the API Locally using python:**

1.we can choose the lambda function and named the application and function as Demoprojectapplication and Demoprojectfunction.Then choose the runtime enivironment as python3.8.

2.Then choose the empty python function don’t give any function trigger and give it some default memory don’t give any security choose automatic generate role then choose next.

3.Then a basic python empty file is created . write a small python program to debug it.

**Python program:**

**a = 3**

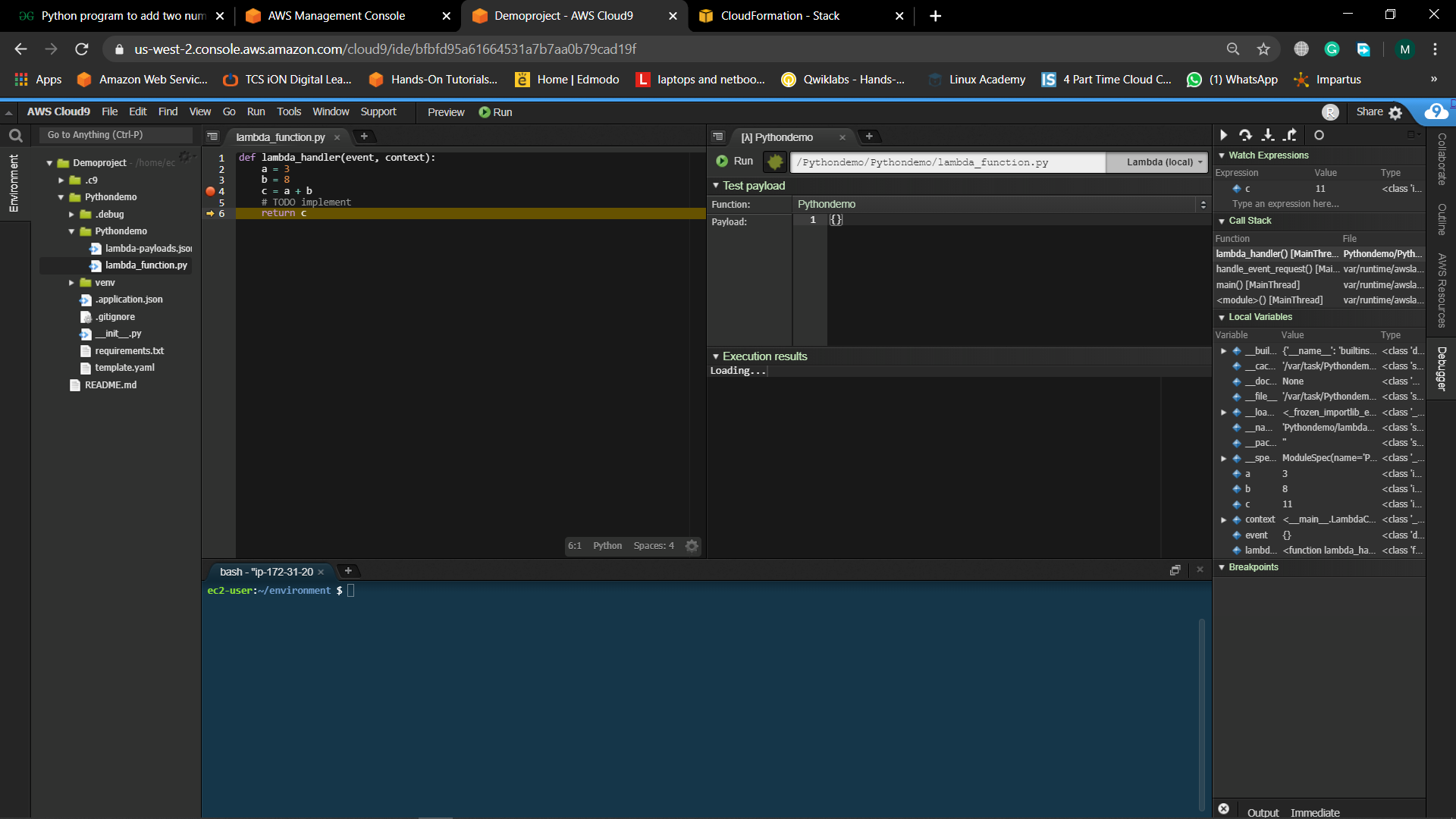
**b = 2**

**c = a + b**

**return c**

4.Add a watch expression to the debugger part and edit the expression as c When the process of debugging happen then it don’t show the output when we click on run when the process happen it stopped at c because of the breakpoint.when we click on the step over option present on top of the watch expression the execution process completed and the output shows near the value part of the watchexpression.

Fig:21



