Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Nimbus: Java Framework Targeting Function-as-a-Service

Author:
Thomas Allerton

Supervisor:
Dr Robert Chatley

Second Marker:
Dr Anthony Field

June 16, 2019

Abstract

This project presents the Nimbus framework, a Java framework specifically designed to write and deploy Function-as-a-Service (FaaS) applications. It makes it simple to define cloud resources and provides clients to interact with these resources. Once the code has been written, Nimbus can deploy the application locally to allow for integration testing and then deploy the same code to a production environment in the cloud without any further changes.

Nimbus solves many of the problems that exist in current FaaS solutions, where large configuration files, prone to human error, are required. A common complaint from FaaS developers is that integration testing is difficult and slow to perform, due to lengthy deployments.

Case studies are presented showing that Nimbus drastically reduces the amount of configuration required, and speeds up the entire deployment cycle, with negligible performance differences in the cloud.

Acknowledgements

I would like to take this opportunity to thank my supervisor Dr Robert Chatley, for all the guidance he has provided throughout the project. His experience proved invaluable when providing feedback and suggestions, and the end result would have been very different without him.

To my family and friends, thank you for all the support you have provided through the years, and for making me many cups of tea.

Contents

1 Introduction **6**

1.1 Serverless and Function-as-a-Service . . . 6

1.2 Objectives . . . 7

1.3 Contributions . . . 8

2 Background Theory **9**

2.1 Cloud Computing Introduction . . . 9

2.2 Function-as-a-Service . . . 9

2.2.1 Highlighted Limitations . . . 10

2.3 Lambda . . . 11

2.3.1 Java Handlers . . . 11

2.4 CloudFormation . . . 11

2.4.1 Issues with Deployment . . . 12

2.5 Existing Frameworks . . . 12

2.5.1 Serverless . . . 13

2.5.2 Spring Cloud . . . 14

2.5.3 Serverless Application Model (SAM) . . . 16

2.6 Function Event Sources . . . 17

2.7 Other Cloud Integration . . . 19

2.8 Annotation Processing . . . 20

2.9 Abstract Syntax Tree . . . 20

2.10 Java Reflection . . . 21

3 Nimbus API **22**

3.1 Deployment API . . . 22

3.1.1 Data Stores . . . 22

3.1.2 Functions . . . 24
3.2 Cloud Resource Clients API . . . 30
3.3 Local Deployment API . . . 30

4 Implementation **32**
4.1 Annotation Processor . . . 32
4.1.1 Annotation Types . . . 32
4.1.2 Overview . . . 33
4.1.3 Resources . . . 34
4.1.4 Functions . . . 36
4.1.5 Permissions . . . 42
4.1.6 Summary . . . 44
4.2 Client Components . . . 44
4.2.1 Implementation . . . 44
4.3 Local Deployment . . . 47
4.3.1 LocalNimbusDeployment Initialisation . . . 47
4.3.2 Local Resources (Unit Tests) . . . 48
4.3.3 Local Functions (Unit Tests) . . . 49
4.3.4 Function Environment . . . 52
4.3.5 Local Clients . . . 53
4.3.6 File Uploads (Local) . . . 53
4.3.7 Local Deployment Servers . . . 54
4.4 Deployment Plugin . . . 55
4.4.1 Creating Shaded JARs . . . 55
4.4.2 Stack Creation . . . 59
4.4.3 File Uploads (Cloud) . . . 60
4.4.4 After Deployment Functions . . . 60
4.4.5 Deployment Optimisation . . . 60

5 Building Applications with Nimbus **61**
5.1 The Application . . . 61
5.1.1 Overall Architecture . . . 62
5.1.2 The WebSocket API . . . 62

5.1.3 The REST API . . . 63
5.1.4 The Client (Frontend) . . . 64
5.1.5 Testing . . . 64
5.1.6 Cloud Deployment . . . 65
5.2 Remarks . . . 65

6 Evaluation **66**
6.1 Performance . . . 66
6.1.1 Handler Performance . . . 66
6.1.2 Client Performance . . . 67
6.1.3 Performance Summary . . . 68
6.2 Development Experience . . . 68
6.2.1 Amount of Code Written . . . 68
6.2.2 Amount of Configuration Written . . . 69
6.2.3 Testing Cycle . . . 70
6.2.4 Vendor Lock-in & Functionality . . . 71
6.2.5 Developer Experience Summary . . . 73
6.3 Local Testing . . . 73
6.3.1 Function Handlers . . . 73
6.3.2 Resources and Clients . . . 73
6.3.3 Deployment Times . . . 74
6.3.4 Supported Resources . . . 74
6.3.5 Conclusion . . . 75
6.4 Deployment . . . 76
6.4.1 Optimised Package Size . . . 76
6.4.2 Effect of Updates . . . 77
6.5 Open Source Evaluation . . . 78
6.6 Community Response . . . 79

7 Conclusion **80**
7.1 Core Achievements . . . 80
7.2 Future Work . . . 81
7.3 Personal Reflections . . . 81

A CloudFormation Template Examples **83**

A.1 File Storage Bucket . . . 83

A.2 Relational Database . . . 84

A.3 Key-Value and Document Stores . . . 86

A.4 Functions . . . 86

A.5 Function Permission . . . 88

A.6 Function Event Source Mapping . . . 89

A.7 REST API . . . 89

A.8 WebSocket API . . . 90

A.9 Queue . . . 91

A.10 Notification . . . 91

Chapter 1

Introduction

The author of this project was first introduced to Functions-as-a-Service while on a placement, tasked to write a Java REST API for sending messages. This API needed to interact with other cloud resources, like a relational database and a NoSQL database. While working on this application, the most time-consuming part and the one that felt most wasteful was trying to successfully deploy code and have it interact with the cloud environment, as opposed to the business logic. The application was deployed, an error revealed, fixed, and then another deployment was performed. This was repeated until the application worked. As deployments took minutes, this was usually an opportune time to get a coffee to drink, but there is only so much coffee someone can handle. The errors that caused the issues ranged from spelling mistakes in the deployment configuration files to incorrect permissions on the functions so that they could not interact with the databases. The deployment was configured in files that grew with the application, and it was difficult to spot errors in these files due to how large they are, and how small the errors are in comparison.

One other pain point was the file size limit on the compiled functions. This limit is set by the cloud provider and defines the maximum file size that can be used for a function. As the application grew, adding more dependencies over time, the file size grew larger and larger as all the functions were compiled together. Eventually, the limit was hit, and a developer had to spend a day and a half refactoring the entire project.

From these experiences, the idea for the Nimbus framework originated: a framework that would speed up the development cycle so that hours were not wasted with trial and error deployments.

1.1 Serverless and Function-as-a-Service

Serverless is a cloud computing architecture where the cloud provider dynamically manages the allocation and provisioning of servers. Serverless systems typically scale based on usage and have a pay-as-you-go model. This model allows developers to focus on the application and not the infrastructure.

Function-as-a-Service (FaaS) is a model of cloud computing and is a prominent implementation of the serverless model, and is also known as serverless computing. The cloud provider, such as Amazon Web Services (AWS), manages the provisioning of the servers that the functions run on. The focus of this project is on AWS, though, all the theory applies to other cloud providers as well. Amazon's FaaS offering is AWS Lambda, where the developer is billed based on how much time the function has spent running. In this model, functions are commonly set up to be triggered by some event, such as an HTTP request or an item being added to a queue.

To use a FaaS offering the developer provides the application source code, which is usually relatively atomic and short running. Cloud resources to trigger the function are then set up, as

well as any that the function will interact with, such as some form of data store.

Current solutions to deploy serverless applications require a configuration file that describes the cloud resources required, though, some are much less verbose than others. As these configurations grow in size with the application, they can become difficult to read when they become massive. As the configuration is decoupled from the associated function code, it can be difficult to determine how a function is deployed when the code is looked at, and difficult to determine what a function does when the configuration file is looked at. These files are also written manually and are prone to errors, for example, spelling mistakes and forgetting parameters. Due to a lack of initial verification, many errors are not found until during the deployment. If an error is found during the deployment, it is reported to the user, and then any progress up to that point rolled back. Additionally, if the file contains more than one error, only the first error is reported. This means that if there are many errors, multiple deployments may be required to discover and fix them all. As deployments usually take minutes, the time taken to fix multiple errors can significantly increase the time taken to deploy an application successfully.

Multiple surveys have been done asking serverless developers what they consider to be the most significant challenges when using FaaS services [1, 2]. One of the most common results was that integration testing is challenging to perform. In most current solutions, integration testing is done by deploying code to a development environment in the cloud and then running tests on that environment. If tests fail, then the code needs to be fixed and deployed again. Due to slow deployments, the testing cycle can be a lengthy process.

Another mentioned issue is vendor lock-in, where the written code is very dependent on a cloud provider. If the user is using AWS Lambda, then AWS likely also provides arguments to the trigger method that provide information about the trigger event. Alternatively, if data is stored, then that storage is likely to be provided by AWS as well. As the APIs for these services will likely be vendor specific, this makes the code vendor specific as well. Cloud-agnostic code is code that does not suffer from vendor lock-in.

The reason that Java was chosen as the supported language is mainly due to the painful personal experience when writing Java serverless applications. Additionally, the most popular scripting languages do not face many of these issues as debugging can be done live in the environment so that deployments are close to instantaneous.

Java functions tend to be larger than functions in other languages [3]. One reason for this is because Java programs have to be compiled as a shaded jar, where the code is bundled with all its dependencies in one jar. This file needs to be kept under the file size limit, and another issue is that the size affects how long the first invocation of the function takes.

1.2 Objectives

The goal of this project is to develop a framework that helps developers write and deploy Java serverless applications. It must:

- Provide a way for developers to deploy the application on a FaaS platform without having to write any additional configuration outside of their source code files.
- Allow developers to write the code in a cloud-agnostic fashion so that they can easily deploy to other cloud providers.
- Provide a way for developers to test their FaaS applications locally so that they can run tests without deploying to the cloud.
- Improve the overall speed of deploying an application.
- Reduce the JAR sizes of functions so that they do not reach the limit.
- Not compromise the performance of the final application.

1.3 Contributions

The result of this project is the Nimbus framework, explicitly designed to write and deploy Java serverless applications. The main features are detailed below.

1. **Annotation-based configuration.** Instead of one large configuration file, functions and resources are defined by placing annotations on classes and methods. This system drastically reduces the amount of configuration required to be written by the user. The use of annotations allows for type checking, catching errors at compile time, which is much faster than existing methods. The API for defining resources is described in Section 3.1 and the implementation discussed in Section 4.1.
2. **Cloud-agnostic code.** The framework allows the user to write cloud agnostic code, with functions being described in a very generic way, as well as providing clients to interact with the cloud resources. The client API is described in Section 3.2 and the implementation discussed in Subsection 4.2.1.
3. **Local Testing Component.** The framework can use the annotations to determine the resources required locally and then simulate them all. The clients that the functions use will now interact with the local resources, which allows for integration testing. The local deployment is created much faster than a deployment to the cloud, so the testing cycle is much faster. The local deployment can be limited to unit tests, or full servers can be spun up. Local deployment API usage is described in Section 3.3 and the implementation discussed in Section 4.3.
4. **Optimised JAR size.** The framework includes a deployment plugin which optimises the size of function JARs by analysing their dependencies. This optimised JAR size helps to reduce cold start times and avoid the function size limit in large projects. The implementation of the deployment plugin is discussed in Section 4.4.

Chapter 2

Background Theory

2.1 Cloud Computing Introduction

Cloud computing entails the accessing of services including storage, applications and servers through the internet. Cloud computing providers, such as Amazon Web Services (AWS), provide these services for a fee. Users of the cloud services can then keep their data and applications in the cloud and not rely on local hard drives or servers.

There are four main types of cloud service models. The first of which is Infrastructure-as-a-Service (IaaS) where the actual infrastructure, like servers, is rented. Here the user manages what OS is used, with all updates, and all the middleware their application needs. The next is Platform-as-a-Service (PaaS), where the user rents a complete development and deployment environment. Here the cloud provider manages the OS and runtime environment. Examples include AWS, Microsoft Azure and the Google Cloud Platform. The next model is Function-as-a-Service (FaaS), a prominent implementation of the serverless model. This model overlaps with PaaS, where now the user only provides application functionality and are oblivious to the actual infrastructure. The cloud provider manages all the capacity management, such as autoscaling based on demand. The final model is the Software-as-a-Service (SaaS) model, where software applications are hosted and then made available to customers over the internet.

