Event Driven Architecture with Amazon API Gateway, Amazon EventBridge and AWS Lambda

**SPL-TF-200-SVGWEB - 1.0.8**

© 2023 Amazon Web Services, Inc. or its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited. All trademarks are the property of their owners.

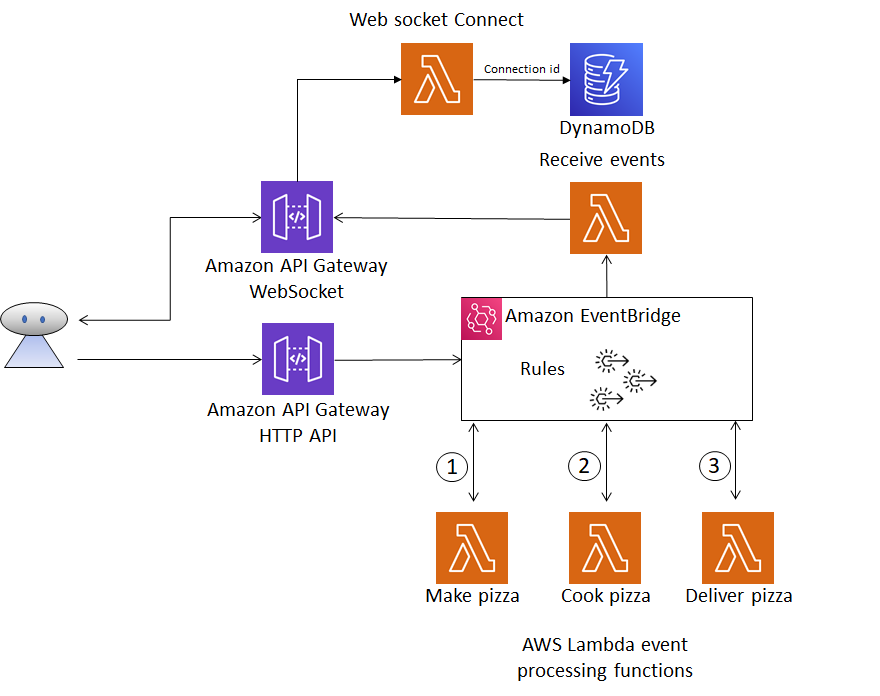
Note: Do not include any personal, identifying, or confidential information into the lab environment. Information entered may be visible to others.

Corrections, feedback, or other questions? Contact us at [*AWS Training and Certification*](https://support.aws.amazon.com/#/contacts/aws-training).

**Lab overview**

Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. An event-driven architecture uses events to start and communicate between decoupled services and is common in modern applications built with microservices. A serverless architecture is a way to build and run applications and services without having to manage infrastructure. This lab brings these concepts together to build an event-driven serverless architecture using API Gateway, EventBridge and Lambda.

In this lab, you configure an HTTP API on API Gateway to redirect requests to EventBridge. You create event bus rules that match a request and forward events to Lambda functions. Events processed by the Lambda functions are sent back to the bus as a new event. Each time an event is published to the event bus, a separate Lambda function receives the event and post it back to client application using a web socket connection hosted on an API Gateway.



*Diagram: The image depicts an AWS architectural diagram after the lab has been completed.*

OBJECTIVES

By the end of this lab, you will be able to:

* Create an Lambda function and configure environment variables.
* Create EventBridge rules targeting Lambda functions
* Create a HTTP API and Web Socket endpoint using API Gateway.
* Create an API Gateway method to integrate with EventBridge.

PREREQUISITES

This lab requires:

* Access to a computer with Microsoft Windows, Mac OS X, or Linux (Ubuntu, SuSE, or Red Hat)
* A modern internet browser such as Chrome or Firefox

DURATION

This lab requires approximately *45* minutes to complete.

ICON KEY

Various icons are used throughout this lab to call attention to different types of instructions and notes. The following list explains the purpose for each icon:

* **Expected output:** A sample output that you can use to verify the output of a command or edited file.
* **Note:** A hint, tip, or important guidance.

**Start lab**

1. To launch the lab, at the top of the page, choose **Start lab**.

 You must wait for the provisioned AWS services to be ready before you can continue.

1. To open the lab, choose **Open Console**.

You are automatically signed in to the AWS Management Console in a new web browser tab.