# Step 6: Run the Function in Production

In this step, you use the IDE to run the function in Lambda itself.

AWS Cloud9 deployed the function to Lambda during [Step 2: Create the Lambda Function and API](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-create-function.html). However, AWS Cloud9 deployed the function before you made the changes in [Step 3: Add Code to the Function](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-add-code.html). That original function was very basic, taking no payload and returning no response. So, you must first deploy your changes to Lambda, and then you can run the deployed function there.

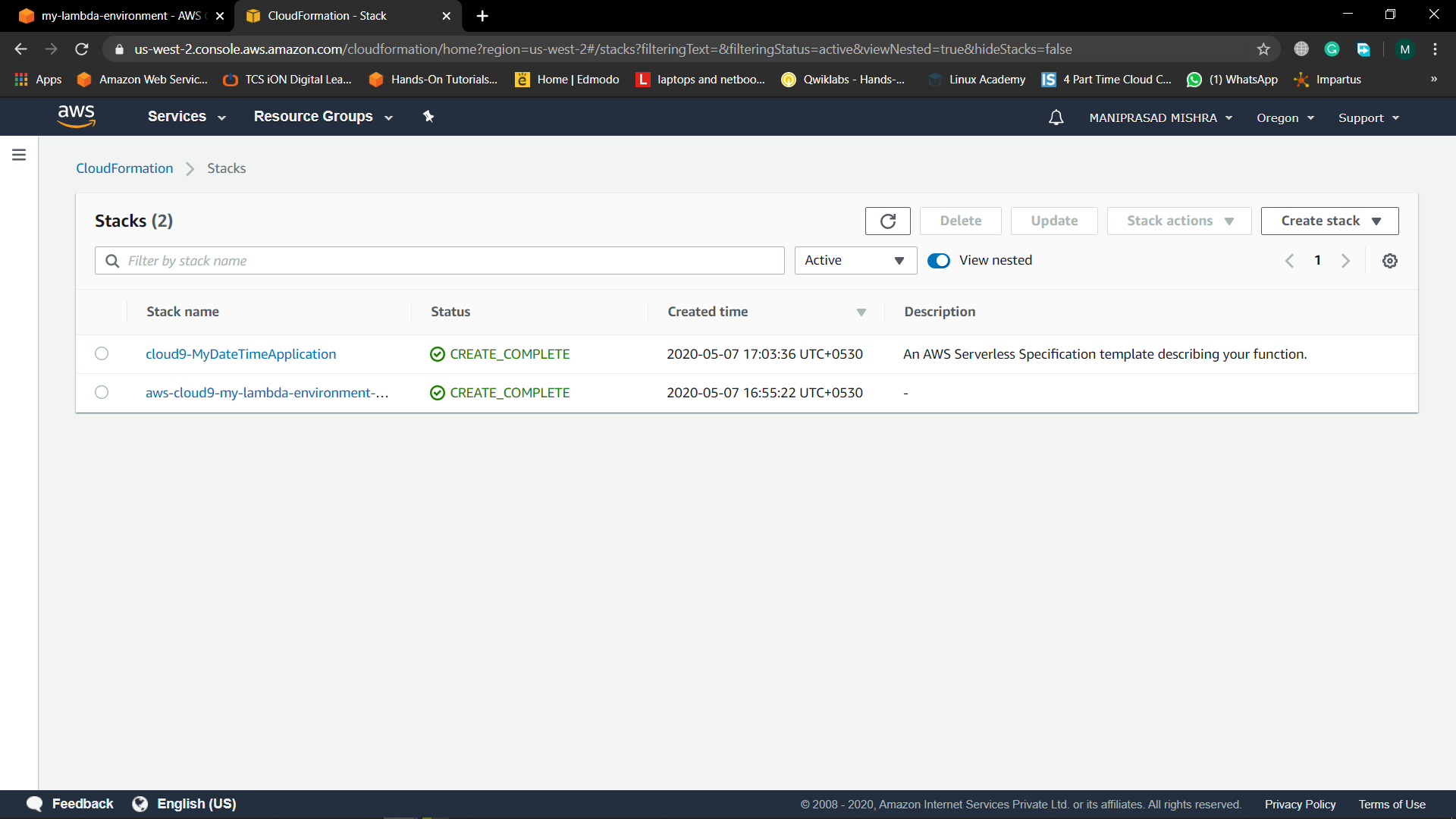
1. In the **Lambda** pane of the **AWS Resources** window, expand **Local Functions**, expand the **MyDateTimeApplication** Lambda folder, right-click the **myDateTimeFunction** Lambda icon, and then choose **Deploy**.
2. After the deployment finishes, right-click the **myDateTimeFunction** Lambda icon, and then choose **Run, Run Remote**.

