

Rapid Prototyping with Serverless and Cloud

Transformative Technology for the Financial Industry

PREPARED BY:
Niveetha Nair
WEBSITE:
www.stacs.io
CONTACT:
info@stacs.io

Contents

Rapid Prototyping with Serverless and Cloud Technology	3
Preface	3
1. Introduction	3
2. High Level Architecture Design	4
2.1 Frontend connectivity to the backend	5
DynamoDB	5
Lambda	6
API Gateway	7
2.2 Periodic Data Refresh	8
EventBridge	8
3. Lambda and DynamoDB Setup	10
4. Conclusion	11

Rapid Prototyping with Serverless and Cloud Technology

Preface

Rapid prototyping is an essential skill for a team running on Agile and Scrum methodologies. Focusing on the deliverables, utilising available resources, and managing substantial and quick modifications requires some finesse — this is what STACS Solutions Engineering team aims to achieve.

We have built and launched large enterprise applications running on top of our STACS Blockchain platform. To be more agile, we have started researching rapid prototyping to collect feedback and increase our feature release and bug fix Turn Around Time (TAT). In the project we are about to present in this article, the focus is set on user interaction and experience. We have found that serverless technology like AWS Lambda is well suited to handle the backend processing which enables the team to focus on the User Experience (UX) to provide a better quality of life for users. This article will share our experience on rapid prototyping where we utilise the AWS cloud services to iterate our build quickly which has been very cost-effective.

1. Introduction

Serverless technology such as AWS Lambda is very useful for basic computational processes and being event-driven allows us to save cost on running full-blown servers for applications that do not have heavy workloads.

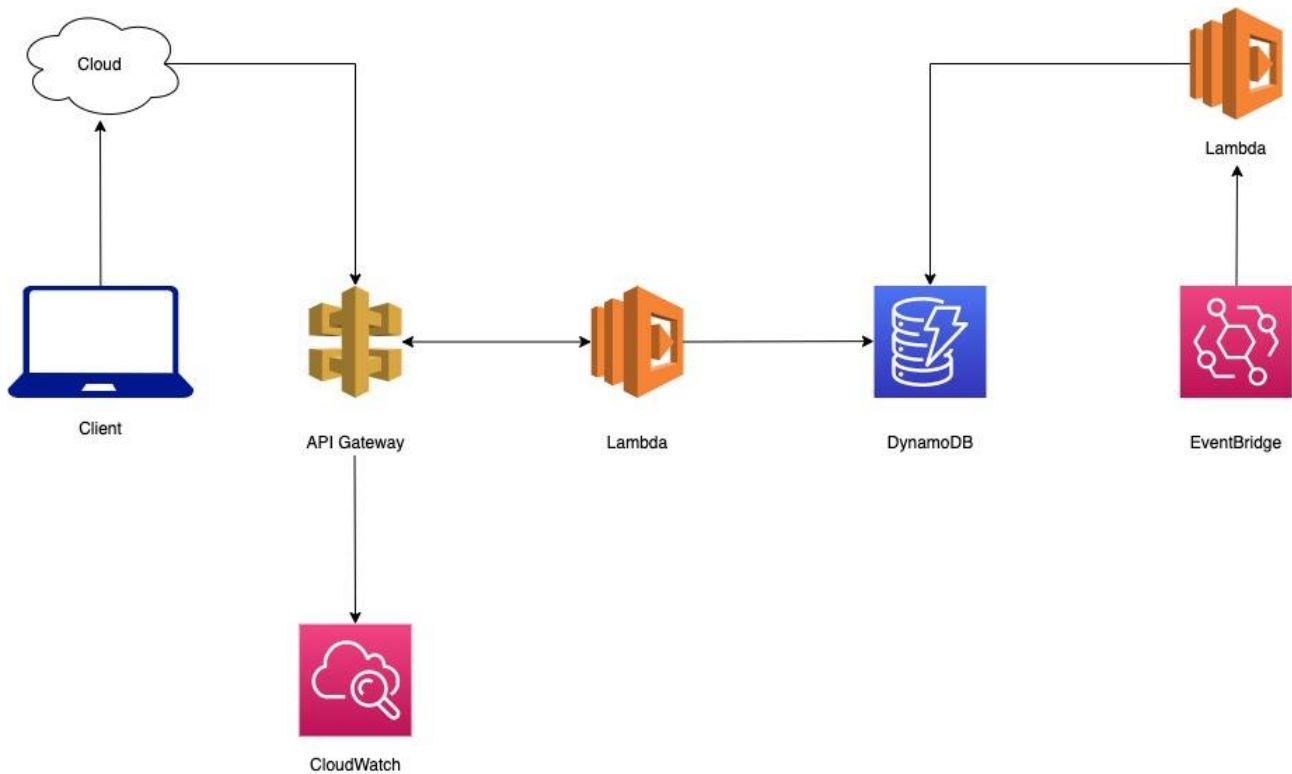
This prototype application uses 4 AWS services in its backend architecture -