**Do not change the Region unless instructed.**

COMMON SIGN-IN ERRORS

**Error: You must first sign out**



If you see the message, **You must first log out before logging into a different AWS account:**

* Choose the **click here** link.
* Close your **Amazon Web Services Sign In** web browser tab and return to your initial lab page.
* Choose **Open Console** again.

**Error: Choosing Start Lab has no effect**

In some cases, certain pop-up or script blocker web browser extensions might prevent the **Start Lab** button from working as intended. If you experience an issue starting the lab:

* Add the lab domain name to your pop-up or script blocker’s allow list or turn it off.
* Refresh the page and try again.

**Task 1: Create Lambda Functions**

In this task, you create five Lambda functions. The first three functions are configured to process specific events registered to the event bus. Each function processes an event, and then sends a new event back to the bus. This starts the next function in sequence. You also create a function that writes the web socket connection ID to an Amazon DynamoDB table. The connection ID is used by the final Lambda function to track and send event information back to the browser using the web socket connection.

You start the lab by creating the **make\_pizza** Lambda function. The function receives an event, update the value of the **eventtype** attribute to **cook\_pizza** and send the event back to the EventBridge. The **cook\_pizza** event matches an EventBridge rule that targets another Lambda function.

1. At the top of the AWS Management Console, in the search bar, search for and choose

Lambda

.

1. Navigate to the Lambda Console and select **Create function** .
2. On the **Create function** screen, fill in the following fields.

* **Author from scratch**
* For **Function name**, enter:

make\_pizza

* For **Runtime**, select: **Python 3.9**
* For **Change default execution role**, choose **Use an existing role** and select: **lab\_lambda\_make\_pizza**

1. Select **Create a function** .

**Expected service output:**

**Successfully created the function make\_pizza. You can now change its code…**

1. Replace the code in **lambda\_function.py** file with the following code and select **Deploy** .

import json

import boto3

import os

from botocore.exceptions import ClientError

client = boto3.client('events')

def lambda\_handler(event, context):

try:

detail = event["detail"]

detail["item"]["eventtype"]="cook\_pizza"

response = client.put\_events(

Entries=[

{

'DetailType': 'eventtype',

'Detail': json.dumps(detail),

'EventBusName': os.environ.get('EVENT\_BUS'),

'Source':"make\_pizza"

},

]

)

print(response)

except ClientError as err:

print(err)

**Expected service output:**

**Successfully updated the function make\_pizza.**

During this lab, you use Lambda environment variables to store URL’s and table names. When the value is required, the key name is used to query the variable.

1. On the **Configuration** tab select **Environment variables**, choose **Edit** and select **Add environment variable** .
2. Configure the variable with the following settings.
   * For **Key**, enter:

EVENT\_BUS

* + For **Value**, enter:

lab\_event\_bus

1. Choose **Save** when you are finished.

**Expected service output:**

**Successfully updated the function make\_pizza.**

Create the **cook\_pizza** Lambda function. The function receives an event, update the value of the **eventtype** attribute to **deliver\_pizza** and send the event back to the EventBridge. The **deliver\_pizza** event matches an EventBridge rule that targets another Lambda function.

1. Navigate back to the Lambda Functions screen, and select **Create function** .
2. On the **Create function** screen, fill in the following fields.

* **Author from scratch**
* For **Function name**, enter:

cook\_pizza

* For **Runtime**, select: **Python 3.9**
* For **Change default execution role**, choose **Use an existing role** and select: **lab\_lambda\_cook\_pizza**

1. Select **Create function** .

**Expected service output:**

**Successfully created the function cook\_pizza. You can now change its code**

1. Replace the code in **lambda\_function.py** file with the following code and select **Deploy** .

import json

import boto3

import os

from botocore.exceptions import ClientError

client = boto3.client('events')

def lambda\_handler(event, context):

try:

detail = event["detail"]

detail["item"]["eventtype"]="deliver\_pizza"

response = client.put\_events(

Entries=[

{

'DetailType': 'eventtype',

'Detail': json.dumps(detail),

'EventBusName': os.environ.get('EVENT\_BUS'),

'Source':"cook\_pizza"

},

]

)

print(response)

except ClientError as err:

print(err)

**Expected service output:**

**Successfully updated the function cook\_pizza.**

1. On the **Configuration** tab select **Environment variables**, choose **Edit** and select **Add environment variable** .
2. Configure the variable with the following settings.
   * For **Key**, enter:

EVENT\_BUS

* + For **Value**, enter:

lab\_event\_bus

1. Choose **Save** when you are finished.

**Expected service output:**

**Successfully updated the function cook\_pizza.**

Create the **deliver\_pizza** Lambda function. The function receives an event, update the value of the **eventtype** attribute to **delivered** and send the event back to the EventBridge.

