# **API GATEWAY WEB SOCKET API**

Hello! folks its awesome to have you here working towards understanding how Web Socket APIs are created using API gateway. Before we get to know the How let’s know the What and Why of Web Socket API.

# **Section 0: Pre-requisites**

## **Web Socket:**

WebSocket is a protocol that provides low-latency data exchange between client and server. The WebSocket protocol maintains persistent two-way communication between parties. This means that a client can receive and send data to the server at any time during the session and vice-versa. In contrast to the HTTP protocol, no extra requests are required.

### **Use Cases:**

You can use API Gateway WebSocket APIs to build secure, real-time communication applications without having to provision or manage any servers to manage connections or large-scale data exchanges. Targeted use cases include real-time applications such as the following:

* Chat applications
* Real-time dashboards such as stock tickers
* Real-time alerts and notifications

As you can see in the below the protocol is quite like a ping pong game between a client and a server once they have done a handshake (maintaining Connection). To close the connection either the client or server can close the connection.

Diagram

Description automatically generated

As we now are clear about the why, where and what of Web Socket API lets get to the interesting part how to implement one of these using AWS API Gateway. To do lets have a step by step tutorial.

# **Section 1: Tutorial -> Building a serverless chat app with a WebSocket API, Lambda and DynamoDB**

## **Diagram Description automatically generated**

### **Prior** **Requirements:**

1. **AWS Account**
2. **AWS IAM user** with console access, more info click [ [here](https://docs.aws.amazon.com/apigateway/latest/developerguide/setting-up.html) ]
3. **Wscat** to connect to our API, more info click [ [here](https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-how-to-call-websocket-api-wscat.html) ]
4. **AWS CloudFormation** template

**Step 1: Creating Lambda functions and DynamoDB Table.**

Use the **starter.yaml** file which is a template to create a [DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html) table to store the app’s client IDs. Each connected client has a unique ID which we will use as the table's partition key. This template also creates [Lambda](https://docs.aws.amazon.com/lambda/latest/dg/welcome.html) functions that update your client connections in DynamoDB and handle sending messages to connected clients.

To create AWS CloudFormation Stack, click [ [here](https://console.aws.amazon.com/cloudformation) ]. Follow it up by choosing the standard choice, then in the **SPECIFY TEMPLATE** option, choose **Upload a template file** for that upload the Starter.Yaml File which you downloaded.

You’d have a similar screen

Graphical user interface, text, application, email

Description automatically generated

After selecting **Next,** Enter the Stack name -> **chat-bot-tutorial**

Graphical user interface, application

Description automatically generated

Choose **Next,** this step involves configuring stack options in this you don’t have to provide any information, choose **Next,** Now we’ll acknowledge that AWS CloudFormation might create IAM resource under the Capabilities section and **create stack.**

Graphical user interface, application

Description automatically generated

The reason we have created **AWS CloudFormation** is that it gives developers and businesses an easy way to create a collection of related AWS and third-party resources, and provision and manage them in an orderly and predictable fashion. In our case AWS CloudFormation provisions the resources specified in the template. It can take a few minutes to finish provisioning your resources. When the status of your AWS CloudFormation stack is **CREATE\_COMPLETE**, you're ready to move on to the next step.

You’d have a similar screen

Graphical user interface, text, application, email

Description automatically generated

**Step 2: Involves Creating a WebSocket API**

To do so click [ [here](https://console.aws.amazon.com/apigateway) ] and choose WebSocket API, then give the API name -> **chat-bot-api** . Then follow it up by giving the Route Selection Expression -> **request.body.action,** This determines the route that API gateway invokes when a client sends a message.

You’d have a similar screen

Graphical user interface, text, application, email

Description automatically generated

Choose **Next** and then For **Predefined Routes** add all the three **$connect**, **$disconnect, $default.**

**Graphical user interface, application

Description automatically generated**

For **Custom Routes** add a route and give the value of the Route Key.  This custom route handles messages that are sent to connected clients.

You’d have a similar screen

Graphical user interface, text, application

Description automatically generated

Choose **Next,** now under **Attach Integrations** add the type as Lambda for all routes. Choose the Lambda function that you created with AWS CloudFormation.

You’d have a similar screen, do the same for all the routes.

Graphical user interface, application

Description automatically generated

Choose **Next,** by default API gateway creates a stage name **Production** and automatically deploys the API to the stage, Choose **Next.**

Choose **Create and Deploy**.

Graphical user interface, application

Description automatically generated

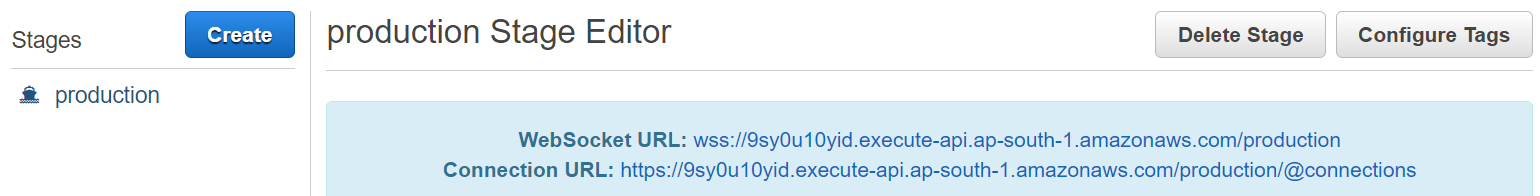
Now All that’s left is to Test your API, Kudos for coming this far!! So, Let’s get started.

**Step 3: Testing the API**

We need to use the wscat command to connect to the API, click [ [here](https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-how-to-call-websocket-api-wscat.html) ]

We also need to know the invoke URL, so click [[here](https://console.aws.amazon.com/apigateway)] and choose your API and then choose **Stages** follow it up by choosing **Production.**

Take a note of your **WebSocket URL**, you’d have a similar screen

****