# Amazon Websocket Configuration and implementation

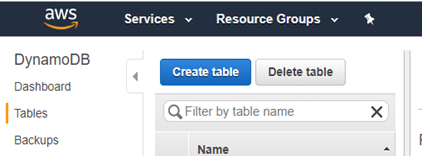
Few months back Amazon brought into websocket to the APIGateway which is useful for real time communication with multiple systems. In this article, we are going to configure the websocket on AWS. Client can be any application supporting websocket protocol. We will use [wscat](https://github.com/websockets/wscat) and Outsystems REST API as clients.

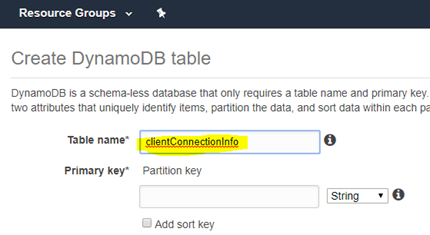
In this article, we are going to create [websocket](https://en.wikipedia.org/wiki/WebSocket) APIGateway to create a full duplex channel communication between 2 clients. To start with we require an AWS console account with permission to create roles (using IAM), policies, lambda functions, API (in APIGateway) and a table (in DynamoDB).

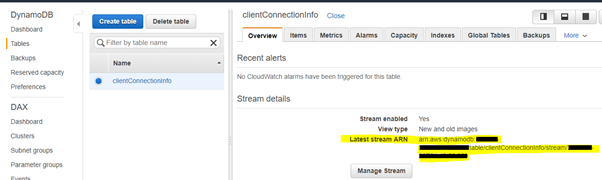
We require a role which has enough permission to preserve the client’s connection information in table, process the message and broadcast the message to the connected clients. We also require a developer account to create them. This developer account can be created by AWS admin. For POC/demo purpose we can create a free development account. So, let's start and login to the AWS console. <https://console.aws.amazon.com/>

## Table in DynamoDB

First, we need to create a table in Dynamo DB to preserve the connection info. This is required to broadcast the message to all connected clients. Name the table as “clientConnectionInfo”. Add the partition key as “connectionid” as string data type. Preserve the arn of the table, it will be required while giving access to policies.



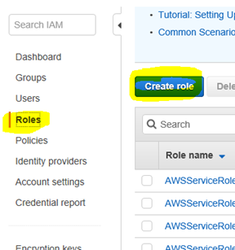




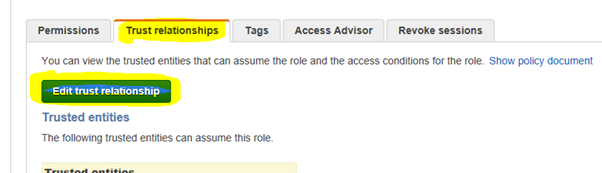
## Role – OutSystemsClientPortal

We require a role which will have policies i.e. permissions to process the clients request on AWS. for simplicity we will create a single role. Of-course multiple roles can be created for each purpose.

Create a role "OutSystemsClientPortal".



Edit the trusted relationship of the role and add Lambda and APIGateway services as trusted services. This is required for Lambda functions and API Gateway to allow the trust the role.



{

  "Version": "2012-10-17",

  "Statement": [

    {

      "Sid": "",

      "Effect": "Allow",

      "Principal": {

        "Service": [

          "[lambda.amazonaws.com](http://lambda.amazonaws.com)",

          "[apigateway.amazonaws.com](http://apigateway.amazonaws.com)"

        ]

      },

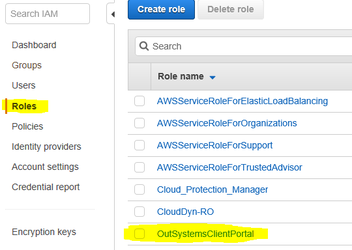
      "Action": "sts:AssumeRole"

    }

  ]

}

Once the role is created it will be displayed in the list. We have not yet defined the policies to the role. Each policy requires access to the yet to be created resources. Hence, we will revisit it again.