1. DynamoDB for unstructured data storage
2. Lambda for event-driven basic computation
3. API Gateway to manage API messages
4. EventBridge

We will look at the high-level architecture design, its use cases, why some of these services were chosen and how they were implemented.

2. High Level Architecture Design

As the focus is on the serverless backend implementation of the project, let us take a deeper look at the AWS architecture implemented to handle functions and data.



The architecture can be split into 2 sections:

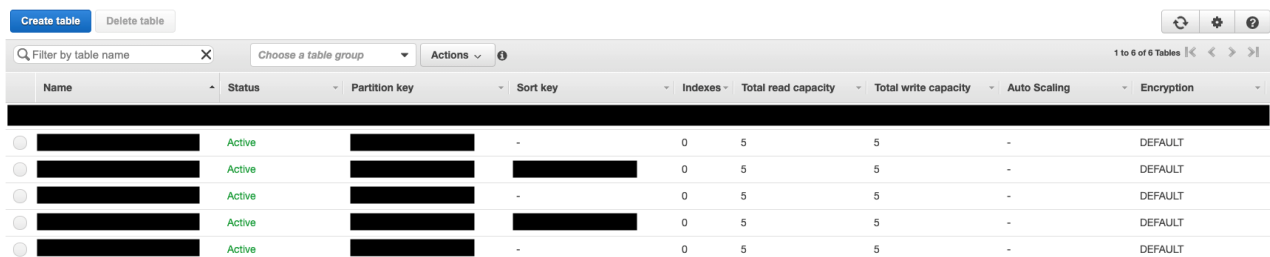
- Event-driven calls from the frontend (static webpage in S3) to DynamoDB for data retrieval or processing
- Periodic refresh of data in DynamoDB from the blockchain using EventBridge and Lambda

2.1 Frontend connectivity to the backend

The first use case of the application is to read data from the database and perform simple computation on data being sent from the user interface and save it persistently. The frontend will send REST HTTP API requests to API Gateway, which will then trigger a Lambda Function call. This function will either read values from DynamoDB to be sent back to the UI or save and update values in the database. These 3 services combined handles user interaction and user data storage of the application.

DynamoDB

Tables in DynamoDB were created to handle the data storage for the application and its ability to handle unstructured data provides us with a flexible way to store incoming data feeds.



The screenshot shows the AWS DynamoDB console interface. At the top, there are buttons for 'Create table' and 'Delete table'. Below these is a search bar labeled 'Filter by table name' and a dropdown for 'Choose a table group'. To the right of the search bar is an 'Actions' dropdown and a status indicator '1 to 6 of 6 Tables'. The main part of the console is a table with the following columns: Name, Status, Partition key, Sort key, Indexes, Total read capacity, Total write capacity, Auto Scaling, and Encryption. There are five rows of data, each representing a table. The first row is selected, indicated by a radio button in the 'Name' column. The 'Status' column for all tables shows 'Active'. The 'Partition key' and 'Sort key' columns show redacted information. The 'Indexes' column shows '0' for all tables. The 'Total read capacity' and 'Total write capacity' columns show '5' for all tables. The 'Auto Scaling' column shows '-' for all tables. The 'Encryption' column shows 'DEFAULT' for all tables.

Name	Status	Partition key	Sort key	Indexes	Total read capacity	Total write capacity	Auto Scaling	Encryption
[Redacted]	Active	[Redacted]	-	0	5	5	-	DEFAULT
[Redacted]	Active	[Redacted]	[Redacted]	0	5	5	-	DEFAULT
[Redacted]	Active	[Redacted]	-	0	5	5	-	DEFAULT
[Redacted]	Active	[Redacted]	[Redacted]	0	5	5	-	DEFAULT
[Redacted]	Active	[Redacted]	-	0	5	5	-	DEFAULT

CRUD data operations were performed on these tables through Lambda.