2.2 Function-as-a-Service

Function-as-a-Service allows developers to write and deploy units of functionality without having to worry about any of the runtime details. These functions are written to be idempotent, as FaaS should not carry state between any function invocations. Compared to previous cloud computing models, there is far less to manage. For example, in AWS EC2, which has an Infrastructure-as-a-Service model, the user manually manages the provided virtualised operating system. This managing might include runtime dependency management and operating system updates. When using EC2, the level of auto-scaling for the service has to be configured, while using Lambda (AWS's FaaS offering), it is handled all automatically. One significant advantage of using FaaS is that it is charged by use, similar to a pay-as-you-go system, compared to paying a set fee every month to rent out a (virtualised) system [4].

One instance of a serverless function is usually only a small part of a more extensive application, like a single HTTP REST endpoint or a listener to a message queue. These functions can receive input data and in some cases, return output data; they can also have side effects like writing to logs or saving to a database. An event triggers the execution of a function; there are many types of events that can come from many different sources. Some correspond to user input, and some are automatically triggered. Each cloud provider has different a different set of triggers, and many are

unique to the cloud provider. The actual functions can be written in different languages, depending on the provider. Most providers support basic JavaScript (node.js), and some support additional languages like Java, Go and Python.

A serverless function is usually deployed inside of a container [5] which is initialised when the function is triggered. This containerisation also makes it easy for the provider to handle autoscaling as when the demand increases, then new containers have to be spun up. When a function is first invoked, a cold start occurs, which usually takes much longer than a typical request. The reason cold starts take longer is primarily due to the container needing to be initialised, such as preparing any underlying OS and needing to load in the function code. Functions which have running containers, which can then be invoked without causing a cold start, are called warm functions.

Every major cloud provider has FaaS offerings, including AWS Lambda [6], Microsoft Azure Functions [7], Google Cloud Functions [8], and IBM Cloud Functions [9]. These offerings usually provide integration with other cloud resources like API Gateway, database services and authorisation and authentication services. They also typically provide monitoring tools like logs and billing tools to help manage function usage.

In terms of tooling to help developers, each provider typically offers Software Development Kits (SDKs) to aid in the integration of services, like making requests to AWS Dynamo. They also have Command Line Interfaces (CLIs) to aid in the deployment of resources. Additionally, they provide an extensive amount of documentation like SDK class and method definitions, tutorials and example code samples. Most providers also have some form of metric collection, if only at an additional cost like with AWS X-Ray. There also exist third-party tools to aid in serverless development, two main ones being the Serverless Framework [10] and Spring Cloud [11]. These will be discussed in more detail later.

2.2.1 Highlighted Limitations

Multiple surveys have been done asking serverless developers what they consider to be the most significant challenges when using FaaS services [1, 2]. Some of the most common answers were: that testing can be difficult, deployment can be a painful process, that it is challenging to avoid vendor lock-in, and cold start times can be too long.

Currently, the primary method of testing functions is via unit tests. It is hard to do integration testing locally and automatically as there may be many interleaving functions and external systems at play, and it is difficult to replicate the entire system on a local machine. One common practice relies on deploying the system to the cloud in a development environment and testing the system there. This does incur additional costs.

In the surveys, it has been noted that deployment is a pain point, and it has been suggested that developers should be able to use declarative approaches to control what is deployed and tools to support it [12].

Vendor lock-in commonly occurs due to integrating with other services the cloud provider offers and using the provided SDKs, which results in tight coupling.

As mentioned previously, it is the container initialisation period that causes a long cold start. During container spin up the function code has to be read from some form of file storage and then the container loads any extra dependencies that are needed. If it is a large function or has many dependencies, this results in longer spin up times. If using a Java runtime, then the function must be compiled and packaged into a shaded jar containing all the function's dependencies. As the dependencies need to be included in the jar, Java function files are much larger than those of other languages, resulting in worse cold start times. Optimising the packaged size of Java functions improves their cold start times [13].

2.3 Lambda

Lambda is Amazon Web Services' FaaS offering [6]. It supports Java, Go, PowerShell, Node.js, C#, Python, and Ruby code, and provides a Runtime API which allows any additional programming languages to be used to author functions.

2.3.1 Java Handlers

A handler is the entry point of a FaaS offering; the first method invoked when the code is executed. A Java Lambda handler can be written in two ways. The first is by defining a function with InputStream, OutputStream, and Context argument (Listing 2.1). The second is by specifying a serialisable return and input type, as well as a Context parameter (Listing 2.2).

```java
public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context)
        throws IOException {
    int letter;
    while((letter = inputStream.read()) != -1)
    {
        outputStream.write(Character.toUpperCase(letter));
    }
}
```

Listing 2.1: Stream Handler

```java
public Response handleRequest(Request request, Context context) {
    String greetingString = String.format("Hello %s %s.", request.firstName, request.lastName);
    return new Response(greetingString);
}
```

Listing 2.2: Method Handler

The context parameter is an AWS specific class that provides information about the request, for example, a request identifier. In the stream case serialisation and deserialisation is done manually, with the data being read and written in the form of a JSON string. In this case, full control is available to do the serialisation in specific ways, for example, a timestamp into a `DateTime` object. In the second case, AWS will do the deserialisation into the request object, and serialise the response object into JSON. Using the second method seems to be 'faster' than the first, as it appears that the deserialisation/serialisation portion of execution is not run by the function. This speed is measured in terms of billable time, not in total execution time that is visible externally.

Regardless of the method used, the time taken to do the deserialisation and serialisation in a warm function is less than one billable time slice (100 ms). This means that no matter what method is used, only in rare cases will the user be charged differently.

2.4 CloudFormation

CloudFormation [14] is the service provided by AWS that allows for automated deployment of cloud resources. A CloudFormation template defines the resources that will be deployed in the cloud and their properties. The template is simply a JSON file, with sections to define resources and outputs. Resources are cloud resources to be deployed, and outputs provide an API to externally query values generated by the CloudFormation stack. The resource section is a JSON object where names map to resources. The output section is a JSON object where names map to output values. CloudFormation templates have built-in functions that are processed and resolved by AWS when the template is uploaded. These include joining values together (similar to concatenation), getting attributes of resources and getting a reference to a resource. The reference to a resource is the function that is used the most as it is common for one resource to interact or connect to another.

This file can then be uploaded to AWS who will then process it and then create or update the resources one by one. If during resource creation/updates an error is found, any changes are rolled back, and that one error reported back to the user.

2.4.1 Issues with Deployment

For large systems, this deployment is a pain point as it can require configuration of a large number of resources. For example, a simple application could be an event database with a REST API to query it. The database resource would need to be created, plus each of the REST API resources. Separate resources are needed for each HTTP resource and HTTP methods, so with an extensive REST API this adds up. Each function resource also needs to be configured, as well as file storage for the function code. Finally, permissions need to be configured in their own resources so that the functions can interact with the database.

One problem with manually creating the CloudFormation template document for this application is that it can take a long time to write due to the number of resources required. If there are multiple errors in the template, multiple deployments will need to be started to have all the errors reported, with each deployment taking minutes. Spelling mistakes or forgetting parameters are simple errors, which are caught so late in the process as there are very few initial checks when parsing the document. All this together leads to successful deployment taking a long time when done manually, assuming some mistakes are made. Using CloudFormation manually also leads to vendor lock-in as it cannot be used to create resources with another provider.

2.5 Existing Frameworks

There are a few different frameworks that aim to help alleviate some of the issues mentioned previously. For Java, these include Serverless [10], Snafu [15], Spring Cloud [11] and Serverless-Application-Model [16]. A study was done to evaluate the performance of these frameworks to a set of criteria [17]. Here, two are examined in more detail with a running example.

The example to be used is an event system that will have a REST API and will use a key-value store to save the events. An event has a title and a description. This system will be deployed onto AWS, using Lambda functions to handle the requests, DynamoDB as the key-value store, and API Gateway to handle the HTTP requests.

Listings 2.3 and 2.4 show the classes common to both frameworks. The DataModel class (2.4) is the model of the user input expected from the HTTP request. The Jackson library is used to read this object from the string input, converting it from JSON. The ResponseModel class (2.3) is the actual response from the REST endpoint that the end user will receive. They will get this as a JSON response, again converted using the Jackson library.

```java
package models;

public class ResponseModel {
    private boolean success;

    public ResponseModel(boolean success) {
        this.success = success;
    }

    public boolean isSuccess() {
        return success;
    }

    public void setSuccess(boolean success) {
        this.success = success;
    }
}
```

Listing 2.3: ResponseModel Class

```java
package models;

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="EventTable")
public class DataModel {

    private String name = "";
    private String description = "";

    public DataModel() {}

    @DynamoDBHashKey(attributeName = "name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "description")
    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Listing 2.4: DataModel Class

2.5.1 Serverless

The first framework discussed is Serverless. Listings 2.5 and 2.6 show the code specific to this framework. This framework aims to make deploying FaaS services straightforward. To deploy this service using Serverless first the function code is written (2.5). Note here the use of the AWS Java SDK to use the Context, APIGatewayProxyRequestEvent and APIGatewayProxyResponseEvent objects in the handler (required for Lambda functions) and also the use of the SDK to save objects to DynamoDB. After writing the code, a configuration file (2.6) has to be written to define what functions and additional resources are needed.

Remarks

The Serverless framework does make deploying functions easier as they handle much of the AWS CloudFormation template generation. It is also good with functions as the configuration file is cloud-agnostic and makes deploying to other providers easy. However, this is only limited to functions and their triggers like API Gateway, and it does not extend to the key-value store or databases. This limitation is essential to note as database services are one of the most common services used with FaaS [2]. To deploy these resources, the CloudFormation template has to be written inside the Serverless configuration file (2.6, lines 19-31). Writing CloudFormation inside the configuration file is not ideal as the file is now not cloud-agnostic and can lead to vendor lock-in. Another important note is that the code itself is not cloud agnostic as it relies explicitly on the AWS SDKs.

Another issue with Serverless is that creating a configuration file which successfully deploys can still take a long time. Things like pointing the function handler to a class and method (2.6, line 39) can easily be messed up, either by a spelling mistake or not understanding the classpath of a function. As with CloudFormation, these errors are not caught until minutes into deployment or the runtime of the function.

```java
public class DynamoHandler implements RequestHandler<APIGatewayProxyRequestEvent,
     APIGatewayProxyResponseEvent> {

    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent requestEvent,
     Context context) {
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        DynamoDBMapper dynamoMapper = new DynamoDBMapper(client);

        ObjectMapper objectMapper = new ObjectMapper();

        try {
            DataModel dataModel = objectMapper.readValue(requestEvent.getBody(), DataModel.class);
            dynamoMapper.save(dataModel);

            String responseBody = objectMapper.writeValueAsString(new ResponseModel(true));
            return new APIGatewayProxyResponseEvent().withBody(responseBody).withStatusCode(200);
        } catch (Exception e) {
            String responseBody = "";
            try {
                responseBody = objectMapper.writeValueAsString(new ResponseModel(false));
            } catch (JsonProcessingException e1) {
                e1.printStackTrace();
            }
            return new APIGatewayProxyResponseEvent().withBody(responseBody).withStatusCode(200);
        }

    }
}
```

Listing 2.5: Serverless handler class

```yaml
#serverless.yml
provider:
  name: aws
  runtime: java8
  region: eu-west-1
  iamRoleStatements:
  - Effect: "Allow"
    Action:
    - "dynamodb:UpdateItem"
    Resource:
        Fn::GetAtt:
            - EventTable
            - Arn

service: serverless-framework-test

resources:
  Resources:
    EventTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: EventTable
        AttributeDefinitions:
          - AttributeName: name
            AttributeType: S
        KeySchema:
          - AttributeName: name
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1

package:
  artifact:
    target/serverless-test.jar

functions:
  insertToDynamo:
    handler: handlers.DynamoHandler::handle
    events:
      - http: post event/new
```

Listing 2.6: Serverless configuration file

2.5.2 Spring Cloud

The second framework discussed is Spring Cloud. Listings 2.7 to 2.9 show the code specific to this framework. This framework aims to make the code written cloud agnostic, removing the dependencies on the cloud SDKs. Again the code is written, this time making use of the provided

wrappers to remove the dependencies on the AWS objects. The Handler class (2.7) may look strange, as it is an empty class, but it has an important role. This class tells Spring Boot that an API Gateway is going to trigger the function, which will then detect the `@Bean` created in 2.8 and use the code there for the Lambda function. Spring Cloud does not provide the ability to deploy resources, so deployment needs to be done manually, or using another framework, like Serverless.

Remarks

One benefit of Spring Cloud is that it provides the functionality to run some functions locally. Locally running functions is beneficial for testing; unfortunately, only functions are deployed locally. In the running example, the key-value store that the function inserts an item into will be the one deployed in the cloud.

One major problem with Spring Cloud is that it does not remove all the dependencies on AWS code. For example, the DynamoDB SDK is directly required to perform updates in the DatabaseService class, 2.9. Thus the problem of vendor lock-in is still present in the code, as well as all the deployment issues with the Serverless framework.

