Module 5 - WebSocket APIs

Important Note
This module is **independent from other modules**, however it's assumed that the **"Getting Started"** part was done already.

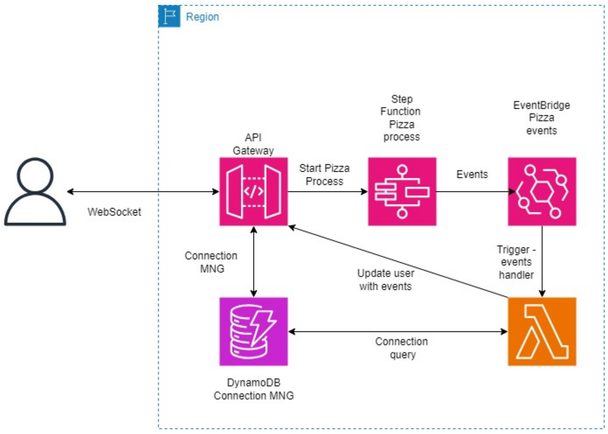
In this module you will learn how to create a WebSocket application using SAM (Serverless Application Model).

WebSocket protocol provides simultaneous two-way communication channels over a single Transmission Control Protocol (TCP) connection. It enables full-duplex interaction between a web browser (or other client application) and server with lower overhead than half-duplex alternatives such as HTTP polling (REST), facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the client without being first requested by the client, and allowing messages to be passed back and forth while keeping the connection open. In this way, a two-way ongoing conversation can take place between the client and the server. Using WebSocket, an application frontend can become an integral part of an EDA (Event Driven Architecture) system, as it can receive events (messages) from the backend, instead of polling the backend periodically.

Some examples for applications and use cases that can benefit from using WebSocket:

1. Real time communication chat based applications (e.g WhatsApp).
2. A need for server to client communication (e.g sending notifications), usually called server-push.
3. Performance & reduced latency: since the connection is already established, each side can send data a lot more efficiently than via a HTTP request (REST) that necessarily contains headers, cookies etc. Also, latency is lower as data can be sent quicker.

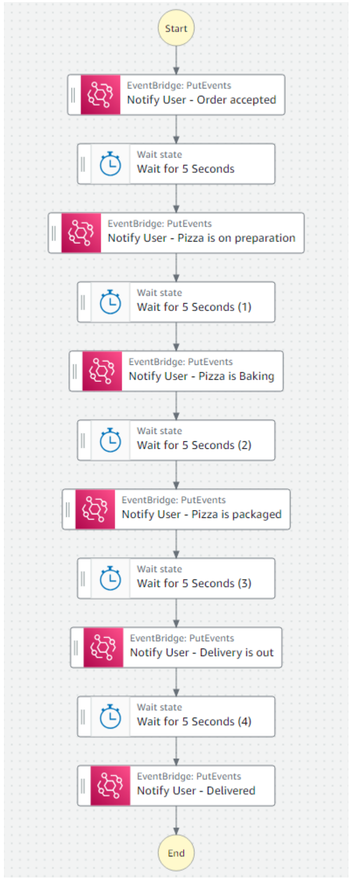
In this module we are going to implement our **AWSome Pizza Bar**. The backend will be able to receive a pizza order from the user frontend. The order of the pizza is being handled by the backend using AWS Step Functions to orchestrate the process. The user frontend will receive periodic updates using API Gateway and WebSocket based API.



This is the flow:

1. A user opens a new connection to its endpoint that uses a WebSocket api, provided by the Amazon API Gateway. API Gateway $connect route is invoked. It inserts a new connection entry in the DynamoDB Sessions table for persisting the connection.
2. The user sends a new order request for a pizza. API Gateway $default route is invoked. This triggers an AWS Step Functions state machine to process the order and prepare the pizza by the backend.
3. The state machine triggers events to AWS EventBridge. For example, if the pizza starts its baking, an event will be send.
4. EventBridge triggers a Lambda function to process each event. The Lambda function processes each event according to it's state. The Lambda function also updates the DynamoDB `Orders` table for its current pizza order status, and pushes notification to the user (via Amazon API Gateway) using the WebSocket connection.
5. When the user closes the connection, or it's timed out, API Gateway $disconnect route is invoked. It deletes the connection entry in the DynamoDB Sessions table.

Now let's zoom in to the AWS Step Function state machine that is processing the order, and preparing the pizza:



We can see there are several steps to notify the user on events during the order and pizza preparation. We also implemented a "Wait State" between the events, to simulate the time it takes to prepare the pizza. In this example we specified 5 seconds between each step, in reality the orchestrion process can take longer and can be manually controlled by the pizzeria employees.

Estimated Duration: 60 min

Set up your AWS SAM Project

1. Navigate to Cloud9 in your AWS console and make sure you are in the correct region.
2. Click **Open** on the **APIGatewayWorkshopWorkspace** environment.
3. The AWS Cloud9 environment comes with some AWS utilities pre-installed. Run the following command in your AWS Cloud9 terminal to verify that it contains an updated version of AWS SAM.

```
sam --version
```

4. Set up your AWS SAM project Folder and Files:

```
mkdir -p module-5 && cd module-5
```

5. Use the below command to create the SAM template file:

```
touch template.yaml
```

- `template.yaml` - This file is the primary AWS SAM configuration file. AWS SAM templates are an extension of AWS CloudFormation templates, with some additional components that make them easier to work with. For the full reference for AWS CloudFormation templates, see AWS CloudFormation Template Reference in the AWS CloudFormation User Guide.

Define the Amazon States Languages file

This module uses AWS Step Functions as an orchestrator for the AWSome Pizza Bar flow. AWS Step Functions is a cloud service that enables you to coordinate and manage the components of distributed applications and microservices using visual workflows. These workflows can be described using Amazon States Languages (ASL). Amazon States Languages (ASL) is a JSON-based language used to define state machines in AWS Step Functions, specifying the sequence of steps and conditions that makes up your workflow.

> Step Functions can control certain AWS services directly from the Amazon States Language (ASL). Step Functions provides a service integration API for integrating with Amazon EventBridge. This lets you build event-driven applications by sending custom events directly from Step Functions workflows.

1. Set up your state machine project folder:

```
mkdir -p statemachine && cd statemachine
```

2. Use the below command to create the ASL definition file:

```
touch api-gw-websocket-asl.json
```

3. Review the code and then copy/paste it into the `api-gw-websocket-asl.json` file.

```
{
  "Comment": "AWSome Pizza Bar Flow",
  "StartAt": "Notify User - Order accepted",
  "States": {
    "Notify User - Order accepted": {
      "Type": "Task",
      "Resource": "arn:aws:states:::events:putEvents",
      "Parameters": {
        "Entries": [
          {
            "Detail": {
              "target": "LAMBDA to notify user via WebSocket",
              "connectionId.$": "$.connectionId",
              "status": "order accepted",
              "orderDetails.$": "$.data"
            },
            "DetailType": "OrderStatusChangedEvent",
            "EventBusName": "${PizzaOrderEventBus}",
            "Source": "WebsocketEvent"
          }
        ]
      },
      "ResultPath": null,
      "Next": "Wait for 5 Seconds"
    },
    "Wait for 5 Seconds": {
      "Next": "Notify User - Pizza is on preparation",
      "Seconds": 5,
      "Type": "Wait"
    },
    "Notify User - Pizza is on preparation": {
      "Type": "Task",
      "Resource": "arn:aws:states:::events:putEvents",
      "Parameters": {
        "Entries": [
          {
            "Detail": {
              "target": "LAMBDA to notify user via WebSocket",
              "connectionId.$": "$.connectionId",
              "status": "order is now being prepared",
              "orderDetails.$": "$.data"
            },
            "DetailType": "OrderStatusChangedEvent",
            "EventBusName": "${PizzaOrderEventBus}",
            "Source": "WebsocketEvent"
          }
        ]
      },
      "ResultPath": null,
      "Next": "Wait for 5 Seconds (1)"
    },
    "Wait for 5 Seconds (1)": {
      "Seconds": 5,
      "Type": "Wait",
      "Next": "Notify User - Pizza is Baking"
    },
    "Notify User - Pizza is Baking": {
      "Type": "Task",
      "Resource": "arn:aws:states:::events:putEvents",
      "Parameters": {
        "Entries": [
          {
            "Detail": {
              "target": "LAMBDA to notify user via WebSocket",
              "connectionId.$": "$.connectionId",
              "status": "order is undergoing the baking process",
              "orderDetails.$": "$.data"
            },
            "DetailType": "OrderStatusChangedEvent",
            "EventBusName": "${PizzaOrderEventBus}",
            "Source": "WebsocketEvent"
          }
        ]
      },
      "ResultPath": null,
      "Next": "Wait for 5 Seconds (2)"
    },
    "Wait for 5 Seconds (2)": {
      "Seconds": 5,
      "Type": "Wait",
      "Next": "Notify User - Pizza is packaged"
    },
    "Notify User - Pizza is packaged": {
      "Type": "Task",
      "Resource": "arn:aws:states:::events:putEvents",
      "Parameters": {
        "Entries": [
          {
            "Detail": {
              "target": "LAMBDA to notify user via WebSocket",
              "connectionId.$": "$.connectionId",
              "status": "order is packed",
              "orderDetails.$": "$.data"
            },
            "DetailType": "OrderStatusChangedEvent",
            "EventBusName": "${PizzaOrderEventBus}",
            "Source": "WebsocketEvent"
          }
        ]
      },
      "ResultPath": null,
      "Next": "Wait for 5 Seconds (3)"
    },
    "Wait for 5 Seconds (3)": {
      "Seconds": 5,
      "Type": "Wait",
      "Next": "Notify User - Delivery is out"
    },
    "Notify User - Delivery is out": {
      "Type": "Task",
      "Resource": "arn:aws:states:::events:putEvents",
      "Parameters": {
        "Entries": [
          {
            "Detail": {
              "target": "LAMBDA to notify user via WebSocket",
              "connectionId.$": "$.connectionId",
              "status": "order is sent for delivery",
              "orderDetails.$": "$.data"
            },
            "DetailType": "OrderStatusChangedEvent",
            "EventBusName": "${PizzaOrderEventBus}",
            "Source": "WebsocketEvent"
          }
        ]
      },
      "ResultPath": null,
      "Next": "Wait for 5 Seconds (4)"
    },
    "Wait for 5 Seconds (4)": {
      "Seconds": 5,
      "Type": "Wait",
      "Next": "Notify User - Delivered"
    },
    "Notify User - Delivered": {
      "Type": "Task",
      "Resource": "arn:aws:states:::events:putEvents",
      "Parameters": {
        "Entries": [
          {
            "Detail": {
              "target": "LAMBDA to notify user via WebSocket",
              "connectionId.$": "$.connectionId",
              "status": "order delivered",
              "orderDetails.$": "$.data"
            },
            "DetailType": "OrderStatusChangedEvent",
            "EventBusName": "${PizzaOrderEventBus}",
            "Source": "WebsocketEvent"
          }
        ]
      },
      "ResultPath": null,
      "End": true
    }
  }
}
```

The Amazon States Languages (ASL) definition outlines a set of States. Each state within the state machine is assigned the responsibility of executing particular actions in our flow. Each state in the Amazon States Language can have an associated **resource** field that specifies the action or service to be performed. To produce events to Amazon EventBrdige during workflow, AWS Step Functions adds an additional resource type `arn:aws:states:::events:putEvents`. The integration between AWS Step Function and Amazon EventBridge is configured with the following Amazon States Language (ASL) parameter fields:

```
"Detail": A valid string or JSON object.
"DetailType": Free-form string used to decide what fields to expect in the event detail.
"EventBusName": The name or ARN of the event bus to receive the event.
"Source": The source of the event.
```

The "Detail" attribute within each state is used to store order information, and its structure is as follows:

```
"Detail": {
    "target": "[Text field containing the event target]",
    "connectionId.$": "[Text field containing the current WebSocket connection ID]",
    "status": "[Text field containing the current order status]",
    "orderDetails.$": "[JSON string containing the actual payload body]"
}
```

The integration between AWS Step Function and Amazon EventBridge requires to create an EventBridge rule in your account that matches the specific pattern of the events you will send. EventBridge Rules use event pattern to select events and send them to targets. When the event matches the event pattern defined for a rule, the rule processes the event data and sends the pertinent information to the target. To match the specific pattern, each Step Function state is set with **DetailType=OrderStatusChangedEvent** and **Source=WebsocketEvent** which than will be used by the EventBridge event pattern on lines 52-56 of the template.yaml in the next step.

Create the Lambda function triggered by EventBridge

The target function which subscribes to the event emitted by the state machine is accountable for either creating or updating our Pizza order. Additionally, it dispatches a notification message to the active WebSocket connection.

