Alan Duffin

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT
FINAL SUBMISSION

TABLE OF CONTENTS

CERTIFICATE CLOUD SOLUTIONS FINAL SUBMISSION - INTRODUCTION

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

The use case selected for my project is Loan origination. This is the process whereby a potential borrower applies for a new loan, and a lender processes that application. Loan Origination includes all the steps from taking a loan application up to drawdown of funds or refusal of the application. The loan origination process has many long running steps, including human decision points.

The central objective of this project is to build a solution for long – running mortgage processing that requires human interaction / approval. The specific problem domain is the mortgage application / approval process. The implementation platform chosen after the Literature Review and Technical Evaluation stages was the AWS Serverless Application Model using AWS Step Functions and AWS Lambda.

The breadth of AWS Services that I required to implement this project was very wide. The full list of AWS Services used are:

- CloudFormation
- Serverless Application Model templates
- IAM
- S3
- DynamoDB
- Lambda
- SNS
- API Gateway Rest API
- API Gateway Websocket API
- Step Functions

In addition to these AWS services, it was necessary to learn a number of other open source frameworks that would allow me to create a useable development / testing environment. This are listed below:

- Git
- Javascript / NodeJS / NPM
- Angular / Typescript
- Docker
- Websockets NPM
- Visual Studio Code IDE

Some simple project metrics from my project are:

Metric	#
Lambda Functions	31
Lines of Code	6160
S3 Buckets	2
SNS Topics	2
DynamoDB Tables	6
API Gateway HTTP Endpoints	14
Step Function States	29

CERT.CLOUD.SOLUTIONS PROJECT PROPOSAL

STUDENT

Student Name: Alan Duffin

Student ID: X00159409

Module: Certificate in Cloud Solutions Architecture (Project)

OVERVIEW

This document is an initial proposal for the final project module of the Certificate in Solutions Architecture at TUD (Tallaght Campus).

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

These systems have a common problem set that must be solved. The workflow pattern must address requirements such as state transition, error handling, component / service integration and human interaction/decision making.

Many vendors have designed solution frameworks around the concept of the Finite State Machine such as BPEL (Business Process Execution Language). The growth of SaaS and Paas platforms has allowed users to take advantage of fully managed, scalable services for many problem domains including workflow.

Amazon Web Services provides access to two managed services for workflow use cases. These are:

1 AWS Step Functions

2 AWS Simple Workflow Service

GOALS

The goal of the proposed project is to compare and contrast AWS Step Functions and AWS Simple Workflow for a problem domain such as Loan Origination under a number of categories including:

- Features
- Ease of use
- Latency
- Cost
- Maintainability / Trobleshooting
- State Maintenance / Error Handling

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Initial Research and Project Plan

Student Name: Alan Duffin

Student ID: X00159409

CONTENTS

OVERVIEW

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

These systems have a common problem set that must be solved. The workflow pattern must address requirements such as state transition, error handling, component / service integration and human interaction/decision making.

Many vendors have designed solution frameworks around the concept of the Finite State Machine such as BPEL (Business Process Execution Language). The growth of SaaS and Paas platforms has allowed users to take advantage of fully managed, scalable services for many problem domains including workflow.

My project proposal is to research the available Workflow/Servlerless services that are available from the three largest SaaS Cloud providers: Google Cloud, Amazon Web Services, Microsoft Azure. I will subsequently create a comparison matrix for these SaaS platforms based on a set of required criteria for the Workflow use case that is to be implemented. This matrix will be used to select a Cloud vendor platform upon which to implement the use case. Some of these criteria/features are:

- Features
- Ease of use
- Latency
- Cost
- Maintainability / Trobleshooting
- State Maintenance / Error Handling.

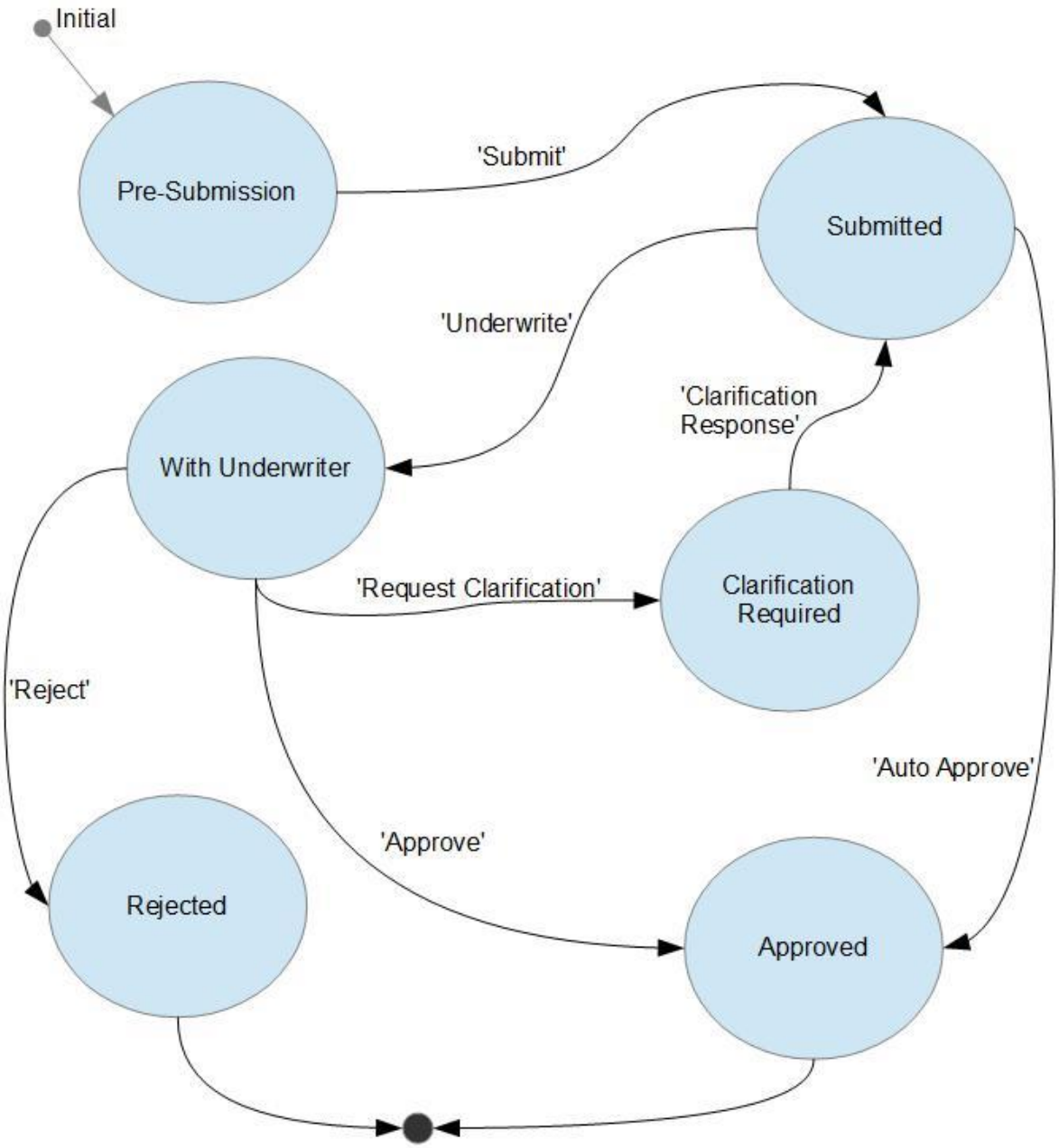
USE CASE OUTLINE

Loan origination is the process by which a borrower applies for a new loan, and a lender processes that application. Origination includes all the steps from taking a loan application up to drawdown of funds or refusal of the application.

The loan origination process has many steps, including:

- Loan application — The homebuyer fills out a loan application form.
- Documents — The buyer submits documentation for income, employment and financial status.
- Underwriting — The lender reviews the loan application, verifies the buyer's credit and determines if the buyer's income and financial status is such that the buyer qualifies for the loan.
- Finalise loan application — Once the terms are agreed on, the loan application is processed by the lender, going through several departments, such as underwriting and documentation processing.
- Loan approval — After final processing, the lender decides to approve or reject the loan application.

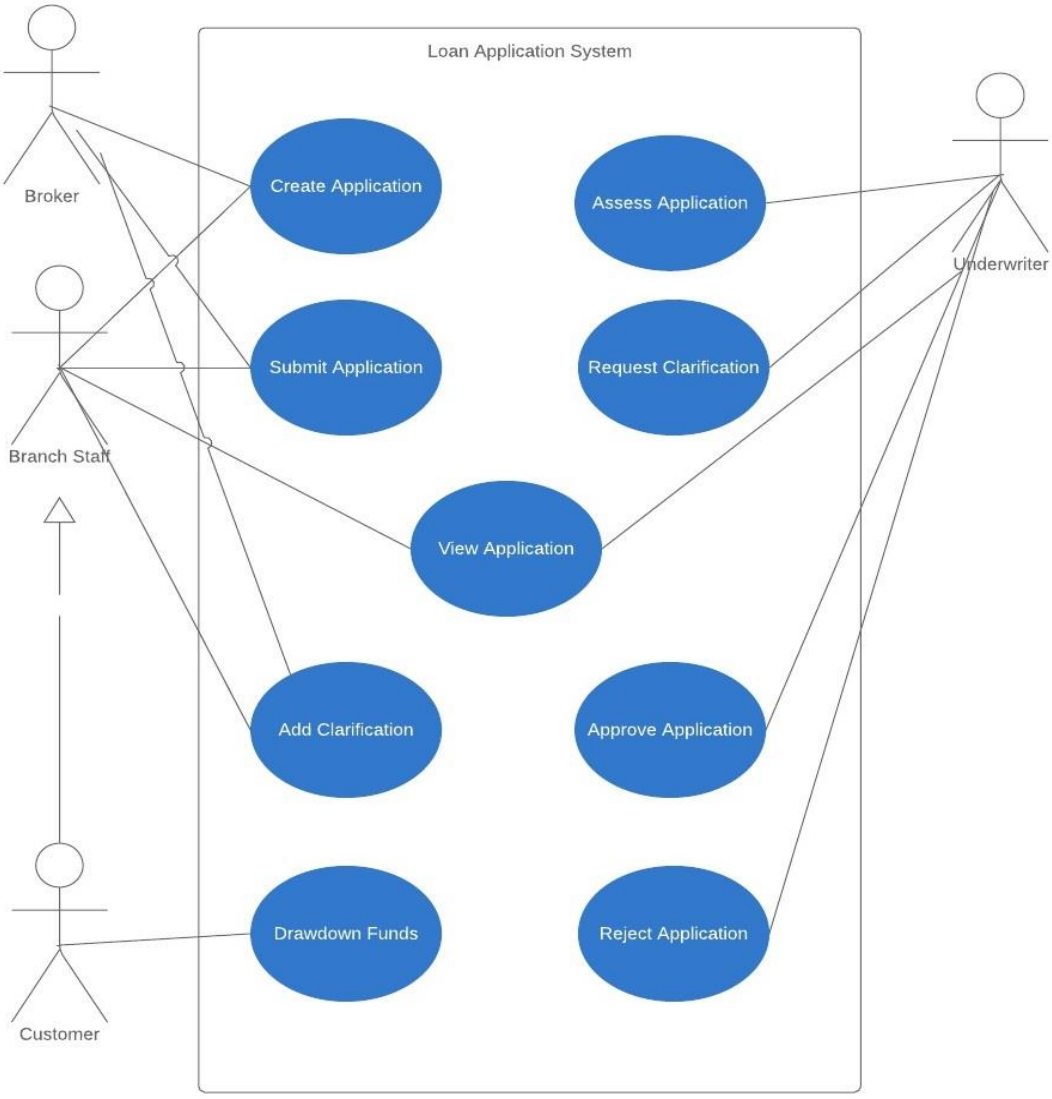
The Lifecycle of a Homeloan Application



Human Actors	
Branch Staff	Actor 1
Broker	Actor 2
Customer	Actor 3
Underwriter	Actor 4

Loan Origination Use Case Diagram

Alan Duffin | October 3, 2019



The Bank Branch Staff Member or Broker (Actor 1) via the user interface will view the list of active Loan Applications.

- Actor 1 can choose to view the details of any individual Loan Application.
- Actor 1 can create a new Loan Application on behalf of a customer.
- Actor 1 can input all required customer and loan details.
- After reviewing all information Actor 1 can submit the Loan application into the Approvals workflow. This should initiate a workflow execution.
- Possibly notifications should be sent to all human actors of loan application state changes. This could be via email, or a front end notification.
- The workflow execution should consist of the steps shown in the table below (not exhaustive).

Step Number	Workflow Component
1	Validate that all mandatory data has been input for the Approval process
2	Run the affordability calculator and validate that the case does not fall outside loan rules e.g. age of applicants, loan term, unsuitable employment, not in current employment for long enough etc.
3	Decision rule to either auto approve or route the application to an Underwriter (Actor 2) for human review.
4	The Auto Approval service will mark the application as approved and notify Actor 1. (Notification method unknown at the moment)
5	The Underwriters have discretion to approve the loan application based on extenuating circumstances such as Junior doctor with immediate prospects of salary increases and stable job prospects.
6	If manual Approval is required the workflow will notify Actor 4 that the Loan Application is awaiting review and pause execution.
7	Actor 4 must open the Loan Application and review details.
8	Actor 4 can then choose to Approve or Decline the application. This is done via the front end and calls a service that restarts the workflow execution.
9	The workflow will then notify Actor 1 and the Customer (Actor 3) using some mechanism.
10	The workflow will terminate.

INITIAL RESEARCH

My initial reseach of the Google, Microsoft and AWS PaaS services, show that the services listed below can be considered as candidate solution platforms for the workflow use-case of this project.

Each of the PaaS cloud vendors provide services that will be required for this use case. These provide solutions for the following problem domains:

1 Workflow Orchestration

2 Servlerless hosting of business logic.

3 Publish/Subcribe for messaging/notification.

4 Queuing for asynchronous task processing.

SaaS Provider	Workflow related Services
Google Cloud	Cloud Composer. https://cloud.google.com/composer/ Apache Airflow. https://airflow.apache.org/ Cloud Functions. https://cloud.google.com/functions/ Cloud Pub/Sub. https://cloud.google.com/pubsub/docs/
Microsoft Azure	Logic Apps. https://azure.microsoft.com/en-us/services/logic-apps/ Microsoft Flow. https://flow.microsoft.com/en-us/ Azure Functions. https://azure.microsoft.com/en-us/services/functions/ Event Grid. https://azure.microsoft.com/en-us/services/event-grid/ Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/
Amazon Web Services	Step Funtions. https://aws.amazon.com/step-functions/ Lambda. https://aws.amazon.com/lambda/ SNS. https://aws.amazon.com/sns/ SQS. https://aws.amazon.com/sqs/

INITIAL PROJECT PLAN

	Tasks	Dates
Literature Review	Amazon Web Services	Sun 13th - 20th Oct
	Microsoft Azure	
	Google Cloud Platform	
Technology Assessment	Amazon Web Services	Sun 20th - 27th Oct
	Microsoft Azure	
	Google Cloud Platform	
Implementation Iteration 1	Create Database Schema. Implement Database CRUD code. Implement / Test Individual Services using Serverless functions and REST API.	Sun 27th Oct - 10th Nov
Implementation Iteration 2	Create initial Workflow orchestration and tasks. Integration testing with services from Iteration 1.	Sun 10th - 24th Nov
Implementation Iteration 3	Create / Test User Interface. Create / Test Nofication mechanisms for Human actors. Final integration testing	Sun 24th Nov - 8th Dec
Prepare Final Report		Sun 8thth - 15th Dec

RISKS

- The Moodle Project schedule lists 1 week for Literature Review and 1 week for R & D of all the required services across the 3 cloud vendors. I do not believe this to be achievable.
- To build a fully functional project implementation will require a combination of many services. For example, if implemented on AWS these would include SNS, SQS, Lambda, S3, API Gateway. A browser-based frontend will be required to allow user interaction. This would require significant software development using HTML, javascript, etc perhaps using AWS frameworks such as ASW Amplify, Cognito, IAM. There is a risk that the software development effort may not be achievable in the allotted timeframe.
- Design/Architecture lacks flexibility, is not fit for purpose or is infeasible.
- Estimates are inaccurate perhaps due to scope creep or required activities are missing from scope.
- Learning curves lead to delays because developer is inexperienced.
- Requirements are incomplete or fail to align with available systems/services.User interface doesn't allow users to complete tasks or is low quality.

VERSION 1.0

12/10/2019

ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

FEASIBILITY AND REFINED RESEARCH PLAN

TABLE OF CONTENTS

PROJECT BACKGROUND AND DESCRIPTION

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

These systems have a common problem set that must be solved. The workflow pattern must address requirements such as state transition, error handling, component / service integration and human interaction/decision making.

Many vendors have designed solution frameworks around the concept of the Finite State Machine such as BPEL (Business Process Execution Language). The growth of SaaS and Paas platforms has allowed users to take advantage of fully managed, scalable services for many problem domains including workflow.

My project proposal is to research the available Workflow/Servlerless services that are available from the three largest SaaS Cloud providers: Google Cloud, Amazon Web Services, Microsoft Azure. I will subsequently create a comparison matrix for these SaaS platforms based on a set of required criteria for the Workflow use case that is to be implemented. This matrix will be used to select a Cloud vendor platform upon which to implement the use case. Some of these criteria/features are:

- Features
- Ease of use
- Latency
- Cost
- Maintainability / Trobleshooting
- State Maintenance / Error Handling.

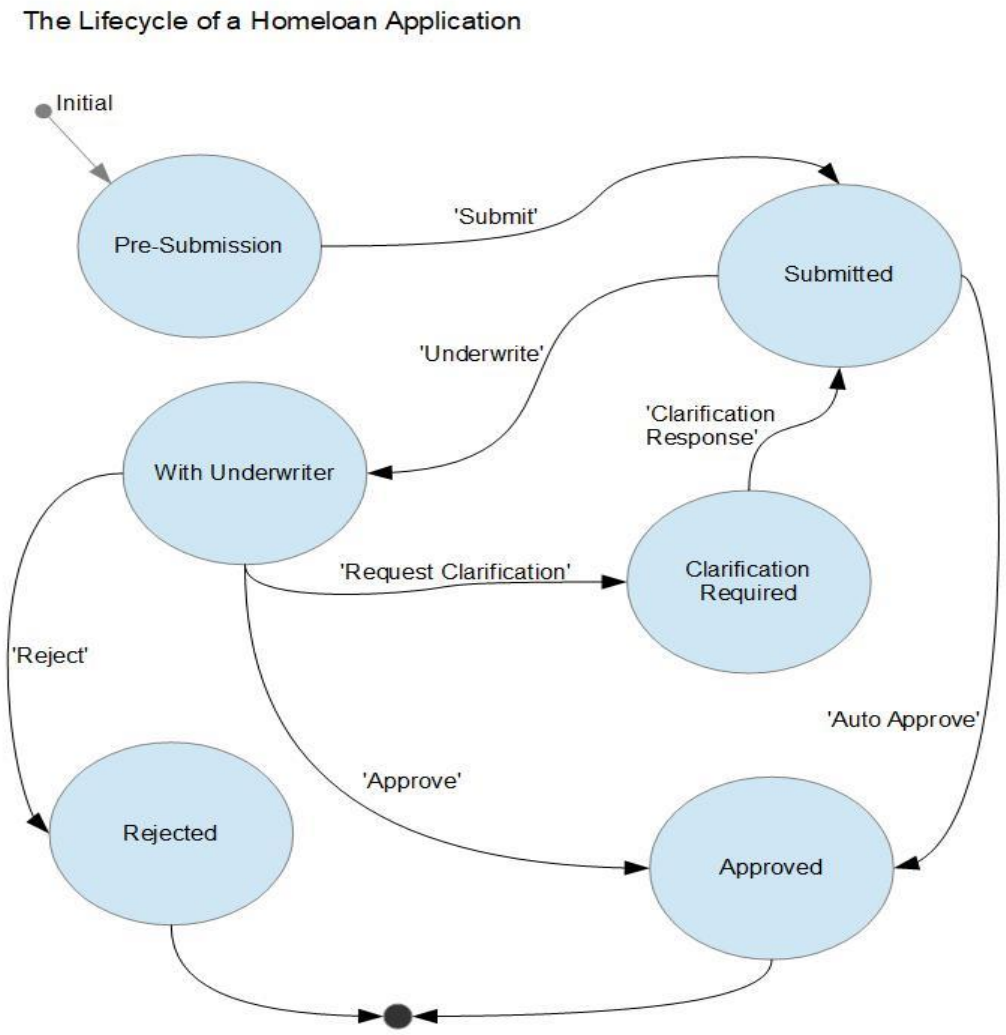
PROJECT DESCRIPTION

Loan origination is the process by which a borrower applies for a new loan, and a lender processes that application. Origination includes all the steps from taking a loan application up to drawdown of funds or refusal of the application.

The loan origination process has many steps, including:

- Loan application — The homebuyer fills out a loan application form.
- Documents — The buyer submits documentation for income, employment and financial status.
- Underwriting — The lender reviews the loan application, verifies the buyer's credit and determines if the buyer's income and financial status is such that the buyer qualifies for the loan.
- Finalise loan application — Once the terms are agreed on, the loan application is processed by the lender, going through several departments, such as underwriting and documentation processing.
- Loan approval — After final processing, the lender decides to approve or reject the loan application.

The Lifecycle of a Homeloan Application



PROJECT USECASE

Human Actors	
Branch Staff	Actor 1
Broker	Actor 2
Customer	Actor 3
Underwriter	Actor 4

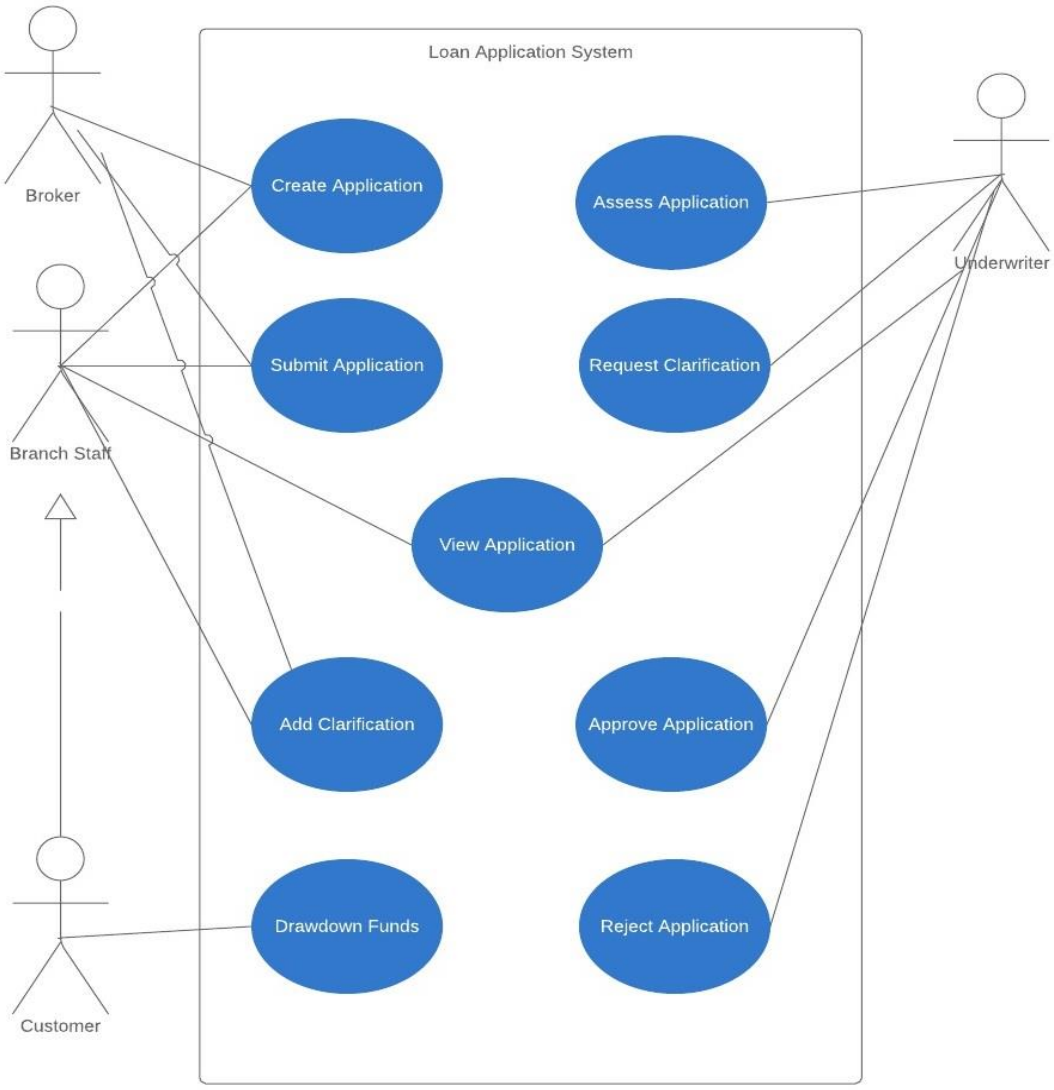
The Bank Branch Staff Member or Broker (Actor 1) via the user interface will view the list of active Loan Applications.

- Actor 1 can choose to view the details of any individual Loan Application.
- Actor 1 can create a new Loan Application on behalf of a customer.
- Actor 1 can input all required customer and loan details.
- After reviewing all information Actor 1 can submit the Loan application into the Approvals workflow. This should initiate a workflow execution.
- Possibly notifications should be sent to all human actors of loan application state changes. This could be via email, or a front end notification.
- The workflow execution should consist of the steps shown in the table below (not exhaustive).

Step Number	Workflow Component
1	Validate that all mandatory data has been input for the Approval process
2	Run the affordability calculator and validate that the case does not fall outside loan rules e.g. age of applicants, loan term, unsuitable employment, not in current employment for long enough etc.
3	Decision rule to either auto approve or route the application to an Underwriter (Actor 2) for human review.
4	The Auto Approval service will mark the application as approved and notify Actor 1. (Notification method unknown at the moment)
5	The Underwriters have discretion to approve the loan application based on extenuating circumstances such as Junior doctor with immediate prospects of salary increases and stable job prospects.
6	If manual Approval is required the workflow will notify Actor 4 that the Loan Application is awaiting review and pause execution.
7	Actor 4 must open the Loan Application and review details.
8	Actor 4 can then choose to Approve or Decline the application. This is done via the front end and calls a service that restarts the workflow execution.
9	The workflow will then notify Actor 1 and the Customer (Actor 3) using some mechanism.
10	The workflow will terminate.

Loan Origination Use Case Diagram

Alan Duffin | October 3, 2019



PROJECT DELIVERABLES

Key Deliverables	Interim Deliverables	Date Due
Initial Research Proposal		6th October
Feasibility and Research Project Plan		13th October
Literature Review/Research		20th October
Technology Research and Development		27th October
Project Implementation	Iteration 1	10th November
	Iteration 2	24th November
	Final Iteration	8th December
Final Report and Presentation		15th December

TECHNICAL AREAS OF INVESTIGATION

The following table lists the PaaS services that will be required to implement a solution for my project use case.

SaaS Provider	Services for Investigation
Google Cloud	Cloud Composer. Apache Airflow. Cloud Functions. Cloud Pub/Sub. Cloud Datastore.
Microsoft Azure	Logic Apps. Microsoft Flow. Azure Functions. Event Grid. Queue Storage. CosmosDB.
Amazon Web Services	Step Funtions. Lambda. SNS. SQS. DynamoDB.

The most important services for research are obviously the workflow/orchestration PaaS services that are provided by Google Cloud Platform, AWS and Azure. These services are Google Orchestrator, AWS Step Functions and Azure Logic Apps.

Cloud Composer is a hosted solution for the open source Apache Airflow platform. It is based on a Directed Acyclic Graph which is slightly different that a State Machine that does allow cycles. It supports a hybrid cloud or multi vendor solution. Only the Python development language is supported.

AWS Step Functions provides a serverless orchestration solution. Step Functions provides pre build steps that implement basic primatives to speed development and improve reliability and reuse. Step Functions also integrate out of the box with other AWS services. It can also support a hybrid cloud model.

The Azure Logic Apps service was designed specifically by Microsoft to allow non-developers to design and build Workflows in the cloud. Logic Apps benefits from a library of hundreds of pre-built Connectors to integrate with existing Azure Services. Logic Apps also support the hybrid cloud model by allowing the connection of distributed systems across on-premises and in-cloud environments.

My research into these services will focus on criteria that will allow speed to market, ease of use, cost, ease of maintainance and diagnostic tools. Ideally the service should have out of the box support for integration with the cloud vendor NoSQL database, Notification service, Servlerless components and Queue services. The initial research should estimate the usage statistics for the application in terms of throughput, storage, fixed costs. The cloud vendor Cost Calulators will then be used to estimate the cost of deployment and running the application for the remainder of the year. Should any of the Cloud vendors prove to be outliers in terms of costs that may prove to be a deciding factor in the ultimate selection decision.

TECHNICAL REQUIREMENTS AND COSTS

In order to build and test prototypes to investigate PaaS services associated with the project use case, I will require an account with each of the three Cloud vendors that are under investigation.

I will also require a local software development environment on my laptop. The three vendors do not all support a common set of programming languages for each of the services that I will be using. The only common language available seems to be Javascript. I will require a Javascript / NodeJS development environment.

Google Cloud Platform, AWS and Azure all have different pricing models even for equivalent services.

Google Cloud Platform and Azure both provide a credit voucher when customers initially subscribe. AWS provide a 12 month period where services are available for free but subject to usage limits.

Azure provide 170 euro free credit and Google Cloud Platform provide $300 credit upon subscription.

The pricing structure for the services that are under investigation for this project are as follows:

	AWS	Azure	Google
Workflow	4,000 free state transitions p/m. First year	€0.000022 per execution.	$280 p/m approx
Servlerless	1 M Requests free p/m	1 M Requests free p/m	2 M Requests free p/m
Queue Service	1 M Requests free p/m	100K operations free p/m	10GB free p/m
Database	25GB free p/m 200M requests free p/m	5-GB and 400 RUs provisioned throughput first year.	1 GB free p/d 50K Reads/20K writes p/d

PROJECT TIMELINES

	Tasks	Dates
Literature Review	Amazon Web Services	Sun 13th - 20th Oct
	Microsoft Azure	
	Google Cloud Platform	
Technology Assessment	Amazon Web Services	Sun 20th - 27th Oct
	Microsoft Azure	
	Google Cloud Platform	
Implementation Iteration 1	Create Database Schema. Implement Database CRUD code. Implement / Test Individual Services using Serverless functions and REST API.	Sun 27th Oct - 10th Nov
Implementation Iteration 2	Create initial Workflow orchestration and tasks. Integration testing with services from Iteration 1.	Sun 10th - 24th Nov
Implementation Iteration 3	Create / Test User Interface. Create / Test Nofication mechanisms for Human actors. Final integration testing	Sun 24th Nov - 8th Dec
Prepare Final Report		Sun 8th - 15th Dec

RISKS AND ISSUES MANAGEMENT

- The Moodle Project schedule prescribes one week for Literature Review and one week for R & D of all the required services across the 3 cloud vendors. I do not believe this to be achievable.
- To build a fully functional project implementation will require a combination of many services. For example, if implemented on AWS these would include SNS, SQS, Lambda, S3, API Gateway. A browser-based frontend will be required to allow user interaction. This would require significant software development using HTML, javascript, etc perhaps using AWS frameworks such as ASW Amplify, Cognito, IAM. There is a risk that the software development effort may not be achievable in the allotted timeframe.
- Design/Architecture lacks flexibility, is not fit for purpose or is infeasible.
- Estimates are inaccurate perhaps due to scope creep or required activities are missing from scope.
- Learning curves lead to delays because developer is inexperienced.
- Requirements are incomplete or fail to align with available systems/services.User interface doesn't allow users to complete tasks or is low quality.

RISK MITIGATION

The core functionality for this project is to implement a workflow orchestration for the mortgage origination use case. The individual services that need to be developed will be done on a Servlerless platform. The applicatoin data will be stored on a NoSQL managed cloud database. The services will be exposed via HTTP endpoints. Ideally there would also be a web based front end to allow a user friendly interface coupled with appropriate Authentication and Authorisation controls. However, due to project deadlines this front end which is not part of the core Workflow research may be ommitted.

REFERENCE LIST AND BIBLIOGRAPHY

Cloud Services FAQs

Google Cloud, Composer. https://cloud.google.com/composer/ (10 Oct, 2019)

Apache Airflow. https://airflow.apache.org/ (10 Oct, 2019)

Google Cloud, Functions. https://cloud.google.com/functions/ (10 Oct, 2019)

Google Cloud, Pub/Sub. https://cloud.google.com/pubsub/docs/ (10 Oct, 2019)

Google Cloud, Datastore. https://cloud.google.com/datastore/ (10 Oct, 2019)

Azure, Logic Apps. https://azure.microsoft.com/en-us/services/logic-apps/ (11 Oct, 2019)

Azure, Functions. https://azure.microsoft.com/en-us/services/functions/ (11 Oct, 2019)

Azure, Event Grid. https://azure.microsoft.com/en-us/services/event-grid/ (11 Oct, 2019)

Azure, Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/ (11 Oct, 2019)

Azure, ComosDB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction/ (11 Oct, 2019)

AWS, Step Functions. https://aws.amazon.com/step-functions/ (12 Oct, 2019)

AWS, Lambda. https://aws.amazon.com/lambda/ (12 Oct, 2019)

AWS, SNS. https://aws.amazon.com/sns/ (12 Oct, 2019)

AWS, SQS. https://aws.amazon.com/sqs/ (12 Oct, 2019)

AWS, Dynamo DB. https://aws.amazon.com/dynamodb/ (12 Oct, 2019)

Cloud Services Pricing

Google Cloud, Composer. https://cloud.google.com/composer/pricing/ (10 Oct, 2019)

Google Cloud, Functions. https://cloud.google.com/functions/pricing/ (10 Oct, 2019)

Google Cloud, Pub/Sub. https://cloud.google.com/pubsub/pricing/ (10 Oct, 2019)

Google Cloud, Datastore. https://cloud.google.com/datastore/pricing (10 Oct, 2019)

Google Cloud, Free Limits. https://cloud.google.com/free/ (10 Oct, 2019)

Azure, Logic Apps. https://azure.microsoft.com/en-us/pricing/details/logic-apps/ (11 Oct, 2019)

Azure, Functions. https://azure.microsoft.com/en-us/pricing/details/functions/ (11 Oct, 2019)

Azure, Event Grid. https://azure.microsoft.com/en-us/pricing/details/event-grid/ (11 Oct, 2019)

Azure, Queue Storage. https://azure.microsoft.com/en-in/pricing/details/storage/queues/ (11 Oct, 19)

Azure, ComosDB. https://azure.microsoft.com/en-us/pricing/details/cosmos-db/ (11 Oct, 2019)

Azure, Free Limits. https://azure.microsoft.com/en-us/free/#new-products

AWS, Step Functions. https://aws.amazon.com/step-functions/pricing/ (12 Oct, 2019)

AWS, Lambda. https://aws.amazon.com/lambda/pricing/ (12 Oct, 2019)

AWS, SNS. https://aws.amazon.com/sns/pricing/ (12 Oct, 2019)

AWS, SQS. https://aws.amazon.com/sqs/pricing/ (12 Oct, 2019)

AWS, Dynamo DB. https://aws.amazon.com/dynamodb/pricing/provisioned/ (12 Oct, 2019)

AWS, Free Limits. https://aws.amazon.com/free/

VERSION 1.0

19/10/2019

ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

LITERATURE REVIEW

TABLE OF CONTENTS

AREAS OF RESEARCH

The project that I have chosen to undertake will attempt to develop an implementation of a use case that relies on the following three concepts:

1 Serverless Computing

2 Workflow

3 Finite State Machines

In order to understand the theory behind each of these concepts and the practical benefits / disadvantages with regard to my specific use-case I will undertake to review a range of academic / peer reviewed literature on each subject.

WHAT IS A FINITE STATE MACHINE

A FSM is defined on Wikipedia as:

"A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. " [5]

"A state is a description of the status of a system that is waiting to execute a transition. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received. For example, when using an audio system to listen to the radio (the system is in the "radio" state), receiving a "next" stimulus results in moving to the next station. When the system is in the "CD" state, the "next" stimulus results in moving to the next track. Identical stimuli trigger different actions depending on the current state. [5]

In some finite-state machine representations, it is also possible to associate actions with a state:

1. *an entry action: performed when entering the state, and*
2. *an exit action: performed when exiting the state."* [5]

A Finite State Machine is a generalisation of a Directed Graph. Graphs of this type are used across the field of computer science to solve problems such as web crawling, network routing, software dependencies etc. FSMs are used to understand the behaviour of a system either a computer system or physical external system. The nodes represent the unique system states. The vertices represent the transitions move the system from one state to another. Vertices can be initiated by events and are associated with 'inputs'. It is usual that the transition will initiate some action that must be executed as the system transitions between states. Actions can also be executed during the entrance or exit from a state.

The most famous implmentation of a Finite State Machine is the *Turing Machine* [7], which is an FSM that can read and write data to tape. The input that is read from the tape instructs the Turing Machine to transition into another state. A Turing Machine has one interesting difference from a State Machine - it can count because it has storage in the form of its tape.

It is this underlying concept of a set of finite states connected by transitions/actions that allows us to model complex systems and implement workflow solutions.

WHAT IS SERVERLESS

Martin Fowler, a leading author and expert on software analysis and design defines Serverless Computing as encompassing two different but overlapping areas:

1 “Serverless was first used to describe applications that significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state.”

2 “These are typically “rich client” applications that use the vast ecosystem of cloud-accessible storage, authentication services, and so on. These types of services have been previously described as “Backend as a Service" or BaaS.” [1]

The term "Serverless" has become synonymous with the FaaS concept and that is what this Literature Review will cover.

Fowler also makes a significant observation about Serverless architectures which is particularly applicable to the project use-case that I have chosen to investigate. He states that whereas for traditional 3-tier applications *"all flow, control, and security was managed by the central server application. In the Serverless version there is no central arbiter of these concerns. Instead we see a preference for choreography over orchestration, with each component playing a more architecturally aware role"*[1].

While the predominant pattern for microservice/servlerless applications is choreography, many applications build on Microservice/Serverless platforms will requires orchestration of tasks. Services like AWS Step Functions are implemented using the concept of a State Machines which are designed to coordinate tasks and will ensure application correctness by using retries, wait timers, and rollbacks. Workflows are by their nature asynchronous and therefore should not be used to process synchronous requests.

Fowler notes the well know benefits of this approach (in common with Microservices patterns) result in systems that "are often more flexible and amenable to change, both as a whole and through independent updates to components;there is better division of concerns; and there are also some fascinating cost benefits[1]". He notes that "whether the benefits of flexibility and cost are worth the added complexity of multiple backend components is very context dependent.[1]"

Fowler also decomposes "Functions as a Service" into a set of practical characteristics [1].

- "Running backend code without managing your own server systems or your own long-lived server applications".
- Essentially FaaS replaces a long running process possibly on a physical machine with something "that doesn't need a provisioned server, nor an application that is running all the time".
- "FaaS offerings do not require coding to a specific framework or library. FaaS functions are regular applications when it comes to language and environment."
- "In a FaaS environment we upload the code for our function to the FaaS provider, and the provider does everything else necessary for provisioning resources, instantiating VMs, managing processes, etc."
- "Horizontal scaling is completely automatic, elastic, and managed by the provider."
- The "compute containers" executing your functions are ephemeral, with the FaaS provider creating and destroying them purely driven by runtime need.
- "Functions in FaaS are typically triggered by event types defined by the provider."
- "Most providers also allow functions to be triggered as a response to inbound HTTP requests"

Fowler also details FaaS restrictions under the following headings:

1 State: FaaS functions as transient and "any state of a FaaS function that is required to be persistent needs to be externalized"[1].

2 Execution duration: Each FaaS function execution is time limited. "This means that certain classes of long-lived tasks are not suited to FaaS functions without re-architecture"[1].

3. Cold Starts: Every FaaS platform must start and "initialize an instance of a function before each event" [1].

Fowler asks the question whether Serverless FaaS functions are a specialisation of PaaS - "If your PaaS can efficiently start instances in 20ms that run for half a second, then call it serverless." [1]

"The key operational difference between FaaS and PaaS is scaling" [1]. When using PaaS consideration must be given to how your to scale, e.g. how many VMs are required. These considerations are completely transparent with a FaaS application. Even PaaS with some form of autoscaling will not achieve scaling at the level of individual requests.

"A true DevOps culture becomes even more important in a Serverless world since those other non-sysadmin activities still need to get done, and often it's developers who are now responsible for them. These activities may not come naturally to many developers and technical leads, so education and close collaboration with operations folk is of utmost importance." [1]

According to the Berkeley View on Servlerless Computing [3], forms of Serverless first appeared in 2009. At the time there were there were two competing approaches to virtualization in the cloud :

"*Amazon EC2 is at one end of the spectrum. An EC2 instance looks much like physical hardware, and users can control nearly the entire software stack, from the kernel upward. ... At the other extreme of the spectrum are application domain-specific platforms such as Google App Engine ... enforcing an application structure of clean separation between a stateless computation tier and a stateful storage tier. App Engine's impressive automatic scaling and high-availability mechanisms ... rely on these constraints." [3]*

"The marketplace eventually embraced Amazon's low-level virtual machine approach to cloud computing" [3] they posit because in the "early days of cloud computing users wanted to recreate the same computing environment in the cloud that they had on their local computers to simplify porting their workloads." "The long list of low-level virtual machine management responsibilities inspired customers with simpler applications to ask for an easier path to the cloud for new applications." Recognition of these needs led to a new option from Amazon in 2015 called the AWS Lambda service. [3]

The Berkeley authors agree with Fowler that Servlerless computing encompasses both FaaS and BaaS offerings.

"While cloud functions—packaged as FaaS (Function as a Service) offerings — represent the core of serverless computing, cloud platforms also provide specialized serverless frameworks that cater to specific application requirements as BaaS (Backend as a Service) offerings [7]. Put simply, serverless computing = FaaS + BaaS. In our definition, for a service to be considered serverless, it must scale automatically with no need for explicit provisioning, and be billed based on usage." [3]

Cloud functions (i.e., FaaS) provide general compute and are complemented by an ecosystem of specialized Backend as a Service (BaaS) offerings such as object storage, databases, or messaging. Specifically, a serverless application on AWS might use Lambda with S3 and DynamoDB, while an application on Google's cloud might use Cloud Functions with Cloud Pub/Sub. [3]

The Berkeley authors provide a table of the contrasting characteristics of Serverless vs Serverless Cloud, shown below in Figure 1/Table 2 [3].

	Characteristic	*AWS Serverless Cloud*	*AWS Serverful Cloud*
PROGRAMMER	When the program is run	On event selected by Cloud user	Continuously until explicitly stopped
	Programming Language	JavaScript, Python, Java, Go, C#, etc.[4]	Any
	Program State	Kept in storage (stateless)	Anywhere (stateful or stateless)
	Maximum Memory Size	0.125 - 3 GiB (Cloud user selects)	0.5 - 1952 GiB (Cloud user selects)
	Maximum Local Storage	0.5 GiB	0 - 3600 GiB (Cloud user selects)
	Maximum Run Time	900 seconds	None
	Minimum Accounting Unit	0.1 seconds	60 seconds
	Price per Accounting Unit	$0.0000002 (assuming 0.125 GiB)	$0.0000867 - $0.4080000
	Operating System & Libraries	Cloud provider selects[5]	Cloud user selects
SYSADMIN	Server Instance	Cloud provider selects	Cloud user selects
	Scaling[6]	Cloud provider responsible	Cloud user responsible
	Deployment	Cloud provider responsible	Cloud user responsible
	Fault Tolerance	Cloud provider responsible	Cloud user responsible
	Monitoring	Cloud provider responsible	Cloud user responsible
	Logging	Cloud provider responsible	Cloud user responsible

Table 2: Characteristics of serverless cloud functions vs. serverful cloud VMs divided into programming and system administration categories. Specifications and prices correspond to AWS Lambda and to on-demand AWS EC2 instances.

The Berkeley authors define precisely three critical distinctions betweeen Serverless and Serverful computing [2]:

1. Decoupled computation and storage. The storage and computation scale separately and are provisioned and priced independently.

2. Executing code without managing resource allocation. Instead of requesting resources, the user provides a piece of code and the cloud automatically provisions resources to execute that code.

3. Paying in proportion to resources used instead of for resources allocated.

"Google's original App Engine, largely rebuffed by the market just a few years before serverless computing gained in popularity, also allowed developers to deploy code while leaving most aspects of operations to the cloud provider. We believe serverless computing represents significant innovation over PaaS and other previous models ." [3] However, modern Serverless offerings are far more advanced in several important factors: better autoscaling, strong isolation, and ecosystem support. [3]

For example they mention how the Autoscaling provided by AWS Lambda tracks the load with much more accuracy that EC2 autoscaling, so customers are billed in a more fine-grained manner of per 100 ms rather than minutes or hours. Customers are charged for execution time rather than per resource provisioned.

The Berkeley authors note that this *"ensured the cloud provider had skin in the game on autoscaling, and consequently provided incentives to ensure efficient resource allocation".* [3]

Martin Fowler outlines a list of more prosaic, less thoughtful concerns with the current serverless platforms. He is concerned with the lack of standard API resulting in Vendor control / Lock In. The repetition of logic across client platforms that results for the use a only BaaS services on the server side. Multiple client side platforms will need to duplicate logic, e.g. iOS, Android, JavaScript, etc. Also Fowler shares the concern of both sets of Berkeley authors for the lack of in-server state for Serverless FaaS. [1]

WORKFLOW SERVICES IN THE CLOUD

"A workflow consists of an orchestrated and repeatable pattern of activity, enabled by the systematic organization of resources into processes that transform materials, provide services, or process information. It can be depicted as a sequence of operations, the work of a person or group, the work of an organization of staff, or one or more simple or complex mechanisms.

From a more abstract or higher-level perspective, workflow may be considered a view or representation of real work.[3] The flow being described may refer to a document, service, or product that is being transferred from one step to another." [8]

The principle of process improvement methodologies such as Sigma 6 [9] and Lean [10] are modeled using workflows.

The American Scientific Research Journal paper *Serverless Computing and Scheduling Tasks* states "With the massive growth of data, scientists in different arenas of science, astronomy, engineering and physics are developing broad applications. These applications normally consist of dependent tasks that form workflows". [6]

"Such workflows are typically complex and extensive in nature, since they consist of large number of processes with extended number of tasks that have dependency constraints. This kind of complex applications require a high performance environment so that they can be performed within a given time constraint" [6]

"Applications workflows can be represented as a Directed Acyclic Graph (DAG), in which each computational task is denoted by a node, and the relation between the precedence-constrained tasks is denoted by directed edges"[6]

Workflow services have many advantages for certain problem domains that require task coordination.

1 The underlying execution is easily to model and reason as a Directed Graph of tasks.

2 The state of individual workflow execution processes is measurable and observable.

3 Workflow platforms provide logging, error handling, resilience etc

4 Workflow platforms manage data state and transmission between independent, individual tasks.

Workflow platforms such as Microsoft Logic Apps also allow non technical users to build (via user-friendly interfaces) moderately complex applications by organising predefined Microsoft services and resources such as Email, Authentication, Office, SQL etc into a sequence of dependent tasks.

Serverless Architectures are designed to be event driven in common with Microservices best practice. However, for many situations the solution required coordinating a series of API calls while maintaining state and this is the problem domain for the workflow pattern.

LogicApps was one of the first successful cloud workflow service, targeting developers and everyone else with a graphical workflow editor and a simple JSON definition language. It provided over 100 ready-to-use connectors to integrate with APIs, Microsoft products, databases, etc.

AWS then introduced StepFunctions, which is gaining market share but suffered initially from well documented operational problems: the execution history is lost if the workflow is amended in any way, it also has no support for workflow patterns like branching and joins. Its definition language is JSON based, simple, and concise, providing data referencing, error-handling with retries and catches.

Google Cloud Platform basically offers a managed environment wrapped around the open-source AirFlow platform called Cloud Composer. AirFlow has a good reputation and uses a Python domain specific language (DSL) for workflow definition. It has good extensibility hooks and provides good cross task communication.

AWS StepFunctions and Azure LogicApps are both charge based on state changes/function calls. Google Composer charges based on the provisioning of VM/Storage resources required to run an underlying AirFlow platform. This seems to disqualify it from the Servlerless category.

THE FUTURE OF SERVERLESS

The following two academic papers from UC Berkeley reflect very different viewpoints on the current state and future of Severless computing:

1. Serverless Computing: One Step Forward, Two Steps Back. J. Hellerstein. 2019
2. A Berkeley View on Serverless Computing. E. Jonas. 2019

Hellerstein is quite critical of the gaps in the current generation of serverless platforms and states that their design is at odds with dominant trends such as data centric and distributed computing. He believes that the current offerings will retard cloud innovation for cloud and data systems.

Hellerstein believes that the cloud today is largely used as an "outsourcing platform for standard enterprise data services, the majority of cloud services are simply multi-tenant, easier-to-administer clones of legacy enterprise data services like object storage, databases, queueing systems, and web/app servers". [4]

"A FaaS offering by itself is of little value, since each function execution is isolated and ephemeral. Building applications on FaaS requires data management in both persistent and temporary storage, in addition to mechanisms to trigger and scale function execution." [4]

The two steps backwards that are referred to by Hellerstein et al, are that the FaaS offerings "ignore the importance of efficient data processing and retard the development of distributed systems which at the core of most current innovations in modern computing." [4]

Hellertein outlines three categories of applications which he believes suit the current suite of FaaS platforms [4].

Embarrassingly parallel functions: These exploit auto-scaling features because independent requests never communicate with each other and require only small amounts of computation.

Orchestration functions: He mentions that Lambda functions are mostly used to preprocess event streams before channelling them to storage or data analytics where the "heavy lifting" of the computation is done, not by Lambda functions.

Function Composition: These are collections of functions that are composed to build applications including workflows of functions chained together via data dependencies.

Thus, he proposes that the current generation of FaaS solutions are suitable only for simple workloads of independent tasks.

The researchers critisize the FaaS offerings using these observations. [4]

FaaS is a Data-Shipping Architecture: “shipping data to code” is an architectural antipattern among system designers.

FaaS Stymies Distributed Computing: FaaS has no network addressability, so functions cooperate only by passing data through slow and expensive storage. There is currently no mechanism for thousands of cores to work together efficiently.

Discourages Open Source innovation: Most popular open source software cannot not run at scale in current serverless offerings. Open-source data systems are be impossible to build on current FaaS offerings which locks users into either using proprietary provider services or maintaining their own server.

Hellerstein et all, then outline three problem domains to illustrate the limitations and high costs associated with FaaS [4]:

1 **Model Training Algorithms**: Lambda is 21× slower and 7.3× more expensive than running on EC2.

2 **Low-Latency Prediction Serving via Batching:** prediction serving relies on access to specialized hardware like GPUs, which are not available through AWS Lambda. A “serverful” implementation had a per batch latency of 2.8ms—127× faster than the optimized Lambda implementation. To scale this application to 1 million messages a second, the SQS request rate alone would cost $1,584 per hour.

3 **Distributed Computing:** Lambda forbids direct network connectivity between functions, so we are forced to try alternative solutions to achieve distributed computation.

Hellerstein believes that old enterprise products will migrate to new, more efficient cloud-based vendors but this type of application problem domain will not drive innovation in the cloud. It will discourage open source development and allow the cloud vendors increase market dominance for their proprietary solutions.

Hellerstein's colleagues at UC Berkeley Jonas et al, in their paper “A Berkeley View of Serverless Computing” share some of his observations but have a more optimistic attitude. Their research

corroborates Hellerstein's assertion that the vast majority of applications currently using Serverless platforms are the trival web/api, data processing use cases. See Figure 2/Table 4.

Percent	Use Case
32.00%	Web and API serving
21.00%	Data Processing, e.g., batch ETL (database Extract, Transform, and Load)
17.00%	Integrating 3rd Party Services
16.00%	Internal tooling
8.00%	Chat bots e.g., Alexa Skills (SDK for Alexa AI Assistant)
6.00%	IOT

Table 4: Popularity of serverless computing use cases according to a 2018 survey [3].

Jonas et al, state that *"Customers benefit from increased programming productivity, and in many scenarios can enjoy cost savings as well, a consequence of the higher utilization of underlying servers. Even if serverless computing lets customers be more efficient, the Jevons paradox suggests that they will increase their use of the cloud rather than cut back as the greater efficiency will increase the demand by adding users."* [3]

They identify similar use cases to Hellerstein, such as high performance databases that rely on shared memory. FaaS components run in isolation so cannot be used. Distributed databases do not share memory but rely on the assumption that nodes remain online and are directly addressible. Neither of those premises hold for Serverless components. They thus rule out implementing databases in a FaaS platform and expect it to remain BaaS.

Jonas identifies four limitations of the current generation of serverless offerings [3].

Inadequate storage for fine-grained operations. This is due to the limitations of cloud storage services. Object storage services (AWS S3, Azure Blob Storage) are scalable with cheap long-term object storage, but have high access costs and latencies. Key-value databases (DynamoDB, Google Cloud Datastore) provide high IOPS, but are expensive and can take a long time to scale up. In-memory storage instances such as Memcached or Redis are not fault tolerant and cannot autoscale.

Lack of fine-grained coordination. To support stateful applications, serverless frameworks must provide a coordination mechanism.

Poor support for Communication patterns. With VM solutions such as EC2, all tasks on the instance share the copy of the data, the communication complexity of the broadcast and aggregation is O(N), where N is the number of VMs. However, with FaaS this complexity is O(N × K), where K is the number of functions per VM.

Predictable Performance. The variability of hardware resources that results from giving the cloud provider flexibility to choose the underlying server.

With regard to the second limitation mentioned above, the authors seem to have overlooked the Workflow/Orchestration services that have been implemented by Cloud vendors (such as Overlooked Step Functions, Cloud Orchestrator) to address the coordination of stateless services to build stateful applications.

Jonas et al, "predict that Serverless use will skyrocket" and that the current offerings are an "important maturation" in the evolution of cloud computing [3]. This is a divergent opinion to their colleagues Hellerstein et al, who also believe that the use of Serverless will increase but that the underlying architecture is not a maturation but a diversion.

They conclude with the following more specific predictions [3]:

• Next generationBaaS storage to allow more types of applications to adopt serverless computing. Such storage will match block storage performance and allow ephemeral and durable storage.

• Secure Serverless computing to become simpler to program.

• Billing models change to allow any application, at any scale, to cost less with serverless computing.

• Applications that cannot be developed on serverless, such as OLTP databases will be offered as services from all cloud providers.

• Serverless computing will become the default computing paradigm of the Cloud Era.[3]

REFERENCE LIST AND BIBLIOGRAPHY

[1] Serverless Architectures. https://martinfowler.com/articles/serverless.html

[2] A Berkeley View on Serverless Computing.
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf

[3] Cloud Programming Simplified: A Berkeley View on Serverless Computing.
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

[4] Serverless Computing: One Step Forward, Two Steps Back. UC Berkely.
http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[5] Finite State Machine https://en.wikipedia.org/wiki/Finite-state_machine

[6] American Scientific Journal. Serverless Computing and Sceduling Tasks in the Cloud.
https://asrjetsjournal.org/index.php/American_Scientific_Journal/article/view/3913/1403

[7] Turing Machine. https://en.wikipedia.org/wiki/Turing_machine (15 Oct, 2019)

[8] Workflow. https://en.wikipedia.org/wiki/Workflow (16 Oct, 2019)

[9] Sigma Six. https://en.wikipedia.org/wiki/Six_Sigma (15 Oct, 2019)

[10] Lean. https://en.wikipedia.org/wiki/Lean_thinking (16 Oct, 2019)

Cloud Services FAQs

Google Cloud, Composer. https://cloud.google.com/composer/ (10 Oct, 2019)

Apache Airflow. https://airflow.apache.org/ (10 Oct, 2019)

Google Cloud, Functions. https://cloud.google.com/functions/ (10 Oct, 2019)

Google Cloud, Pub/Sub. https://cloud.google.com/pubsub/docs/ (10 Oct, 2019)

Google Cloud, Datastore. https://cloud.google.com/datastore/ (10 Oct, 2019)

Azure, Logic Apps. https://azure.microsoft.com/en-us/services/logic-apps/ (11 Oct, 2019)

Azure, Functions. https://azure.microsoft.com/en-us/services/functions/ (11 Oct, 2019)

Azure, Event Grid. https://azure.microsoft.com/en-us/services/event-grid/ (11 Oct, 2019)

Azure, Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/ (11 Oct, 2019)

Azure, ComosDB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction/ (11 Oct, 2019)

AWS, Step Functions. https://aws.amazon.com/step-functions/ (12 Oct, 2019)

AWS, Lambda. https://aws.amazon.com/lambda/ (12 Oct, 2019)

AWS, SNS. https://aws.amazon.com/sns/ (12 Oct, 2019)

AWS, SQS. https://aws.amazon.com/sqs/ (12 Oct, 2019)

AWS, Dynamo DB. https://aws.amazon.com/dynamodb/ (12 Oct, 2019)

Cloud Services Pricing

Google Cloud, Composer. https://cloud.google.com/composer/pricing/ (10 Oct, 2019)

Google Cloud, Functions. https://cloud.google.com/functions/pricing/ (10 Oct, 2019)

Google Cloud, Pub/Sub. https://cloud.google.com/pubsub/pricing/ (10 Oct, 2019)

Google Cloud, Datastore. https://cloud.google.com/datastore/pricing (10 Oct, 2019)

Google Cloud, Free Limits. https://cloud.google.com/free/ (10 Oct, 2019)

Azure, Logic Apps. https://azure.microsoft.com/en-us/pricing/details/logic-apps/ (11 Oct, 2019)

Azure, Functions. https://azure.microsoft.com/en-us/pricing/details/functions/ (11 Oct, 2019)

Azure, Event Grid. https://azure.microsoft.com/en-us/pricing/details/event-grid/ (11 Oct, 2019)

Azure, Queue Storage. https://azure.microsoft.com/en-in/pricing/details/storage/queues/ (11 Oct, 19)

Azure, ComosDB. https://azure.microsoft.com/en-us/pricing/details/cosmos-db/ (11 Oct, 2019)

Azure, Free Limits. https://azure.microsoft.com/en-us/free/#new-products

AWS, Step Functions. https://aws.amazon.com/step-functions/pricing/ (12 Oct, 2019)

AWS, Lambda. https://aws.amazon.com/lambda/pricing/ (12 Oct, 2019)

AWS, SNS. https://aws.amazon.com/sns/pricing/ (12 Oct, 2019)

AWS, SQS. https://aws.amazon.com/sqs/pricing/ (12 Oct, 2019)

AWS, Dynamo DB. https://aws.amazon.com/dynamodb/pricing/provisioned/ (12 Oct, 2019)

AWS, Free Limits. https://aws.amazon.com/free/

VERSION 1.0

24/10/2019

ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

TECHNICAL ASSESSMENT

TABLE OF CONTENTS

AREAS OF INVESTIGATION

In order to begin the Implementation Phase of the project it will be necessary to undertake a preliminary technical assessment of the available Functions as a Service (FaaS), Serverless and Workflow / Orchestration services that are provided by the main Cloud vendors.

To this end, this document will outline the specific services that are available from Microsoft Azure and Amazon Web Services. To build a working system other services such as NoSQL and Authorisation/Authentication will be required and they will also be covered although in less depth are they are not the focus of the problem domain that is under investigation.

The list of Services are outlined next.

Amazon Web Services provides the following services that would be required:

- AWS Identify and Access Management
- AWS DynamoDB (NoSQL)
- AWS Lambda (Serverless)
- AWS Step Functions (Workflow)

Microsoft Azure provides the following equivalent services:

- Azure Role Based Access Control (RBAC)
- Azure Cosmos DB (NoSQL)
- Azure Functions (Serverless)
- Azure Logic Apps (Workflow)

TECHNICAL ARCHITECTURE

As described above the basic architecture for the project if implemented on the AWS platform would involve the servers shown in the architecture diagram below. (Figure AWS.ARCH.1)

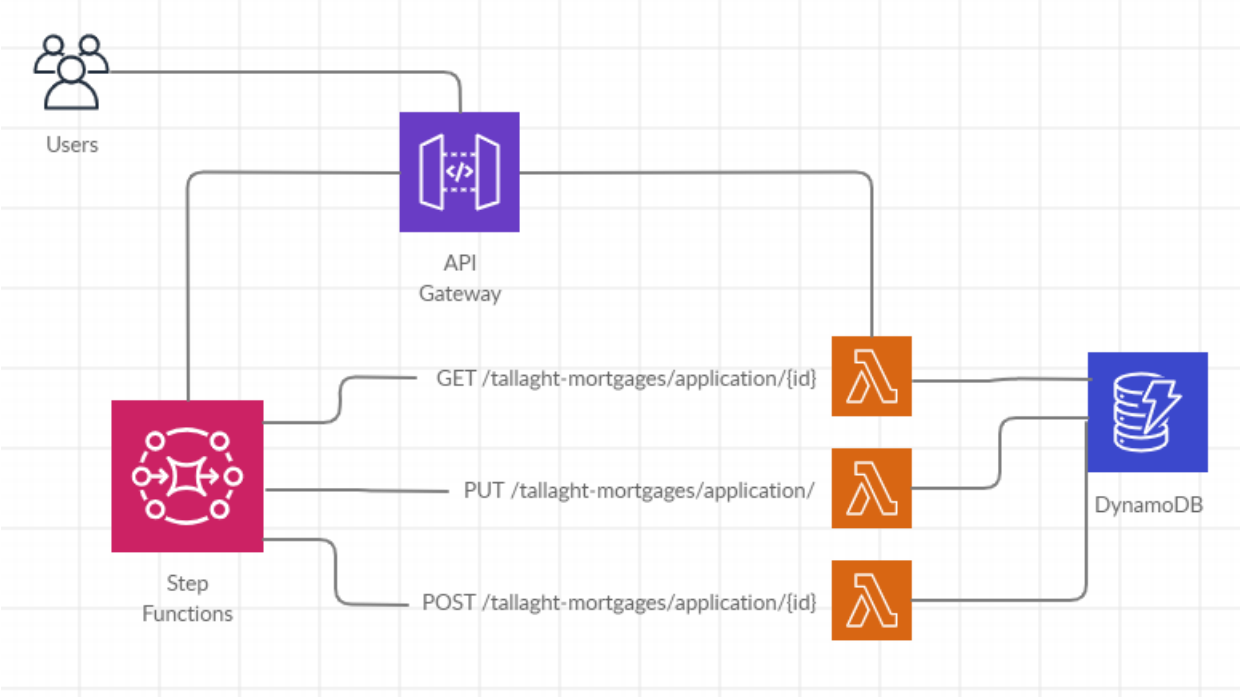


Figure AWS.ARCH.1

AZURE SERVERLESS (FUNCTIONS AS A SERVICE)

The Azure Serverless offering is called Azure Function Apps. In this section I will outline the steps that I have taken to build and deploy a simple component.