```java
1 package application;
2 
3 import org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler;
4 
5 public class Handler extends SpringBootApiGatewayRequestHandler {}
```

Listing 2.7: Spring Cloud handler class

```java
1 package application;
2 
3 import org.springframework.boot.autoconfigure.SpringBootApplication;
4 import org.springframework.cloud.function.context.FunctionalSpringApplication;
5 import org.springframework.context.ApplicationContextInitializer;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.support.GenericApplicationContext;
8 import org.springframework.messaging.Message;
9 import org.springframework.messaging.MessageHeaders;
10 
11 import java.util.function.Function;
12 
13 @SpringBootApplication
14 public class Config implements ApplicationContextInitializer<GenericApplicationContext> {
15 
16 
17   public Config() {
18   }
19 
20 
21   @Bean
22   public Function<Message<DataModel>, Message<ResponseModel>> function()  {
23     return message -> {
24       boolean success = new DatabaseService().insert(message.getPayload());
25       return new Message<ResponseModel>() {
26         @Override
27         public ResponseModel getPayload() {
28           return new ResponseModel(success);
29         }
30 
31         @Override
32         public MessageHeaders getHeaders() {
33           return null;
34         }
35       };
36     };
37   }
38 
39   public static void main(String[] args) {
40     FunctionalSpringApplication.run(Config.class, args);
41   }
42 
43   @Override
44   public void initialize(GenericApplicationContext context) {}
45 
46 }
```

Listing 2.8: Spring Cloud Application Config class

```java
package application;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;

public class DatabaseService {

    boolean insert(DataModel input) {
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        DynamoDBMapper dynamoMapper = new DynamoDBMapper(client);

        try {
            System.out.println("Saving input $dataModel to dynamo");
            dynamoMapper.save(input);

            return true;
        } catch (Exception e) {
            System.out.println("Error saving to dynamo! ${e.message}");
            return false;
        }
    }
}
```

Listing 2.9: Spring Cloud DatabaseService class

2.5.3 Serverless Application Model (SAM)

The AWS Serverless Application Model (SAM) [16] is an open-source framework for building serverless applications. It provides shorthand syntax to express functions, APIs, databases, and event source mappings. As it is an AWS framework, it only supports AWS. This framework requires a configuration file, and Listing 2.10 shows the configuration file required for the running example. The code for the handler is the same as in the Serverless case, shown in Listing 2.5. SAM can deploy the application by converting the configuration file into CloudFormation and then using that to deploy. SAM also supports deploying the functions locally, and can also host HTTP servers that can invoke HTTP functions. The functions are hosted in Docker containers that are spun up locally.

```yaml
AWSTemplateFormatVersion : "2010-09-09"
Transform: AWS::Serverless-2016-10-31
Description: A serverless stack using API Gateway, Lambda and DynamoDB
Resources:
  EventTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: name
          AttributeType: S
      KeySchema:
        - AttributeName: name
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 1
        WriteCapacityUnits: 1
  InsertToDynamo:
    Type: AWS::Serverless::Function
    Properties:
      Runtime: java8
      Handler: handlers.DynamoHandler::handle
      CodeUri: ./lambda/put-dynamodb/
      Environment:
        Variables:
          DynamoDBTableName: !Ref "EventTable"
      Events:
        GetApiEndpoint:
          Type: Api
          Properties:
            Path: /event/new
            Method: POST
      Policies:
        - Version: "2012-10-17"
          Statement:
            - Effect: Allow
              Action:
                - dynamodb:PutItem
              Resource: !GetAtt "EventTable.Arn"
```

Listing 2.10: SAM configuration file

Remarks

Similar to Serverless, creating a configuration file which successfully deploys can still take a long time, for the same reasons.

Locally running functions does make testing easier; unfortunately, only functions are deployed locally. Similar to Spring Cloud, the key-value store that the function connects to will be deployed in the cloud. To install SAM, the AWS CLI is needed, which requires python to run, and Docker is needed to deploy the functions locally.

2.6 Function Event Sources

Each cloud provider supports multiple types of event sources for triggering a FaaS service. To provide a cloud agnostic framework, these sources need to be distilled into a more general description that can be applied across all the platforms. Here two platforms are looked at in particular (as their FaaS offerings have Java support). The first is AWS [18]; the second is Microsoft Azure [19].

It is important to note that while this project will usually deploy new resources to trigger the functions, in some cases, it is desirable to support connecting to existing resources. To support this, the configuration options for the triggers should support naming an existing resource (e.g. via an Amazon Resource Name (ARN) in AWS), though this does make the code lose the cloud agnostic benefits.

File Storage

Amazon offers S3, advertised as object storage built to store and retrieve any amount of data from anywhere. Azure offers Blob Storage, advertised as massively scalable object storage for unstructured data.

In AWS, Lambdas can be triggered by S3 bucket events, such as object creation or object-deletion. Azure provides similar functionality in terms of triggering functions when an event occurs in a blob.

Key-Value Store

Amazon offers DynamoDB, a fast and flexible NoSQL database service for any scale. Azure offers Cosmos DB, a globally distributed, multi-model database service that supports document, key-value, wide column and graph databases.

DynamoDB can trigger Lambda functions in response to updates made to a DynamoDB table. Similarly, Azure can trigger functions based on updates and inserts, however not deletes.

Event Streams

Event streams are useful to help provide real-time insight to a business. Amazon provides Kinesis, to easily collect, process, and analyse video and data streams in real time. Azure provides Event Hubs, a big data streaming platform and event ingestion service.

Lambdas can be configured to poll Kinesis streams and process new records like click streams, and similarly, Event Hubs allows functions to respond to an event on a stream. This event is passed into the function as a string.

Notification Services

AWS offers the Simple Notification Service (SNS), which provides fully managed pub/sub messaging for microservices, distributed systems, and serverless applications. Azure has the Notification Hub, advertised as being able to send push notifications to any platform from any back-end.

SNS can trigger Lambdas when a message is published to a topic, and pass the message payload as a parameter. In the documentation, it is noted that this allows Lambdas to be triggered by log alarms and other AWS services that use SNS. Azure does not directly support function triggers from the Notification Hub.

Queue Services

Amazon offers the Simple Queue Service (SQS), providing fully managed message queues for microservices, distributed systems, and serverless applications. Azure offers Service Bus, providing reliable cloud messaging as a service and simple hybrid integration.

SQS enables asynchronous workflows to be built, and AWS Lambda can be configured to poll for these messages as they arrive and then pass the event to a Lambda function invocation. Azure Service bus can trigger a function to respond to messages from a queue or topic.

Scheduled Events

Both AWS and Azure provide methods to trigger functions in a regular, scheduled manner. Both support using CRON expressions.

HTTP Event

Amazon offers API Gateway, where secure APIs can be created and maintained at any scale. These include REST APIs. Azure offers HTTP triggers for functions.

In both cases, REST resources can be configured and can be pointed to a function that will execute when the resource is accessed with the correct method. The function then sends a response back to the user who accessed the resource.

Other

AWS offers a few more event sources that are much less generic and hard to define in general terms, or that Azure does not support. One of these is triggering a Lambda from the Simple Email Service, where a message can be received and trigger a Lambda. Another event source is Amazon Cognito, a simple and secure user sign-up, sign-in, and access control service. Lambdas can also be triggered by CloudFormation events, for example, when a stack updates. AWS CodeCommit, a private git repository host, can invoke Lambdas as well, for example, when a user makes a commit. Skills on Amazon Alexa can be provided by Lambdas, providing new functionality to the voice assistant. A similar trigger is Amazon Lex, which allows developers to build conversational interfaces. Finally, Lambdas can be triggered by IoT buttons, based on the Dash Button hardware.

In terms of this project, these triggers are less attractive as they are less generic and introduce vendor lock-in as Azure does not offer similar alternatives. However, for completeness, these could be tackled in future work.

2.7 Other Cloud Integration

Functions being triggered by events in external cloud resources is not the only form of integration done. Usually, it is desirable during function execution to connect to cloud resources. For example, as serverless is stateless, a database might be connected to for data storage.

As with the triggers, instead of creating a new resource, it may be desirable to connect to existing resources. In this case, the configuration options should support naming an existing resource.

File Storage

Amazon offers S3, and Azure offers Blob Storage. The main parts of the API's include creating containers for objects and uploading and retrieving individual objects.

Key-Value Store

Amazon offers DynamoDB, and Azure offers Cosmos DB.

DynamoDB provides an API that allows for scanning and querying of data based on the primary keys. A query only looks at primary keys and is done based on a hash key and so it very fast. A scan searches through all the items in a table. Items can easily be inserted, removed, or modified.

Azure allows the use of a SQL based language to query, put and update data.

The API provided by Nimbus could be similar to DynamoDB so that it supports a more generic key-value store model. When using Cosmos DB, the framework would translate to SQL queries. Alternatively, an SQL based API could be used, and when using DynamoDB, this would have to be translated into the more constrained query and scan model.

Relational Database

Amazon offers three types of relational databases. The first is Aurora, with support for MySQL and PostgreSQL. Aurora is supposed to be cheaper compared to similar offerings. The second is the Relational Database Service (RDS). RDS supports MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server. The final Amazon offering is Redshift, a fast, simple and cost-effective data warehouse that can extend queries to a data lake. Azure offers multiple relational databases with support for SQL Server, MySQL, PostgreSQL, and MariaDB. Azure also offers a SQL Data warehouse.

Many Java frameworks make using a relational database easier, such as Hibernate and MyBatis. This project would aim to be able to work with these frameworks. If the database already exists and is not part of the deployment, then no changes have to be made, as those frameworks will be configured, as usual. The frameworks are configured by supplying a connection to the database, or the database URL, username, and password so that the framework can obtain the connection.

One issue occurs when the user wants to deploy a new database, as the user might want to configure tables before running a function, and also will not know the URL. This project could support initialising code to solve the configuration issue. This initialising code would be run after deployment completes to initialise any resources as specified by the user. To solve not knowing details such as the connection URL, this could be supplied to a serverless function as an environment variable, and the user would use the provided variables to configure their relational database framework.

In this project, a parameter could be supplied to select the desired querying language. The framework will then deploy a database that supports that language and configures it correctly.

2.8 Annotation Processing

Annotations are a form of metadata that can provide information about a program. They are not part of the program itself [20] and are widely used [21]. Annotations have several uses, one of which is allowing for compile-time processing. This processing lets annotation processors generate additional files and code. In terms of this project, this means that the files required for deployment to cloud providers can be generated at compile time. Another use of annotations is runtime processing, as some annotations are examinable at runtime. This can be used to configure clients by finding the details in the associated annotations, for example, determining names of tables.

One common component when writing an annotation processor is the `Element` class, which represents a program element such as a class, method, or package. This class provides functionality to get enclosing `Elements`, `Elements` that it encloses, and any annotations it may be annotated with.

One advantage of using annotation processing to configure the cloud resources is that the configuration is kept local to the code that will use it. Having an annotation to mark a method as a serverless function, with information about what the event source is and configuration options, means that determining what a method is used for is easy by just looking at it. In comparison, when a single file is used for configuration, determining what a method is used for when looking at is difficult as no information is provided. An advantage of the single file method is that it is easier to see the overall configuration of the whole system. In this project, feedback could be generated to give the user a better idea of their system as a whole before deploying the system to alleviate this issue of using annotations.

One disadvantage of annotation processing is that only the annotations and class and method information can be observed, not the contents of a method. This means permission annotations are needed as an annotation processor cannot detect what the methods connect to.

An example of a serverless framework that uses some annotations is Zappa [22]. Zappa allows functions to be marked with a route that makes it look like a web-server, but is deployed as a serverless function:

```
1 @app.route("/event/{id}")
2 def index():
3     return {"eventId": id}
```

2.9 Abstract Syntax Tree

An abstract syntax tree (AST) is a data structure representing the program structure, with each node in the tree representing some construct in the code.

An alternative to using an annotation processor to configure cloud resources would be to create interfaces that users would implement, and then create an abstract syntax tree using a parser. The abstract syntax tree could be analysed to discover the implementations of the interfaces which could represent specific cloud functions or other cloud resources.

The advantage of this technique is that more analysis can be performed on serverless methods that will be deployed. This analysis could discover what cloud resources the function interacts with, from which permission information can be configured automatically. Another advantage is that the reliance on extending interfaces will result in a strict code format and allow the standard Java compiler to highlight any errors.

One way to generate the AST would be to write a custom parser and then bundle the deployment into a separate application. This method has the advantage of allowing full control of the AST, but writing a custom parser would be a significant investment of time, and users would prefer to have a more integrated solution.

A much better alternative is to use existing parsers; one example is the Eclipse Abstract Syntax Tree [23] for Java. To gain access to the generated AST an Eclipse IDE [24] plugin could be written that could then generate the deployment templates, and then proceed to deploy.

If a specific IDE plugin was written, a disadvantage is that users are required to use that specific IDE. The code could be written in a way that makes it easy to swap AST implementations, for example, by using the adapter pattern, which would alleviate this issue. Another disadvantage is that it is difficult for users to provide specific configuration of cloud resources.

2.10 Java Reflection

Reflection is a feature of Java that allows an executing program to examine and manipulate the internal properties of itself. Reflection makes it possible to inspect classes, fields and methods at runtime without needing to know the specifics at compile time. Reflection can be used to instantiate new objects, invoke methods, and set fields of objects. At runtime, reflection can be used in conjunction with annotations to find all the elements annotated with specific annotations, which can then be further processed. An example of using reflection to dynamically load in a class, and then find its fields can be found in Listing 2.11. One problem with reflection is that it drastically increases the complexity of static analysis of the code, due to details not being available at compile time [25]. The means that determining the dependencies of a Java class is difficult in cases where reflection is used.