1. Navigate to the root Project folder

```
cd ~/environment/module-5/
```

2. Create the Lambda Project folders

```
mkdir -p handlers && cd handlers
```

3. Use the below command to create the following files:

```
touch lambda-target-example.js common.js package.json
```

4. Review the code and the comments below, then copy/paste it into the `lambda-target-example.js` file.

```
1   import { v4 as uuidv4 } from 'uuid';
2   import { Common  }  from "./common.js"
3
4   const sessionTable = process.env.SESSION_TABLE
5   const orderTable = process.env.ORDER_TABLE
6
7   export const handler = async (event) => {
8      console.log('testing Websocket event:', JSON.stringify(event));
9
10     // get request details
11     const { orderDetails, status, connectionId } = event.detail;
12
13     try {
14        // get session data
15        const sessionRecord = await Common.dynamodb_get("connectionId", connectionId, sessionTable);
16
17        if (!sessionRecord){
18            throw Error(`API integration failed ${connectionId} not found in ${sessionTable}`);
19        }
20
21        let currentRecord = {};
22
23        // check if order already exists
24        if (sessionRecord.orderId){
25
26          console.log("order already exists... updating")
27
28          // fetch order record
29          const orderRecord = await Common.dynamodb_get("orderId", sessionRecord.orderId, orderTable);
30
31          // update order with new details
32          currentRecord = { ...orderRecord, ...orderDetails, status }
33          await Common.dynamodb_write(currentRecord, orderTable);
34
35        } else {
36
37          const newOrderId = uuidv4();
38          console.log("order id: ", newOrderId);
39
40             // set session with order id
41          const sessionData = {
42              ...sessionRecord,
43              orderId : newOrderId
44          }
45
46          currentRecord = {
47              ...orderDetails,
48              connectionId:connectionId,
49              orderId:newOrderId,
50              domainName: sessionRecord.domainName,
51              stage: sessionRecord.stage,
52              status: status
53          }
54
55          // update session table with new order
56          await Common.dynamodb_write(sessionData, sessionTable);
57
58          // update order table with new order
59          await Common.dynamodb_write(currentRecord, orderTable);
60        }
61
62        // send websocket message
63        await Common.websocket_send(currentRecord.domainName,currentRecord.stage,connectionId,status);
64        return Common.http_ok({message: JSON.stringify(currentRecord)});
65
66    } catch(error) {
67        return Common.http_notfound({message: JSON.stringify({
68          message: 'invalid order',
69          err: error
70       })});
71    }
72  }
```

> Note: Event pattern that contains **DetailType=OrderStatusChangedEvent** and **Source=WebsocketEvent** will target the same Lambda function function above.

On **line (11)** we are using the ES6 destructuring assignment syntax to unpack values from the recieved event.

On **line (15)** the lambda function will fetch the details related to the existing connection.

For simplicity, in order to ascertain whether the current event is categorized as "UPDATE" or "NEW," we will place the "order_id" value in the temporary Sessions table. If the "order_id" value is not discovered in the Sessions table, a new order will be generated, and its ID will be established within the Session table. Moreover, a new order entry will be appended to the Orders Table. Conversely, if the "order_id" value is located in the Sessions table, we will only update the existing order with the new status.

Lines (24-35) responsible for updating the status of an order.

Lines (35-53) responsible for adding a new order.

WebSocket API in API Gateway has stages for managing deployments, custom domains for friendlier URLs, and connection IDs for uniquely identifying and managing individual WebSocket connections. These elements collectively contribute to the configuration and functionality of WebSocket communication in the context of API Gateway.

Line (63) uses the following WebSocket connection HTTP requests to send a callback message to a connected client. For enabling server callbacks, the **Stage**, **Domain** and **Connection ID** are necessary.

5. The provided code below includes utility functions featuring DynamoDB, HTTP, and WebSocket API calls, designed to be utilized by the `lambda-target-example.js` code. Review the code and the comments below then copy/paste it into the `common.js` file

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand, PutCommand} from "@aws-sdk/lib-dynamodb";
import { ApiGatewayManagementApiClient, PostToConnectionCommand } from "@aws-sdk/client-apigatewaymanagementapi";

const documentClient = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(documentClient);

// Get item from Dynamo db table
async function dynamodb_get(hash, value, TableName) {
    const command = new GetCommand({
        TableName,
        Key: {
            [hash] : value
        },
    });

    const response = await docClient.send(command);
    if (!response || !response.Item) {
        throw Error(`There was an error fetching the data for ID of ${ID} from ${TableName}`);
    }
    console.log(response);
    return response.Item;
}

// Write item to Dynamo db table
async function dynamodb_write(item, TableName) {

    const command = new PutCommand({
        TableName,
        Item: item
    });

    await docClient.send(command);
}

// Return 200 status code
function http_ok(data = {}) {
    return {
        headers: {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Methods': '*',
            'Access-Control-Allow-Origin': '*',
        },
        statusCode: 200,
        body: JSON.stringify(data),
    };
}

// Return 404 status code
function http_notfound(data = {}) {
    return {
        headers: {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Methods': '*',
            'Access-Control-Allow-Origin': '*',
        },
        statusCode: 404,
        body: JSON.stringify(data),
    };
}

// Sending message to a websocket connection
async function websocket_send (domainName, stage, connectionId, status) {

    const callback = `https://${domainName}/${stage}`;

    console.log(`sending connection preparing sending message ${status} to ${callback}`);

    const client = new ApiGatewayManagementApiClient({
       endpoint: callback,
    });

    const message = {
       name : "STATUS_CHANGED_EVENT",
       status: status
    }

    const messageEvent = {
         type: "CHANGED_EVENT",
         message : JSON.stringify(message)
    }

    const requestParams = {
       ConnectionId: connectionId,
       Data: JSON.stringify(messageEvent)
    };

    const command =  new PostToConnectionCommand(requestParams)

    console.log(`sending to ${connectionId} ...`);
    await client.send(command);
    console.log(`${status} sent`);
 };

export const Common = {
   http_ok,
   http_notfound,
   dynamodb_get,
   dynamodb_write,
   websocket_send
};
```

6. The lambda handler requires integrating the specified SDK libraries. Copy and paste the following code into the `package.json` file.

```
{
    "name": "webocket_sample",
    "version": "1.0.0",
    "description": "websocket sample es6 compatible",
    "type": "module",
    "dependencies": {
        "@aws-sdk/client-dynamodb": "^3.445.0",
        "@aws-sdk/lib-dynamodb": "^3.445.0",
        "uuid": "^9.0.1"
    }
```

Build the AWSome Pizza Bar with AWS SAM

Use AWS SAM to create an API Gateway WebSocket API with Amzon Step Function integration

1. Using AWS Cloud9 console, return to the root folder `module-5`.
2. This code belongs in your SAM template file `template.yaml`. Review the code and the comments below, then **copy/paste** it into the `template.yaml` file.