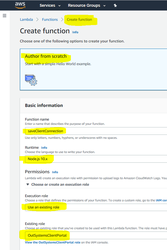
## Lambda functions

Now we are set to define our Lambda functions. We will require few lambda functions for this application. These functions handles the mechanism to communicate over the channel with clients. We are using Node.js while other languages like .Net, Java, Pythin, etc, are also supported. Anyway to make a basic websocket communication, nodejs sample should suffix.

### Save Client’s Connection

When a client sends a connect request to the websocket server, the lambda function saveClientConnection will preserve the connection info. This is required to broadcast the message. So, let’s go and define the function.

Traverse to the Lambda > Create function. Specify the name as “saveClientConnection”. Select the existing role defined earlier. Click Create function.



### Add the code

Remove the default code and add the below code

const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient();

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

    const myconnectionId = event.requestContext.connectionId;

    addConnectionId(myconnectionId).then(() => {

    callback(null, {

        "statusCode": 200

        })

    });

}

function addConnectionId(myconnectionId) {

    return ddb.put({

        TableName: 'clientConnectionInfo',

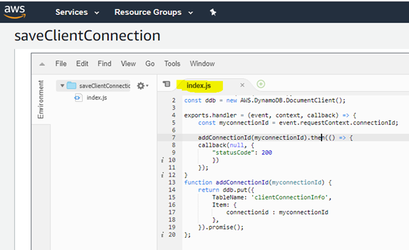
        Item: {

            connectionid : myconnectionId

        },

    }).promise();

};



### Delete Client’s connection

Create another lambda function “deleteClientConnection” to delete the client’s connection from the table on disconnection.

Use the existing role option and define the code as below.

const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient();

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

    const myconnectionId = event.requestContext.connectionId;

    deleteConnectionId(myconnectionId).then(() => {

    callback(null, {

        statusCode: 200,

        })

    });

}

function deleteConnectionId(myconnectionId) {

    return ddb.delete({

        TableName: 'clientConnectionInfo',

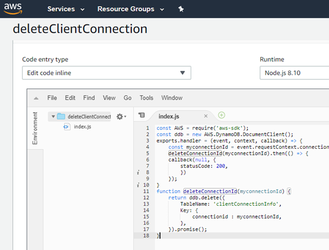
        Key: {

            connectionid : myconnectionId,

        },

    }).promise();

}



### Broadcast Message from client

This function broadcasts the message received from one of the connected clients to all other connected clients. Below implementation just passes on the message to other clients and do not send it back to itself. This function will be invoked by clients connected using wss:// protocol. You may like to update the code to broadcast the message back to the sender as well or something else based on the business requirement. In the below implementation, we are checking the secret key 1234 within the body of the message.

Create the function “broadcastMessage”. Use the existing role option and define the code as below.

const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient();

require('./patch.js');

let send = undefined;

function init(event) {

  const apigwManagementApi = new AWS.ApiGatewayManagementApi({

    apiVersion: '2018-11-29',

    endpoint: event.requestContext.domainName + '/' + event.requestContext.stage

  });

  send = async (connectionId, data) => {

  await apigwManagementApi.postToConnection({ ConnectionId: connectionId, Data: `${data}` }).promise();

  }

}

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

                const myconnectionId = event.requestContext.connectionId;

                var hasValidKey = false;

                if (event.body) {

                                let body = JSON.parse(event.body)

                                if (body.key) {

                                                if (body.key == '1234') {

                                                                hasValidKey = true;

                                                }

                                }

                }

                if (! hasValidKey) {

                                callback(null, {

                                                "statusCode": 403

                                                });

                                return {}

                }

                init(event);

                let message = JSON.parse(event.body).message

                getConnections().then((data) => {

                                data.Items.forEach(function(connection) {

                                  if (myconnectionId != connection.connectionid) {

                                                send(connection.connectionid, message);

                                  }

                                });

                });

                callback(null, {

                                "statusCode": 200

                                });

                return {}

};

function getConnections(){

    return ddb.scan({

        TableName: 'clientConnectionInfo',

    }).promise();

}

We also require another patch file to load the apigatewaymanagementapi service. So, add a new file “patch.js” to the same folder. Use below code and save the file in same folder.

require('aws-sdk/lib/node\_loader');

var AWS = require('aws-sdk/lib/core');

var Service = AWS.Service;

var apiLoader = AWS.apiLoader;

apiLoader.services['apigatewaymanagementapi'] = {};