```
1 public void givenClassName_whenCreatesObject_thenCorrect(){
2     Class<?> clazz = Class.forName("com.example.java.reflection.ReflectionClass");
3     Field[] fields = clazz.getDeclaredFields();
4 
5     for (Field field : fields) {
6         //Some Processing
7     }
8 }
```

Listing 2.11: Using reflection to inspect a class' fields

Chapter 3

Nimbus API

This chapter details the usage of the Nimbus APIs, for deployment, cloud integration, and testing. The deployment section details all the cloud resources that Nimbus supports, as well as other useful deployment features.

3.1 Deployment API

Nimbus supports the deployment of many different cloud resources. This section details the supported resources.

Nimbus supports deploying to different stages, for example, a stage for development testing in the cloud and the user-facing environment. Stages let the user have different configurations, like different pricing tiers, per stage. The user can specify resources not to deploy to some stages. In Nimbus, this is configured with a stage parameter for all resources, and repeating annotations.

3.1.1 Data Stores

Due to the stateless nature of function-as-a-service offerings, some form of data store is usually deployed with a function so it can keep some state.

File Storage Bucket

File storage buckets allow files of any type to be easily stored. These can also be configured as static websites that are exposed to the public. A class is annotated with `@FileStorageBucket`, specifying an id, to declare a bucket. Listing 3.1 shows a basic example of a file storage bucket.

```
1 @FileStorageBucket(bucketName = "imagebucket")
2 public class ImageBucket {}
```

Listing 3.1: Basic File Storage Bucket Example

Setting the static website flag to true defines a file storage bucket as a static website. The website has an index file and error file that defaults to `index.html` and `error.html`, respectively. These can be changed with the fields in the annotation. The index file is returned when the "/" path is requested. Whenever an error occurs, for example, a file is not found, then the error page is returned.

To easily upload files to the bucket as part of a deployment, the `@FileUpload` annotation can be used. A single file path, or directory path, can be specified and will be uploaded after all

resources have been deployed. The content-type used for a file when hosted in a static website is determined automatically by looking at the file uploaded. This file type detection means that if an HTML file is uploaded, then it is likely to be hosted with a content-type of `text/html`, however, is not guaranteed to work. If it does not work, the default value is `text/plain`. Listing 3.2 shows an example of a file storage bucket configured as a static website.

```java
@FileStorageBucket(
        bucketName = Website.WEBSITE_BUCKET,
        staticWebsite = true
)
@FileUpload(bucketName = Website.WEBSITE_BUCKET,
            localPath = "src/website",
            targetPath = "",
            substituteNimbusVariables = true)
public class Website {
    public static final String WEBSITE_BUCKET = "NimbusExampleWebsite";
}
```

Listing 3.2: Static Website File Storage Bucket Example

The website hosted may also want to connect to other resources created by the deployment, like a REST API, a WebSocket API, or even another file storage bucket. The URLs required to connect to these are not available until a deployment is completed, so these cannot be inserted manually without doing an initial deployment. To solve this problem, Nimbus can perform variable substitution to insert these URLs to uploaded files.

Document Store

A document store allows objects to be stored and are indexed by a key field inside of the object, similar to how a Java `Set<T>` works, where the comparison is made on the key field. A document store is declared by annotating a class with `@DocumentStore`. Fields that are to be stored in the table must be annotated. One field must be specified as the primary key and annotated with `@Key` and the other fields annotated with `@Attribute`. Listing 3.3 shows a basic document store.

```java
@DocumentStore
public class UserDetail {

    @Key
    private String username = "";

    @Attribute
    private String fullName = "";

    // Getters and setters, constructors, etc.
}
```

Listing 3.3: Document Store Example

Key-Value Store

A key value store allows objects to be stored, where the objects are indexed by an external key, similar to how a Java `Map<K, V>` works. Declaring a key-value store is done by annotating a class with `@KeyValueStore`, and a key type specified. Fields that are to be stored in the table must be annotated with `@Attribute`. Listing 3.4 shows a simple key-value store.

```java
@KeyValueStore(keyType = String.class)
public class ConnectionDetail {

    @Attribute
    private String username = "";

    @Attribute
    private int age = 0;

    // Getters and setters, constructors, etc.
}
```

Listing 3.4: Key-Value Store Example

Relational Database

A traditional relational database can be specified by using the @RelationalDatabase annotation, specifying a database language, size and login credentials. Listing 3.5 shows a simple relational database configuration.

```
@RelationalDatabase(
        name = "NimbusExampleDatabase",
        username = "master",
        password = "387tyiehfjkd7f6s8",
        databaseLanguage = DatabaseLanguage.MYSQL
)
public class ProjectDatabase {}
```

Listing 3.5: Relational Database Example

The database performance can be configured using the databaseSize parameter.

The user will likely not want to expose the database password inside of the code base. The password can be set using environment variables to solve this privacy issue, as in Listing 3.6. In this example, at compile time, the password will be taken from the DATABASE_PASSWORD environment variable.

```
@RelationalDatabase(
        name = "NimbusExampleDatabase",
        username = "master",
        password = "${DATABASE_PASSWORD}",
        databaseLanguage = DatabaseLanguage.MYSQL
)
public class ProjectDatabase {}
```

Listing 3.6: Secure Relational Database Example

Having to manage the database schema in serverless projects can be an issue. In a more traditional application, the schema could be created/updated on application startup, and as the application is left running, it does not hinder performance. In a serverless system where no state is assumed then either the schema is managed manually, running migration scripts separately from the deployment, or every time a serverless function is run it checks if the database schema needs to be created/updated. The former can easily lead to issues, for example, forgetting to deploy or other user error, or the function downtime delay where the function and database versions are different. The latter has significant performance downsides. The @AfterDeployment annotation was created to alleviate this issue. The annotation marks a method to be run by the deployment component after a deployment has completed. The user can then annotate a method to perform database migration scripts as soon as a deployment has been completed.

3.1.2 Functions

In Nimbus, to deploy a function, a method in the code is annotated with one of the function annotations. All these annotations will deploy the code to a FaaS offering but differ in how the function will be triggered.

It is important to note that all serverless functions must be declared in a class with a default constructor, as the class needs to be created at runtime, and the framework cannot correctly instantiate it if there are parameters.

All function annotations support memory and timeout parameters, allowing the user to configure the amount of memory the function runs with, and how long the function is allowed to run for before timing out.

HTTP Functions

HTTP functions are triggered when an HTTP request is made to a configurable endpoint. To create an HTTP function, a method is annotated with @HttpServerlessFunction and provided a

unique path and method. This function will be triggered when a request is made that matches this path and method. Listing 3.7 shows a simple HTTP function example. A GET request would be sent to `BASE_URL/example/path` to trigger this example.

```
class RestHandlers {

    @HttpServerlessFunction(method = HttpMethod.GET, path = "example/path")
    public String helloWorldExample() {
        return "HELLO_WORLD";
    }

}
```

Listing 3.7: Basic HTTP Function Example

An HTTP serverless function method can have at most two parameters. One of these is a `HttpEvent` type, which contains the header, path and query parameters. The second method parameter available is a custom user type which will be read and deserialised from the JSON body of the request. For example, `String` can be used to read the body in as a string.

In the `@HttpServerlessFunction` 'path' parameter, HTTP path parameters can be defined by including them inside curly braces, for example, `path = "event/{id}"`.

The default behaviour of HTTP functions is to return a 200 status code if the method returns successfully, and with the serialised return value set as the response body. If the method throws an exception, then the function will return a 500 status code, stating a server error. In Listing 3.7 a String return type is specified. If an HTTP request is sent to this function, the response will be a JSON string `"HELLO_WORLD"`.

Listing 3.8 shows a more complicated examplewith more complex input and output objects. When this endpoint receives a request, it will respond with a JSON body of the person in the request and with a status code of 200.

```
class RestHandlers {

    @HttpServerlessFunction(method = HttpMethod.POST, path = "person/{id}")
    public String echoPerson(Person person, HttpEvent event) {
        String log = event.getQueryStringParameters().get("log");
        String id = event.getPathParameters().get("id");
        if (log != null && log.equals("true")) {
            System.out.println(person);
        }
        return person;
    }

    class Person {
        public String firstName;
        public String lastName;

        @Override
        public String toString() {
            return "Person{" +
                    "firstName='" + firstName + '\'' +
                    ", lastName='" + lastName + '\'' +
                    '}';
        }
    }
}
```

Listing 3.8: Complicated HTTP Function Example

It is likely in some cases that a user will want to control the status codes of the response and other response parameters. The return type of the method can be set to `HttpResponse` to allow for this control. This class allows the user to set the status code and headers of the response, with functions to allow setting the body from a String or a JSON serializable class.

WebSocket Functions

WebSocket functions enable the creation of a WebSocket API for real-time communication with clients. A method is annotated with `@WebSocketServerlessFunction`, with a topic specified, to create a WebSocket function. When a message is sent to this topic via the WebSocket

protocol, this function will be triggered. There are three important reserved topics: `$default, $connect, and $disconnect`. Functions with these topics will be triggered when no topics are matched, when a client first connects, and when a client disconnects, respectively.

A WebSocket serverless function method can have at most two parameters. One of these is a `WebSocketEvent` type, which contains the header and query parameters, as well as the connection ID. The second method parameter available is a custom user type which will be read and deserialised from the JSON body of the request. Listing 3.9 shows an example of a basic WebSocket function.

```
class WebSocketHandlers {

    @WebSocketServerlessFunction(topic="newUser")
    public void newUser(NewUser newUser, WebSocketEvent event) {
        String connectionId = event.getRequestContext().getConnectionId();
        System.out.println("ConnectionId " + connectionId + " registered as new user: " + newUser);
    }

    class NewUser {
        public String username;
        public String password;
    }
}
```

Listing 3.9: WebSocket Function Example

Document/Key-Value Store Function

A document/key-value store function is triggered whenever a document/key-value store is changed, whether that is an insert, delete, or update. A method is annotated with `@DocumentStoreServerlessFunction` or `@KeyValueStoreFunction`, with a parameter defining the store it is associated with, and what kind of change it triggers it.

There are three cases for the parameter configuration of the methods. If it is an INSERT function, i.e. adding a new item triggers the function, then two parameters are expected, the new item and the store event (Listing 3.10). If it is a REMOVE function, i.e. deleting an item triggers the function, then two parameters are expected, the old item and the store event (Listing 3.11). If it is a MODIFY function, i.e. replacing an item triggers the function, then three parameters are expected, the old item, the new item and the store event (Listing 3.12).

```
public class StoreHandlers {
    @DocumentStoreServerlessFunction(
            dataModel = UserDetail.class,
            method = StoreEventType.INSERT
    )
    public void newItemDocument(UserDetail newItem, StoreEvent event) {
        System.out.println("New user was created! " + newItem);
    }
    @KeyValueStoreServerlessFunction(
            dataModel = ConnectionDetail.class,
            method = StoreEventType.INSERT
    )
    public void newItemKeyValue(ConnectionDetail newItem, StoreEvent event) {
        System.out.println("New connection was created! " + newItem);
    }
}
```

Listing 3.10: Insert Store Function Example

```
public class StoreHandlers {

    @DocumentStoreServerlessFunction(
            dataModel = UserDetail.class,
            method = StoreEventType.REMOVE
    )
    public void newItem(UserDetail oldItem, StoreEvent event) {
        System.out.println("User was deleted! " + oldItem);
    }

    //Similarly for a key value store function
}
```

Listing 3.11: Remove Store Function Example

```java
public class StoreHandlers {

    @DocumentStoreServerlessFunction(
            dataModel = UserDetail.class,
            method = StoreEventType.MODIFY
    )
    public void newItem(UserDetail oldItem, UserDetail newItem, StoreEvent event) {
        System.out.println("User was changed from " + oldItem + " to " + newItem);
    }

    //Similarly for a key value store function
}
```

Listing 3.12: Modify Store Function Example

Notification Function

A notification function is one which will be triggered whenever an item is posted to a notification topic. There can be multiple listeners to the topic, each which will be triggered when a new item is posted. To declare a notification function a method is annotated with `@NotificationFunction`, providing a topic name.

A notification function can have at most two parameters. One is a custom user type that is deserialised from the message in the notification; the second is the `NotificationEvent` parameter. This parameter provides details about the notification, like the timestamp and message ID. Listing 3.13 shows an example of a notification function.