```yaml
1
2   AWSTemplateFormatVersion: '2010-09-09'
3   Transform: AWS::Serverless-2016-10-31
4   Description: >
5     module-5-WebSocket: SAM Template for using websocket with API Gateway.
6
7   Globals:
8     Function:
9       Timeout: 30
10      Runtime: nodejs18.x
11  Parameters:
12    ApiStageName:
13      Description: Name of WebSockets API stage
14      Type: String
15      Default: api
16
17  # Comment each resource section to explain usage
18  Resources:
19  ########################################################
20  #   Lambda function to be executed by eventbridge rule
21  ########################################################
22    LambdaPizzaHandler:
23      Type: AWS::Serverless::Function
24      Properties:
25        CodeUri: ./handlers
26        Handler: lambda-target-example.handler
27        Environment:
28          Variables:
29            SESSION_TABLE: !Sub ${AWS::StackName}-Sessions
30            ORDER_TABLE: !Sub ${AWS::StackName}-Orders
31        Policies:
32          - DynamoDBCrudPolicy:
33              TableName: !Sub ${AWS::StackName}-Sessions
34          - DynamoDBCrudPolicy:
35              TableName: !Sub ${AWS::StackName}-Orders
36          - Statement:
37              - Sid: ManageWebSocketConnectionsPolicy
38                Effect: Allow
39                Action:
40                  - execute-api:ManageConnections
41                Resource: !Sub
42                  - "arn:aws:execute-api:${AWS::Region}:${AWS::AccountId}:${API_ID}/api/POST/@connections/*"
43                  - API_ID: !Ref WebSocketApi
44
45    LambdaRule:
46      Type: AWS::Events::Rule
47      Properties:
48        EventBusName: !Ref PizzaOrderEventBus
49        Targets:
50          - Arn: !GetAtt LambdaPizzaHandler.Arn
51            Id: LambdaPizzaHandler
52        EventPattern:
53          source:
54            - WebsocketEvent
55          detail-type:
56            - OrderStatusChangedEvent
57
58    EventBridgeInvokeLambda:
59      Type: AWS::Lambda::Permission
60      Properties:
61        FunctionName: !Ref LambdaPizzaHandler
62        Action: "lambda:InvokeFunction"
63        Principal: "events.amazonaws.com"
64        SourceArn: !GetAtt LambdaRule.Arn
65
66
67
68  ########################################################
69  #   EventBridge bus
70  ########################################################
71
72    PizzaOrderEventBus:
73      Type: AWS::Events::EventBus
74      Properties:
75        Name: !Sub '${AWS::StackName}-EventBus'
76
77    WorkflowExecutionRole:
78      Type: AWS::IAM::Role
79      Properties:
80        AssumeRolePolicyDocument:
81          Version: 2012-10-17
82          Statement:
83            - Effect: Allow
84              Principal:
85                Service: states.amazonaws.com
86              Action: sts:AssumeRole
87        Policies:
88          - PolicyName: AllowEventBridgePutEvents
89            PolicyDocument:
90              Version: 2012-10-17
91              Statement:
92                - Effect: Allow
93                  Action: events:PutEvents
94                  Resource: !GetAtt PizzaOrderEventBus.Arn
95
96    MyLogGroup:
97      Type: AWS::Logs::LogGroup
98      Properties:
99        LogGroupName: !Sub '/aws/events/${AWS::StackName}'
100
101   EventBusLogRule:
102     Type: AWS::Events::Rule
103     Properties:
104       Description: Send all events to CloudWatch Logs
105       EventBusName: !Ref PizzaOrderEventBus
106       EventPattern:
107         source:
108           - prefix: ""
109       State: ENABLED
110       Targets:
111         - Arn: !Sub "arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-group:${MyLogGroup}"
112           Id: CloudWatchLogTarget
113     DependsOn: PizzaOrderEventBus
114
115
116 ########################################################
117 #   API Gateway WebSocket API
118 ########################################################
119   WebSocketApi:
120     Type: AWS::ApiGatewayV2::Api
121     Properties:
122       Name: !Sub "${AWS::StackName}-WebSocketApi"
123       Description: "API Gateway WebSocket API"
124       ProtocolType: WEBSOCKET
125       RouteSelectionExpression: "$request.body.action"
126
127   Deployment:
128     Type: AWS::ApiGatewayV2::Deployment
129     DependsOn:
130       - DefaultRoute
131     Properties:
132       ApiId: !Ref WebSocketApi
133
134   Stage:
135     Type: AWS::ApiGatewayV2::Stage
136     Properties:
137       StageName: !Ref ApiStageName
138       DeploymentId: !Ref Deployment
139       ApiId: !Ref WebSocketApi
140
141 ##########################################################################
142 #   Default route implementation including request and response
143 ##########################################################################
144
145   DefaultRoute:
146     Type: AWS::ApiGatewayV2::Route
147     Properties:
148       ApiId: !Ref WebSocketApi
149       RouteKey: $default
150       AuthorizationType: NONE
151       OperationName: DefaultRoute
152       Target: !Join
153         - /
154         - - integrations
155           - !Ref DefaultRouteIntegration
156
157   DefaultRouteIntegration:
158     Type: AWS::ApiGatewayV2::Integration
159     Properties:
160       ApiId: !Ref WebSocketApi
161       IntegrationType: AWS
162       IntegrationMethod: POST
163       IntegrationUri: !Sub "arn:aws:apigateway:${AWS::Region}:states:action/StartExecution"
164       CredentialsArn: !Sub "${StepFunctionsAsyncExecutionRole.Arn}"
165       TemplateSelectionExpression: \$default
166       RequestTemplates:
167         "$default" :
168           Fn::Sub: >
169             #set($sfn_input=$util.escapeJavaScript($input.body).replaceAll("\\'","'"))
170             {
171               "input": "{\"data\":$sfn_input, \"connectionId\":\"$context.connectionId\"}",
172               "stateMachineArn": "${APIGatewayStepFuncState}"
173             }
174
175   DefaultRouteResponse:
176     Type: AWS::ApiGatewayV2::RouteResponse
177     Properties:
178       RouteId: !Ref DefaultRoute
179       ApiId: !Ref WebSocketApi
180       RouteResponseKey: $default
181
182   DefaultRouteIntegrationResponse:
183     Type: AWS::ApiGatewayV2::IntegrationResponse
184     Properties:
185       ApiId: !Ref WebSocketApi
186       IntegrationId: !Ref DefaultRouteIntegration
187       IntegrationResponseKey: $default
188
189 ########################################################
190 #   Connect route implementation
191 ########################################################
192   ConnectRoute:
193     Type: AWS::ApiGatewayV2::Route
194     Properties:
195       ApiId: !Ref WebSocketApi
196       RouteKey: $connect
197       AuthorizationType: NONE
198       OperationName: ConnectRoute
199       Target: !Join
200         - /
201         - - integrations
202           - !Ref ConnectRouteIntegration
203
204   ConnectRouteIntegration:
205     Type: AWS::ApiGatewayV2::Integration
206     Properties:
207       ApiId: !Ref WebSocketApi
208       IntegrationType: AWS
209       IntegrationMethod: POST
210       IntegrationUri: !Sub "arn:aws:apigateway:${AWS::Region}:dynamodb:action/PutItem"
211       CredentialsArn: !Sub "${SessionsTableAccessRole.Arn}"
212       TemplateSelectionExpression: \$default
213       RequestTemplates:
214         "$default" :
215           Fn::Sub: >
216             #set($ttl = $context.requestTimeEpoch + 86400)
217             {
218                 "TableName": "${SessionsTable}",
219                 "Item": {
220                     "connectionId": {
221                         "S": "$context.connectionId"
222                     },
223                     "domainName": {
224                         "S": "$context.domainName"
225                     },
226                     "stage": {
227                         "S": "$context.stage"
228                     },
229                     "headers": {
230                         "S": "$input.params().get('header')"
231                     },
232                     "querystring": {
233                         "S": "$input.params().get('querystring')"
234                     },
235                     "ttl": {
236                         "N": "$ttl"
237                     },
238                     "orderId" : {
239                         "S": ""
240                     }
241                 }
242             }
243
244   ConnectRouteResponse:
245     Type: AWS::ApiGatewayV2::RouteResponse
246     Properties:
247       RouteId: !Ref ConnectRoute
248       ApiId: !Ref WebSocketApi
249       RouteResponseKey: $default
250
251   ConnectRouteIntegrationResponse:
252     Type: AWS::ApiGatewayV2::IntegrationResponse
253     Properties:
254       ApiId: !Ref WebSocketApi
255       IntegrationId: !Ref ConnectRouteIntegration
256       IntegrationResponseKey: /200/
257       TemplateSelectionExpression: \$default
258
259 ########################################################
260 #   Disconnect route implementation
261 ########################################################
262   DisconnectRoute:
263     Type: AWS::ApiGatewayV2::Route
264     Properties:
265       ApiId: !Ref WebSocketApi
266       RouteKey: $disconnect
267       AuthorizationType: NONE
268       OperationName: DisconnectRoute
269       Target: !Join
270         - /
271         - - integrations
272           - !Ref DisconnectRouteIntegration
273
274   DisconnectRouteIntegration:
275     Type: AWS::ApiGatewayV2::Integration
276     Properties:
277       ApiId: !Ref WebSocketApi
278       IntegrationType: AWS
279       IntegrationMethod: POST
280       IntegrationUri: !Sub "arn:aws:apigateway:${AWS::Region}:dynamodb:action/DeleteItem"
281       CredentialsArn: !Sub "${SessionsTableAccessRole.Arn}"
282       TemplateSelectionExpression: \$default
283       RequestTemplates:
284         "$default" :
285           Fn::Sub: >
286             {
287                 "TableName": "${SessionsTable}",
288                 "Key": {
289               "connectionId": {
290                         "S": "$context.connectionId"
291                         }
292                 }
293             }
294
295   DisconnectRouteResponse:
296     Type: AWS::ApiGatewayV2::RouteResponse
297     Properties:
298       RouteId: !Ref DisconnectRoute
299       ApiId: !Ref WebSocketApi
300       RouteResponseKey: $default
301
302   DisconnectRouteIntegrationResponse:
303     Type: AWS::ApiGatewayV2::IntegrationResponse
304     Properties:
305       ApiId: !Ref WebSocketApi
306       IntegrationId: !Ref DisconnectRouteIntegration
307       IntegrationResponseKey: /200/
308       TemplateSelectionExpression: \$default
309
310 ########################################################
311 #   order storage
312 ########################################################
313   OrderTable:
314     Type: AWS::DynamoDB::Table
315     Properties:
316       TableName: !Sub  ${AWS::StackName}-Orders
317       BillingMode: PAY_PER_REQUEST
318       AttributeDefinitions:
319         - AttributeName: orderId
320           AttributeType: S
321       KeySchema:
322         - AttributeName: orderId
323           KeyType: HASH
324       StreamSpecification:
325         StreamViewType: NEW_AND_OLD_IMAGES
326
327 ########################################################
328 #   Data storage for connection tracking
329 ########################################################
330   SessionsTable:
331     Type: AWS::DynamoDB::Table
332     Properties:
333       TableName: !Sub  ${AWS::StackName}-Sessions
334       AttributeDefinitions:
335         - AttributeName: connectionId
336           AttributeType: S
337       KeySchema:
338         - AttributeName: connectionId
339           KeyType: HASH
340       TimeToLiveSpecification:
341           AttributeName: ttl
342           Enabled: true
343       BillingMode: PAY_PER_REQUEST
344
345   SessionsTableAccessRole:
346     Type: AWS::IAM::Role
347     Properties:
348       AssumeRolePolicyDocument:
349         Version: "2012-10-17"
350         Statement:
351           - Action:
352               - "sts:AssumeRole"
353             Effect: Allow
354             Principal:
355               Service:
356                 - apigateway.amazonaws.com
357       Path: /
358       Policies:
359         - PolicyName: DynamoDBAccess
360           PolicyDocument:
361             Version: "2012-10-17"
362             Statement:
363               - Effect: Allow
364                 Action:
365                   - "dynamodb:DeleteItem"
366                   - "dynamodb:PutItem"
367                 Resource:
368                   - !Sub "arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${SessionsTable}"
369
370
371
372 ########################################################
373 #   Step Functions Standard for asynchronous execution
374 ########################################################
375   APIGatewayStepFuncState:
376     Type: AWS::Serverless::StateMachine
377     Properties:
378       Type: STANDARD
379       DefinitionUri: statemachine/api-gw-websocket-asl.json
380       DefinitionSubstitutions:
381         APIEndPoint: !Sub "${WebSocketApi}.execute-api.${AWS::Region}.amazonaws.com"
382         PizzaOrderEventBus: !Sub '${AWS::StackName}-EventBus'
383       Role: !Sub "${StepFuncStateRole.Arn}"
384       Tracing:
385         Enabled: true
386
387   StepFuncStateRole:
388     Type: AWS::IAM::Role
389     Properties:
390       AssumeRolePolicyDocument:
391         Version: "2012-10-17"
392         Statement:
393           - Action:
394               - "sts:AssumeRole"
395             Effect: Allow
396             Principal:
397               Service:
398                 - states.amazonaws.com
399       Path: /
400       Policies:
401         - PolicyName: AllowEventBridgePutEvents
402           PolicyDocument:
403             Version: 2012-10-17
404             Statement:
405               - Effect: Allow
406                 Action: events:PutEvents
407                 Resource: !GetAtt PizzaOrderEventBus.Arn
408
409
410   StepFunctionsAsyncExecutionRole:
411     Type: AWS::IAM::Role
412     Properties:
413       AssumeRolePolicyDocument:
414         Version: "2012-10-17"
415         Statement:
416           - Action:
417               - "sts:AssumeRole"
418             Effect: Allow
419             Principal:
420               Service:
421                 - apigateway.amazonaws.com
422       Path: /
423       Policies:
424         - PolicyName: StepFunctionsAsyncExecution
425           PolicyDocument:
426             Version: "2012-10-17"
427             Statement:
428               - Effect: Allow
429                 Action:
430                   - "states:StartExecution"
431                 Resource: !Ref APIGatewayStepFuncState
432
433 # List all common outputs for usage
434 Outputs:
435   APIEndpoint:
436     Description: "API Gateway WebSocket endpoint URL"
437     Value: !Sub "wss://${WebSocketApi}.execute-api.${AWS::Region}.amazonaws.com/${ApiStageName}"
438   SessionsTable:
439     Description: "DynamoDB sessions table for WebSocket API connection ID tracking"
440     Value: !Ref SessionsTable
```

The above template contains a mixture of both AWS SAM and CloudFormation resources. As SAM templates are an extension of CloudFormation, its possible to mix resources from both types as upon deployment, the SAM resources are converted to CloudFormation resources.

1. The LambdaPizzaHandler, spanning **lines (22-43)**, creates the Lambda target function that is triggered when the event pattern matches an Amazon EventBridge rule. **Lines 32-35** are used to link an IAM Execution role, providing the Lambda function with CRUD operations authority over the DynamoDB Tables. On line 36 we add a policy which configures IAM Authorization access control and sets the required permissions for AWS Lambda to send messages over the websocket connection.
2. WorkflowExecutionRole, spanning **lines (77-94)** responsible for creating an IAM role for our State machine. The service principal who assumes the role is allowed to put events to the specified Amazon EventBridge resource.
3. WebSocketApi, spanning **lines (119-125)** configures the WebSocket API endpoint. There are three predefined routes that can be used: $connect, $disconnect, and $default.

- API Gateway calls the $default route if the route selection expression cannot be evaluated against the message or if no matching route is found.
- API Gateway calls the $connect route when a persistent connection between the client and a WebSocket API is being initiated.
- API Gateway calls the $disconnect route when the client or the server disconnects from the API.

4. **Lines (145-155)** define the $default route. We are setting up a WebSocket API integration request in API Gateway with AWS Step Functions as the backend integration endpoint. The incoming client request is transformed before it is sent to the AWS Step Functions backend integration using a Request Templates. The Request Template defines the following variables:

- The "data" variable which contains the user's request payload.
- The "connectionId" variable which contains the current WebSocket Connection Id.

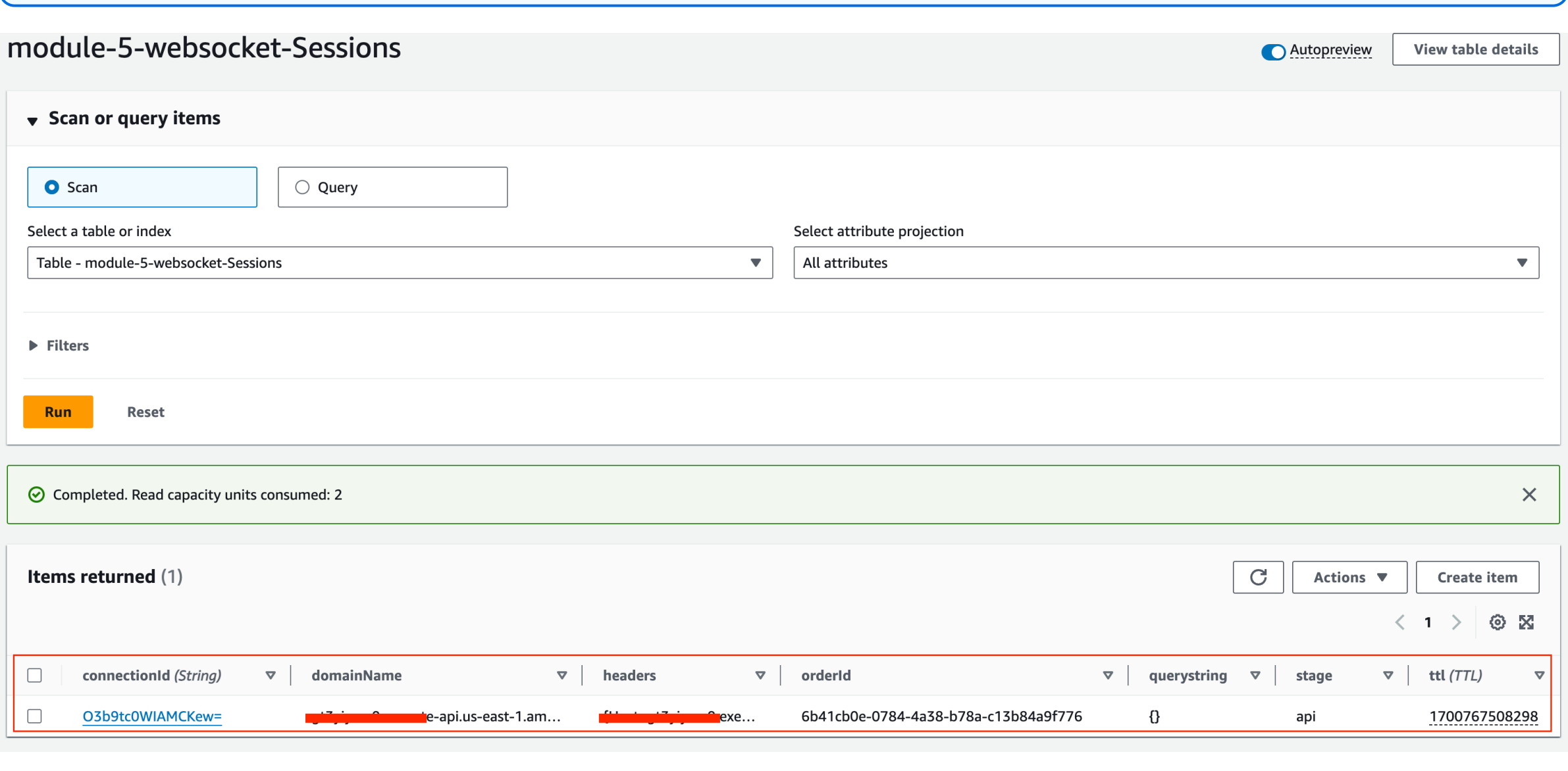
> Notice how the the variables above are then accessed by the AWS Step Function backend integration.

5. **Lines (192-202)** define the $connect route. We are setting up a WebSocket API integration request in API Gateway with DynamoDB as the backend integration endpoint.

The Request Template defines the following variables:

- "connectionId": A connection ID is a unique identifier assigned to each WebSocket connection established with your API Gateway WebSocket API.
- "domainName": In the context of WebSocket APIs, a domain refers to the custom domain name that you can associate with your API Gateway WebSocket API.
- "stage": A stage in Amazon API Gateway represents a named deployment of an API.
- "headers": Represents the value of the header parameter from the incoming request.
- "queryString": Represents the value of the query string parameter from the incoming request.
- "ttl": Represents a calculated value for the "time to live" (TTL), calculated as the current request time epoch plus 86400 seconds (24 hours).
- "orderId": Represents a random GUID generated for a new Pizza order.

> We will demonstrate how the DynamoDB Sessions table is populated when a new WebSocket connection is created in the next section.



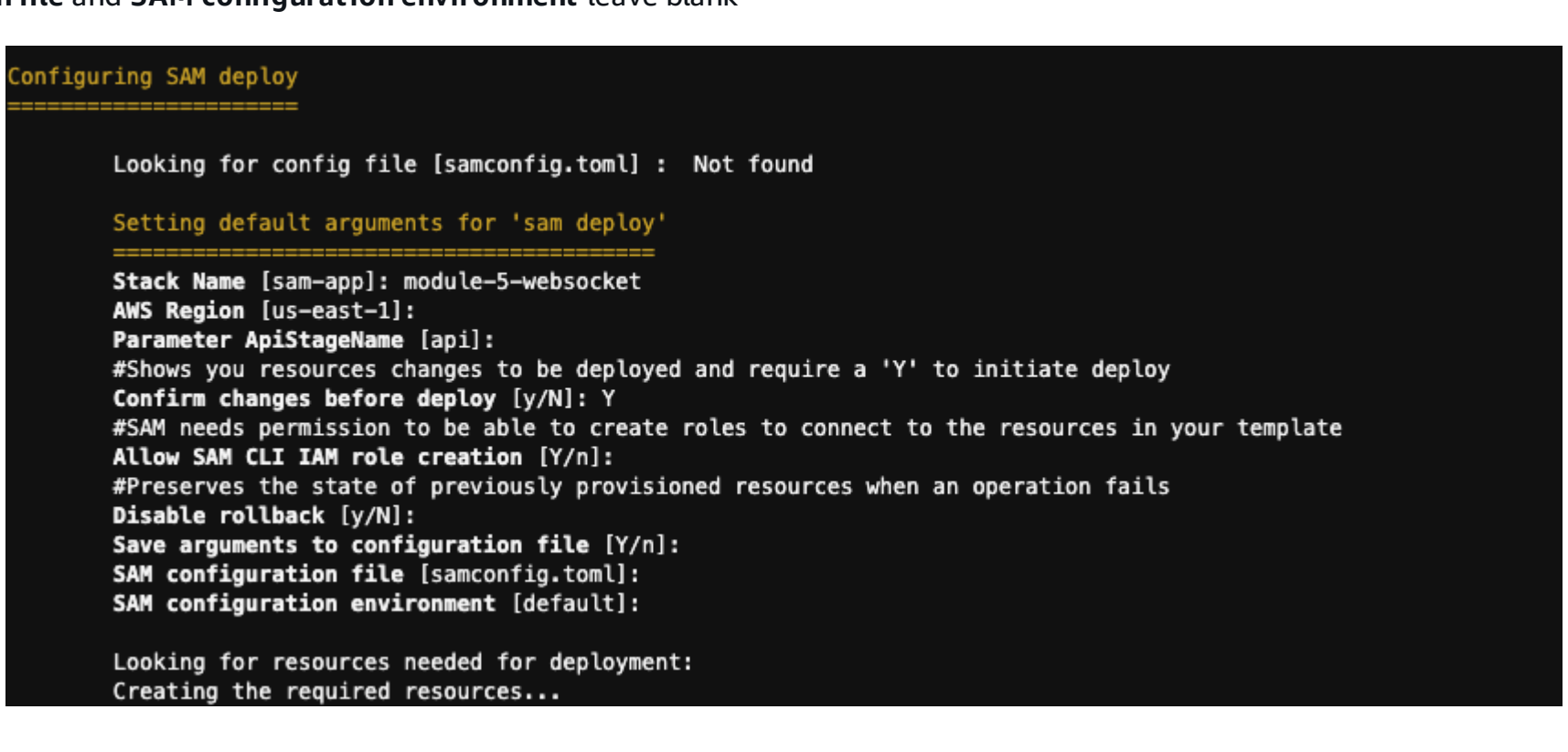
Deploy the project

1. To deploy the API Gateway WebSocket module to your AWS account, run the following commands from the application root `module-5`, where the `template.yaml` file for the sample application is located:

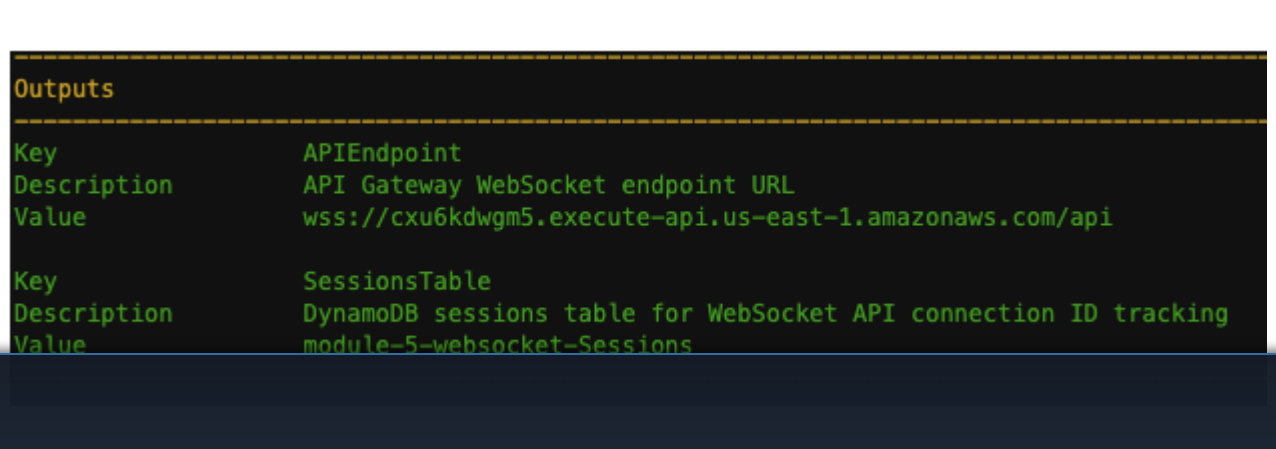
```
cd ~/environment/module-5
sam build && sam deploy --guided
```

The first time that you run the `sam deploy --guided` command, AWS SAM starts an AWS CloudFormation deployment. In this case, you need to specify the configurations that you want SAM to have in order to get the guided deployment. Please configure it as below:

- **Stack Name**: `module-5-websocket`
- **AWS Region**: Put the chosen region to run the workshop. **e.g.** `us-east-1`
- **Confirm changes before deploy**: `Y`
- **Allow SAM CLI IAM role creation**: `Y`
- **Disable rollback**: `N`
- **Save arguments to configuration file**: `Y`
- **SAM configuration file** and **SAM configuration environment** leave blank



2. After configuring the deployment, AWS SAM will display assets that will be created. But first, it will automatically upload the template to a temporary S3 bucket it creates. Then, it will ask you to confirm the changes. Type `y` to confirm.
3. Once the deployment has been successful, you will see an **'Outputs'** section that contains the API Gateway WebSocket invoke URL and the DynamoDB Session table that will be populated on every new WebSocket connections.



The Amazon API Gateway Workshop

Introduction
Getting Started
Module 1 - Introduction to Amazon API Gateway
Module 2 - Deploy your first API with IaC
Module 3 - API Gateway REST Integrations
Module 4 - Observability in API Gateway
Module 5 - WebSocket APIs
Module Goals
Set up your AWS SAM Project
Build the AWSome Pizza Bar with AWS SAM
Test the backend using a WebSocket client
Module 6 - Enable fine-grained access control for your APIs
Clean up
Resources

Test the backend using a WebSocket client

Test the deployed API using wscat

We are going to use wscat as our CLI tool for communicating with the API Gateway using the WebSocket protocol.

1. Navigate to Cloud9 in your AWS console and run the following command to install the wscat package.