AWS.ApiGatewayManagementApi = Service.defineService('apigatewaymanagementapi', ['2018-11-29']);

Object.defineProperty(apiLoader.services['apigatewaymanagementapi'], '2018-11-29', {

  get: function get() {

    var model = {

      "metadata": {

        "apiVersion": "2018-11-29",

        "endpointPrefix": "execute-api",

        "signingName": "execute-api",

        "serviceFullName": "AmazonApiGatewayManagementApi",

        "serviceId": "ApiGatewayManagementApi",

        "protocol": "rest-json",

        "jsonVersion": "1.1",

        "uid": "apigatewaymanagementapi-2018-11-29",

        "signatureVersion": "v4"

      },

      "operations": {

        "PostToConnection": {

          "http": {

            "requestUri": "/@connections/{connectionId}",

            "responseCode": 200

          },

          "input": {

            "type": "structure",

            "members": {

              "Data": {

                "type": "blob"

              },

              "ConnectionId": {

                "location": "uri",

                "locationName": "connectionId"

              }

            },

            "required": [

              "ConnectionId",

              "Data"

            ],

            "payload": "Data"

          }

        }

      },

      "shapes": {}

    }

    model.paginators = {

      "pagination": {}

    }

    return model;

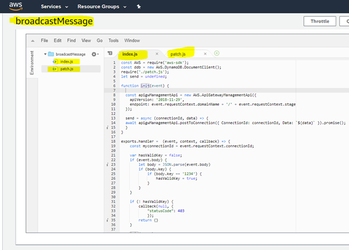
  },

  enumerable: true,

  configurable: true

});

module.exports = AWS.ApiGatewayManagementApi;



### Broadcast API Message

Previous lambda function broadcastMessage clients can send the message through webscoket connection to other connected clients. This new function (broadcastAPIMessage) is invoked by an exposed REST API to broadcast the message to the connected clients of websocket. This API can be invoked from POSTMAN or any other service and pass the message over HTTP GET.

Unlike webscoket communication of full message, let's do slight modification in the expected message through REST API. We allow only a number(i.e. ProgramId) as message from the clients sending to other connected clients using REST API. Code is self explanatory to validate the data type of the program id.

So, create a lambda function “broadcastAPIMessage”. Use the existing role option and define the code as below.

const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient();

require('./awsPatch.js');

let send = undefined;

function init(event) {

  const domain = event.domain;

  const stage = event.stage;

  const apigwManagementApi = new AWS.ApiGatewayManagementApi({

    apiVersion: '2018-11-29',

    endpoint: domain + '/' + stage

  });

  send = async (connectionId, data) => {

    await apigwManagementApi.postToConnection({ ConnectionId: connectionId, Data: `${data}` }).promise();

  }

}

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

    const programId = Number(event.programId);

    if (programId > 0) {

        init(event);

        getConnections().then((data) => {

            data.Items.forEach(function(connection) {

              send(connection.connectionid, programId);

            });

        });

    }

    var response = {

        "statusCode": 200

    };

    callback(null, response);

};

function getConnections(){

    return ddb.scan({

        TableName: 'clientConnectionInfo',

    }).promise();

}

“broadcastAPIMessage” function receives the program id in the event from REST API end point. It verifies if data type of the program id is number and broadcast it to connected clients through websocket. Of-course we can pass the full text message to the connected clients. If so, we don’t require to check the content of the message.

This function gets the list of connections from clientConnectionInfo table and passes to all connected clients by executing postToConnection. It requires domain and stage name of the hosted websocket end point. These values can be passed in query string or in the body of the request. We will pass the parameters when we define the REST API end point.

As of now, the function sends the success response as 200. You may return forbidden response as 403 if the data type is mismatched or any other status code as per the business requirement.