**Note**

You can also do this by choosing the **API Gateway (local)** list on the run tab from the previous step, and then choosing **Lambda (remote)**.

1. In the **Payload** pane on the run tab, be sure one of the payloads from [Step 4.1: Run the Function Locally](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-or-debug-local-function.html#tutorial-lambda-run-local-function) is visible. If it isn't there, add it.
2. Choose **Run**, and see the results in the **Response** area. These should be the same results as in [Step 4.1: Run the Function Locally](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-or-debug-local-function.html#tutorial-lambda-run-local-function).

Fig:22



# Step 7: Run the API in Production

In this step, you use the IDE to run the API in API Gateway itself.

**Note**

AWS Cloud9 deployed the API to API Gateway during [Step 2: Create the Lambda Function and API](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-create-function.html). Because you haven't made any changes to the API since AWS Cloud9 first deployed it, you don't need to deploy the API again before you can run it in API Gateway itself.

1. In the **Lambda** pane of the **AWS Resources** window, right-click the **myDateTimeFunction** Lambda icon, and then choose **Run, Run APIGateway Remote**.

**Note**

You can also do this by choosing the **Lambda (remote)** list on the run tab from the previous step, and then choosing **API Gateway (remote)**.

1. Ensure **Path** is still set to **/**, **Method** is still set to **POST**, and **Body** is still set to one of the bodies in [Step 5.1: Run the API Locally](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-or-debug-local-api.html#tutorial-lambda-run-local-api). If any of these aren't set correctly, set them.
2. Choose **Run**, and see the results in the **Response** area, which should be the same results as in [Step 5.1: Run the API Locally](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-or-debug-local-api.html#tutorial-lambda-run-local-api).

# Step 8: Change the Function and Deploy the Change

## Step 8.1: Change the Function Locally

In this step, you use the IDE to make a small change to the Lambda function. You then test the changed function locally and the original function in production to see the differences.

1. In the index.js file, on line 41, add a time value to the result variable, as follows.
2. result = {
3. "time": d.toTimeString(),
4. "hour": h,
5. "minute": mi,
6. "second": s