```
npm install -g wscat
```

2. Make sure it's properly installed:

```
wscat -V
```

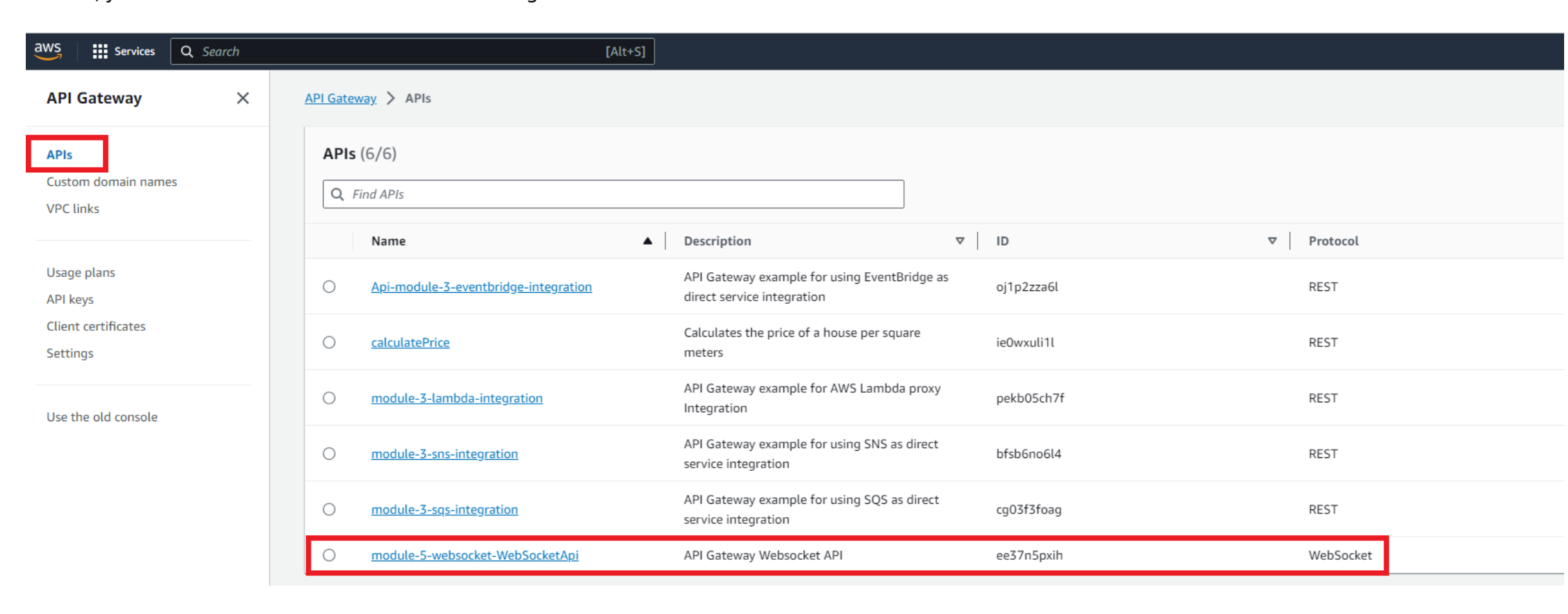
3. Now, we want to connect to our API endpoint. Copy the following wscat command and paste it into the terminal window, replacing `<api-id>` with your API ID and `<region>` with the region where your API is deployed. You may have copied this URL from the CloudFormation output in the previous step.