We also require another patch file to load the apigatewaymanagementapi service. So, add a new file “awsPatch.js” to the same folder. Use below code and save the file in same folder.

require('aws-sdk/lib/node\_loader');

var AWS = require('aws-sdk/lib/core');

var Service = AWS.Service;

var apiLoader = AWS.apiLoader;

apiLoader.services['apigatewaymanagementapi'] = {};

AWS.ApiGatewayManagementApi = Service.defineService('apigatewaymanagementapi', ['2018-11-29']);

Object.defineProperty(apiLoader.services['apigatewaymanagementapi'], '2018-11-29', {

  get: function get() {

    var model = {

      "metadata": {

        "apiVersion": "2018-11-29",

        "endpointPrefix": "execute-api",

        "signingName": "execute-api",

        "serviceFullName": "AmazonApiGatewayManagementApi",

        "serviceId": "ApiGatewayManagementApi",

        "protocol": "rest-json",

        "jsonVersion": "1.1",

        "uid": "apigatewaymanagementapi-2018-11-29",

        "signatureVersion": "v4"

      },

      "operations": {

        "PostToConnection": {

          "http": {

            "requestUri": "/@connections/{connectionId}",

            "responseCode": 200

          },

          "input": {

            "type": "structure",

            "members": {

              "Data": {

                "type": "blob"

              },

              "ConnectionId": {

                "location": "uri",

                "locationName": "connectionId"

              }

            },

            "required": [

              "ConnectionId",

              "Data"

            ],

            "payload": "Data"

          }

        }

      },

      "shapes": {}

    }

    model.paginators = {

      "pagination": {}

    }

    return model;

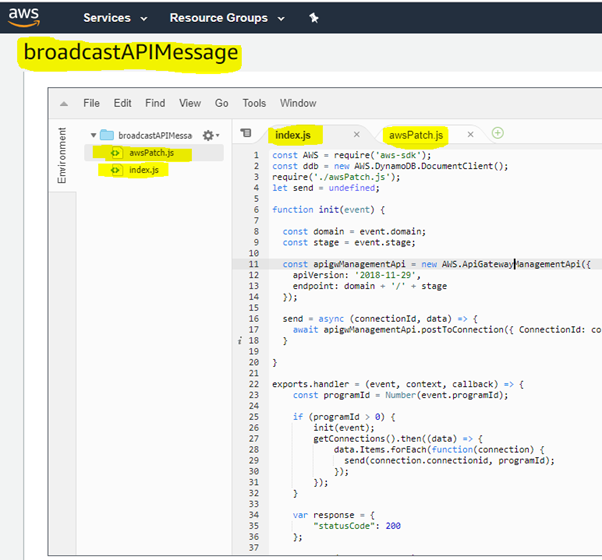
  },

  enumerable: true,

  configurable: true

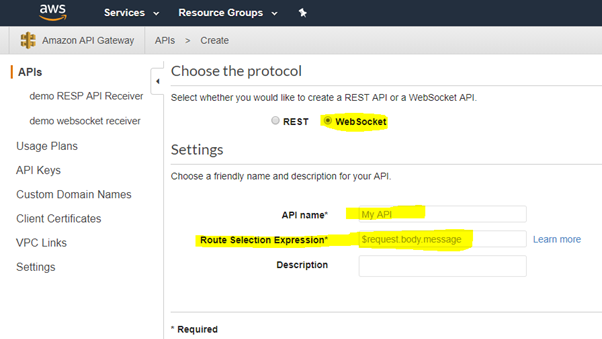
});

module.exports = AWS.ApiGatewayManagementApi;

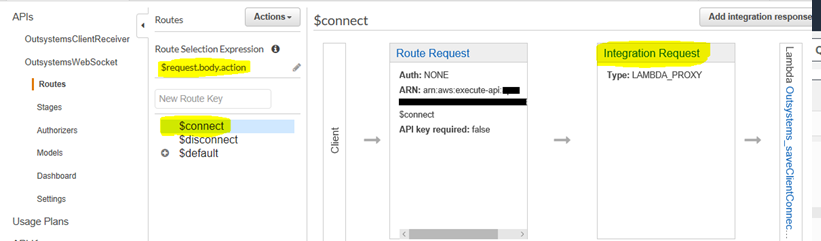


## Websocket API

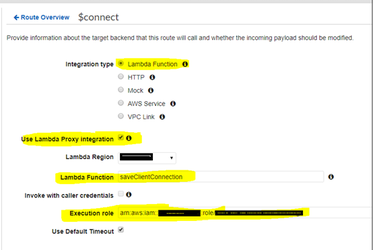
Let's create our websocket endpoint. Traverse to the Amazon API Gateway and click Create API to define a websocket API. Give a name “OutsystemsWebSocket”. Provide "$request.body.action" in the Route Selection Expression. AWS will parse the message and look for action tag in the json to redirect to appropriate API action. So as per the tags inside the message body, we can define multiple routes and process them differently.