1. Navigate back to the Lambda Functions screen, and select **Create function** .
2. On the **Create function** screen fill in the following fields.

* **Author from scratch**
* For **Function name**, enter:

deliver\_pizza

* For **Runtime**, select: **Python 3.9**
* For **Change default execution role**, choose **Use an existing role** and select: **lab\_lambda\_deliver\_pizza**

1. Select **Create function** .

**Expected service output:**

**Successfully created the function deliver\_pizza. You can now change its code**

1. Replace the code in **lambda\_function.py** file with the following code and select **Deploy** .

import json

import boto3

import os

from botocore.exceptions import ClientError

client = boto3.client('events')

def lambda\_handler(event, context):

try:

detail = event["detail"]

detail["item"]["eventtype"]="delivered"

response = client.put\_events(

Entries=[

{

'DetailType': 'eventtype',

'Detail': json.dumps(detail),

'EventBusName': os.environ.get('EVENT\_BUS'),

'Source':"deliver\_pizza"

},

]

)

except ClientError as err:

print(err)

**Expected service output:**

**Successfully updated the function deliver\_pizza.**

1. On the **Configuration** tab select **Environment variables**, choose **Edit** and select **Add environment variable** .
2. Configure the variable with the following settings.
   * For **Key**, enter:

EVENT\_BUS

* + For **Value**, enter:

lab\_event\_bus

1. Choose **Save** when you are finished.

**Expected service output:**

**Successfully updated the function deliver\_pizza.**

Web socket connections are two-way persistent connections, often used by chat applications. When a web socket connection is opened with the API Gateway, a connection ID is generated to uniquely identify the connection. The next Lambda function that you create, gets invoked by the API Gateway and saves the connection ID to an Amazon DynamoDB table. The ID is used to send events back through the open connection to the client. The web socket connection is opened when you select **Order Pizza**, the application makes a call to establish the connection before calling the HTTP API endpoint and placing the order.

1. Create the Lambda function to save the web socket connection ID. Navigate to the Lambda Functions screen, and select **Create function** .
2. On the **Create function** screen fill in the following fields.

* **Author from scratch**
* For **Function name**, enter:

websocket\_connect

* For **Runtime**, select: **Python 3.9**
* For **Change default execution role**, choose **Use an existing role** and select: **lab\_lambda\_websocket\_connection**

1. Select **Create function** .

**Expected service output:**

**Successfully created the function websocket\_connect. You can now change its code**

1. Replace the code in **lambda\_function.py** file with the following code and select **Deploy** .

import boto3

import json

import os

from botocore.exceptions import ClientError

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table(os.environ.get('TABLENAME'))

def lambda\_handler(event, context):

try:

order\_id=event['queryStringParameters']['order\_id']

response = table.put\_item(

Item={

'order\_id': order\_id,

'connection\_id': event["requestContext"]["connectionId"]

})

return {

"statusCode": 200,

"headers": {

"Content-Type": "application/json"

}

}

except ClientError as err:

print(err)

**Expected service output:**

**Successfully updated the function websocket\_connect.**

1. On the **Configuration** tab select **Environment variables**, choose **Edit** and select **Add environment variable** .
2. Configure the variable with the following settings.
   * For **Key**, enter:

TABLENAME

* + For **Value**, enter:

websocket\_connections

1. Choose **Save** when you are finished.

**Expected service output:**

**Successfully updated the function websocket\_connect.**

The client application needs to track the events as they get added to the event bus. To do this, involves two AWS services; API Gateway hosts the web socket API and an Lambda function receives the events from the event bus and post the events to the open web socket connection.

1. Create the **receive\_events** Lambda function. Navigate back to the Lambda Functions screen, and select **Create function** .
2. On the **Create function** screen fill in the following fields.