Step 1: Navigate to the Microsoft Azure Function App console and select Create. (Figure AZ.FUNC.1)

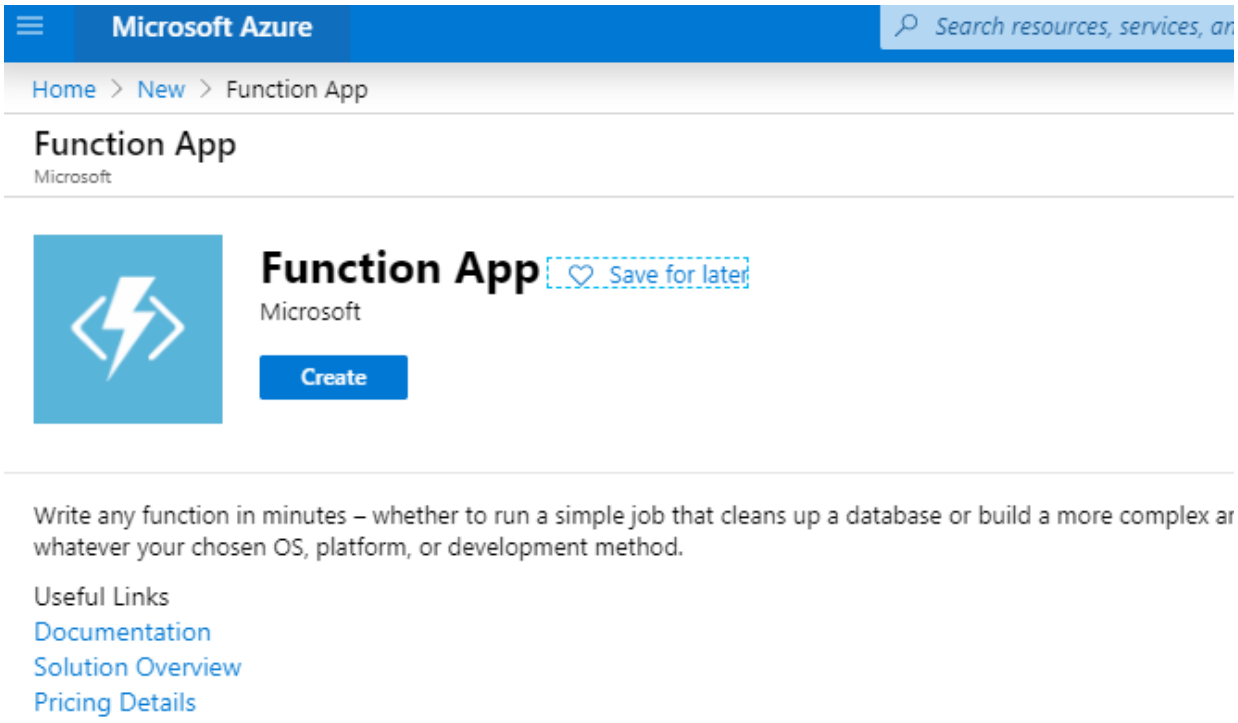


Figure AZ.FUNC.1

Step 2: Create a Resource Group and select a Function App name. (Figure AZ.FUNC.2)

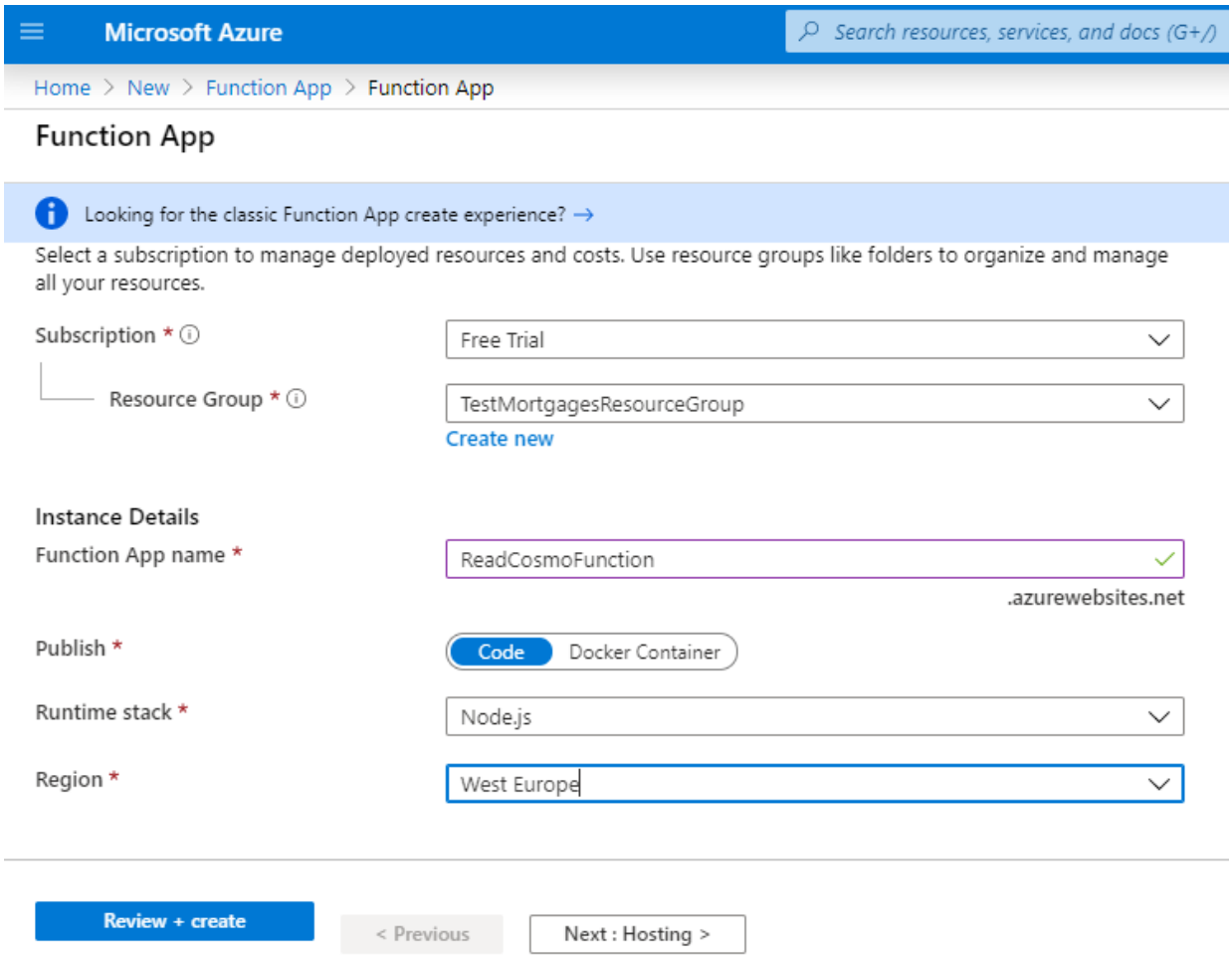


Figure AZ.FUNC.2

Step 3: Select the Create option. (Figure AZ.FUNC.3)

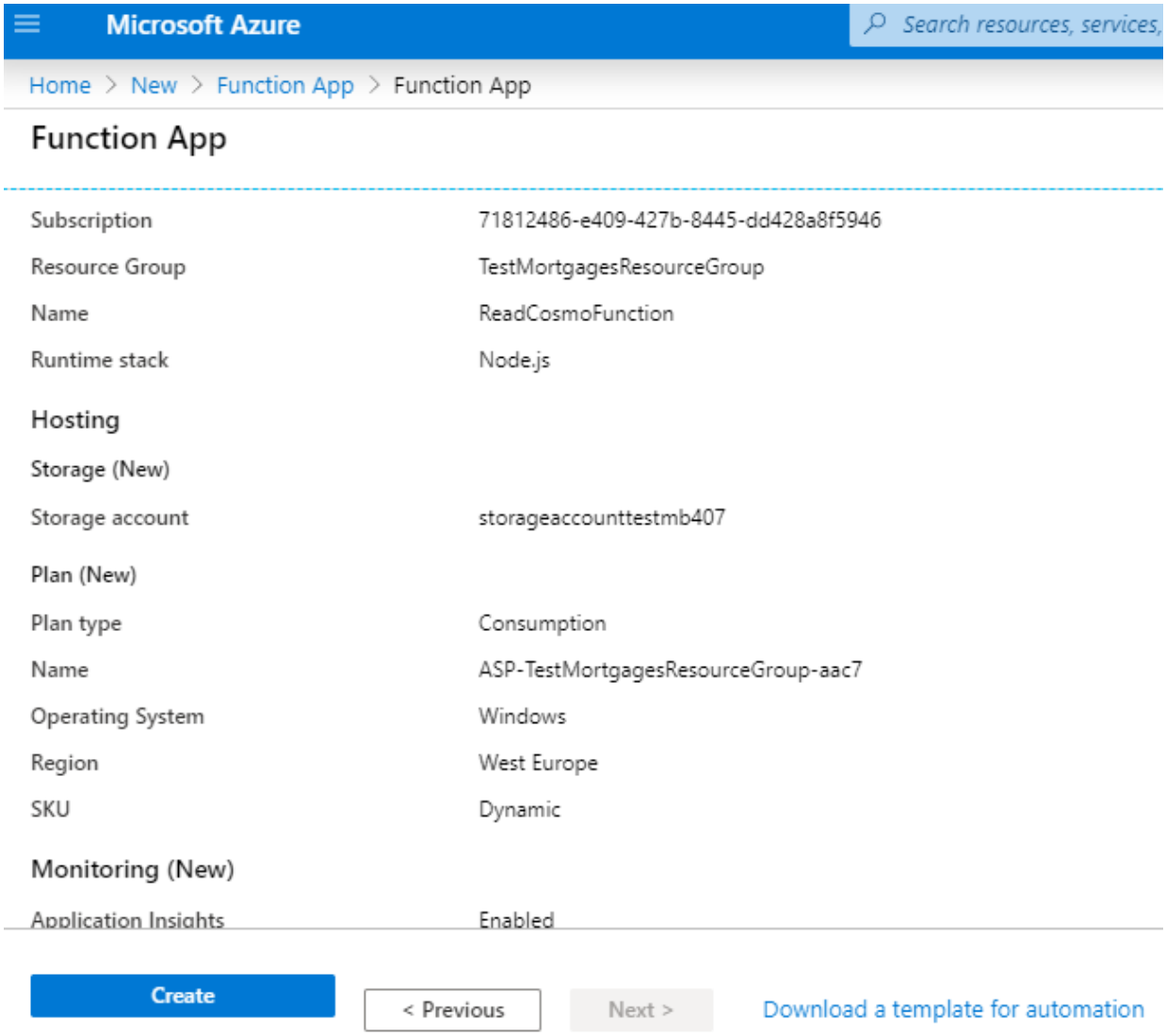


Figure AZ.FUNC.3

Step 4: The Function App should take a few minutes to create. (Figure AZ.FUNC.4)

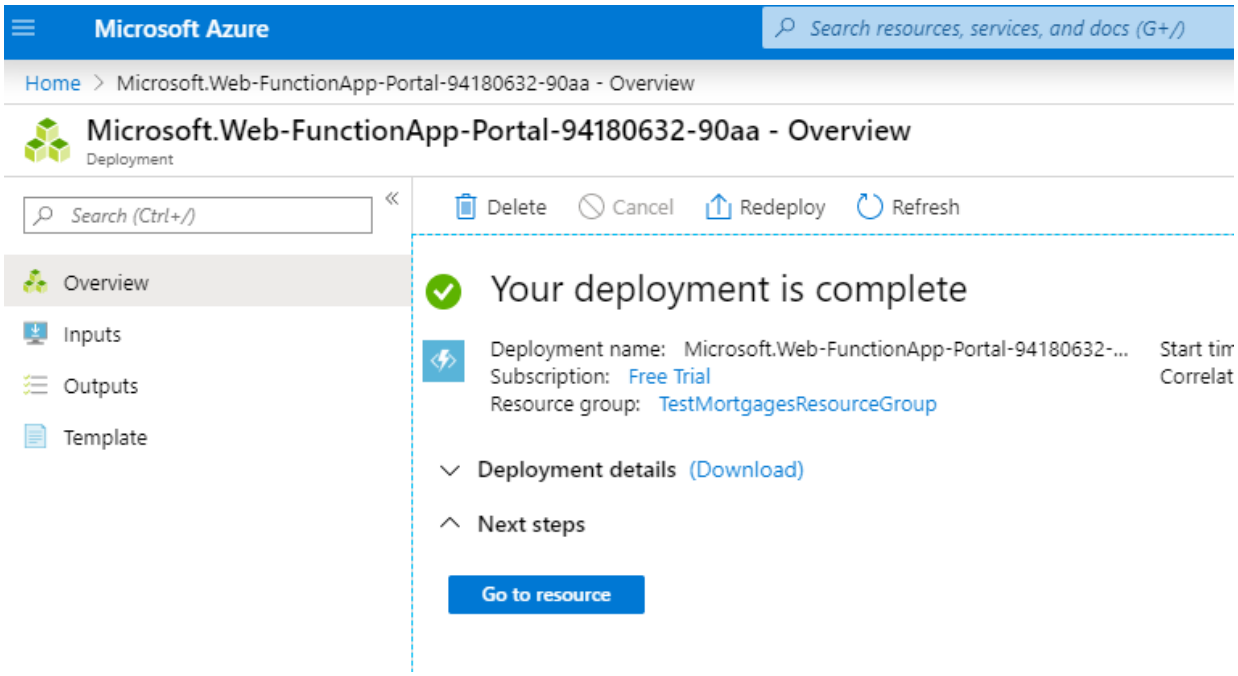


Figure AZ.FUNC.4

Step 5: Select Go to Resource. (Figure AZ.FUNC.5)

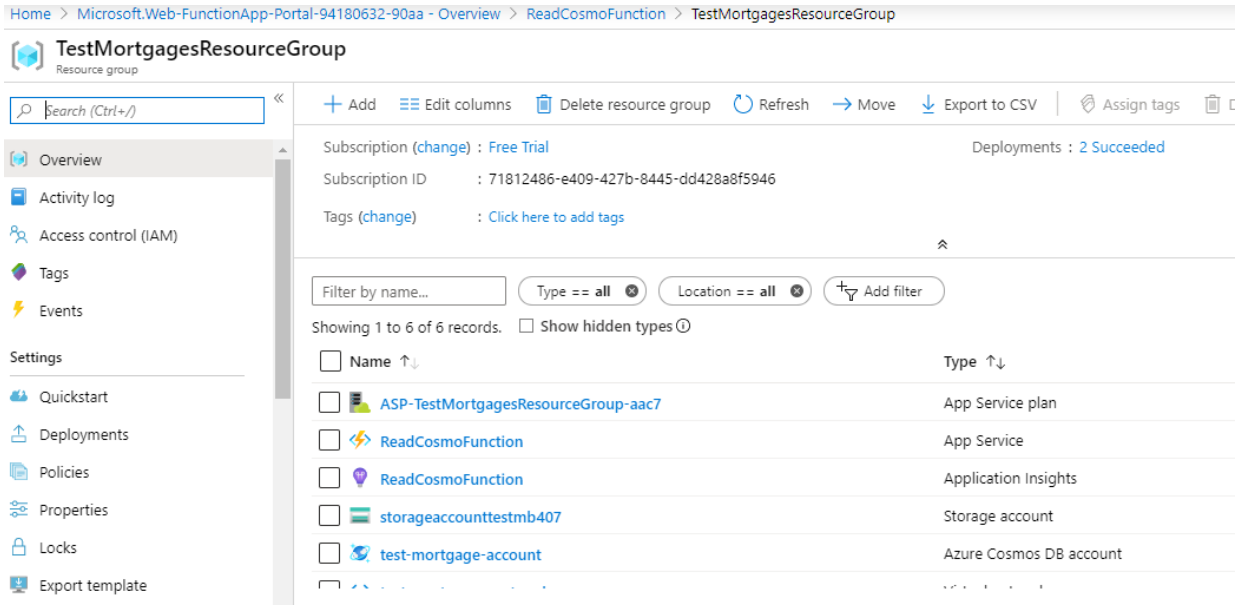


Figure AZ.FUNC.5

Step 6: Add the Function Code. (Figure AZ.FUNC.6)

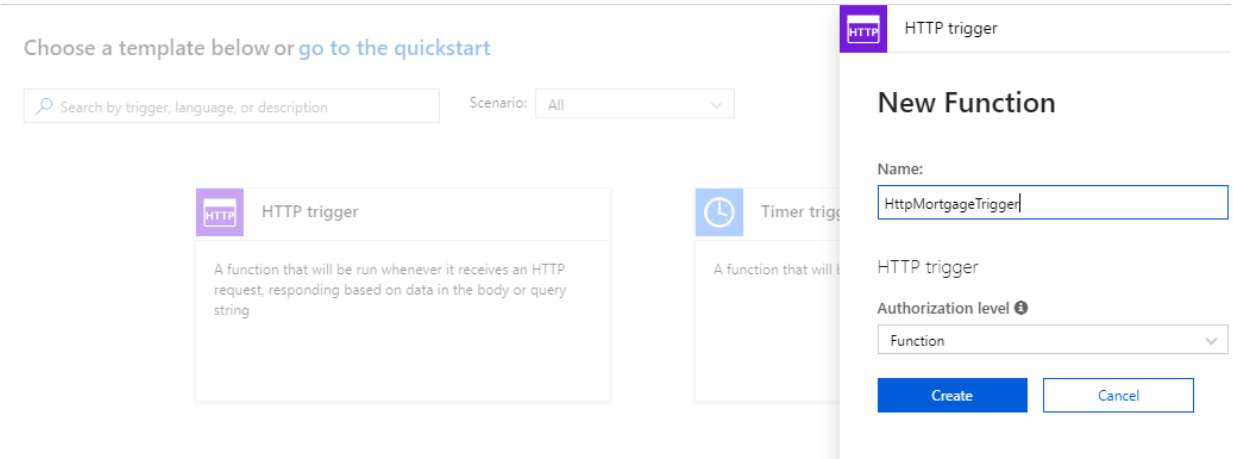


Figure AZ.FUNC.6

Step 7: In order to use NodeJS modules it is necessary to login to the Kudu Configuration terminal for the Function App. (Figure AZ.FUNC.7)

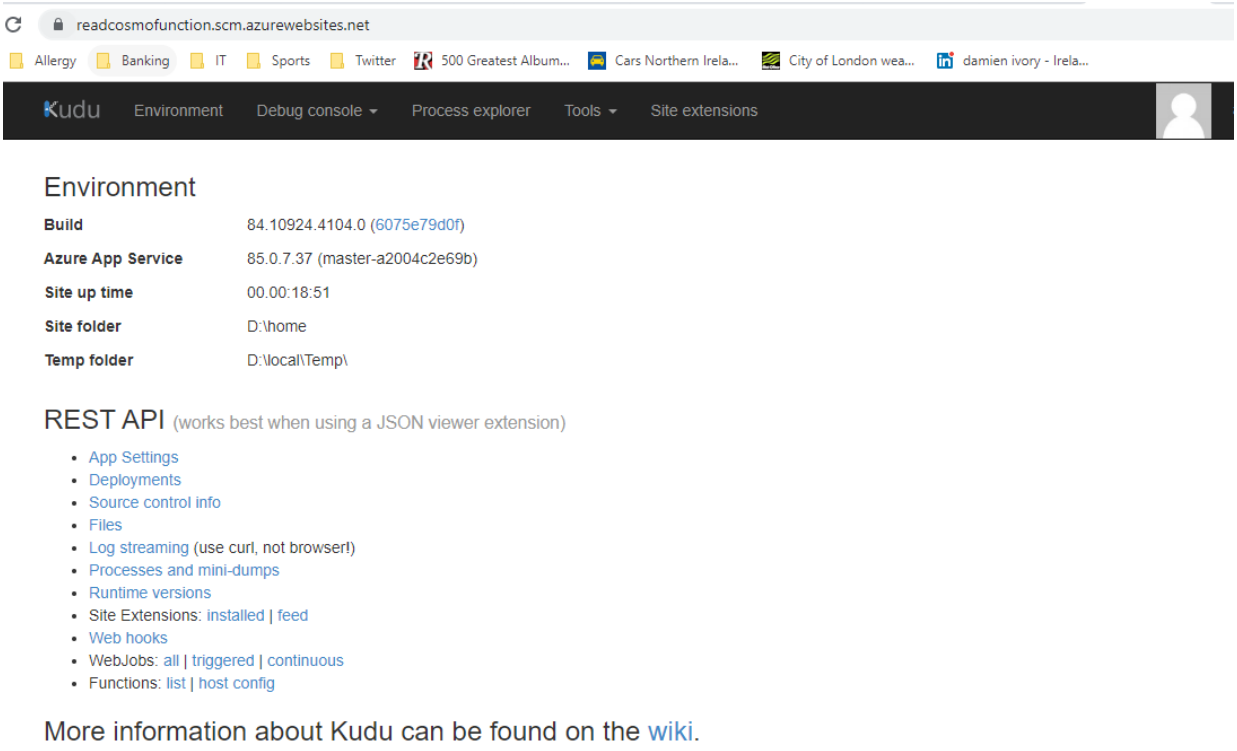


Figure AZ.FUNC.7

Step 8: Create a package.json file in the folder wwwroot/. (Figure AZ.FUNC.8)

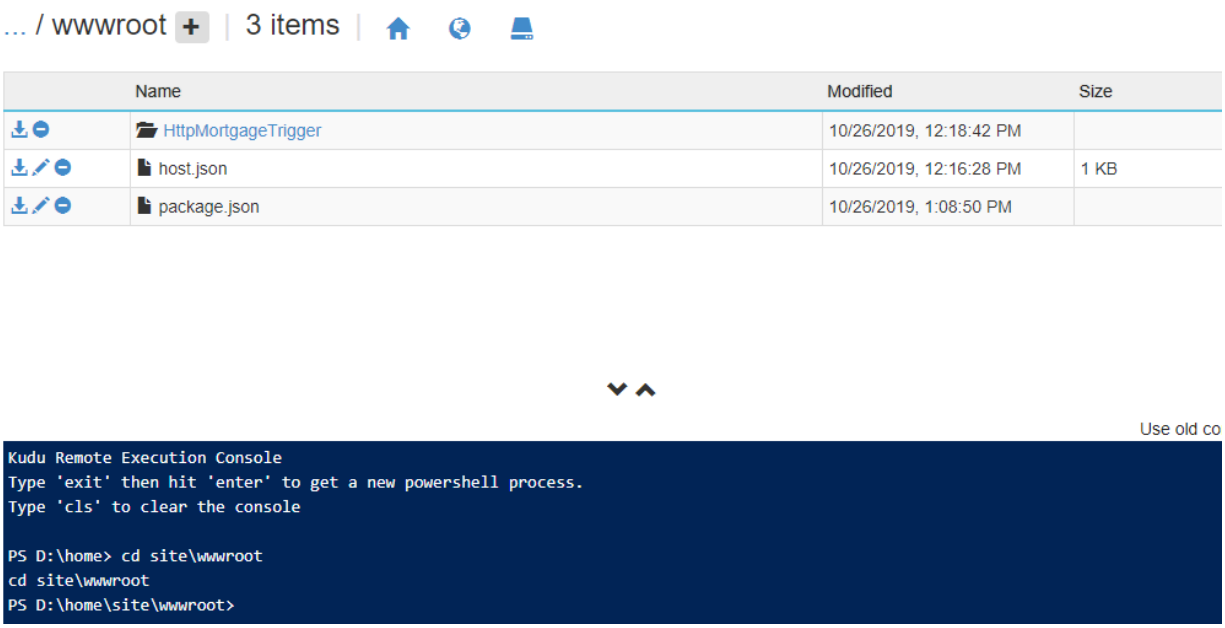


Figure AZ.FUNC.8.

Step 9: Edit the code to connect to the Database server. (Figure AZ.FUNC.9)

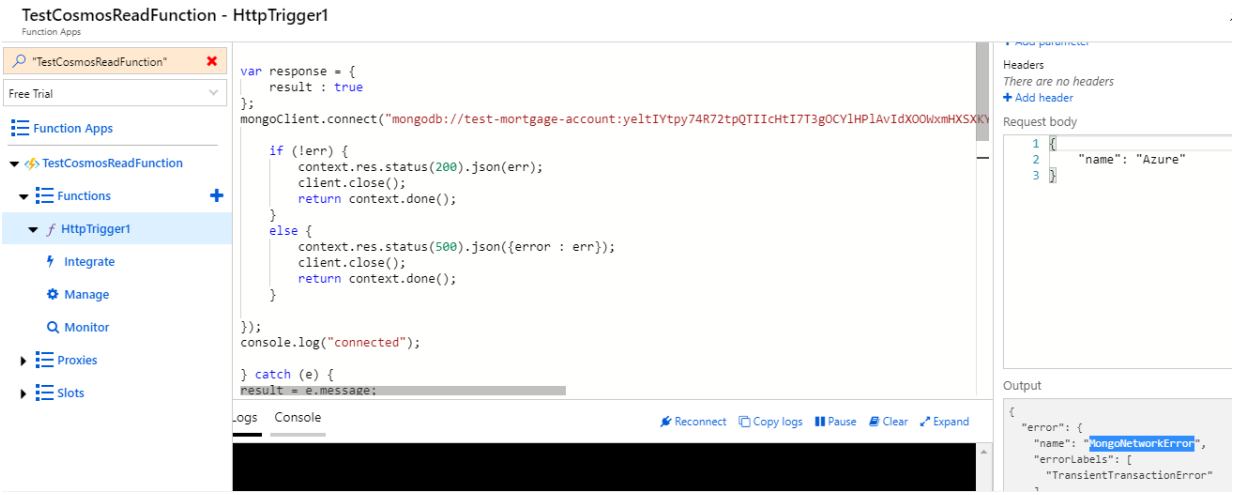


Figure AZ.FUNC.9.

I have been unable to get the code to work successfully before the project deadline. There have been various errors such as:

```
{
  "error": {
    "name": "MongoNetworkError",
    "errorLabels": [
      "TransientTransactionError"
    ]
  }
}
```

Other errors have include “MongoError – Authentcation Error”.

I have found the Azure services, user console, development guides, logging and other features to inferior to the AWS development environment and literature.

My attempt to use the Visual Studio Code IDE and deployment tools did not succeed either. The error produced is seen in Figure AZ.FUNC.9. Apparently Azure Function bindings can only be used with SQL libraries.

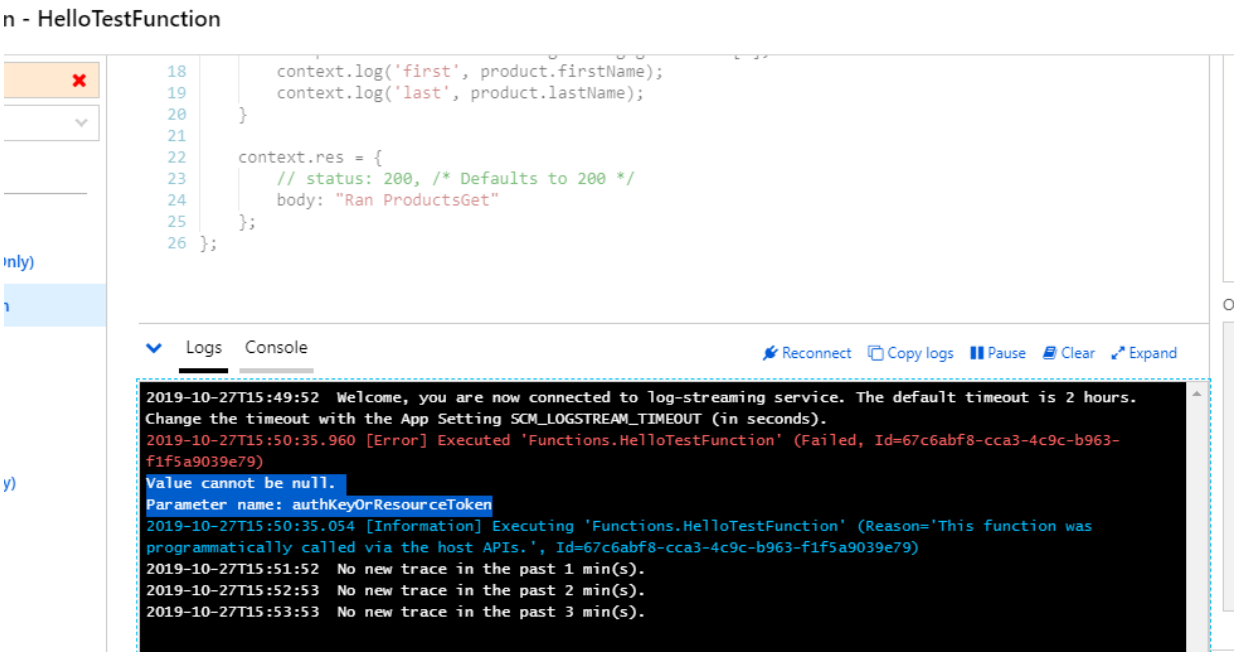


Figure AZ.FUNC.9.

AZURE NOSQL

Step 1: Create a Cosmos DB Account. (Figure AZ.COSMOS.1)

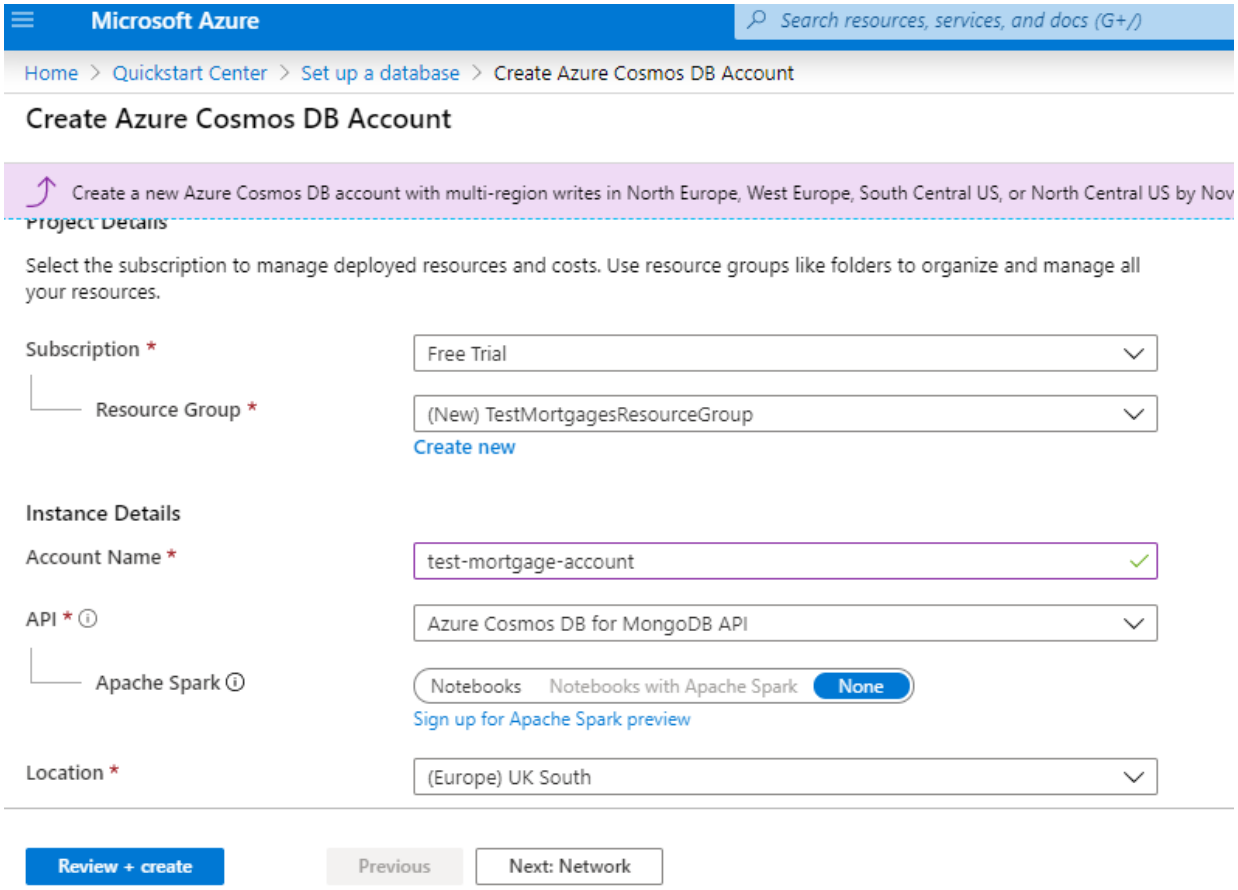


Figure AZ.COSMOS.1

Step 2: Configure the Virtual Network and Firewall. (Figure AZ.COSMOS.2)

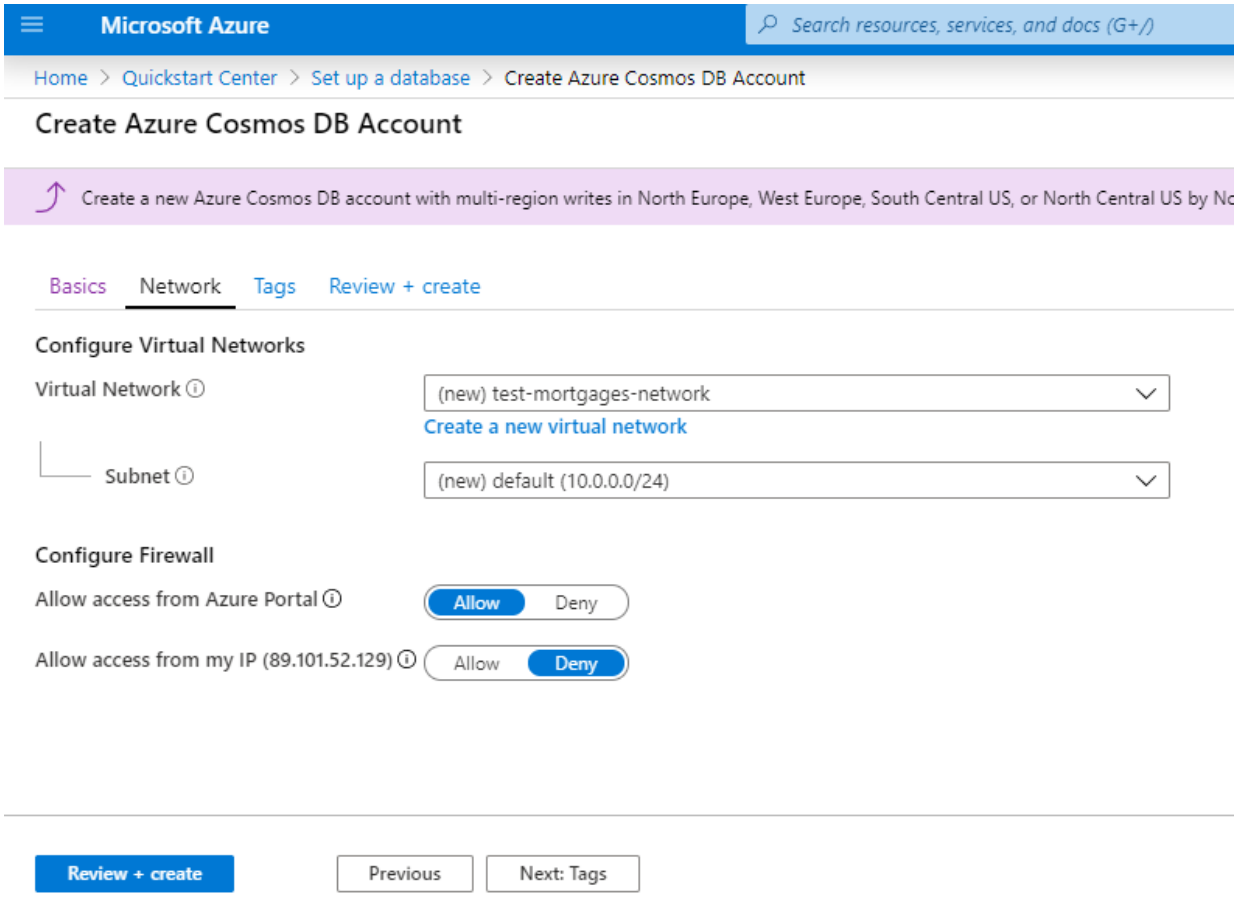


Figure AZ.COSMOS.2

Step 3: Review the Cosmos DB Account. (Figure AZ.COSMOS.3)

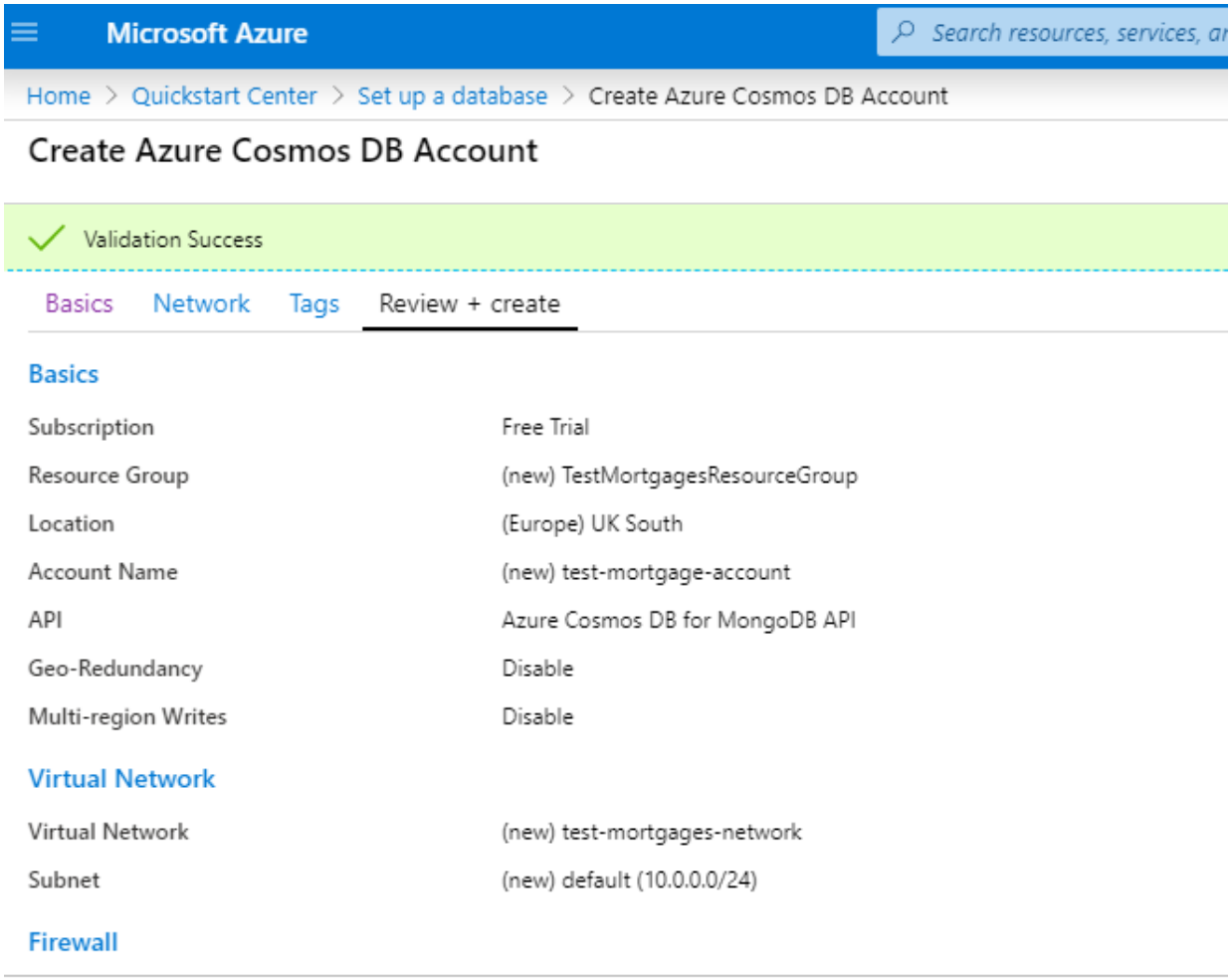


Figure AZ.COSMOS.3.

Step 4: The deployment will take a number of minutes to complete. (Figure AZ.COSMO.4)

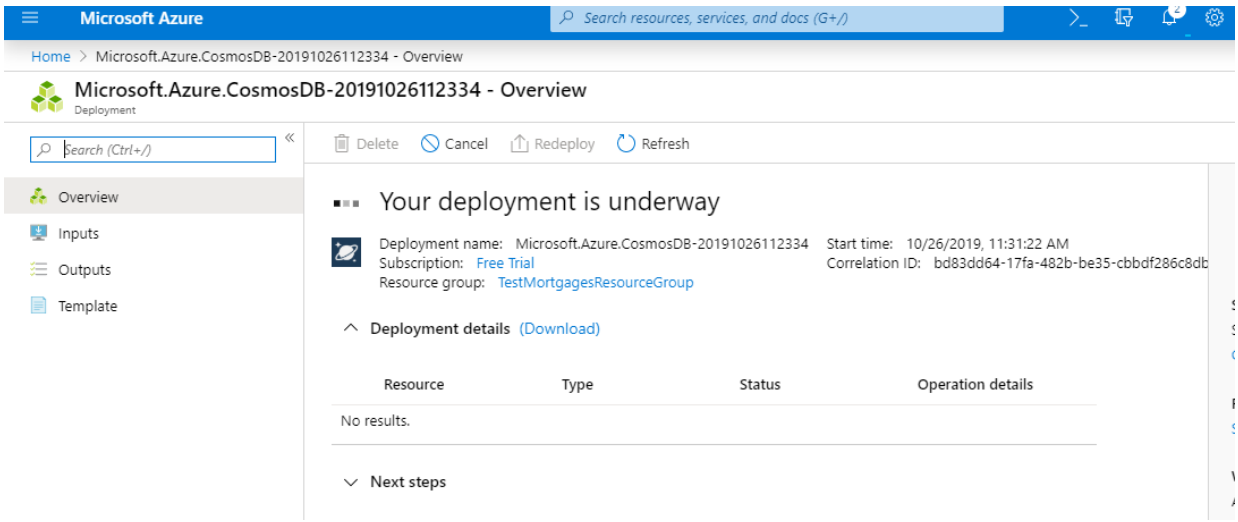


Figure AZ.COSMO.4

Step 5: Once the deployment is complete, select Go to resource. (Figure AZ.COSMOS.5)

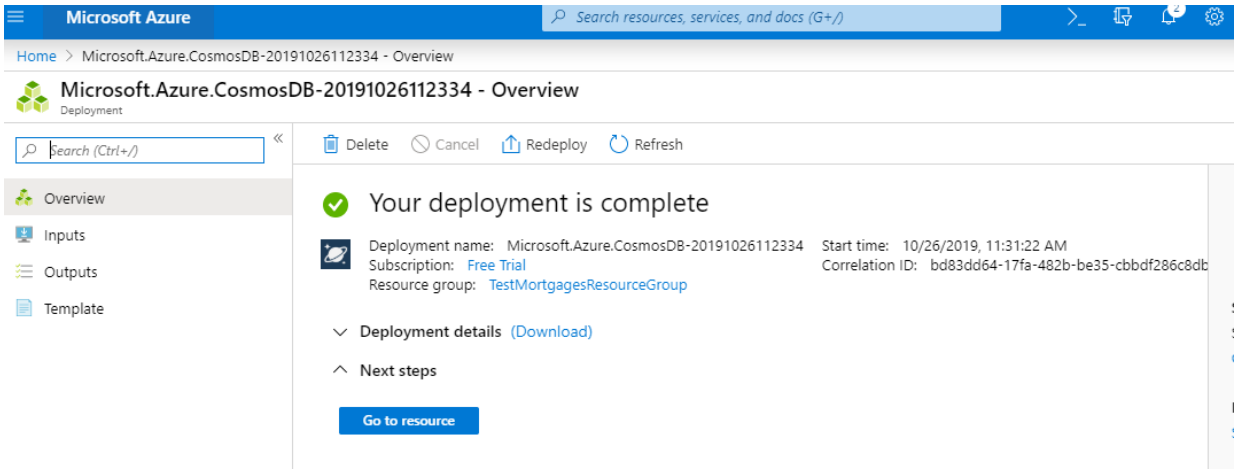


Figure AZ.COSMOS.5

Step 6: Select the NodeJS platform in order to obtain starter code that will allow us to connect to the database. (Figure AZ.COSMOS.6)

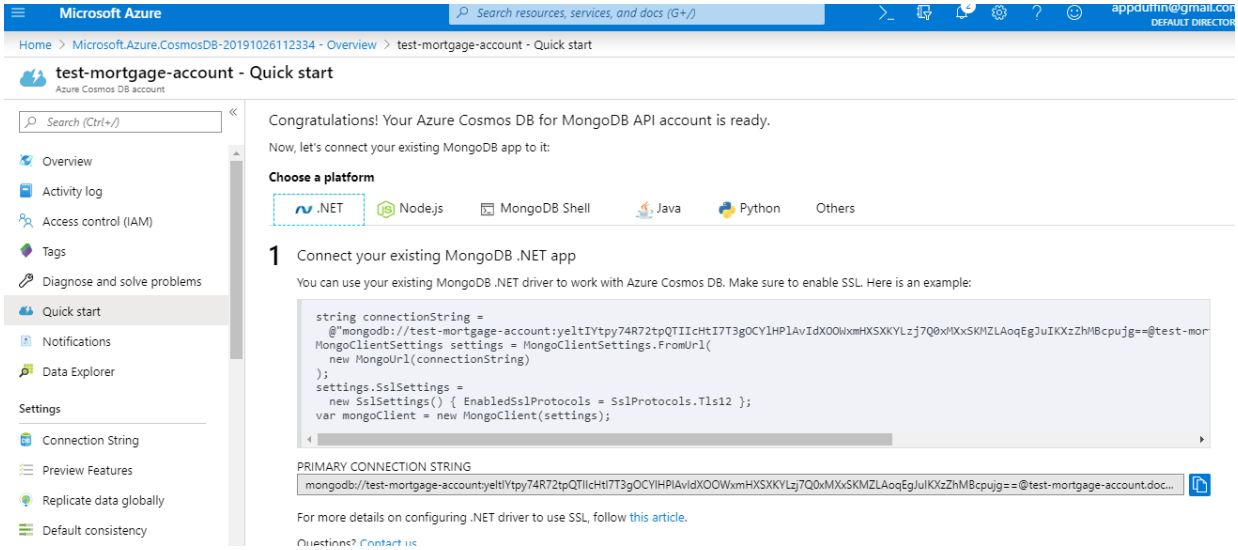


Figure AZ.COSMOS.6

Step 7: Add a Collection to the new NoSQL database. (Figure AZ.COSMOS.7)

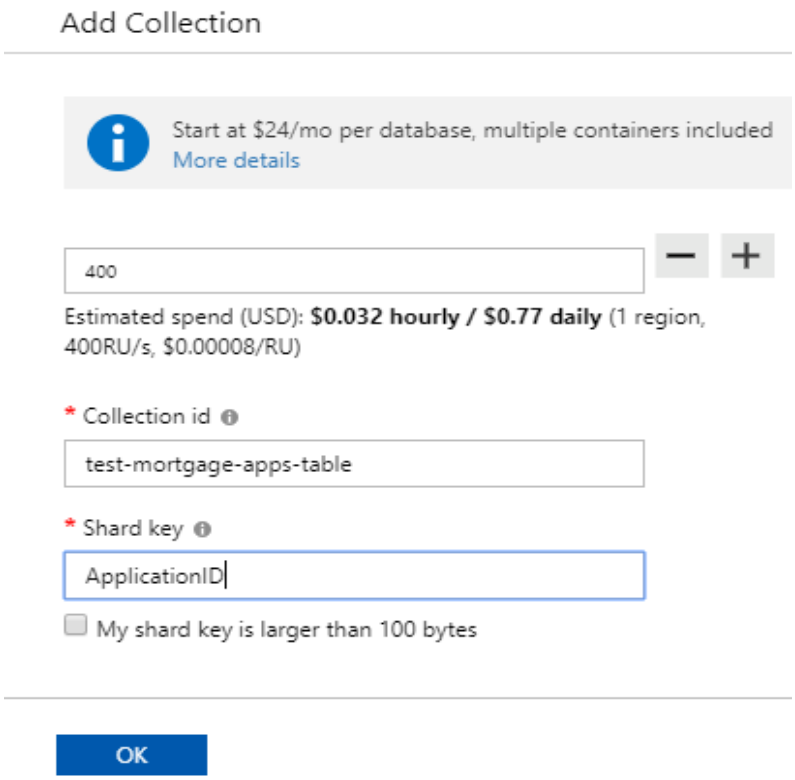


Figure AZ.COSMOS.7

Step 8: Add the local IP address to the Firewall to allow access from the home device via the Web Console. (Figure AZ.COSMOS.8)

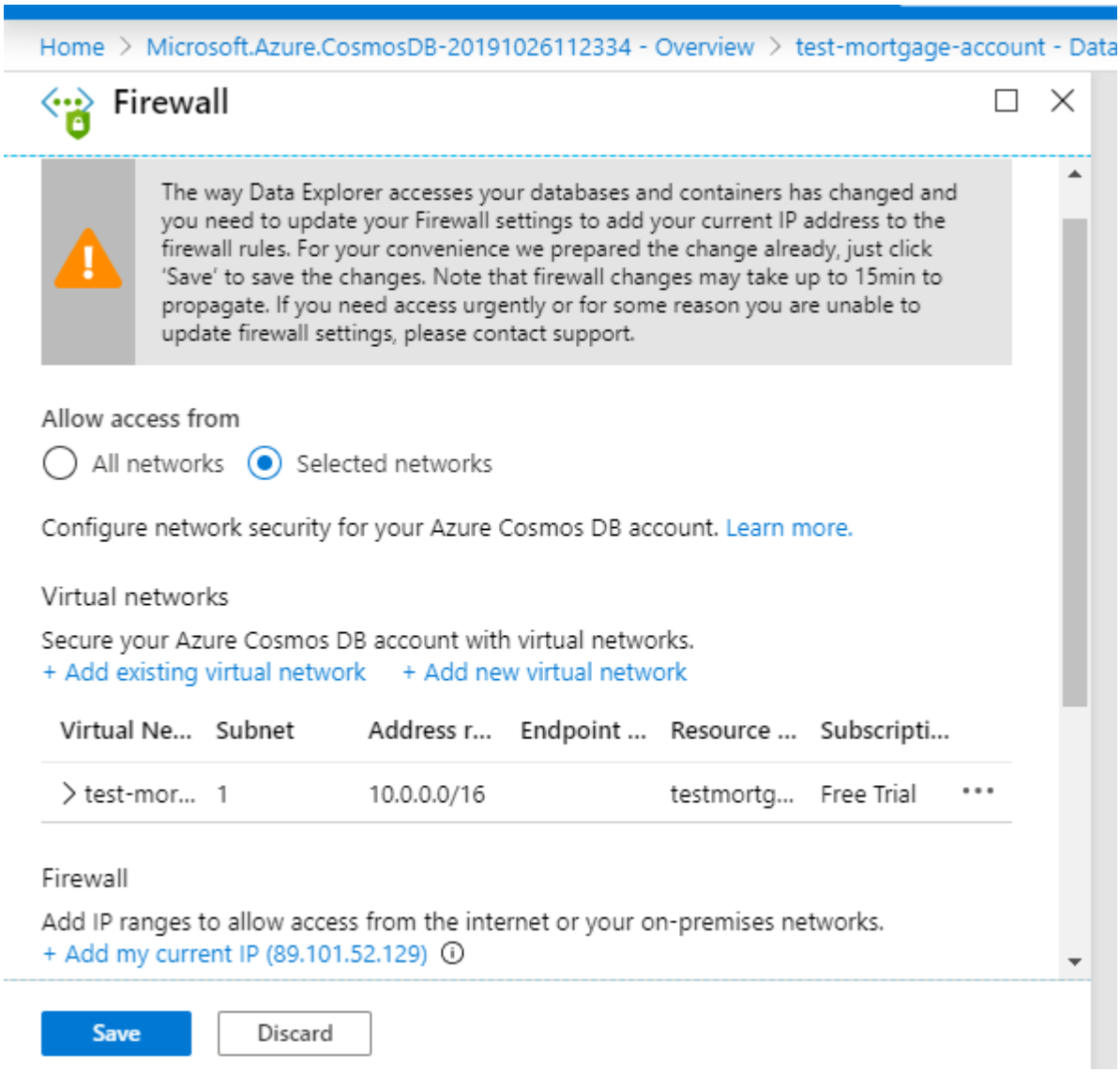


Figure AZ.COSMOS.8

Step 9: Add a document/Item to the Collection that we can use during testing of the Azure Function.

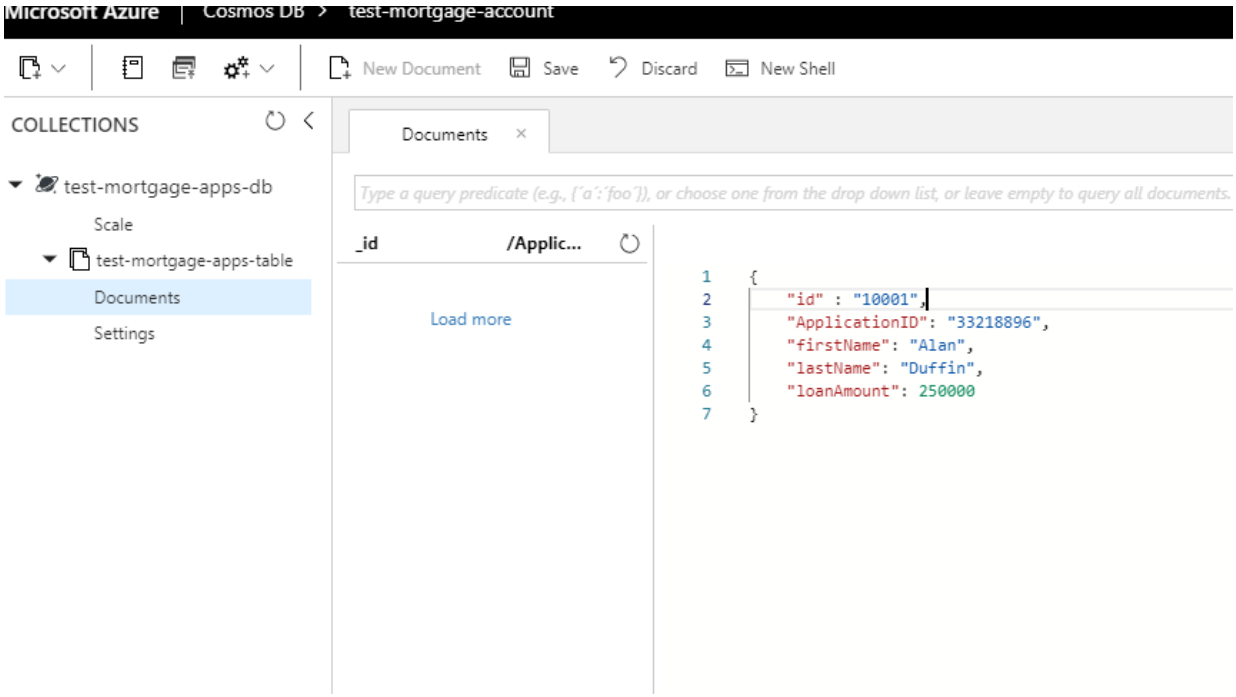


Figure AZ.COSMOS.9.

AWS IAM

To allow another AWS service or custom code to access the datastore, in this case DynamoDB, an IAM Role needs to be created that contains the appropriate permissions. For this investigation phase we can include just Scan, Query and GetItem permissions.

Step 1: Navigate to the IAM Console.

Step 2: Select Create Role. (Figure AWS.IAM.1)

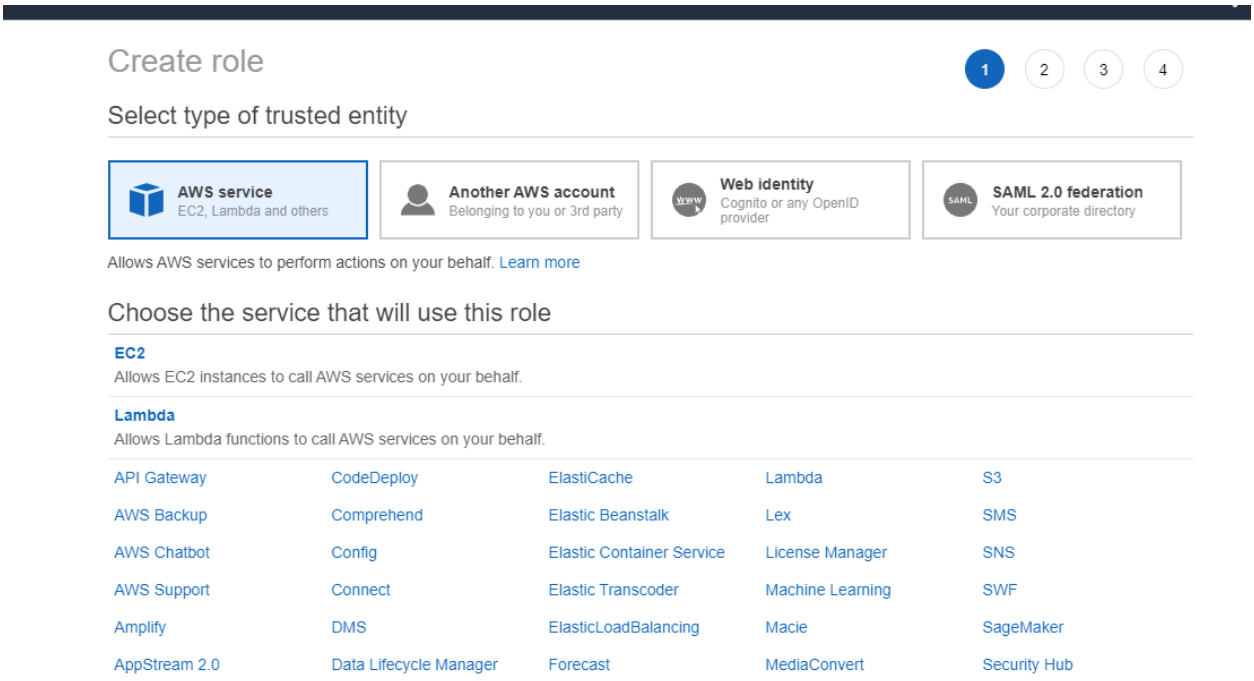


Figure AWS.IAM.1

Step 3: Select the required Access Permissions. (Figure AWS.IAM.2)

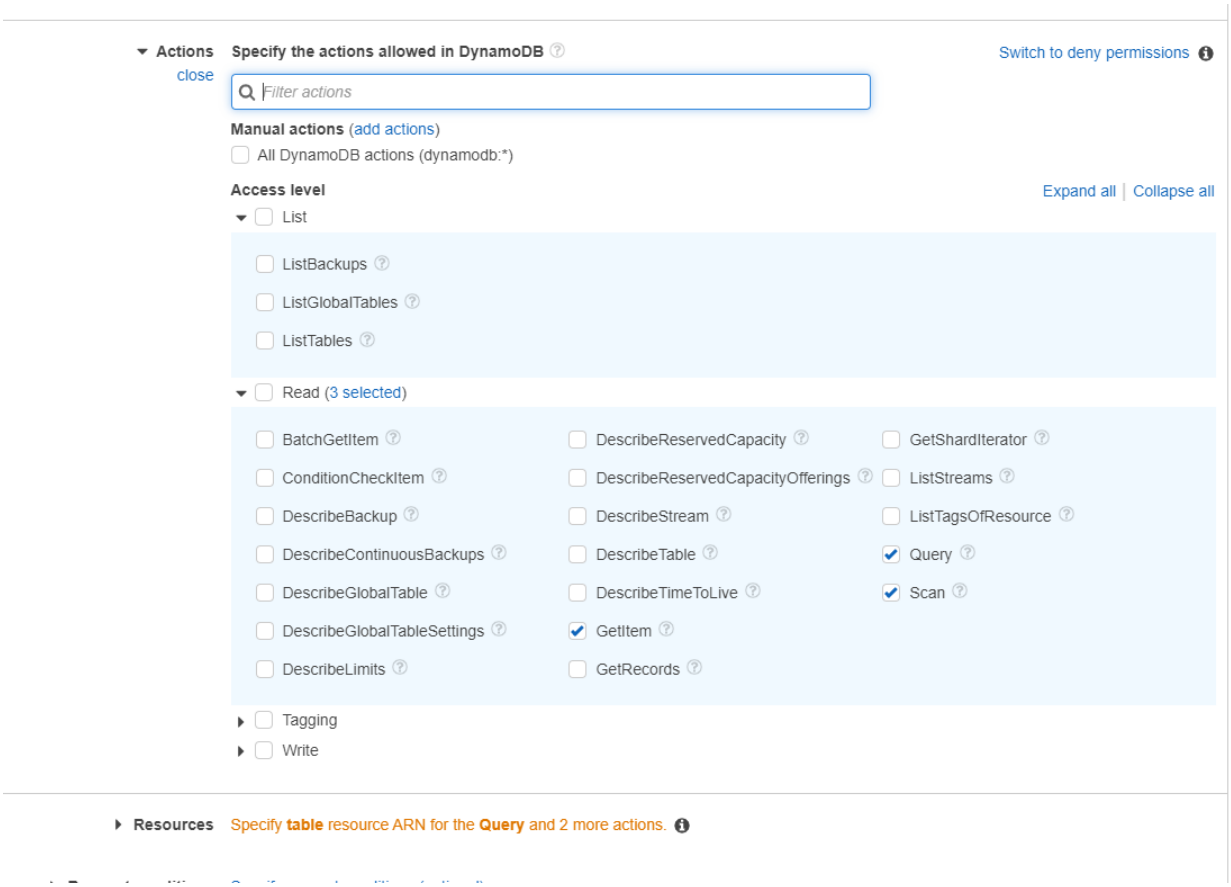


Figure AWS.IAM.2

Step 4: Name and Review the Policy. (Figures AWS.IAM.3, AWS.IAM.4)

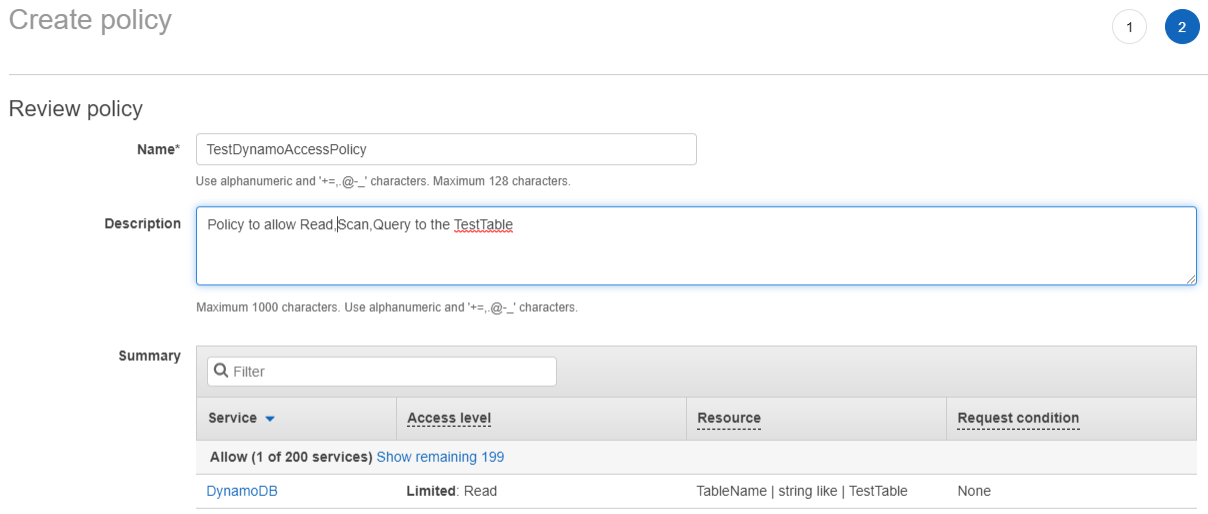


Figure AWS.IAM.3

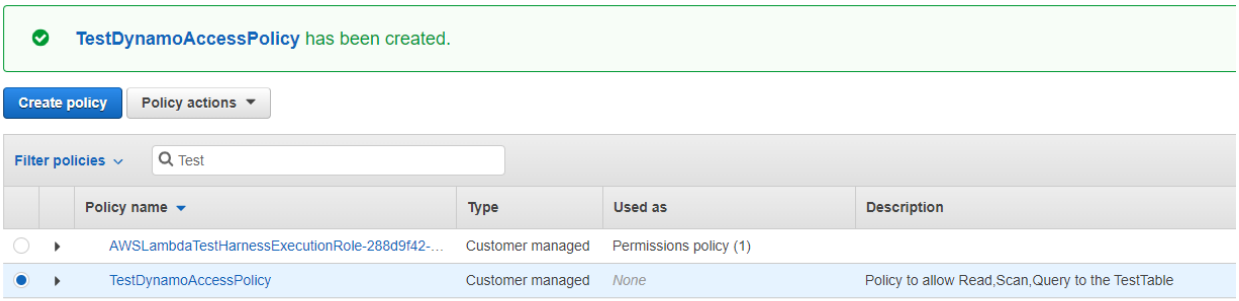


Figure AWS.IAM.4

Step 5: Add the Policy to the new Role. (Figure AWS.IAM.5)

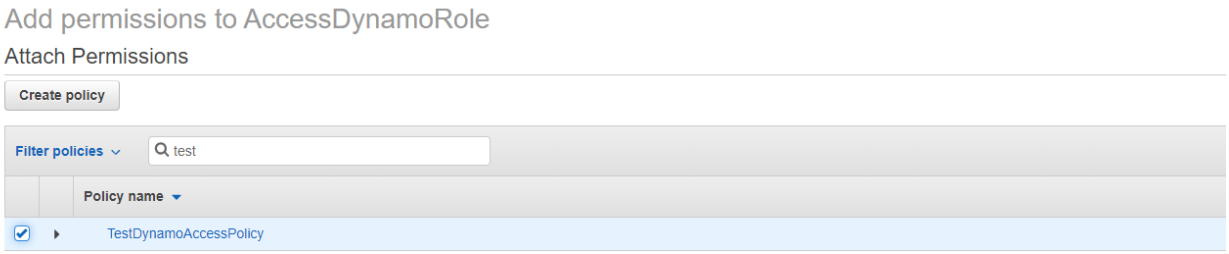


Figure AWS.IAM.5

AWS SERVERLESS (FUNCTIONS AS A SERVICE)

The AWS FaaS/Serverless offering is called AWS Lambda. In order to implement the project I will need to create a small suite of Lambda functions that will perform tasks such as reading, writing to a persistent data store, sending notifications of Mortgage Application state changes etc.

The following are the basic steps that are required in order to create a Lambda function.