```java
public class NotificationHandler {
    @NotificationServerlessFunction(topic="newParty")
    public gotNotification(Party newParty, NotificationEvent event) {
        System.out.println("Got new party notification " + newParty);
    }
    class Party {
        public String name;
        public String location;
    }
}
```

Listing 3.13: Notification Function Example

Queue Function

Adding an item to a queue can trigger an associated queue function. Unlike a notification topic, consumers compete with each other, which means that only one consumer will ingest the new item(s). In contrast, adding a new item to a notification topic notifies all consumers.

To define a queue function, a method is annotated with `@QueueServerlessFunction`, specifying a queue id and a batch size. This batch size specifies the maximum amount of items an invocation of the serverless function will contain (not necessarily the annotated method).

The annotated method can have at most two parameters. There are two cases: if the batch size is one, or more than one. If the batch size is one, then one parameter is a custom user type, deserialised from the queue message, and one is of type `QueueEvent`. Listing 3.14 shows an example of a queue with a batch size of 1.

```java
class QueueFunction {
    @QueueServerlessFunction(id = "messageQueue", batchSize = 1)
    public void consumeQueue(QueueItem item, QueueEvent event) {
        if (item.priority > 5) {
            System.out.println("GOT HIGH PRIORITY MESSAGE " + item.messageToProcess);
        }
        /* Additional Processing */
    }
    class QueueItem {
        public String messageToProcess;
        public int priority;
    }
}
```

Listing 3.14: Basic Queue Function Example

If the batch size is more than one, then one parameter is a custom user type, again deserialised from the queue message, and one is of type `QueueEvent`. Alternatively, one parameter is a `List` of custom user types and the second is a `List` of `QueueEvents`. In the first case, Nimbus calls the function multiple times with each parameter pair. In the second, the list of parameters and events are passed directly to the function. Nimbus guarantees that both lists are of the same size and that index i in both correspond to the same item. This case allows the function to aggregate the data if desired. Listing 3.15 shows examples of queue functions with a batch size of more than one.

```
class QueueFunction {

    @QueueServerlessFunction(id = "messageQueue", batchSize = 10)
    public void consumeQueue(QueueItem item, QueueEvent event) {
        if (item.priority > 5) {
            System.out.println("GOT HIGH PRIORITY MESSAGE " + item.messageToProcess);
        }
        /* Additional Processing */
    }

    @QueueServerlessFunction(id = "messageQueue", batchSize = 10)
    public void consumeQueue(List<QueueItem> items, List<QueueEvent> events) {
        boolean foundHighPriority = false;
        StringBuilder highPriorityMessages = "";
        for (QueueItem item : items) {
            if (item.priority > 5) {
                foundHighPriority = true;
                highPriorityMessages += item.messageToProcess + " ";
            }
        }
        if (foundHighPriority) System.out.println("Found high priority messages " +
     highPriorityMessages);
        /* Additional Processing */
    }

    class QueueItem {
        public String messageToProcess;
        public int priority;
    }
}
```

Listing 3.15: Batched Queue Function Example

File Storage Function

A file storage function is triggered whenever a File Storage Bucket either has an item created or deleted. Defining a file storage function is done by annotating a method with `@FileStorageServerlessFunction`, specifying a bucket name and an event type.

The method can have at most one parameter, a `FileStorageEvent`. This event class contains the key of the file in the file storage bucket and the size of the file. Listing 3.16 shows an example of a simple file storage function.

```
class FileStorageHandlers {
    @FileStorageServerlessFunction(bucketName = "ImageBucket", eventType = FileStorageEventType.
     OBJECT_CREATED)
    public void newObject(FileStorageEvent event) {
        System.out.println("New file added: " + event.getKey() + " with size " + event.getSize() + "
     bytes");
    }
}
```

Listing 3.16: File Storage Function Example

After Deployment Function

After deployment functions are functions that are run once after a deployment has been completed and are useful as they can be used to insert constant data into stores, for example, they can be used to set up a test user account or create a schema for a database. After deployment functions are deployed and run in the cloud.

Annotating a method with `@AfterDeployment` declares it as an after deployment function. No parameters are required. Listing 3.17 shows a simple example that creates a notification subscription.

```
1 class AfterDeploymentFunction {
2
3     @AfterDeployment
4     @UsesNotificationTopic(topic = "FileUpdates")
5     public String addSubscription() {
6         String id = notificationClient.createSubscription(Protocol.EMAIL, "admin@nimbusframework.com"
          );
7         return "Added subscription with ID: " + id;
8     }
9 }
```

Listing 3.17: After Deployment Function Example

Basic Function

A basic function does not rely on any other cloud resource to be triggered. It can be invoked manually from other functions, possibly returning some data, or triggered on a cron schedule.

Defining a basic function is done by annotating a method with `@BasicServerlessFunction`. The method can have at most one parameter, which is a custom user type. If the method is triggered on a cron schedule, then it can have no parameters. Listing 3.18 shows a basic function that can be triggered manually.

```
1 class BasicHandler {
2
3     @BasicServerlessFunction
4     public long getCurrentTime() {
5         Calendar cal = Calendar.getInstance();
6         return cal.getTimeInMillis();
7     }
8 }
```

Listing 3.18: Basic Function Example

Environment Variables

Environment variables can be used to provide functions with external values and variables that can change in different stages.

A function annotated with any of the `@ServerlessFunction` annotations can be annotated with the `@EnvironmentVariable` annotation. This annotation has the key and the value of the desired environment variable specified. The variable can then be accessed using the EnvironmentVariableClient. Listing 3.19 shows an example of a basic function with some environment variable. In this case, the production value will be set by looking for the PRIVATE_PROD_URL in the local machine environment variables.

```
1  public class EnvironmentVariableExample {
2
3      @BasicServerlessFunction(cron="rate(1 day)", stages={"dev", "prod"})
4      @EnvironmentVariable(key="EXTERNAL_URL", value="http://example-dev-url.com", stages={"dev"})
5      @EnvironmentVariable(key="EXTERNAL_URL", value="${PRIVATE_PROD_URL}", stages={"prod"})
6      public void connectToExternalService() {
7          EnvironmentVariableClient client = ClientBuilder.getEnvironmentVariableClient();
8          String url = client.get("EXTERNAL_URL");
9          /* Some processing ... */
10     }
11 }
```

Listing 3.19: Environment Variables Example

3.2 Cloud Resource Clients API

In the previous section, it was detailed how to create cloud resources and deploy cloud functions. One useful feature is being able to interact with the deployed cloud resources, like document stores and key-value stores. This interaction within functions allows for much more complex and integrated applications.

Below is a list of clients supported by Nimbus:

- Document Store Client - Access/save/remove items from document stores.
- Key-Value Store Client - Access/save/remove items from key-value stores.
- File Storage Bucket Client - Access/save/remove items from file storage buckets.
- WebSocket Management Client - Send messages to connected clients via WebSocket
- Basic Serverless Function Client - Invoke basic functions
- Notification Topic Client - Add subscribers and send notifications to a topic
- Queue Client - Add items to a queue
- Relational Database Client - Get a Connection object for a database
- Environment Variable Client - Access environment variables

The `ClientBuilder` class is used to obtain one of these clients. Listing 3.20 shows how to obtain a document store client.

```
1 DocumentStoreClient<DocumentClass> client = ClientBuilder.getDocumentStoreClient(DocumentClass.class)
```

Listing 3.20: Client Builder Example

The reason that the client builder is required is that it substitutes different clients depending on if the code is running locally or in the cloud, which allows the same code to be run in the cloud and run locally with no changes made.

If a serverless function needs to use one of the clients, then an additional annotation, corresponding to the client needs to be added to handle cloud permissions. For example, if a function wants to use a document store client, then the method needs to be annotated with `@UsesDocumentStore`.

3.3 Local Deployment API

Nimbus local deployments simulate the cloud environment locally, so that defined resources in the project can be interacted with. This allows users to ensure that functions interact correctly with the cloud environment. This simulation includes running all after deployment functions and file uploads locally. If file upload variable substitution is used, then local values will be substituted instead of the cloud ones.

For example, a unit test could ensure that a function correctly stores data in a store when triggered with the correct information.

For more than unit testing, full deployment servers can be spun up locally, allowing the user to interact with HTTP functions, static websites and WebSocket functions. The local deployment is useful when there is a front-end for the application that can be configured to point to the local deployment. This means that testing can be done using the front-end and the local deployment, allowing for even more integration testing.

A local deployment is done by initialising a `LocalDeployment` object, as shown in Listing 3.21. Once this object has been obtained, the deployment is running and can be interacted with. The `LocalDeployment` object provides the functionality to interact with all mocked resources, such as to send mocked HTTP requests, add items to stores, and obtain a representation of a serverless function.

```
class NimbusTest {

    @Test
    public void testFunctionPutsItemIntoStore() {
        LocalNimbusDeployment localNimbusDeployment =
                LocalNimbusDeployment.getNewInstance(Test.class);