```
wscat -c "wss://<api-id>.execute-api.<region>.amazonaws.com/api"
```

The result should look like this:

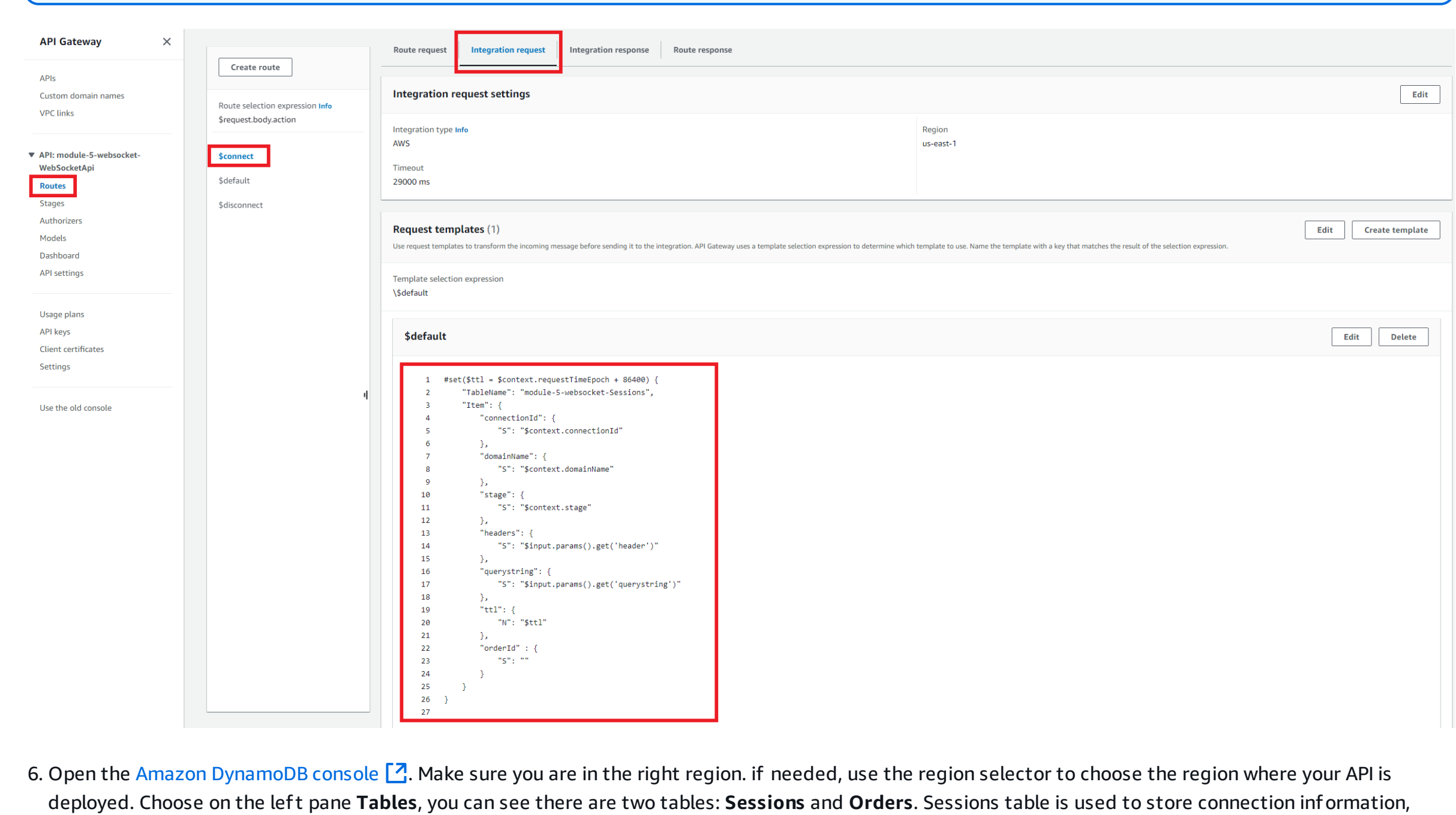
The client (wscat) is now connected to its backend API.

4. Let's review now the API Gateway **$connect** route. When a new connection is establish, the API Gateway $connect route is invoked. We have configured the integration request to insert a new connection entry in the DynamoDB Sessions table for persisting the connection. Open the Amazon API Gateway console and sign in. Make sure you are in the right region. if needed, use the region selector to choose the region where your API is deployed. Choose on the left pane **APIs**, you can see there our WebSocket API starting with module-5-XXXXXXX.

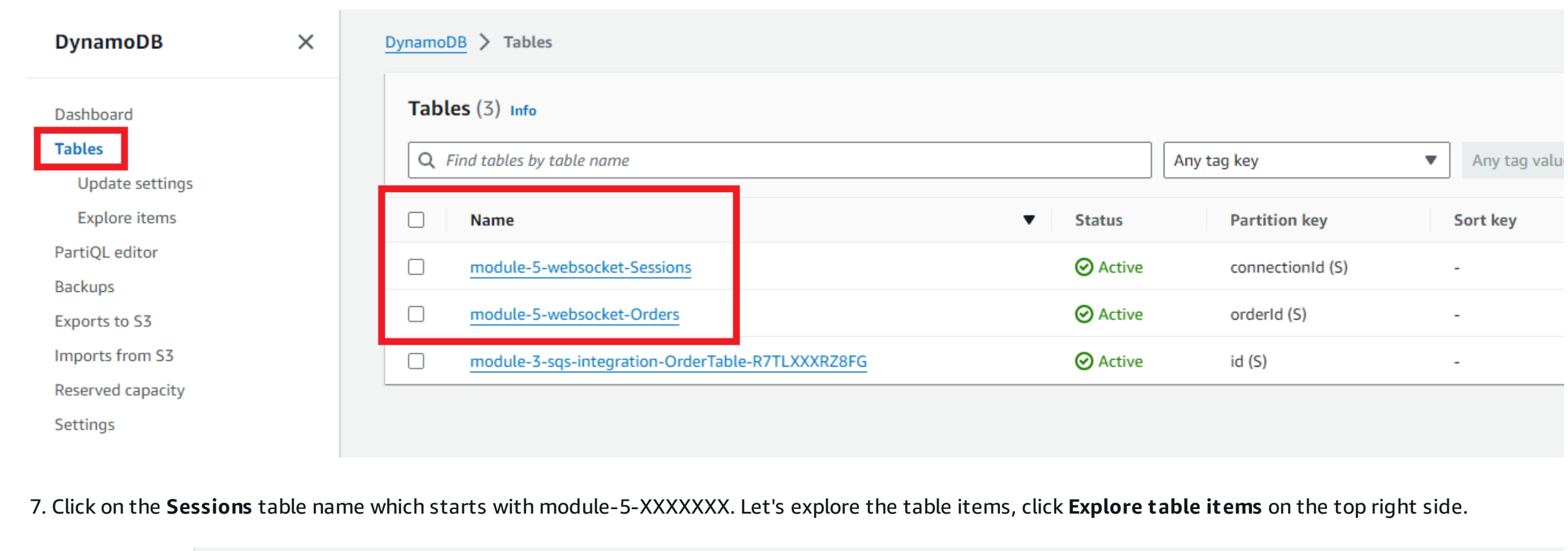


5. Click on the API starting with module-5-XXXXXXX. Click on the **$connect** route and choose the **Integration request** tab. Our API is directly integrated with DynamoDB, by using the mapping template we are writing a new entry to the Sessions table in DynamoDB. Some of these key-value entries (like connectionId) will be used by our Lambda function to process the Pizza order and send notifications from the server to the client.

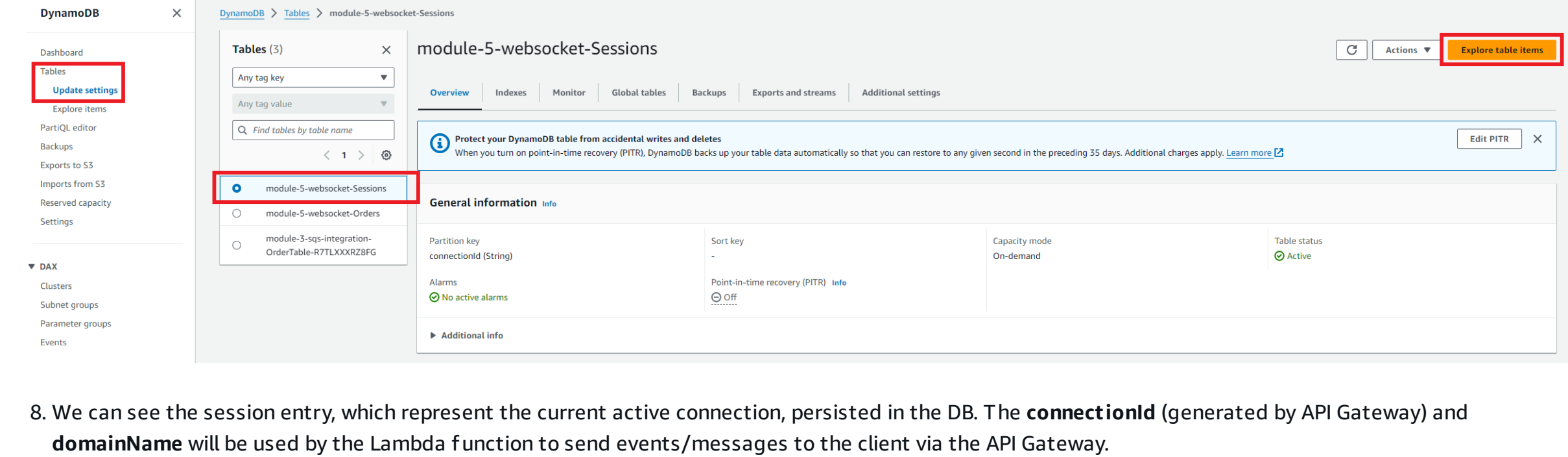
> **Tip**
> If you want to learn more on API Gateway and DynamoDB integration, refer to "Module 3 - API Gateway REST Integrations: Amazon DynamoDB".



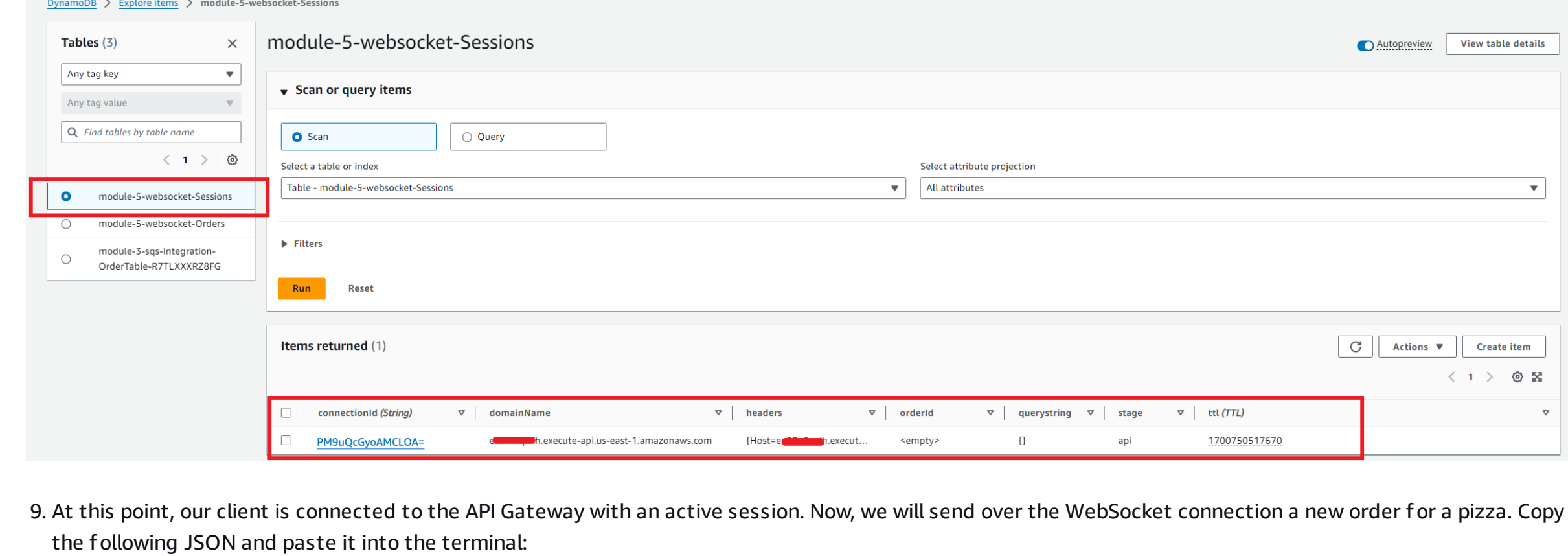
6. Open the Amazon DynamoDB console. Make sure you are in the right region. if needed, use the region selector to choose the region where your API is deployed. Choose on the left pane **Tables**, you can see there are two tables: **Sessions** and **Orders**. Sessions table is used to store connection information, while Orders table is used to store Pizza Order information.



7. Click on the **Sessions** table name which starts with module-5-XXXXXXX. Let's explore the table items, click **Explore table items** on the top right side.



8. We can see the session entry, which represent the current active connection, persisted in the DB. The **connectionId** (generated by API Gateway) and **domainName** will be used by the Lambda function to send events/messages to the client via the API Gateway.



9. At this point, our client is connected to the API Gateway with an active session. Now, we will send over the WebSocket connection a new order for a pizza. Copy the following JSON and paste it into the terminal:

```
{"order" : "pizza"}
```

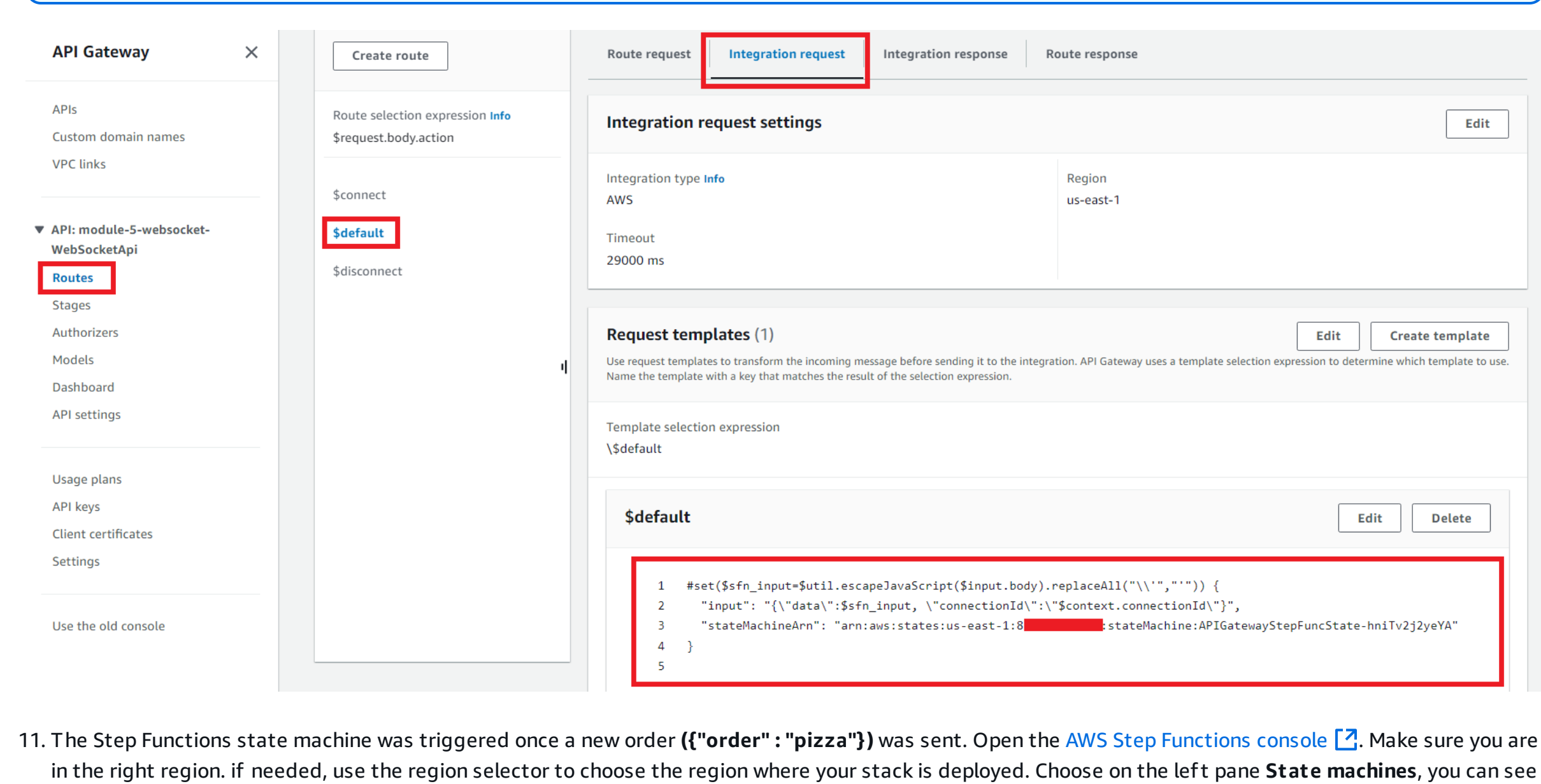
> **Tip**
> In case the connection was closed or timed out, you can connect back as specified on step 3.

Sending this JSON will invoke on the backend the processing of the order. API Gateway will invoke the Step Function state machine, that is orchestrating the process.

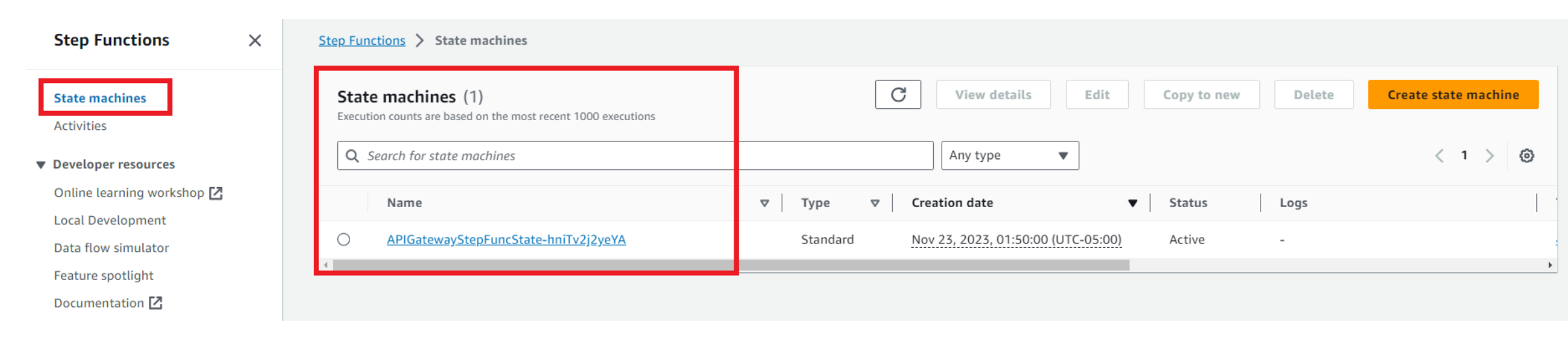
As you can see, the backend is sending the client every 5 seconds a state notification for the order. There are total of 6 events, for each a notification is sent.

10. Now, let's explore the flow in the backend. Let's review now the API Gateway $default route. When date is sent over the WebSocket connection, the API Gateway $default route is invoked. We have configured the integration request to trigger our Step Functions state machine using a direct integration. On the left pane, choose **Routes**, and on the routes list choose **$default**. Choose the **Integration request** tab. Our Api is directly integrated with Step Functions, and using the **request template - $default** we are passing required info to it like the connection ID and the JSON data.

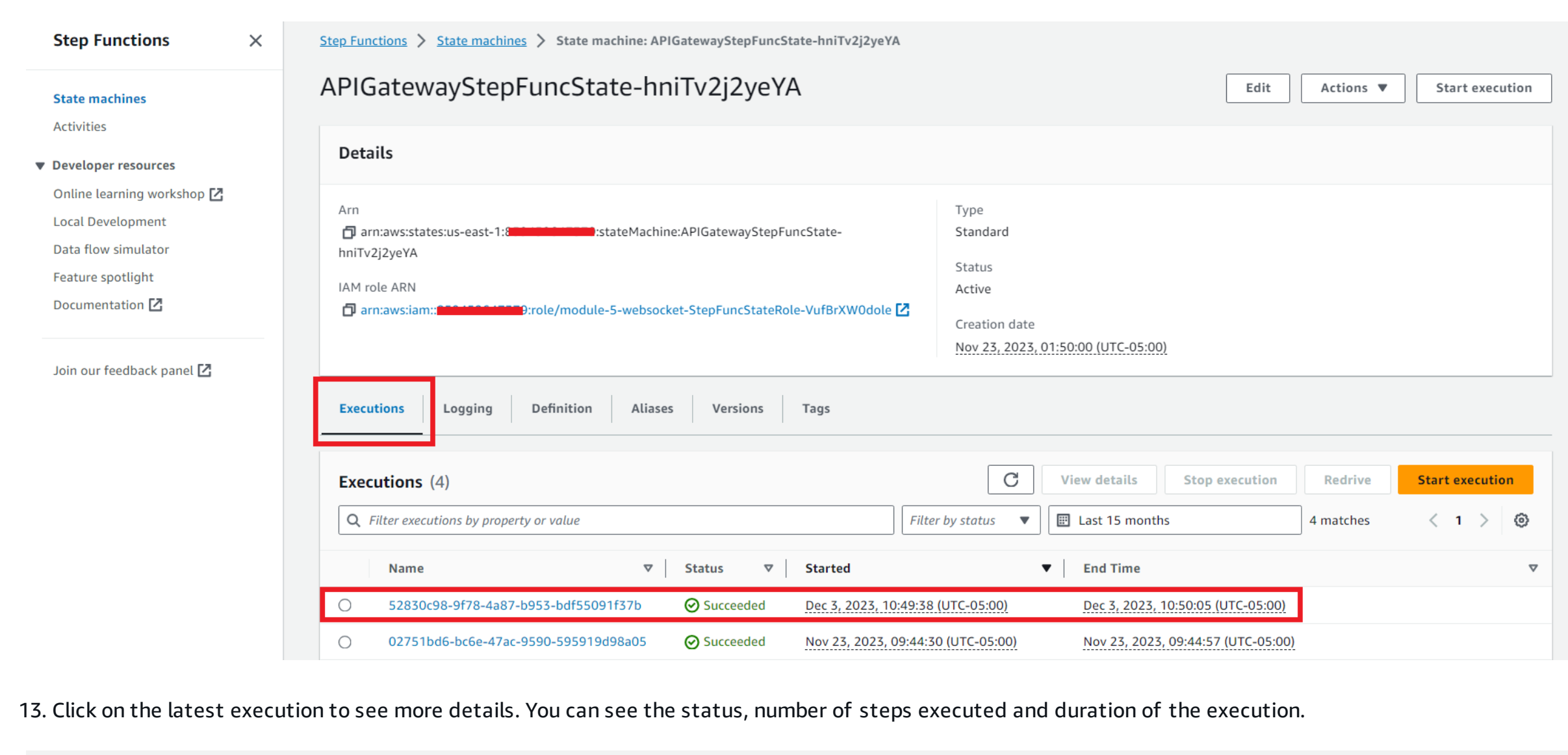
> **Tip**
> If you want to learn more on API Gateway and Step Functions integration, refer to "Module 3 - API Gateway REST Integrations: AWS Step Functions".



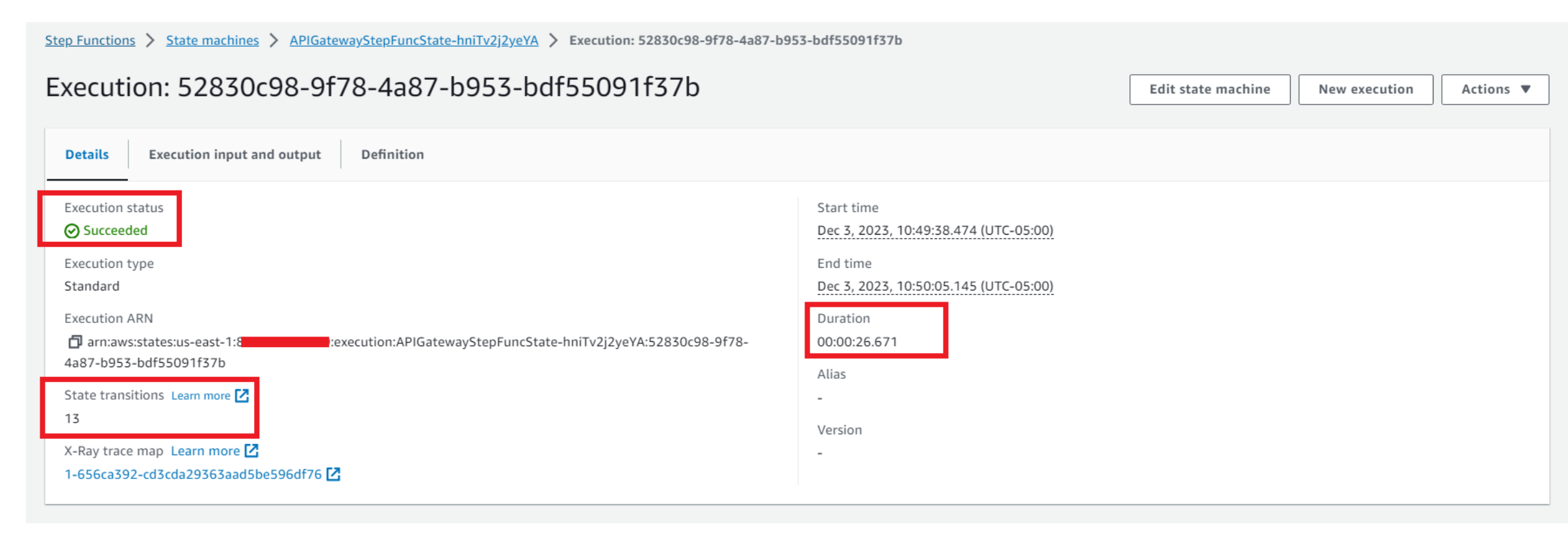
11. The Step Functions state machine was triggered once a new order **({"order" : "pizza"})** was sent. Open the AWS Step Functions console. Make sure you are in the right region. if needed, use the region selector to choose the region where your stack is deployed. Choose on the left pane **State machines**, you can see there our state machine starting with APIGatewayStepFuncState-XXXXXXX.