Step 1: Navigate to the AWS Lambda Console. (Figure AWS.LAMBDA.1)

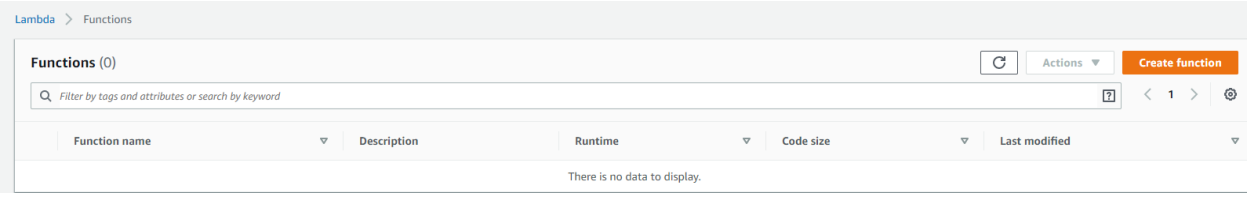


Figure AWS.LAMBDA.1

Step 2: Select Create Function. (Figure AWS.LAMDBA.2)

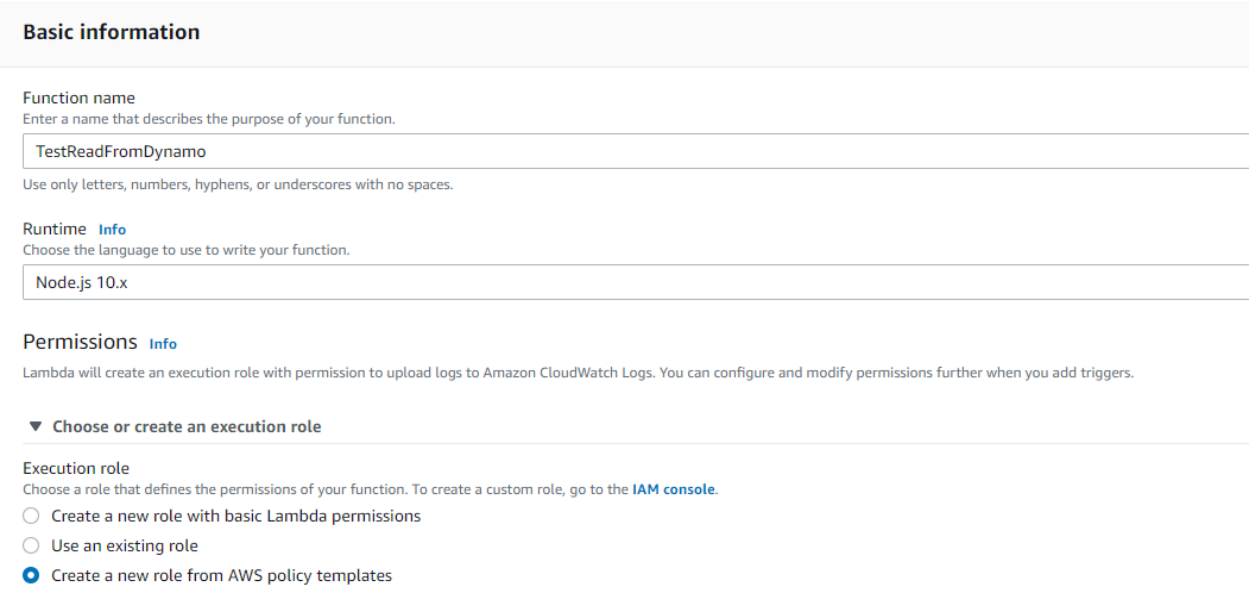


Figure AWS.LAMBDA.2

Step 3: Use an existing IAM Role and Choose the Role from previous section. (Figure AWS.LAMBDA.3)

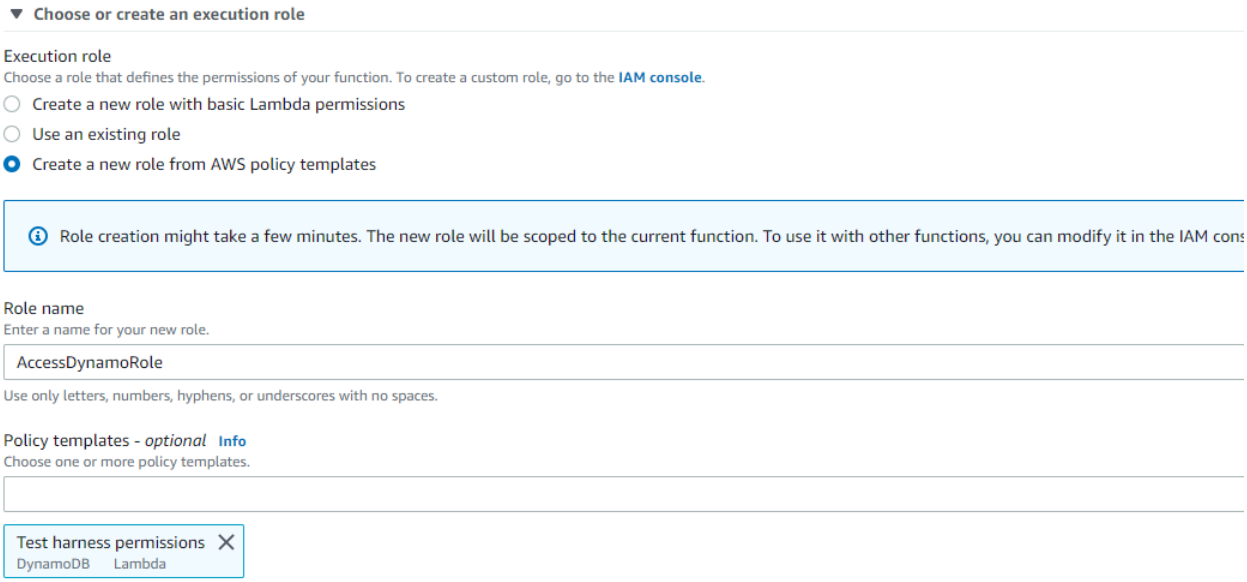


Figure AWS.LAMBDA.3

Step 4: Confirmation of new Function. (Figure AWS.LAMBDA.4)

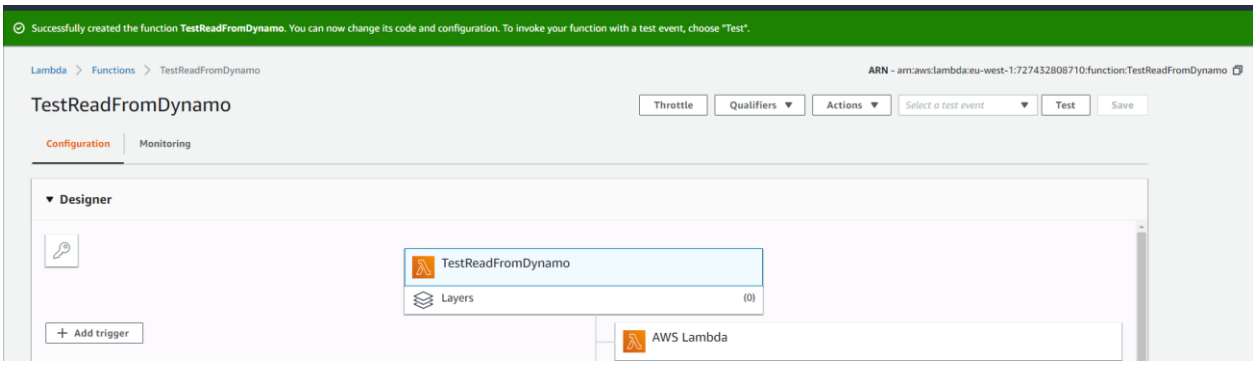


Figure AWS.LAMBDA.4

Step 4: Begin to input the source code. (Figure AWS.LAMBDA.5)

This code makes a simple call to read an Item from the NoSQL data store.

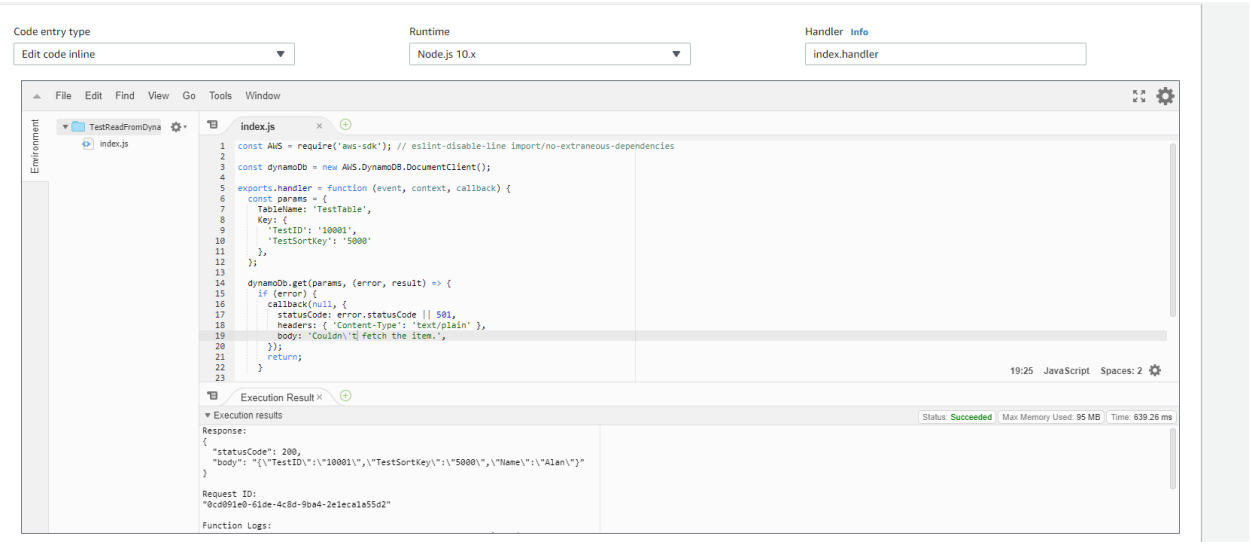


Figure AWS.LAMBDA.5

Step 5: Create a Test Event. (Figure AWS.LAMBDA.6)

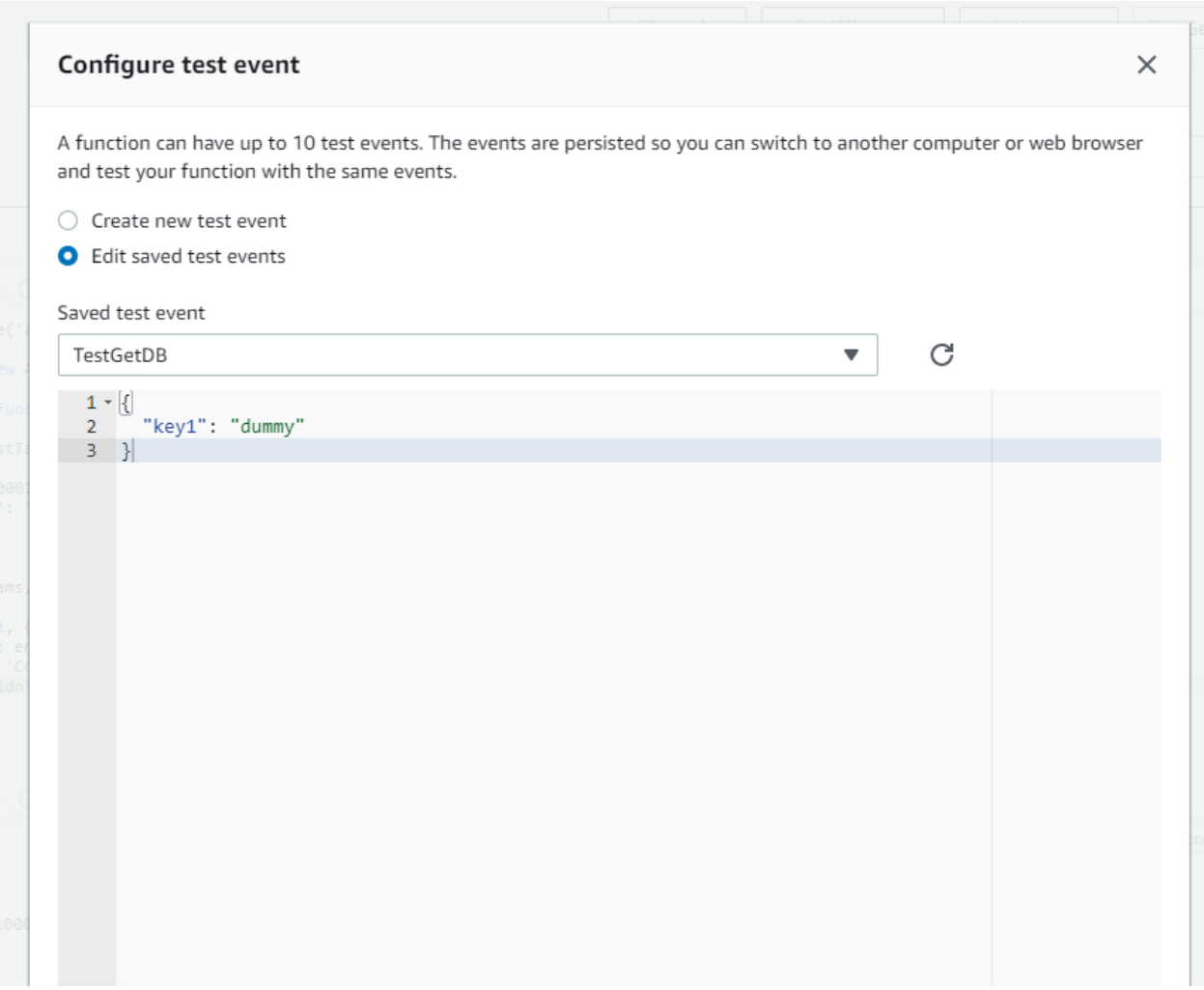


Figure AWS.LAMBDA.6

Step 6: Fire the Test Event. (Figure AWS.LAMBDA.7)

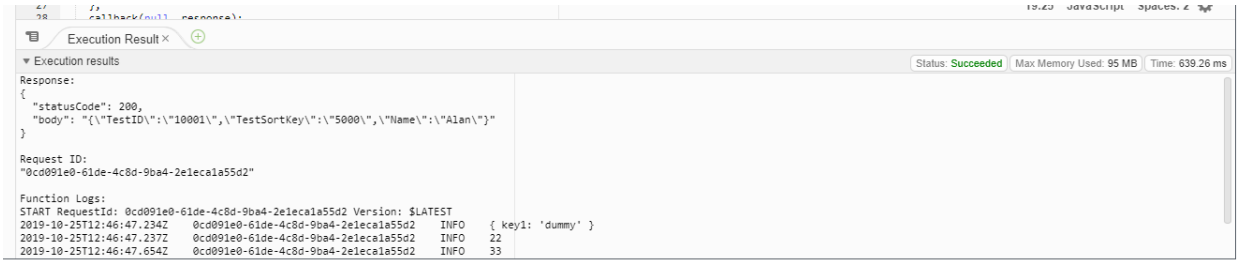


Figure AWS.LAMBDA.7

AWS NOSQL

It will be necessary to store individual Mortage Applications, therefore I require a Cloud provisioned Database. For expediency, I will investigate the most popular NoSQL services that are available from Azure and AWS. The AWS NoSQL offering is called DynamoDB. The following steps outline the simple tasks that are required to create a DynamoDB table.

Step 1: Navigate to the DynamoDB Console. (Figure AWS.DB.1)

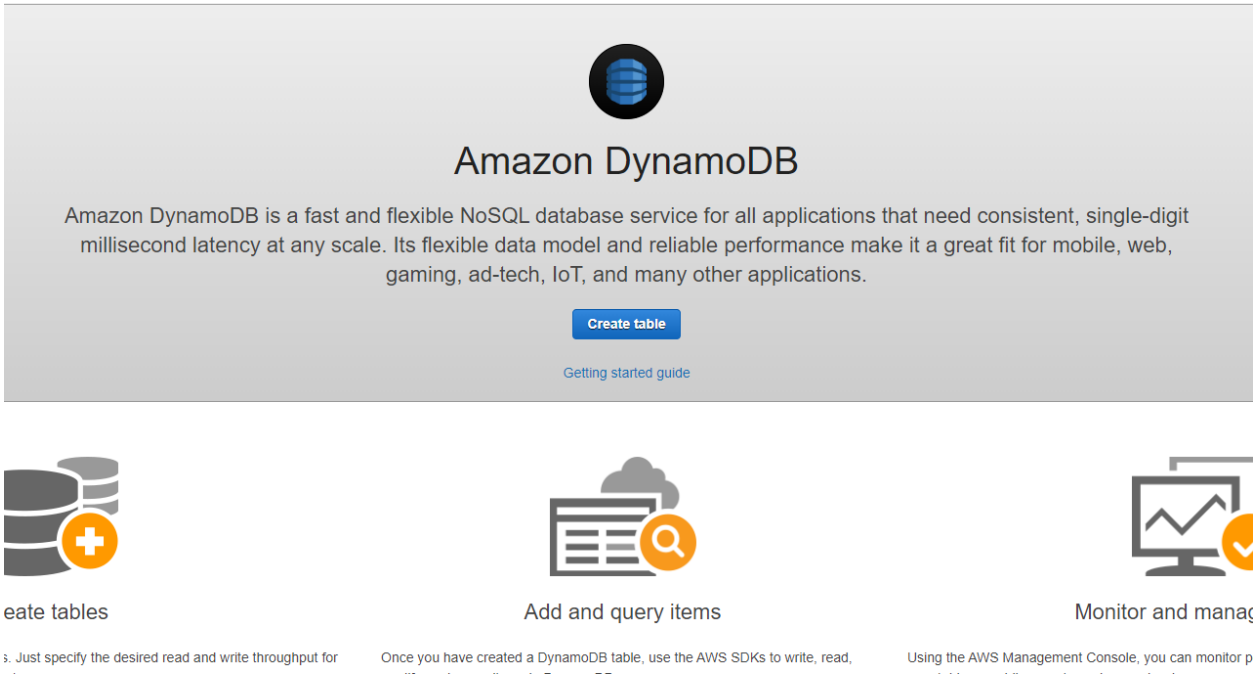


Figure AWS.DB.1

Step 2: Select Create Table. Figure (AWS.DB.2)

Step 3: Input a Partition Key Name and Type. The Partition Key must be an efficient Hash value that will result in effective distribution of table Items across logical partitions. Figure (AWS.DB.2)

Step 4: If required, Create a Composite Primary Key by specifying a Sort Key. Figure (AWS.DB.2)

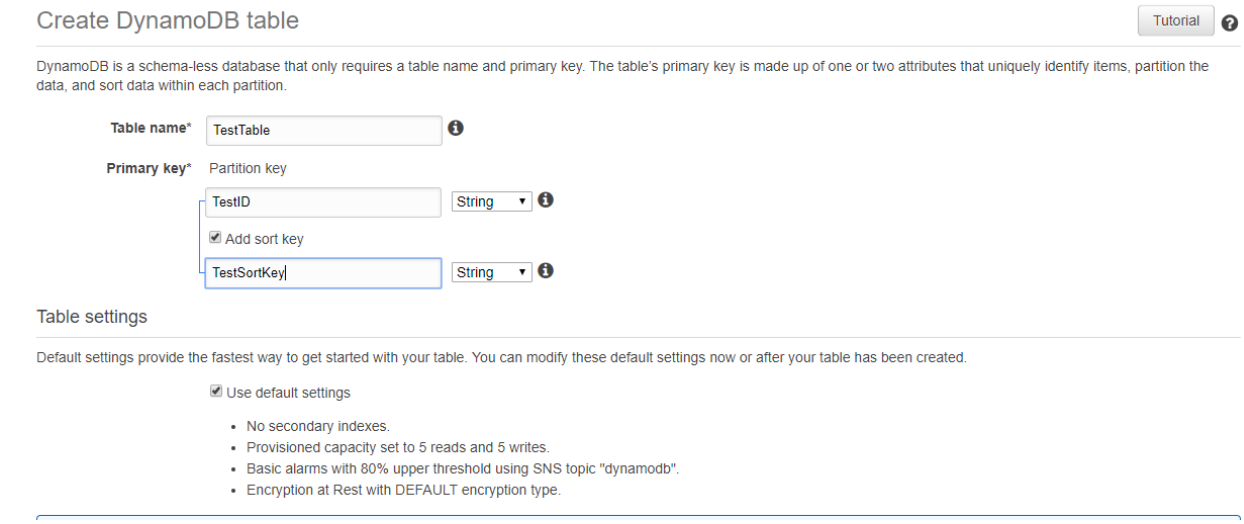


Figure (AWS.DB.2)

Step 5: Create the Table. Wait for the Table to be Provisioned. (Figure AWS.DB.3)

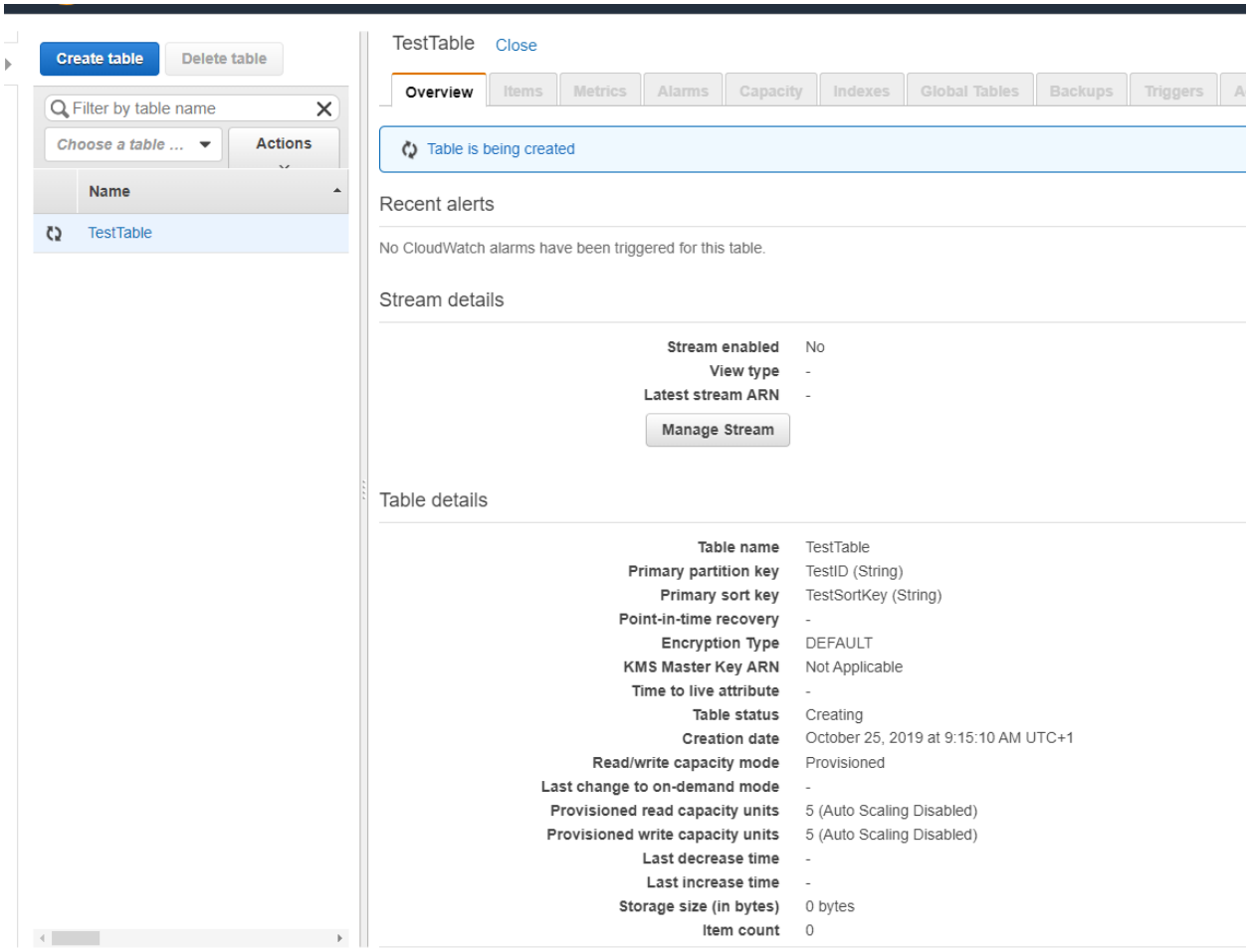


Figure AWS.DB.3

Step 6: Select Create Item. (Figure AWS.DB.4)

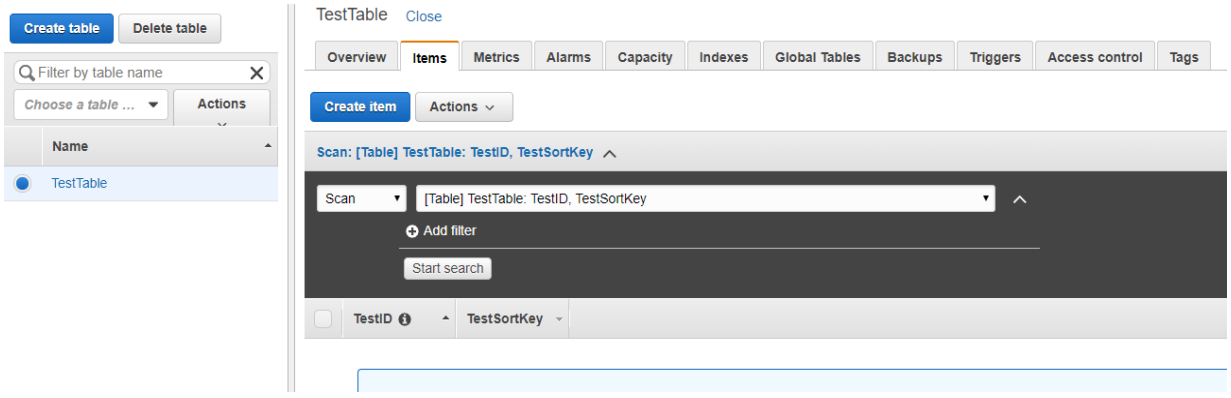


Figure AWS.DB.4

Step 7: Input the Item values. (Figure AWS.DB.5)

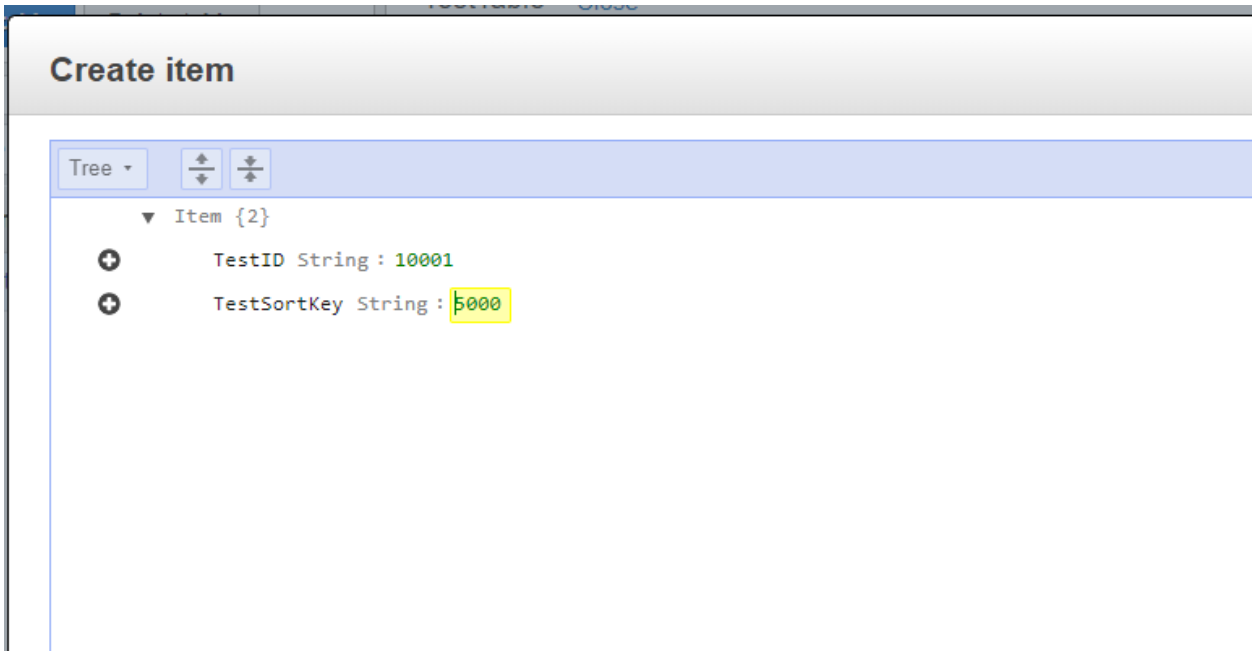


Figure AWS.DB.5

Step 8: View the Table Items. Figure AWS.DB.6

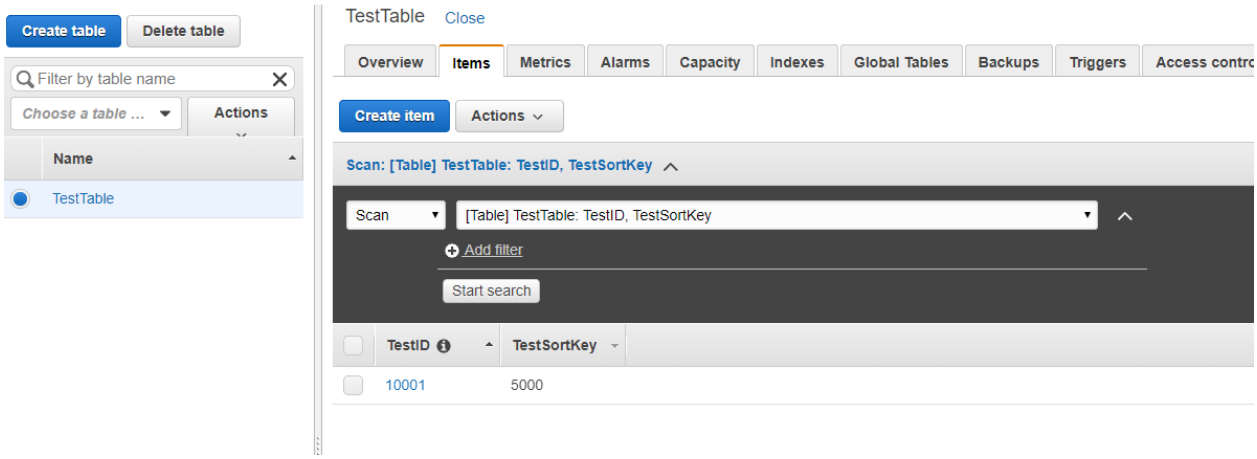


Figure AWS.DB.6

AWS WORKFLOW AND ORCHESTRATION

The AWS offering for Workflow and Orchestration is called the Step Functions service. In order to gain an understanding of the basic actions that are required to create a Workflow process using AWS Step Functions I will follow the Getting Started tutorials that are provided on the AWS website.

This small example will allow us to create a Step Function State Machine that contains a Task that will invoke the Lambda function that we created in the previous section.

Step 1: Create an IAM Role that will be used to allow the Step Function State Machine to invoke a Lambda function. (Figure AWS.STEP.1)

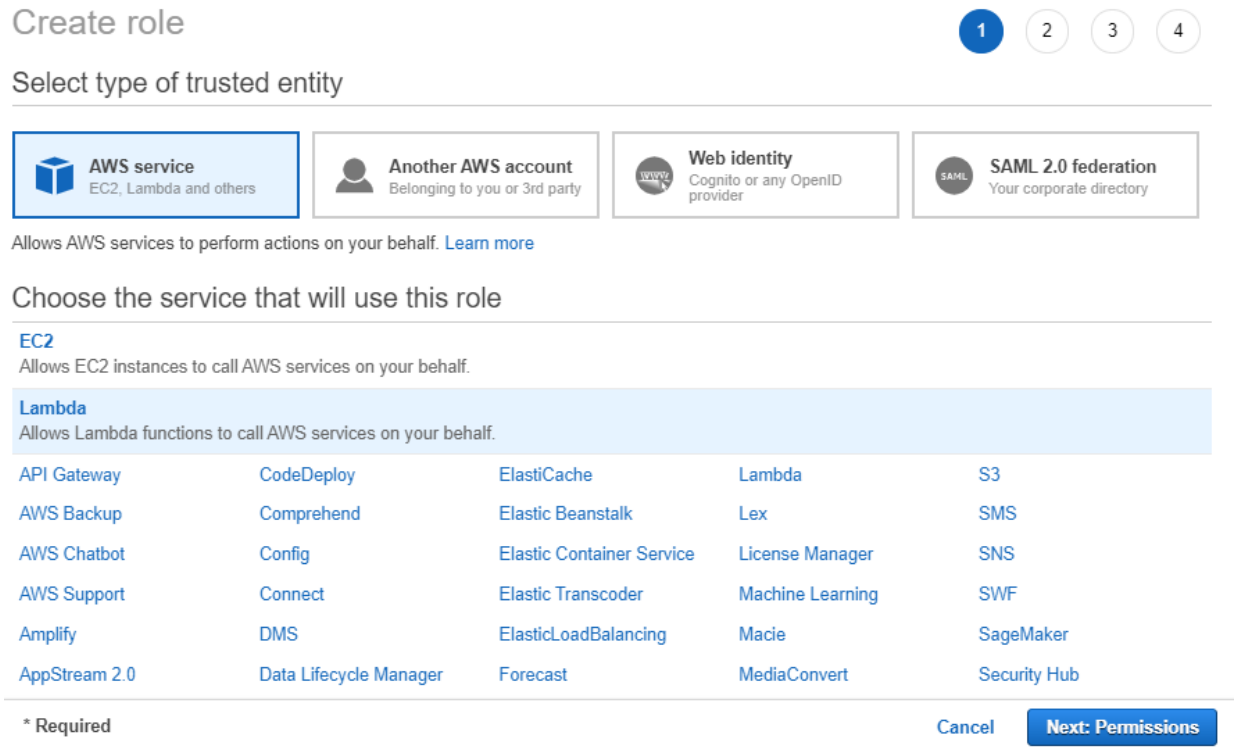


Figure: AWS.STEP.1

Step 2: Attach the AWSLambdaRole Policy to the new Role. (Figure AWS.STEP.2)

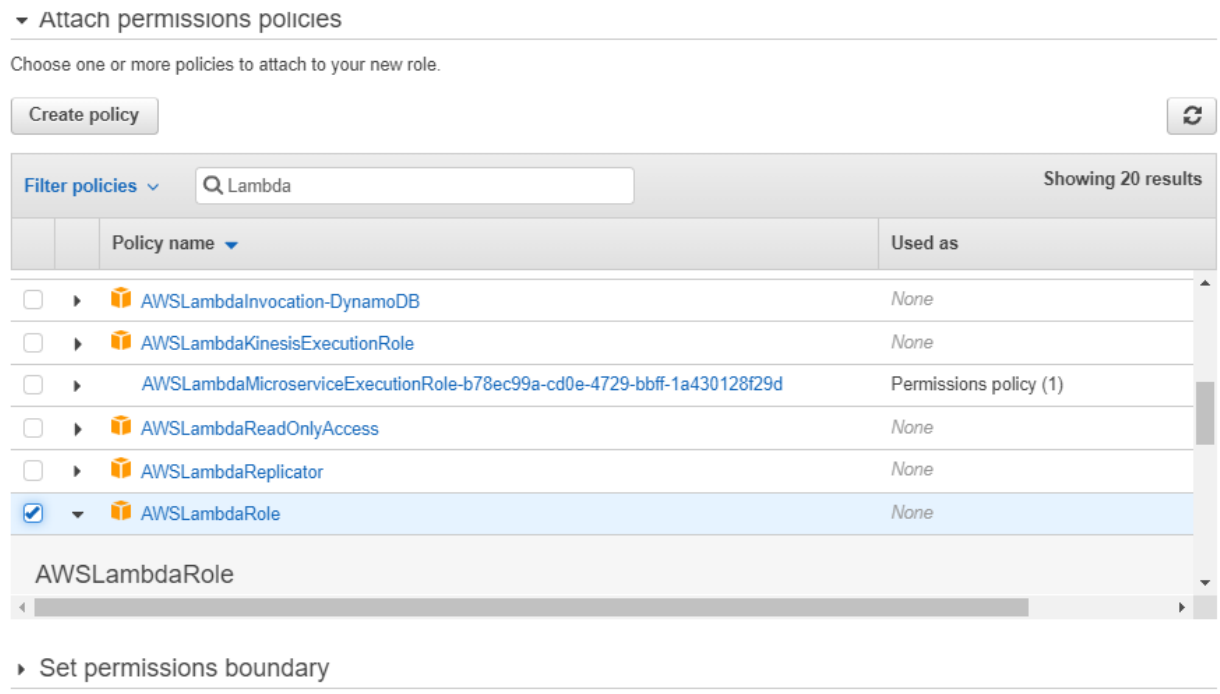


Figure AWS.STEP.2

Step 3: Name and Complete the Role creation. (Figure AWS.STEP.3)

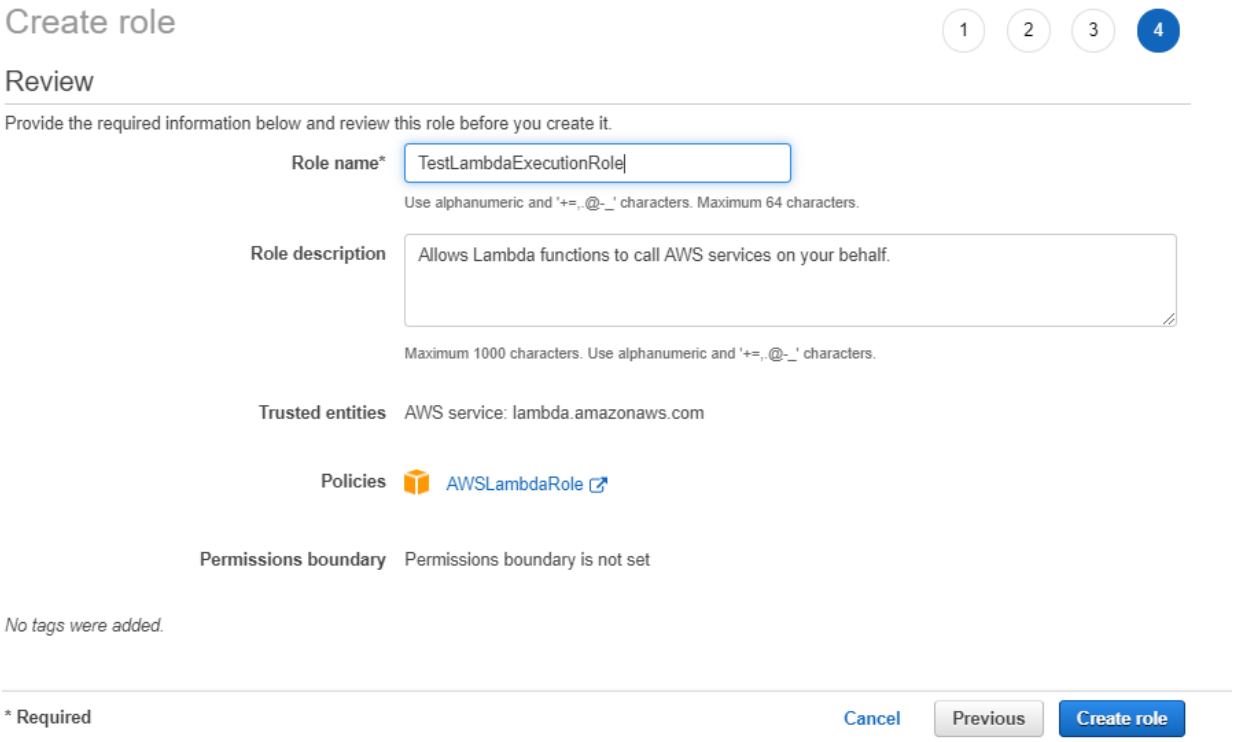


Figure AWS.STEP.3

Step 4: Create a new Step Function State Machine. (Figure AWS.STEP.4)

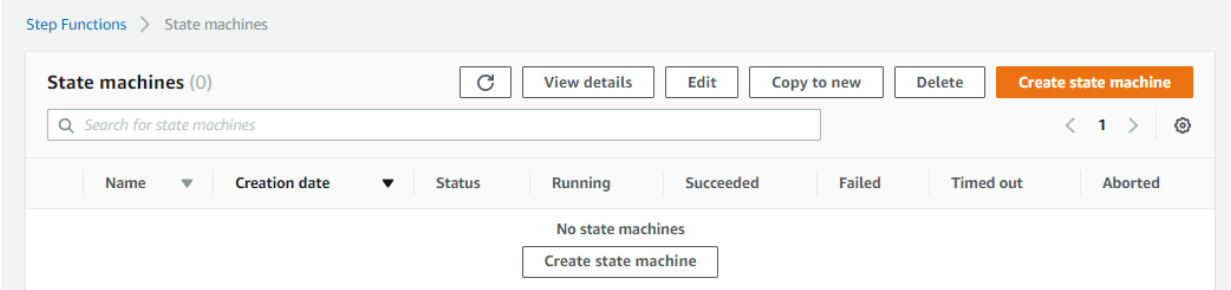


Figure AWS.STEP.4

Step 5: Select 'Author with code snippets'. (Figure AWS.STEP.5)

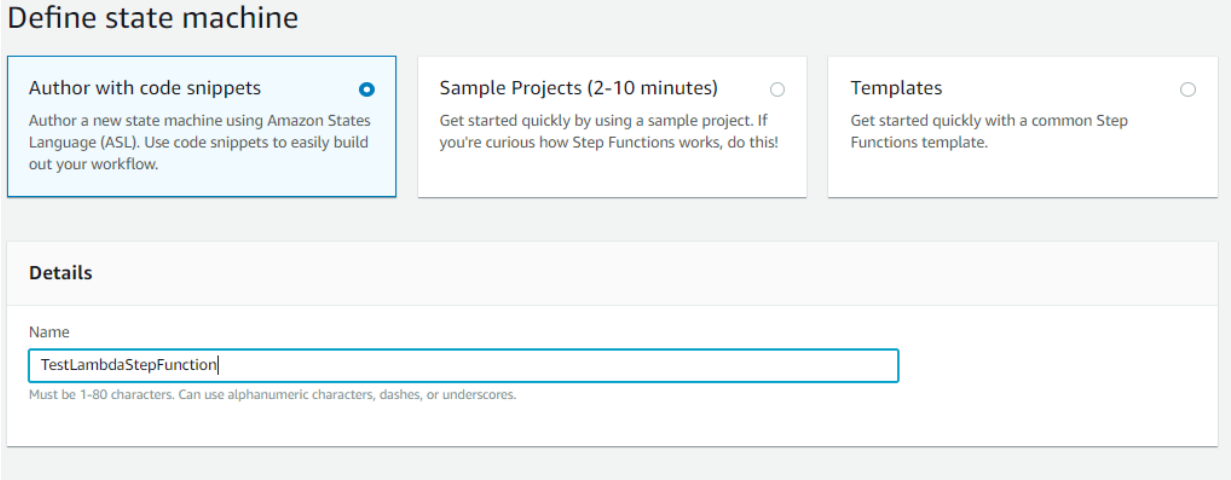


Figure AWS.STEP.5

Step 6: Define a Task state and specify the ARN for the existing Lambda. (Figure AWS.STEP.6)

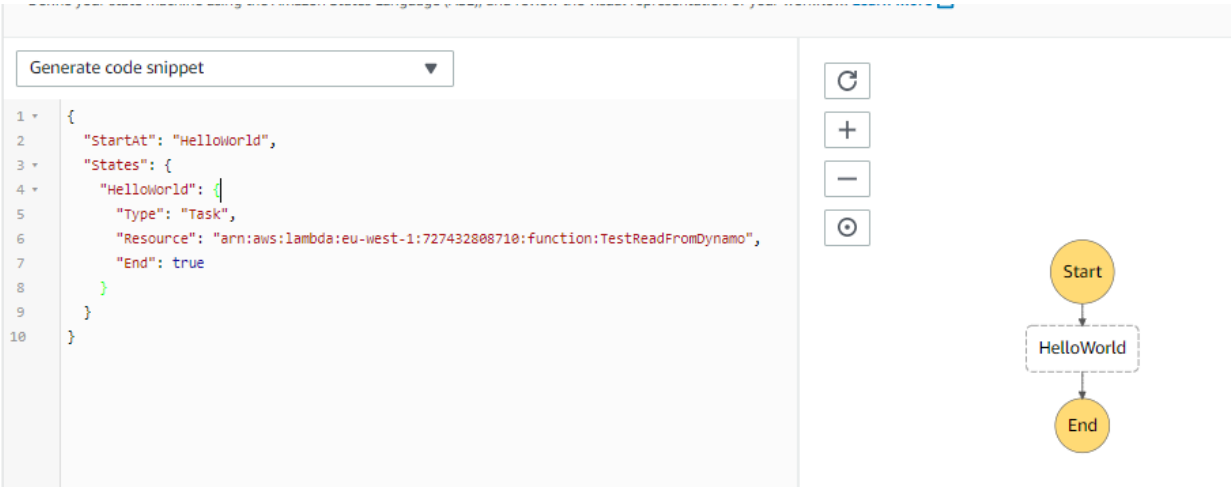


Figure AWS.STEP.6

Step 7: Choose the existing IAM Role. (Figure AWS.STEP.7)

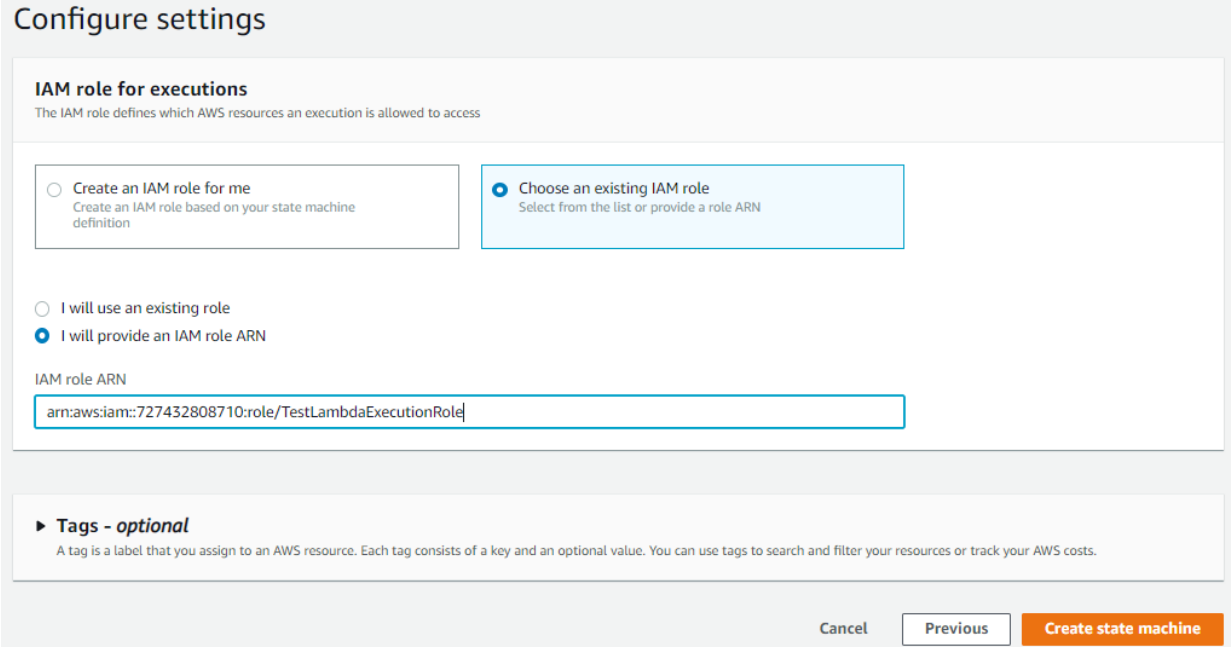


Figure AWS.STEP.7

Step 8: Create the State Machine and Save the ARN. (Figure AWS.STEP.8)

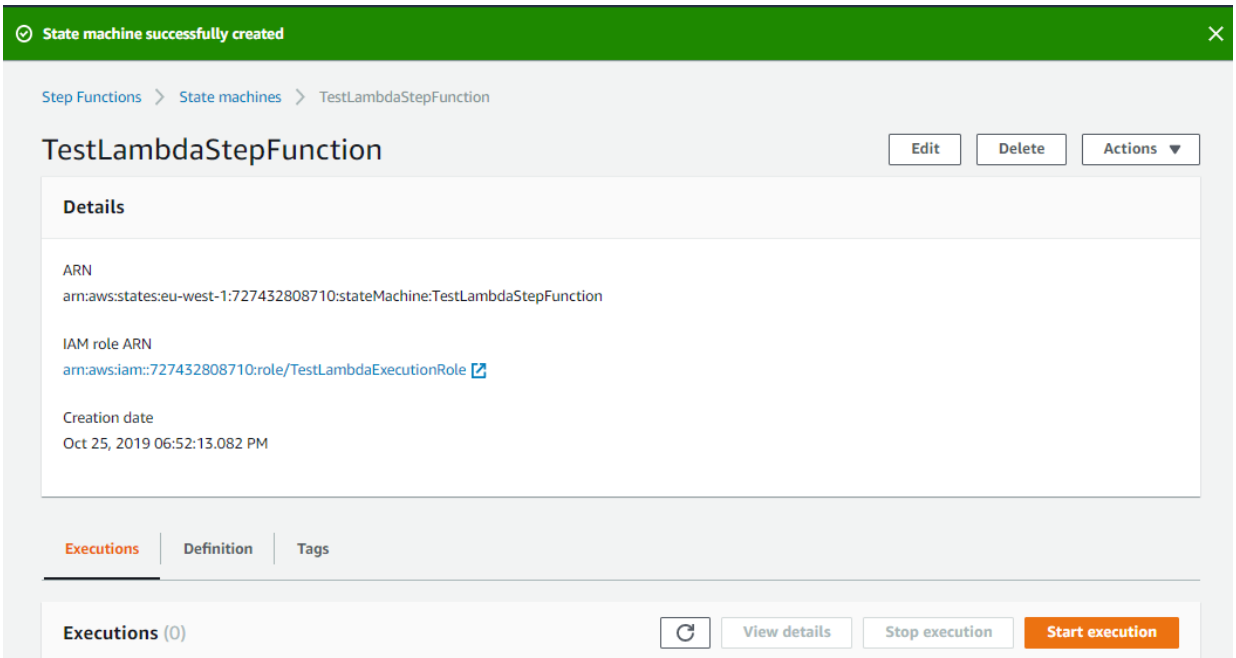


Figure AWS.STEP.8

Step 9: Test the new Step Function by creating a new Execution. (Figure AWS.STEP.9)

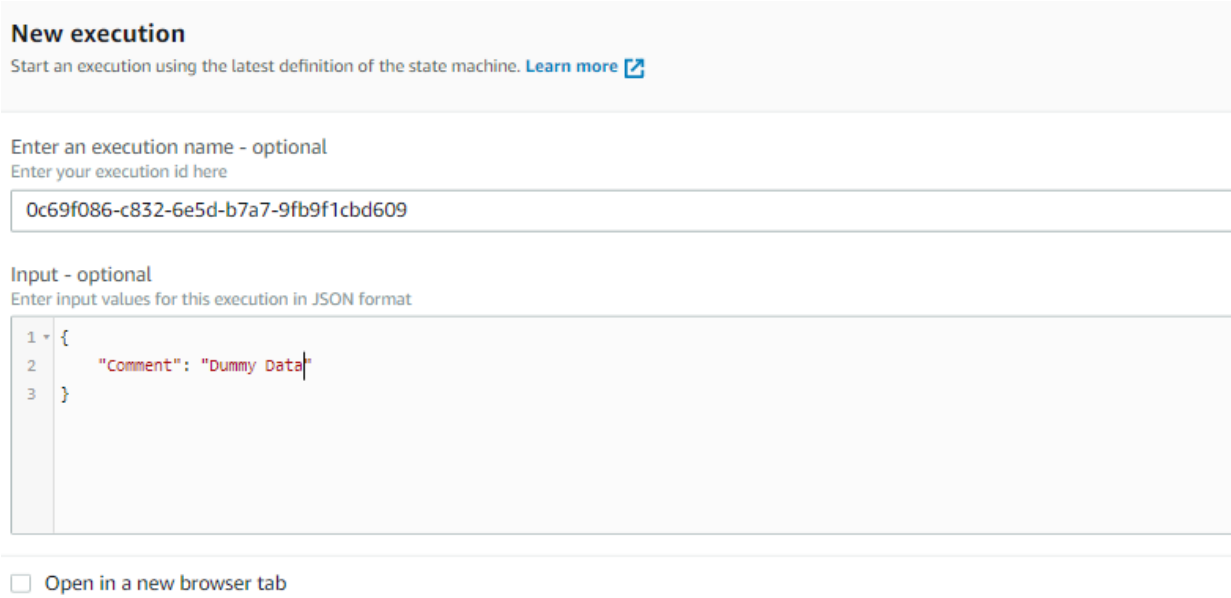


Figure AWS.STEP.9

Step 10: The Execution has failed. (Figure AWS.STEP.10, AWS.STEP.11)

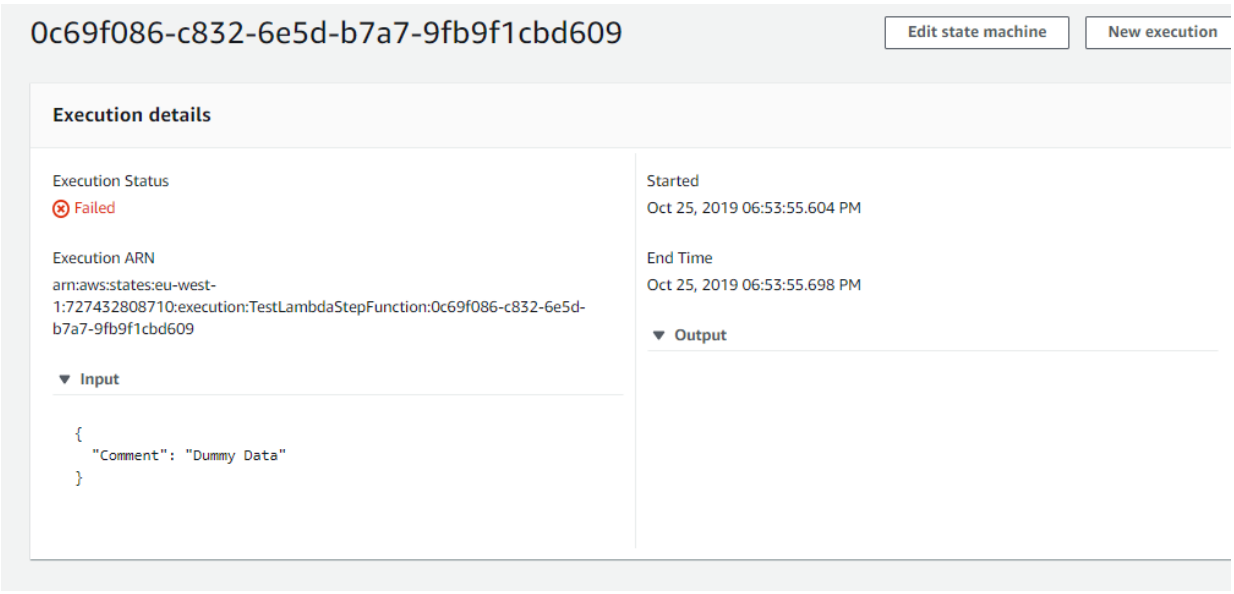


Figure AWS.STEP.10

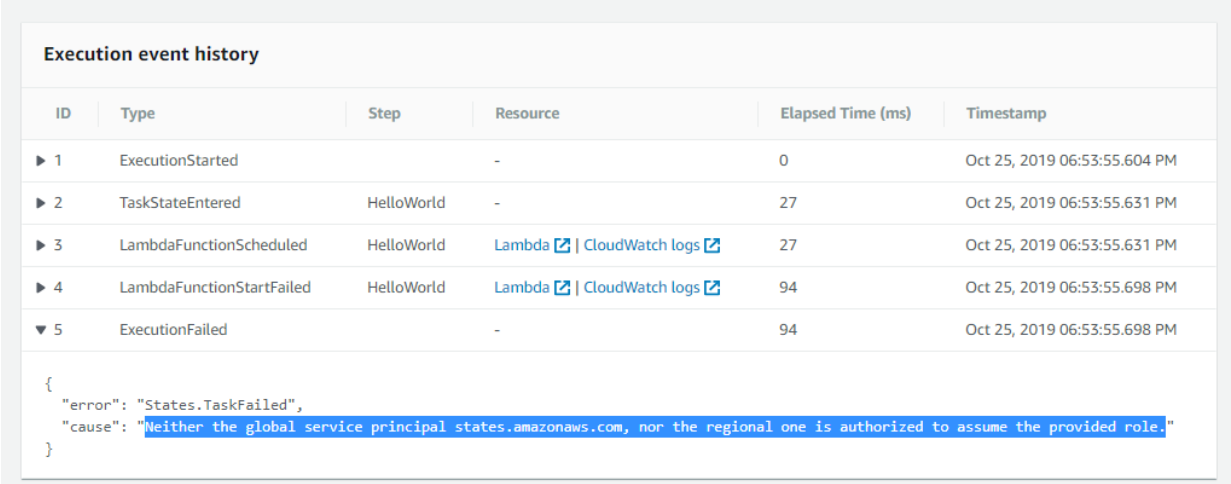


Figure AWS.STEP.11

After investigation, the error seems to be caused due to the Policy that was specified. It does not seem to be allowed to specify an unrestricted Resource section "*". The error was overcome by specifying the Resource to be Lambda functions scoped to the specific AWS account.

```
"Action": [
  "lambda:InvokeFunction"
],
"Resource": [
  "*"
]
```

```
"Action": [
  "lambda:InvokeFunction"
],
"Resource": [
  "arn:aws:lambda:eu-west-1:727432808710:function:*"
]
```

Step 11: Amend the Policy Permission as shown below. (Figure AWS.STEP.12)

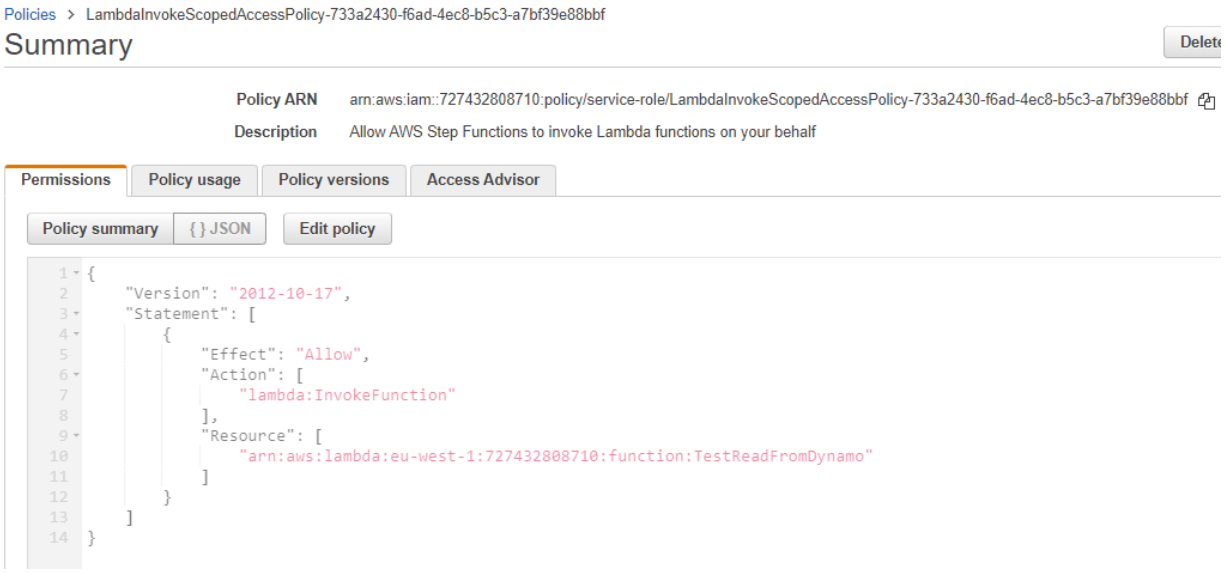


Figure AWS.STEP.12

Step 12: Retest the Step Function. (Figure AWS.STEP.13)

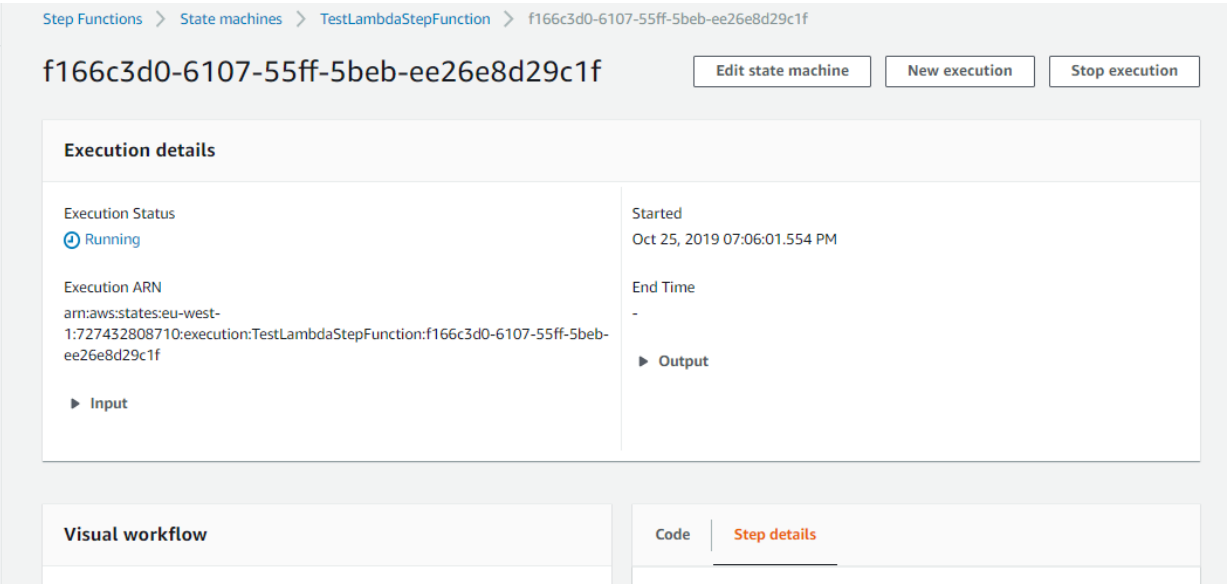
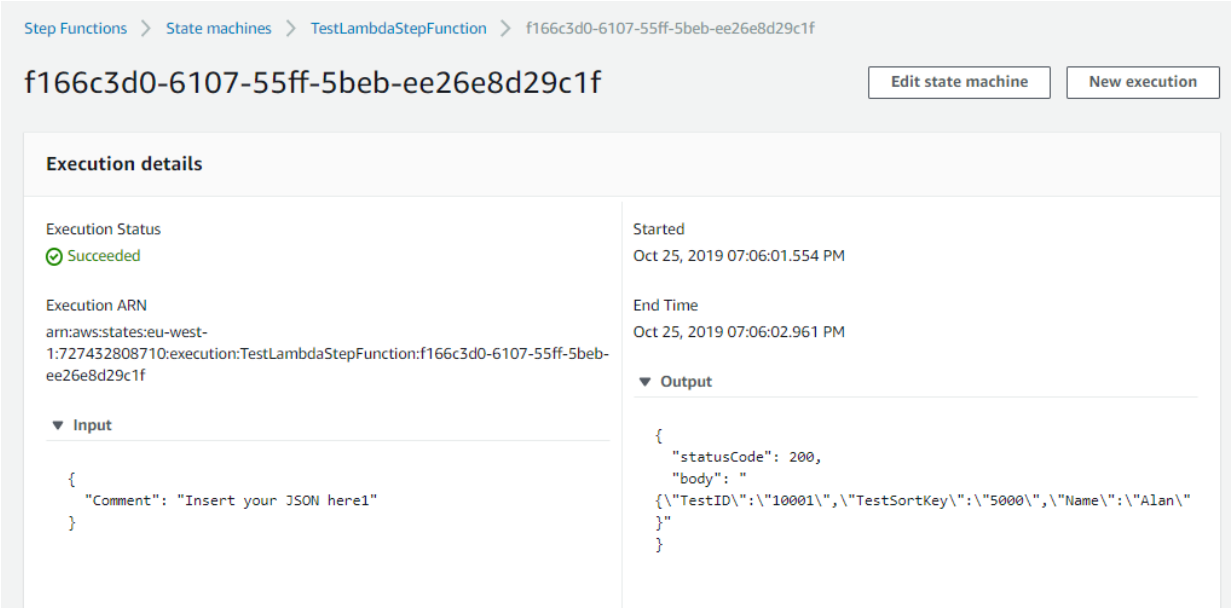


Figure AWS.STEP.13

Step 13: Note the Step Function Successful Output. (Figure AWS.STEP.14)



The output demonstrates that the Step Function invoked the Lambda function which responded with the DynamoDB table Item.

TECHNICAL DECISIONS

My investigations of the Azure and AWS Servlerless platforms as outlined in previous sections has allowed me to confirm that the AWS platform will be the cloud platform upon which I will develop this project.

In the two weeks of investigation I have formed the opinion that the developer experience and speed to market is superior on the AWS platform. In terms of ease of development and deployment, documentation, debugging, tooling I have found the AWS platform to be much more user friendly.

A percentage of this may be attributable to existing familiarity, but I have found the Azure documentation, support sites, technology stack to be below the standard I expected.

Therefore the technology resources the I will use are AWS services and open source development tools. The list is provided below.

1. NodeJS v13
2. Visual Studio Code
3. AWS DynamoDB
4. AWS Lambda
5. AWS Step Functions

PROJECT MILESTONES

The follow table lists the basic constituent components that are required to build the project implementation. This also give an indication of the milestone dates that should be met in order to deliver the project by the required date.