* **Author from scratch**
* For **Function name**, enter:

receive\_events

* For **Runtime**, select: **Python 3.9**
* For **Change default execution role**, choose **Use an existing role** and select: **lab\_lambda\_receive\_events**

1. Select **Create function** .

**Successfully created the function receive\_events. You can now change its code**

1. Replace the code in **lambda\_function.py** file with the following code and select **Deploy** .

import boto3

import json

import os

from botocore.exceptions import ClientError

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table(os.environ.get('TABLENAME'))

management = boto3.client('apigatewaymanagementapi', endpoint\_url=os.getenv('APIGW\_ENDPOINT'))

def lambda\_handler(event, context):

try:

response = table.get\_item(

Key={

'order\_id': event["detail"]['item']['order\_id']

}

)

management.post\_to\_connection(

Data=json.dumps(event["detail"]),

ConnectionId=response["Item"]["connection\_id"]

)

except ClientError as err:

print(err)

1. On the **Configuration** tab select **Environment variables**, choose **Edit** and select **Add environment variable** .
2. Configure the variable with the following settings.
   * For **Key**, enter:

TABLENAME

* + For **Value**, enter:

websocket\_connections

1. Choose **Save** when you are finished.

**Expected service output:**

**Successfully updated the function receive\_events.**

**Congratulations!** You have created the Lambda functions that process the events for your event-driven architecture. You have also created environment variables to store configuration settings used by the function logic.

**Lambda Functions**

* make\_pizza
* cook\_pizza
* deliver\_pizza
* websocket\_connect
* receive\_events

**Task 2: Configure EventBridge**

In this task, you create an EventBridge event bus. You create the bus with rules to redirect events to Lambda targets. Each rule is configured with a custom pattern and a target Lambda function. You also create a single rule with multiple patterns, this demonstrates how different Lambda functions can be invoked simultaneously to process an event.

The first rule that is created invokes the **make\_pizza** lab function. This rule is configured to match a request body that includes an attribute called **eventtype** with the value **make\_pizza**. It is also important to note that the rule needs to match the nesting of the **eventtype** attribute, not just the value.

1. At the top of the AWS Management Console, in the search bar, search for and choose

EventBridge

.

1. From the left-hand navigation pane of the EventBridge console, under the **Buses** section, choose **Event buses** and select **Create event bus** .
2. Create an event bus with the following settings, and choose **Create** when you are finished.
   * For **Name**, enter:

lab\_event\_bus

1. Copy and save the ARN for the EventBus you just created, you need this later.
2. From the left-hand navigation pane of the EventBridge console, under the **Buses** section, choose **Rules** then select **Create rule** . Configure the rule with the following settings and select **Next** .
   * For **Name** enter:

lab\_make\_pizza\_rule

* + For **Event bus** select: **lab\_event\_bus**

You can either choose an AWS service as an event source or create a custom event pattern for your own application. In this case you are going to create a custom event pattern.

1. On the **Build event pattern** screen, under the **Creation method** section, select **Custom pattern (JSON editor)** for the **Method**. This enables the pattern editor, allowing you to paste in a custom pattern.
2. Copy and paste the following json object into the **Event pattern** editor, you can use the **{} Prettify** option to make the snippet easier to read. Select **Next** when you are finished.

{

"detail": {

"item": {

"eventtype": ["make\_pizza"]

}

}

}

1. On the **Select target(s)** page, from the **Target 1** section under **Target types**, choose  **AWS service**.
2. From the **Select a target** drop-down list, choose **Lambda function**.

**Note:** If you do not see **Lambda function** in the drop-down list, choose the search field to search for it.

1. From the **Function** drop-down list, choose **make\_pizza**.
2. Select **Next** and select **Next** again on the **Configure tags - optional** screen.
3. On the **Review and create** screen, select **Create rule**.

**Expected service output:**

**Rule lab\_make\_pizza\_rule was created successfully**

**Note:** To view the rules that you have created, choose **lab\_event\_bus** from the **Event bus** dropdown list on the **Rules** screen.

Create the rule to invoke the **cook\_pizza** Lambda function. This rule is configured to match a request body that includes an attribute called **eventtype** with the value **cook\_pizza**.