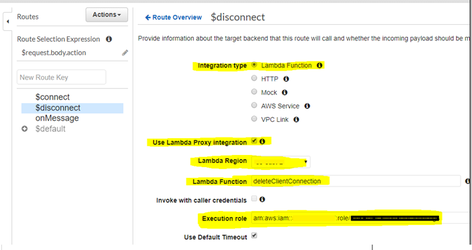
We will define 3 routes for our implementation. Click the plus icon in front of Connect route.



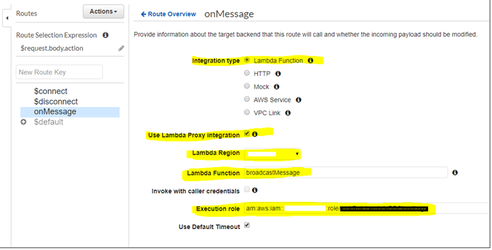
It's time to integrate websocket API call with Lambda function. Click the Integration Request to configure it. Select Integration type as Lambda Function. Specify respective region and type in the Lambda function name. When clients connect, we would like to save the connection info and hence API should invoke the saveClientConnection function. We must provide the role with appropriate policies (i.e. permission). So, copy paste the ARN of the role.



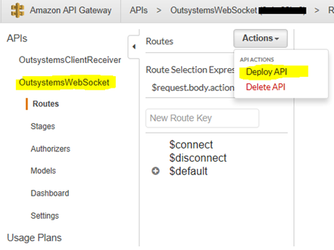
Similarly define the route disconnect.



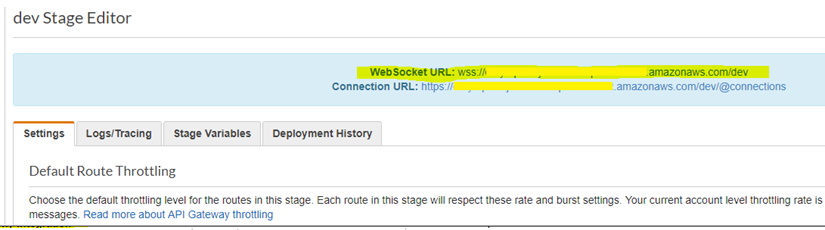
If we want the websocket clients to pass the message AS IS to the connected clients implement the route onMessage. To define this type onMessage in the New Route Key. The lambda function required here is broadcastMessage. Don’t forget to specify the role ARN.



We are ready to deploy. So, go ahead and select deploy option.



Once deployed AWS will provide with the endpoints to connect with the websocket api. Note down these, client will require this information to connect.

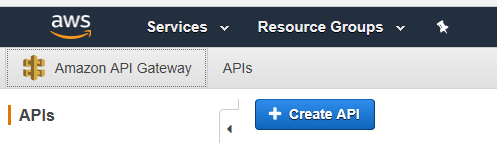


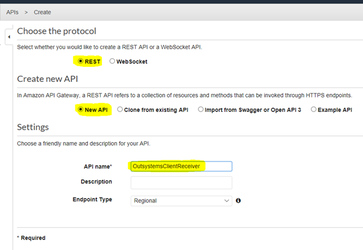
## REST API

Shall we expose a REST API to broadcast message to the connected client? If yes, let's create a REST API. Let's add some security to restrict clients to connect and pass the message. It’s optional for our implementation.

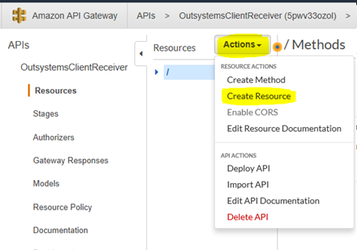
Traverse to the Amazon API Gateway and click Create API to define a REST API.

Give the name “OutsystemsClientReceiver”.