Due Date	Deliverables	
3rd November	Workflow Design CRUD Design Database Schema	
10th November	Database Build CRUD API Build	Unit Testing
17th November	Component Services Design	
24th November	Component Services Build	Unit + Integration Testing
1st December	Workflow Implmentation	Integration Testing
8th December	Workflow Implementation II	Integration Testing

REFERENCE LIST AND BIBLIOGRAPHY

Cloud Services FAQs

Google Cloud, Composer. https://cloud.google.com/composer/ (10 Oct, 2019)

Apache Airflow. https://airflow.apache.org/ (10 Oct, 2019)

Google Cloud, Functions. https://cloud.google.com/functions/ (10 Oct, 2019)

Google Cloud, Pub/Sub. https://cloud.google.com/pubsub/docs/ (10 Oct, 2019)

Google Cloud, Datastore. https://cloud.google.com/datastore/ (10 Oct, 2019)

Azure, Logic Apps. https://azure.microsoft.com/en-us/services/logic-apps/ (11 Oct, 2019)

Azure, Functions. https://azure.microsoft.com/en-us/services/functions/ (11 Oct, 2019)

Azure, Event Grid. https://azure.microsoft.com/en-us/services/event-grid/ (11 Oct, 2019)

Azure, Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/ (11 Oct, 2019)

Azure, ComosDB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction/ (11 Oct, 2019)

AWS, Step Functions. https://aws.amazon.com/step-functions/ (12 Oct, 2019)

AWS, Lambda. https://aws.amazon.com/lambda/ (12 Oct, 2019)

AWS, SNS. https://aws.amazon.com/sns/ (12 Oct, 2019)

AWS, SQS. https://aws.amazon.com/sqs/ (12 Oct, 2019)

AWS, Dynamo DB. https://aws.amazon.com/dynamodb/ (12 Oct, 2019)

Cloud Services Pricing

Google Cloud, Composer. https://cloud.google.com/composer/pricing/ (10 Oct, 2019)

Google Cloud, Functions. https://cloud.google.com/functions/pricing/ (10 Oct, 2019)

Google Cloud, Pub/Sub. https://cloud.google.com/pubsub/pricing/ (10 Oct, 2019)

Google Cloud, Datastore. https://cloud.google.com/datastore/pricing (10 Oct, 2019)

Google Cloud, Free Limits. https://cloud.google.com/free/ (10 Oct, 2019)

Azure, Logic Apps. https://azure.microsoft.com/en-us/pricing/details/logic-apps/ (11 Oct, 2019)

Azure, Functions. https://azure.microsoft.com/en-us/pricing/details/functions/ (11 Oct, 2019)

Azure, Event Grid. https://azure.microsoft.com/en-us/pricing/details/event-grid/ (11 Oct, 2019)

Azure, Queue Storage. https://azure.microsoft.com/en-in/pricing/details/storage/queues/ (11 Oct, 19)

Azure, ComosDB. https://azure.microsoft.com/en-us/pricing/details/cosmos-db/ (11 Oct, 2019)

Azure, Free Limits. https://azure.microsoft.com/en-us/free/#new-products

AWS, Step Functions. https://aws.amazon.com/step-functions/pricing/ (12 Oct, 2019)

AWS, Lambda. https://aws.amazon.com/lambda/pricing/ (12 Oct, 2019)

AWS, SNS. https://aws.amazon.com/sns/pricing/ (12 Oct, 2019)

AWS, SQS. https://aws.amazon.com/sqs/pricing/ (12 Oct, 2019)

AWS, Dynamo DB. https://aws.amazon.com/dynamodb/pricing/provisioned/ (12 Oct, 2019)

AWS, Free Limits. https://aws.amazon.com/free/

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Implementation Iteration 1

Student Name: Alan Duffin

Student ID: X00159409

November 2019

ITERATION 1 TASKS

Project Phase	Service / Area	Implementation Task
Design / Planning	AWS Step Functions	Design Inital Workflow / State Machine
	AWS Lambda	Design API/Services required for Workflow / Application tasks
	AWS DynamoDB	Design Database Schema
Build / Development	Development Environment / Toolkit	Create AWS Free Tier Account Install Developer IDE / SDK / CLI
Build / Development	AWS DynamoDB	Build CloudFormation template. Database Tables / Secondary Indexes
Build / Development	AWS Lambda	Build Serverless (SAM) template for customer / mortgage services.
Test		Test CRUD / Component Services + Database Interaction
Build / Development	AWS Step Functions	Build CloudFormation template for inital Workflow.
Test		Test State Transitions / Events
Build / Development	AWS SNS	Build SNS Topics and Subscriptions.
Test		Test Notification Events and Subscriptions

WORKFLOW DESIGN

The cental objective of this project is to build a solution for long – running applications that require human interaction / approval. The specific problem domain is the mortgage application / approval process. The implementation platform choosen as suitable for this type of application is the Serverless Application Model. This will involve development of a suite of AWS Lambda services that will act as tasks within the Workflow. Figure WORK.1

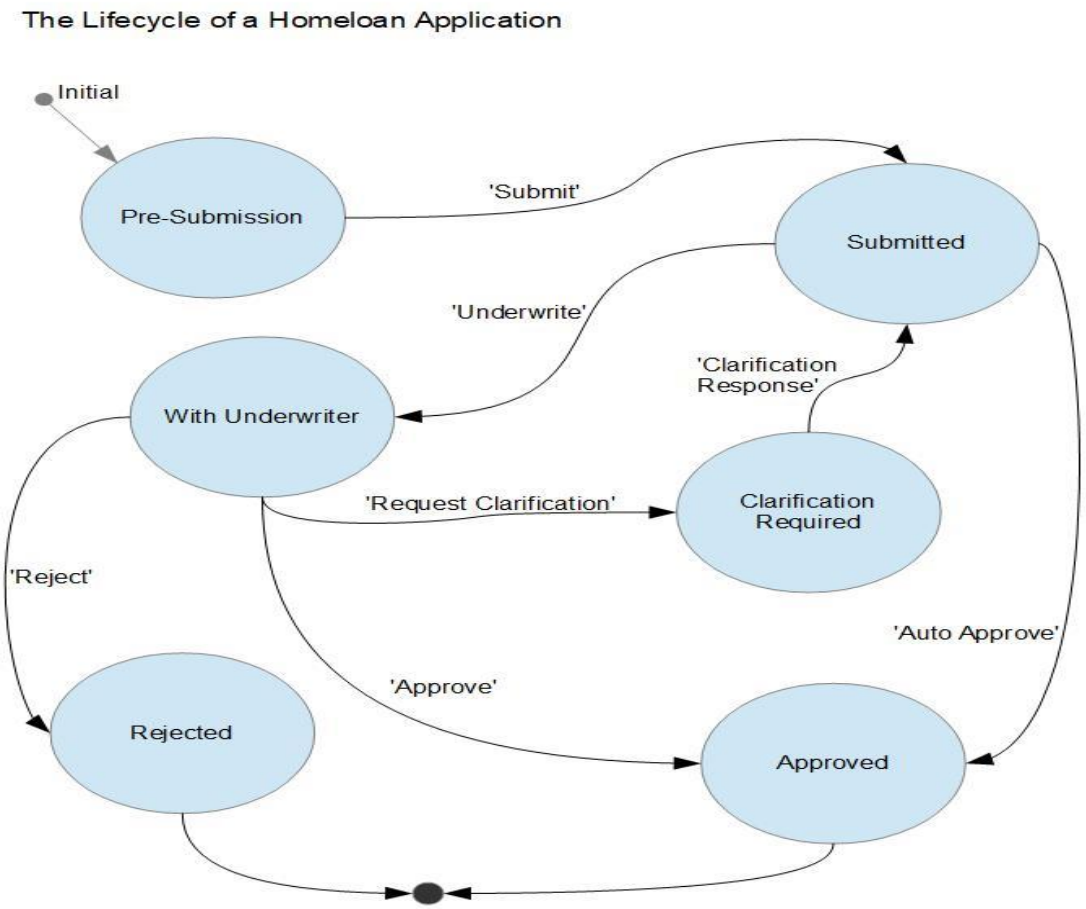


Figure WORK.1

SERVICES DESIGN

In order to build the workflow it is first necessary to put in place the substrate infrastructure / components that will be used as tasks within the workflow orchestration. The services are outlined below.

Service	URI	ACTION
Create Customer	/customer/	PUT
Get Customer	/customer/{id}	GET
Update Customer	/customer/{id}	POST
Create Mortgage	/mortgage/	PUT
Get Mortgage	/mortgage/{id}	GET
List Mortgages by State	/mortgage/status={statusText}	GET
Update Mortgage	/mortgage/{id}	POST
Mortgage Underwriter Notification	/mortgage/{morgageId}/underwriter/	POST
Mortgage Branch Notification	/mortgage/{morgageId}/branch/	POST
Mortgage Affordability Calculation	/mortgage/{morgageId}/afforability	POST
Mortgage Approval	/mortgage/{morgageId}/approve	POST
Submit Mortgage	/mortgage/{morgageId}/submit	POST

DATABASE SCHEMA DESIGN

The data model for this project consists of a Customer and Mortgage. There is a Many to One relationship between customers and mortgage application objects. In order to allow users to seach mortgages that require action, it is necessary to allow query the mortgage table by status. This requires a Global Secondary Index on the Mortgage table that is indexed by the *mortgageStatus* attribute. See Figure DB.1

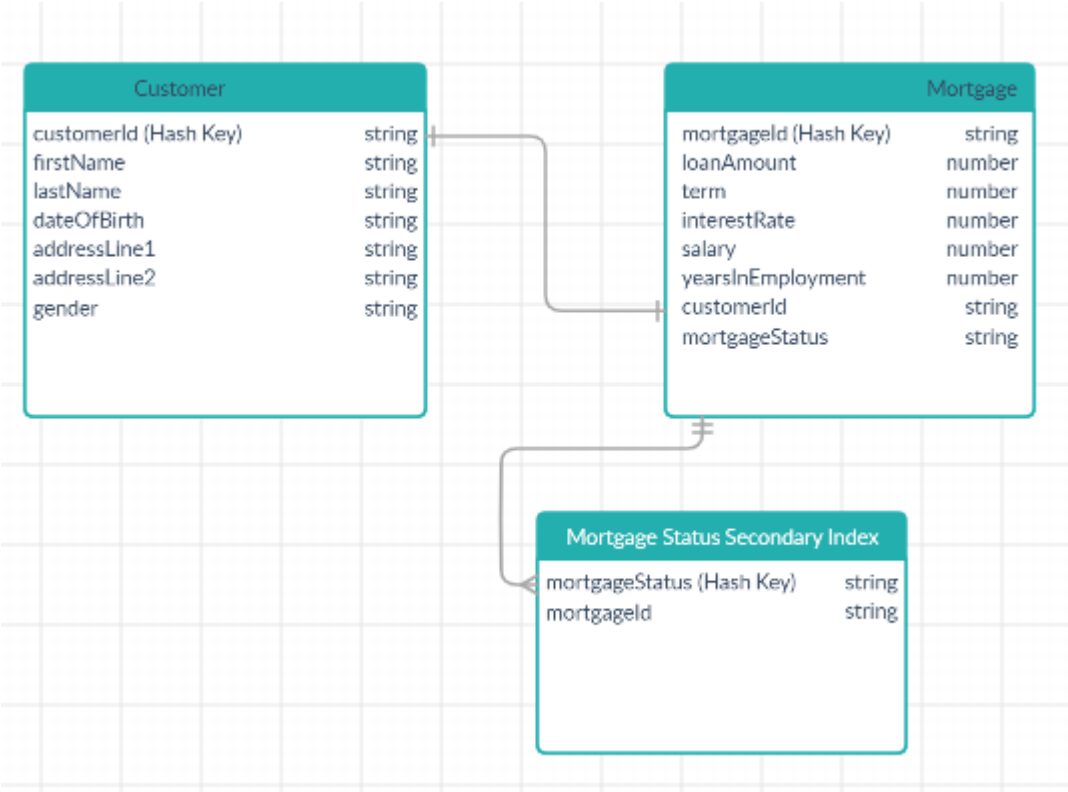


Figure DB.1

DEVELOPMENT ENVIRONMENT – AWS ACCOUNT

Navigate to the URL listed [1]. Follow the AWS instructions.

DEVELOPMENT ENVIRONMENT – AWS CLI

Navigate to the URL listed [2]. Follow the AWS instructions.

DEVELOPMENT ENVIRONMENT – AWS CLI SAM

Navigate to the URL listed [3]. Follow the AWS instructions.

DEVELOPMENT ENVIRONMENT – NODE JS / NPM

Navigate to the URL listed [4]. Follow the installation instructions.

DEVELOPMENT ENVIRONMENT – CONFIGURE AWC CLI

Create an AWS IAM Administrator Role. Figure IAM.1

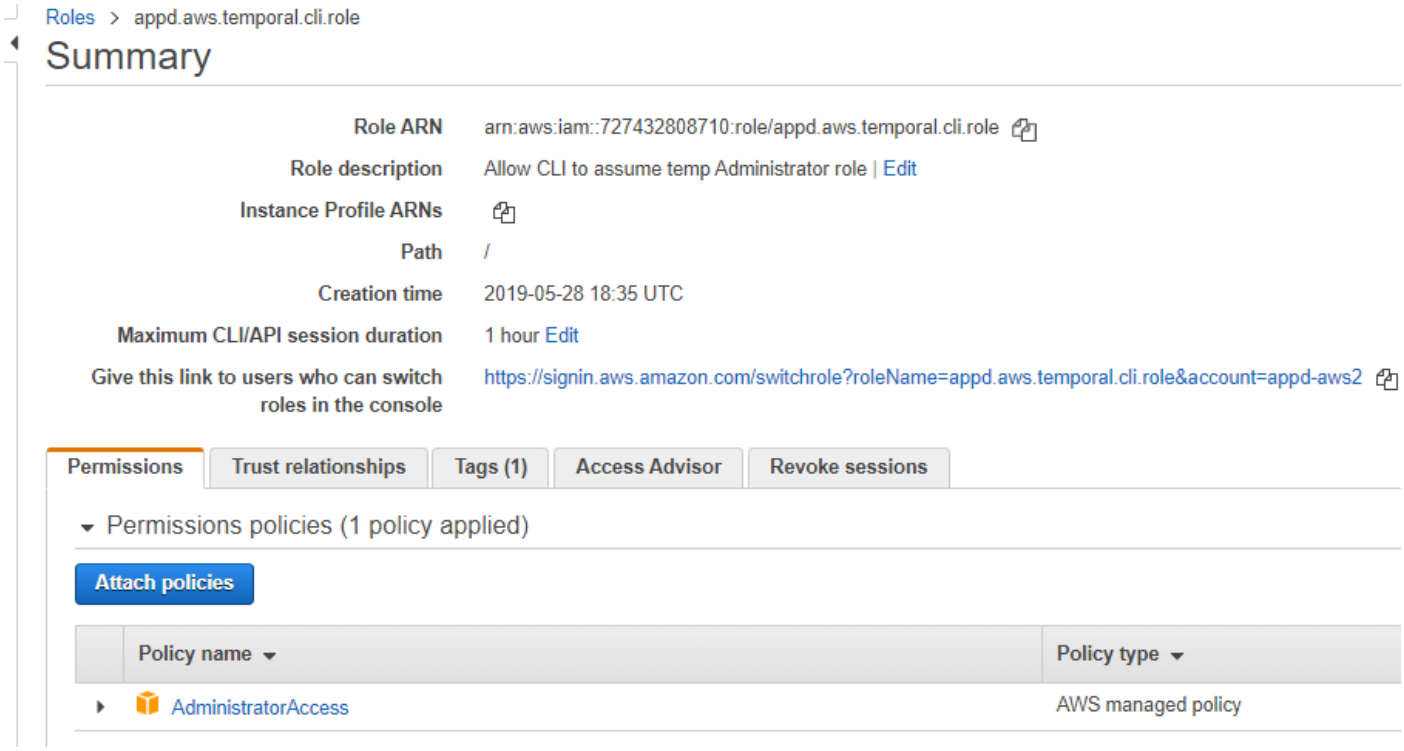


Figure IAM.1

Create an AWS IAM Group for Developers. Assign Policy to allow this Group to peform temporary assume-role of the Administrator Role. Figure IAM.2

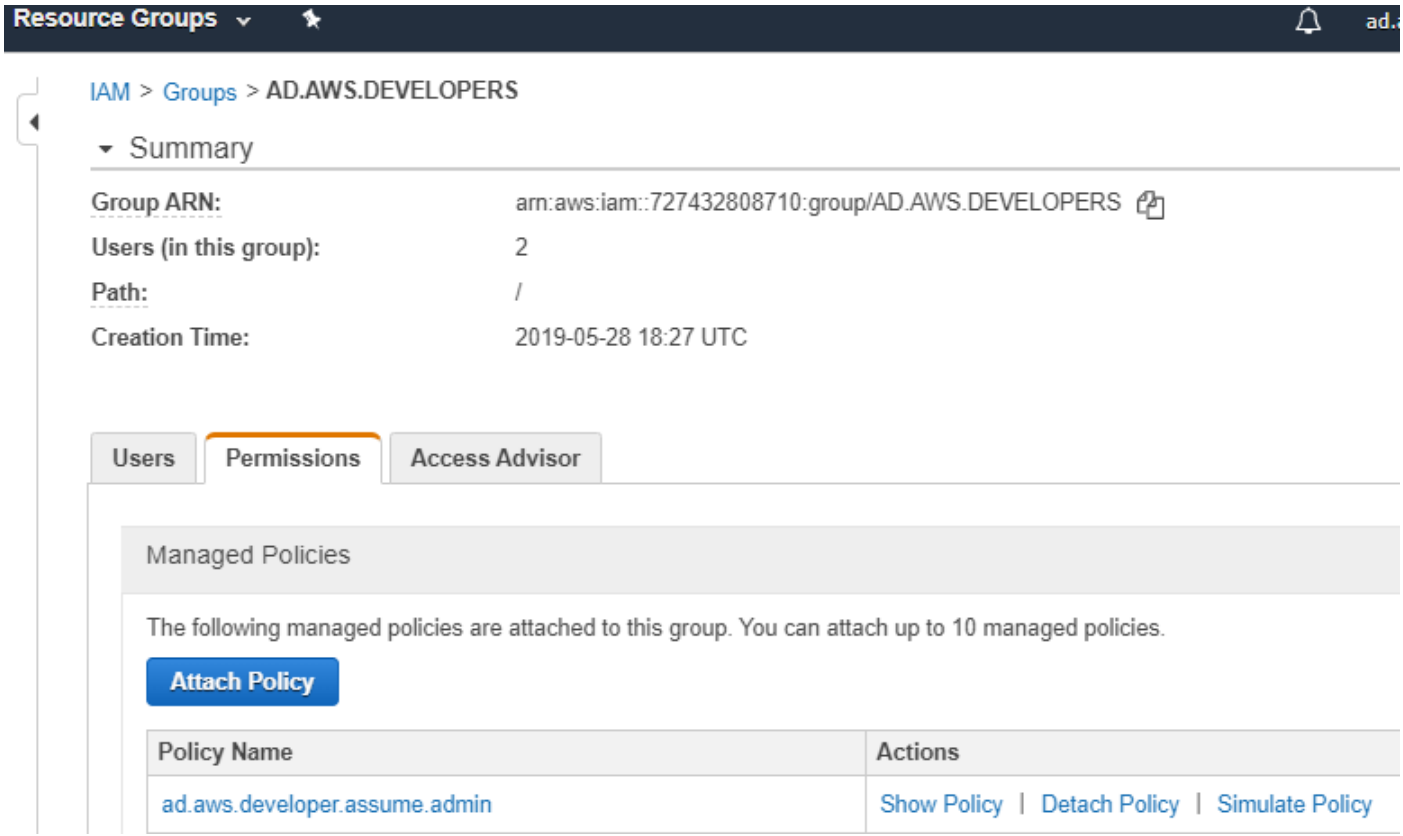


Figure IAM.2

Create an IAM Developer User and Assign to the Developer Group. Figure IAM3.

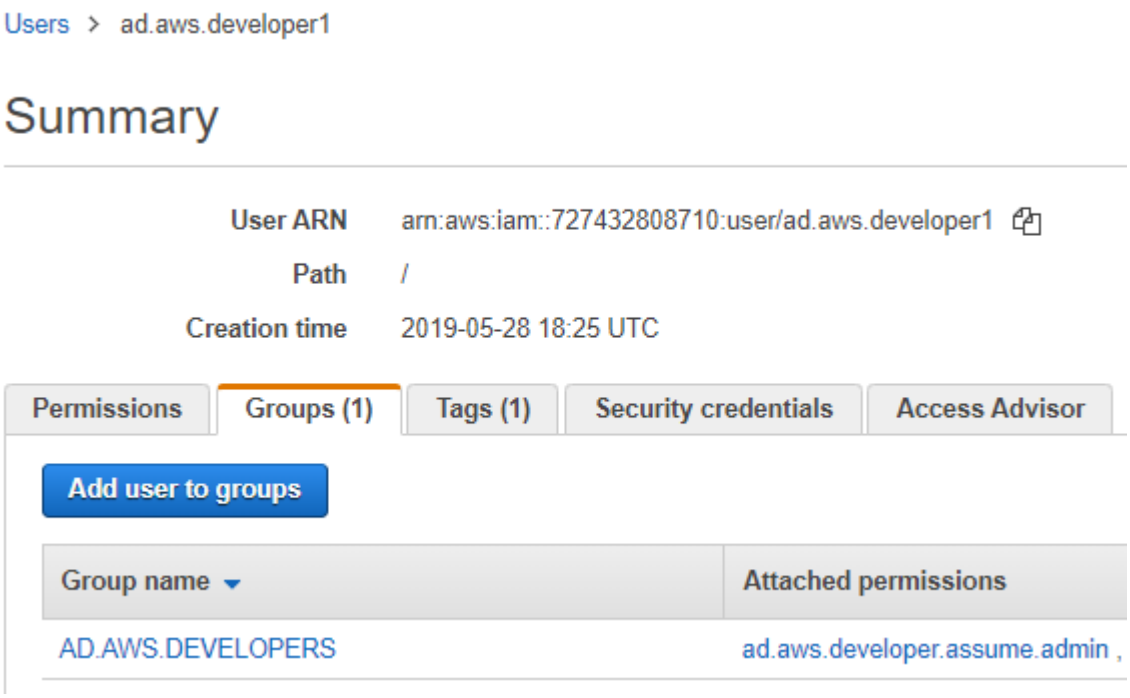


Figure IAM3.

DEVELOPMENT ENVIRONMENT – CONFIGURE AWS CLI

See instructions [5]. Using the Access Key ID and Secret Access Key downloaded when creating the Developer user we now must configure the CLI.

DEVELOPMENT ENVIRONMENT – INSTALL VISUAL STUDIO CODE

Navigate to the URL listed [6]. Follow the installation instructions.

DEVELOPMENT ENVIRONMENT – CONFIGURE VISUAL STUDIO CODE

Navigate to the URL listed [7]. Follow the installation instructions. Figure VS.1

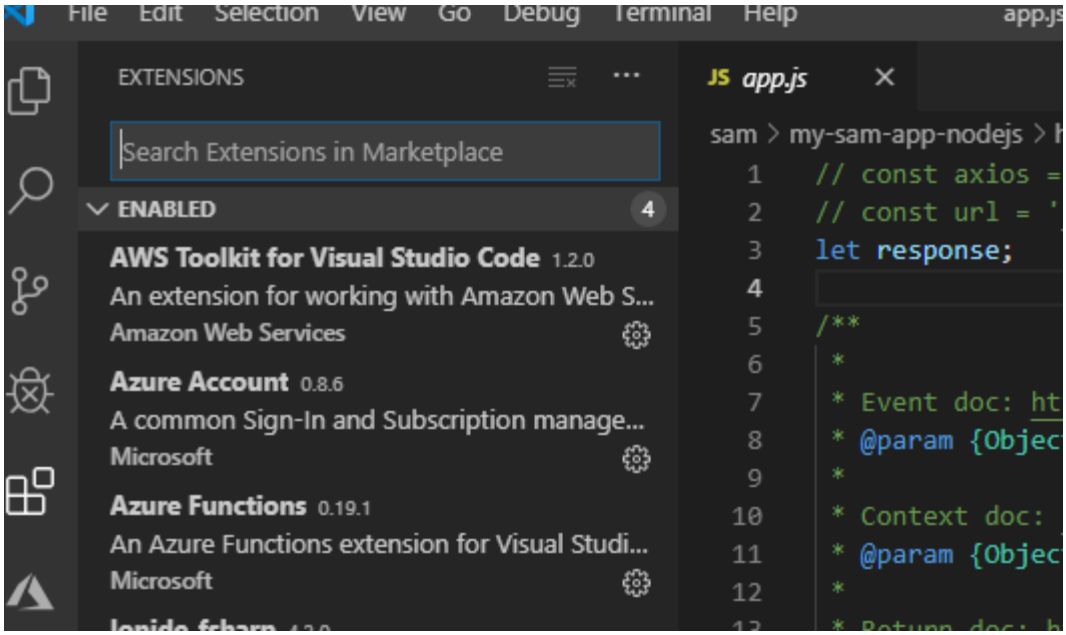


Figure VS.1

DEVELOPMENT ENVIRONMENT – DEPLOY SAMPLE SAM – COMMAND LINE

In order to test the toolchain I tried to deploy a sample application from the Command Line. Figure SAM.CMD.1

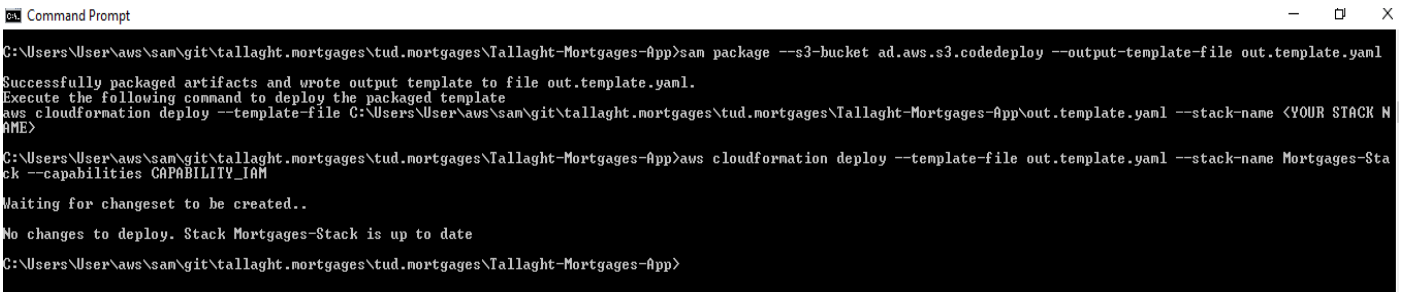


Figure SAM.CMD.1

DEVELOPMENT ENVIRONMENT – DEPLOY SAMPLE SAM – VS CODE

In order to test the toolchain and IDE I tried to deploy a sample application from the Visual Studio Code IDE. Figure SAM.VSC.1, Figure SAM.VSC.2.

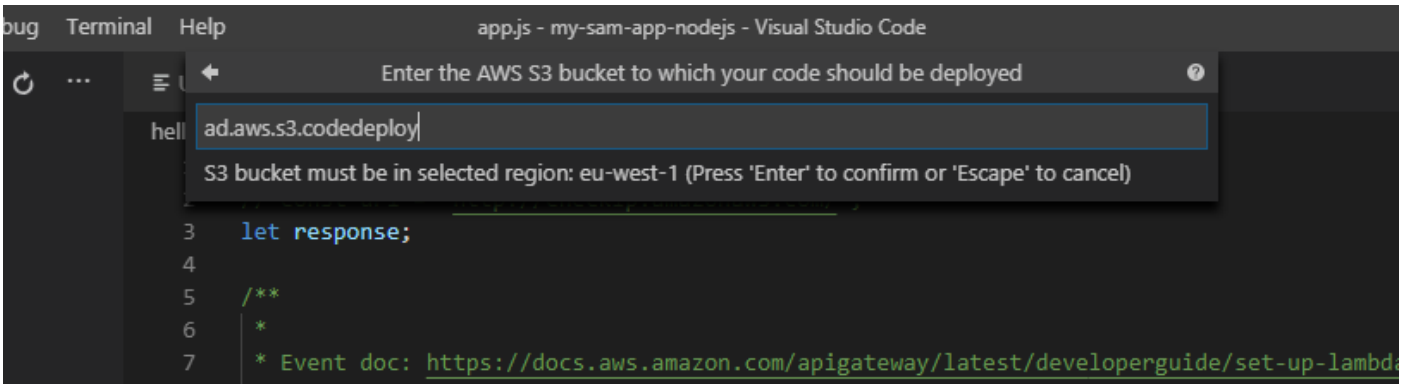


Figure SAM.VSC.1

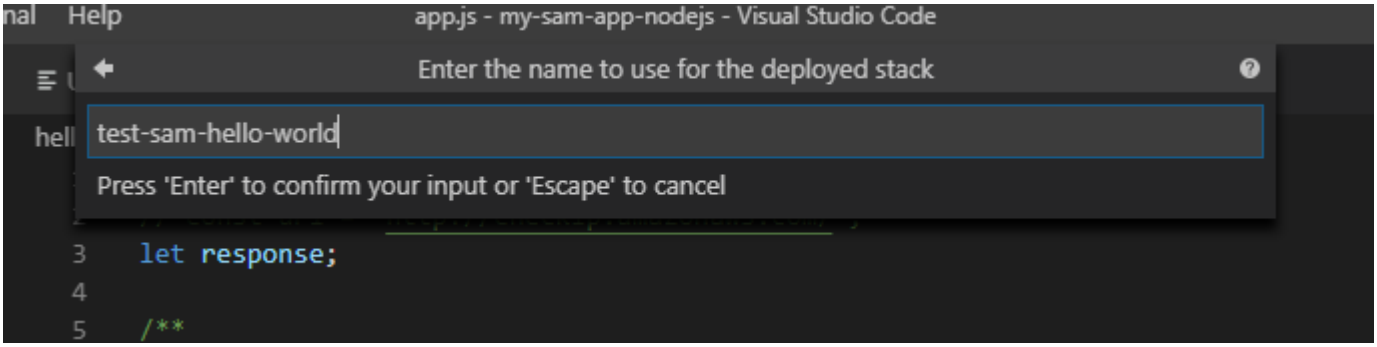


Figure SAM.VSC.2

My first attempt to deploy the code from the IDE failed. Figure SAM.VSC.3

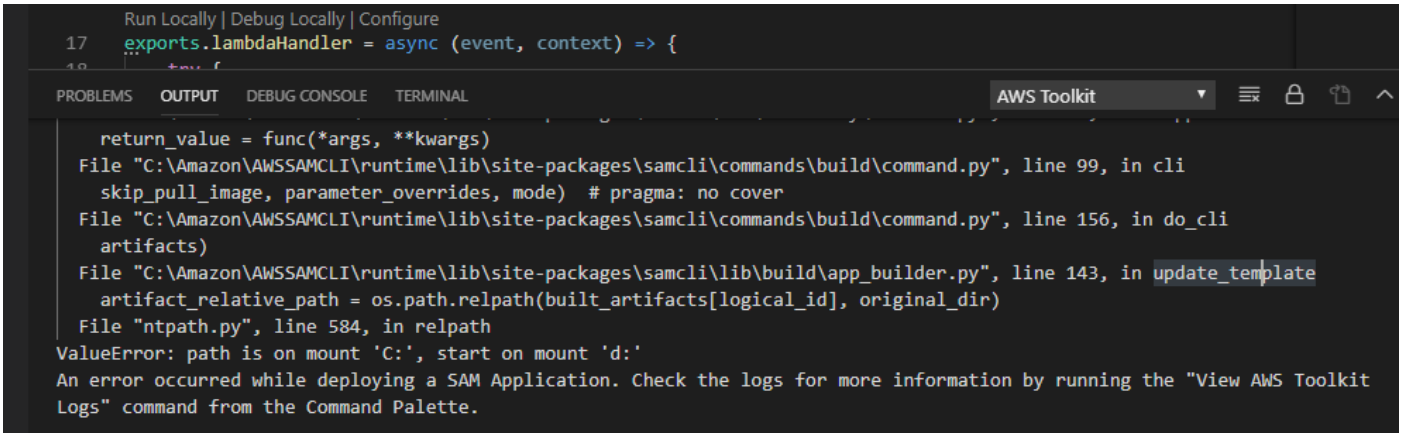


Figure SAM.VSC.3

It appears that the VSCode could not create a relative path between the IDE workspace installed on C: and the project code location on D:. I was not able to find a configuration setting to resolve this error. However the workaround was to move the project code to the same drive.

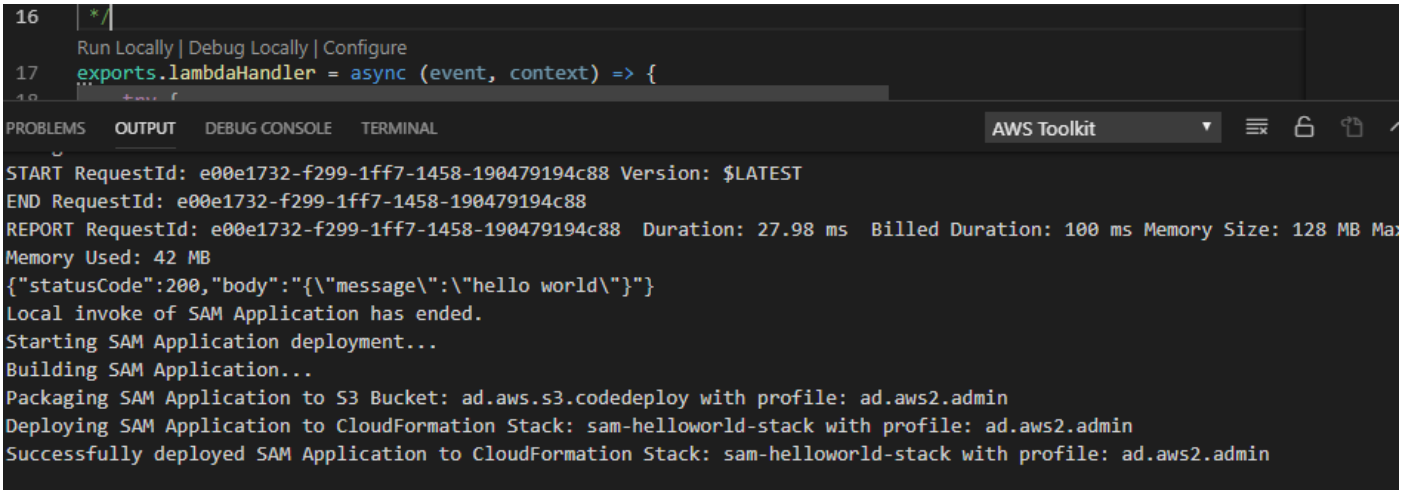


Figure SAM.VSC.3

DEVELOPMENT ENVIRONMENT – TEST SAM LOCALLY

In order to aid and speed up development it is possible to configure a Serverless Application Model application to run locally for testing purposes. However to do that it is required to install Docker.

Initially I installed Docker Desktop. But this would not run unless a commercial licence for Windows Server exists. I had to uninstall Docker Desktop. I then installed Docker Toolbox [8]. I had numerous errors trying to deploy the SAM application locally on Docker. There were permissions issues, path problems, IDE configuration issues. It took 2 evenings to resolve all of the errors. However it is now possible for me to deploy and debug locally. Figure SAM.VSC.4

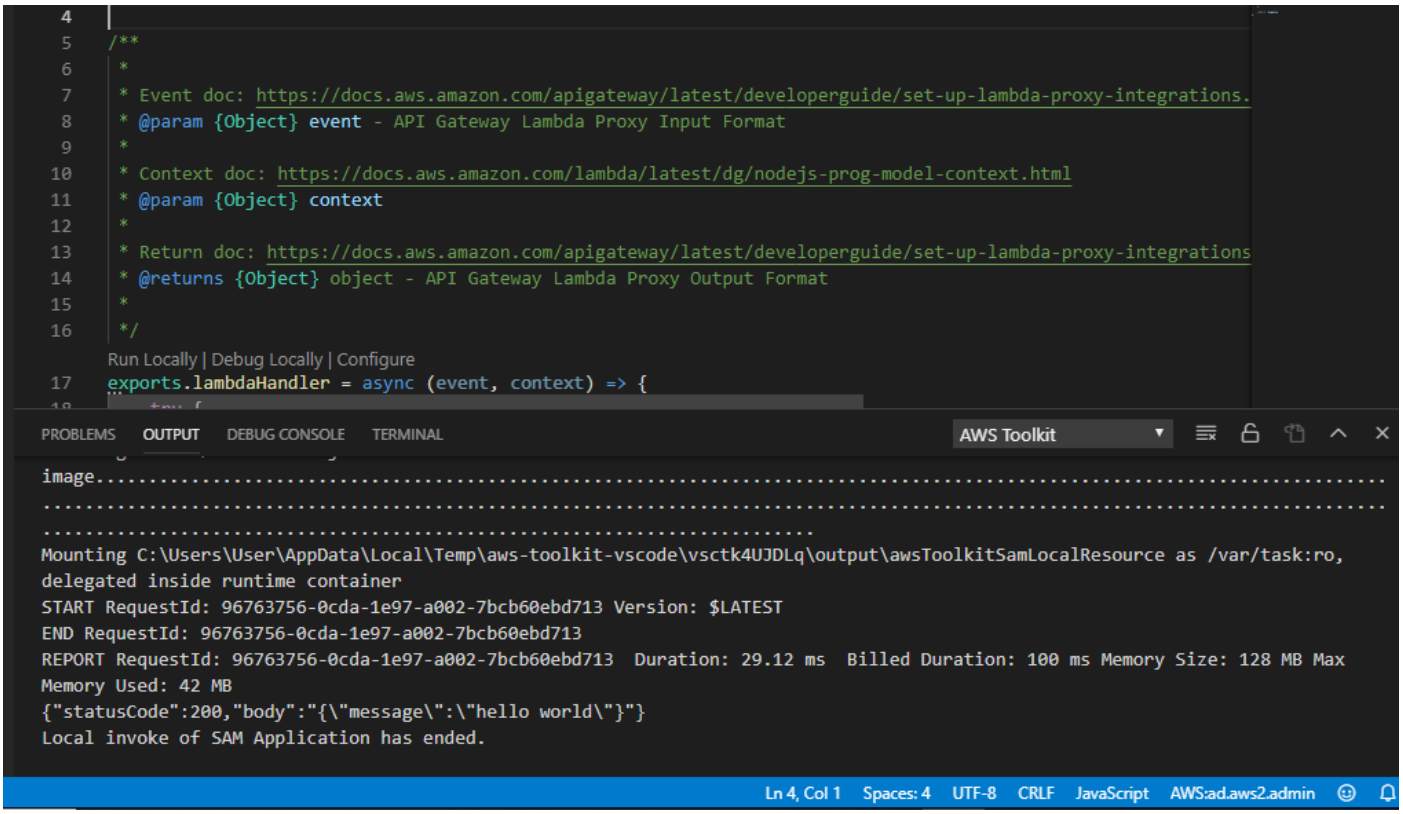


Figure SAM.VSC.4

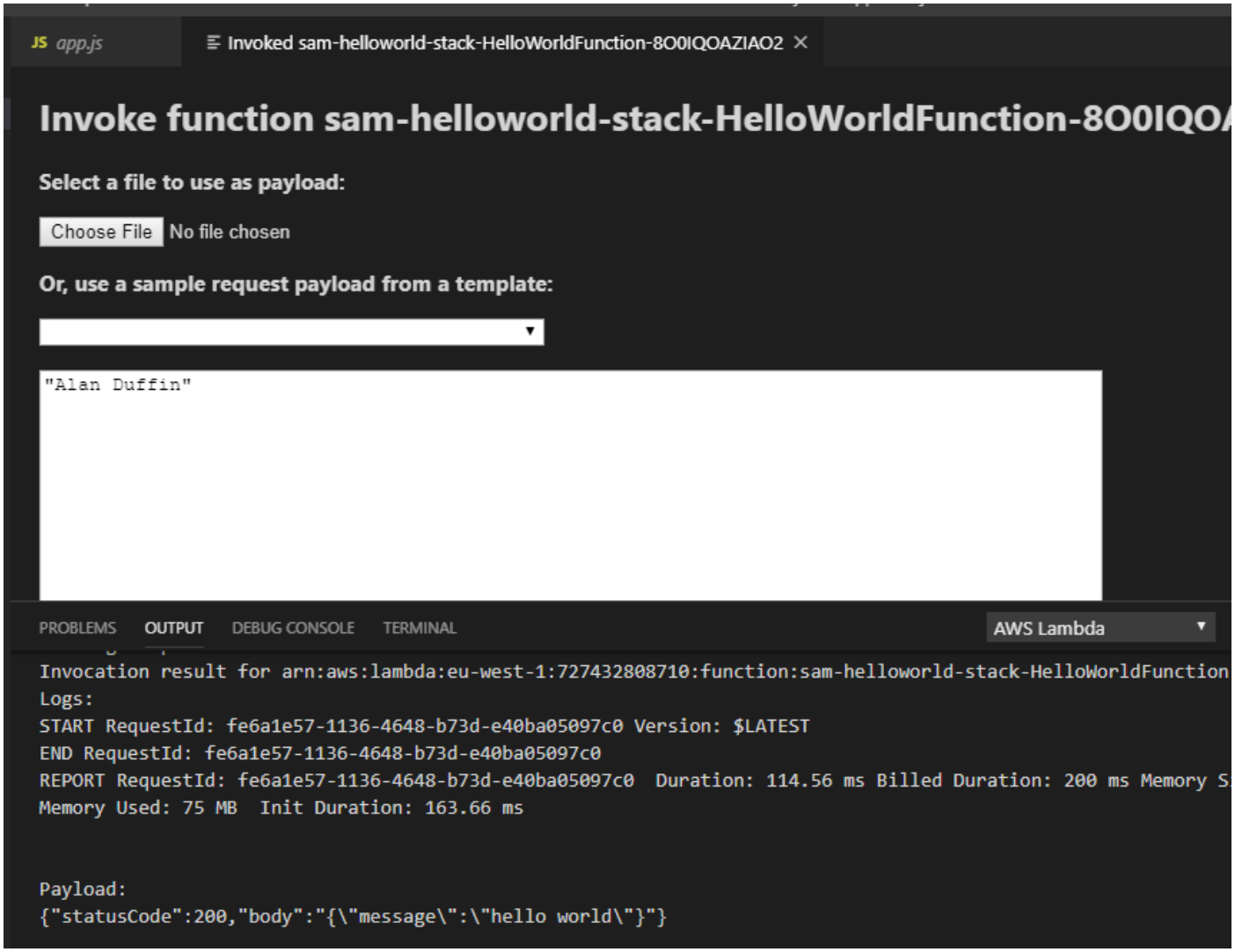


Figure SAM.VSC.5

DEVELOPMENT – BUILD DYNAMODB STACK

Cloudformation Stack link [9]. This CloudFormation template will create the tables listed in Figure DB.1.

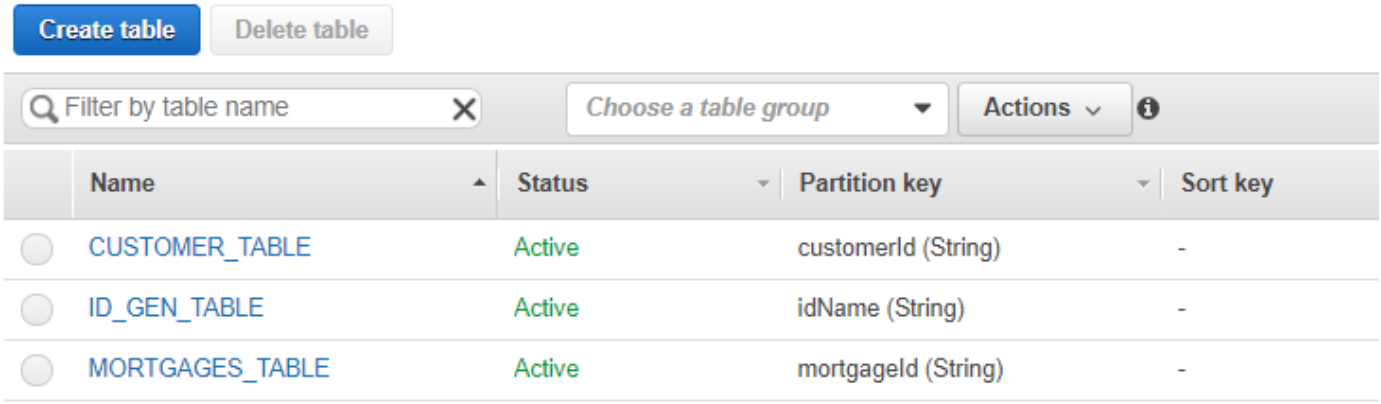


Figure DB.1

In order to create unique Identifiers for Customers and Mortgage applications I needed to create a table that would store a counter for the ids and allow atomic updates when new customers and mortgage applications are created. UUIDs would not be suitable for customer id or mortgage application numbers.

The ID_GEN_TABLE structure is shown in Figure DB.2

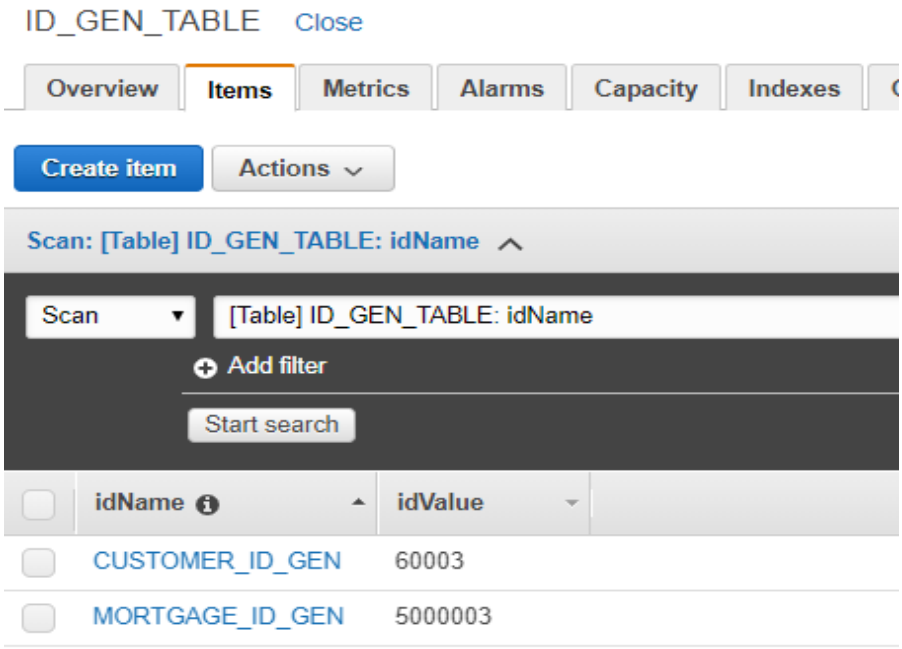


Figure DB.2

With regard to the Mortgages table, there is a requirement to allow application users to search all mortgage applications by the status code, e.g. 'PreSubmission' or 'In Assessment'. I tried a number of attempts to structure the table to achieve this but failed in the first 2 or 3 attempts. I had though originally that the statusCode should be a Range Key and make up part of a Composite primary key. However this does not provide the required access. When a table has a composite primary key then all getItem operations MUST contain both the Hash and Range key. However in this scenario we do not know both the Hash and Range key. So the solution was to create Secondary Global Indez which allows a different Hash Key that the main table. In this way we can use the getItem on the table when a single mortgage is required and we know the id. Then when seaching for all mortgages that currently have a particular status we can perform a query operation on the Index. Figure DB.3

```
KeySchema:
  - AttributeName: mortgageId
    KeyType: HASH
BillingMode: PAY_PER_REQUEST
GlobalSecondaryIndexes:
  - IndexName: MortgateStatusIndex
    KeySchema:
      - AttributeName: mortgageStatus
        KeyType: HASH
    Projection:
      ProjectionType: ALL
```

Figure DB.3

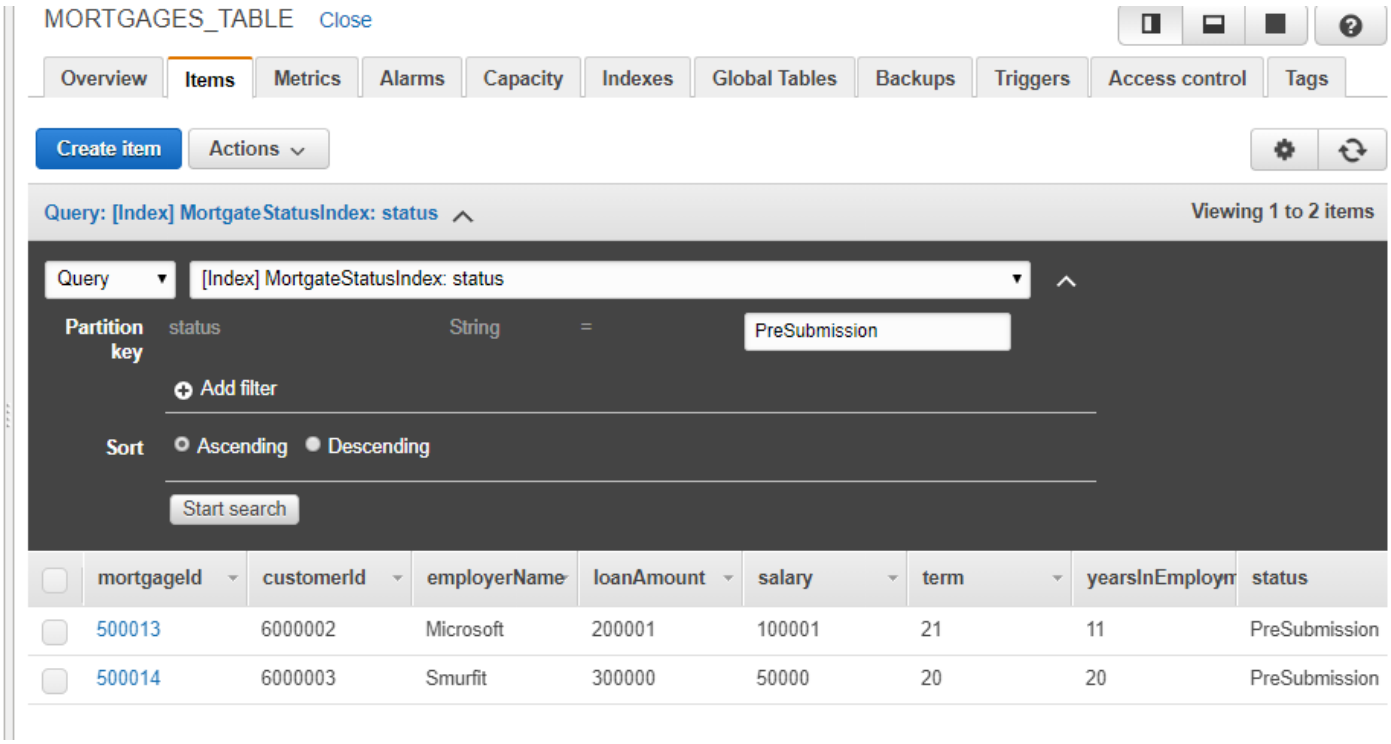

MORTGAGES_TABLE
Close
Overview
Items
Metrics
Alarms
Capacity
Indexes
Global Tables
Backups
Triggers
Access control
Tags
Create item
Actions
Query: [Index] MortgateStatusIndex: status
Viewing 1 to 2 items
Query
[Index] MortgateStatusIndex: status
Partition key
status
String
=
PreSubmission
Add filter
Sort
Ascending
Descending
Start search
mortgageId
customerId
employerName
loanAmount
salary
term
yearsInEmploym
status
500013
6000002
Microsoft
200001
100001
21
11
PreSubmission
500014
6000003
Smurfit
300000
50000
20
20
PreSubmission

Figure DB.4

DEVELOPMENT – BUILD SERVERLESS SERVICES – LAMBDA.

Cloudformation Stack link [10].

It has taken between 16-18 hours this week alone to get a small suite of HTTP services built, deployed and tested. There were many issues and problems to be worked through. There was the learning curve on the following areas:

1 How to properly structure Restful HTTP operations.

2 Learning how to work with NodeJS.

3 Learning the AWS Serverless Application Model framework concepts and syntax.

4 Once the basic service APIs were deployed, I then had to learn how to used the AWS SDK to integrate with DynamoDB. This took quite a long time to get even the simplest operations working. The DynamoDB operations have a complex set of required parameters and difficult syntax. See Figure LAMBDA.1

6 Understanding the Promise and asyn / await mechanisms for calling asynchronous AWS SDK operations also took time.

```
try {
  const genIdResponse = await dynamoDb.updateItem({
      "TableName": "ID_GEN_TABLE",
      "ReturnValues": "UPDATED_NEW",
      "ExpressionAttributeValues": {
        ":incr":{"N":"1"}
      },
      "UpdateExpression": "SET idValue = idValue + :incr",
      "Key": {
        "idName": {
          "S": "MORTGAGE_ID_GEN"
        }
      }
    }
  ).promise();
```

Figure LAMBDA.1

This figure illustrates the required query that is necessary to atomically read and increment a DynamoDB attribute.

Figure LAMBDA.2 illustrates the required operation structure for a simple update on the mortgage items.

```
var newMortgage = {
 TableName: "MORTGAGES_TABLE",
  Key: {
   "mortgageId": {
     S: mortgageId
    }
  },
  ExpressionAttributeNames: {
    "#A": "loanAmount",
    "#B": "yearsInEmployment",
    "#C": "salary",
    "#D": "employerName",
    "#E": "term",
    "#F": "mortgageStatus"
  },
  UpdateExpression: "set #A = :a, #B = :b, #C = :c, #D = :d, #E = :e, #F = :f",
  ExpressionAttributeValues: {
    ":a": { "N": mortgageRequest.loanAmount },
    ":b": { "N": mortgageRequest.yearsInEmployment },
    ":c": { "N": mortgageRequest.salary },
    ":d": { "S": mortgageRequest.employerName },
    ":e": { "N": mortgageRequest.term },
    ":f": { "S": mortgageRequest.mortgageStatus }

  },
  ReturnValues: "ALL_NEW"
};
```

Figure LAMBDA.2

There were many errors encountered. A sample can be found illustrated in Figure LAMBDA.3 and LAMBDA.4

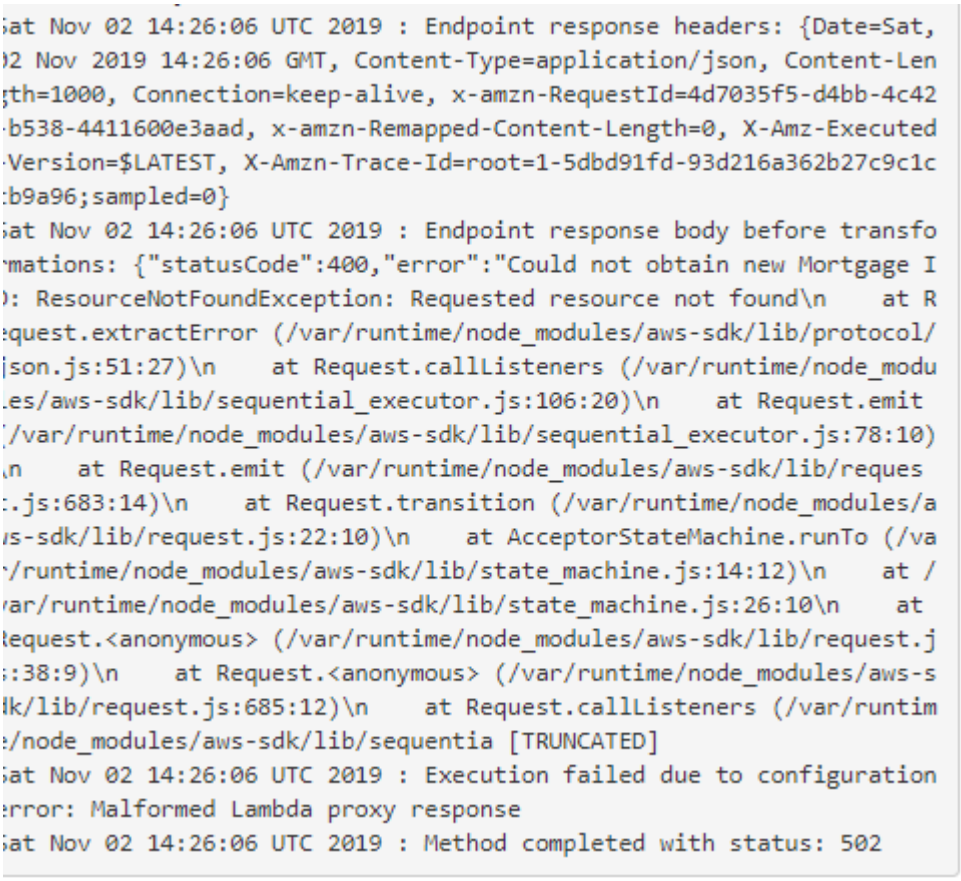


Figure LAMBDA.3

```
ration latency: 1527 ms
Sat Nov 02 14:39:27 UTC 2019 : Endpoint response headers: {Date=Sat,
02 Nov 2019 14:39:27 GMT, Content-Type=application/json, Content-Len
gth=1288, Connection=keep-alive, x-amzn-RequestId=e984d21b-7bbb-48ff
-bd66-c907c96d23f1, x-amzn-Remapped-Content-Length=0, X-Amz-Executed
-Version=$LATEST, X-Amzn-Trace-Id=root=1-5dbd951e-5462164ee1e75ae8fd
5659f5;sampled=0}
Sat Nov 02 14:39:27 UTC 2019 : Endpoint response body before transfo
rmations: {"statusCode":400,"error":"Could not obtain new Mortgage I
D: MultipleValidationErrors: There were 4 validation errors:\n* Inva
lidParameterType: Expected params.Item['term'].N to be a string\n* I
nvalidParameterType: Expected params.Item['salary'].N to be a string
\n* InvalidParameterType: Expected params.Item['yearsInEmployment'].
N to be a string\n* InvalidParameterType: Expected params.Item['loan
Amount'].N to be a string\n    at ParamValidator.validate (/var/runt
ime/node_modules/aws-sdk/lib/param_validator.js:40:28)\n    at Reque
st.VALIDATE_PARAMETERS (/var/runtime/node_modules/aws-sdk/lib/event_
listeners.js:126:42)\n    at Request.callListeners (/var/runtime/nod
e_modules/aws-sdk/lib/sequential_executor.js:106:20)\n    at callNex
tListener (/var/runtime/node_modules/aws-sdk/lib/sequential_executo
r.js:96:12)\n    at /var/runtime/node_modules/aws-sdk/lib/event_list
eners.js:86:9\n    at finish (/var/runtime/node_modules/aws-sdk/lib/
config.js:349:7)\n    at /var/runtim [TRUNCATED]
Sat Nov 02 14:39:27 UTC 2019 : Execution failed due to configuration
error: Malformed Lambda proxy response
Sat Nov 02 14:39:27 UTC 2019 : Method completed with status: 502
```

LAMBDA.4

The Lambda functions created so far are shown in Figure LAMBDA.5

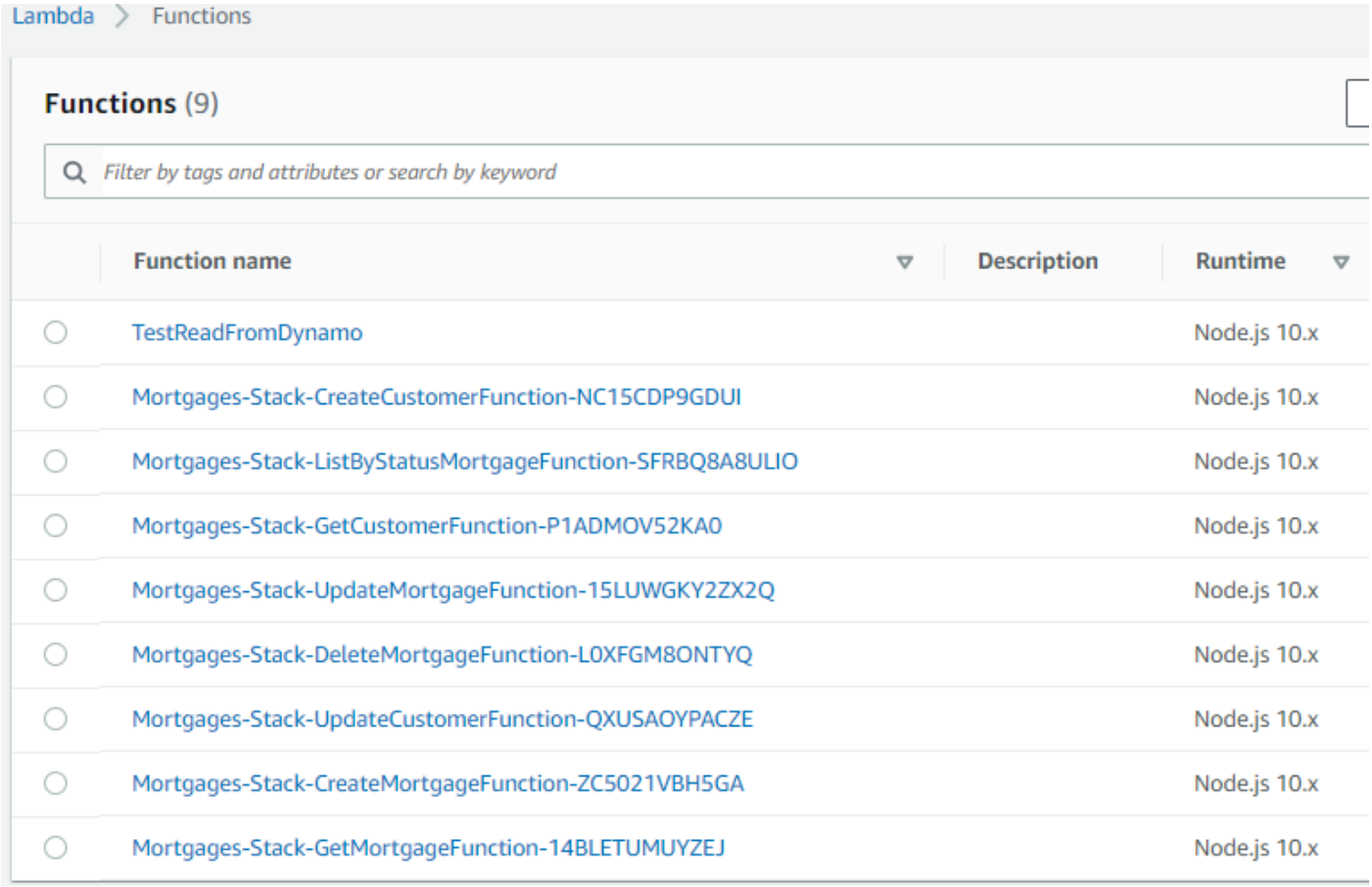


Figure LAMBDA.5

The API Gateway resource that have been create are listed in Figure LAMBDA.6

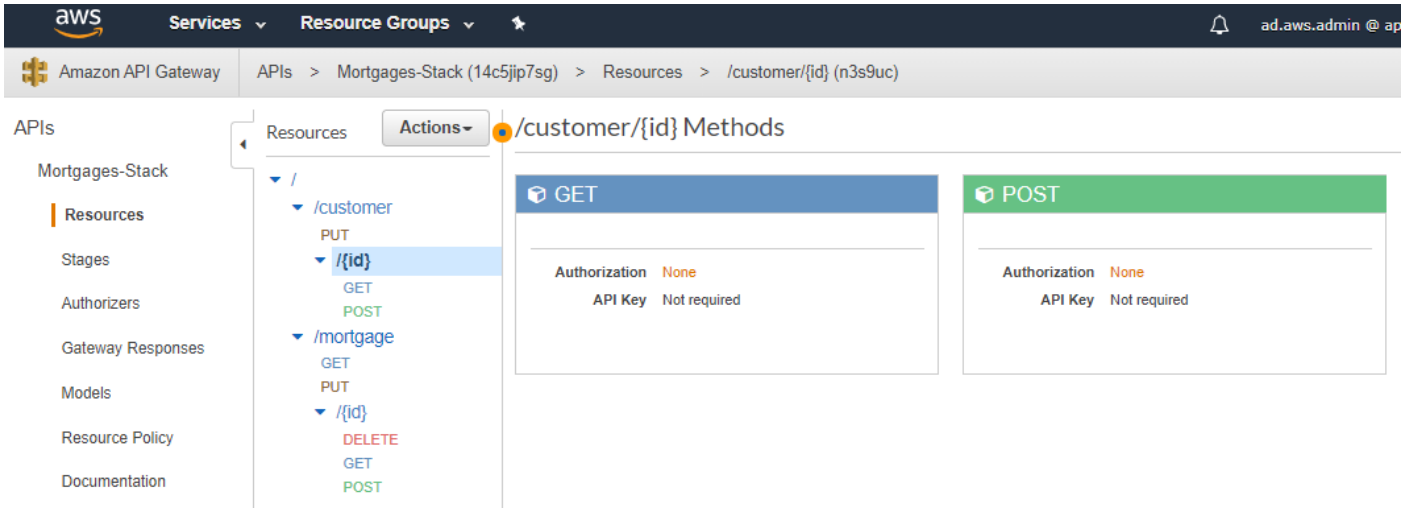


Figure LAMBDA.6

DEVELOPMENT – WORKFLOW IMPLMENTATION – STEP FUNCTION.