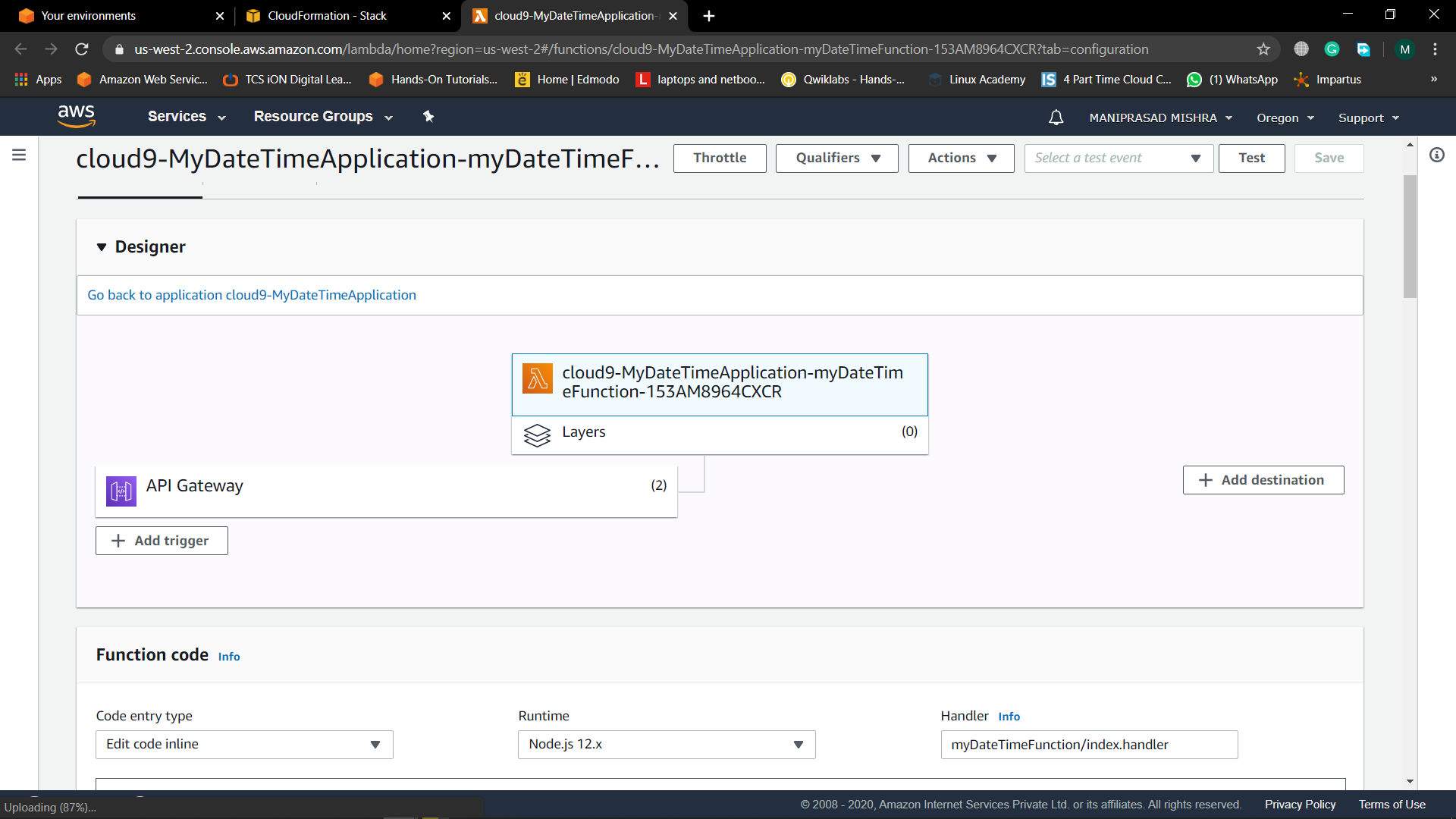
};

1. On line 86, add a date value to the result variable, as follows.
2. result = {
3. "date": d.toDateString(),
4. "month": mo,
5. "day": da,
6. "year": y

};

1. Save your changes to the index.js file.
2. Run the changed function locally by following the instructions in [Step 4.1: Run the Function Locally](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-or-debug-local-function.html#tutorial-lambda-run-local-function). Notice that the time or date value is now displayed in the response.
3. Run the original function in production by following the instructions in [Step 6: Run the Function in Production](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-deployed-function.html). Notice that the time or date value doesn't display in the response yet. This is because you have not deployed the changed function into production. You do this in the next step.