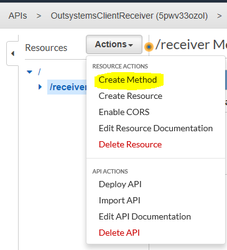


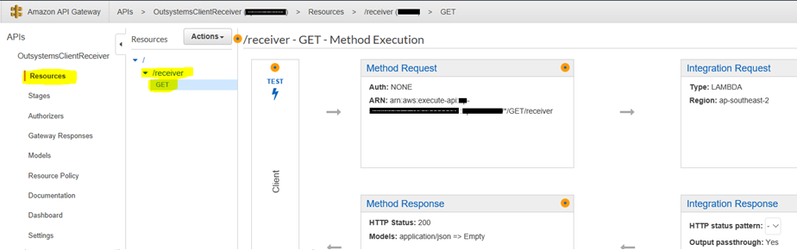


We can create multiple sub paths for the REST API. For our implementation we will use only one. Select the action Create Resource and give the name “receiver”.

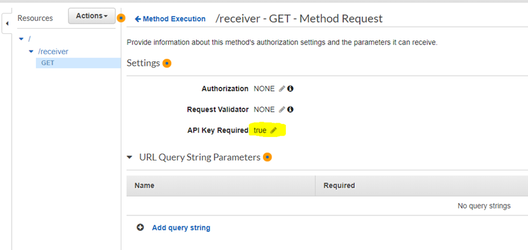


Now, select the resource “receiver” and define a REST method by selecting “Create Method”. Select the “Get” method from the dropdown and click the check mark to confirm.



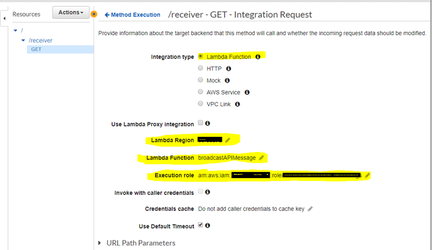


If you want to make the REST API secured, then update the “Method Request”. Here, we will use the API Key in header for authorization. As of now just select the "true" option for API Key required. We will define the API key later in this article.



REST API will invoke the lambda function to broadcast the message (as an example, we are passing Program Id as actual message) to all the connected clients. Clients are expected to send program id (i.e. message), websocket domain and stage name. Hence, we need to update the Integration Request of the Get method to invoke the lambda function and map the corresponding message (i.e. Program Id), web socket domain and stage name in the HTTP request. The API Gateway needs to map these parameters to lambda function.

Click Integration request, to configure it. Select Lamdda function option. Select the respective region and type in the lambda function. Copy the ARN of the defined role.



We need to define the mapping so expand the Mapping templates and click application/json.

Copy the below json

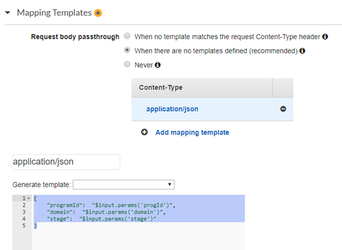
{

    "programId":  "$input.params('progId')",

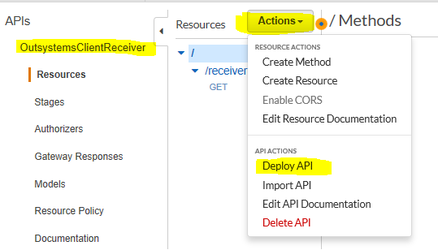
    "domain":  "$input.params('domain')",

    "stage":  "$input.params('stage')"

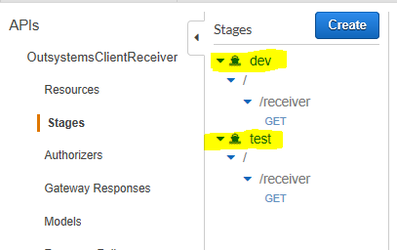
}



API is now ready for getting hosted. Select Deploy API option to do that.