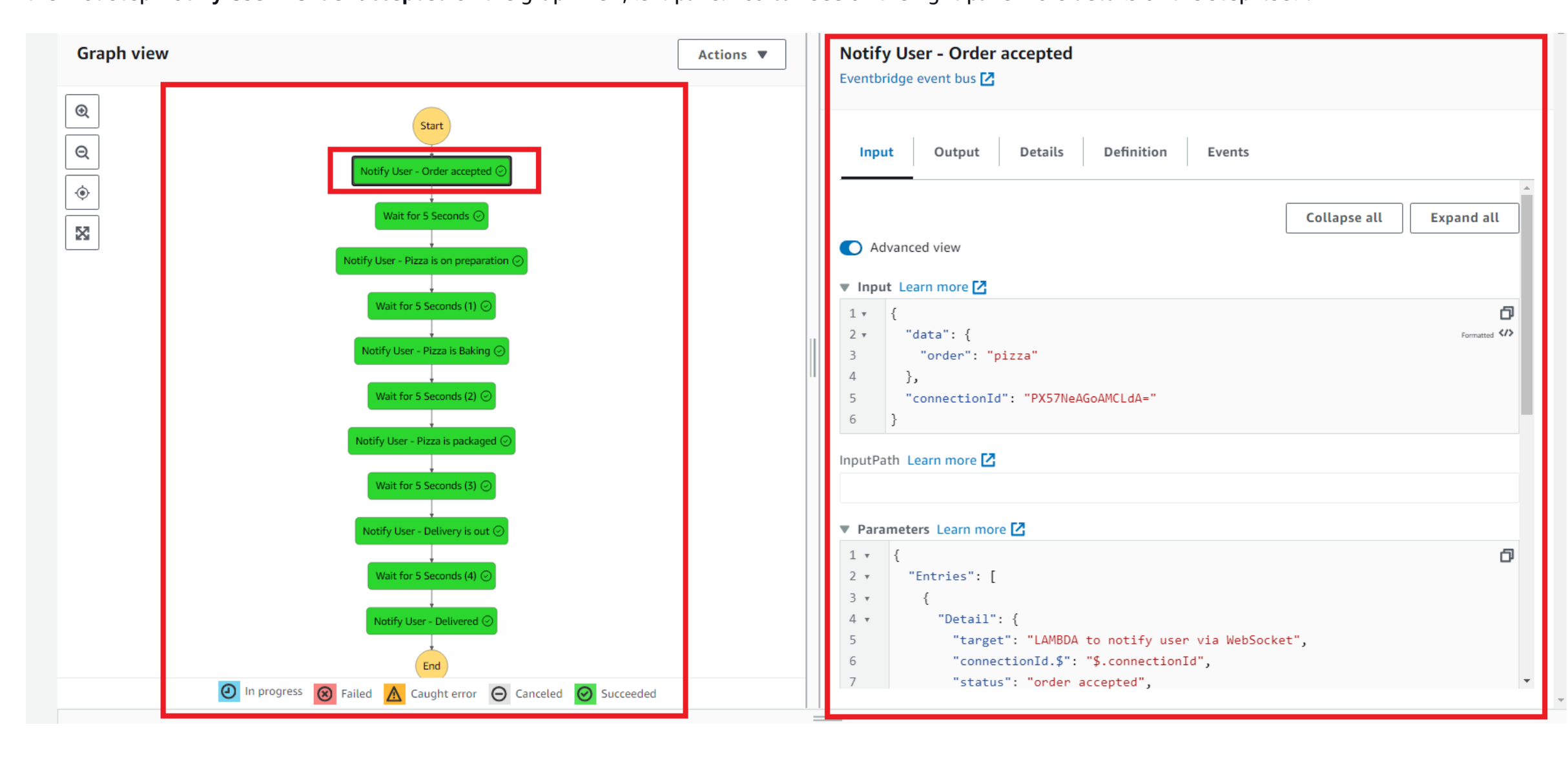
12. Click on the state machine to view more details. Below the **Executions** tab you should be able to see at least one successful invocation of it. Every pizza order will create a new execution.



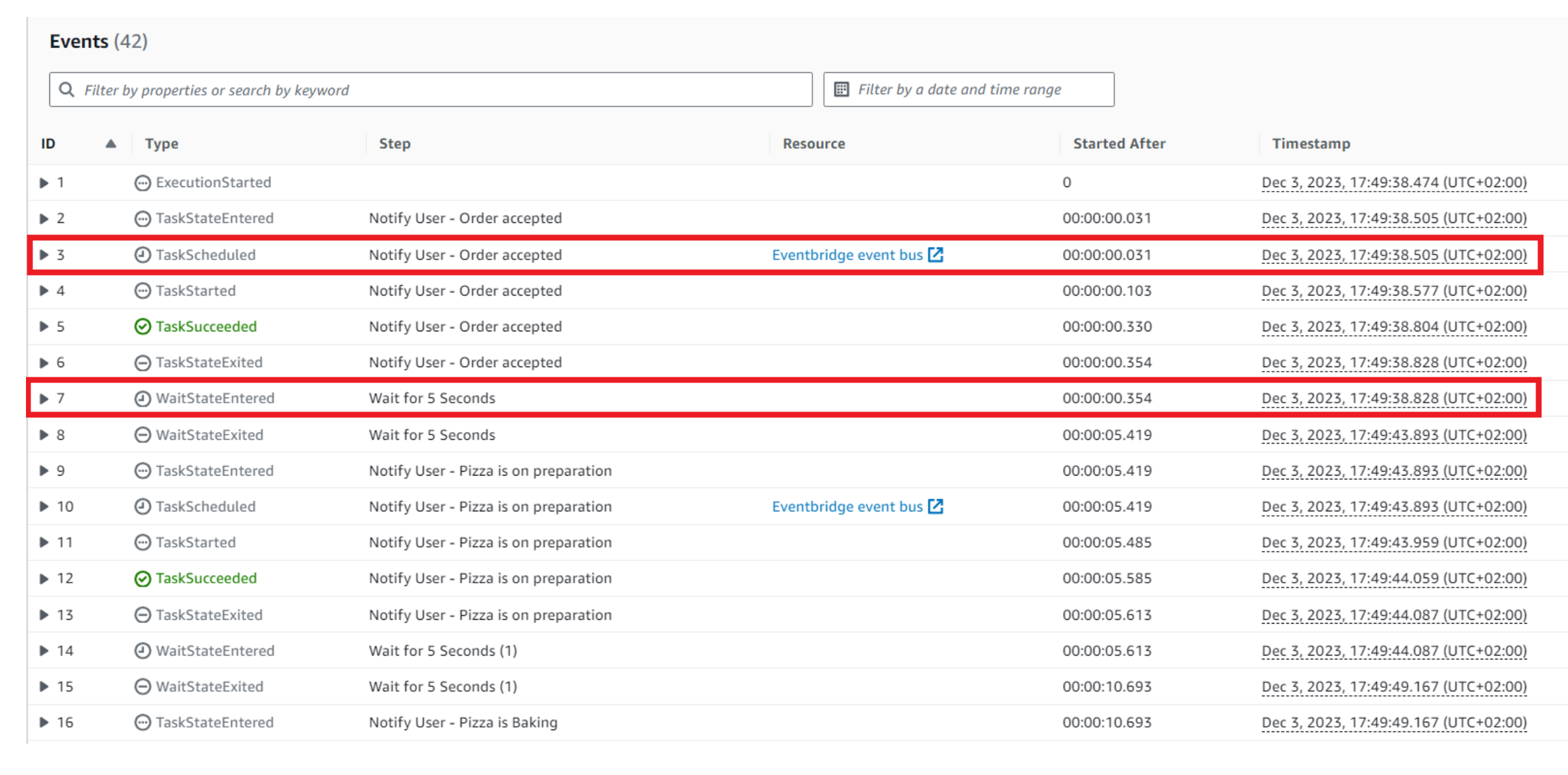
13. Click on the latest execution to see more details. You can see the status, number of steps executed and duration of the execution.



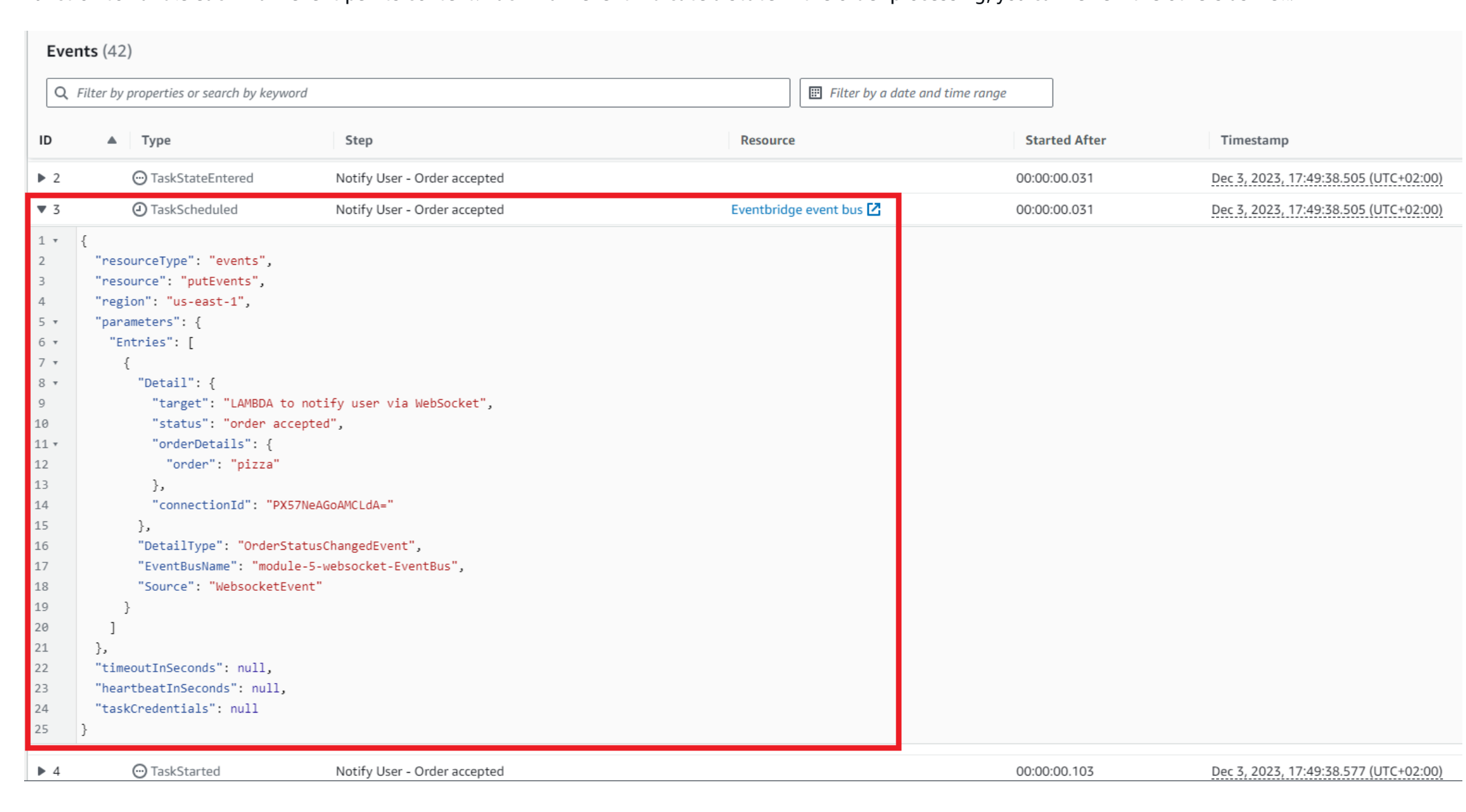
Scroll down, you can see the state machine steps that were executed. They are colored green, it indicates that all of them were successfully executed. Click on the first step **Notify User - Order accepted** on the graph view, left pane. You can see on the right pane more details of the step itself.



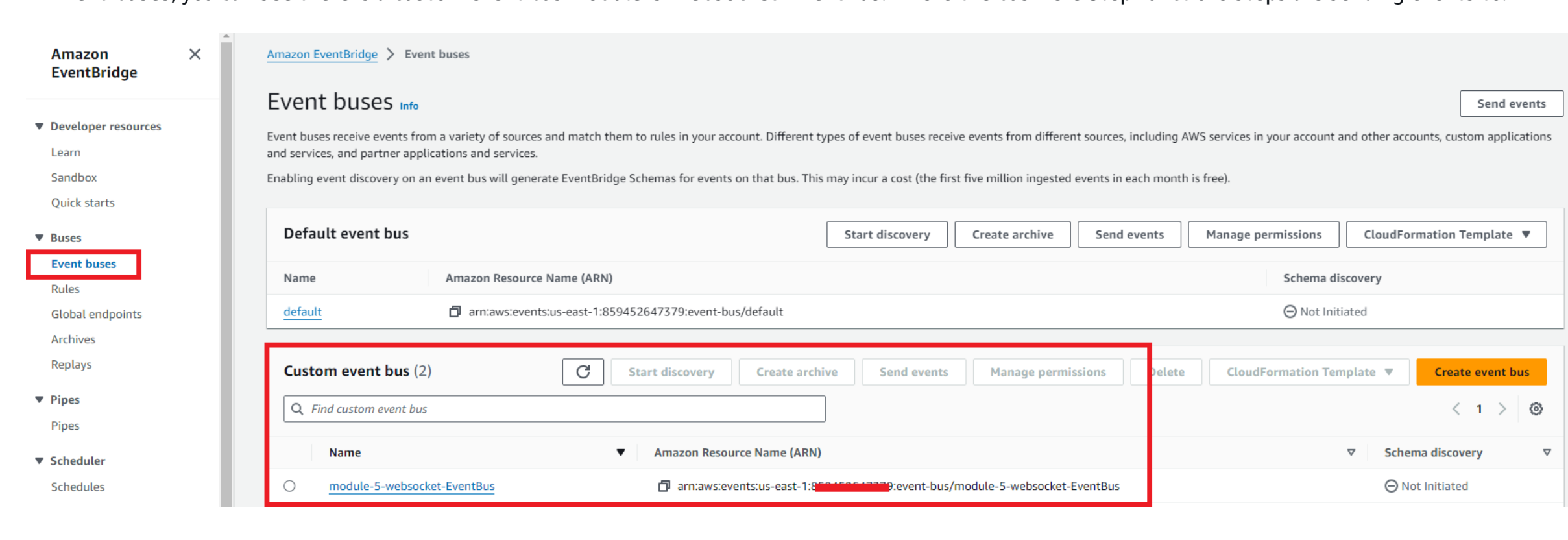
Scroll down to the **Events** section. here you can see all of the events that Step Functions processed. The pizza order is processed in 6 main steps, and between each one of them we have a **wait** state for 5 seconds. Each main step is sending an event to an **Eventbridge bus**, which in turn triggers a Lambda function.



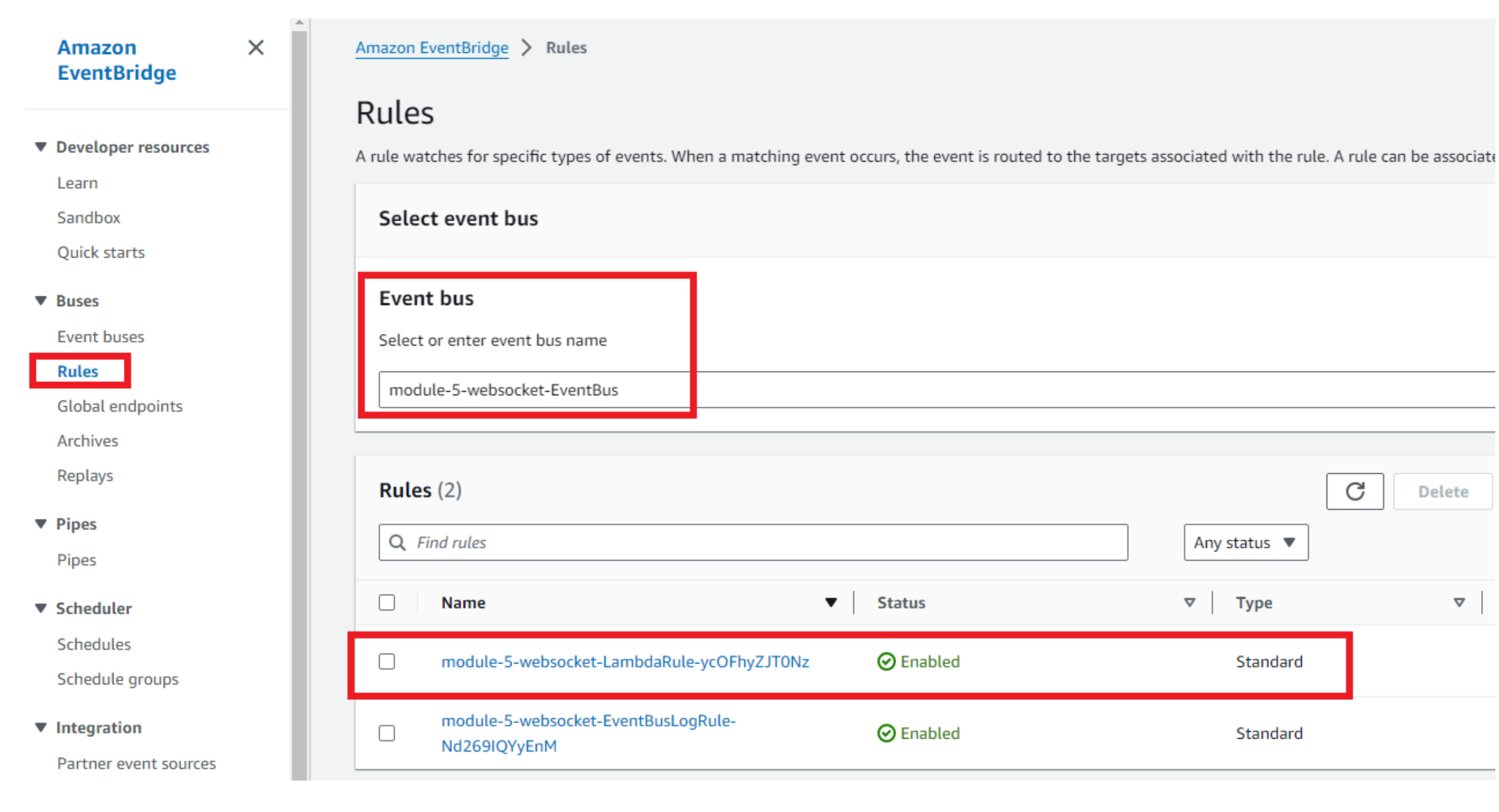
Expend the **Notify User - Order accepted** step. This is the JSON input that is passed as a context in for an Eventbridge event and is used by the Lambda function to handle each main event per its context. Each main event indicate a state in the order processing, you can review the others as well.



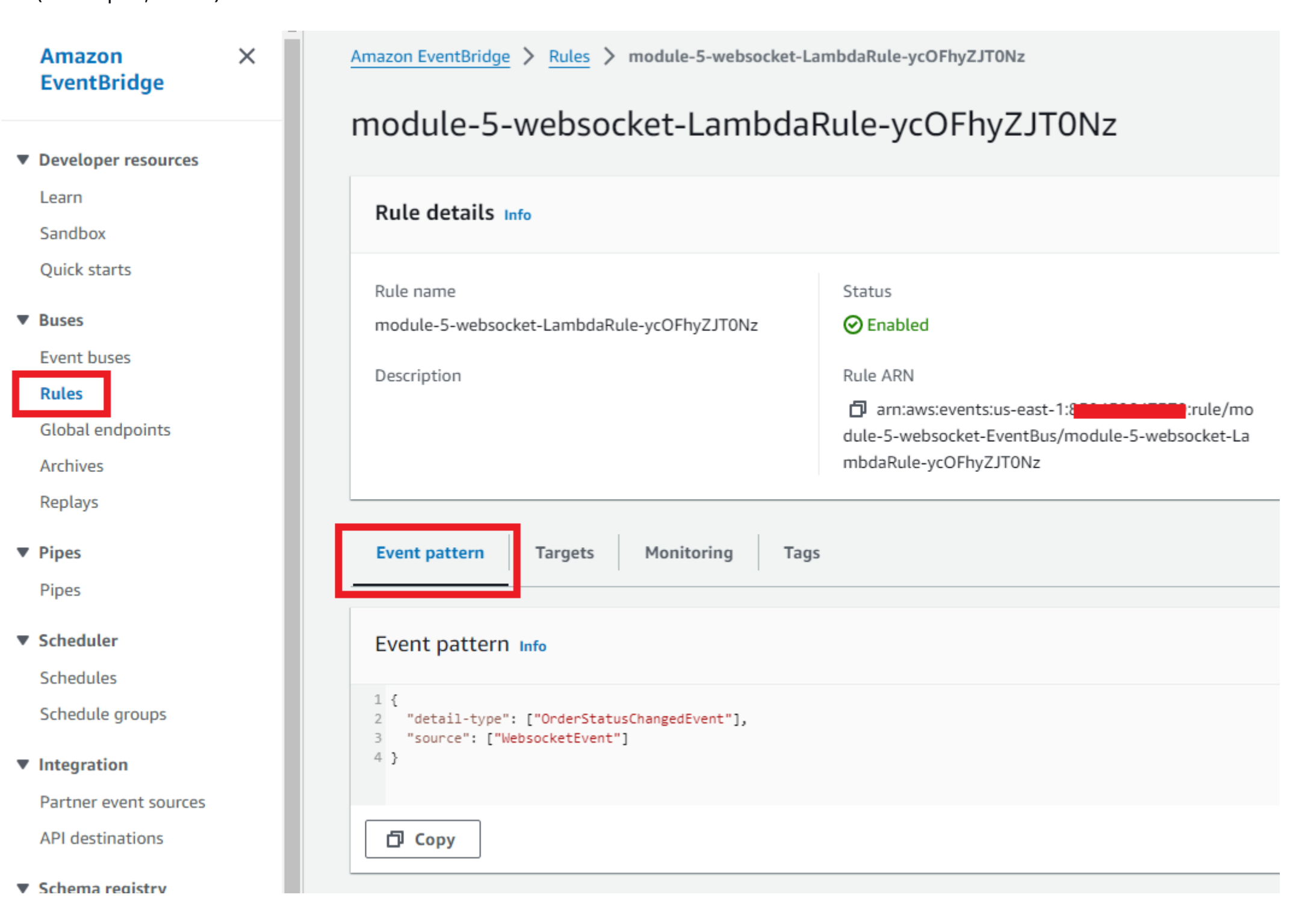
14. Click on the **Eventbridge event bus** link in the resource column, a new windows will be opened with the Eventbridge console page. On the left pane choose **Event buses**, you can see there is a custom event bus **module-5-websocket-EventBus**. This is the bus were Step Functions steps are sending events to.