Fig:23



## Step 8.2: Deploy the Changed Function into Production

In this step, you deploy the changed function again to Lambda itself. You then test the changed function in production to confirm the deployment.

1. In the **Lambda** pane of the **AWS Resources** window, right-click the **myDateTimeFunction** Lambda icon, and then choose **Deploy**.
2. After the deployment succeeds, run the changed function in production by following the instructions in [Step 6: Run the Function in Production](https://docs.aws.amazon.com/cloud9/latest/user-guide/tutorial-lambda-run-deployed-function.html). Notice that the time or date value now appears in the response.

Remember, whenever you make a change to the local function and you want to deploy those changes to Lambda (and API Gateway, if an accompanying API exists), be sure to follow this step.

# Step 9: Clean Up

**Step 9.1: Delete the Function and the API from Lambda and API Gateway**

For AWS Cloud9 to create the function and its associated API, behind the scenes AWS Cloud9 uses the AWS Serverless Application Model (AWS SAM) to create a stack in AWS CloudFormation. This stack then creates the function and its associated API. In this procedure, you use the IDE to have AWS CloudFormation delete the stack, which also deletes the function and the API. (You could use the Lambda and API Gateway consoles instead of AWS CloudFormation to delete the function and its associated API. However, that approach takes longer and still leaves the stack in AWS CloudFormation when it's no longer needed.)

**Warning**

Deleting a stack cannot be undone. When you delete this stack, the associated function and its API are deleted from Lambda and API Gateway and cannot be recovered.

1. From the IDE, use the AWS Command Line Interface (AWS CLI) in the terminal to run the AWS CloudFormationdelete-stack command, specifying the name and the AWS Region ID for the stack. (To display the terminal, on the menu bar, choose **Window, New Terminal**). This stack's name follows the format cloud9-APPLICATION\_NAME, so you would specify cloud9-MyDateTimeApplication for this tutorial. To get the AWS Region ID (represented in the following command as us-east-2), see the corner of the **Lambda** pane in the **AWS Resources** window.

aws cloudformation delete-stack --stack-name cloud9-MyDateTimeApplication --region us-east-2

If the command ran successfully, no output and no error message are displayed.