Lambda

Functions used by the application either read values from database or perform simple computation before storing the values into the database.

A simple read function on Lambda would look like this:



```
1 const AWS = require('aws-sdk');
2 const ddb = new AWS.DynamoDB.DocumentClient({region: [REDACTED]});
3
4 exports.handler = async (event, context, callback) => {
5   await getTransaction().then(data => {
6     data.Items.forEach(item => {
7       console.log(item);
8     });
9     callback(null, {
10      statusCode: 200,
11      body: data.Items,
12      headers: {
13        'Access-Control-Allow-Origin': '*',
14      }
15    });
16  }).catch(err => {
17    console.log(err);
18  });
19 };
20
21
22 const getTransaction = () => {
23   const params = {
24     TableName: [REDACTED]
25   };
26   return ddb.scan(params).promise();
27 };
```

As DynamoDB was used as the data storage provider for this application, interaction between Lambda and the data layer was easily configured using the *AWS.DynamoDB.DocumentClient* import. The document client import simplifies basic operations on DynamoDB. The Lambda function uses the AWS SDK for JavaScript to query and scan tables using these methods of the DynamoDB Document Client class - GET, PUT, UPDATE, QUERY and DELETE.

To allow Lambda to perform these actions on DynamoDB, its execution role IAM policy had to be updated.

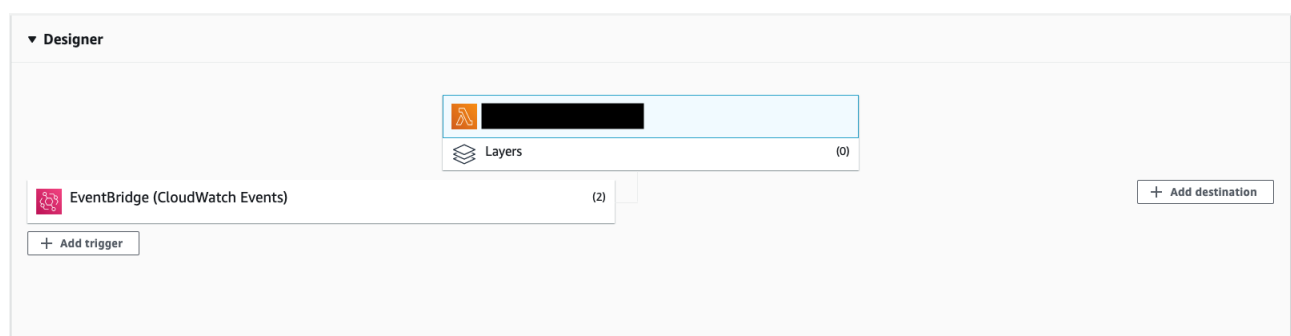
API Gateway

API Gateway was used to manage the endpoints called by the application. In the Integration Request section of the Method Execution in API Gateway, Lambda Function integration type was used. The associated Lambda region and function were also selected in the Integration Request.

The screenshot shows the 'Method Execution' page for a GET method in the Integration Request section. The 'Integration type' is set to 'Lambda Function'. Below this, there are options for 'HTTP', 'Mock', 'AWS Service', and 'VPC Link', all of which are unselected. The 'Use Lambda Proxy integration' checkbox is also unselected. The 'Lambda Region' and 'Lambda Function' fields are both set to '[REDACTED]'. The 'Execution role' field is empty. The 'Invoke with caller credentials' checkbox is unselected. The 'Credentials cache' is set to 'Do not add caller credentials to cache key'. The 'Use Default Timeout' checkbox is checked.

All method call endpoints used by the application were added to API Gateway. Each endpoint used the same Integration Request, only varying the Lambda Function invoked.

Functions invoked by the application through API Gateway were computed using Lambda. As API Gateway was the function trigger, the Lambda design is as such,



Lambda functions invoked by API Gateway would have to update their policy to allow for this function call to pass.

2.2 Periodic Data Refresh

Our application requires streaming of data constantly, which brings us to the second use case of our backend design. To input data into the database, a Lambda function was created to send API requests to the blockchain and write values to the database. As the function had to be triggered continuously, an EventBridge was set up to handle this.

EventBridge

Before the EventBridge was set up, the Lambda function had to be created first. Once we have that, we can proceed with creating a new Rule.

Name and description

Name

Maximum of 64 characters consisting of lower/upper case letters, -, ., ~, _.