The template for the inital implementation can be found at the link [11]. An illustration of the Step Function implementation is shown in Figure STEP.1.

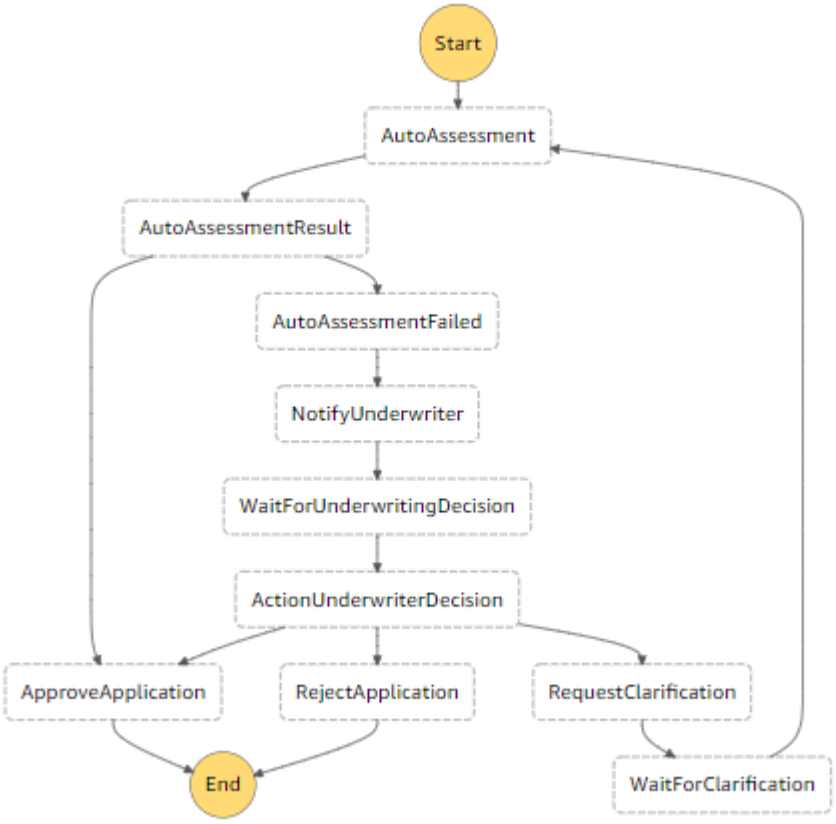


Figure STEP.1

REFERENCE LIST AND BIBLIOGRAPHY

[1] AWS Free Tier. https://aws.amazon.com/free/

[2] AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html

[3] AWS CLI SAM. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html

[4] Node JS. https://nodejs.org/en/download/

[5] Configure AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

[6] Visual Studio Code. https://code.visualstudio.com/download

[7] Configure AWS on Visual Studio Code. https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html

[8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/

[9] DynamoDB Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml

[10] Lambda Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml

[11] Step Function State Machine Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV1.json

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Implementation Iteration 2.1

Student Name: Alan Duffin

Student ID: X00159409

November 2019

SUBMISSION MATERIAL

Source Code NodeJS / Lambda	https://github.com/GeneralYeager/tallaght.mortgages/tree/master/tud.mortgages/Tallaght-Mortgages-App
Workflow Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt
DynamoDB Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml
Lambda / API Gateway Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml
Angular SPA Source Code	https://github.com/GeneralYeager/tallaght.mortgages/tree/master/mort-app-web

Table A.

ITERATION 2.1 TASKS

Building upon last weeks progress report, this document outlines the additional project tasks that have been completed for implementation phase 2. The follow tasks were completed this week.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Websockets	Human interaction requires notification with User Interface.
Build / Development	AWS Lambda	Build Lambda functions to manage API Gateway Websockets and notifications.
Build / Development	DynamoDB	Build table to manage API Websocket connection tokens.
Build / Development	Angular SPA	Build Browser based Front End to allow users to create/modify/submit Mortgage Applications for workflow assessment.
Test		Test Websocket Notification Events and subscriptions
Build / Development	AWS Lambda	Additiona Lambda functions to support Step Function execution.
Build / Development	AWS Step Functions	Extend Step Function state machine to support Human decision making.
Design	AWS Step Functions	Integrate Step Function generation into Cloudformation template.
Test	AWS Step Functions	Test State Transitions / Events

Table B.

I have extended the suite of Lambda services that will allow a the Step function state machine to transition Mortgage application states and allow notification to different user categories (Brokers and Underwriters). In order to support this I have had to build an intial Browser based Single Page Application using the Javascript Angular framework. At this point Brokers can create and amend mortgage applications. I have built Angular services that connect to the AWS API Gateway Websocket API that allows traffic to be initiated either by the client or the server. This is necessary so that uses can be notified of Mortgage application state changes. So users will be notified of State changes and required tasks via two channels SNS (email) and real time browser based websockets.

It has been necessary to build additional Lambda functions to support the integration between the Step Function State Machine and the API Gateway Websocket layer. Also some additional Lambda functions were developed specifically to transition the Mortgage Applications between states.

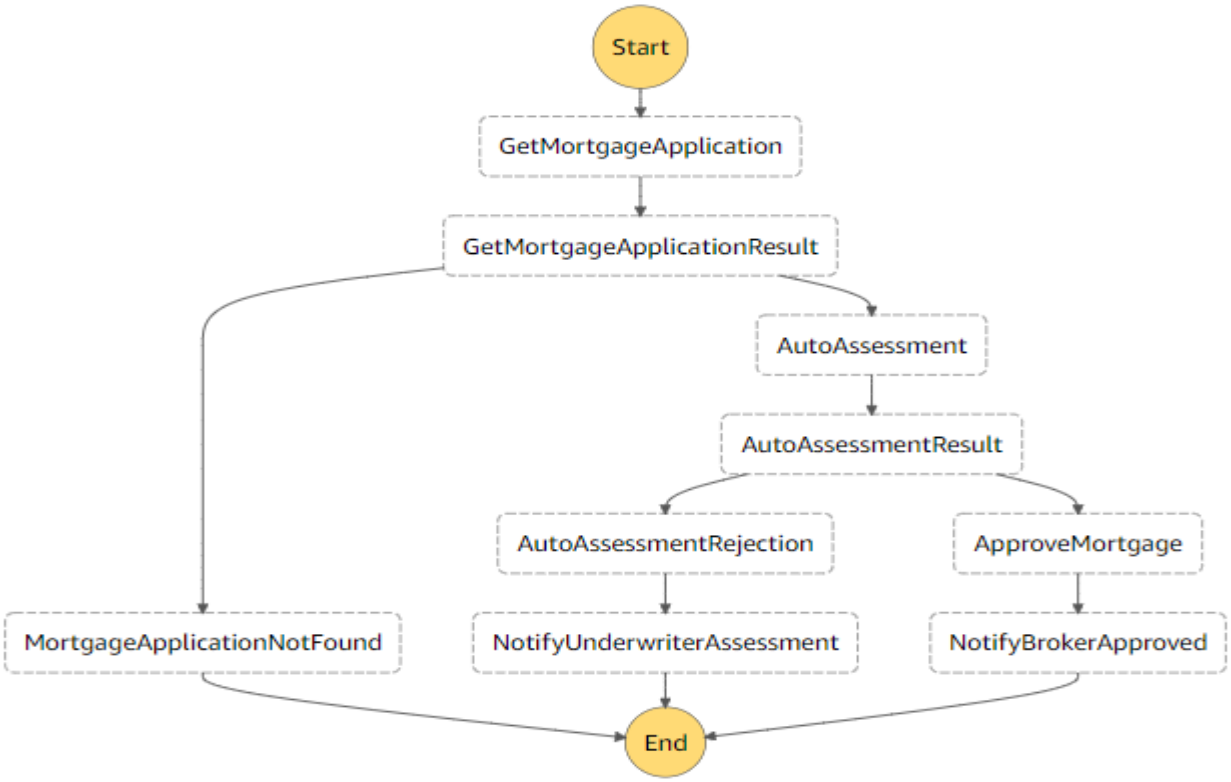
PLAN FOR ITERATION 2.2

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Complete the Step Function Workflow state machine.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Build / Development	AWS Lamda	Build the Lambda Functions to support pausing the workflow to await human decisions. Also manage the Step Function tokens.
Build / Development	DynamoDB	Implement new DB schema to map mortgage application to Step Function execution id. This is required to allow users approve or reject applications without reference to the step function execution id.
Build / Development	Angular	Extends the Browser application to allow submission of mortgages for assessment. Allow for clarification requests and Underwriter decision making.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table C.

The workflow state machine currently exists as seen in Figure AWS.STEP.1

Figure AWS.STEP.1


Start
GetMortgageApplication
GetMortgageApplicationResult
AutoAssessment
AutoAssessmentResult
AutoAssessmentRejection
ApproveMortgage
MortgageApplicationNotFound
NotifyUnderwriterAssessment
NotifyBrokerApproved
End

SERVLERLESS / LAMBDA FUNCTIONS

The following Lambda functions have now been completed as of this progress report. Figure AWS.LAMBDA.1

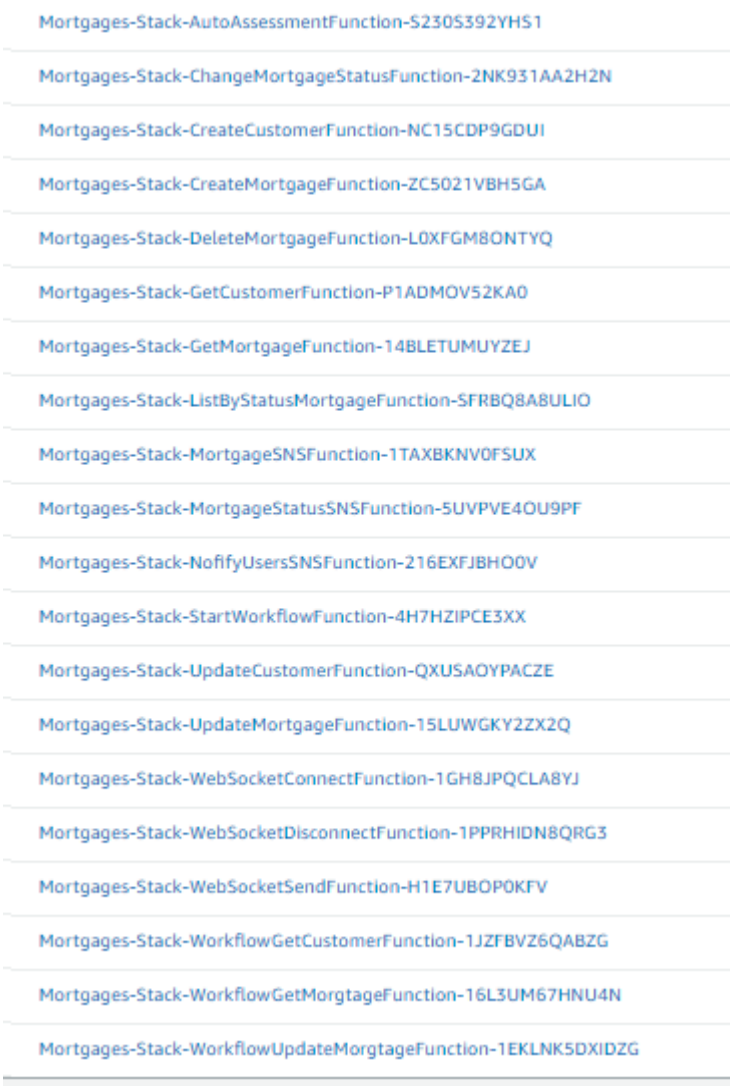


Figure AWS.LAMBDA.1

This table lists the purpose of each of the completed Lambda Functions. Table AWS.LAMBDA.1

Lambda Function	Purpose
Mortgages-Stack-AutoAssessmentFunction	Perform Mortgage Rules check
Mortgages-Stack-ChangeMortgageStatusFunction	Modify the Mortgage Application Status and store.
Mortgages-Stack-CreateCustomerFunction	HTTP API Create Customer
Mortgages-Stack-CreateMortgageFunction	HTTP API Create Mortgage Application
Mortgages-Stack-DeleteMortgageFunction	HTTP API Delete Mortgage Application
Mortgages-Stack-GetCustomerFunction	HTTP API Find By PK Customer
Mortgages-Stack-GetMortgageFunction	HTTP API Find By PK Mortgage Application
Mortgages-Stack-ListByStatusMortgageFunction	HTTP API Find By Status Mortgage Application
Mortgages-Stack-MortgageStatusSNSFunction	Step Function Update Mortgage Status
Mortgages-Stack-NofifyUsersSNSFunction	Step Function Notify State Change
Mortgages-Stack-StartWorkflowFunction	HTTP API Execute Mortgage Workflow
Mortgages-Stack-UpdateCustomerFunction	HTTP API Update Customer
Mortgages-Stack-UpdateMortgageFunction	HTTP API Update Mortgage
Mortgages-Stack-WebSocketConnectFunction	Manage Websocket connection and store token.
Mortgages-Stack-WebSocketDisconnectFunction	Manage Websocket disconnect and delete token.
Mortgages-Stack-WebSocketSendFunction	Step Function send message to API Gateway Websocket for Browser broadcast.
Mortgages-Stack-WorkflowGetCustomerFunction	Step Function Find By PK Customer
Mortgages-Stack-WorkflowGetMorgtageFunction	Step Function Find By PK Mortgage
Mortgages-Stack-WorkflowUpdateMorgtageFunction	Step Function Update Mortgage

Table AWS.LAMBDA.1

The source code and deployment template for these functions are available at the GitHub repository listed in Table A or in the Appendix.

I have had to implement new Lambda functions to manage API Gateway Websocket functionality.

API GATEWAY WEBSOCKETS

Last year AWS announced API Gateway support for the Websocket protocol [12]. This allows for the creation of ...

"bidirectional communication applications using WebSocket APIs in Amazon API Gateway without having to provision and manage any servers.

HTTP-based APIs use a request/response model with a client sending a request to a service and the service responding synchronously back to the client. WebSocket-based APIs are bidirectional in nature. This means that a client can send messages to a service and services can independently send messages to its clients.

This bidirectional behavior allows for richer types of client/service interactions because services can push data to clients without a client needing to make an explicit request. WebSocket APIs are often used in real-time applications such as chat applications, collaboration platforms, multiplayer games, and financial trading platforms."

The following API Gateway Websocket API has been created. Figure AWS.API.1

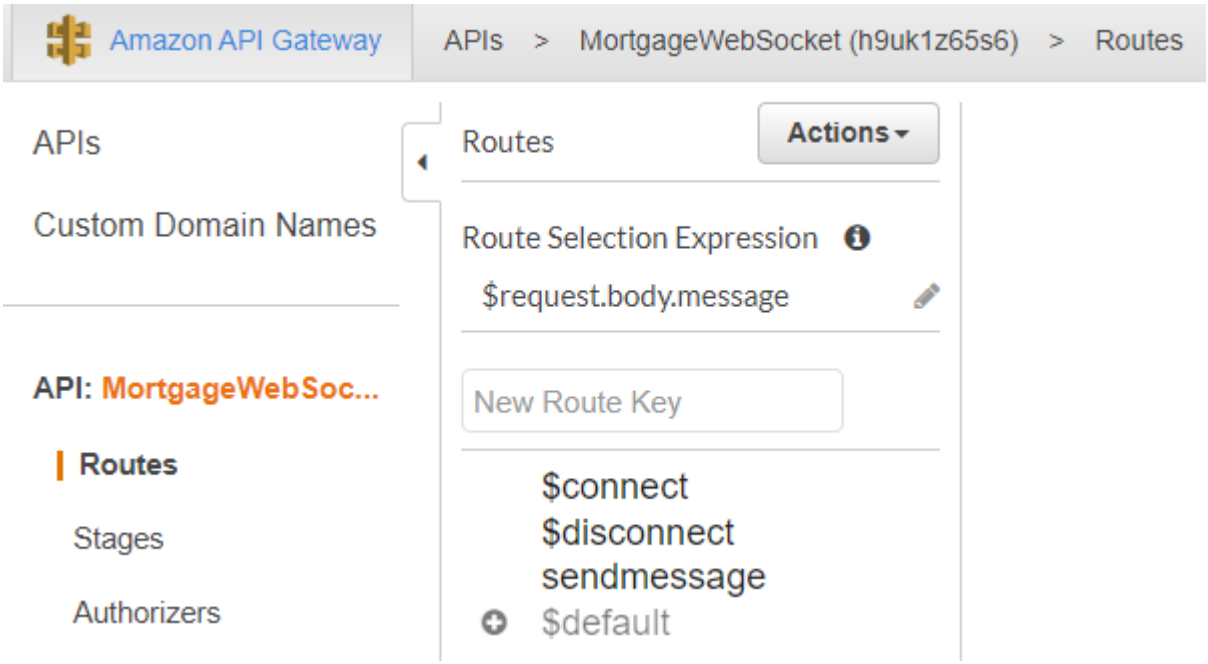


Figure AWS.API.1

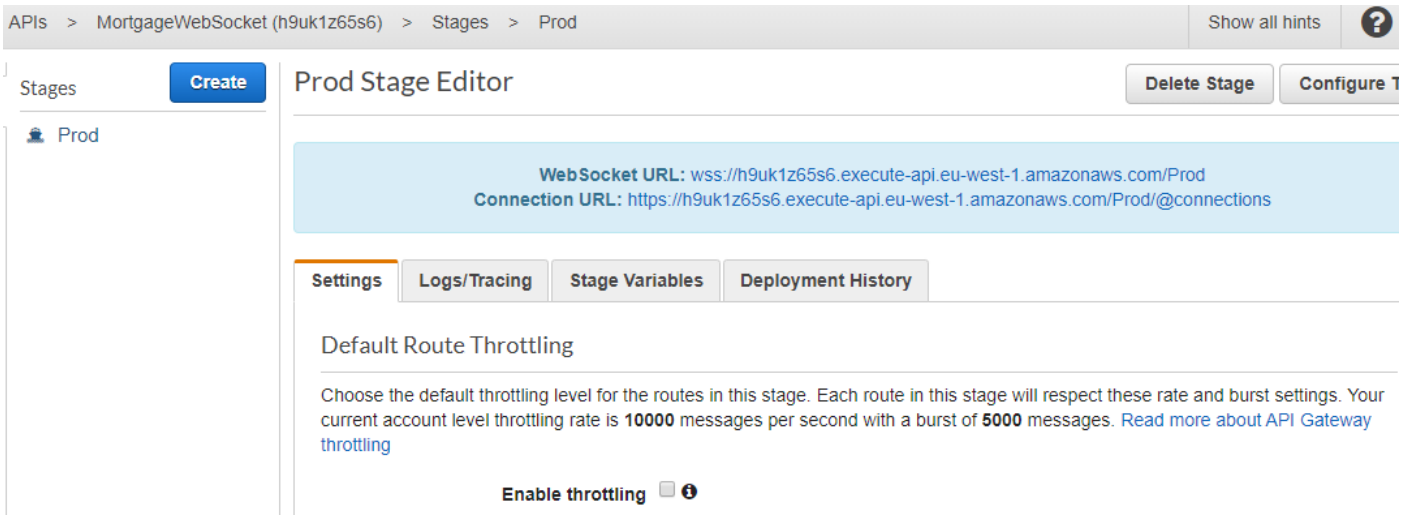


Figure AWS.API.2

Because API Gateway support for Websockets is very recent, the online help is minimal. It took many hours to get the deployment and develoment correct.

There were many errors encountered such as incorrect endpoint format and futile attempts to get Lambda functions to establish wss protocol connections.

This error was caused by an invalid endpoint format.

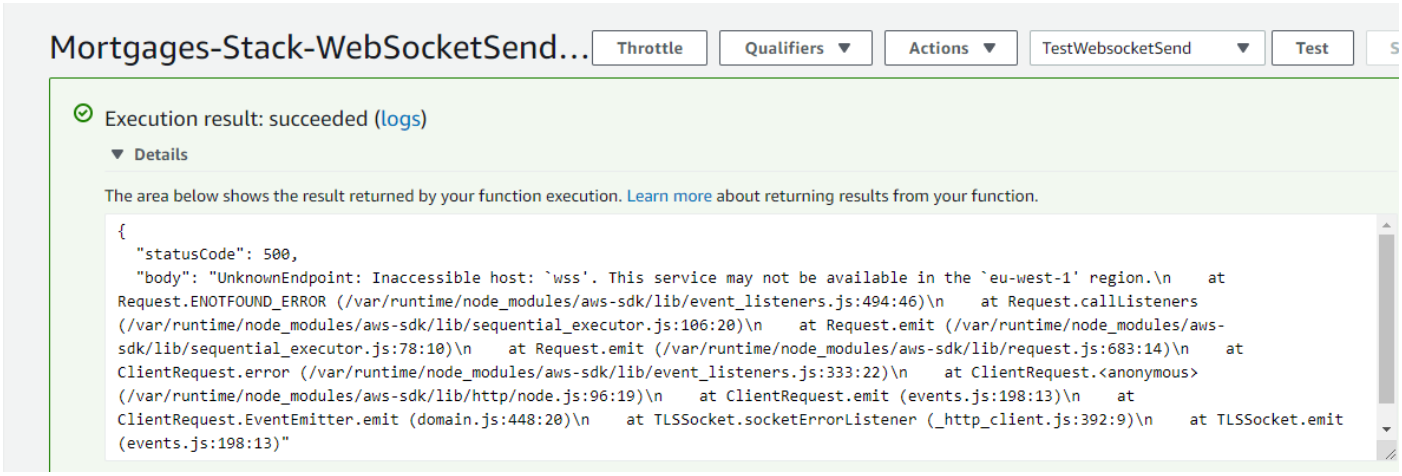


Figure AWS.SOCKET.1

It is not possible for Lambda functions to connect to an external WSS endpoint. They must directly use the 'AWS.ApiGatewayManagementApi' sdk and connect via the API Gateway instance.

```
395     Runtime: nodejs10.x
396     Environment:
397       Variables:
398         TABLE_NAME: !Ref WebsocketConnectionTableName
399       # WEBSOCKET_ENDPOINT: !Join [ '', [ 'wss://', !Ref MortgageWebSocket, '.execute-api.',!Ref 'A
400         WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-api.',!Ref 'AWS::Regior
401     Policies:
```

Figure AWS.SOCKET.2

Websocket Success. Figure AWS.SOCKET.3

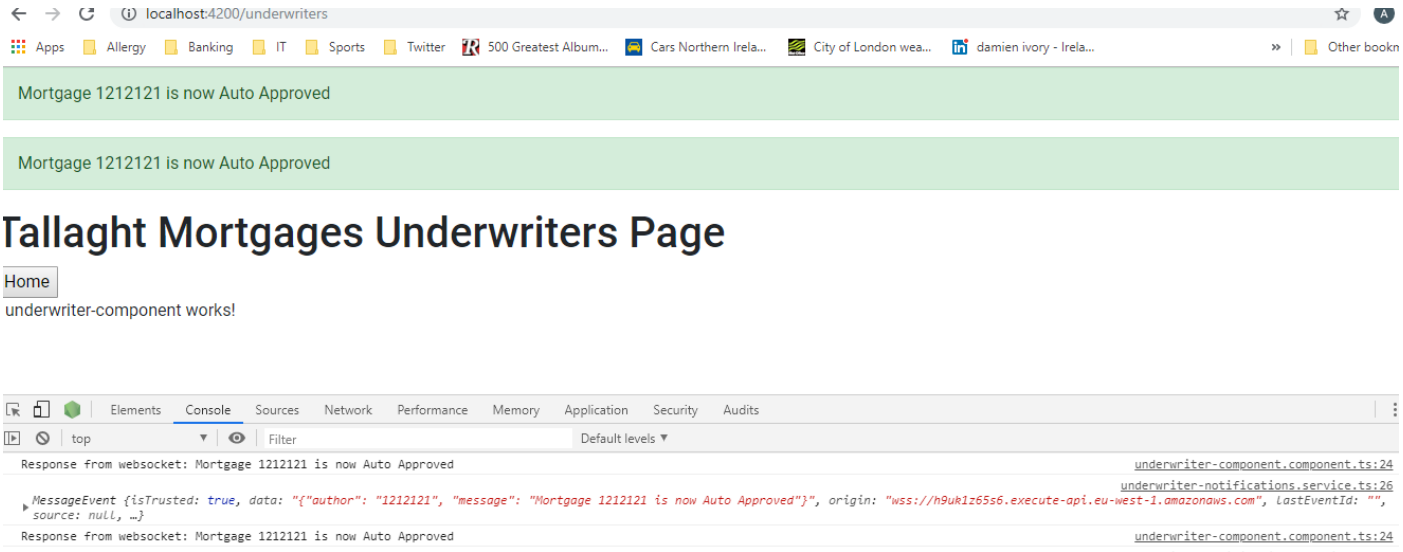


Figure AWS.SOCKET.3

Socket Broadcast Source Code:

```
exports.handler = async (event, context) => {
 let connectionData;
 console.log(event);
 try {
  connectionData = await ddb.scan({ TableName: TABLE_NAME, ProjectionExpression:
 'connectionId' }).promise();
 } catch (e) {
  return { statusCode: 500, body: e.stack };
 }

 const apigwManagementApi = new AWS.ApiGatewayManagementApi({
  apiVersion: '2018-11-29',
  endpoint: WEBSOCKET_ENDPOINT //event.requestContext.domainName + '/' + event.r
equestContext.stage
 });

 //const postData = JSON.parse(event.body).data;
 const postData = '{\"author\": \"1212121", \"message\": \"Mortgage 1212121 is
now Auto Approved"}';


 const postCalls = connectionData.Items.map(async ({ connectionId }) => {
  try {
   await apigwManagementApi.postToConnection({ ConnectionId: connectionId, Data:
 postData }).promise();
  } catch (e) {
   if (e.statusCode === 410) {
    console.log(`Found stale connection, deleting ${connectionId}`);
    await ddb.delete({ TableName: TABLE_NAME, Key: { connectionId } }).promise()
;
   } else {
    throw e;
   }
  }
 });

 try {
  await Promise.all(postCalls);
 } catch (e) {
  return { statusCode: 500, body: e.stack };
 }
```

```
 return { statusCode: 200, body: 'Data sent.' };
};
```

Web Browser Websocket Connection Source Code:

```
import { Injectable } from '@angular/core';

import { Observable, Observer, Subject } from 'rxjs'

@Injectable({
 providedIn: 'root'
})
export class WebsocketService {
 constructor() {}

 private subject: Subject<MessageEvent>;

 public connect(url): Subject<MessageEvent> {
  console.log("connect url");
  if (!this.subject) {
   this.subject = this.create(url);
   console.log("Successfully connected: " + url);
  }
  return this.subject;
 }

 private create(url): Subject<MessageEvent> {
  let ws = new WebSocket(url);
  console.log("create ws");
  let observable = Observable.create((obs: Observer<MessageEvent>) => {
   console.log("in obs");
   //ws.onmessage = obs.next.bind(obs);
   ws.onmessage = function(event) {
    console.log("WebSocket message received:", event);
    obs.next(event);
   };
   ws.onerror = obs.error.bind(obs);
   ws.onclose = obs.complete.bind(obs);
   return ws.close.bind(ws);
  });
  let observer = {
   next: (data: Object) => {
    console.log("in next");
    if (ws.readyState === WebSocket.OPEN) {
```

```
      ws.send(JSON.stringify(data));
    }
  }
 };
 return Subject.create(observer, observable);
}
}
```

ANGULAR SINGLE PAGE APPLICATION

In order to build a Javascript based Bowser application to demonstrate and interact with the Step Function Workflow I have built a first iteration of an Angular SPA application.

Steps to set up the development environment are:

1 Install npm. https://nodejs.org/en/

2 Install Angular CLI. https://angular.io/guide/setup-local

3 Install Angular Material Design. https://material.angular.io/guide/getting-started

4 Install Websocket NPM Package. https://www.npmjs.com/package/websocket

5 The source code completed so far is stored in Git Hub.

At the moment the SPA allows users to create / modify / list mortgage application. View Websocket notifications.

The look and feel is very basic but will be sufficient to provide a working system.

Run the SPA locally – shown below.

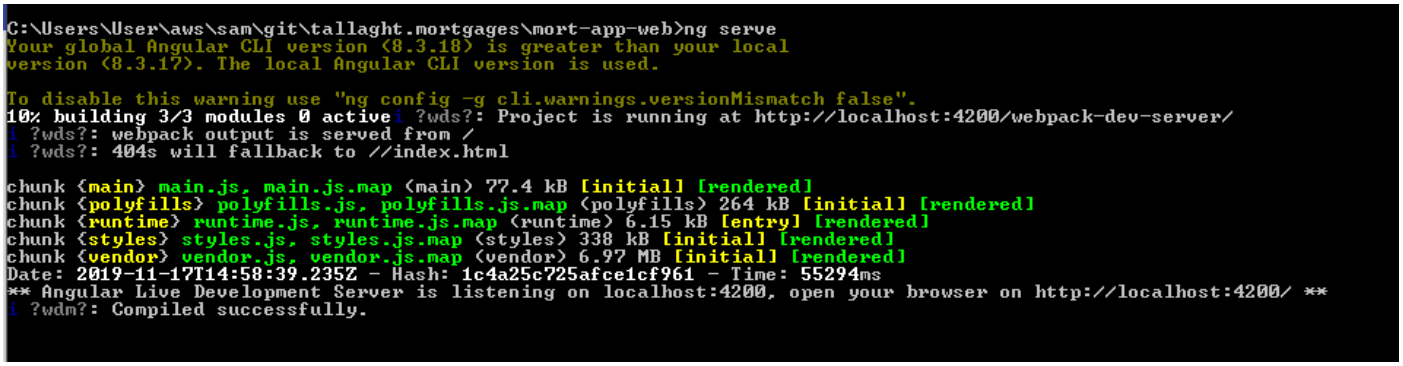
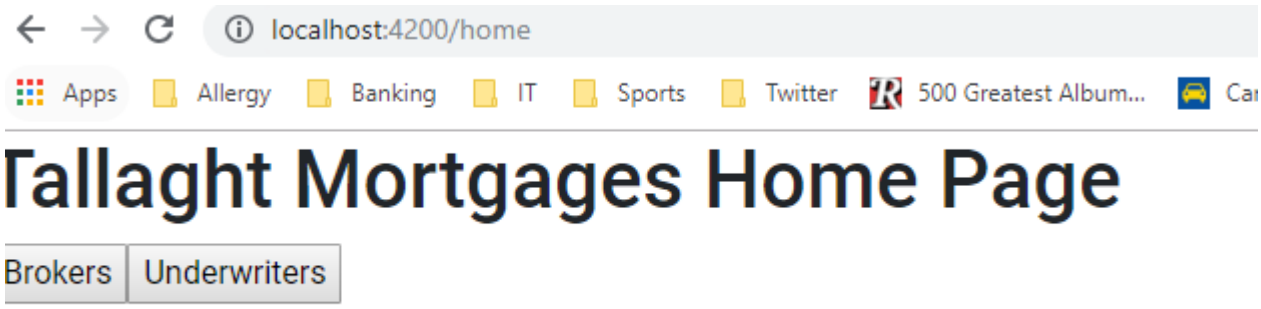
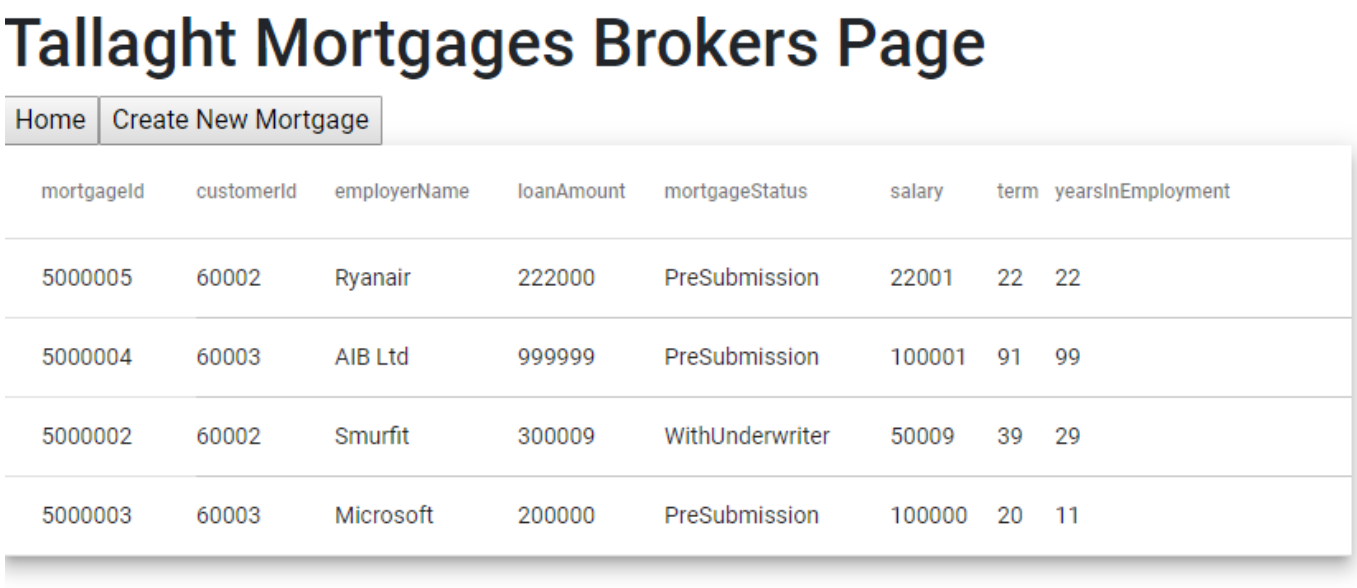


Figure AWS.ANGULAR.1

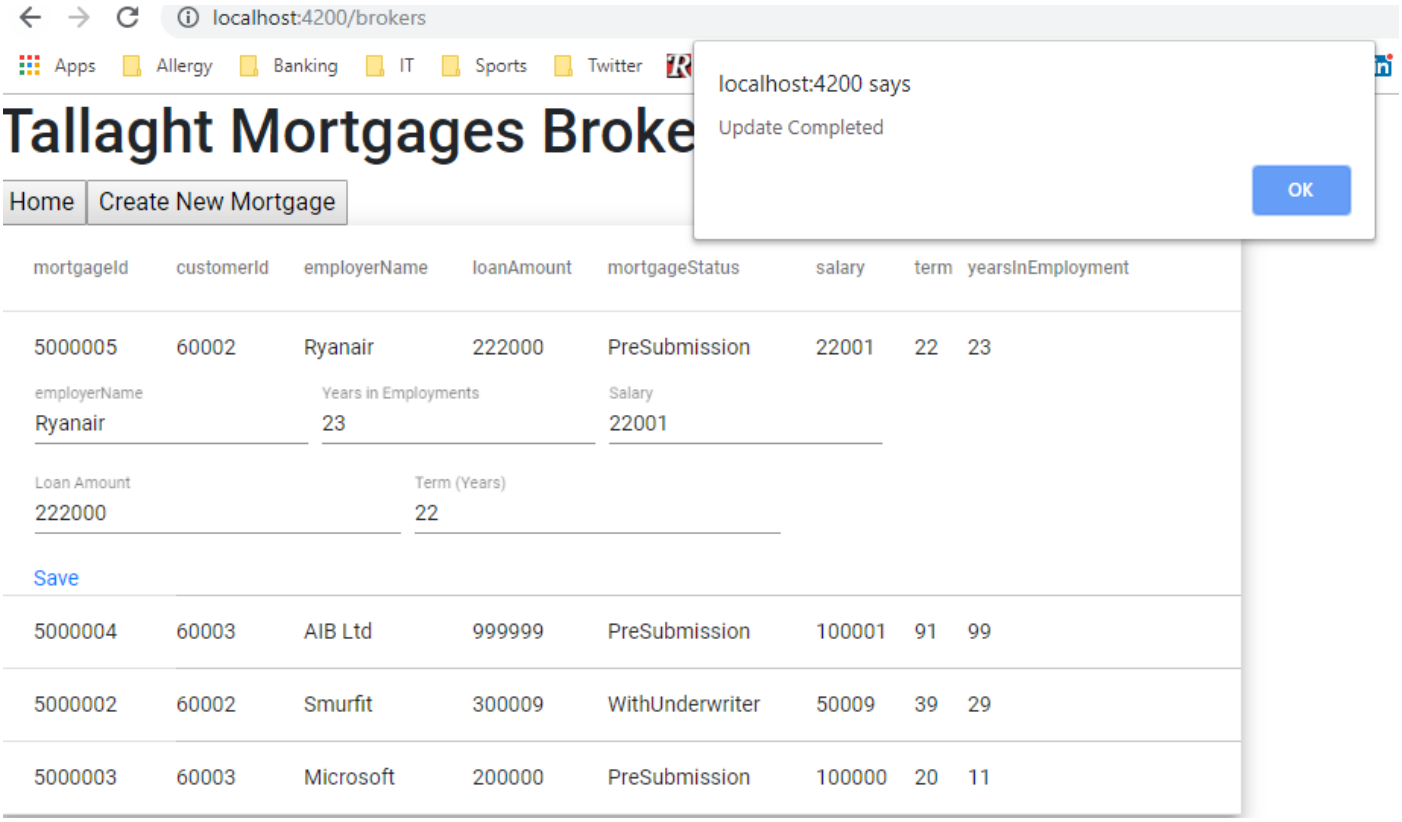
Home Page. Figure AWS.ANGULAR.2



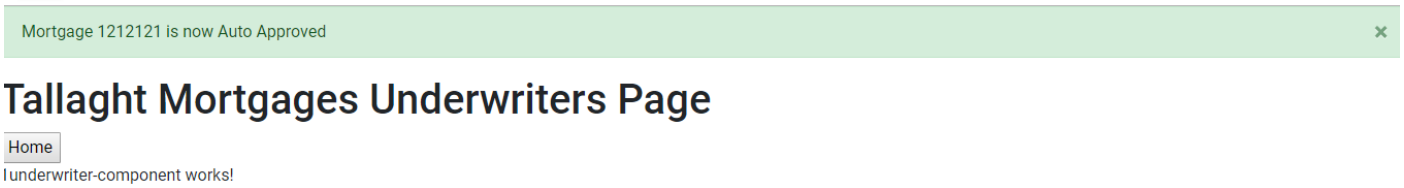


Broker Page. Figure AWS.ANGULAR.3

Modify Morgages. Figure AWS.ANGULAR.4



Recieve Mortgage Status Notifications. Figure AWS.LAMBDA.5



Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a **cross-origin HTTP request** when it requests a resource that has a different origin (domain, protocol, or port) from its own.

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

The AWS documentation enabling CORS on API Gateway via Cloudformation is lacking. It took a couple of days for me to get the HTTP API calls working from my local development environment.

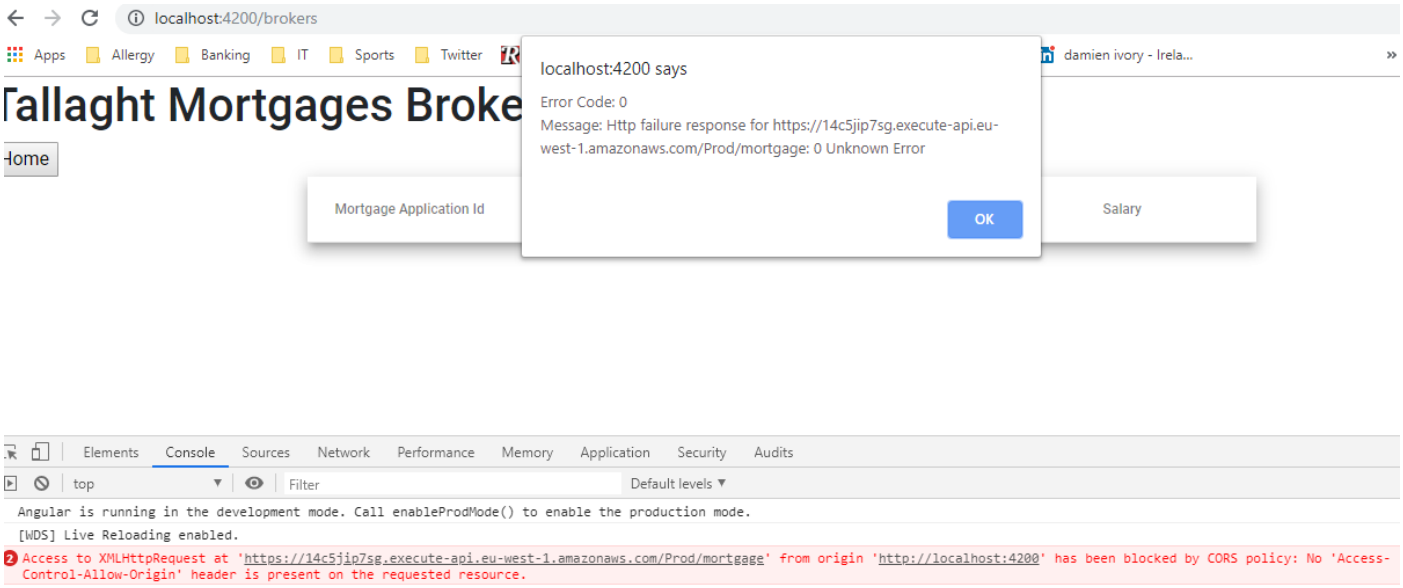


Figure AWS.CORS.1

Attempt to enable CORS post deployment.

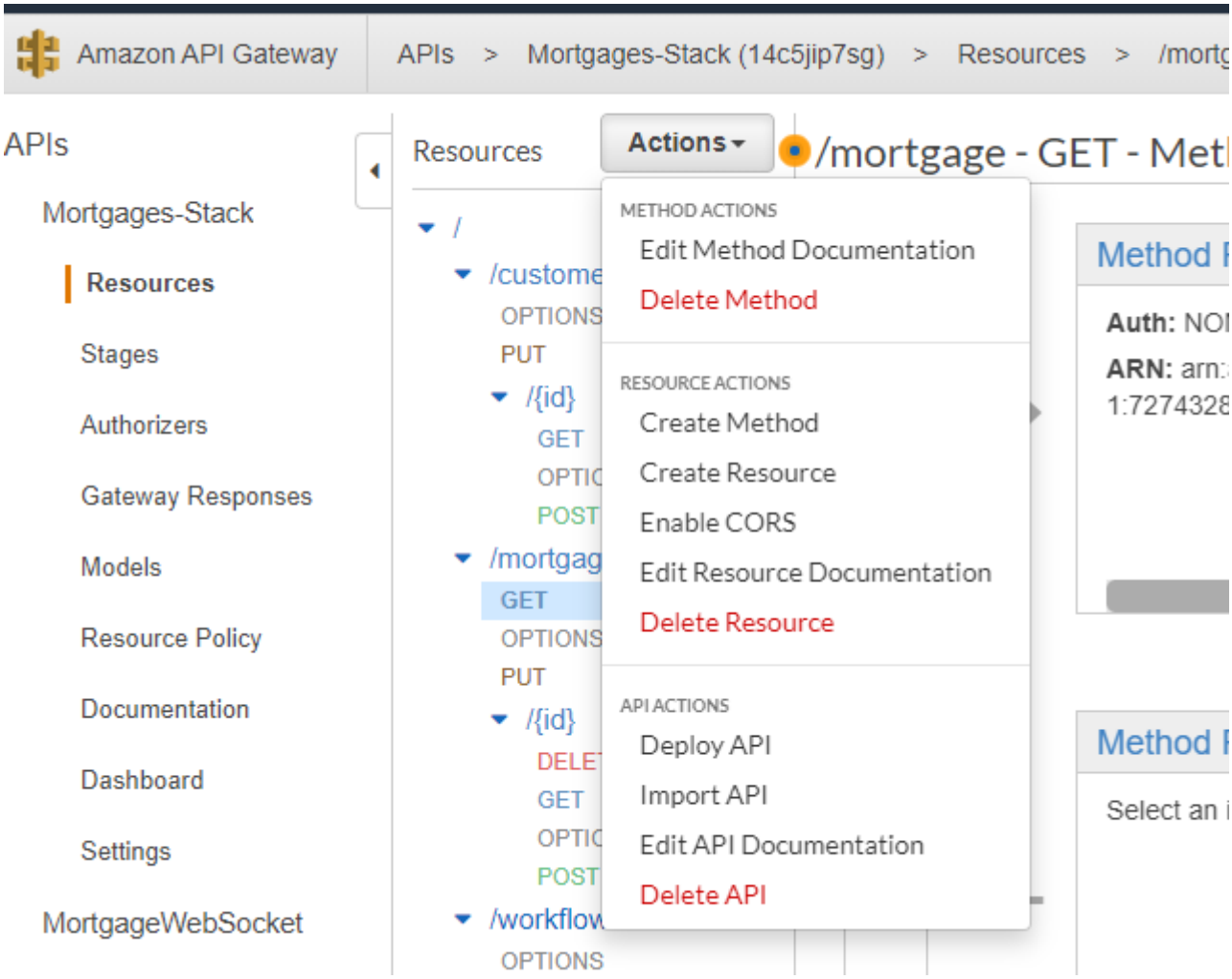


Figure AWS.CORS.2

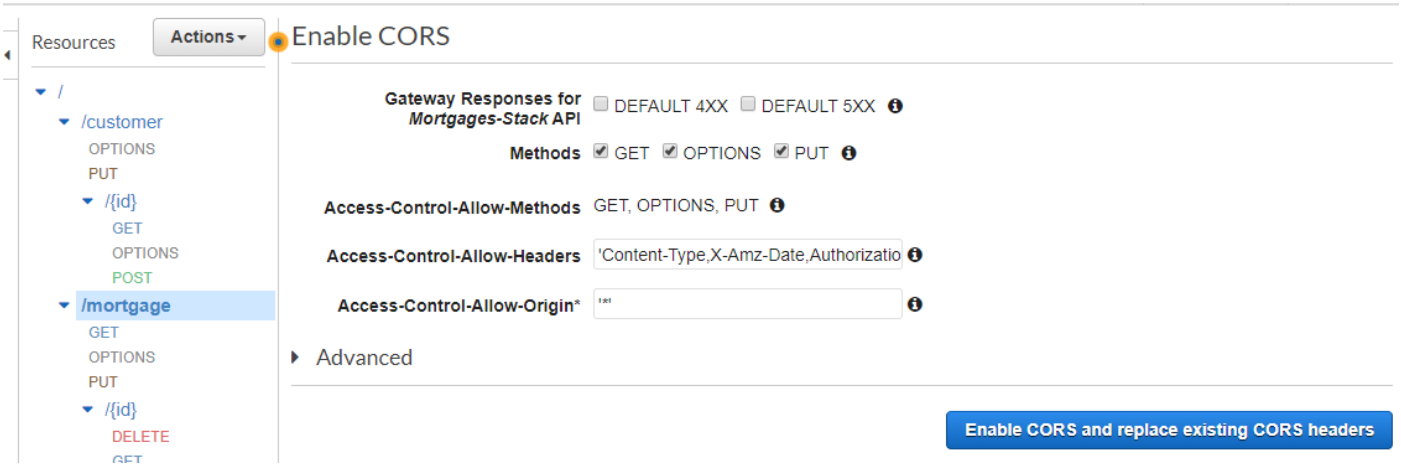


Figure AWS.CORS.3

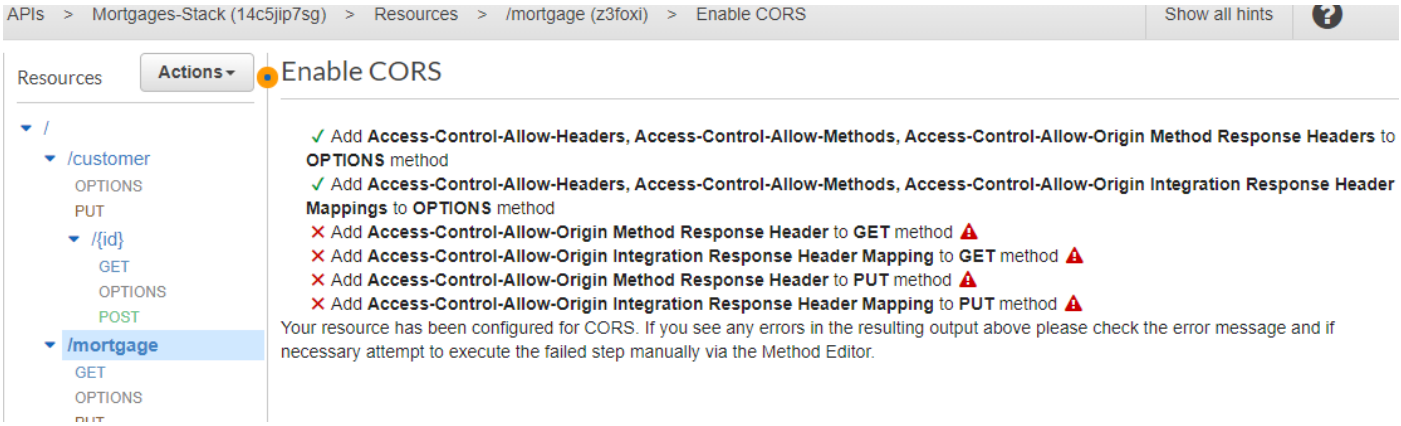


Figure AWS.CORS.4

Some sites indicated that the correct Cloudformation format to enable CORS for API Gateway as shown below.

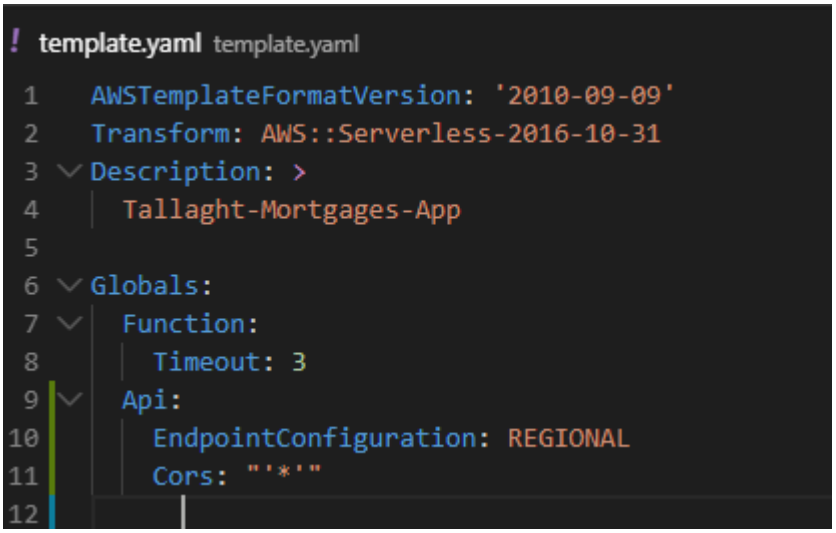


Figure AWS.CORS.5

After much web surfing and trial and error the correct template is shown below:

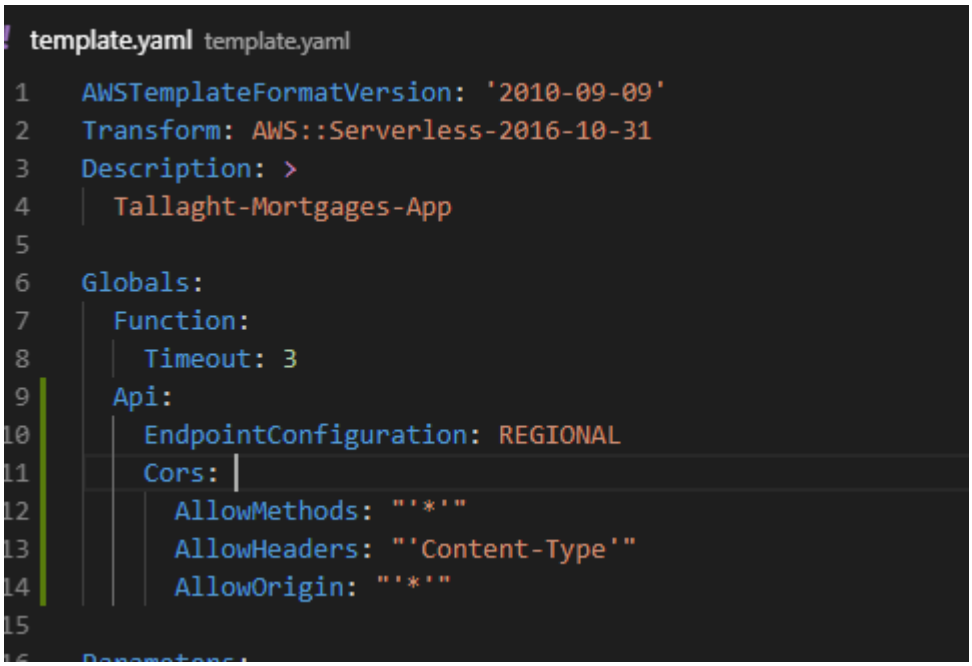


Figure AWS.CORS.6

API Gateway with Lambda integration (rather than HTTP integration) does not fully abstract out CORS functionality. Individual Lambda function must add CORS headers as seen in Figure AWS.CORS.7

```
const findMortgageResponse = await dynamoDb.scan(findAllMortgages).promise();
console.log(findMortgageResponse);
return {
  statusCode: 200,
  headers: { 'Content-Type': 'application/json', "Access-Control-Allow-Origin": "*" },
  body: JSON.stringify(findMortgageResponse.Items)
};
```

Figure AWS.CORS.7

DEVELOPMENT – WORKFLOW IMPLMENTATION – STEP FUNCTION.

The second iteration of the Step Function State Machine was shown in Figure AWS.STEP.1

The workflow has been extended only slightly this week but in the coming week I hope to have completed the workflow by adding the following steps:

1 Add Clarification, Rejection and Approval processes based on the human underwriter decision.

2 Pause the workflow when the Mortgage Application is assigned to a human underwriter.

3 Pause the workflow when the Mortgage Application is returned to the Broker for clarification.

4 Associate the workflow restart token with the mortgage Id and persist to DynamoDB.

5 Complete all of the Websocket and SNS notification paths.

6 Integrate with the Browser SPA, allow users to start and restart the workflow process.

The small enhancements that I have completed for the state machine this week have allowed me to become more familiar with the Step Function functionality and notation. It is very important to have a clear design for the input and outputs of each Task/Lambda functionmethod and the transformations that are required between Tasks to allow subsequent tasks in the workflow to access the required data.

The testing and bug fixing that I performed is summarised now in the following snapshots of two scenarios : Auto Assessment rejection and Auto Assessment success.

Auto Assessment Rejection

Submit a Mortgage Application that does not meet the Auto Assessment requirements. Figure AWS.AUTO.1

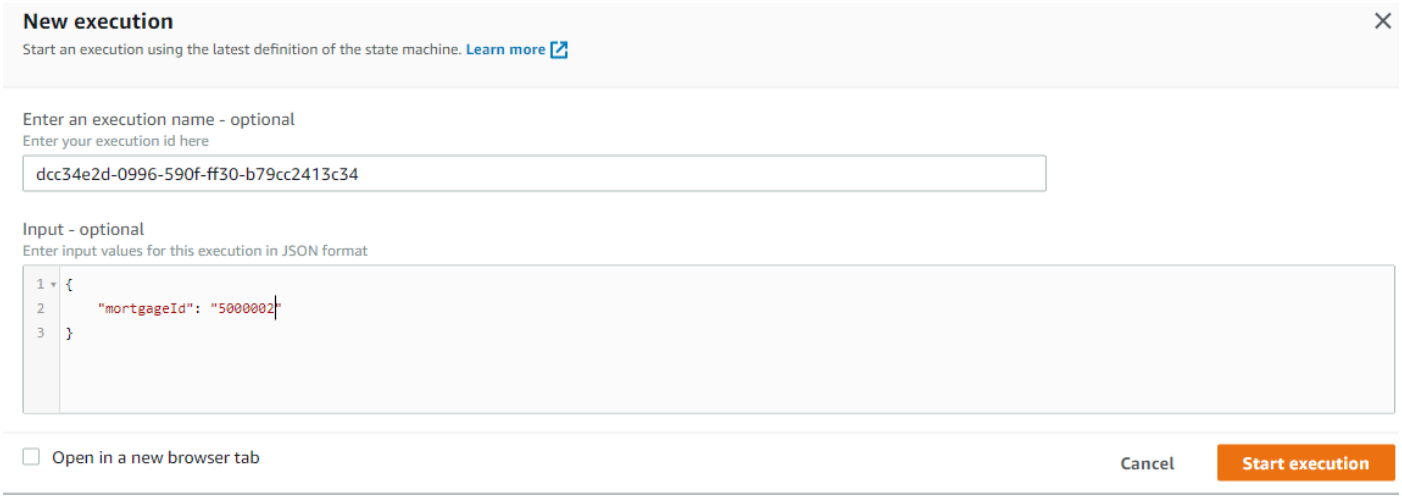
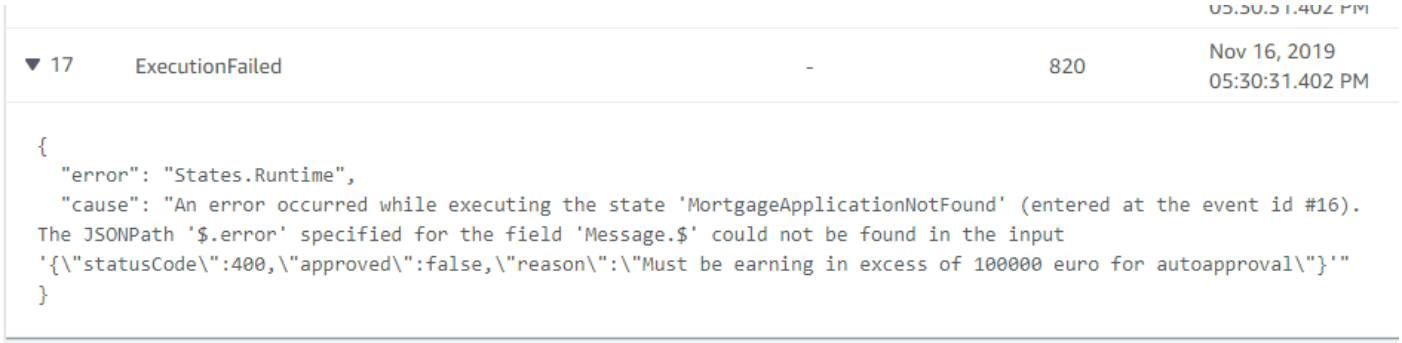


Figure AWS.AUTO.1

Invalid JSON path.