1. From the left-hand navigation pane under **Events**, choose **Rules** then select **Create rule** . Configure the rule with the following settings and select **Next** .
   * For **Name** enter:

lab\_cook\_pizza\_rule

* + For **Event bus** select: **lab\_event\_bus**

1. On the **Build event pattern** screen, under the **Creation method** section, choose **Custom patterns (JSON editor)** for the **Method**.
2. Copy and paste the following json object into the **Event pattern** editor. Select **Next** when you are finished.

{

"detail": {

"item": {

"eventtype": ["cook\_pizza"]

}

}

}

1. On the **Select target(s)** page, from the **Target 1** section under **Target types**, choose  **AWS service**.
2. From the **Select a target** drop-down list, choose **Lambda function**.
3. From the **Function** drop-down list choose **cook\_pizza**.
4. Select **Next** and select **Next** again on the **Configure tags - optional** screen.
5. On the **Review and create** screen, select **Create rule** .

**Expected service output:**

**Rule lab\_cook\_pizza\_rule was created successfully**

Create the rule to invoke the **deliver\_pizza** Lambda function. This rule is configured to match a request body that includes an attribute called **eventtype** with the value **deliver\_pizza**.

1. From the left-hand navigation pane under **Events**, choose **Rules** then select **Create rule** . Configure the rule with the following settings and select **Next** .
   * For **Name** enter:

lab\_deliver\_pizza\_rule

* + For **Event bus** select: **lab\_event\_bus**

1. On the **Build event pattern** screen, under the **Creation method** section, choose **Custom patterns (JSON editor)** for the **Method**.
2. Copy and paste the following json object into the **Event pattern** editor. Select **Next** when you are finished.

{

"detail": {

"item": {

"eventtype": ["deliver\_pizza"]

}

}

}

1. On the **Select target(s)** page, from the **Target 1** section under **Target types**, choose  **AWS service**.
2. From the **Select a target** drop-down list, choose **Lambda function**.
3. From the **Function** drop-down, list choose **deliver\_pizza**.
4. Select **Next** and select **Next** again on the **Configure tags - optional** screen.
5. On the **Review and create** screen, select **Create rule** .

**Expected service output:**

**Rule lab\_deliver\_pizza\_rule was created successfully**

Create the last run that includes a pattern which tracks all the events and invoke the **receive\_events** Lambda function. This rule is configured to match a request body that includes an attribute called **eventtype**, with the value that can be **make\_pizza**, **cook\_pizza** or **deliver\_pizza**. Since there are two rules that match each of the pizza values, two Lambda functions are invoked simultaneously by separate rules to process the event.

1. From the left-hand navigation pane under **Events**, choose **Rules** then select **Create rule** . Configure the rule with the following settings and select **Next** .
   * For **Name** enter:

lab\_receive\_events\_rule

* + For **Event bus** select: **lab\_event\_bus**

1. On the **Build event pattern** screen, under the **Creation method** sections, choose **Custom patterns (JSON editor)** for the **Method**.
2. Copy and paste the following json object into the **Event pattern** editor. Select **Next** when you are finished.

{

"detail": {

"item": {

"eventtype": ["make\_pizza","cook\_pizza","deliver\_pizza"]

}

}

}

1. On the **Select target(s)** page, from the **Target 1** section under **Target types**, choose  **AWS service**.
2. From the **Select a target** drop-down list, choose **Lambda function**.
3. From the **Function** drop-down list, choose **receive\_events**.
4. Select **Next** and select **Next** again on the **Configure tags - optional** screen.
5. On the **Review and create** screen, select **Create rule** .

**Expected service output:**

**Rule lab\_receive\_events\_rule was created successfully**

**Congratulations!** You have just configured EventBridge with an event bus. You have also created event rules to redirect custom events to specific Lambda targets.

**Event Bus Rules**

* lab\_make\_pizza\_rule
* lab\_cook\_pizza\_rule
* lab\_deliver\_pizza\_rule
* lab\_receive\_events\_rule

**Task 3: Configure API Gateway**

In this task, you create and configure an HTTP API endpoint to integrate with EventBridge. Requests sent to the API by the web application redirects to the event bus, where the request body matches to a rule, and a target function invoked. You also create a web socket endpoint, which allows the client application to establish a persistent connection. This connection receives and then displays events in the browser as they occur.