Description - *optional*

Define pattern

Build or customize an Event Pattern or set a Schedule to invoke Targets.

☐ Event pattern [Info](#)
Build a pattern to match events

☒ Schedule [Info](#)
Invoke your targets on a schedule

☒ Fixed rate every

1

Minutes

☐ Cron expression

CRON expression have six required fields, which are separated by white space. [Learn more about CRON expression.](#) [🔗](#) Enter CRON expression below to see the next 10 trigger date(s).


0/5 * * * ? *

[▶ Sample event\(s\)](#)

Select event bus

Select an event bus for this rule.

- ☐ AWS default event bus
- ☐ Custom or partner event bus

 Custom or partner event bus is not supported when Schedule is selected.

☒ Enable the rule on the selected event bus

Select targets

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule)

Target

Remove

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule)

Lambda function ▼

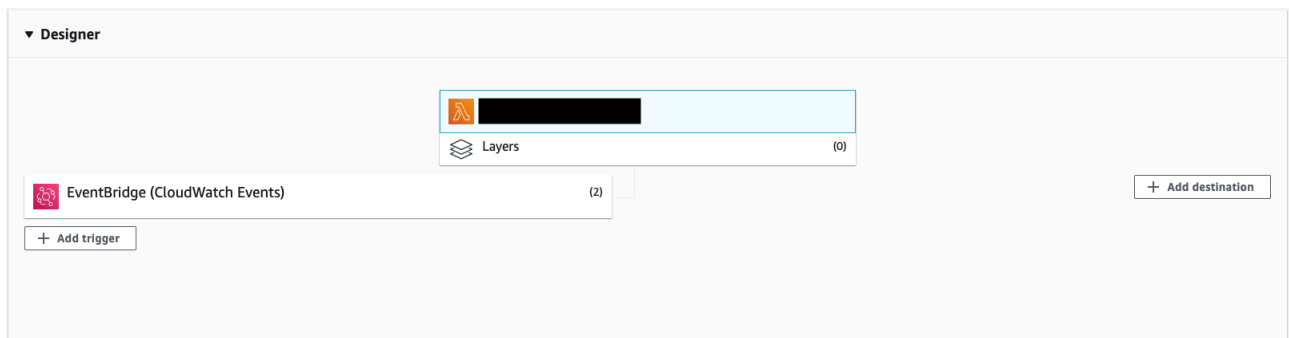
Function

▼

- ▶ Configure version/alias
- ▶ Configure input
- ▶ Retry policy and dead-letter queue

Add target

The smallest unit of time in an EventBridge Rule is 1 minute. This means the function is triggered by EventBridge every minute. Once the rule has been created, it is now set as the trigger for the Lambda function.



3. Lambda and DynamoDB Setup

AWS Lambda is a serverless compute service that lets us run code without provisioning or managing servers. Lambda functions can be written in multiple languages (as the UI was written in JavaScript, we stuck with JavaScript when coding the function). As cost is essential to hosting a web server, using Lambda is favourable due to its pay-as-you-go setup without the need to pre-provision infrastructure.

To create a Lambda function is simple:

In the previous section, we mentioned that Lambda would need explicit access to DynamoDB and be allowed to be accessed by API Gateway — these policies should be attached to the function.

Lambda also provides testing of its functions:

A test JSON object can be passed into the Lambda function. Once the test cases have been configured, they can be used to ensure the function is working as it should (the above example is the test for a GET function, hence an empty JSON object is passed into the function).

4. Conclusion

With this set up in place, it is now quick and simple to add new API endpoints and functions to the application. API Gateway also provides additional security to our APIs with authentication and authorisation capabilities, allowing us to offload security and access control to AWS API Gateway.

Using an event-driven serverless backend removes the need to host a dedicated backend for the basic computation which brings down hosting costs and manpower costs for supporting the services, allowing us to focus on development work. Additionally, it improves our application uptime with minimal manpower needs since the service is managed entirely by AWS.

Moving forward, we will be exploring the use of AWS Lambda Applications instead of Functions to see what additional benefits serverless technologies can bring to rapid prototyping and to explore an asynchronous update of blockchain data to the frontend with a message queue rather than rely on a periodic polling approach using EventBridge rather than rely on a periodic polling approach using EventBridge.



Transformative Technology for the Financial Industry

STACS
Designed for Finance

Contact

STACS

www.stacs.io

info@stacs.io

Jin Ser

Solutions Architect Director

jin.ser@stacs.io