The InputPath on the Task must isolate the required JSON node to provide the correct input to the Lambda function. Figure AWS.AUTO.2

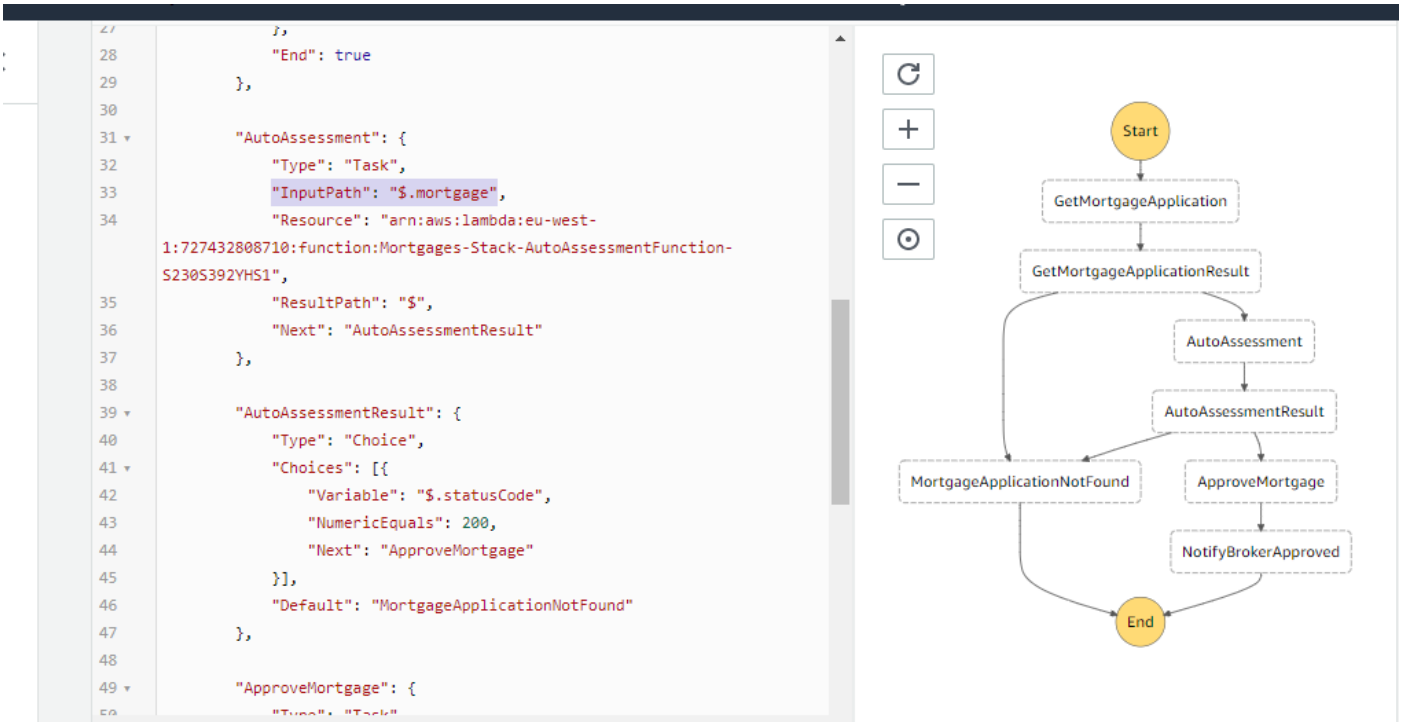


Figure AWS.AUTO.2

Also the Lambda Fucntion must return any and all data that is required for the next task in the chain. Figure AWS.AUTO.3

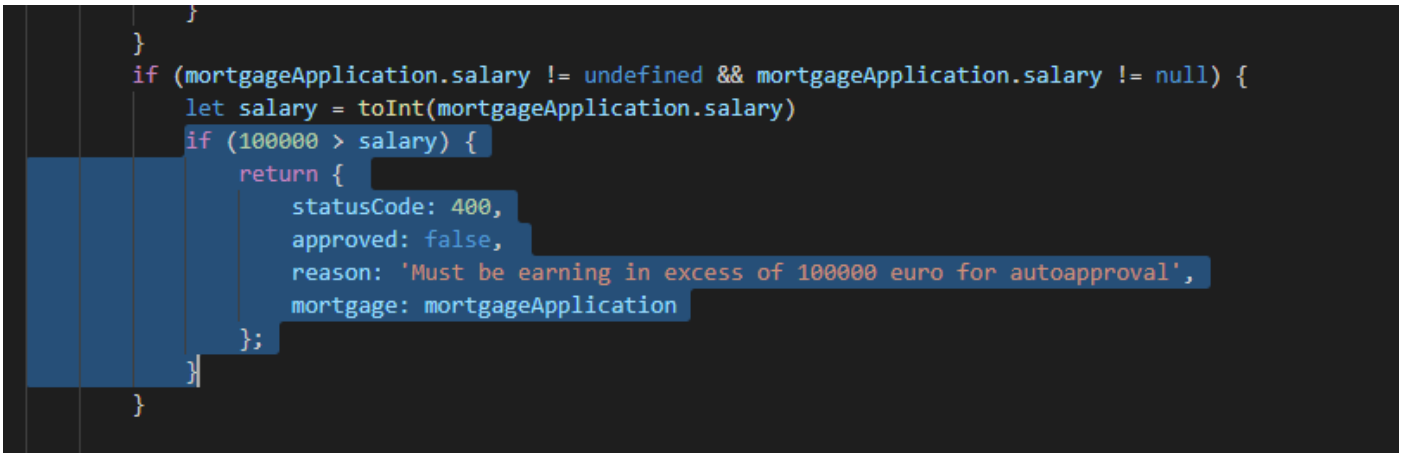


Figure AWS.AUTO.3

If the input and output data is not correctly mediated then the Step Function execution will fail. Figure AWS.AUTO.4

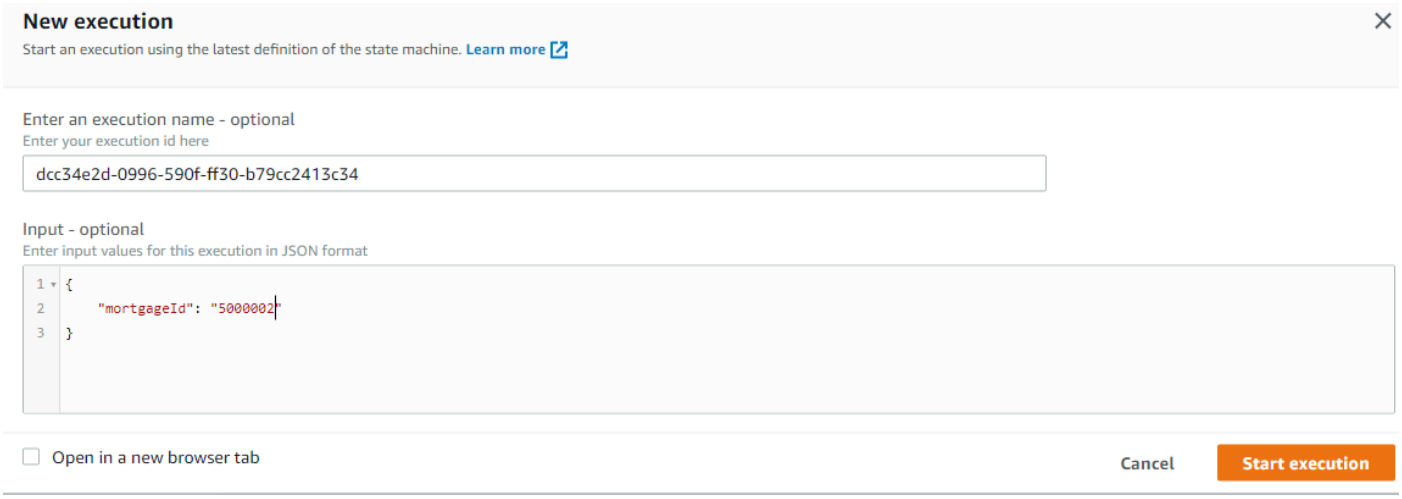


Figure AWS.AUTO.4

The AWS Console will highlight failed states as shown below. Figure AWS.AUTO.5

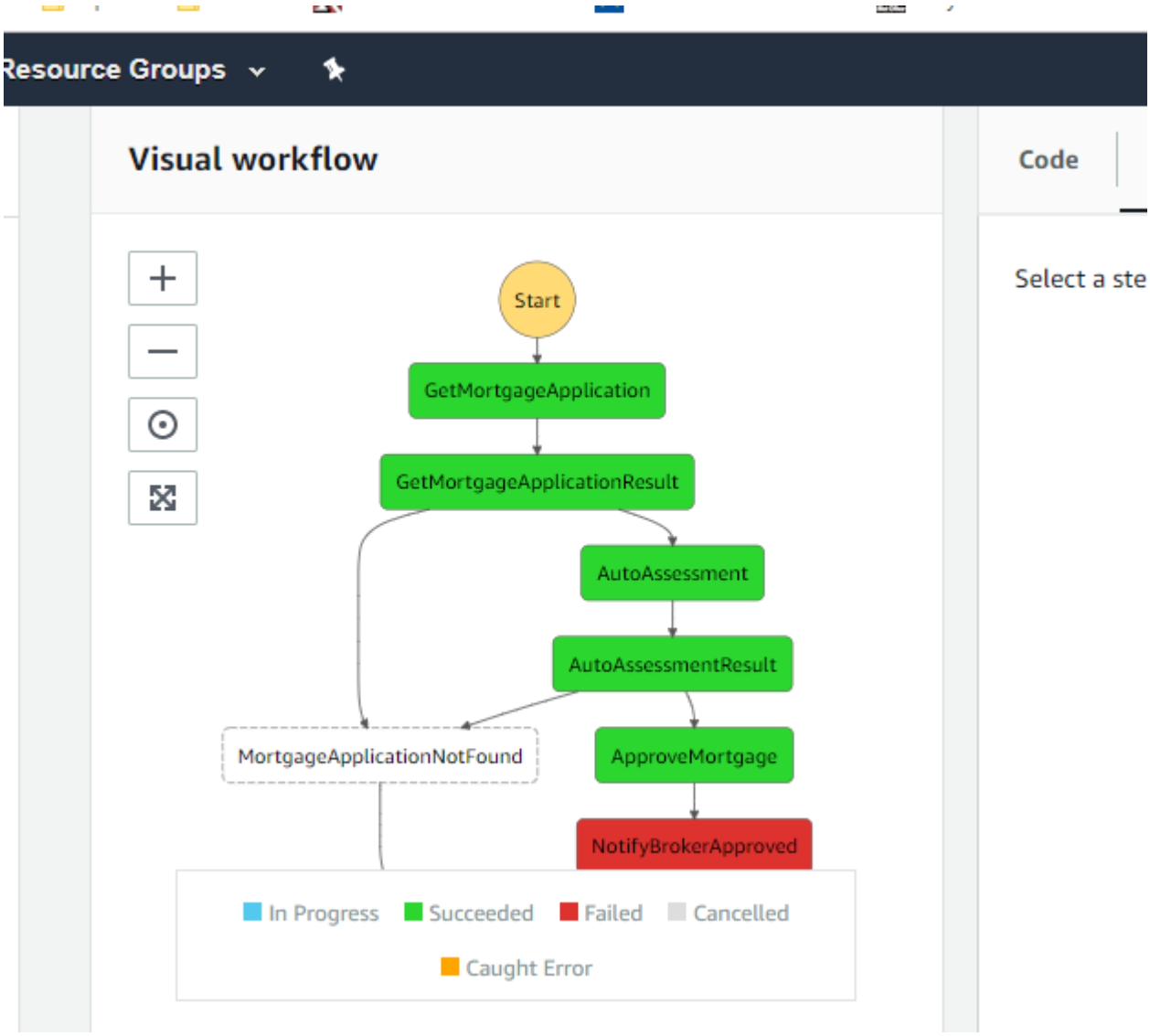


Figure AWS.AUTO.5

Another badly formatted Step Function Task input. Figure AWS.AUTO.6

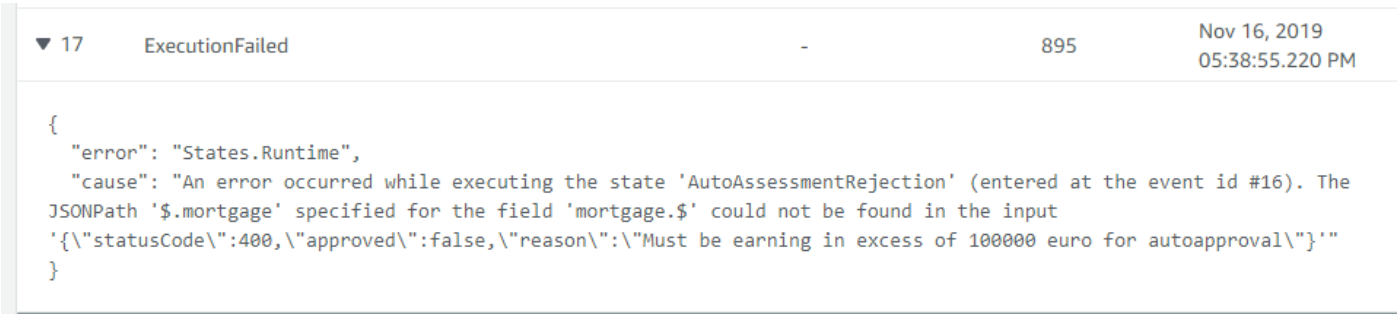


Figure AWS.AUTO.6

Once these errors are rectified then the Step Function execution will success. Figure AWS.AUTO.7

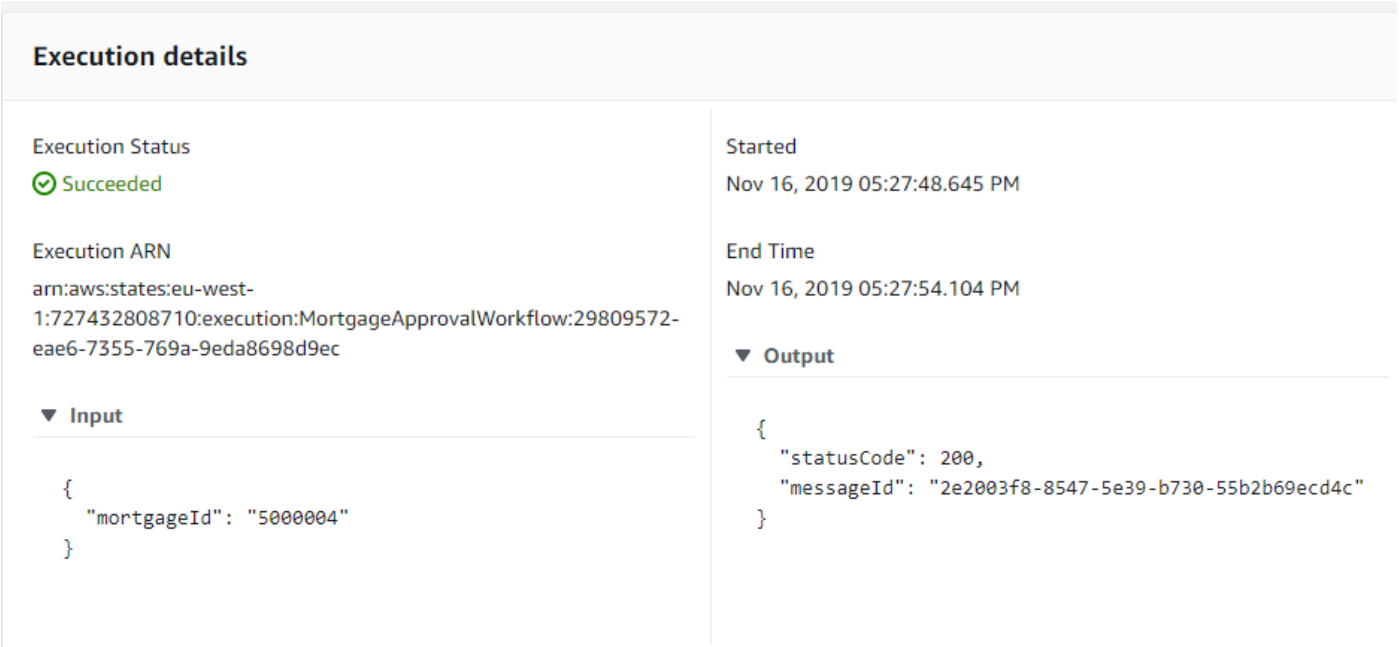
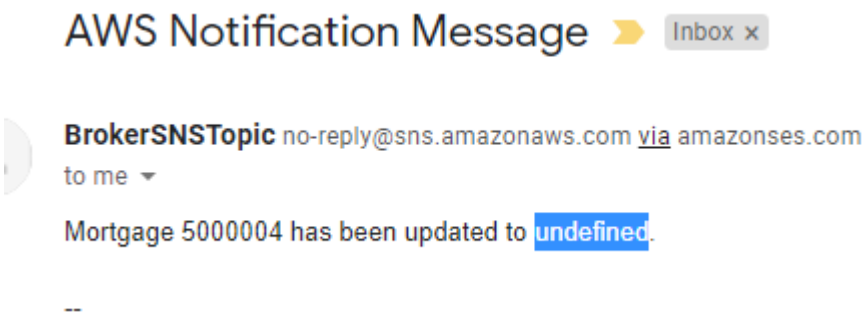


Figure AWS.AUTO.5

The Step Function output correctly shows that the last Task returned a 200 status completion code and also the ID of the SNS message that was send to the users.

However another coding error show that the new state is not correctly rendered in the email. Figure



AWS.AUTO.6

Figure AWS.AUTO.6

Once resolved we see the correct message in the email. Figure AWS.AUTO.7

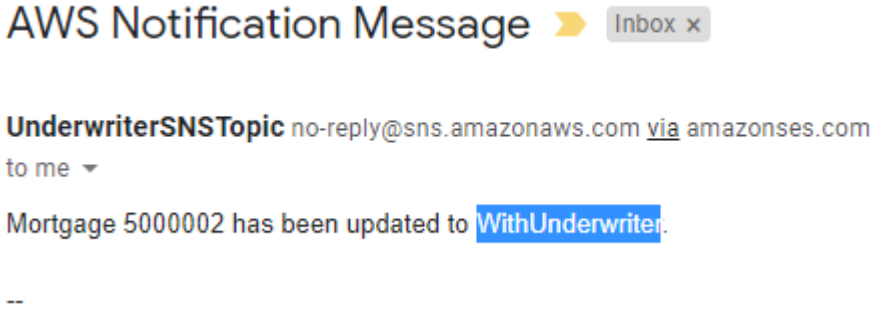


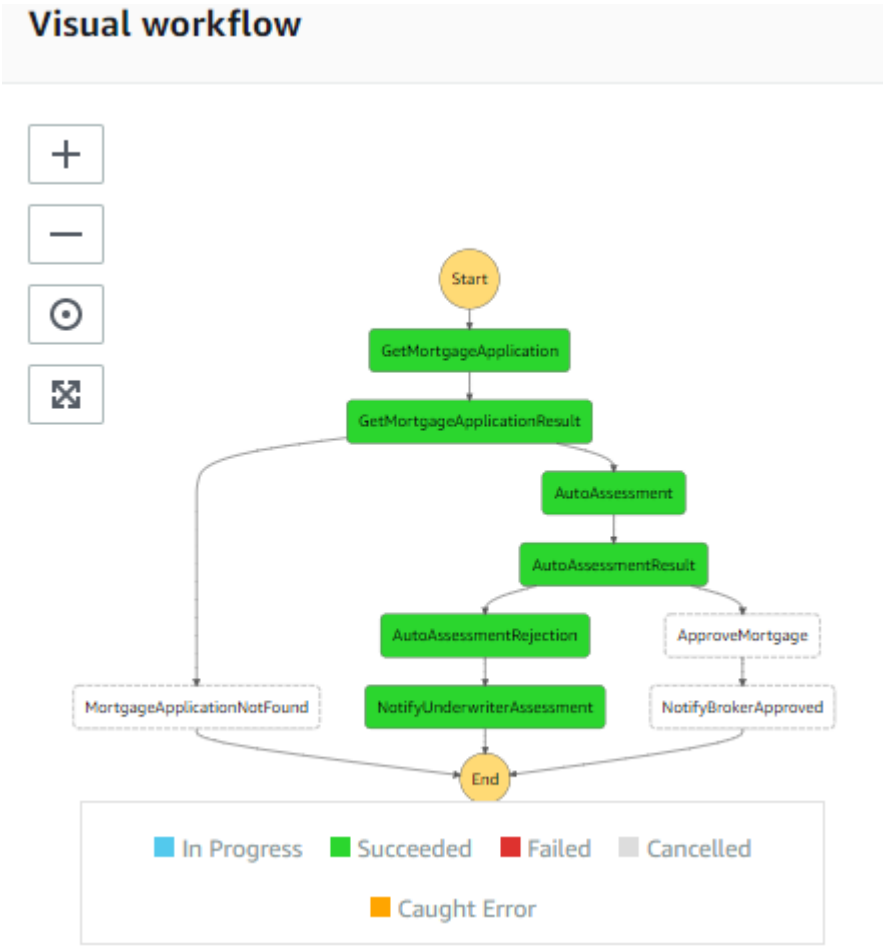
Figure AWS.AUTO.7

The mortgage application has also been correctly updated in DynamoDB. Figure AWS.AUTO.8

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary
5000002	60002	Smurfit	300009	WithUnderwriter	50009
5000003	60003	Microsoft	200000	PreSubmission	100000
5000004	60003	AIB Ltd	999999	PreSubmission	100001

Figure AWS.AUTO.8

The Step Function execution shows the sequence of tasks.



Auto Assessment Success

Select a Mortgage Application that meets ALL Auto Assessment criteria. Figure AWS.AUTO.9

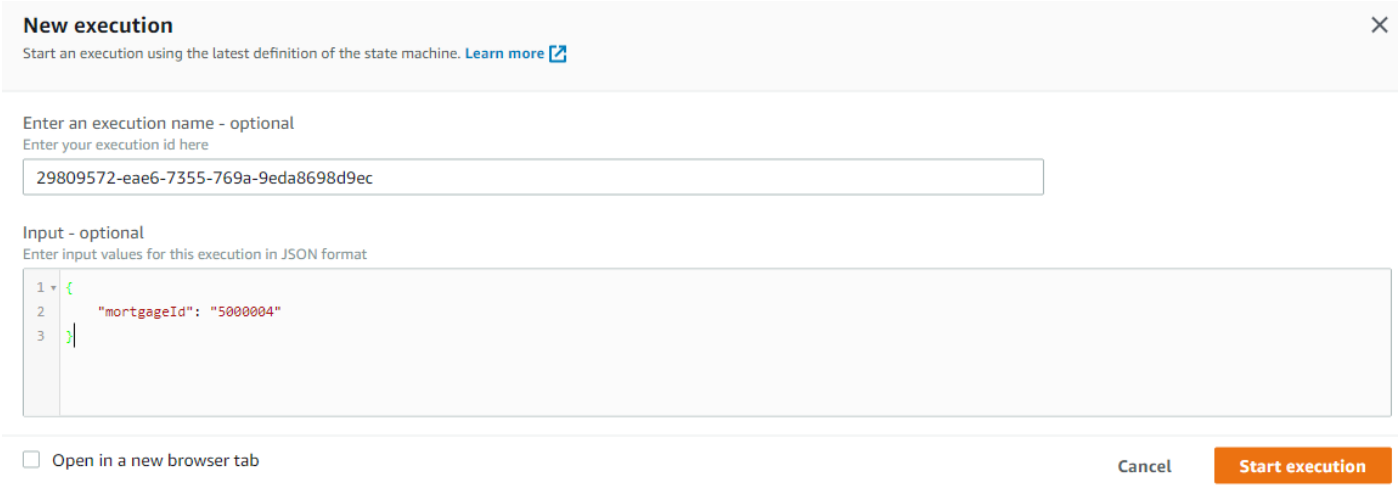


Figure AWS.AUTO.9

Successful email SNS Notification. Figure AWS.AUTO.10

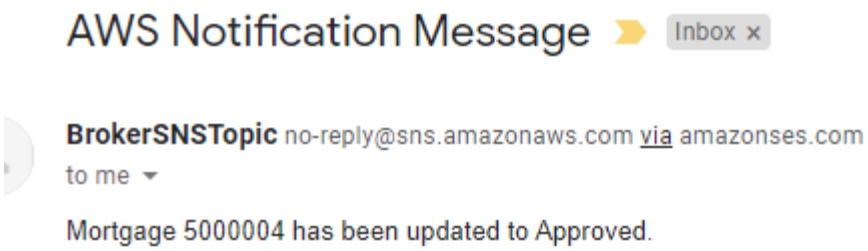


Figure AWS.AUTO.10

DynamoDB has also been updated. Figure AW.S.AUTO.11

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	ter
5000002	60002	Smurfit	300009	PreSubmission	50009	39
5000003	60003	Microsoft	200000	PreSubmission	100000	20
5000004	60003	AIB Ltd	999999	Approved	100001	98

Figure AW.S.AUTO.11

Again the Step Function execution diagram shows the sequene of events. Figure AWS.AUTO.12

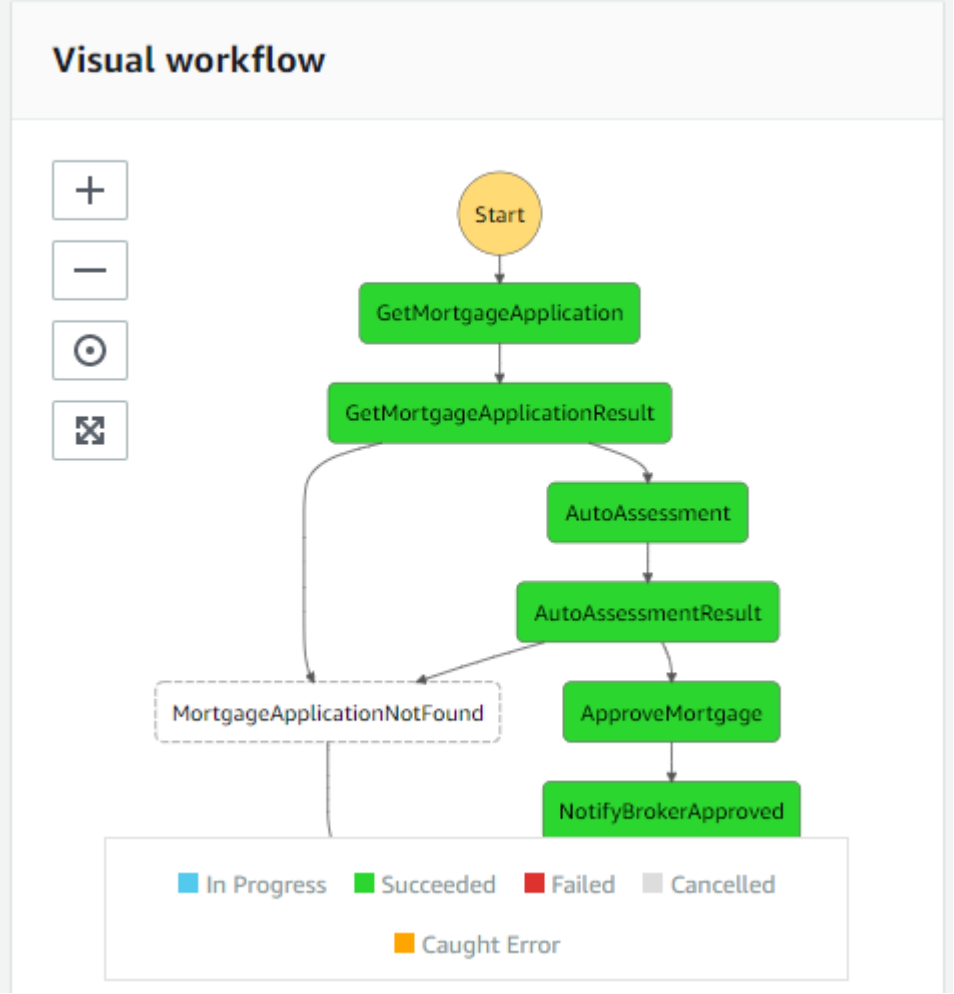


Figure AWS.AUTO.12

APPENDICES

The following sections list some key / interesting source code / template snippets.

WebSocket Cloudformation Template

```
 # WEBSOCKET APIS
 MortgageWebSocket:
  Type: AWS::ApiGatewayV2::Api
  Properties:
   Name: MortgageWebSocket
   ProtocolType: WEBSOCKET
   RouteSelectionExpression: "$request.body.message"
 ConnectRoute:
  Type: AWS::ApiGatewayV2::Route
  Properties:
   ApiId: !Ref MortgageWebSocket
   RouteKey: $connect
   AuthorizationType: NONE
   OperationName: ConnectRoute
   Target: !Join
    - '/'
    - - 'integrations'
      - !Ref MortgateConnectionIntegration
 MortgateConnectionIntegration:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
   ApiId: !Ref MortgageWebSocket
   Description: Connect Integration
   IntegrationType: AWS_PROXY
   IntegrationUri:
    Fn::Sub:
      arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketConnectFunction.Arn}/invocations
 DisconnectRoute:
  Type: AWS::ApiGatewayV2::Route
  Properties:
   ApiId: !Ref MortgageWebSocket
   RouteKey: $disconnect
   AuthorizationType: NONE
   OperationName: DisconnectRoute
```

```
  Target: !Join
    - '/'
    - - 'integrations'
      - !Ref MortgateDisconnectInteg
 MortgateDisconnectInteg:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
   ApiId: !Ref MortgageWebSocket
   Description: Disconnect Integration
   IntegrationType: AWS_PROXY
   IntegrationUri:
    Fn::Sub:
      arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketDisconnectFunction.Arn}/invocations
 SendRoute:
  Type: AWS::ApiGatewayV2::Route
  Properties:
   ApiId: !Ref MortgageWebSocket
   RouteKey: sendmessage
   AuthorizationType: NONE
   OperationName: SendRoute
   Target: !Join
    - '/'
    - - 'integrations'
      - !Ref SendInteg
 SendInteg:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
   ApiId: !Ref MortgageWebSocket
   Description: Send Integration
   IntegrationType: AWS_PROXY
   IntegrationUri:
    Fn::Sub:
      arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketSendFunction.Arn}/invocations
 Deployment:
  Type: AWS::ApiGatewayV2::Deployment
  DependsOn:
  - ConnectRoute
  - SendRoute
  - DisconnectRoute
  Properties:
   ApiId: !Ref MortgageWebSocket
 Stage:
  Type: AWS::ApiGatewayV2::Stage
```

```
  Properties:
   StageName: Prod
   Description: Prod Stage
   DeploymentId: !Ref Deployment
   ApiId: !Ref MortgageWebSocket

 # WEBSOCKET DYNAMO DB TABLE
 WebsocketConnectionsTable:
  Type: AWS::DynamoDB::Table
  Properties:
   AttributeDefinitions:
   - AttributeName: "connectionId"
    AttributeType: "S"
   KeySchema:
   - AttributeName: "connectionId"
    KeyType: "HASH"
   ProvisionedThroughput:
    ReadCapacityUnits: 5
    WriteCapacityUnits: 5
   SSESpecification:
    SSEEnabled: True
   TableName: !Ref WebsocketConnectionTableName

 # WEBSOCKET LAMBDA API
 WebSocketConnectFunction:
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: websockets/onconnect/
   Handler: app.handler
   MemorySize: 256
   Runtime: nodejs10.x
   Environment:
    Variables:
     TABLE_NAME: !Ref WebsocketConnectionTableName
   Policies:
   - DynamoDBCrudPolicy:
     TableName: !Ref WebsocketConnectionTableName
 OnConnectPermission:
  Type: AWS::Lambda::Permission
  DependsOn:
   - MortgageWebSocket
   - WebSocketConnectFunction
  Properties:
   Action: lambda:InvokeFunction
```

```
    FunctionName: !Ref WebSocketConnectFunction
    Principal: apigateway.amazonaws.com
  WebSocketDisconnectFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: websockets/ondisconnect/
      Handler: app.handler
      MemorySize: 256
      Runtime: nodejs10.x
      Environment:
        Variables:
          TABLE_NAME: !Ref WebsocketConnectionTableName
      Policies:
      - DynamoDBCrudPolicy:
          TableName: !Ref WebsocketConnectionTableName
  OnDisconnectPermission:
    Type: AWS::Lambda::Permission
    DependsOn:
      - MortgageWebSocket
      - WebSocketDisconnectFunction
    Properties:
      Action: lambda:InvokeFunction
      FunctionName: !Ref WebSocketDisconnectFunction
      Principal: apigateway.amazonaws.com
  WebSocketSendFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: websockets/sendmessage/
      Handler: app.handler
      MemorySize: 256
      Runtime: nodejs10.x
      Environment:
        Variables:
          TABLE_NAME: !Ref WebsocketConnectionTableName
          WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-
api.',!Ref 'AWS::Region','.amazonaws.com/Prod'] ]
      Policies:
      - DynamoDBCrudPolicy:
          TableName: !Ref WebsocketConnectionTableName
      - Statement:
        - Effect: Allow
          Action:
          - 'execute-api:ManageConnections'
          Resource:
```

```
    - !Sub 'arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}:${MortgageWebSocket}/*'
 WebSocketSendPermission:
  Type: AWS::Lambda::Permission
  DependsOn:
   - MortgageWebSocket
   - WebSocketSendFunction
  Properties:
   Action: lambda:InvokeFunction
   FunctionName: !Ref WebSocketSendFunction
   Principal: apigateway.amazonaws.com
```

REFERENCE LIST AND BIBLIOGRAPHY

[1] AWS Free Tier. https://aws.amazon.com/free/

[2] AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html

[3] AWS CLI SAM. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html

[4] Node JS. https://nodejs.org/en/download/

[5] Configure AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

[6] Visual Studio Code. https://code.visualstudio.com/download

[7] Configure AWS on Visual Studio Code. https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html

[8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/

[9] DynamoDB Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml

[10] Lambda Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml

[11] Step Function State Machine Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt

[12] https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Implementation Iteration 2.1
and
Implementation Iteration 2.2 (Page 28)

Student Name: Alan Duffin

Student ID: X00159409

SUBMISSION MATERIAL

PLEASE visit the video for demostration of progress.

https://s3-eu-west-1.amazonaws.com/appd.cloud.project/IT-Tallaght-Mortgage-Workflow-WebApp-2019-11-23+13-54-23.mp4

Source Code Repository Details

Source Code NodeJS / Lambda	https://github.com/GeneralYeager/tallaght.mortgages/tree/master/tud.mortgages/Tallaght-Mortgages-App
Workflow Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt
DynamoDB Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml
Lambda / API Gateway Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml
Angular SPA Source Code	https://github.com/GeneralYeager/tallaght.mortgages/tree/master/mort-app-web

Table A.

ITERATION 2.1 TASKS

Building upon last weeks progress report, this document outlines the additional project tasks that have been completed for implementation phase 2. The follow tasks were completed this week.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Websockets	Human interaction requires notification with User Interface.
Build / Development	AWS Lambda	Build Lambda functions to manage API Gateway Websockets and notifications.
Build / Development	DynamoDB	Build table to manage API Websocket connection tokens.
Build / Development	Angular SPA	Build Browser based Front End to allow users to create/modify/submit Mortgage Applications for workflow assessment.
Test		Test Websocket Notification Events and subscriptions
Build / Development	AWS Lambda	Additiona Lambda functions to support Step Function execution.
Build / Development	AWS Step Functions	Extend Step Function state machine to support Human decision making.
Design	AWS Step Functions	Integrate Step Function generation into Cloudformation template.
Test	AWS Step Functions	Test State Transitions / Events

Table B.

I have extended the suite of Lambda services that will allow a the Step function state machine to transition Mortgage application states and allow notification to different user categories (Brokers and Underwriters). In order to support this I have had to build an intial Browser based Single Page Application using the Javascript Angular framework. At this point Brokers can create and amend mortgage applications. I have built Angular services that connect to the AWS API Gateway Websocket API that allows traffic to be initiated either by the client or the server. This is necessary so that uses can be notified of Mortgage application state changes. So users will be notified of State changes and required tasks via two channels SNS (email) and real time browser based websockets.

It has been necessary to build additional Lambda functions to support the integration between the Step Function State Machine and the API Gateway Websocket layer. Also some additional Lambda functions were developed specifically to transition the Mortgage Applications between states.

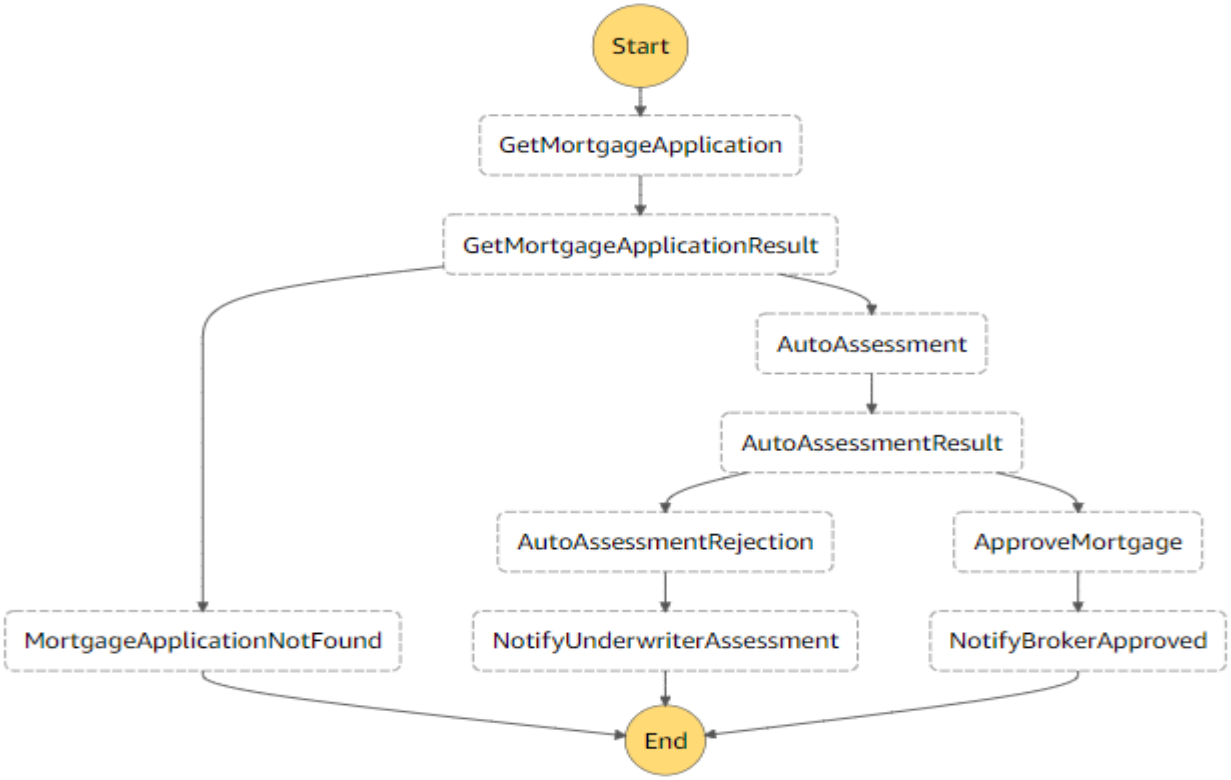
PLAN FOR ITERATION 2.2

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Complete the Step Function Workflow state machine.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Build / Development	AWS Lamda	Build the Lambda Functions to support pausing the workflow to await human decisions. Also manage the Step Function tokens.
Build / Development	DynamoDB	Implement new DB schema to map mortgage application to Step Function execution id. This is required to allow users approve or reject applications without reference to the step function execution id.
Build / Development	Angular	Extends the Browser application to allow submission of mortgages for assessment. Allow for clarification requests and Underwriter decision making.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table C.

The workflow state machine currently exists as seen in Figure AWS.STEP.1

Figure AWS.STEP.1


Start
GetMortgageApplication
GetMortgageApplicationResult
AutoAssessment
AutoAssessmentResult
AutoAssessmentRejection
ApproveMortgage
MortgageApplicationNotFound
NotifyUnderwriterAssessment
NotifyBrokerApproved
End

SERVLERLESS / LAMBDA FUNCTIONS

The following Lambda functions have now been completed as of this progress report. Figure AWS.LAMBDA.1

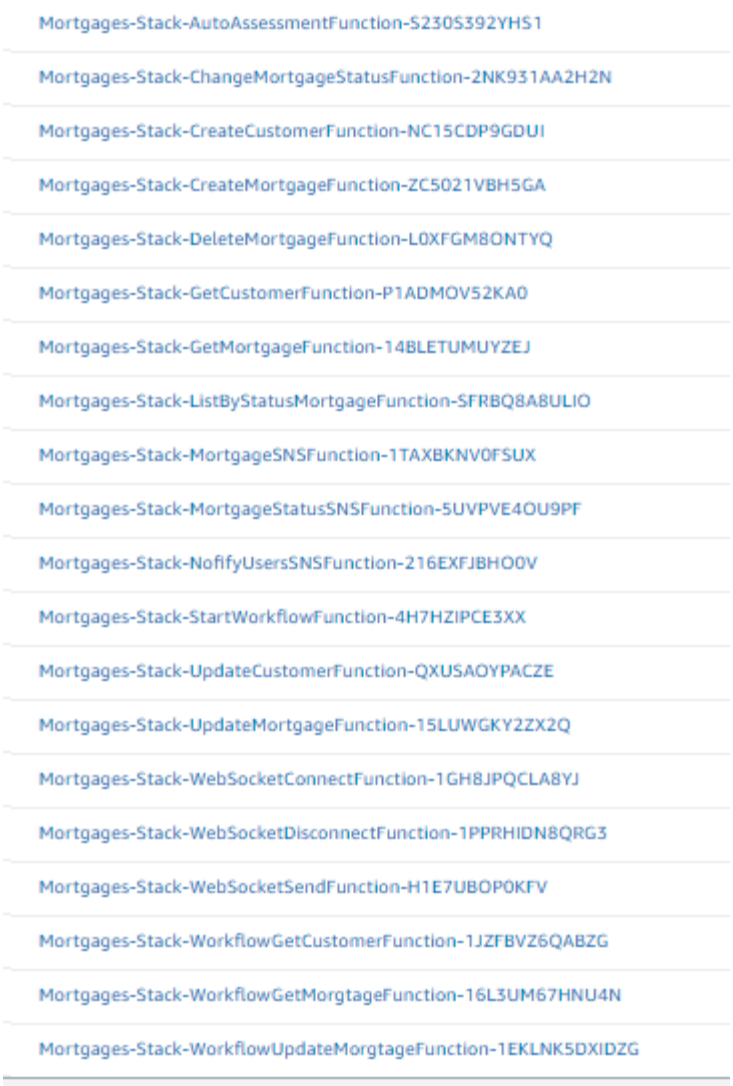


Figure AWS.LAMBDA.1

This table lists the purpose of each of the completed Lambda Functions. Table AWS.LAMBDA.1

Lambda Function	Purpose
Mortgages-Stack-AutoAssessmentFunction	Perform Mortgage Rules check
Mortgages-Stack-ChangeMortgageStatusFunction	Modify the Mortgage Application Status and store.
Mortgages-Stack-CreateCustomerFunction	HTTP API Create Customer
Mortgages-Stack-CreateMortgageFunction	HTTP API Create Mortgage Application
Mortgages-Stack-DeleteMortgageFunction	HTTP API Delete Mortgage Application
Mortgages-Stack-GetCustomerFunction	HTTP API Find By PK Customer
Mortgages-Stack-GetMortgageFunction	HTTP API Find By PK Mortgage Application
Mortgages-Stack-ListByStatusMortgageFunction	HTTP API Find By Status Mortgage Application
Mortgages-Stack-MortgageStatusSNSFunction	Step Function Update Mortgage Status
Mortgages-Stack-NofifyUsersSNSFunction	Step Function Notify State Change
Mortgages-Stack-StartWorkflowFunction	HTTP API Execute Mortgage Workflow
Mortgages-Stack-UpdateCustomerFunction	HTTP API Update Customer
Mortgages-Stack-UpdateMortgageFunction	HTTP API Update Mortgage
Mortgages-Stack-WebSocketConnectFunction	Manage Websocket connection and store token.
Mortgages-Stack-WebSocketDisconnectFunction	Manage Websocket disconnect and delete token.
Mortgages-Stack-WebSocketSendFunction	Step Function send message to API Gateway Websocket for Browser broadcast.
Mortgages-Stack-WorkflowGetCustomerFunction	Step Function Find By PK Customer
Mortgages-Stack-WorkflowGetMorgtageFunction	Step Function Find By PK Mortgage
Mortgages-Stack-WorkflowUpdateMorgtageFunction	Step Function Update Mortgage

Table AWS.LAMBDA.1

The source code and deployment template for these functions are available at the GitHub repository listed in Table A or in the Appendix.

I have had to implement new Lambda functions to manage API Gateway Websocket functionality.

API GATEWAY WEBSOCKETS

Last year AWS announced API Gateway support for the Websocket protocol [12]. This allows for the creation of ...

"bidirectional communication applications using WebSocket APIs in Amazon API Gateway without having to provision and manage any servers.

HTTP-based APIs use a request/response model with a client sending a request to a service and the service responding synchronously back to the client. WebSocket-based APIs are bidirectional in nature. This means that a client can send messages to a service and services can independently send messages to its clients.

This bidirectional behavior allows for richer types of client/service interactions because services can push data to clients without a client needing to make an explicit request. WebSocket APIs are often used in real-time applications such as chat applications, collaboration platforms, multiplayer games, and financial trading platforms."

The following API Gateway Websocket API has been created. Figure AWS.API.1

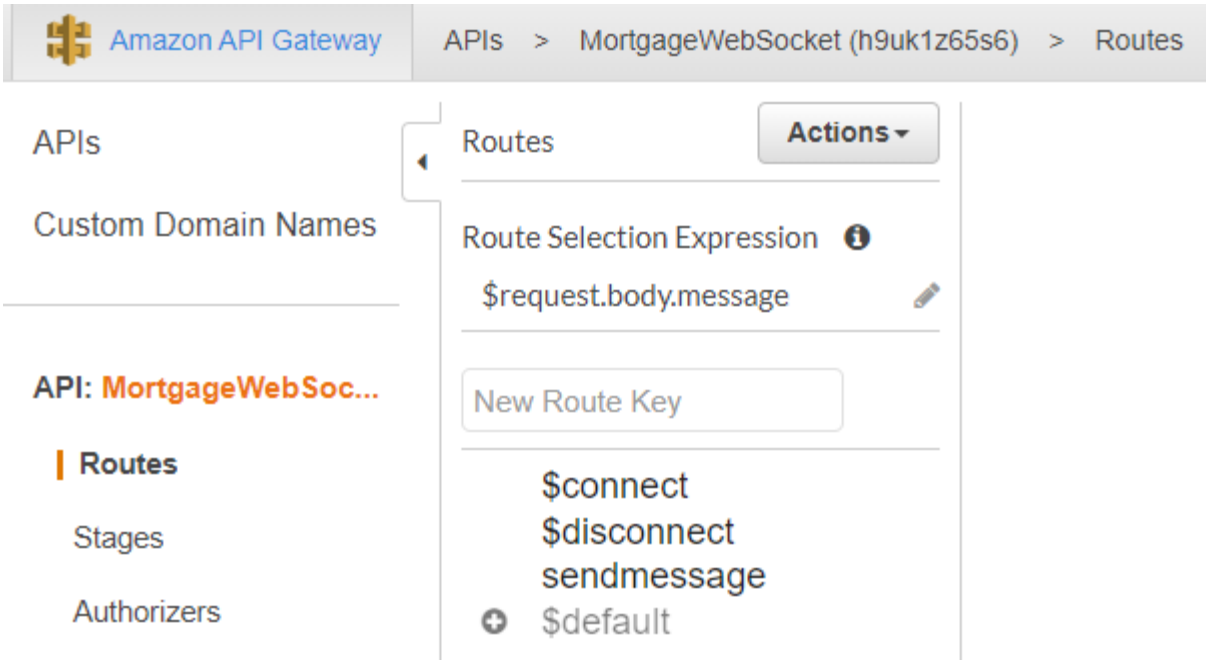


Figure AWS.API.1

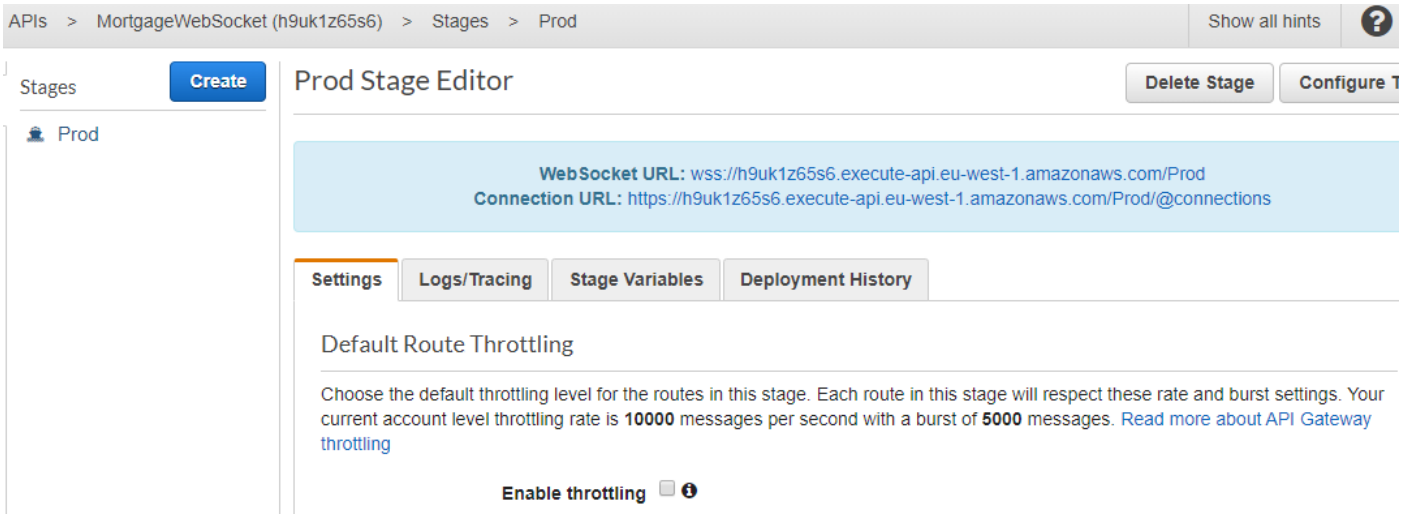


Figure AWS.API.2

Because API Gateway support for Websockets is very recent, the online help is minimal. It took many hours to get the deployment and develoment correct.

There were many errors encountered such as incorrect endpoint format and futile attempts to get Lambda functions to establish wss protocol connections.

This error was caused by an invalid endpoint format.

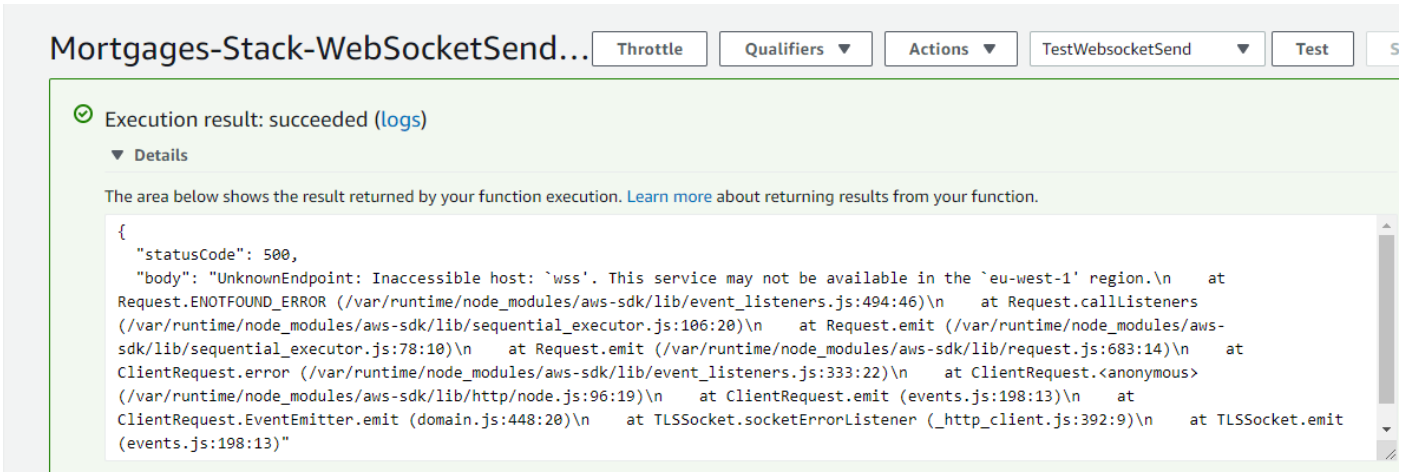


Figure AWS.SOCKET.1

It is not possible for Lambda functions to connect to an external WSS endpoint. They must directly use the 'AWS.ApiGatewayManagementApi' sdk and connect via the API Gateway instance.

```
395     Runtime: nodejs10.x
396     Environment:
397       Variables:
398         TABLE_NAME: !Ref WebsocketConnectionTableName
399       # WEBSOCKET_ENDPOINT: !Join [ '', [ 'wss://', !Ref MortgageWebSocket, '.execute-api.',!Ref 'A
400         WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-api.',!Ref 'AWS::Regior
401     Policies:
```

Figure AWS.SOCKET.2

Websocket Success. Figure AWS.SOCKET.3

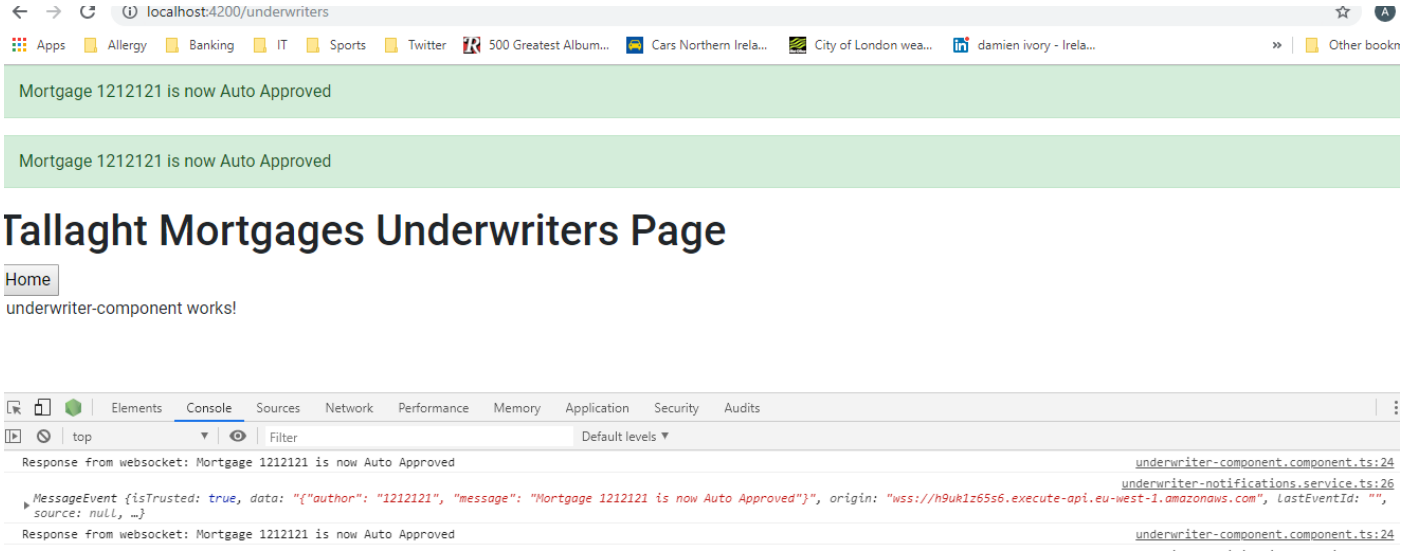


Figure AWS.SOCKET.3

Socket Broadcast Source Code:

```
exports.handler = async (event, context) => {
 let connectionData;
 console.log(event);
 try {
  connectionData = await ddb.scan({ TableName: TABLE_NAME, ProjectionExpression:
 'connectionId' }).promise();
 } catch (e) {
  return { statusCode: 500, body: e.stack };
 }

 const apigwManagementApi = new AWS.ApiGatewayManagementApi({
  apiVersion: '2018-11-29',
  endpoint: WEBSOCKET_ENDPOINT //event.requestContext.domainName + '/' + event.r
equestContext.stage
 });

 //const postData = JSON.parse(event.body).data;
 const postData = '{\"author\": \"1212121", \"message\": \"Mortgage 1212121 is
now Auto Approved"}';


 const postCalls = connectionData.Items.map(async ({ connectionId }) => {
  try {
   await apigwManagementApi.postToConnection({ ConnectionId: connectionId, Data:
 postData }).promise();
  } catch (e) {
   if (e.statusCode === 410) {
    console.log(`Found stale connection, deleting ${connectionId}`);
    await ddb.delete({ TableName: TABLE_NAME, Key: { connectionId } }).promise()
;
   } else {
    throw e;
   }
  }
 });

 try {
  await Promise.all(postCalls);
 } catch (e) {
  return { statusCode: 500, body: e.stack };
 }
```

```
 return { statusCode: 200, body: 'Data sent.' };
};
```

Web Browser Websocket Connection Source Code:

```
import { Injectable } from '@angular/core';

import { Observable, Observer, Subject } from 'rxjs'

@Injectable({
 providedIn: 'root'
})
export class WebsocketService {
 constructor() {}

 private subject: Subject<MessageEvent>;

 public connect(url): Subject<MessageEvent> {
  console.log("connect url");
  if (!this.subject) {
   this.subject = this.create(url);
   console.log("Successfully connected: " + url);
  }
  return this.subject;
 }

 private create(url): Subject<MessageEvent> {
  let ws = new WebSocket(url);
  console.log("create ws");
  let observable = Observable.create((obs: Observer<MessageEvent>) => {
   console.log("in obs");
   //ws.onmessage = obs.next.bind(obs);
   ws.onmessage = function(event) {
    console.log("WebSocket message received:", event);
    obs.next(event);
   };
   ws.onerror = obs.error.bind(obs);
   ws.onclose = obs.complete.bind(obs);
   return ws.close.bind(ws);
  });
  let observer = {
   next: (data: Object) => {
    console.log("in next");
    if (ws.readyState === WebSocket.OPEN) {
     ws.send(JSON.stringify(data));
```

```
    }
  }
 };
 return Subject.create(observer, observable);
}
}
```

ANGULAR SINGLE PAGE APPLICATION

In order to build a Javascript based Bowser application to demonstrate and interact with the Step Function Workflow I have built a first iteration of an Angular SPA application.

Steps to set up the development environment are:

1 Install npm. https://nodejs.org/en/

2 Install Angular CLI. https://angular.io/guide/setup-local

3 Install Angular Material Design. https://material.angular.io/guide/getting-started

4 Install Websocket NPM Package. https://www.npmjs.com/package/websocket

5 The source code completed so far is stored in Git Hub.

At the moment the SPA allows users to create / modify / list mortgage application. View Websocket notifications.

The look and feel is very basic but will be sufficient to provide a working system.

Run the SPA locally – shown below.

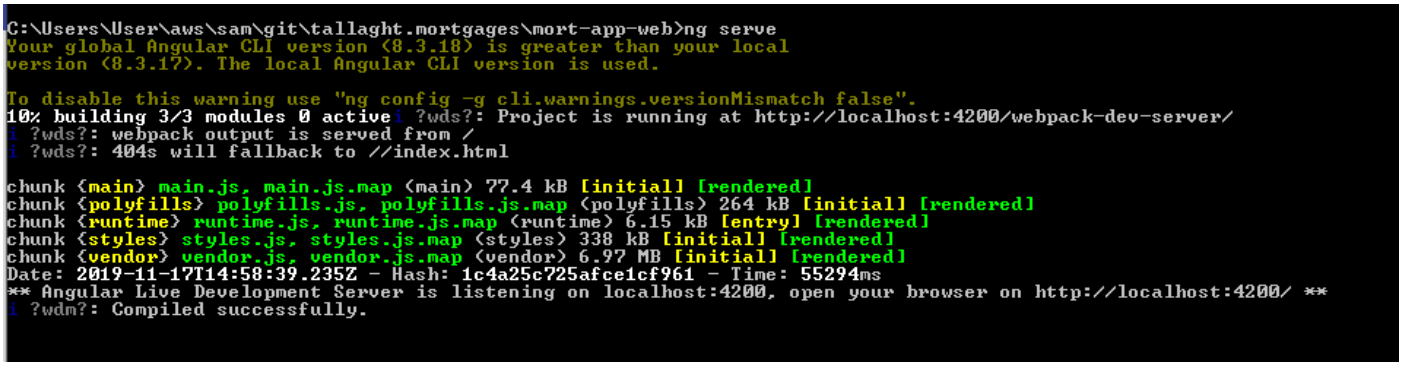
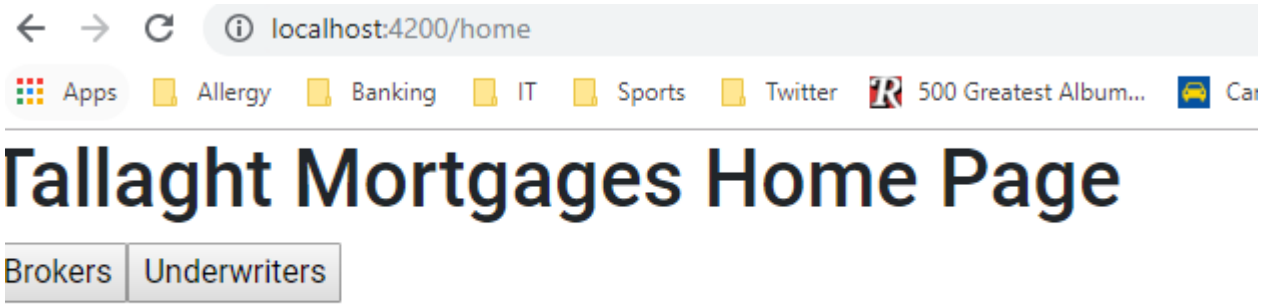
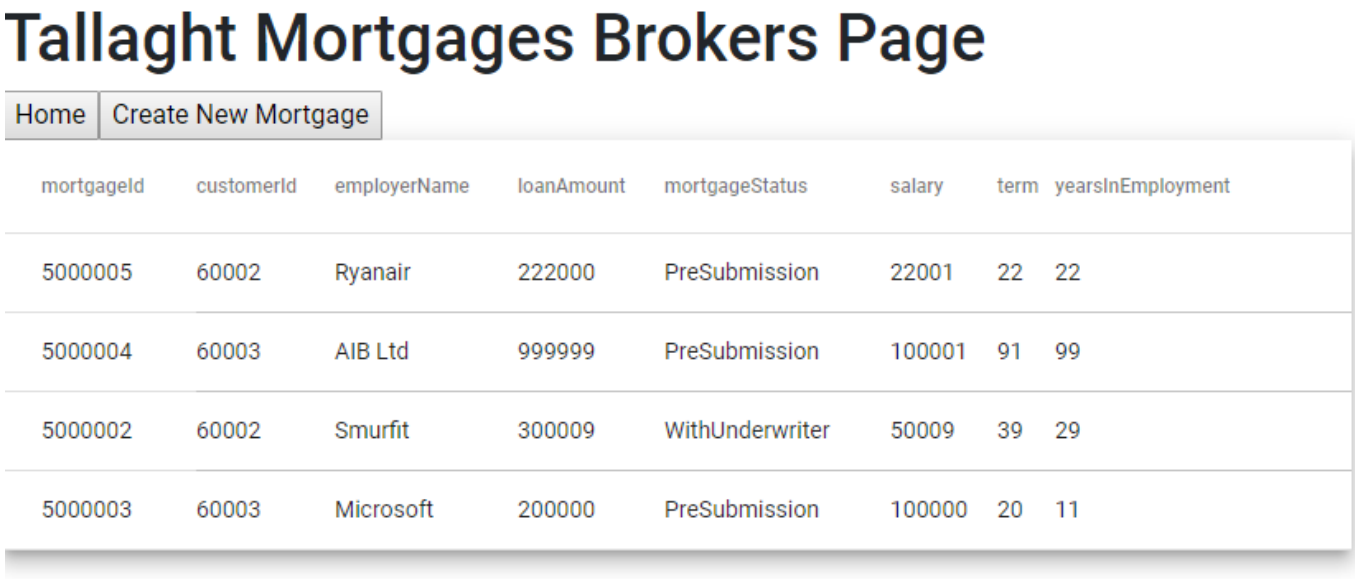


Figure AWS.ANGULAR.1

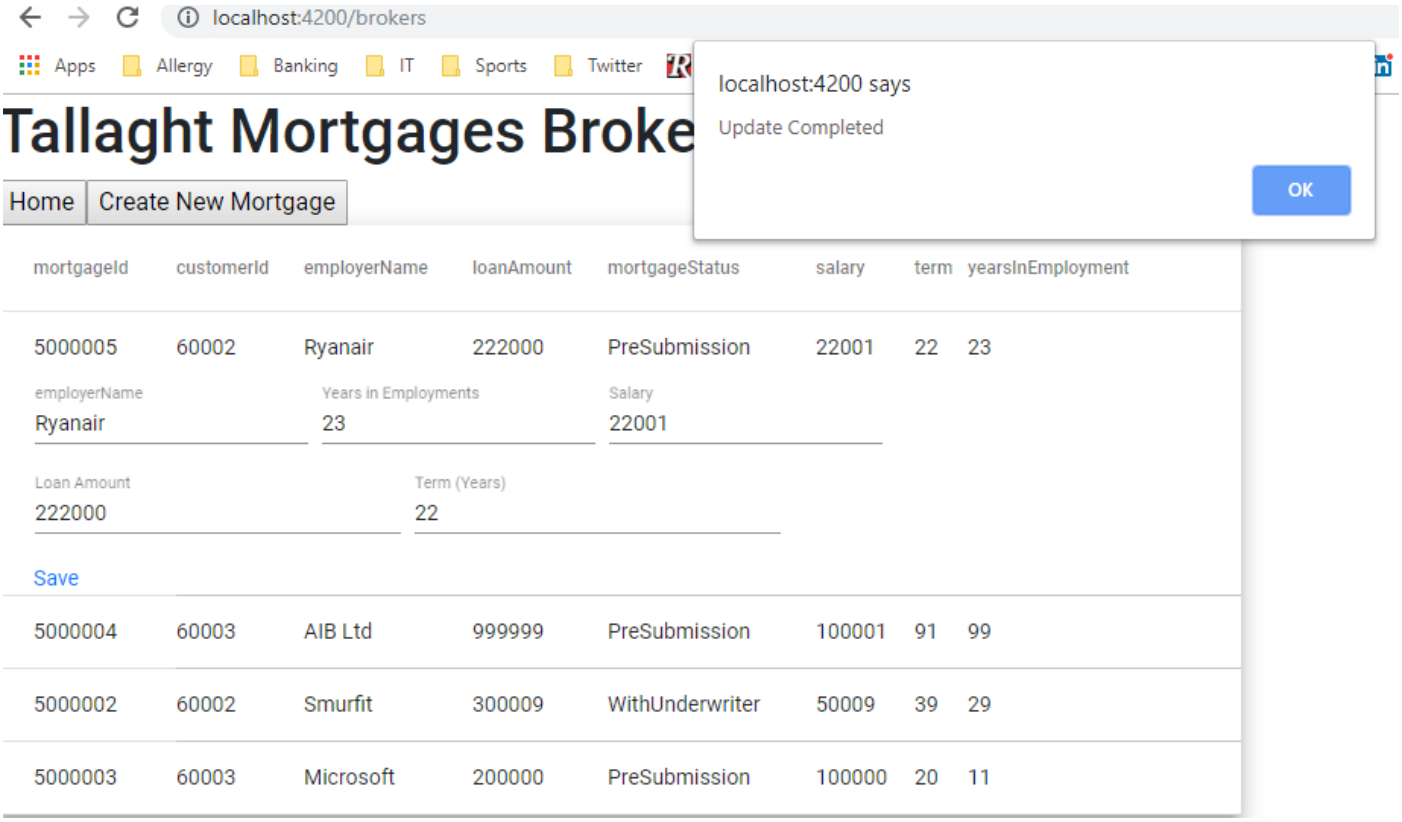
Home Page. Figure AWS.ANGULAR.2



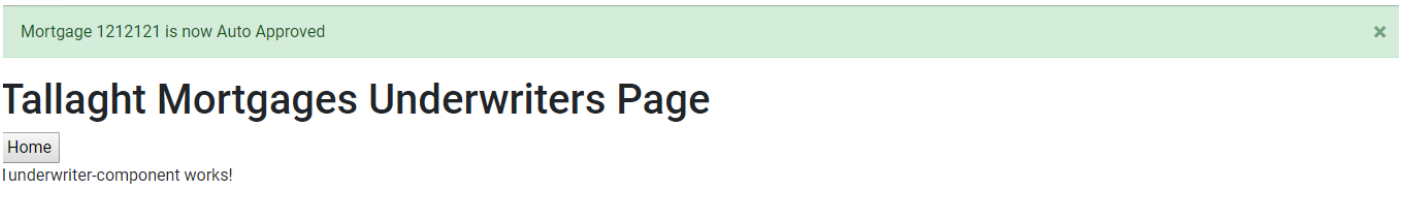


Broker Page. Figure AWS.ANGULAR.3

Modify Morgages. Figure AWS.ANGULAR.4



Recieve Mortgage Status Notifications. Figure AWS.LAMBDA.5



Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a **cross-origin HTTP request** when it requests a resource that has a different origin (domain, protocol, or port) from its own.

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

The AWS documentation enabling CORS on API Gateway via Cloudformation is lacking. It took a couple of days for me to get the HTTP API calls working from my local development environment.

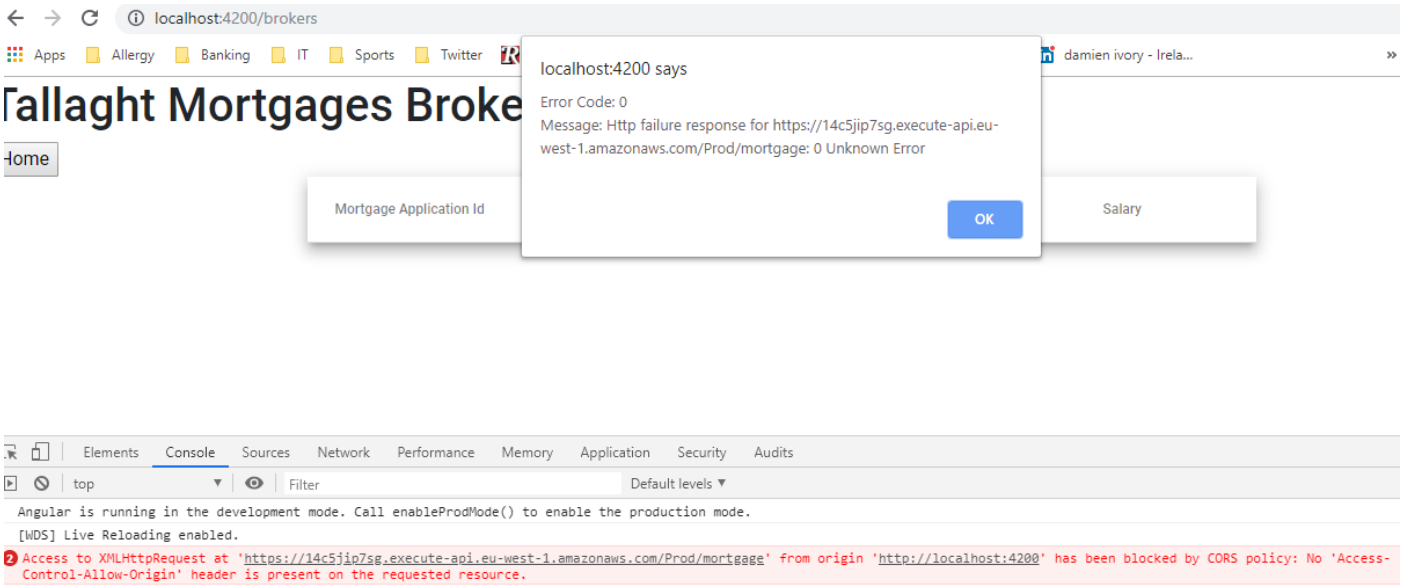


Figure AWS.CORS.1

Attempt to enable CORS post deployment.