REST APIs and HTTP APIs are both RESTful API products. REST APIs support more features than HTTP APIs, while HTTP APIs are designed with minimal features so that they can be offered at a lower price. You can use HTTP APIs to send requests to Lambda functions or other AWS Services, such as EventBridge.

1. At the top of the AWS Management Console, in the search bar, search for and choose

API Gateway

.

1. In the **Choose an API type** select **Build** from the **HTTP API** section.
2. In the **API Name** field, enter

lab\_http\_api

1. Select **Next** until you get to the **Review and Create** screen, then choose **Create** .

**Expected service output:** Your identifier at the end differs from this example.

**Successfully created API lab\_http\_api (dr8sknw1gk)**

1. Under the **Develop** section in the left-hand navigation pane, select **Routes**, then choose **Create** .
2. Select **POST** from the drop-down list, and select **Create** .
3. Under the **Develop** select **Integrations**, choose the POST route link and then select **Create and attach an integration** .
4. For **Integration target**, configure the following options.
   * For **Integration type**, select: **Amazon EventBridge**
   * For **Integration action**, select: **PutEvents**
5. For **EventBridge - PutEvents**, configure the following options.
   * For **Detail**, enter:

$request.body

* + For **Detail type**, enter:

eventtype

* + For **Source**, enter:

lab\_http\_api

* + For **Invocation role**, copy the **APIExecutionRoleARN** from the **left side of the instructions** pane.

1. Expand **Advanced settings**, and configure the remaining fields. Select **Create** when you are finished.
   * For **Event bus name -** *optional*, enter the event bus ARN you saved in Task 2.
   * For **Region** - *optional*, copy the **AWSRegion** from the **left side of the instructions** pane.
2. From the left-hand navigation pane, select **CORS** and choose **Configure** in the top right corner.
3. Use the following settings to configure **CORS**, making sure you select **Add** when configuring the settings. Select **Save** .
   * For **Access-Control-Allow-Origin** enter:

\*

* + For **Access-Control-Allow-Methods** select: **POST**
  + For **Access-Control-Allow-Headers** enter:

\*

1. From the navigation menu on the left-hand side of the screen, navigate to the endpoint details screen by selecting **API: lab\_http\_api**. Copy and save the **Invoke URL** from the **Stages for lab\_http\_api** section of the **API: lab\_api** screen. The URL is needed in Task 4.

Now create the web socket API endpoint, which is used to send events back to the web application. Web socket connections are two-way persistent connections allowing bi-directional communication and are often used by chat applications.

1. At the top of the left-hand navigation pane, select **APIs**, then select **Create API** .
2. In the **Choose an API type** select **Build** from the **WebSocket API** section.
3. Configure the following fields and select **Next** .
   * For the **API Name** field, enter

lab\_websocket\_api

* + For **Route selection expression**, enter

request.body.action

1. On the **Add routes** page choose **Add $connect route** and select **Next** .

The **websocket\_connect** function invokes when the application establishes the connection. This function saves the connection ID to a DynamoDB table.

1. For **Integration for $connect**, choose **Lambda** from the dropdown list and choose the **websocket\_connect** function. Select **Next** .
2. On the **Add Stages** page select **Next**, followed by **Create and deploy** .

**Expected service output:**

 You have successfully created lab\_websocket\_api API.

1. Select **Stages** from the left-hand navigation page and choose the **Production** stage.
2. Copy and save the **WebSocket URL**, as it is needed shortly.
3. Copy the **Connection URL** up to, but do not include the **@connections** text.

The image displays a screenshot of the WebSocket and connection URL, indicating which URL needs to be copied in the previous step.