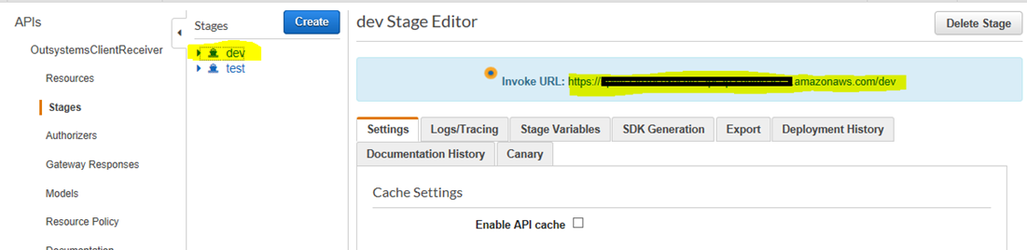


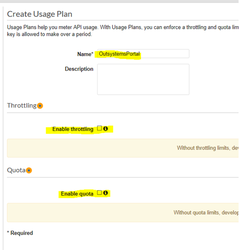
Give the stage name as dev. You may define more than one stages as per the business requirement. In the below screen shot, we have 2 stages: dev and test.



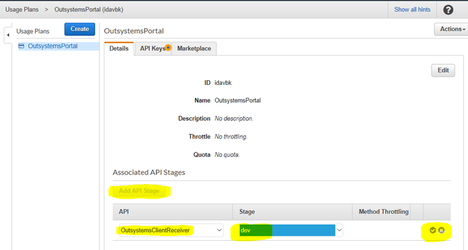
Once deployed AWS will provide with the endpoints to connect with the REST API. Note down these, client will require this information to connect.

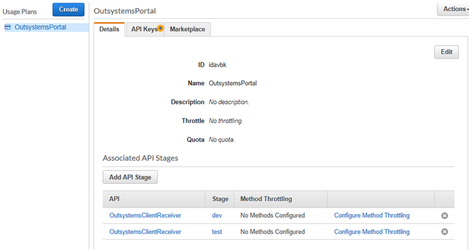
It’s time to get the secret API Key. Before we create the key, we need to create a usage plan. Select Usage Plans from left panel and click Create. Give a name and specify the parameter values as per the requirement.



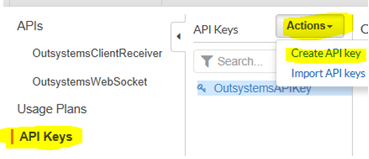


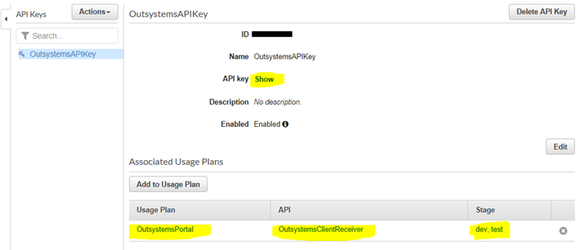
Now, click Add API Stage. Select the API from the drop down and respective Stage. Click tick to confirm. You may assign multiple API and their stages to the same usage plan.





Once the usage plan is created, its time to create a key. Click Create API Key and give a name. Add the usage plan to the Key.



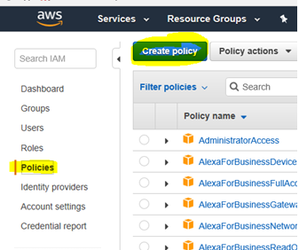


Copy the Key and preserve it. It should be provided in clients request to the REST API.

## Policies

After creating the Lambda functions, API’s and table it’s time to define the appropriate restrictions/permissions to the role. This can be achieved by assigning appropriate policies to the role.

Traverse to IAM -> Policies -> Create policy



We need to create 3 policies. Follow the steps. We can define the policies using the editor or json.

### OutsystemsDBAccessToTable

Below is the json of the policy.

{

    "Version": "2012-10-17",

    "Statement": [

        {

            "Sid": "VisualEditor0",

            "Effect": "Allow",

            "Action": [

                "dynamodb:PutItem",

                "dynamodb:DeleteItem",

                "dynamodb:GetItem",

                "dynamodb:Scan",

                "dynamodb:Query",

                "dynamodb:UpdateItem",

                "dynamodb:GetRecords"

            ],

            "Resource": "arn:aws:dynamodb: <aws region>:<account id>:table/clientConnectionInfo"

        },

        {

            "Sid": "VisualEditor1",

            "Effect": "Allow",

            "Action": "dynamodb:ListStreams",

            "Resource": "arn:aws:dynamodb: <aws region>:<account id>:table/clientConnectionInfo"

        }

    ]

}

### OutsystemsLambdaExceutePolicy

We require APIGateway ARN in this policy. As of now define as below json, we will come back after API is created and assign the ARN details of it.