        // Code to interact with resources and assert that
        // functions interact correctly.
    }
}
```

Listing 3.21: Initialising Local Deployment

Chapter 4

Implementation

Nimbus currently only supports Amazon Web Services (AWS); therefore, all the details for the implementation are specific for AWS. However, all the names of classes and interfaces are designed to be cloud agnostic so that new cloud providers can be easily added in the future.

There are four main components of Nimbus:

1. The annotation processor, which creates a CloudFormation template based on the annotations in the project.
2. The client component, which allows users to interact with cloud resources easily.
3. The local deployment component, which allows for unit tests and running applications locally.
4. The deployment plugin, which takes the CloudFormation template and uploads it to the cloud to create the resources.

The annotation processor, clients, and local deployment component are bundled together inside of the `nimbus-core` maven dependency, and the deployment component found inside the `nimbus-deployment-maven-plugin` maven dependency.

4.1 Annotation Processor

An annotation processor design was chosen over a design that used an abstract syntax tree because it will work for any user, regardless of IDE, and would not require an external application. Annotations also provide an easy way to add any additional configuration.

The task of the annotation processor is to create CloudFormation templates that detail the configuration of all the resources that will be created in the cloud environment. It has to ensure that all defined resources will be created and work as expected, including all defined permissions. At this stage, checks will be made where possible to ensure that the user has configured everything correctly, and if not, these errors will be reported. The annotations defined by Nimbus can be broken down into three types: functions, resources, and permissions.

4.1.1 Annotation Types

Function annotations are used to annotate methods in the project and will be deployed as serverless functions, like AWS Lambdas. Examples of function annotations include `@HttpServerlessFunction`, `@NotificationServerlessFunction` and

@WebSocketServerlessFunction. Many of these annotations also require creating additional cloud resources; for example, the @HttpServerlessFunction annotation also creates the HTTP endpoints that trigger the function.

Resource annotations are used to annotate classes in the project. Resources are cloud resources in the cloud environment that can be interacted with, but are not functions. For example, any form of data store is defined as a resource. Example resource annotations are @DocumentStore and @FileStorageBucket.

Permission annotations are used to annotate serverless methods (i.e. methods that are annotated with a function annotation) and describe what resources the method is allowed to interact with. For example, the @UsesDocumentStore annotation is required to ensure that a serverless method can interact with a document store. These annotations are needed as the annotation processor cannot detect what clients a method is using, and so cannot infer permissions automatically.

4.1.2 Overview

Internal CloudFormation Representation

CloudFormation templates were introduced in Section 2.4. As a summary, a CloudFormation template is a JSON file that contains mappings of names to resources and outputs. Resources are cloud resources to be deployed, and outputs provide an API to externally query values generated by the CloudFormation stack.

Annotation processing occurs in rounds, as processing the project once might generate more classes with annotations that need to be processed. For this reason, the CloudFormation JSON is not generated and written until all rounds have been completed. An internal representation of a CloudFormation file is kept that keeps track of its resources and outputs and provides a method that converts it into JSON.

The internal representation of the CloudFormation file is CloudFormationTemplate. As the names that map to resources are strings, this representation contains a Map<String, Resource> and a Map<String, Output>. Resource is an abstract class from which all concrete resources inherit, and similarly, Output is an abstract class from which all concrete outputs inherit. When it is mentioned that a resource is added, it is added to this resource map. Inheriting from Resource or Output requires implementing a method that converts the resource or output to its CloudFormation JSON. Examples of CloudFormation objects for each resource can be found in Appendix A.

When a CloudFormationTemplate is converted to its JSON representation, each of the maps are iterated through, and each Resource or Output is converted to its JSON form. This is combined together to produce the final JSON output. The GSON library[1] is used to build the JSON objects as it allows for easily configurable pretty printing, which was helpful when debugging the CloudFormation files.

For each stage, separate CloudFormation files are created, which is done so that separate cloud environments are created and can be managed and deployed independently. Two files are created per stage. One is a 'create' file, which details the first deployment that is done initially and only contains the S3 bucket that is used to host the function code. The create file is only needed when the stack does not exist in the cloud. The second is the 'update' file, which contains all of the resources needed for the user's deployment, as well as the code S3 bucket.

[1] https://github.com/google/gson

Processing

The primary processing can be summarised as follows. The annotation processor first processes resources, then functions, and then finally permissions. This produces instances of `CloudFormationTemplate`, which are then converted to JSON and written to disk. The reason for processing the annotations in this order is that resources do not rely on anything, while functions may rely on resources, and permissions rely on both resources and functions.

On initialisation, the annotation processor creates resource creators for each type of annotation. The role of a resource creator is to create concrete `Resources` and `Outputs`.

These resource creators extend `FunctionResourceCreator` if processing a function annotation, `CloudResourceResourceCreator` if processing a resource annotation, and `UsesResourcesHandler` if processing a permission annotation. The resource creator is then added to one of three lists (one for functions, resources, and permissions) that store each of the resource creators to process. The lists are then processed in the order of `CloudResourceResourceCreators`, then `FunctionResourceCreators` and finally `UsesResourcesHandlers`. The reason for this design is that it makes it simple to add more resources, as new creators just need to be added to the corresponding lists.

The annotation processor also persists some data into the `nimbus-state` file. This file is used to store information that is needed by the deployment plugin, for example, the compilation timestamp and list of functions to be deployed.

The next sections describe the exact processing done for each resource, function, and permission. They detail the CloudFormation resources required to be created for an annotation, and any challenges faced when trying to create these resources.

4.1.3 Resources

A resource creator that extends `CloudResourceResourceCreator` creates the intermediate representation of a resource, which extends `Resource`. These resource creators find all Java `Elements` that are annotated with the corresponding annotations and then finds the annotations themselves.

All intermediate representations of resources extend the abstract `Resource` class. This abstract class stores the properties of the resource that will be converted into the properties field of the resource in the CloudFormation template. It also stores a list of `Resources` that the resource depends on, which is converted into the `DependsOn` field of the resource in the CloudFormation template. This `DependsOn` field is used to ensure that resources are created in the right order so that dependency errors are not thrown when deploying. The abstract class provides methods that allow properties to be added and create CloudFormation functions. These CloudFormation functions are: get the Amazon Resource Name (ARN) of the resource, get an attribute of the resource, join values together (similar to concatenation), get the AWS region, get a reference to the resource, and substitute values into a string.

File Storage Bucket

Amazon's file storage service is S3. The intermediate representation of a file storage bucket is a `FileBucket`, which stores the bucket name, allowed Cross-Origin-Resource-Sharing (CORS) origins, if there is a serverless function associated with it, and if the bucket will be used as a static website.

As this is the first resource described an example of the CloudFormation resource is provided, for all other resources, the CloudFormation examples can be found in Appendix A. Listing 4.1 shows a CloudFormation object that describes an S3 bucket. The `FileBucket` intermediate resource

needs to be able to produce this JSON, and similarly for all other intermediate representations.

```
"WebSocketChatServerWebChatNimbusFileBucket": {
  "Type": "AWS::S3::Bucket",
  "Properties": {
    "NotificationConfiguration": {
      "LambdaConfigurations": [
        {
          "Event": "s3:ObjectCreated:*",
          "Function": {
            "Fn::GetAtt": [
              "WebsiteonFileDeletedFunction",
              "Arn"
            ]
          }
        }
      ]
    },
    "AccessControl": "PublicRead",
    "WebsiteConfiguration": {
      "IndexDocument": "webchat.html",
      "ErrorDocument": "error.html"
    },
    "BucketName": "webchatnimbusdev"
  },
  "DependsOn": [
    "WebsiteonFileDeletedFunction",
    "WebsiteonFileDeletedFunctionPermissionApiGateway"
  ]
}
```

Listing 4.1: S3 Bucket CloudFormation Template

The `NotificationConfiguration` is only present if there is a file storage bucket serverless function associated with it, and is configured by the file bucket function resource creator. `AccessControl` is set to public read, and the `WebsiteConfiguration` parameter is set if the bucket will be used as a static website. Otherwise, these fields are left blank. These fields are controlled in fields of the `FileBucket` class.

If the file bucket is configured as a static website, then an additional bucket policy resource needs to be created. This policy allows the bucket to retrieve items from itself, which means that when items are requested from an HTTP request, the bucket can then read the requested file and return it. The intermediate representation for this is `FileStorageBucketPolicy`, which stores a reference to the file bucket resource.

Relational Database

The relational database service in AWS is RDS, and the intermediate representation is `RdsResource`. This stores the configuration of the database, and references to other required resources. The additional resources are a virtual private cloud (VPC), security group, two public subnets combined to form a subnet group, an internet gateway with a VPC gateway attachment, a routing table, a specific route entry in the table, and finally two route table associations that attach the route table to the subnets.

One design challenge was deciding how to allow users to connect to the database, as there are two distinct ways of doing this. One method is to connect with a username and password directly, and another is to connect using IAM credentials. IAM tokens are generally considered more secure as they expire after a certain amount of time (minimum 15 minutes, maximum 36 hours). One disadvantage of using IAM credentials is that the function must be deployed inside of the same VPC as the database, and functions inside a VPC have a much longer cold start time than functions not in a VPC. For this reason, it was decided to let users connect using a username and password, and so a username and password are required in the configuration annotation.

Another design challenge faced when implementing relational database deployments was deciding how to allow a user to customise the computational resources available to the database. In AWS this is done by specifying a `DBInstanceClass`, for example, 'db.t2.micro' (Free Tier) and 'db.m4.16xlarge'. These are not cloud-agnostic and so forcing users to use this parameter loses the desired cloud agnostic property. The instance class is specified as an optional parameter

to be used when the user needs precise control of the configuration to avoid this problem. In other cases, a database size parameter can be provided, with values of FREE, SMALL, MEDIUM, LARGE, XLARGE, and MAXIMUM. The use of this parameter does not provide any information on what computational resources are available or how much it will cost (except the FREE value) but makes the code cloud-agnostic. The use of both parameters together allows the user to tailor the deployment based on their needs.

Document And Key-Value Stores

AWS provides a service called DynamoDB, a NoSQL database service with which the document and key-value stores are deployed. The intermediate representation is `DynamoResource`, which stores the table name and other configuration. This intermediate resource is not cloud-agnostic as DynamoDB is very different from other offerings, like Azure's Cosmos DB.

The main challenge faced was implementing DynamoDB support was deciding what interface to use. As mentioned in the background research, a DynamoDB like interface could be used, where items are accessed based on a key. Alternatively, an SQL like language could be used to query the data. For simplicity, it was chosen to allow a DynamoDB like interface, where items are accessed based on a key. DynamoDB provides the scan operation to find data, which requires going through each item in the table. AWS recommends avoiding this operation altogether, suggesting that the data should be structured so that the query operation is used to find data [26]. The query operation is much more efficient as it can determine the location of data based on the primary keys. Due to this suggestion, Nimbus provides interfaces for DynamoDB that only require primary keys. Two are currently provided, document and key-value stores, though more could be added in the future.

To deploy a DynamoDB table a key name and type must be specified, where the type can be any of 'N' (Number), 'S' (String), 'BOOL' (Boolean), 'B' (Binary), 'SS' (String Set), 'NS' (Number Set) and 'BS' (Binary Set). Listing 4.2 shows an example of an object with a `String` name and an `int` ID in the DynamoDB format.

```
1 {
2     "name": {
3         "S": "thomas"
4     },
5     "id": {
6         "N": "105"
7     }
8 }
```

Listing 4.2: DyanmoDB representation of an object with two fields: a name and identifier

In a document store, the field annotated with `@Key` defines the key, with the key name set as the name of the field, and the key type defined based on the type of the field.

In the case of a key-value store, the key name is the constant 'PrimaryKey', and the key type is chosen based on the `keyType` parameter of `@KeyValueStore`.

The basic Java types of (anything that extends) `Number`, `String` and `boolean` are directly mapped to the corresponding DynamoDB type. For custom objects and classes, instead of determining a set or map structure and using the corresponding DynamoDB type, the Jackson library is used to convert the object into a JSON string, and the DynamoDB type used is an 'S' (String).

4.1.4 Functions

AWS's serverless function offering is Lambda. A Lambda function is configured with a name, method endpoint (the code), amount of memory and timeout duration. There are two main challenges when deploying functions. The first is allowing for cloud-agnostic methods that are intuitive to use, and the second is defining all the cloud resources required for the different types of functions.

With all of the mentioned functions, a function object is added to the CloudFormation template. This function object includes a variable that specifies the version, which the deployment plugin substitutes on a deployment.

A resource creator that extends `FunctionResourceCreator` handles a function annotation. This resource creator first finds all `Elements` annotated with the corresponding annotation and then finds the actual annotation. With the annotation, the resource creator then creates any required resources. Again, all intermediate representations of cloud resources created by the function resource creator extend `Resource`, as defined in Subsection 4.1.3. The intermediate representation of a function is `FunctionResource`, which stores information on the associated method, as well as a reference to its permission resource and environment variables. Each function resource creator usually creates a new function resource and *function information object*, which contains the corresponding `Element` and `FunctionResource` objects. The function information object is then added to an internal list to be used later when processing permissions.

When creating a function resource, an IAM role resource and a log group resource are also created. The IAM role resource is the resource which stores the permissions of a function. Permissions are added to the IAM role that allows the function to write to the log. The log is useful as any print or log lines added to the function will show up in the log group, which can help the user debug their function. The IAM role is also used later by the permissions handlers to add additional permissions to the function.

Intuitive Functions

To provide a consistent developer experience, all functions follow the same general structure. Nimbus functions are always methods which can usually have two different types of parameters. The first parameter is a custom user type which is passed along in the request/event, and the second is a Nimbus event type, which contains information about the current request/event. All parameter information is detailed in Section 3.1. For clarity, Listing 4.3 shows a brief example, where the `httpFunction` method is a Nimbus function.

```
1 public class HttpHandler {
2
3     @HttpServerlessFunction(path="echo", method=HttpMethod.GET)
4     public ResponseModel httpFunction(DataModel input, HttpEvent event) {
5         return new ResponseModel(true);
6     }
7 }
```

Listing 4.3: HttpHandler Class

In this case, the input parameter is the custom user type. This parameter is found as part of the request and is generally the information sent in the request; in this case, it would be the body of the HTTP request. In the case of a queue function, it would be the item removed from the queue and so on for the other functions. The custom user type can be any class that can be read in from a JSON object. The user can remove this parameter if they do not need it.

In this case, the event parameter is the Nimbus event type. The event type contains details about the request itself, and in the case of an HTTP function, it contains additional parameters like query and path parameters and headers.

This code is straightforward and is intuitive due to how concise the code is to perform this simple task. However, writing this same function without Nimbus or any other framework is more complicated:

```
1 public class NoNimbusHttpHandler implements RequestHandler<APIGatewayProxyRequestEvent,
       APIGatewayProxyResponseEvent> {
2     public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent requestEvent,
       Context context) {
3         String body = requestEvent.getBody();
4         return new APIGatewayProxyResponseEvent().withBody(body).withStatusCode(200);
5     }
6 }
```

Listing 4.4: NoNimbusHttpHandler Class

The code in Listing 4.4 is not cloud-agnostic with the references to API gateway classes. In this case, the input is simple as the body is already a String, but if it were a custom user type like a class that contained user details, then this would have to be deserialised manually. Nimbus handles the deserialisation and serialisation of inputs and outputs for the user.

To help make the code intuitive, the method definitions are flexible. Parameters can be omitted, or the order changed, and the method will still compile. This means that if the event parameter is not needed, it can simply be left out, making the code even more concise. This also means that developers do not have to remember the exact order that the parameters come in, again making it easier.

Function Handlers

The user's function cannot be directly called by Lambda, so a new class is generated, called a *function handler* class or file. This is done for each of the user's serverless functions. This class is generated by a service that extends `ServerlessFunctionFileBuilder`, which corresponds to the function type. These generated classes handle the deserialisation and serialisation of inputs and outputs and call the user's annotated method. If there is a case where the user would prefer to perform the deserialisation and serialisation themselves, then the method can have parameters of `(Inputstream input, Outputstream output, Context context)`. However, doing this loses the cloud-agnostic benefits.

The Jackson library[2] was chosen to perform all the serialisation and deserialisation in the function handlers as it is easily configurable. Additionally, the user can configure their objects with annotations so that they do not have to have access to the actual class that performs the serialisation. The generated class file for the example in Listing 4.3 is shown in Listing 4.5.

To handle the different method parameter options, for example, no parameters or parameters in different orders, first the file builder determines the index of the custom user type and the event type in the user's method. It then places the parameters in the correct order, or omits them, when generating the function handler.

The next sections detail any resources or permissions that need to be added for specific functions and any additional file builder processing.

Basic Function

A basic function consists of only the Lambda, and possibly a cron rule. If it is just the Lambda function, then it is added to the CloudFormation file like all other functions.

If a cron rule is specified, then a scheduled event resource is also added to the CloudFormation file, with the Lambda function as a target. After this, a Lambda permission resource has to be defined that allows the cron rule to trigger the function.

When a basic function is added, it is added to the internal list of invokable functions, which is used later when allowing functions to invoke one another.

HTTP Function

In AWS serverless HTTP endpoints are created by using API Gateway, with the endpoints then set up to trigger the Lambda functions.

CloudFormation has three API Gateway resources that are needed for HTTP functions. Those are RestApi, RestResource, and Method. RestApi is a root node for the API, with each RestRe-

[2] `https://github.com/FasterXML/jackson`

source defining a new path part in a tree-like structure. For a path "example/path" there would be two RestResource created and one RestApi. The RestApi would be created first, with the "example" RestResource parent set to be the RestApi, and the "path" RestResource parent set to be the "example" resource. Only one RestApi is permitted per Nimbus project, therefore adding more HTTP functions adds to this tree-like structure. Methods correspond to HTTP Methods that can be executed on a resource. A Method refers to its parent, either a RestApi when at the top level or a RestResource when moving down a path structure. This Method then refers to the deployed Lambda function that will be called when a request is made to the corresponding resource with the selected HTTP method. Initially, this was tested, and all the endpoints showed up in the AWS console and could be invoked via the web console; however, it was later realised that there were no actual exposed HTTP endpoints, as the API had not been deployed. To create a deployment, an API Gateway deployment resource has to be added to the CloudFormation file. If the name of this resource remains constant, then when a deployment is done, the API will not be updated, as its CloudFormation template has not changed. The name of the resource then includes a compilation timestamp so that the API is updated every deployment.

One problem that was found when testing HTTP functions was that when a request was sent from a web page in a browser, the request was blocked due to the cross-origin settings. Parameters were then added to `@HttpServerlessFunction` that lets the user configure allowed CORS origins and headers. In API Gateway an additional OPTIONS HTTP method needs to be added to the same API Gateway Resource as the function is triggered on. This OPTIONS method is a mocked endpoint that only returns the 'Access-Control-Allow-Method', 'Access-Control-Allow-Headers', and 'Access-Control-Allow-Origin' headers. The allowed origins header corresponds to the value specified by the user, the allowed headers corresponds to the values specified by the user plus the 'origin' and 'content-type' headers, and the allowed method header corresponds to the HTTP method the user specifies for the function. Finally, the HTTP response of the function must include those headers, which is handled by the generated class. Listing 4.5 shows the generated file for the example in Listing 4.3.

```
1  public class HttpServerlessFunctionHttpFunctionhttpFunction implements
2      RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent>{
3
4    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent input, Context
        context) {
5      try {
6        String requestId = context.getAwsRequestId();
7        HttpFunction handler = new HttpFunction();
8        ObjectMapper objectMapper = new ObjectMapper();
9        HttpEvent event = new HttpEvent(input, requestId);
10       DataModel parsedType = objectMapper.readValue(input.getBody(), DataModel.class);
11       models.ResponseModel result = handler.httpFunction(parsedType, event);
12       APIGatewayProxyResponseEvent responseEvent = new APIGatewayProxyResponseEvent()
13                                              .withStatusCode(200);
14       String responseBody = objectMapper.writeValueAsString(result);
15       responseEvent.setBody(responseBody);
16       if (responseEvent.getHeaders() == null) {
17         responseEvent.setHeaders(new HashMap<>());
18       }
19       if (!responseEvent.getHeaders() .containsKey("Access-Control-Allow-Origin")) {
20         String allowedCorsOrigin = System.getenv("NIMBUS_ALLOWED_CORS_ORIGIN");
21         if (allowedCorsOrigin != null && !allowedCorsOrigin.equals("")) {
22           responseEvent.getHeaders().put("Access-Control-Allow-Origin", allowedCorsOrigin);
23         }
24       }
25       return responseEvent;
26     } catch (Exception e) {
27       e.printStackTrace();
28       APIGatewayProxyResponseEvent errorResponse = new APIGatewayProxyResponseEvent()
29                                             .withStatusCode(500);
30       if (errorResponse.getHeaders() == null) {
31         errorResponse.setHeaders(new HashMap<>());
32       }
33       if (!errorResponse.getHeaders() .containsKey("Access-Control-Allow-Origin")) {
34         String allowedCorsOrigin = System.getenv("NIMBUS_ALLOWED_CORS_ORIGIN");
35         if (allowedCorsOrigin != null && !allowedCorsOrigin.equals("")) {
36           errorResponse.getHeaders().put("Access-Control-Allow-Origin", allowedCorsOrigin);
37         }
38       }
39       return errorResponse;
40     }
41   }
42 }
```

Listing 4.5: Generated HttpHandler Class

WebSocket Function

In AWS WebSocket endpoints are created using API Gateway, with the endpoints set up to trigger Lambda functions.

CloudFormation has five resources needed to create a WebSocket API. These are API Gateway API, integration, route, deployment, and stage. API is the top level, which all the other resources reference. Integration specifies a Lambda function to trigger. Route specifies the route key for an endpoint and references an integration resource to associate the route with a function. Finally, deployment and stage configure the deployment so that actual endpoints are created. The name of the deployment changes every time for the same reason as in the HTTP functions.

Document and Key-Value Store Functions

In AWS both these stores are managed by DynamoDB. In the framework, it is required that the tables must have been created elsewhere (i.e. as described in the Document Store and Key-Value Store resource sections). The class that these stores are defined on is called the *store class*.

As the store class is passed into the annotation that defines the store functions, reflection is used to access the information stored in the annotation of the store class. This information can be used to determine the resource name, which is used to find the internal representation of the table. The table is guaranteed to exist (if it has been defined correctly) as the resources are processed first.

The first thing that needs to be added after creating the function resource is an event source mapping resource. This resource tells Lambda to poll the event source for new items and if any exist, then trigger the function. Once this has been configured, then a policy needs to be added to the IAM role of the Lambda that allows the function to access the DynamoDB event stream and take data off of it.

Lambdas set to be triggered from a store will be triggered on any type of change (i.e. INSERT, REMOVE, or MODIFY), and when the function is invoked the type of change is specified in a parameter to the function. However, the user specifies the desired change that triggers their function, which enables the framework to deserialise the arguments correctly. To avoid changes triggering undesired functions, as part of the function handler code generation an if-check is placed around the user code that prevents it being called unless in the desired case.

The three store changes have different associated parameters. In an insert case, only the new item is present and is deserialised, and in the remove case, only the removed item is present and is deserialised. In the modify case, both old and new items are present and need to be deserialised. All of these cases also require compile-time checks to validate that the user has a valid configuration of method parameters for the selected case. For example, if a method has three parameters and is configured to be triggered on an insert, a compile-time error will be thrown, but if configured to be triggered on a modify, an error will not necessarily be thrown.

In the store functions, the deserialisation into the user's parameters is not as simple as converting from JSON due to DynamoDB's proprietary format. Listing 4.2 shows the DynamoDB format of objects. For each field of the DynamoDB object, the deserialisation first detects what annotated field in the Java object it corresponds to. It then checks that the types are compatible, and if they are, it converts the DynamoDB value to the Java value and sets that field in the Java object.

Notification Function

AWS's notification service is called SNS (Simple Notification Service). It can be set up with topics that notify endpoints such as SMS numbers, email addresses, and Lambda functions.

After the function has been defined, the notification topic needs to be created in CloudFormation with an intermediate representation `NotificationTopicResource`. In this step, the function is set as the notification topic endpoint, and the topic name configured. The topic name is defined as the topic supplied by the user appended with the stage, which allows different topics to exist for different stages.

Once this is done, a new Lambda permission resource needs to be added that allows the notification topic to trigger the function.

Queue Function

SQS (Simple Queue Service) is AWS's messaging queue service. When a Lambda is set to trigger from the queue, the Lambda service polls the queue and invokes the function in batches.

After the function has been added to the CloudFormation template, the queue needs to be defined. The name of the queue is defined from the `id` provided by the user appended with the stage. This is done so that a new queue is created for each stage. Another critical parameter of a queue is its visibility timeout, which is the time between items being taken from the queue by a consumer, and it actually being removed from the queue. According to AWS documentation[3], this should be set to at least 6 times the timeout of the function, which is what Nimbus sets it to. The intermediate representation of a queue is a `QueueResource`, which stores the configuration parameters. Now that the queue has been created an event source mapping resource is defined, with user-specified batch size. This specifies how the Lambda service should poll the queue. Finally, a policy needs to be added to the function's IAM role that allows the function to poll the queue.

Due to the configurable batch size, it was decided that the user could have two ways of specifying the parameters of the function. The first is when the parameters of the method are single instances, which can be used regardless of the batch size. The second is when the parameters are lists of the previous instances, which can be used when a batch size of greater than one is specified. Specifying a batch size of more than one is allowed as the user may want to do an optimisation that aggregates the data, or they may want the function to call in batches as it reduces the total number of function invocations. The code generation handles both these cases by collecting all items into a list and then either calling the user method one at a time or with all items at once.

File Storage Function

The file storage bucket service provided by AWS is called S3. In a file storage function, the bucket could be new or already defined in the project by the `@FileStorageBucket` annotation.

Once the function has been created, the existing file bucket is found, or a new one created. The created bucket is similar to how buckets were created in the Resource section, but with no static website configuration. A new notification configuration is added to the bucket that specifies the type of storage event that triggers the function, and the function to actually trigger. Once this is done, a Lambda permission resource needs to be added that allows the bucket to trigger the function.

With S3 there is no user data that can be deserialised from files, as any file could be uploaded. Furthermore, AWS does not provide file data inside the parameters to the lambda function. Thus it was decided not to allow users a custom user type parameter to the method, as if the file data is required it can be found by using the file storage client and the key found in the file bucket event.

[3] https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html

After Deployment Function

An after deployment function is simply a basic function that will be triggered by the deployment plugin once deployment has been completed. Thus only a function is created, with no configured triggers. Additionally, the function name is added to the list of after deployment functions in the `nimbus-state` file so that the deployment plugin knows what to trigger.

File Uploads

Although not actually a function, the resource creator for file uploads extends `FunctionResourceCreator`. No CloudFormation resources need to be created here; the `@FileUpload` annotations are processed by adding a description of the file upload to the `nimbus-state` file. This description is just the local path, target path, and whether or not to substitute variables.

4.1.5 Permissions

When a user wants to use a client inside of a function, then a corresponding permissions annotation must be added to the function's method. For example, if a key-value store client is being used, then the `@UsesKeyValueStore` annotation must be added. In terms of the CloudFormation, the permissions are handled by adding policies to the IAM role of the annotated function. Additionally, environment variables may need to be added to the function resource so that the clients will work correctly.

There was some consideration of how the permissions would be handled. One constraint is that CloudFormation templates have a limit of 200 resources, and each IAM role counts as one resource, so ideally as few IAM roles as possible should be created. To this end, a single IAM role could be created for all functions and given all permissions. However, this has the downside of having poor security. To make this better only the permissions declared could be added to this single IAM role. However, this means that some functions will have unnecessary permissions. In the end, a more secure design was chosen that follows the principle of least privilege. In this design, each function has its own IAM role that only has the permissions declared on that function. The resource limit is not a huge concern, and if it is hit, then the project should be split into separate parts (as suggested already in AWS documentation).

Processing starts by taking the list of function information objects generated by the function annotation processing. Each item in the list is passed to a permissions resource creator, which then does the processing. The exact processing done depends on the permission annotations, and is detailed in the next sections.

At this stage, all the functions and the clients they use are persisted in the `nimbus-state` file.

The next sections detail the permissions added to the IAM role of a function when processing permission annotations, and any describe any additional resources that need to be created.

Basic Function Permissions

A method is annotated with `@UsesBasicServerlessFunction`. Two environment variables are added to the function, one that specifies the project name and another that specifies which stage the function is deployed in. Finally, policies are added to the IAM role of the function that allows it to trigger all functions in the list of invokable functions.

Document Store and Key-Value Store Permissions

A method is annotated with `@UsesDocumentStore` or `@UsesKeyValueStore`. Only one permission needs to be added that allows the function to interact with the DynamoDB table.

File Storage Bucket Permissions

A method is annotated with `@UsesFileStorage`. In this case, a file bucket is created if one has not been declared. The new bucket is added to the CloudFormation file as a very basic bucket, with no website configuration. The bucket is found if it already exists. Regardless, policies are added to the IAM role of the function that allows it to get, delete and put objects into the bucket, and other policies to do the same to all items at nested levels of the bucket. Finally, an environment variable is added to the function that specifies the stage it is deployed in.

Notification Topic Permissions

A method is annotated with `@UsesNotificationTopic`. In this case, a notification topic is created if one has not been declared. The new topic is added to the CloudFormation file with the name declared in the annotation. Policies are then added to the function's IAM role that allows it to subscribe and unsubscribe clients to the notification topic, as well as publish items to the topic. Finally, an environment variable is added to the function that specifies the ARN of the topic.

Queue Permissions

A method is annotated with `@UsesQueue`. No queue is created if one does not exist. An environment variable is added to the function that specifies the URL of the queue. Finally, a policy is added to the function's IAM role that allows the function to send items to the queue.

Relational Database Permissions

A method is annotated with `@UsesRelationalDatabase`. No database is created if one does not exist. Environment variables are added to the function that specifies the connection URL, username, and password. No permissions need to be added. At this point, the function is marked as requiring a specific database driver class in the `nimbus-state` file. The database driver depends on the language being used.

WebSocket Management Permissions

A method is annotated with `@UsesServerlessFunctionWebSocket`. It is required that the function is a WebSocket function. An environment variable is added to the function that specifies the WebSocket API management endpoint. Finally, a policy is added to the IAM role of the function that allows the function to manage the connections to the WebSocket API.

Environment Variables

A method is annotated with `@EnvironmentVariable`. When the user specifies their own environment variables, they are added in this step. The environment variable specified by the annotation is added to the function resource.

4.1.6 Summary

Once the resources, functions, and permissions have been processed, the CloudFormation templates are complete and are written to disk. At this point, the `nimbus-state` file is also written to disk, and the annotation processing is complete.

4.2 Client Components

The clients allow users to interact with the resources in a cloud-agnostic fashion. Initially, there was one client per cloud resource that would interact with the cloud resource, providing a cloud-agnostic API. The user would then initialise one of these clients directly. However, once the local deployment functionality was introduced this broke as the local functions would initialise clients that attempted to connect to cloud resources. Separate clients were written for each resource, one for cloud deployment and another for local deployment, each sharing the same interface. It was desired that the user would not have to change their code to switch between cloud and local deployment, so it was decided that some form of dependency injection would be used to pick which client was provided to the user.

A Spring-like technique was considered where annotations would be used to mark variables that would be injected. This is tricky due to the complexity involved with generating classes and is not possible unless at runtime, meaning that a user would have to write some other initialising code. Thus the `ClientBuilder` class was implemented, which creates instances of the clients and returns cloud clients or local clients, depending on the environment. The client constructors were then marked as package-private so that the user cannot directly initialise them. Listing 4.6 shows an example of how to use the `ClientBuilder` to initialise a client.