*Image: Shows example URLs for the WebSocket URL and the Connection URL less the @connections text.*

1. Navigate back to the **Lambda console** and open the **receive\_events** Lambda function.
2. On the **configuration** tab select **Environment variables**, choose **Edit** and select **Add environment variable**.
3. Configure the variable with the following settings, and select **Save** when you are finished.
   * For **Key**, enter:

APIGW\_ENDPOINT

* + For **Value**, enter: the copied **Connection URL**

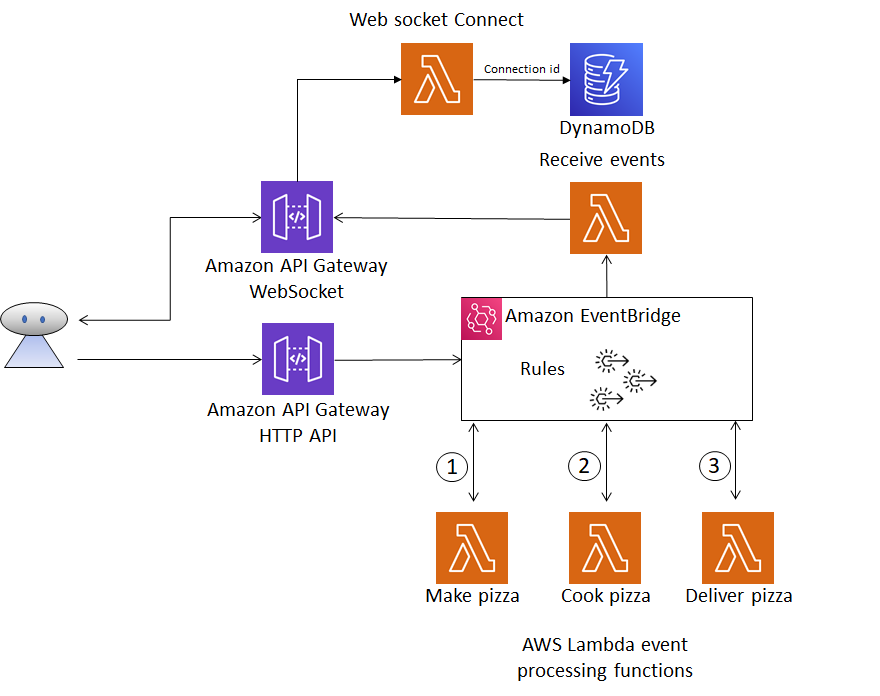
**Expected service output:**

**Successfully updated the function receive\_events.**

**Congratulations!** You have just integrated a HTTP API endpoint with EventBridge, as well as integrating a web socket endpoint with a Lambda function.

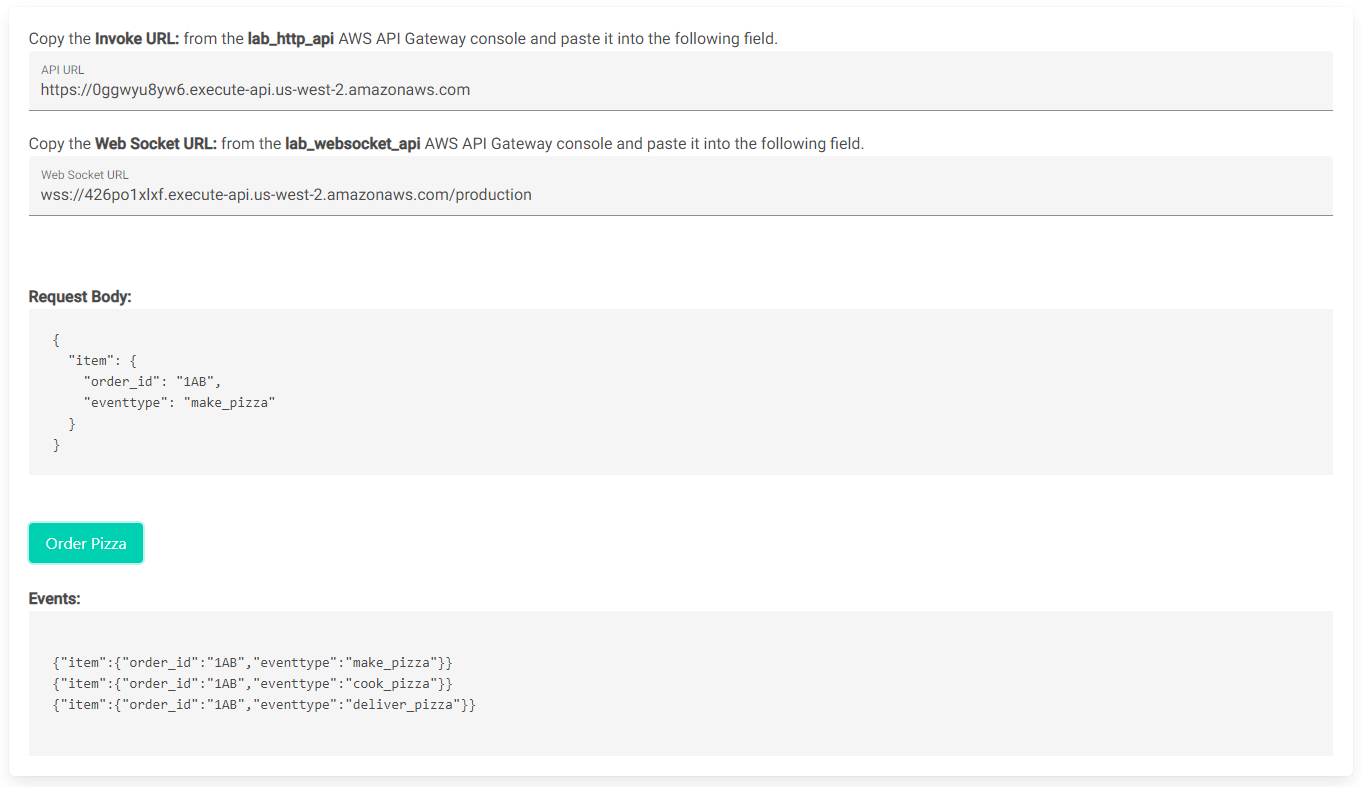
**Task 4: Testing the Event-Driven Architecture**