{

    "Version": "2012-10-17",

    "Statement": [

        {

            "Sid": "VisualEditor0",

            "Effect": "Allow",

            "Action": "lambda:InvokeFunction",

            "Resource": "\*"

        },

        {

            "Sid": "VisualEditor1",

            "Effect": "Allow",

            "Action": "execute-api:\*",

            "Resource": "arn:aws:execute-api: <aws region>:<account id>:<API Gateway Id>/\*/Post/\*"

        }

    ]

}

### OutsystemsManageAPI

We require APIGateway ARN in this policy. As of now define as below json, we will come back after API is created and assign the ARN details of it.

{

    "Version": "2012-10-17",

    "Statement": [

        {

            "Sid": "VisualEditor0",

            "Effect": "Allow",

            "Action": [

                "execute-api:Invoke",

                "execute-api:InvalidateCache"

            ],

            "Resource": "arn:aws:execute-api: <aws region>:<account id>:<API Gateway Id>/\*/\*/\*"

        },

        {

            "Sid": "VisualEditor1",

            "Effect": "Allow",

            "Action": "execute-api:ManageConnections",

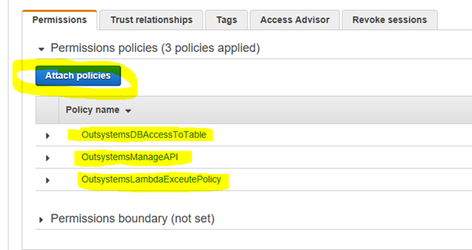
            "Resource": "\*"

        }

    ]

}

Attach the 3 policies to the role. Remove other attached policies, if any.

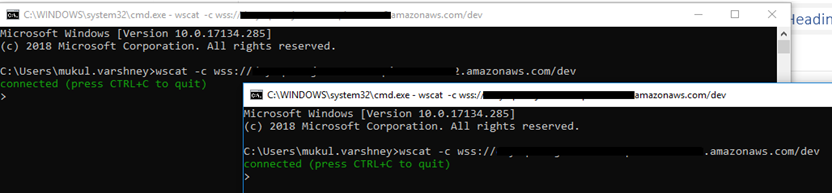


## Development testing

We are ready to test now. For our testing we will use wscat. You can download it from <https://github.com/websockets/wscat>.

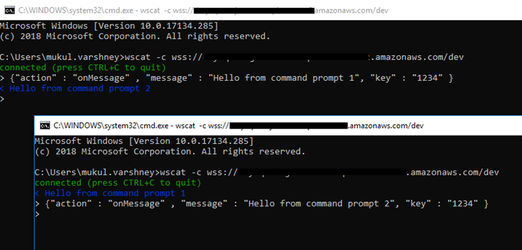
Open 2 command line instances and enter the command

wscat -c wss://xxxxxxx.zzzzzzzz.yyyyyyy.amazonaws.com/dev

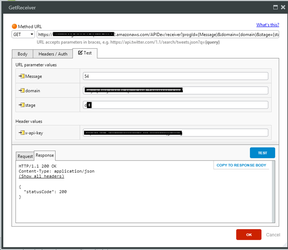


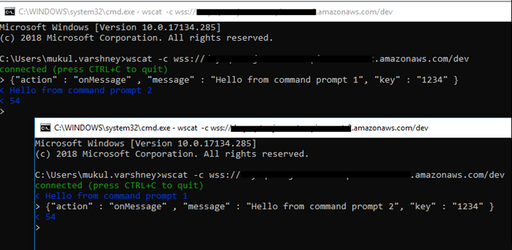
Type the message

{"action" : "onMessage" , "message" : "Hello from command prompt 2", "key" : "1234" }



To test the REST API,





A demo/sample app in Outsystems is published at forge.

<https://www.outsystems.com/forge/component-overview/5813/simple-websocket-connector-sample>

Web socket connector plugin

<https://www.outsystems.com/forge/component-overview/5812/websocketconnector>

References:

<https://www.freecodecamp.org/news/real-time-applications-using-websockets-with-aws-api-gateway-and-lambda-a5bb493e9452/>

<https://hackernoon.com/websockets-api-gateway-9d4aca493d39>