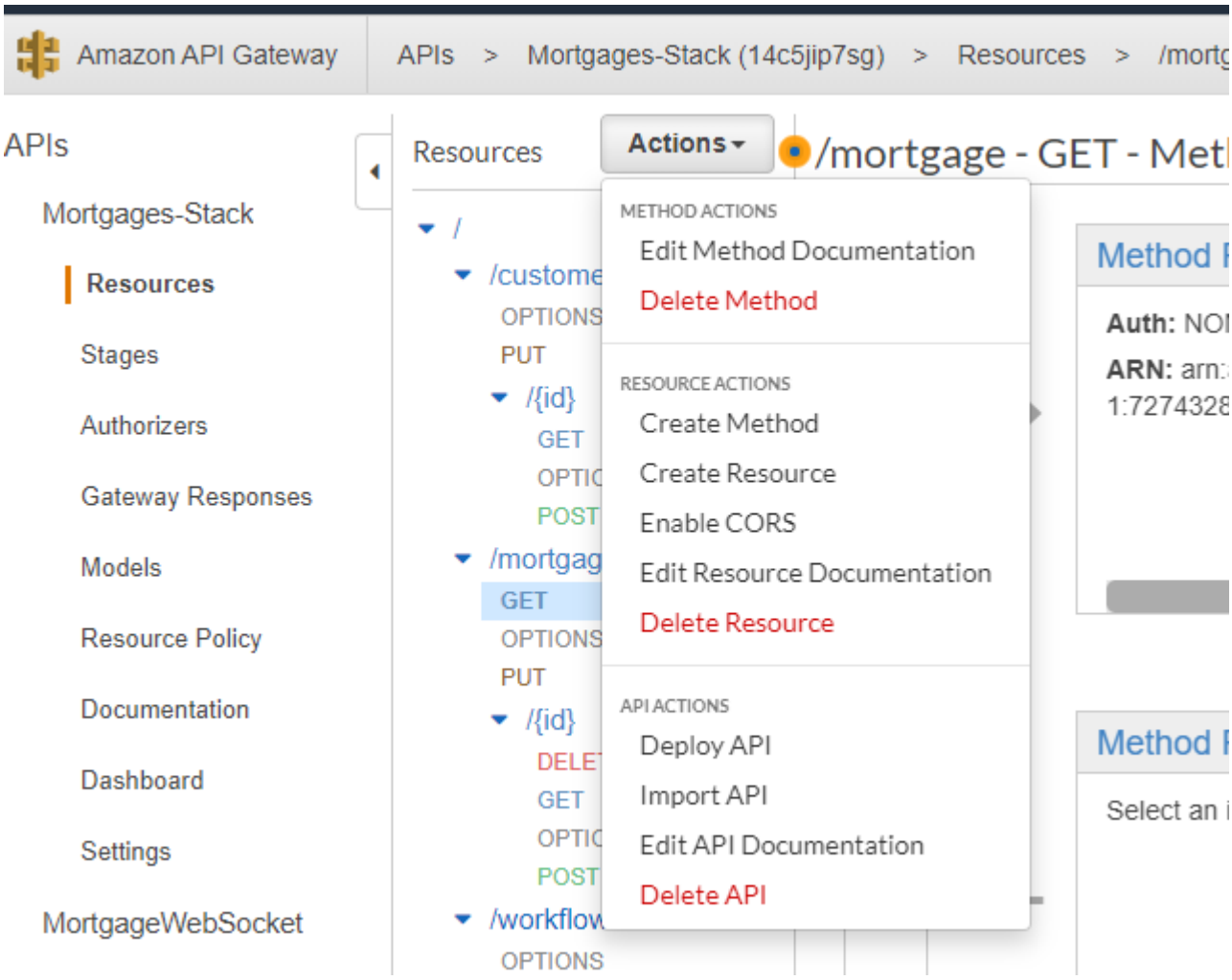


Figure AWS.CORS.2

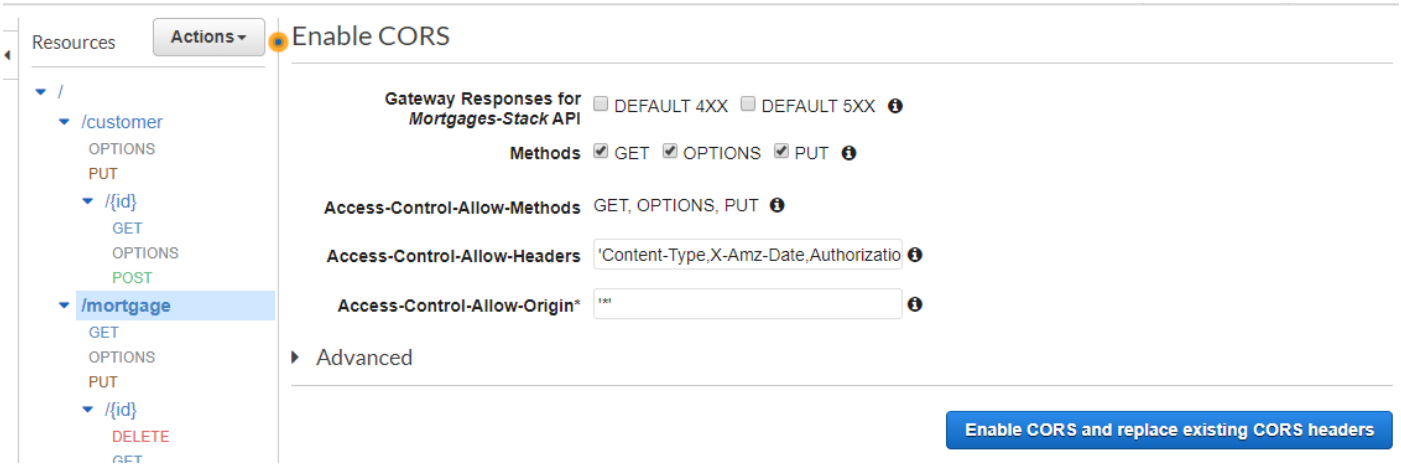


Figure AWS.CORS.3

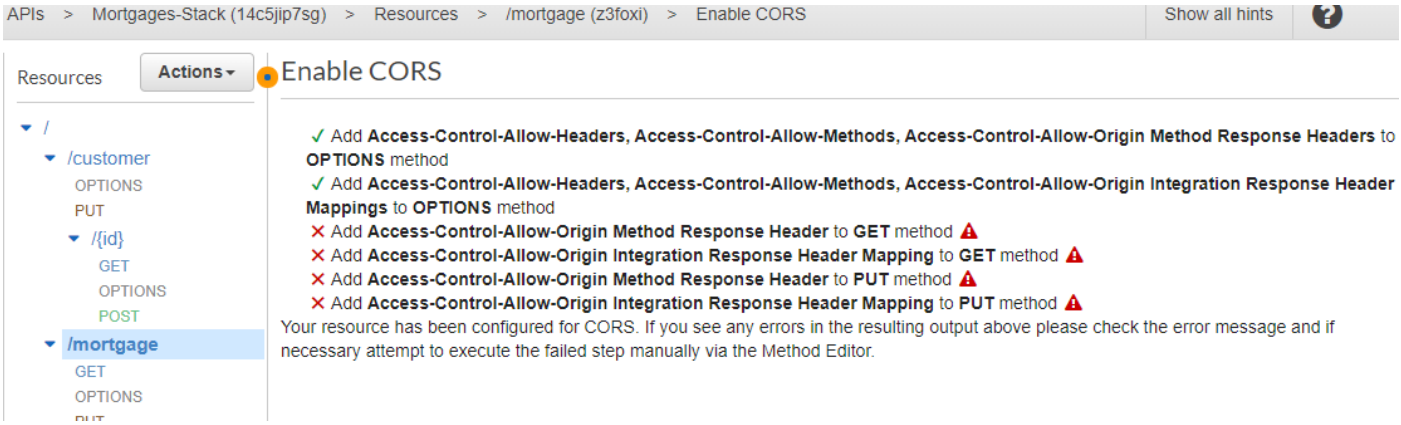


Figure AWS.CORS.4

Some sites indicated that the correct Cloudformation format to enable CORS for API Gateway as shown below.

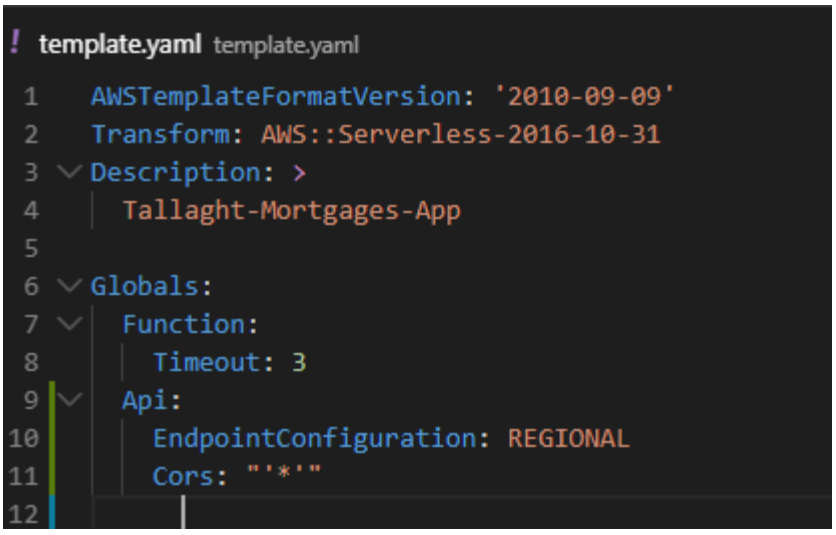


Figure AWS.CORS.5

After much web surfing and trial and error the correct template is shown below:

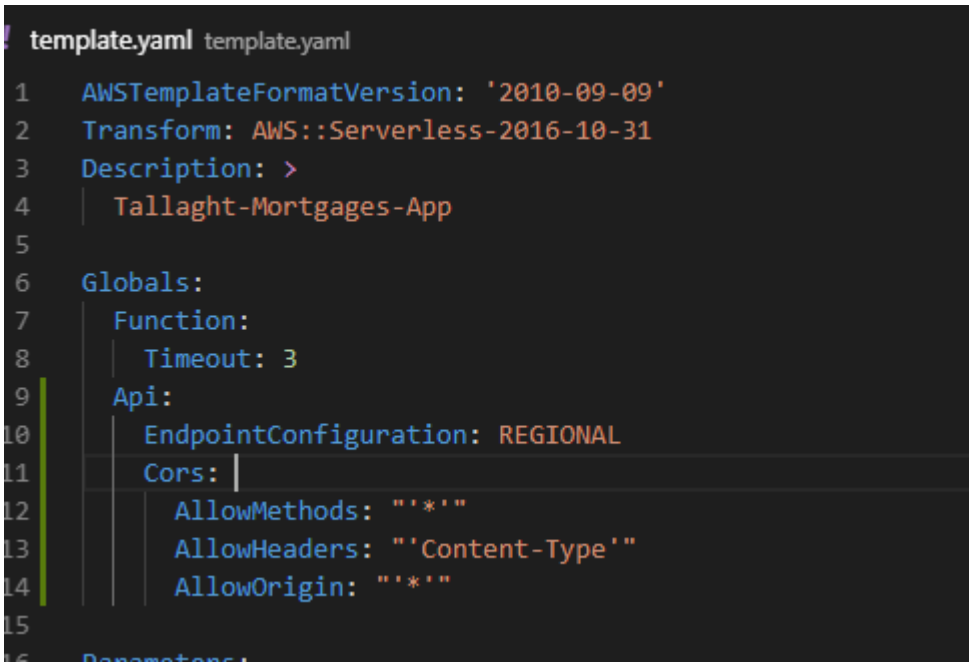


Figure AWS.CORS.6

API Gateway with Lambda integration (rather than HTTP integration) does not fully abstract out CORS functionality. Individual Lambda function must add CORS headers as seen in Figure AWS.CORS.7

```
const findMortgageResponse = await dynamoDb.scan(findAllMortgages).promise();
console.log(findMortgageResponse);
return {
  statusCode: 200,
  headers: { 'Content-Type': 'application/json', "Access-Control-Allow-Origin": "*" },
  body: JSON.stringify(findMortgageResponse.Items)
};
```

Figure AWS.CORS.7

DEVELOPMENT – WORKFLOW IMPLMENTATION – STEP FUNCTION.

The second iteration of the Step Function State Machine was shown in Figure AWS.STEP.1

The workflow has been extended only slightly this week but in the coming week I hope to have completed the workflow by adding the following steps:

1 Add Clarification, Rejection and Approval processes based on the human underwriter decision.

2 Pause the workflow when the Mortgage Application is assigned to a human underwriter.

3 Pause the workflow when the Mortgage Application is returned to the Broker for clarification.

4 Associate the workflow restart token with the mortgage Id and persist to DynamoDB.

5 Complete all of the Websocket and SNS notification paths.

6 Integrate with the Browser SPA, allow users to start and restart the workflow process.

The small enhancements that I have completed for the state machine this week have allowed me to become more familiar with the Step Function functionality and notation. It is very important to have a clear design for the input and outputs of each Task/Lambda functionmethod and the transformations that are required between Tasks to allow subsequent tasks in the workflow to access the required data.

The testing and bug fixing that I performed is summarised now in the following snapshots of two scenarios : Auto Assessment rejection and Auto Assessment success.

Auto Assessment Rejection

Submit a Mortgage Application that does not meet the Auto Assessment requirements. Figure AWS.AUTO.1

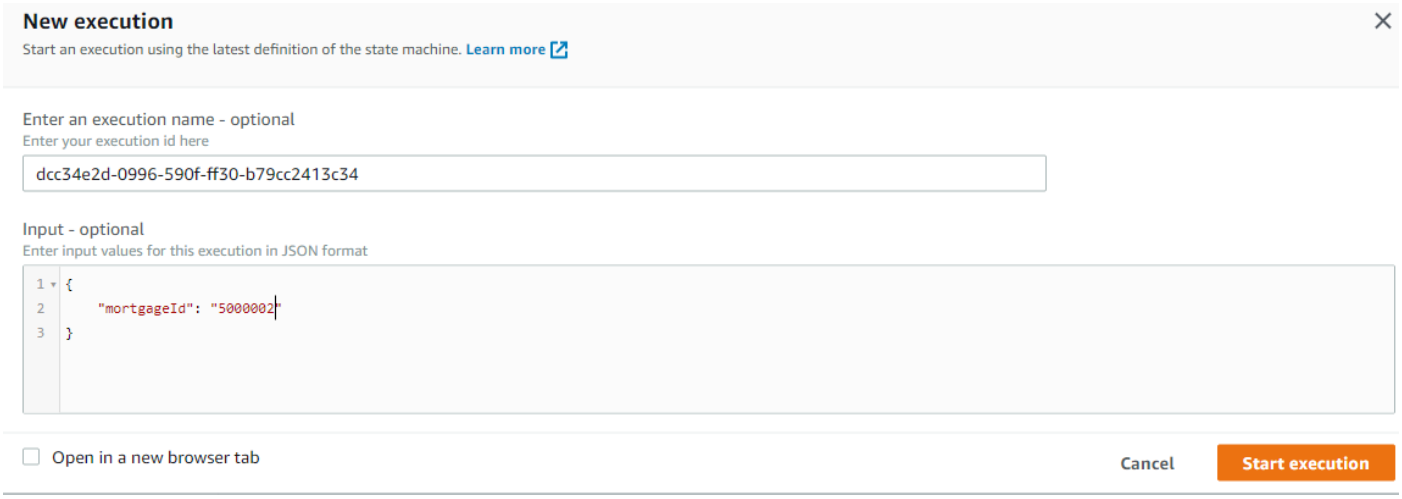
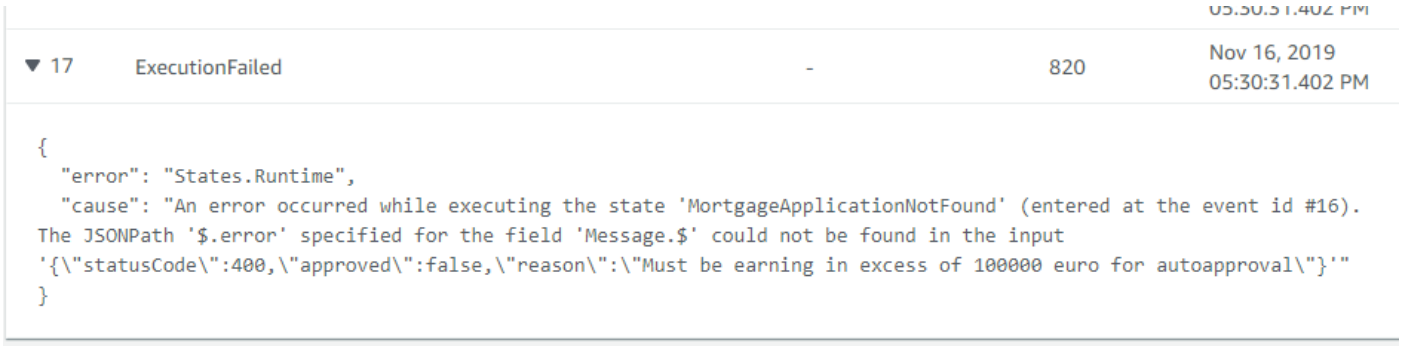


Figure AWS.AUTO.1

Invalid JSON path.



The InputPath on the Task must isolate the required JSON node to provide the correct input to the Lambda function. Figure AWS.AUTO.2

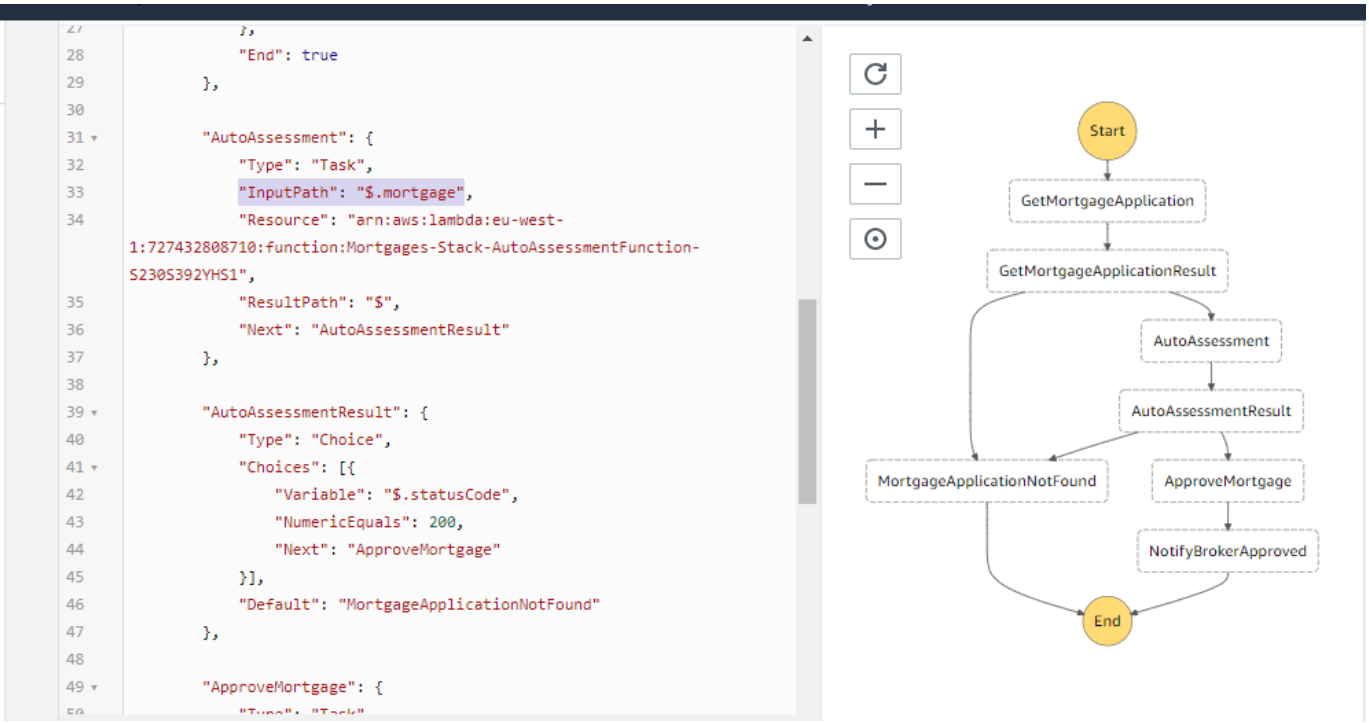


Figure AWS.AUTO.2

Also the Lambda Fucntion must return any and all data that is required for the next task in the chain. Figure AWS.AUTO.3

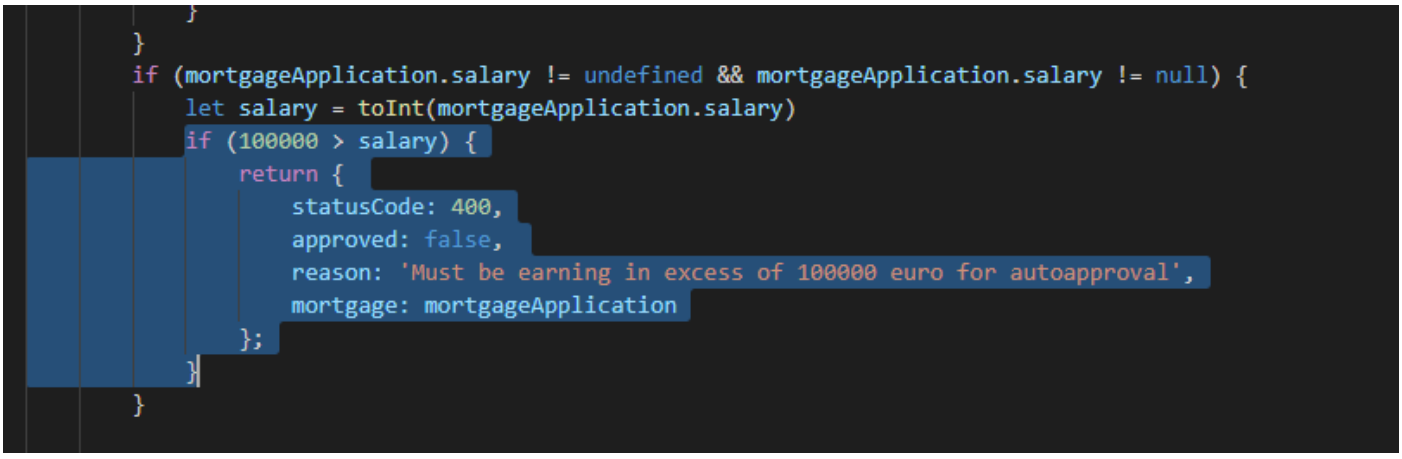


Figure AWS.AUTO.3

If the input and output data is not correctly mediated then the Step Function execution will fail. Figure AWS.AUTO.4

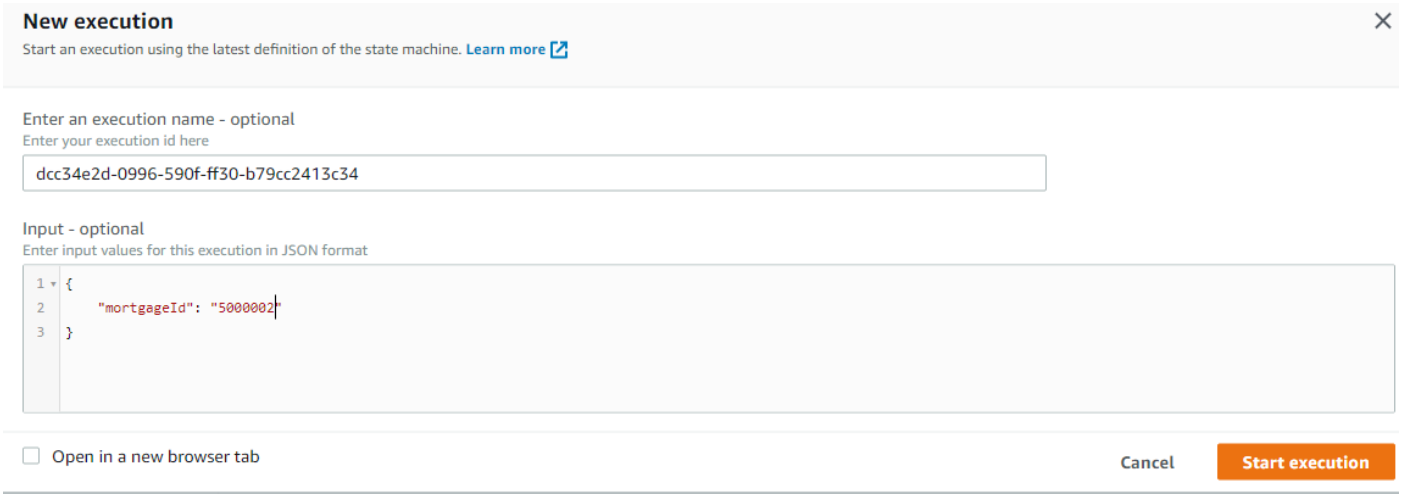


Figure AWS.AUTO.4

The AWS Console will highlight failed states as shown below. Figure AWS.AUTO.5

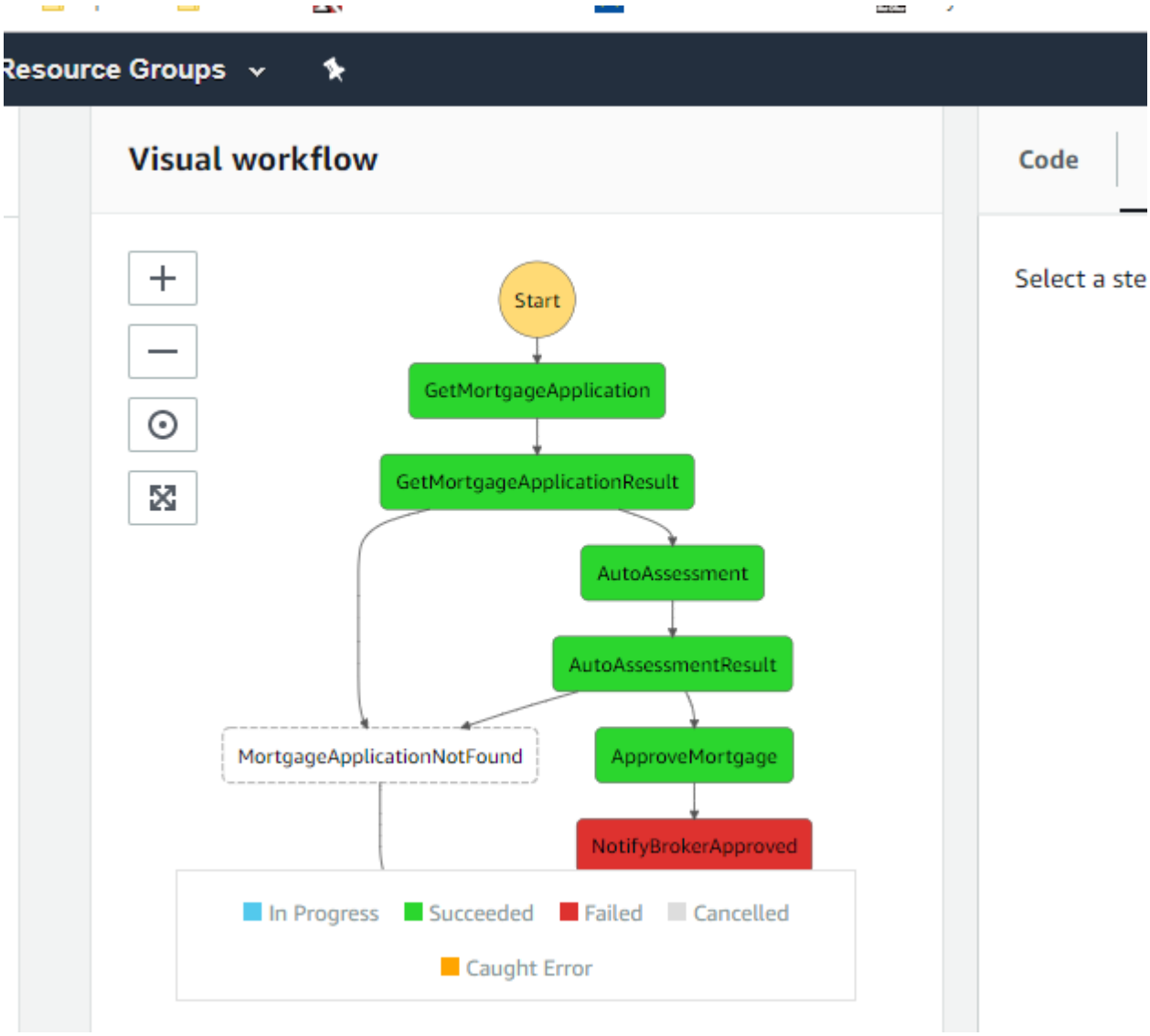


Figure AWS.AUTO.5

Another badly formatted Step Function Task input. Figure AWS.AUTO.6

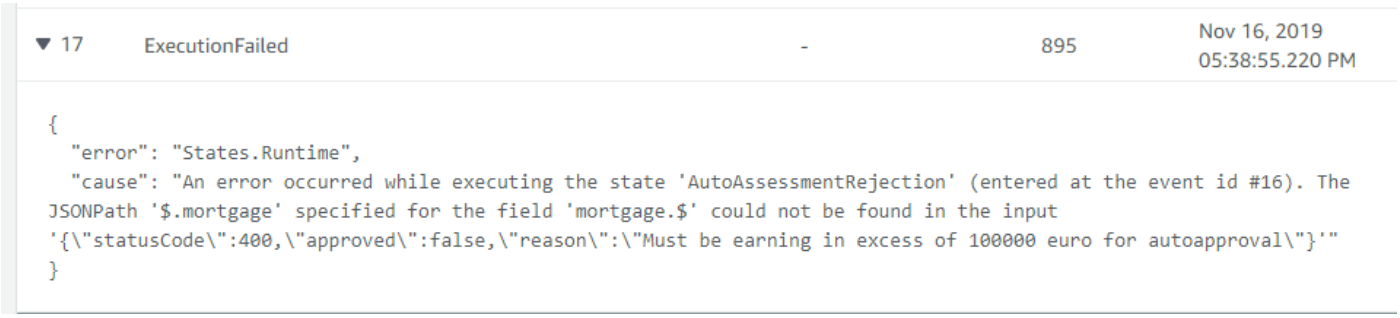


Figure AWS.AUTO.6

Once these errors are rectified then the Step Function execution will success. Figure AWS.AUTO.7

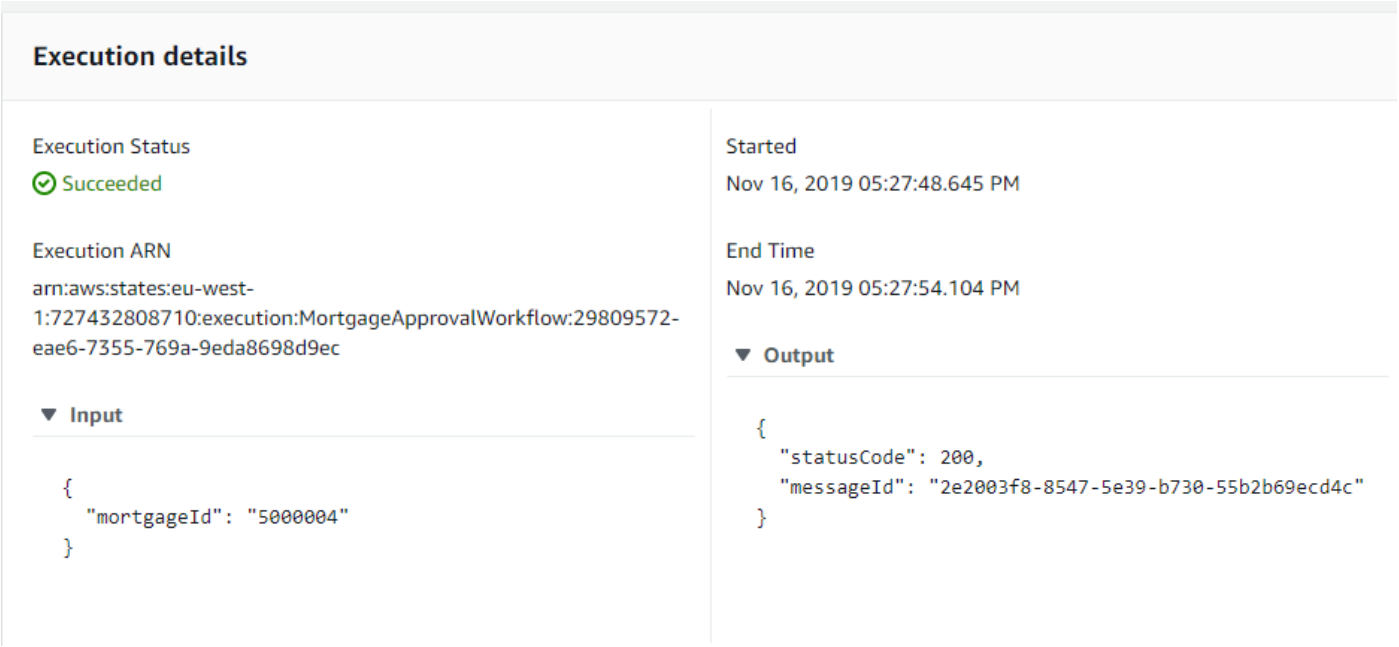
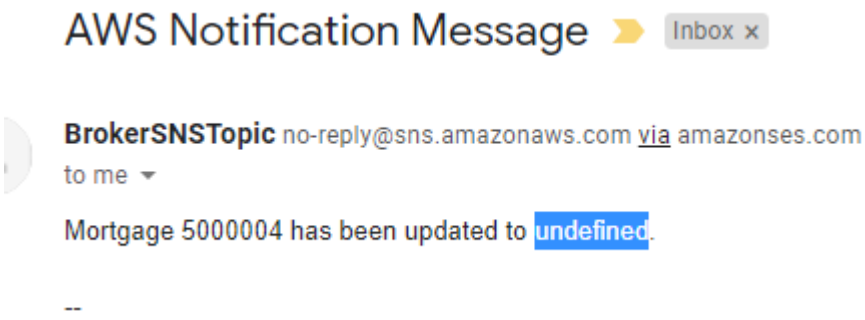


Figure AWS.AUTO.5

The Step Function output correctly shows that the last Task returned a 200 status completion code and also the ID of the SNS message that was send to the users.

However another coding error show that the new state is not correctly rendered in the email. Figure



AWS.AUTO.6

Figure AWS.AUTO.6

Once resolved we see the correct message in the email. Figure AWS.AUTO.7

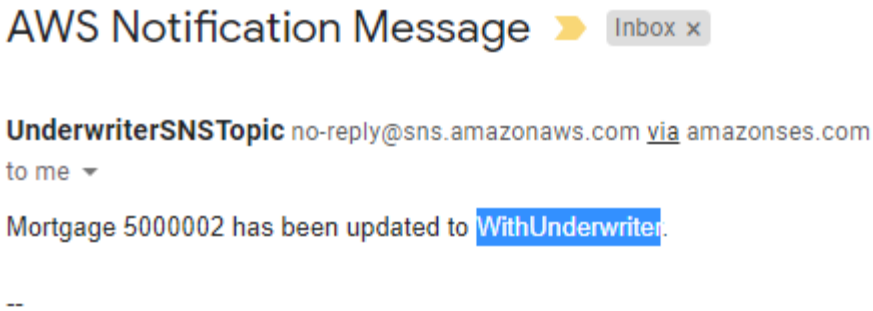


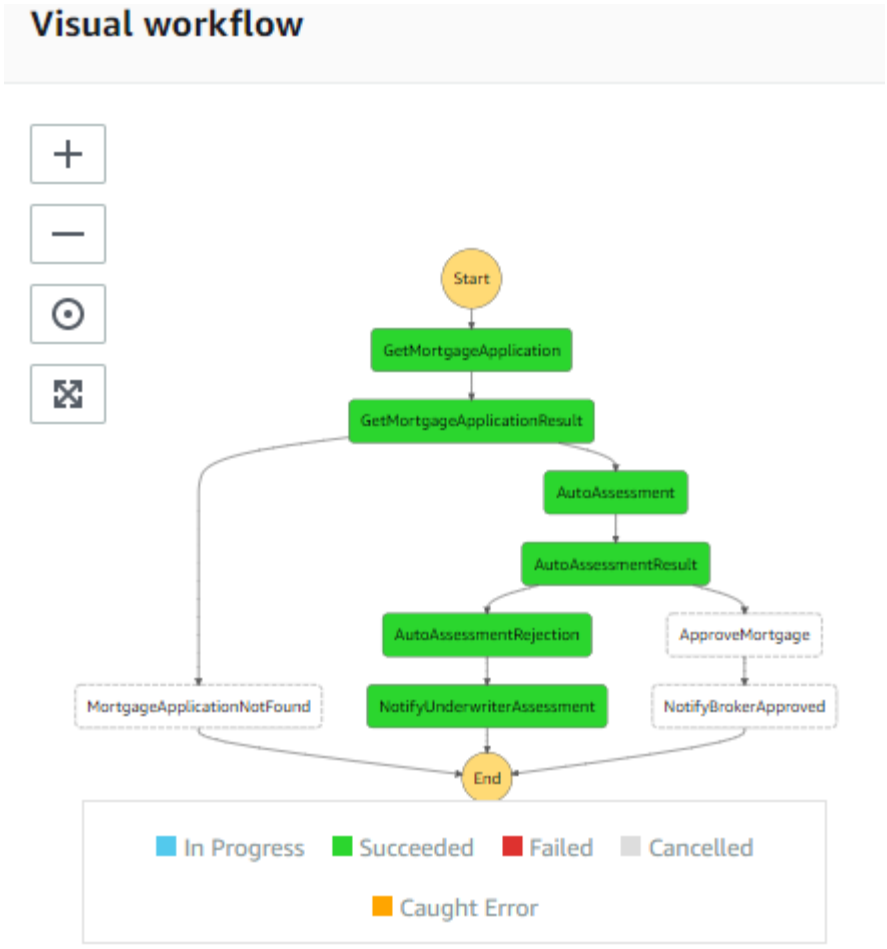
Figure AWS.AUTO.7

The mortgage application has also been correctly updated in DynamoDB. Figure AWS.AUTO.8

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary
5000002	60002	Smurfit	300009	WithUnderwriter	50009
5000003	60003	Microsoft	200000	PreSubmission	100000
5000004	60003	AIB Ltd	999999	PreSubmission	100001

Figure AWS.AUTO.8

The Step Function execution shows the sequence of tasks.



Auto Assessment Success

Select a Mortgage Application that meets ALL Auto Assessment criteria. Figure AWS.AUTO.9

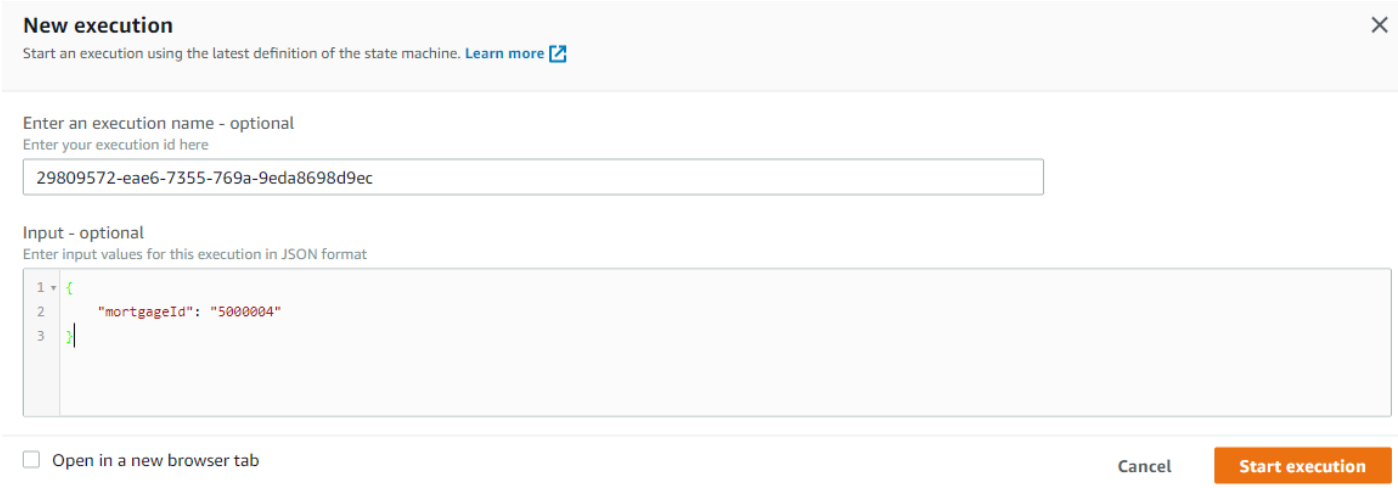


Figure AWS.AUTO.9

Successful email SNS Notification. Figure AWS.AUTO.10

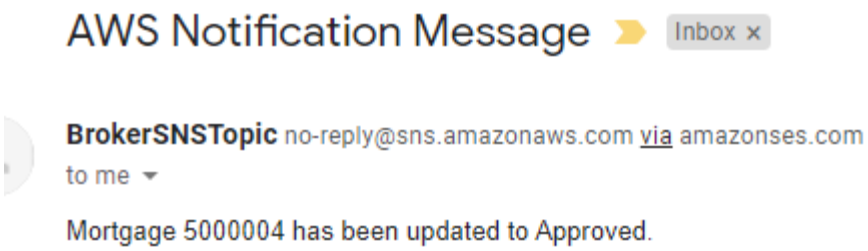


Figure AWS.AUTO.10

DynamoDB has also been updated. Figure AW.S.AUTO.11

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	ter
5000002	60002	Smurfit	300009	PreSubmission	50009	39
5000003	60003	Microsoft	200000	PreSubmission	100000	20
5000004	60003	AIB Ltd	999999	Approved	100001	98

Figure AW.S.AUTO.11

Again the Step Function execution diagram shows the sequene of events. Figure AWS.AUTO.12

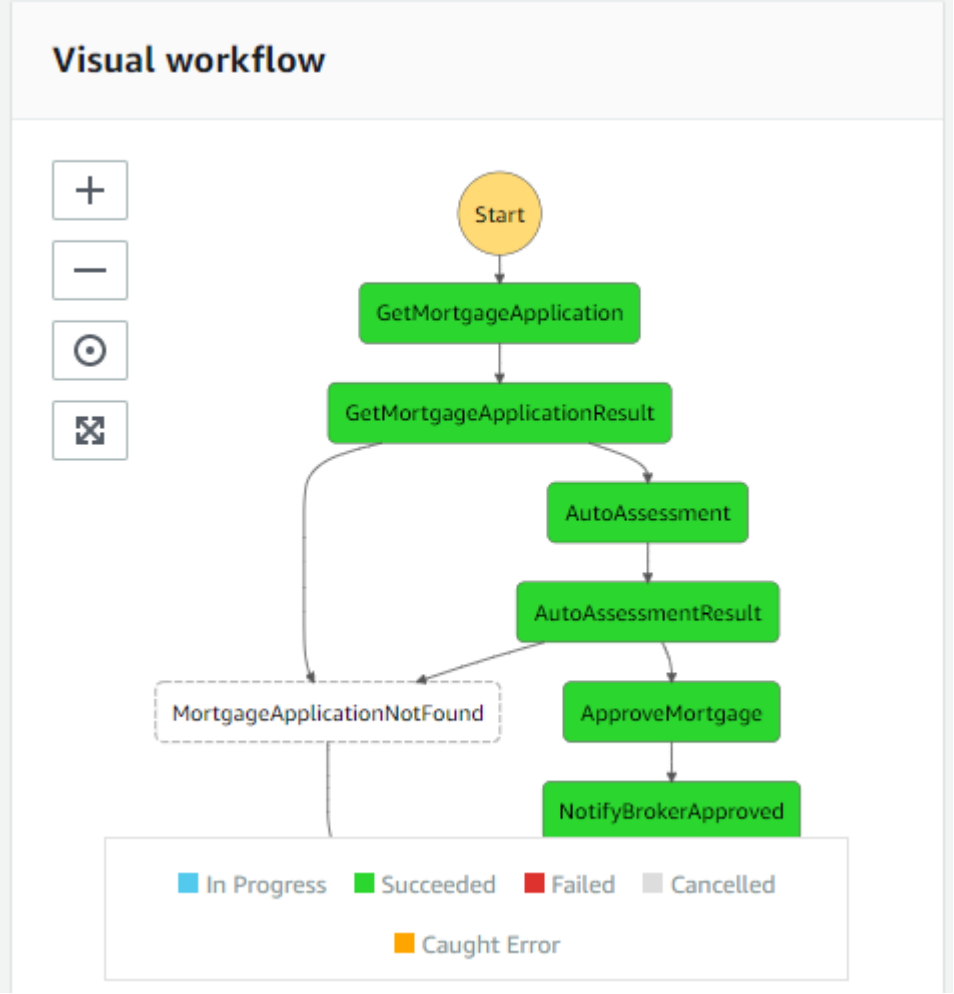


Figure AWS.AUTO.12

PROGRESS FOR SECOND WEEK OF ITERATION TWO.
ITERATION 2.2 TASKS

Building upon last weeks progress report, this document outlines the additional project tasks that have been completed this week for the final implementation phase 2 submission. The follow tasks were completed this week.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Significantly refactor the Step Function Workflow state machine. The original design of the workflow was done without a full understanding of the details and limitations of the State Machine Definition Language in terms of data state and mediation.
Build / Development	AWS Cloudformation	I have not had time to integrate the deployment of the Workflow state machine into the project Cloud formation template.
Build / Development	AWS Lamda	Completed the development of Lambda Functions to allow workflow to await human/underwriting decisions.
Build / Development	DynamoDB	Implement new DB schema to map mortgage application to Step Function execution id. This is required to allow users approve or reject applications without reference to the step function execution id.
Build / Development	Angular	Extends the Browser application to allow submission of mortgages for assessment. I have not had time to extend the functionality to allow for clarification requests.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table D.

Definining the data inputs and outputs of each State within the Step Function workflow requires working within the restrictions of the unility functions provided by the State Machine Definition Language [13]. Input and Output processing [14] must be done using the 3 processing methods : InputPath, ResultPath and OutputPath. These can be restrictive and documentation for complex scenarios is not easily available even within the Step Function Developer Guide [15]. Integration with existing Lambda functions (especially if they where written to used as HTTP endpoints) can require tricky data manipulation. For my Workflow definition I needed to maintain the Mortgage state data and pass it through every state, which does not happen by default as the Lambda output will overwrite the State data if the Lambda result is not saved to a distinct javascript element / node to avoid overwrites. Also removing stale or State specific data can require for example a Pass state that can build new data state for the next state.

PLAN FOR ITERATION 3.1

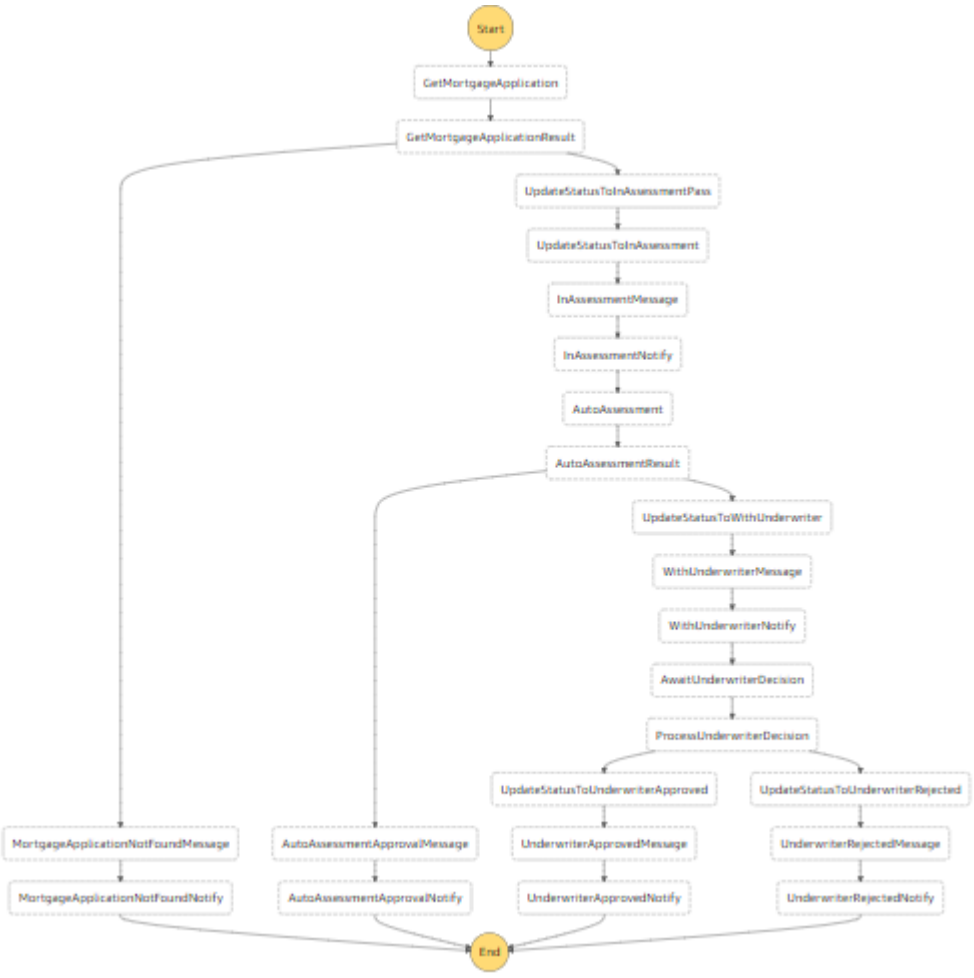
Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker to allow questions be submitted and answered regarding the mortgage application or applicant. The Broker must then be able to restart the workflow.
Build / Development	AWS Lambda	Add persistance to the Step Function tokens to allow users to restart the state machine execution from the Browser. Use either SQS or DynamoDB or both.
Build / Development	AWS S3 / Angular	Add Underwriter screens to the Front end app to allow the brokers to perform their tasks.
Build / Development	AWS S3 / Angular	Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table E.

ITERATION 2 SUBMISSION WORKFLOW STATE MACHINE

The workflow state machine currently exists as seen in Figure AWS.STEP.2.1

Figure AWS.STEP.2.1



ITERATION 2 SUBMISSION TECHNICAL ARCHITECTURE

The overall current technical architecture is outlined in Figure AWS.2.2

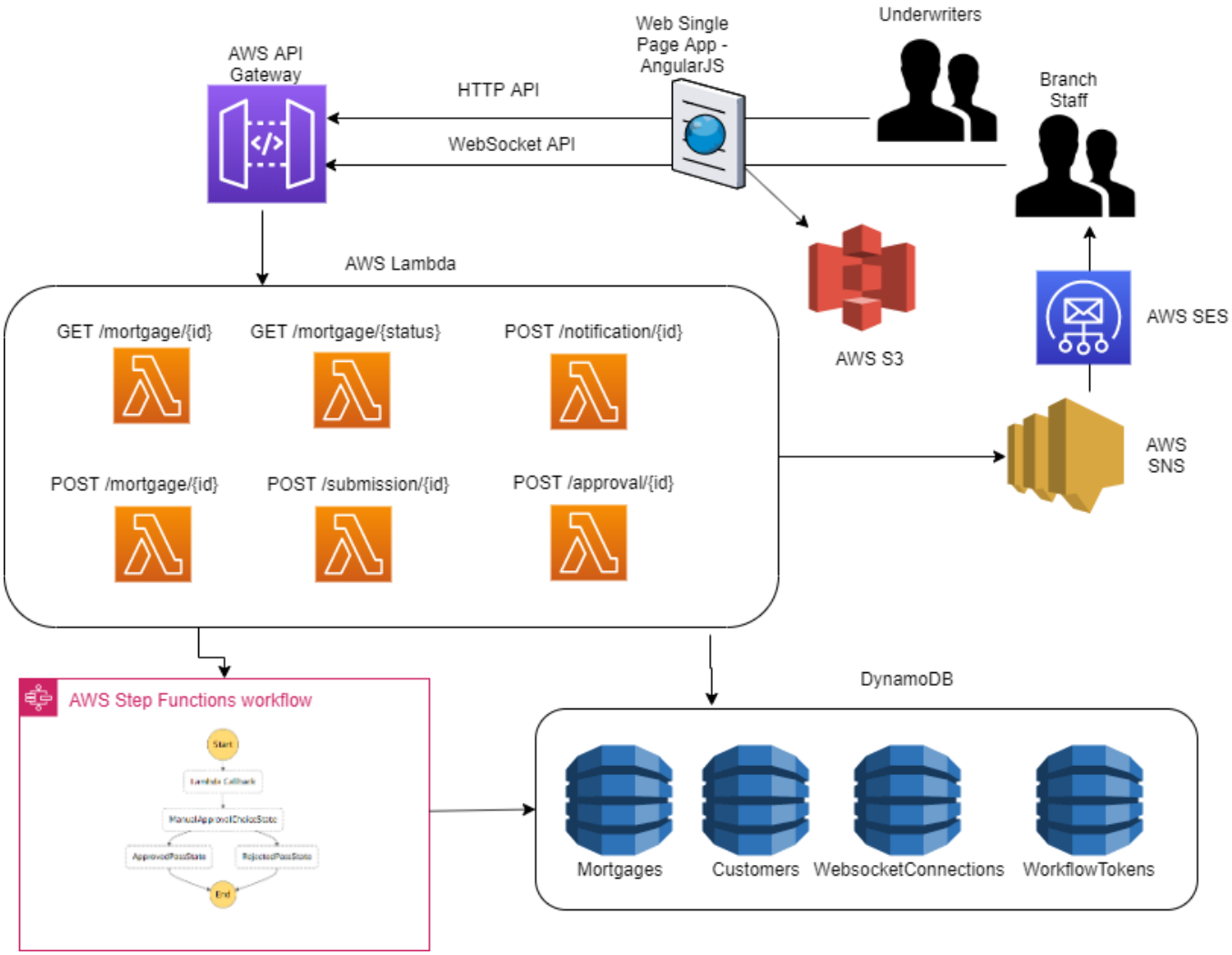


Figure AWS.2.2

ITERATION 2 LAMBDA FUNCTIONS

This table lists the new Lambda functions that I have developed this week in addition to those already developed (previously listed in Table AWS.LAMBDA.1). Table AWS.LAMBDA.2.1

Lambda Function	Purpose
CreateNotificationMessageFunction	This function is used by many states in the workflow to develop appropriate situational notification messages for brokers and underwriters to give them clarity on the current processing state of the mortgage application.
ReceiveUnderwriterDecisionEmailFunction	The Lambda function is exposed as a HTTP API via API Gateway to allow the Underwriter to submit their decision and restart the Workflow execution to process that decision.
SendNotificationMessageFunction	This function is used in combination with CreateNotificationMessageFunction to send the generated message to the approriate SNS topic and the Websocket API for transmision to all connected browser apps.
SendUnderwriterDecisionEmailFunction	The Lambda function is called from the State Machine waitForTask State. It will generate the notification email to allow the underwriter to approve or reject the application. The email will contain the ReceiveUnderwriterDecisionEmailFunction HTTP API endpoint.

Table AWS.LAMBDA.2.1

The source code and deployment template for these functions are available at the GitHub repository listed in Table A or in the Appendix.

ITERATION 2 WORKFLOW HUMAN DECISIONS

This week I built out the functionality to allow the Step Function workflow to pause its execution to allow for Human decision making. Initially I followed to online documentation [16] to build a small test case. Figure AWS.STEP.2.1

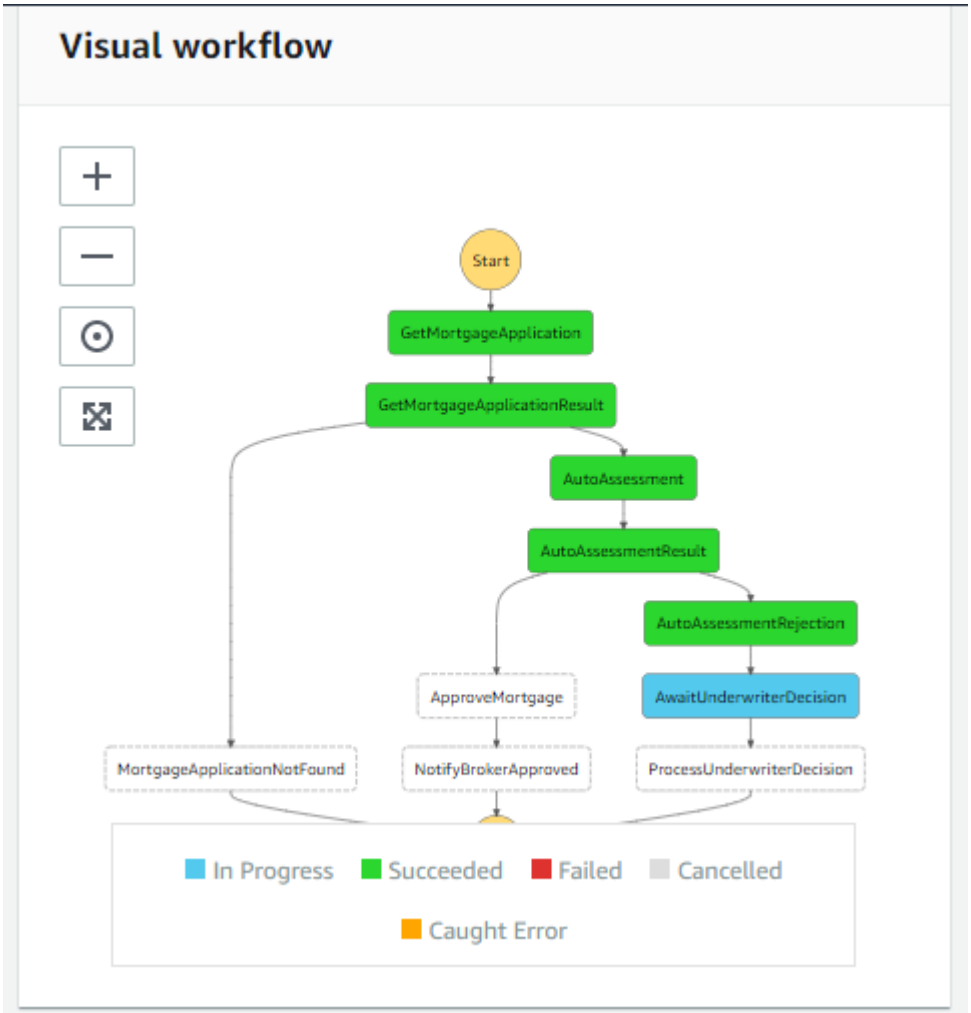


Figure AWS.STEP.2.1

The input to the execution is the Mortgage Id. Figure AWS.STEP.2.2

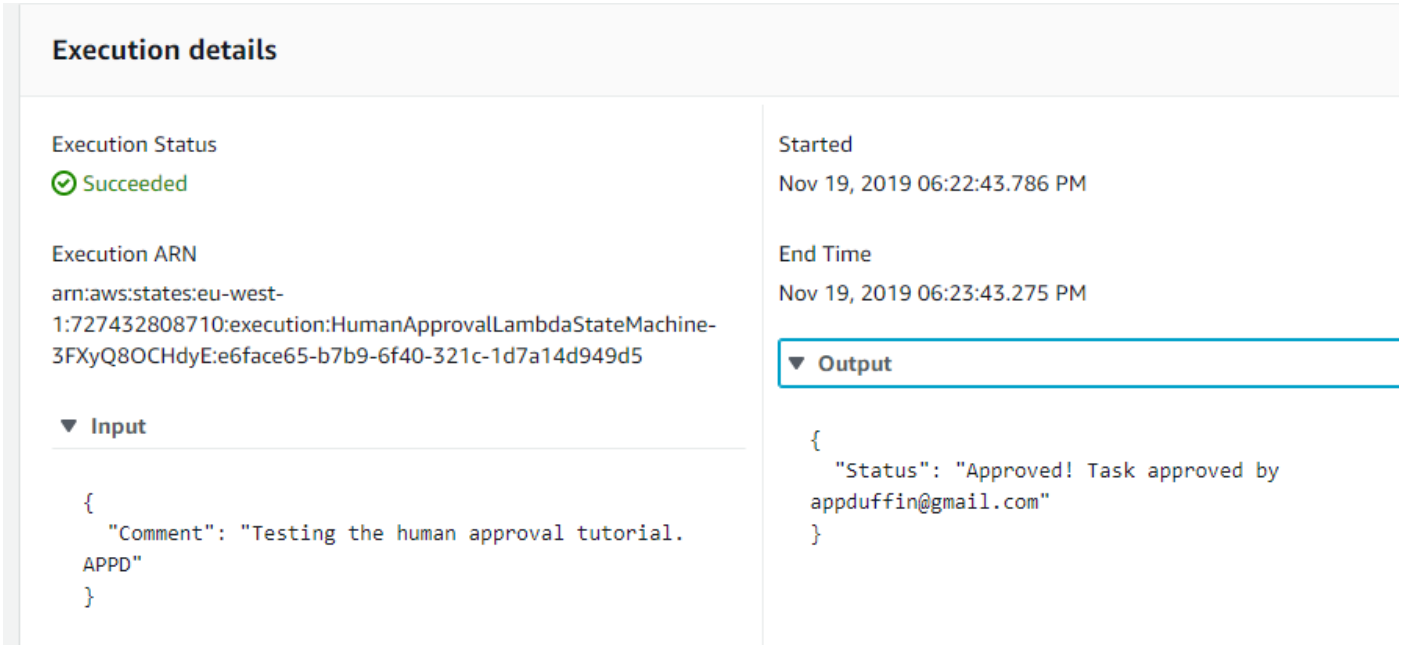


Figure AWS.STEP.2.2

During execution we can see that the state will pause waiting for a Human decision. Figure AWS.STEP.2.3

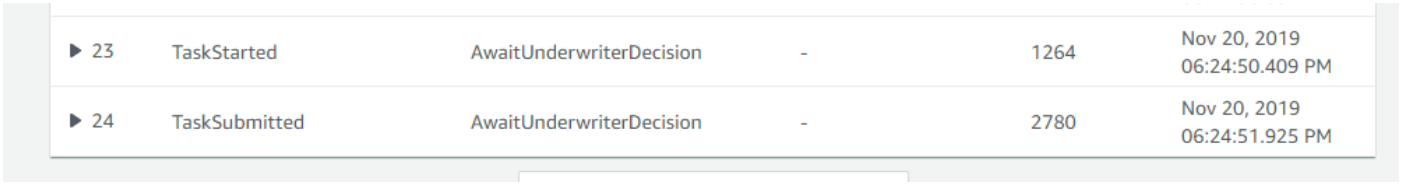


Figure AWS.STEP.2.3

The email is delivered to the registered email address and contains embeded URLs that the underwriter can use to either Approve or reject the application. Figure AWS.STEP.2.4

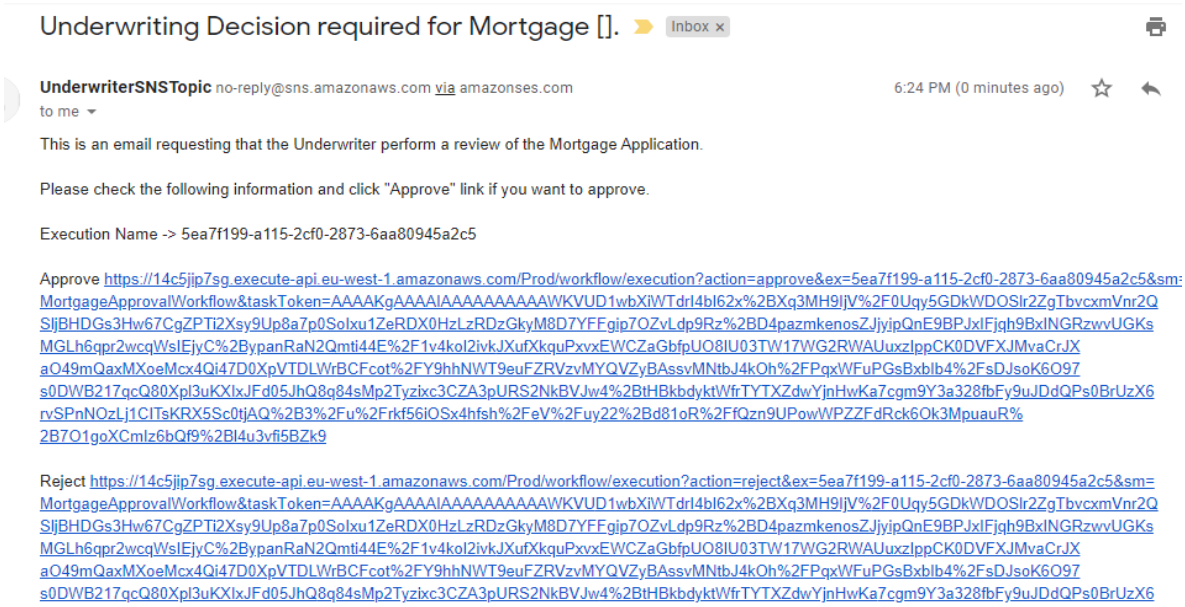


Figure AWS.STEP.2.4

APPENDICES

The following sections list some key / interesting source code / template snippets.