In this task, test the event-driven architecture by using a web application to send events to the API Gateway. The HTTP endpoint redirects the event to the event bus, which uses a rule to invoke a Lambda function. The processing functions work in sequence, the order is determined by the event data sent back by the function. The receive event function is invoked for each event simultaneously with the other functions.



*Diagram: The image depicts an AWS architectural diagram after the lab has been completed.*

1. Copy the WebServerIP address from the **left side of the instructions** pane, paste the IP address into a new browser tab and navigate to the website.
2. Copy the **Invoke URL** from the **lab\_http\_api** AWS API Gateway console and paste it into the **API URL** field.
3. Copy the **WebSocket URL** from the **lab\_websocket\_api** AWS API Gateway console and paste it into the **WebSocket URL** field.
4. Select **Order Pizza** and events start to appear in the **Events** text area. The events are being picked up by the **receive\_events** Lambda function and posted to the web socket connection. The same events are also being processed by the Lambda functions that are started in sequence. In an event-driven architecture, multiple worker processes can process the same event simultaneously.



*Image: The image displays a screenshot of a successful outcome using the testing web application. You see an event for make\_pizza, cook\_pizza, and deliver\_pizza.*

**Congratulations!** You have successfully tested the event-driven architecture.

**Conclusion**

During this lab you created a serverless event-driven architecture using EventBridge, API Gateway and Lambda. This type of architecture is ideal for improving agility, moving quickly and scalability. An event-driven architecture is commonly found in modern applications that use microservices, or any application that has decoupled components.

**Congratulations!** You have successfully:

* Created an Lambda function and configure environment variables.
* Created EventBridge rules targeting Lambda functions
* Created a HTTP API and Web Socket endpoint using API Gateway.
* Created an API Gateway method to integrate with EventBridge.

**End lab**

Follow these steps to close the console and end your lab.

1. Return to the **AWS Management Console**.
2. At the upper-right corner of the page, choose **AWSLabsUser**, and then choose **Sign out**.
3. Choose **End lab** and then confirm that you want to end your lab.

**Additional resources**

* For more information about how to use Lambda, see [Lambda Documentation](https://docs.aws.amazon.com/lambda/index.html).
* For more information about how to use API Gateway, see [API Gateway Documentation](https://docs.aws.amazon.com/apigateway/index.html).
* For more information about how to use EventBridge, see [EventBridge Documentation](https://docs.aws.amazon.com/eventbridge/?id=docs_gateway).

For more information about AWS Training and Certification, see [*https://aws.amazon.com/training/*](https://aws.amazon.com/training/).

*Your feedback is welcome and appreciated.*  
If you would like to share any feedback, suggestions, or corrections, please provide the details in our [*AWS Training and Certification Contact Form*](https://support.aws.amazon.com/#/contacts/aws-training).