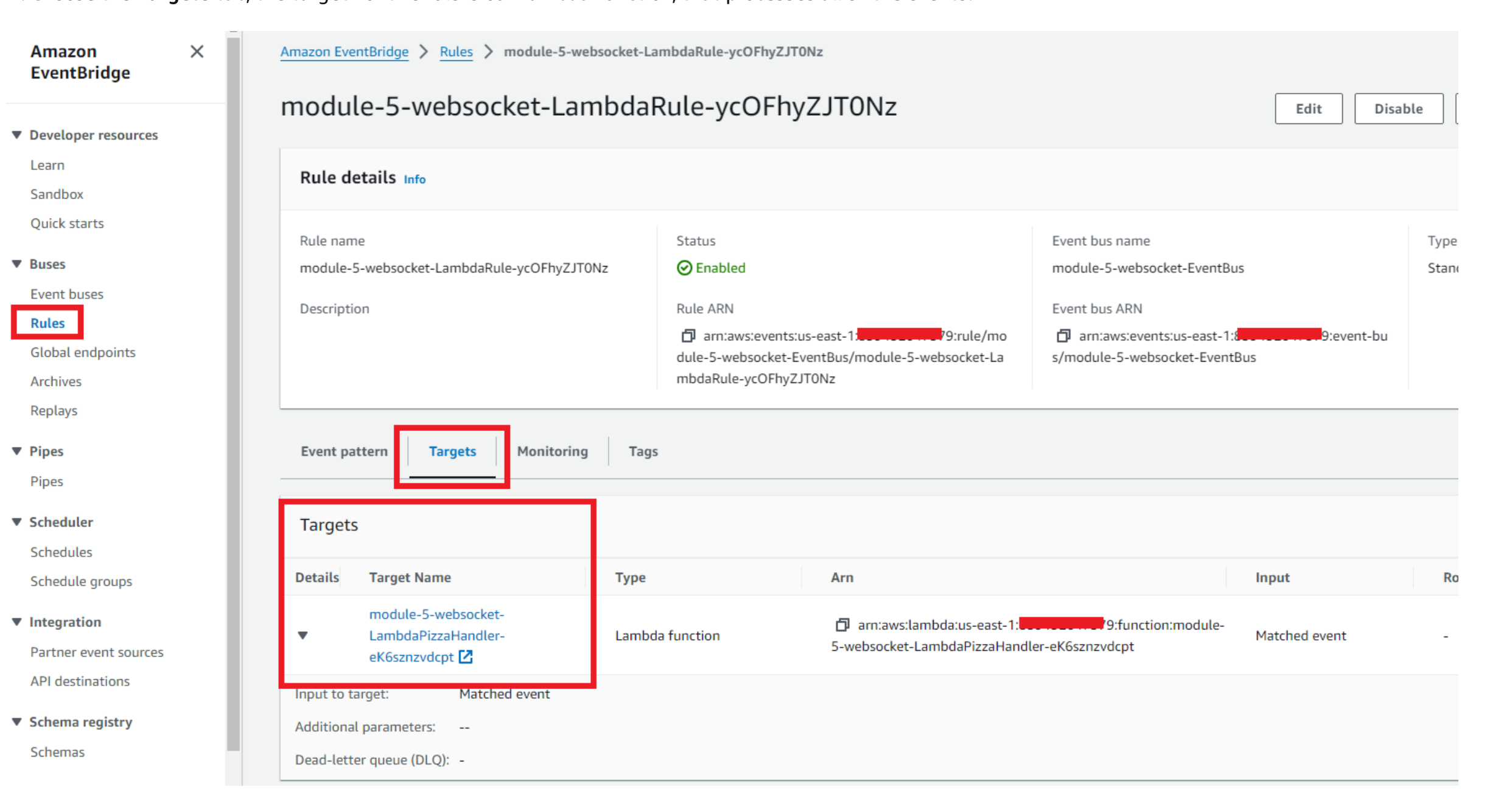
15. Click on **Rules** on the left pane. Choose **module-5-websocket-EventBus** in the drop down selection of **Event bus**. There are two rules for the bus: **module-5-websocket-EventBusLogRule-XXXXXXX** is used for logging to CloudWatch and **module-5-websocket-LambdaRule-XXXXXXX** which is used to match events from Step Functions. Click on the **module-5-websocket-LambdaRule-XXXXXXX** rule to review it.



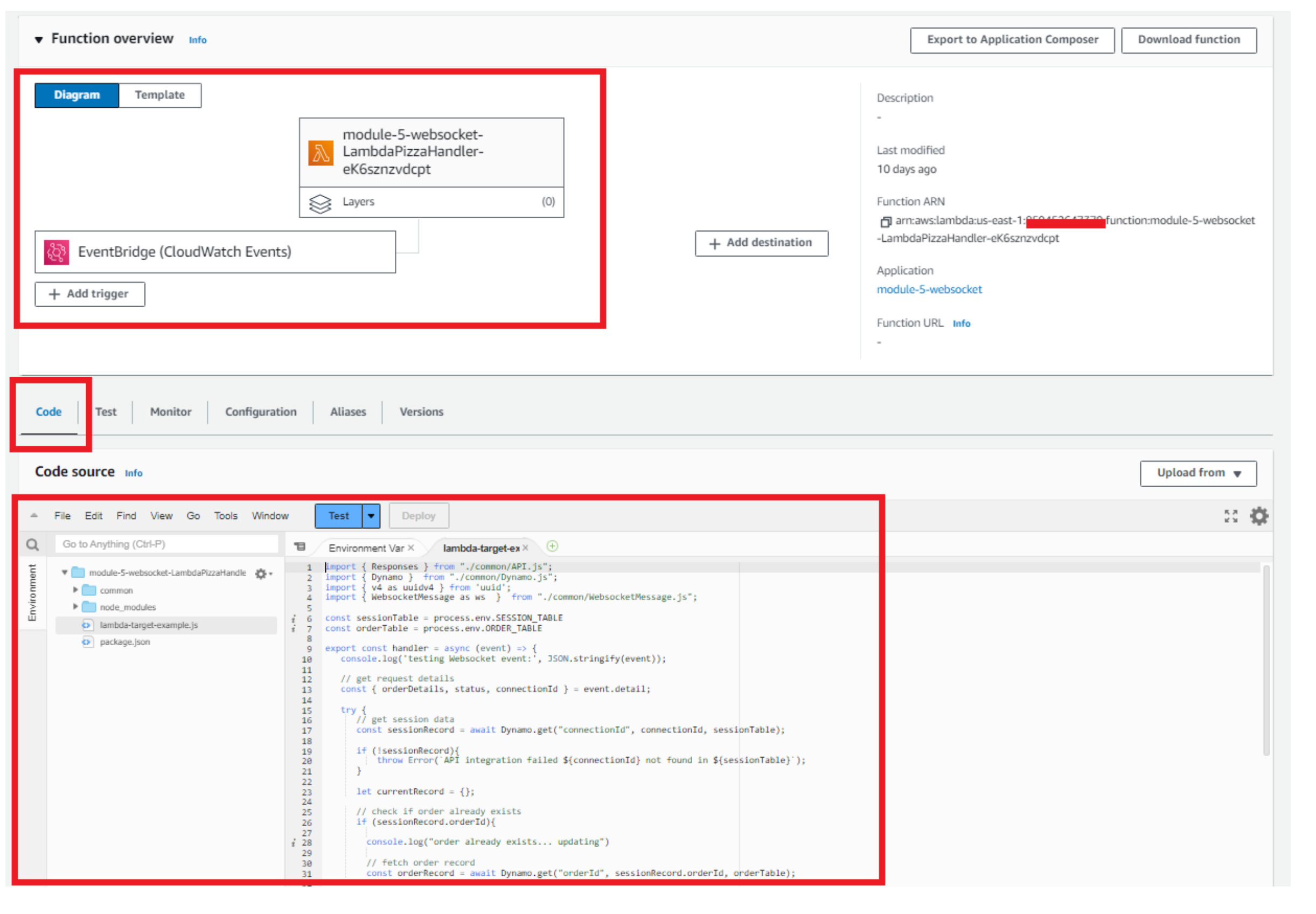
16. On the **Event pattern** tab, you can see that the event matches the **"detail-type"** and **"source"** properties, which are provided in the JSON input by the Step Functions (see step 13, Events).



17. Choose the **Targets** tab, the target for this rule is our Lambda function, that processes all of the events.



18. Click on the Lambda target link, a new window with the Lambda console page will be opened. In the function overview diagram, you can see that the lambda is triggered by **EventBridge (CloudWatch Events)**. You can also review its code, and learn how it processes the different events coming from Step Functions via the event bus. The Lambda is using the DynamoDB for connection tracking and for updating the order status. It also notifies the user via the WebSocket connection for event changes.



19. Go back to the cloud9 terminal, and disconnect the WebSocket connection by issuing:

```
CTRL-c
```

You should be able to see the terminal prompt again.

When a connection is terminated, the API Gateway $disconnect route is invoked. We have configured the integration request to delete the connection entry in the DynamoDB Sessions table.