**Note**

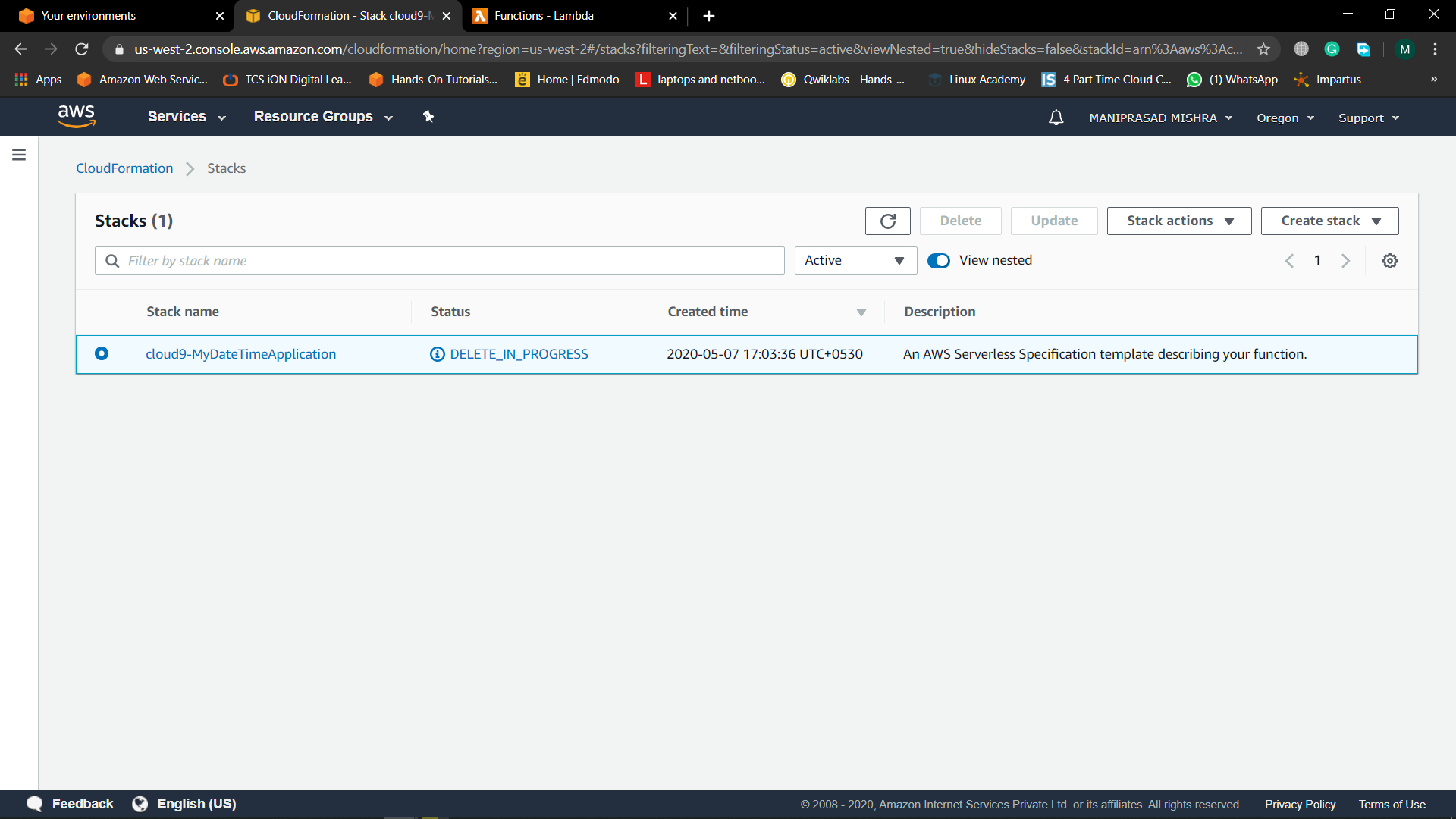
If you use an IAM user to run this command for this tutorial, instead of an AWS account root user or an IAM administrator, the IAM user must have the following additional AWS access permissions.

* + cloudformation:ListStacks
  + cloudformation:DeleteStack

If you cannot add these permissions to the IAM user, see your organization's AWS account administrator.

1. To verify that the stack is deleted, use the AWS CLI to run the AWS CloudFormationdescribe-stacks command. If the function is deleted, a message is displayed that the stack doesn't exist. (You might need to run this command multiple times until that message is displayed.)

Fig:24



aws cloudformation describe-stacks --query 'Stacks[0].StackStatus' --output text --stack-name cloud9-MyDateTimeApplication --region us-east-2

1. If you no longer want to keep the local function in the IDE, delete the ~/environment/MyDateTimeApplication folder (for example, by running the command rm -rf ~/environment/MyDateTimeApplication).

**Note**

In the IDE, ~/environment is the same as specifying the root directory in the **Environment** window.

**Step 9.2: Delete the Environment from AWS Cloud9**

**Warning**

Deleting an environment cannot be undone. Also, when you delete an EC2 environment, AWS Cloud9 also terminates the Amazon EC2 instance that it previously launched and connected to the environment. Once terminated in Amazon EC2, the instance cannot be reactivated or recovered.

1. From the IDE, open the dashboard in the AWS Cloud9 console. To do this, on the menu bar, choose **AWS Cloud9**, **Go To Your Dashboard**.
2. Do one of the following:
   * Choose the title inside of the **my-demo-lambda-environment** card, and then choose **Delete**.
   * Select the **my-demo-lambda-environment** card, and then choose **Delete**.
3. In the **Delete** dialog box, type Delete, and then choose **Delete**.

Fig:25