WebSocket Cloudformation Template

```
AWSTemplateFormatVersion: '2010-09-09'

Transform: AWS::Serverless-2016-10-31
Description: >
 Tallaght-Mortgages-App

Globals:
 Function:
  Timeout: 3
 Api:
  EndpointConfiguration: REGIONAL
  Cors:
   AllowMethods: "'*'"
   AllowHeaders: "'Content-Type'"
   AllowOrigin: "'*'"

Parameters:
 LambdaRoleParameter:
  Type: String
  Default: "arn:aws:iam::727432808710:role/TUD.Mortgage.Lambda.Role"
  Description: Enter the ARN of an applicable IAM Role for the Lambda Functions
 StepFunctionArnParameter:
  Type: String
  Default: "arn:aws:states:eu-west-
1:727432808710:stateMachine:MortgageApprovalWorkflow"
  Description: Enter the ARN of the Workflow Step Function State Machine
 WebsocketConnectionTableName:
  Type: String
  Default: 'WEBSOCKET_CONNECTIONS_TABLE'
 StepFunctionTokenTableName:
  Type: String
  Default: 'STEPFUNCTION_TOKEN_TABLE'

Resources:

 # SNS TOPICS
 UnderwriterSNSTopic:
  Type: AWS::SNS::Topic
  Properties:
```

```
  DisplayName: UnderwriterSNSTopic
  Subscription:
   - Endpoint: appduffin@gmail.com
    Protocol: email
  TopicName: UnderwriterSNSTopic
BrokerSNSTopic:
 Type: AWS::SNS::Topic
 Properties:
  DisplayName: BrokerSNSTopic
  Subscription:
   - Endpoint: appduffin@gmail.com
    Protocol: email
  TopicName: BrokerSNSTopic
```

```
# CUSTOMER APIS
GetCustomerFunction:
 Type: AWS::Serverless::Function
 Properties:
  CodeUri: customers/
  Handler: get.get
  Runtime: nodejs10.x
  Role:
   Ref: LambdaRoleParameter
  Events:
   GetCustomerEvent:
    Type: Api
    Properties:
     Path: /customer/{id}
     Method: GET
CreateCustomerFunction:
 Type: AWS::Serverless::Function
 Properties:
  CodeUri: customers/
  Handler: create.create
  Runtime: nodejs10.x
  Role:
   Ref: LambdaRoleParameter
  Events:
   CreateCustomerEvent:
    Type: Api
    Properties:
     Path: /customer/
     Method: PUT
```

```
 UpdateCustomerFunction:
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: customers/
   Handler: update.update
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Events:
    UpdateCustomerEvent:
     Type: Api
     Properties:
      Path: /customer/{id}
      Method: POST

 # MORTGAGE APIS
 GetMortgageFunction:
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: mortgages/
   Handler: get.get
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Events:
    GetMortgageEvent:
     Type: Api
     Properties:
      Path: /mortgage/{id}
      Method: GET
 CreateMortgageFunction:
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: mortgages/
   Handler: create.create
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Events:
    CreateMortgageEvent:
     Type: Api
     Properties:
      Path: /mortgage/
      Method: PUT
 UpdateMortgageFunction:
```

```
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: mortgages/
   Handler: update.update
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Events:
    CreateMortgageEvent:
     Type: Api
     Properties:
      Path: /mortgage/{id}
      Method: POST
 DeleteMortgageFunction:
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: mortgages/
   Handler: delete.delete
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Events:
    DeleteMortgageEvent:
     Type: Api
     Properties:
      Path: /mortgage/{id}
      Method: DELETE
 ListByStatusMortgageFunction:
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: mortgages/
   Handler: list.list
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Events:
    ListByStatusMortgageEvent:
     Type: Api
     Properties:
      Path: /mortgage/
      Method: GET

 # STEP FUNCTION HUMAN TOKEN DYNAMO DB TABLE
 StepFunctionTokenTable:
  Type: AWS::DynamoDB::Table
```

```
  Properties:
    TableName: !Ref StepFunctionTokenTableName
    AttributeDefinitions:
      - AttributeName: "mortgageId"
        AttributeType: "S"
#       - AttributeName: "stepFunctionToken"
#         AttributeType: "S"
    KeySchema:
      - AttributeName: "mortgageId"
        KeyType: "HASH"
    BillingMode: PAY_PER_REQUEST

  # WORKFLOW APIS
  NofifyUsersSNSFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/notifications/snsNotifications.handler
      Runtime: nodejs10.x
      Role:
        Ref: LambdaRoleParameter
      Environment:
        Variables:
          UNDERWRITER_TOPIC_ARN: !Ref UnderwriterSNSTopic
          BROKER_TOPIC_ARN: !Ref BrokerSNSTopic
    DependsOn:
      - UnderwriterSNSTopic
      - BrokerSNSTopic

  CreateNotificationMessageFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/notifications/notificationMessages.handler
      Runtime: nodejs10.x
      Role:
        Ref: LambdaRoleParameter
      Environment:
        Variables:
          UNDERWRITER_TOPIC_ARN: !Ref UnderwriterSNSTopic
          BROKER_TOPIC_ARN: !Ref BrokerSNSTopic
    DependsOn:
      - UnderwriterSNSTopic
      - BrokerSNSTopic

  SendNotificationMessageFunction:
    Type: AWS::Serverless::Function
```

```
  Properties:
    Handler: stepfunctions/notifications/sendNotifications.handler
    Runtime: nodejs10.x
  #  Role:
     # Ref: LambdaRoleParameter
    Environment:
     Variables:
      TABLE_NAME: !Ref WebsocketConnectionTableName
      WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-
api.',!Ref 'AWS::Region','.amazonaws.com/Prod'] ]
    Policies:
    - DynamoDBCrudPolicy:
      TableName: !Ref WebsocketConnectionTableName
    - Statement:
     - Effect: Allow
      Action:
      - 'sns:*'
      Resource:
      - '*'
    - Statement:
     - Effect: Allow
      Action:
      - 'execute-api:ManageConnections'
      Resource:
      - !Sub 'arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}:${MortgageWebSocket}/*'
  DependsOn:
    - UnderwriterSNSTopic
    - BrokerSNSTopic
    - WebsocketConnectionsTable
    - MortgageWebSocket

  WorkflowGetCustomerFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/customers/findByPK.handler
      Runtime: nodejs10.x
      Role:
       Ref: LambdaRoleParameter
  WorkflowGetMorgtageFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/mortgages/findByPK.handler
      Runtime: nodejs10.x
      Role:
```

```
    Ref: LambdaRoleParameter
  WorkflowUpdateMorgtageFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/mortgages/updateMortgage.handler
      Runtime: nodejs10.x
      Role:
        Ref: LambdaRoleParameter
  StartWorkflowFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/startWorkflow.handler
      Runtime: nodejs10.x
      Role:
        Ref: LambdaRoleParameter
      Environment:
        Variables:
          MORTGAGE_WORKFLOW_ARN: !Ref StepFunctionArnParameter
      Events:
        StartWorkflowEvent:
          Type: Api
          Properties:
            Path: /workflow/execution
            Method: PUT
  AutoAssessmentFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: workflow/autoassessment.handler
      Runtime: nodejs10.x
      Role:
        Ref: LambdaRoleParameter
  ChangeMortgageStatusFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/mortgages/changeMortgageStatus.handler
      Runtime: nodejs10.x
      Role:
        Ref: LambdaRoleParameter
  MortgageStatusSNSFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: stepfunctions/notifications/mortgageStatusSend.handler
      Runtime: nodejs10.x
      Role:
        Ref: LambdaRoleParameter
```

```
 SendUnderwriterDecisionEmailFunction:
  Type: AWS::Serverless::Function
  Properties:
   Handler: workflow/sendUnderwriterApprovalEmail.handler
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Environment:
    Variables:
     UNDERWRITER_DECISION_API: !Sub "https://${ServerlessRestApi}.execute-
api.${AWS::Region}.amazonaws.com/Prod/workflow/execution"
#          UNDERWRITER_DECISION_API: !Join [ 'https://', [ !Ref ServerlessRestA
pi, '.execute-
api.',!Ref 'AWS::Region','.amazonaws.com/Prod/workflow/execution'] ]
     UNDERWRITER_TOPIC_ARN: !Ref UnderwriterSNSTopic

 ReceiveUnderwriterDecisionEmailFunction:
  Type: AWS::Serverless::Function
  Properties:
   Handler: workflow/receiveUnderwriterDecision.handler
   Runtime: nodejs10.x
   Role:
    Ref: LambdaRoleParameter
   Events:
    UnderwriterDecisionEvent:
     Type: Api
     Properties:
      Path: /workflow/execution
      Method: GET

 # WEBSOCKET APIS
 MortgageWebSocket:
  Type: AWS::ApiGatewayV2::Api
  Properties:
   Name: MortgageWebSocket
   ProtocolType: WEBSOCKET
   RouteSelectionExpression: "$request.body.message"
 ConnectRoute:
  Type: AWS::ApiGatewayV2::Route
  Properties:
   ApiId: !Ref MortgageWebSocket
   RouteKey: $connect
   AuthorizationType: NONE
   OperationName: ConnectRoute
   Target: !Join
```

```
    - '/'
    - - 'integrations'
      - !Ref MortgateConnectionIntegration
 MortgateConnectionIntegration:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
   ApiId: !Ref MortgageWebSocket
   Description: Connect Integration
   IntegrationType: AWS_PROXY
   IntegrationUri:
    Fn::Sub:
      arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketConnectFunction.Arn}/invocations
 DisconnectRoute:
  Type: AWS::ApiGatewayV2::Route
  Properties:
   ApiId: !Ref MortgageWebSocket
   RouteKey: $disconnect
   AuthorizationType: NONE
   OperationName: DisconnectRoute
   Target: !Join
    - '/'
    - - 'integrations'
      - !Ref MortgateDisconnectInteg
 MortgateDisconnectInteg:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
   ApiId: !Ref MortgageWebSocket
   Description: Disconnect Integration
   IntegrationType: AWS_PROXY
   IntegrationUri:
    Fn::Sub:
      arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketDisconnectFunction.Arn}/invocations
 SendRoute:
  Type: AWS::ApiGatewayV2::Route
  Properties:
   ApiId: !Ref MortgageWebSocket
   RouteKey: sendmessage
   AuthorizationType: NONE
   OperationName: SendRoute
   Target: !Join
    - '/'
    - - 'integrations'
      - !Ref SendInteg
```

```
 SendInteg:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
   ApiId: !Ref MortgageWebSocket
   Description: Send Integration
   IntegrationType: AWS_PROXY
   IntegrationUri:
    Fn::Sub:
      arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketSendFunction.Arn}/invocations
 Deployment:
  Type: AWS::ApiGatewayV2::Deployment
  DependsOn:
  - ConnectRoute
  - SendRoute
  - DisconnectRoute
  Properties:
   ApiId: !Ref MortgageWebSocket
 Stage:
  Type: AWS::ApiGatewayV2::Stage
  Properties:
   StageName: Prod
   Description: Prod Stage
   DeploymentId: !Ref Deployment
   ApiId: !Ref MortgageWebSocket

 # WEBSOCKET DYNAMO DB TABLE
 WebsocketConnectionsTable:
  Type: AWS::DynamoDB::Table
  Properties:
   TableName: !Ref WebsocketConnectionTableName
   AttributeDefinitions:
    - AttributeName: "connectionId"
     AttributeType: "S"
   KeySchema:
    - AttributeName: "connectionId"
     KeyType: "HASH"
   BillingMode: PAY_PER_REQUEST

 # WEBSOCKET LAMBDA API
 WebSocketConnectFunction:
  Type: AWS::Serverless::Function
  Properties:
   CodeUri: websockets/onconnect/
   Handler: app.handler
```

```
  MemorySize: 256
  Runtime: nodejs10.x
  Environment:
   Variables:
    TABLE_NAME: !Ref WebsocketConnectionTableName
  Policies:
  - DynamoDBCrudPolicy:
    TableName: !Ref WebsocketConnectionTableName
OnConnectPermission:
 Type: AWS::Lambda::Permission
 DependsOn:
  - MortgageWebSocket
  - WebSocketConnectFunction
 Properties:
  Action: lambda:InvokeFunction
  FunctionName: !Ref WebSocketConnectFunction
  Principal: apigateway.amazonaws.com
WebSocketDisconnectFunction:
 Type: AWS::Serverless::Function
 Properties:
  CodeUri: websockets/ondisconnect/
  Handler: app.handler
  MemorySize: 256
  Runtime: nodejs10.x
  Environment:
   Variables:
    TABLE_NAME: !Ref WebsocketConnectionTableName
  Policies:
  - DynamoDBCrudPolicy:
    TableName: !Ref WebsocketConnectionTableName
OnDisconnectPermission:
 Type: AWS::Lambda::Permission
 DependsOn:
  - MortgageWebSocket
  - WebSocketDisconnectFunction
 Properties:
  Action: lambda:InvokeFunction
  FunctionName: !Ref WebSocketDisconnectFunction
  Principal: apigateway.amazonaws.com
WebSocketSendFunction:
 Type: AWS::Serverless::Function
 Properties:
  CodeUri: websockets/sendmessage/
  Handler: app.handler
  MemorySize: 256
```

```
  Runtime: nodejs10.x
  Environment:
   Variables:
    TABLE_NAME: !Ref WebsocketConnectionTableName
    WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-
api.',!Ref 'AWS::Region','.amazonaws.com/Prod'] ]
  Policies:
  - DynamoDBCrudPolicy:
    TableName: !Ref WebsocketConnectionTableName
  - Statement:
   - Effect: Allow
    Action:
    - 'execute-api:ManageConnections'
    Resource:
    - !Sub 'arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}:${MortgageWebSocket}/*'
 WebSocketSendPermission:
  Type: AWS::Lambda::Permission
  DependsOn:
   - MortgageWebSocket
   - WebSocketSendFunction
  Properties:
   Action: lambda:InvokeFunction
   FunctionName: !Ref WebSocketSendFunction
   Principal: apigateway.amazonaws.com

Outputs:
 MortgageApi:
  Description: "API Gateway endpoint URL for Prod stage for GetMortgageFunctionR
ole"
  Value: !Sub "https://${ServerlessRestApi}.execute-
api.${AWS::Region}.amazonaws.com/Prod/mortgage/"
 GetCustomerFunction:
  Description: "GetCustomerFunctionARN"
  Value: !GetAtt GetCustomerFunction.Arn
 CreateCustomerFunction:
  Description: "CreateCustomerFunctionARN"
  Value: !GetAtt CreateCustomerFunction.Arn
 UpdateCustomerFunction:
  Description: "UpdateCustomerFunctionARN"
  Value: !GetAtt UpdateCustomerFunction.Arn

 WebSocketConnectFunctionArn:
  Description: "Websocket OnConnect function ARN"
  Value: !GetAtt WebSocketConnectFunction.Arn
```

```
 WebSocketDisconnectFunctionArn:
  Description: "Websocket OnDisconnect function ARN"
  Value: !GetAtt WebSocketDisconnectFunction.Arn

 WebSocketSendFunctionArn:
  Description: "Websocket SendMessage function ARN"
  Value: !GetAtt WebSocketSendFunction.Arn

 WebSocketURI:
  Description: "Websocket WSS Protocol URI"
  Value: !Join [ '', [ 'wss://', !Ref MortgageWebSocket, '.execute-
api.',!Ref 'AWS::Region','.amazonaws.com/',!Ref 'Stage'] ]
```

REFERENCE LIST AND BIBLIOGRAPHY

[1] AWS Free Tier. https://aws.amazon.com/free/

[2] AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html

[3] AWS CLI SAM. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html

[4] Node JS. https://nodejs.org/en/download/

[5] Configure AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

[6] Visual Studio Code. https://code.visualstudio.com/download

[7] Configure AWS on Visual Studio Code. https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html

[8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/

[9] DynamoDB Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml

[10] Lambda Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml

[11] Step Function State Machine Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt

[12] https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/

[13] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html

[14] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html

[15] https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html

[16] https://docs.aws.amazon.com/step-functions/latest/dg/connect-to-resource.html#connect-wait-token

01/12/2019

ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

IMPLEMENTATION PHASE 3.1

TABLE OF CONTENTS

PROJECT PLAN FOR ITERATION 3

The table below (Table A) lists the tasks that I will attempt to complete before the final Iteration 3 deadline.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker to allow questions be submitted and answered regarding the mortgage application or applicant. The Broker must then be able to restart the workflow.
Build / Development	AWS Lambda	Add persistance to the Step Function tokens to allow users to restart the state machine execution from the Browser. Use either SQS or DynamoDB or both.
Build / Development	AWS S3 / Angular	Add Underwriter screens to the Front end app to allow the brokers to perform their tasks.
Build / Development	AWS S3 / Angular	Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table A.

PROJECT PROGRESS

This week I completed the following tasks:

- Add persistance to the Step Function tokens to allow users to restart the state machine execution from the Browser. Use either SQS or DynamoDB or both.
- Add Underwriter screens to the Front end app to allow the brokers to perform their tasks.
- Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
- Test State Transitions / Events / Human interaction

Please review the video (MP4) at the following URL:

https://s3-eu-west-1.amazonaws.com/appd.cloud.project/Cloud-Project-Implementation-Phase3.1-2019-12-01+11-44-04.mp4

WORKFLOW / FRONTEND INTEGRATION AND TOKEN PERSISTENCE

In order to allow Underwriters approve / reject / clarify mortgage applications from the Browser based front end there must be a persistence mechanism to map the mortgage application id to the Step Function Execution restart token. So the first task is to build a DynamoDB table. This is done with the SAM CloudFormation template. Figure 3.1.1

```
# STEP FUNCTION HUMAN TOKEN DYNAMO DB TABLE
 StepFunctionTokenTable:
  Type: AWS::DynamoDB::Table
  Properties:
   TableName: !Ref StepFunctionTokenTableName
   AttributeDefinitions:
    - AttributeName: "mortgageId"
     AttributeType: "S"
   KeySchema:
    - AttributeName: "mortgageId"
     KeyType: "HASH"
   BillingMode: PAY_PER_REQUEST
```

Figure 3.1.1

We then need to modify the Underwriter Decision process to store the token that is issued by the Step Function *waitForTaskToken* function. Figure 3.1.2.

```
      "AwaitUnderwriterDecision": {
        "Type": "Task",
        "Resource": "arn:aws:states:::lambda:invoke.waitForTaskToken",
        "Parameters": {
         "FunctionName": "${SendUnderwriterDecisionEmailArn}",
         "Payload": {
          "mortgage.$": "$.mortgage",
          "ExecutionContext.$": "$$"
         }
        },
        "TimeoutSeconds": 180,
        "ResultPath": "$.lambdaresult",
        "Next": "ProcessUnderwriterDecision"
       },
```

Figure 3.1.2

We must modify the Lambda functions that are involved in this process.

startUnderwriterDecisionProcess.js

```
    var dbTokenParam = {
      TableName: STEP_FUNCTION_TABLE_NAME,
      Item: {
        'mortgageId': mortgage.mortgageId,
        'stepFunctionToken': taskToken
      }
    };

    await dynamoDB.put(dbTokenParam).promise();
```

underwriterDecisionPOST.js

```
var dbTokenParam = {
      TableName: STEP_FUNCTION_TABLE_NAME,
      Key: {
        'mortgageId': decision.mortgageId
      }
    };

    const tokenItem = await dynamoDB.get(dbTokenParam).promise();
    await dynamoDB.delete(dbTokenParam).promise();
```

underwriterDecisionGET.js

```
 const action = event.queryStringParameters.action;
 const taskToken = event.queryStringParameters.taskToken;
 const mortgageId = event.queryStringParameters.mortgageId;

 var dbTokenParam = {
    TableName: STEP_FUNCTION_TABLE_NAME,
    Key: {
      'mortgageId': mortgageId
    }
 };

 const response = stepTokenUtil.restartWorkflow(mortgageId, taskToken, action);
 await dynamoDB.delete(dbTokenParam).promise();
```

It was necessary to refactor the Step Function restart process because there are now two pathways for the Underwriters to submit a decision. When the Underwriter interacts with the State Machine execution via email the decision is submitted by embedded URLs with perform HTTP GET operations on API Gateway endpoints. Alternatively the Underwriter can submit the decision using the Browser based Angular frontend. In that scenario the decision is submitted via a HTTP POST operation.

In order to facilitate this two new Lamda Functions were creates that act as a thin wrapper layer to extract the requried data from the HTTP operations either from the HTTP POST body or the HTTP GET path variables. Both new functions then invoke common utilitity code to restart the Step Function workflow. This is found in the */workflow/utils.restartWorkflow.js* source code file.

FRONT END ENHANCEMENTS

I have enhanced the front end Angular code to allow underwriters to view Mortgage Applications that require their attention. Also they can inspect the mortgage application information and approve or reject applications from within the browser web app.

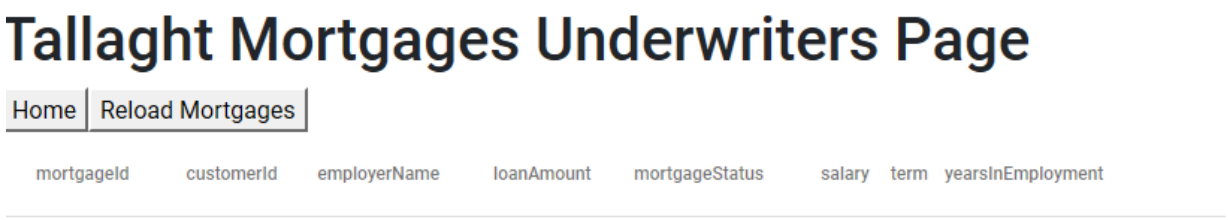


Figure 3.1.3

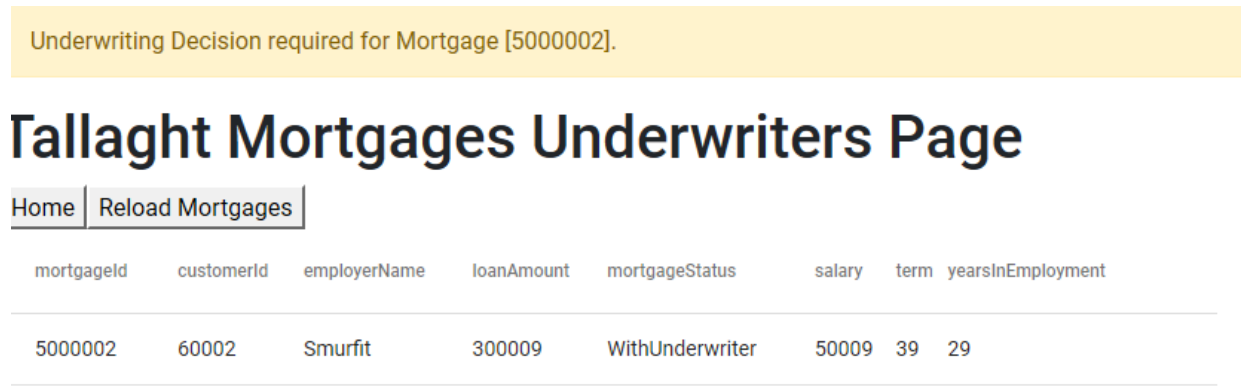


Figure 3.1.4

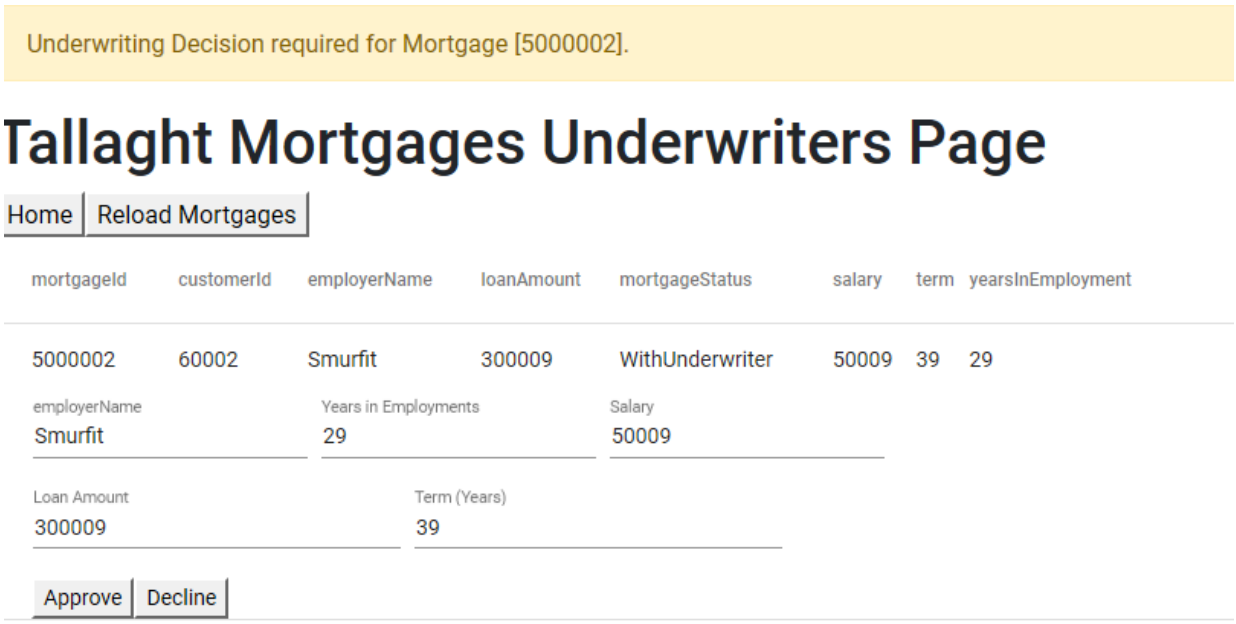

Underwriting Decision required for Mortgage [5000002].
Tallaght Mortgages Underwriters Page
Home
Reload Mortgages
mortgageId
customerId
employerName
loanAmount
mortgageStatus
salary
term
yearsInEmployment
5000002
60002
Smurfit
300009
WithUnderwriter
50009
39
29
employerName
Smurfit
Years in Employments
29
Salary
50009
Loan Amount
300009
Term (Years)
39
Approve
Decline

Figure 3.1.5

STEP FUNCTIONS AND CLOUDFORMATION

Integrate the deployment of the Workflow state machine into the project Cloud formation template.

Also this week I have hoped to integrate the Step Function State Machine definition into the CloudFormation stack template. However after spending 2 days investigating, numerous attempted deployments and restructuring of the template I have not been able to get this working.

To define a State Machine definition in the CloudFormation YAML template it is necessary to define a JSON definition string which is counter intuitive – to specify JSON in a YAML file.

```
Resources:
 MortgagesStateMachine:
 Type: AWS::StepFunctions::StateMachine
  Properties:
   StateMachineName: Tallaght-Workflow
   DefinitionString:
    !Sub
     - |-
      {
       "StartAt": "GetMortgageApplication",
       "States": {
        "GetMortgageApplication": {
         "Type": "Task",
         "Resource": "${GetMorgtageFunctionArn}",
         "ResultPath": "$.lambdaresult",
         "Next": "GetMortgageApplicationResult"
        },
```

The advantages of having the Step Function definition in the CloudFormation template are that we can reference the ARNs of the Lambda Functions during stack deployment. Lambda functions that need to refer to the Step Function ARN could also capture the up to date ARN in Environment variables.

However after a number of hours correctly formatting the State Machine definition in the CloudFormation template, during the deployment I received Circular Dependancy errors due to the dependancies between Lambda Functions, API Gateway and the Step Function definition. This points to a problem I have introduced somewhere. I believe the cause of this problem is the auto generation of the API Gateway Resouce operations within CloudFormation. By using the 'Event' definition on the Lambda functions to create HTTP endpoints, CloudFormation is automatically generating IAM policies/roles which may be required in some part of the definition before they are actually being created. The fix would be to explicity define the HTTP Resources in an API Gateway Resource. However then I would need to add the other Resources in one at a time to find any conflicts. This will just take too long at the moment. The other solution would be to break the Cloudformation template into smaller logically defined areas, e.g. Workflow, Lambda, Database, etc and chain the Outputs and Inputs.

However after many attempts to find a solution (that does not include building the template again from the ground up) I have to acknowledge that I do not have sufficient time. I must focus on extending the State Machine next week to include the ability to request clarifications about the mortgage applications.

Figure 3.1.6

NEXT TASKS

My main priority this week is to implement the Clarification functionality. This will require code changes in the Step Function definition, new Lambda Functions, modifications to existing ones and Angular front end changes. So there is a significant amount of coding still remaining.

If time permits then I will revisit the Circular dependancy error that I am encountering with the CloudFormation Step Function definition.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker to allow questions be submitted and answered regarding the mortgage application or applicant. The Broker must then be able to restart the workflow.
Build / Development	AWS S3 / Angular	Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

REFERENCE LIST AND BIBLIOGRAPHY

[1] AWS Free Tier. https://aws.amazon.com/free/

[2] AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html

[3] AWS CLI SAM. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html

[4] Node JS. https://nodejs.org/en/download/

[5] Configure AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

[6] Visual Studio Code. https://code.visualstudio.com/download

[7] Configure AWS on Visual Studio Code. https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html

[8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/

[9] DynamoDB Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml

[10] Lambda Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml

[11] Step Function State Machine Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt

[12] https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/

[13] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html

[14] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html

[15] https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html

[16] https://docs.aws.amazon.com/step-functions/latest/dg/connect-to-resource.html#connect-wait-token

VERSION 1.0

07/12/2019

ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

IMPLEMENTATION PHASE 3.2

TABLE OF CONTENTS

PROJECT PLAN FOR ITERATION 3.2

The table below (Table A) lists the tasks that I planned to complete for the final implementation deadline.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker.
Build / Development	AWS Lambda	Develop 4 new Lambda functions to support the clarification use cases. These include functions to support CRUD operations and functions for notifications and pause / restart the Step Function workflow.
Build / Development	AWS Angular	Implement new Front end screens to support the clarifications use case.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table A.

PROJECT PROGRESS

This week I completed the following tasks:

- Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker.
- Develop 4 new Lambda functions to support the clarification use cases. These include functions to support CRUD operations and functions for notifications and pause / restart the Step Function workflow.
- Implement new Front end screens to support the clarifications use case.
- I have tested all new state Transitions / Events / Human interaction

Please review the video (MP4) demostrating the final version of this project at the following URL:

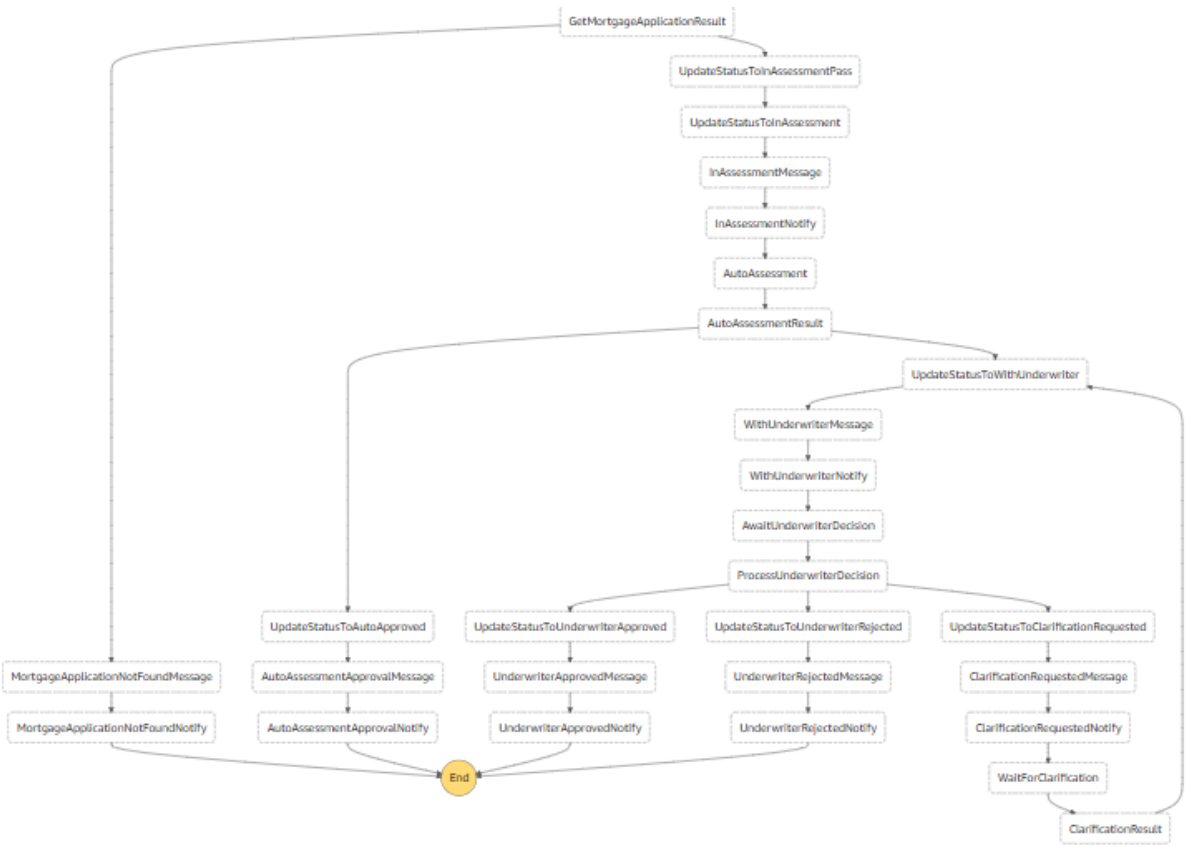
https://s3-eu-west-1.amazonaws.com/appd.cloud.project/IT-Tallaght-Mortgage-Workflow-WebApp-2019-12-08.mp4

Due to time constraints I was unable to refactor the Cloudformation Serverless Application Model template to integrate the Step Function deployment. The current iteration of the CloudFormation tempate includes the packaging and deployment of the following AWS components:

- WebSocket API Gateway Resources
- HTTP Restful API Gateway Resources
- DynamoDB Tables: Mortgages, Customers, Step Function Tokens, Clarifications, WebSocket connections, ID generation.
- Lambda Websocket Functions
- Lambda Mortgage / Customer / Clarification CRUD Functions
- Lambda Step Functions Support Functions
- Lambda SNS Functions
- SNS Topics
- SES Endpoints

STEP FUNCTIONS – CLARIFICATIONS - FINAL VERSION

The final version of my Step Function workflow is shown in Figure AWS.STEP.1. In the past week it have been enhanced to include the branches that allow clarifications between the humans involved in the Mortgage applicatoin process. It allows an unlimited number of question / response clarifications to happen between the Underwriter and the Broker to provide a clear understanding of the financial / personal circumastances of the applicants.



AWS.STEP.1

LAMBDA FUNCTIONS - CLARIFICATIONS

In order to suppor the enhancements to the Workflow that allow Underwriters approve / reject / clarify mortgage applications from the Browser based front end a number of additional Lambda functions are required.

The following Lambda Functions have been added :

- Mortgages-Stack-BrokerClarificationPOSTFunction
- Mortgages-Stack-CreateNotificationMessageFunction
- Mortgages-Stack-AddClarificationFunction
- Mortgages-Stack-BrokerClarificationPOSTFunction
- Mortgages-Stack-DeleteClarificationFunction
- Mortgages-Stack-GetClarificationsFunction
- Mortgages-Stack-WaitForBrokerClarificationFunction

Example Code.

Source Code on Git Hub: https://github.com/GeneralYeager/tallaght.mortgages

ClarificationDB.js ; Utility Function.

```
'use strict';

const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient({ apiVersion: '2012-08-10' });
const { CLARIFICATION_TABLE_NAME } = process.env;

exports.queryClarifications = async function (mortgageId) {

  try {
    var clarificationParams = {
      TableName : CLARIFICATION_TABLE_NAME,
      KeyConditionExpression: "mortgageId = :mortId",
      ExpressionAttributeValues: {
        ":mortId": mortgageId
      }
```

```
    };

    let clarifications = await ddb.query(clarificationParams).promise();
    return clarifications;
  } catch (error) {
    console.log(error);
    return null;
  }
};

exports.addClarification = async function (mortgageId, currMessageId, text) {

 try {
  var newMessage = {
   TableName: CLARIFICATION_TABLE_NAME,
   Item: {
    "mortgageId": mortgageId,
    "messageId": currMessageId + 1,
    "text": text
   }
  };
  const result = await ddb.put(newMessage).promise();
  return result;
 } catch (error) {
   return null;
 }
};

exports.deleteClarifications = async function (clarification) {
 try {
  var params = {
   TableName: CLARIFICATION_TABLE_NAME,
   Key: {
    'mortgageId' : clarification.mortgageId,
    'messageId' : clarification.messageId
   }
  };
  const deleteResult = await ddb.delete(params).promise();
  return deleteResult;
 } catch (error) {
   return null;
 }
};
```

FRONT END CLARIFICATIONS

I have enhanced the front end Angular code to allow brokers and underwriters to interact via a new real time clarification process.

The new screenflow is shown in the Screen shots below.

The Broker submits a Mortgage application for Decision. Figure ANGULAR.1.

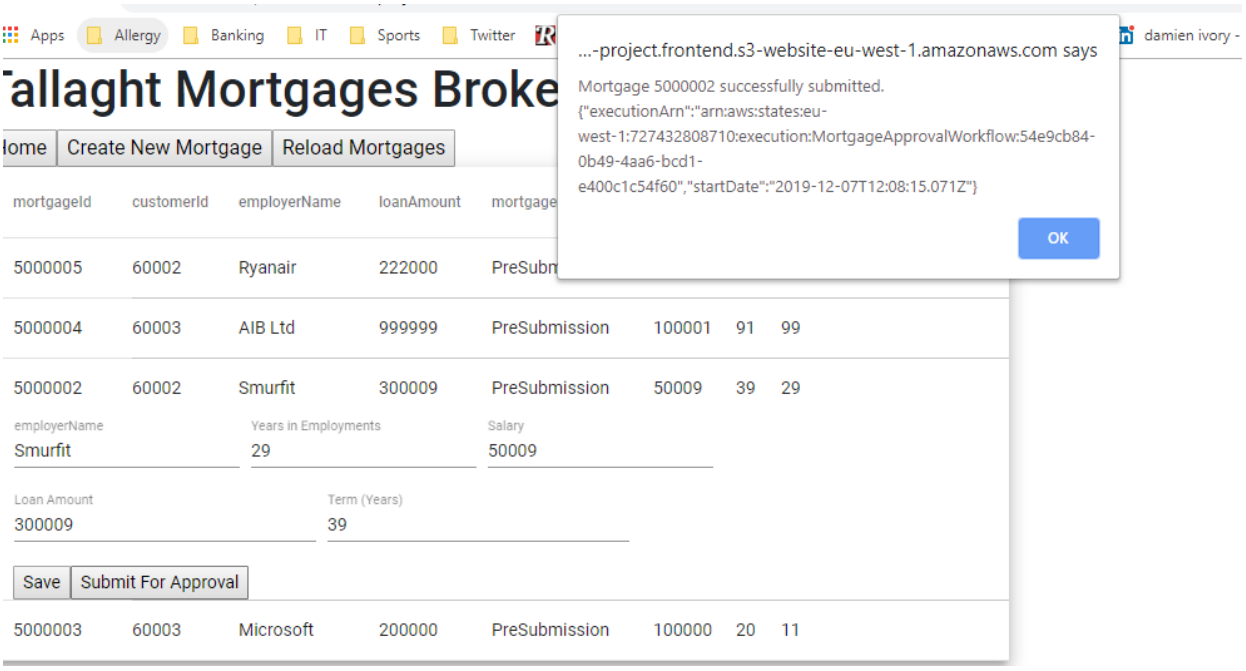


Figure ANGULAR.1.

The Underwriter recieves a notification of the requirement to approve or reject the mortgage application if the application was not automatically passed by the Auto Assessment rules. Figure ANGULAR.2

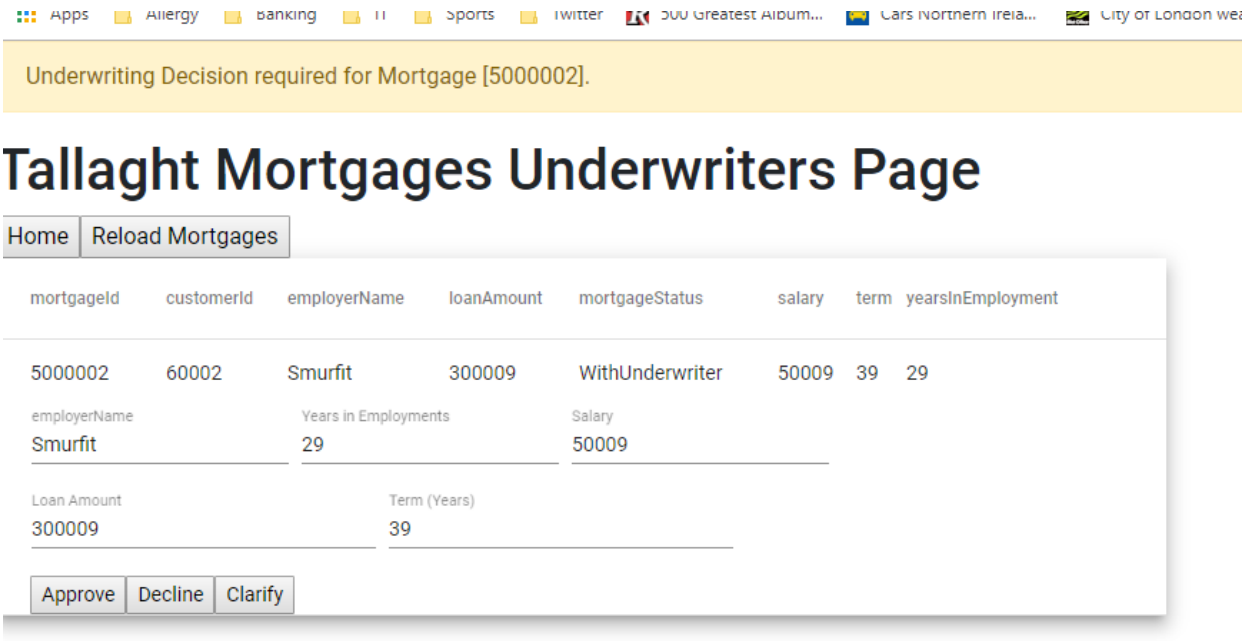


Figure ANGULAR.2.

The Underwriter can input a question or request if they require clarification or action by the Broker that is dealing with the applicants. Figure Angular.3

Figure Angular.3

The Broker will receive notification that the Underwriter has changed the status of the application to 'ClarificationRequested' and that the need to respond or take action regarding documentation. FIGURE ANGULAR.4

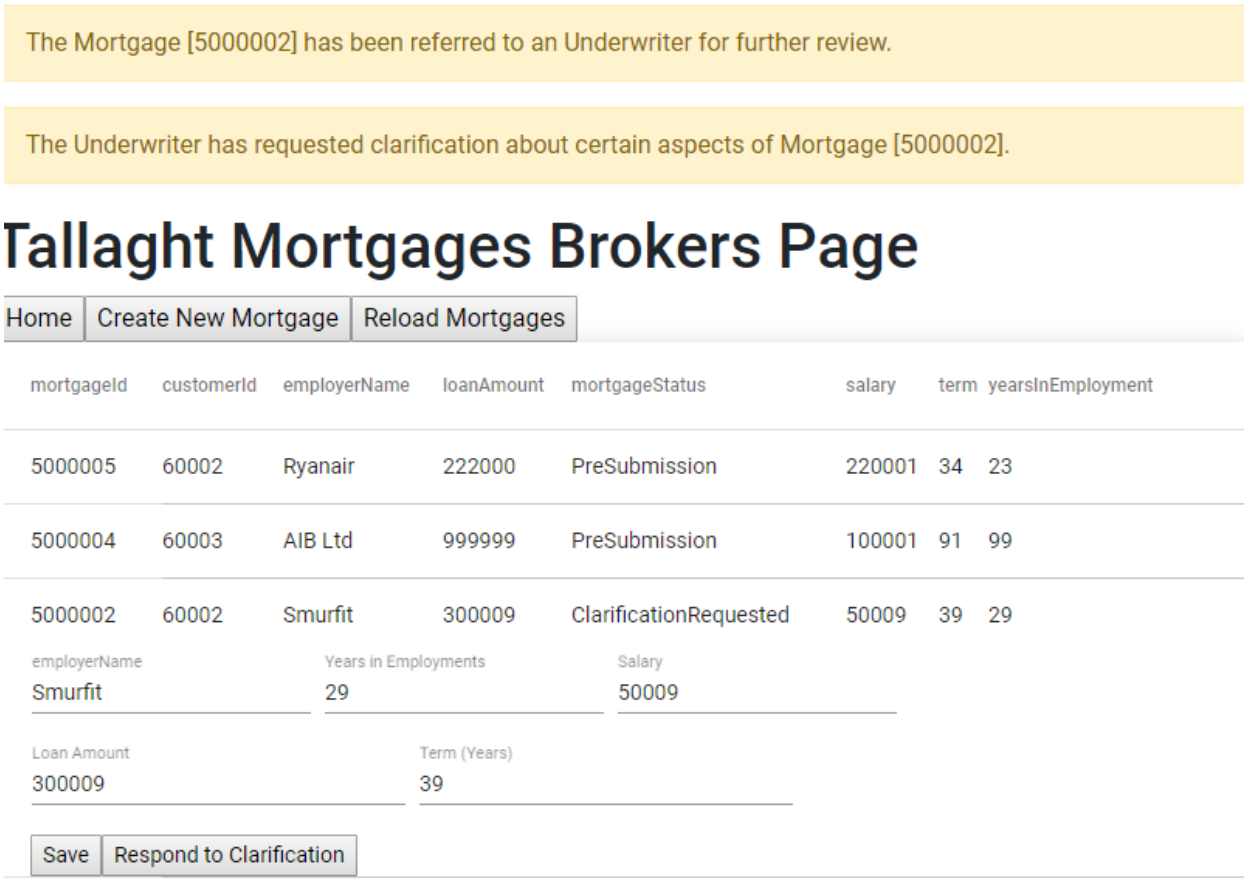


FIGURE ANGULAR.4

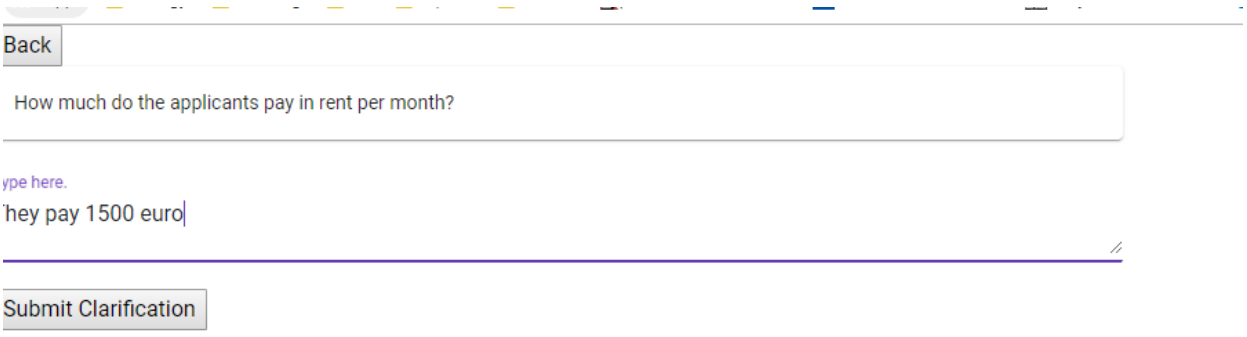


FIGURE ANGULAR.5

Once the Broker has completed the Clarification task and resubmitted the mortgage application the Workflow returns the control to the Underwriter for further clarifications or final assessment. FIGURE ANGULAR.6

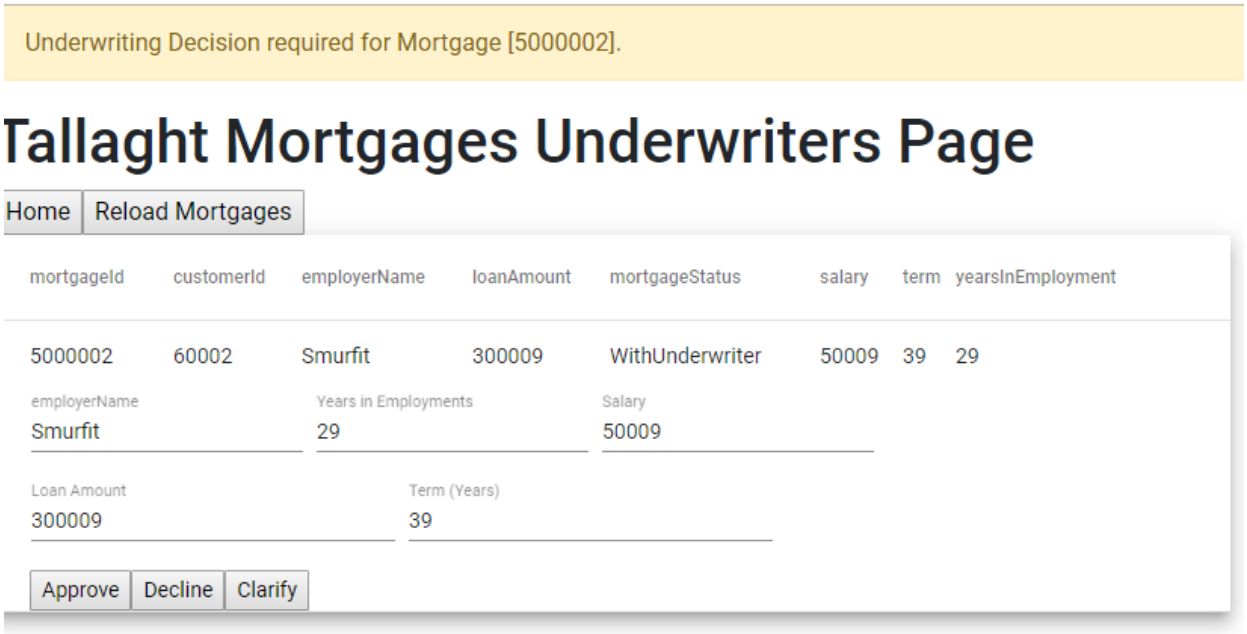


FIGURE ANGULAR.6

The Underwriter can view the Broker feedback. FIGURE ANGULAR.7

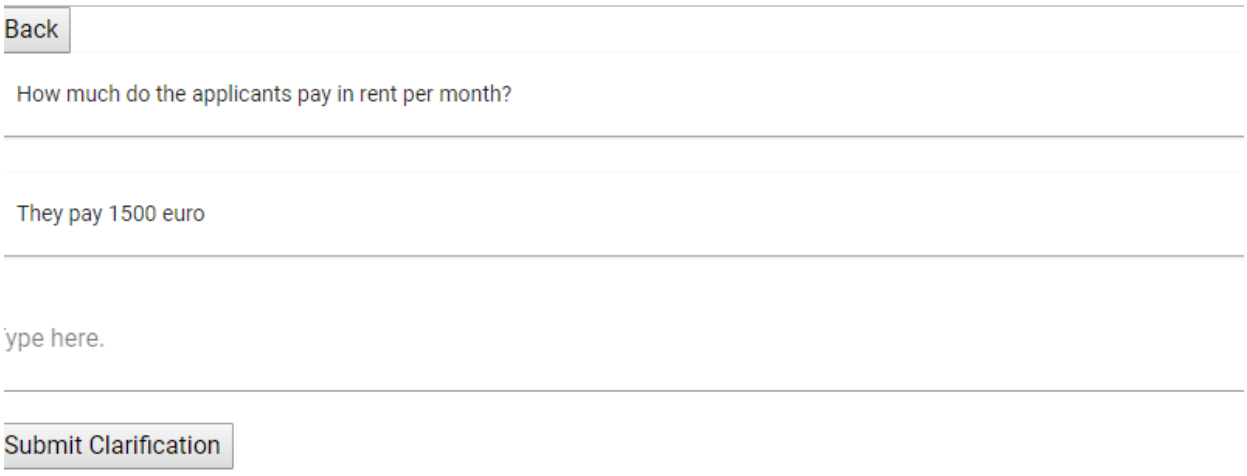


FIGURE ANGULAR.7

In this case the Underwriter has received clarification about the applicant rental payments and decides to Approve the application. FIGURE ANGULAR.8

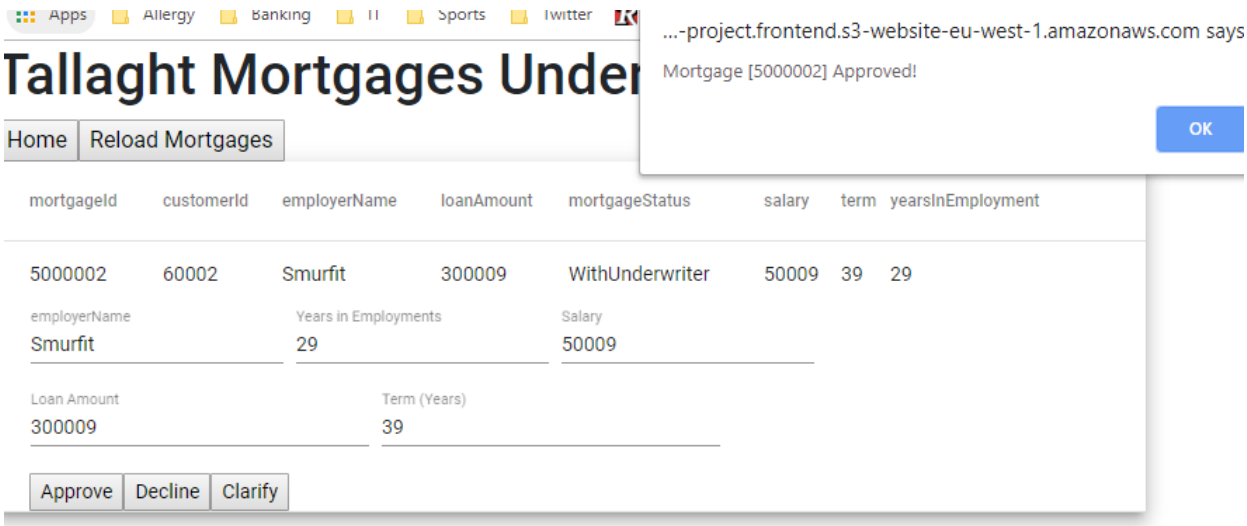


FIGURE ANGULAR.8

Finally the Broker recieves a notification that the Mortgage Application has been Approved. FIGURE ANGULAR.9

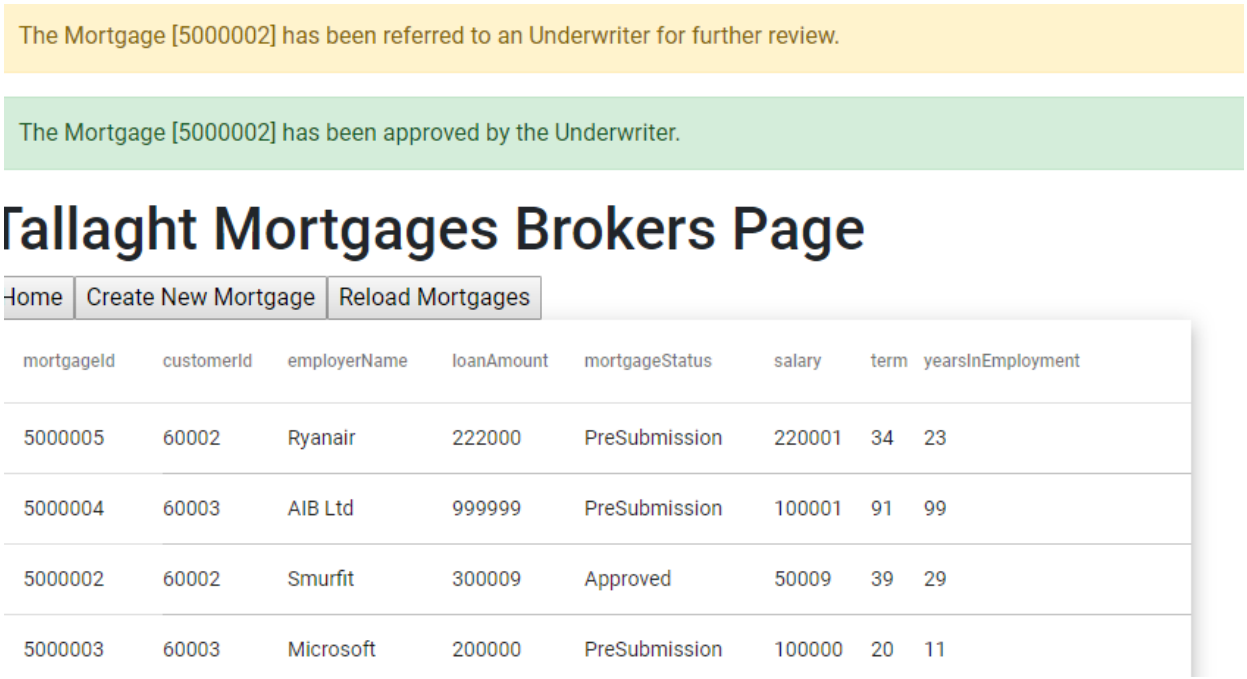


FIGURE ANGULAR.9

REFERENCE LIST AND BIBLIOGRAPHY

[1] AWS Free Tier. https://aws.amazon.com/free/

[2] AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html

[3] AWS CLI SAM. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html

[4] Node JS. https://nodejs.org/en/download/

[5] Configure AWS CLI. https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

[6] Visual Studio Code. https://code.visualstudio.com/download

[7] Configure AWS on Visual Studio Code. https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html

[8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/

[9] DynamoDB Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml

[10] Lambda Cloudformation Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml

[11] Step Function State Machine Template. https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt

[12] https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/

[13] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html

[14] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html

[15] https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html

[16] https://docs.aws.amazon.com/step-functions/latest/dg/connect-to-resource.html#connect-wait-token

CERT.CLOUD.SOLUTIONS FINAL PROJECT CONCLUSION

Serverless architecture is a development paradigm where applications (composed of many small, independent services/components) are run in cloud PaaS services perhaps on containers temporarily created for executing individual functions. Serverless is also referred to as Function as a Service (FaaS).

The advantages are rapid and automatic scaling and the cost advantages of paying only for what you use, i.e. the number of requests and function execution duration.

There are however also significant drawbacks. These include:

- Vendor lock-in: Once you have invested in building the software and the operations processes around a serverless application like AWS Lambda it will be expensive and time-consuming to port to any other vendor. The workaround for this is to create a separate function for your business logic and then call that from a Lambda handler. Although there are some ways to mitigate vendor lock-in, there are none for vendor control.
- Execution Constraints: The temporary containers uses in AWS Lambda result in limits on the amount of memory and execution time. Currently Lambda functions are limited 128 KB for event request body. Memory is limited at 2GB RAM and 5 minutes of execution time. Disk space is also limited to 512 MB. So for functions that may be short lived but need bursts of memory, perhaps to read / transform large files would not be suitable for AWS Lambda. The workaround for limited execution time could be to use a step function (which we discuss later) or SQS to build a sequence of small functions or using Lambda to start and AWS Batch process.
- Cold Starts: AWS Lambda uses temporarily created containers, so in order to execute your function as soon as the client submits the request AWS Lambda must create a temporary container, deploy all dependencies and run the required code. After the request is completed, the container may be destroyed. This process can take between 100 milliseconds to 2 minutes. This Cold Start can cause significant response delays. The workaround is cumbersome in that it requires the developer to try to keep the function in a Warm state for example by pinging the function at regular intervals to prevent AWS Lambda reclaiming the container.

Some of these restrictions may be viewed as enforcing good architectural principles. Long running processess are better suited to Beanstalk, AWS Batch or EC2. Large payload requests in many cases indicates a function / service is trying to do too much. A review of how to refactor the function into more modular, composable services/endpoints should be undertaken. Some of these limitation prevent developers trying to abuse the Serverless architecture by deploying monolithic applications in an unsuitable way.

Lambda is a very versatile managed service but I would suggest that at this point is has a set of suitable problem domains and does not at this point provide a platform that will accomodate all use cases. The most suitable use cases that I am aware of at the moment are listed below.

- AWS Lambda and S3 hosted static websites. Host the web frontend files (html, javascript, images etc) on S3, and use content delivery via CloudFront. Lambda functions can host the logic for the API Gateway HTTP endpoints and persist data to a managed Database. Lambda, API Gateway and S3 pricing is pay only for the traffic, so the only fixed cost will be the database.
- Processing S3 Objects. Using S3 event notifications as triggers for Lambda execution. This solution is very cost effective and scaling is managed by AWS.
- Data Transformation. AWS Lambda is highly scalable which provides an ideal platform for ETL type processing and transfering data between S3, Redshift, Kinesis and database services.

Many user of AWS Lambda have found gaps whenever they need the following features:

- Imposing a sequence on a set of Functions
- Error handling / retry behaviour over that set of Functions
- Choose Function execution based on real time data.
- Parallelism
- Long running tasks

Manage state between stateless functions involve setting up queues and databases. This increases overhead and can be time-consuming and complex. AWS Step Functions provide a managed service that allow Scaling, State managements, error handling and auditing. Step Functions provide state machines that manage workflows of many Serverless functions using extensive JSON declaritive language. They provide a resilient way to iterate though a sequence of AWS services that allows building multi-step applications. It allows the combination of synchronous external invocations such as HTTP API operations via AWS API Gateway with robust asynchronous state machine processing.

The AWS Lambda team restrict the use of recursive Lambda functions because it is very prone to error. An orchestration service like Step Functions shoud be used instead. You can put explicit branching checks in place and enforce timeouts at the workflow level. It helps prevent accidental infinite recursions.

The barrier to entry is cost. Step Fuction pricing at $0.025 per 1,000 executions, is 100x times more expensive per invocation than Lambda. This price level makes it difficult to be cost-effective to replace existing custom Lambda solutions that maintain state.

Step Functions have many useful features for the business process use cases - executions can span 12 months, allow task timeouts and interrupts, configure tasks with heartbeats. However, it would be much easier to adopt if it supported low cost, high-scalable state machines using an event-driven paradigm. Modern software development encourages loosely coupled, event driven design. However, Step Functions Workers (called activities) must poll for tasks, and Lambda invocations by Step Functions are synchronous.

One feature that I think would be very useful is to allow the specification of tasks as an ARN or HTTP endpoint (e.g. asynchronous Lambda, SNS topic, HTTP endpoint, etc.), with a URL provided in the payload for posting back the output. This would allow Step Function state machines to more fully integrate with event-driven architectures. The current Step Function solution to this allows developers to call the waitForTaskToken method from the State Machine task and pause until a developer calls the Step Function SDK using a token to resume the execution. It would be neater if this could be done directly by using a specific execution HTTP endpoint.

Step Functions makes it easy to set up state management early on, and it continues to work well as your application scales and you add more services into the mix. The state machine can be seen as used as a visual construct in the AWS Admin console and is easy to understand. The States Language supports re-try, loop, back-off, timeout, parallel and choice constructs.

However, the lack of tooling / external IDEs may be an impediment to adoption. It is relatively quick to create complex sequences of tasks Orchestrating a sequence individual Serverless functions.

For complex workflows that involve many different states and branching logic, the visual workflow is an intuitive design and diagnostic tool. Developers and DevOps can look at the workflow diagram for running or completed execution and diagnose. It is intuitive to understand the state of the system without knowing the details of its implementation. It is possible because the workflow rule and design decisions in the workflow have been lifted out of the code and made explicit in a visual format that is easy to follow. It is also good practice to decouple workflow logic from business logic. To do otherwise increases the complexity of the applications and reduces their reuse value. In addition, managing state separately from serverless functions allows developers to focus on business logic.

While AWS Step Functions provides solutions to a number of design challenges in the Serverless framework, transitioning all your orchestration layers may not be suitable for all use cases.

The drawbacks of Step Functions include:

- Configuration with the Amazon States Language Amazon States Language is quite complex; its syntax is based on JSON and therefore optimized for machine readability instead of readability by humans. Learning the language is not practical for non-developers. It is also AWS specific and not open source.
- Decoupling services from the orchestration provides scalable, composable functions, but the developers need to learn an entirely separate service.

However, if you need rapid iteration and all your application actions are performed using AWS Lambda functions, AWS Step Functions is an ideal solution. It will reduce the time spent developing orchestration logic and therefore allow you to focus more on your business logic.

Step Functions doesn't support schedules, and AWS Lambda already has an integrated scheduling system. It's better to use Serverless and Lambda scheduling functionality directly.

Step Functions supports more integration with other AWS Services than it was possible for me to investigate in this project. For example it is possible to directly access DynamoDB, SQS, and SNS without writing Lambda functions for that purpose. There are trade-offs in that the Lambda functions would be available for reuse by other components.

Other points to note:

Pricing. For complex workflow, AWS Step Functions is the only available AWS solution but the price is high. AWS even charge for states that only wait or just pass to the next one.

Modern software development favours event driven architecture and AWS services encourage that model. If Step Functions allowed us to initiate execution in response to an event or automatically pull messages from an Amazon SQS Queue.