```
DocumentStoreClient<DocumentClass> client = ClientBuilder.getDocumentStoreClient(DocumentClass.class)
```

Listing 4.6: Client Builder Example

4.2.1 Implementation

Document Store Client

The document store client is created from a class that is annotated with `@DocumentStore` and the stage. An instance of this class is referred to as a *store object*. The class is provided by the user, and the stage is taken from the environment variable generated by the permission annotation. An instance of the type of the field that is annotated with `@Key` is called a *key object*.

First, the name of the table is determined using reflection to find the parameter to the document store annotation along with the stage. Next, a map of attributes names to the corresponding Java `Field` is created, the *object definition*, by using reflection to determine what fields are annotated with `@Key` and `@Attribute`. Finally, the `Field` annotated with `@Key` is saved as the key field. The AWS DynamoDB library is used as underlying implementation.

A map of key column names to DynamoDB values is called a *key map*. A map of all attributes (including the key) column names to DynamoDB values is called an *attribute map*.

The interface exposed with a document store client is similar to a Java `Set`. Thus there are methods to put, delete, and get items. In the underlying table, all items are accessed with a primary key, the type of this key will be one of the DynamoDB types, such as N or S. Helper methods were written that transform any Java object into its corresponding DynamoDB value, and back again.

A put operation takes a store object and saves it to the database. The underlying DynamoDB library expects an attribute map, so the client uses its internal attribute map to read the values

of the required fields and transforms them into the correct DynamoDB value. Once this map is created, the library is then used to place the item into the table.

A delete operation takes either a store object or a key object. The underlying DynamoDB library function expects a key map, though in this case there is only one key. If a store object is provided, then the key needs to be extracted using the key field and then transformed into the required map of DynamoDB values. If a key object is provided, then this is transformed into its DynamoDB value, and into the key map. Once this map is created, the library is then used to remove the item into the table.

A get operation uses a key object to query the table to find a store object. The underlying DynamoDB library function expects a key map. Similar to the delete operation, the key object is transformed into a key map, and then the library used to query the table. Now the client has an attribute map an instance of the store object can be created. The attribute map is modified into a map of field names to Java objects, using one of the helper methods. The Jackson library is then used to transform this map into a concrete store object, which is returned to the user.

Key-Value Store Client

The *value class* is the class that is annotated with the `@KeyValueStore` annotation, and an instance of this class is called the *value object*. The *key class* is the class that is defined as the `keyType` parameter to the `@KeyValueStore` annotation, and an instance of this class is called a *key object*

The client is instantiated by taking a key class and value class from the user, as well as a stage parameter taken from the environment variables. First, the name of the table is determined using reflection on the value class to find the parameter to the `@KeyValueStore` annotation along with the stage. Similarly to the document store client, an *object definition* is created. The *key map* and *attribute map* are defined in the document store client section above.

The interface exposed with a key-value store client is similar to a Java `Map`. Thus there are methods to put, delete, and get items. As with the document store client, the underlying store is DynamoDB, so the same issues with type conversions take place. The helper methods in the document store client are also available here.

A put operation takes a key object and a value object and saves them to the database. This operation is performed similarly as in the document store client. Both objects are converted into one attribute map by combining them together, and then the underlying library inserts it into the table. This has to be done as the value object on its own does not contain the primary key.

A delete operation takes a key object and removes it from the database. This operation is performed exactly the same as the document store delete.

A get operation uses a key object to query the table to find a value object. This operation is performed similarly to the document store get. Once the attribute map is obtained, it is modified into a map of field names to Java objects, using one of the helper methods. One key difference here is that care needs to be taken to ignore the key column as this will not map into the value object. The Jackson library is then used to transform this map into a concrete value object, which is returned to the user.

File Storage Client

The file storage client is created from a bucket name specified by the user and allows interaction with the bucket identified by that bucket name. The AWS S3 library is used as an underlying implementation. This client provides the functionality to save files to the bucket, possibly forcing a user-specified content type, as well as retrieving and deleting files from the bucket.

All of these methods simply take the user's input and then hands it over to the AWS library. The most complex method provides the functionality to list all the files in a bucket. In this case, the library is used to request all item information, where the result is paginated. Thus multiple requests are made until all information on all the items is obtained, and then just the file keys are returned to the user.

Basic Serverless Function Client

The basic serverless function client is created from the project name and the deployed stage. Both of these variables are obtained from environment variables that are attached to the function by the permission annotation. These values are used to determine the name of the Lambda function to be invoked. The AWS Lambda library is used as an underlying implementation. This client allows users to invoke basic functions in a synchronous or asynchronous fashion.

If a return value is desired, then the function has to be called synchronously. The invoked function will return a JSON value, and this is deserialised into the user's desired type. In all cases, the user's parameters are simply passed to the library code with no additional transformations.

Environment Variable Client

This client simply accesses the environment variables of the container the function is running in. It provides two methods, one to check if the key is present, and one to get the value of a key in the environment variables.

Notification Client

The notification client is created from the topic name of the notification topic that will be interacted with. Once this topic name is obtained, the environment variables are inspected to find the topic ARN. With this ARN the AWS SNS library can be used to interface with the topic. The client provides the functionality to create and delete subscriptions from the topic, as well as sending messages to the topic.

Most of the methods merely take the user-supplied data and pass them to the underlying library. The only exception is a send notification method that allows the user to supply any type of object that the client will serialise into JSON before passing it to the SNS library.

Queue Client

The queue client is created from the `id` of the queue that will be interacted with. Once this `id` is obtained, the environment variables are inspected to find the queue URL. With this queue URL, the AWS SQS library can be used to interface with the queue. The client provides the functionality to send messages to the queue.

One send method simply takes the user-supplied data and passes it to the SQS library, and the other can take any object and serialise it into JSON before passing to the SQS library.

Relational Database Client

The relational database client is created from a class annotated with `@RelationalDatabase`. First, reflection is used to get the database parameters from the class, including the name and query language used. The environment variables are then inspected to obtain the connection URL, username and password. The client provides the functionality to get a `Connection` object

to the relational database. The client uses all the previously mentioned information to create a Java Database Connectivity (JDBC) URL and then uses the `java.sql.DriverManager` class to return a connection to the user.

WebSocket Management Client

The WebSocket management client requires no parameters for construction but immediately looks at the environment variables to find the WebSocket management endpoint and the AWS region. These variables are used to instantiate the underlying AWS API gateway library. This client provides the functionality to send messages to WebSocket connections.

One method simply takes the user data and passes it to the library, and the second can take any type of object and convert it into JSON before sending.

4.3 Local Deployment

The local deployment component aims to simulate the cloud environment locally so that users can test if their code integrates correctly. Simulating an exact replica of the cloud environment is infeasible due to the size and complexity of each system running asynchronously, and the fact that the inner workings of the cloud systems are unknown. For this reason, a simpler environment is simulated with all actions being synchronous.

The core part of the local deployment component is the `LocalNimbusDeployment` class. When this class is created, a package is specified, and all Nimbus annotations are processed in this package. Resources are simulated from the found annotations which can then be interacted with. Additionally, new clients were added that interact with the simulated resources instead of cloud ones.

Methods are provided by the `LocalNimbusDeployment` class that allows the users to write unit tests. Alternatively, server instances can be created, which is useful for the development of REST and WebSocket APIs. These servers can be tested with clients for a faster feedback loop, instead of making a change and then waiting minutes for it to deploy in the cloud.

The next sections detail how each of the cloud resources are detected, modelled and then simulated.

4.3.1 LocalNimbusDeployment Initialisation

A class or package to analyse is specified to the constructor of `LocalNimbusDeployment`. A list of all the classes in the package is obtained using the Reflections library. Similar to the annotation processor, annotations are processed in the order of resources, functions and then permissions. This processing will create the local resources and store a reference to them in the `LocalNimbusDeployment` instance.

A static factory method is used by the user to obtain an instance of the `LocalNimbusDeployment`. A new instance can be requested, which does the above processing, or the current existing deployment can be requested. This is analogous to a singleton pattern where a new instance can be created that replaces the existing one. This is used so that clients can request the existing deployment and then find the local resource that can then be interacted with.

When a `LocalNimbusDeployment` instance is created, a static flag is set which specifies that the current running environment is a local deployment. This flag is used by the `ClientBuilder` class to determine whether to use the cloud or local clients.

4.3.2 Local Resources (Unit Tests)

Document Store

The internal representation of a document store is a map of any type of object to a string. The object represents the field annotated with `@Key` in the document store, and the string is a JSON representation of an object. Additionally, a list of Java methods that are annotated with `@DocumentStoreServerlessFunction` corresponding to this store is kept in the local representation.

In the cloud document store, client items are serialised and deserialised from a Java Map. The same library is used to create the JSON representation that is stored to help find any errors with this serialisation. This ensures that if a serialisation error occurs in the cloud, it will also occur locally. If a plain old Java object was stored in the internal representation, then additional care would have to be taken. When an item is retrieved from this table and then modified, then the table item is also modified due to the pass by reference nature of Java. This is not the behaviour of the cloud table. A copy of the item would have to be made before being returned to the user to fix this; however, the use of JSON solves this problem as a new object is created every time the JSON is deserialised.

The local document store provides all the functionality of the `DocumentStoreClient`. Items can be inserted, retrieved, and removed from the store. For insertion and deletion, the key is extracted from the object by inspecting the field annotated with `@Key` and that value used to index the internal map. When the get method is called the supplied key object is used directly to index the internal map. Depending on the operation, corresponding functions are called in the following manner:

- If an item is inserted, then the insert document store methods are invoked with the new item.
- If an item is removed, then the remove document store methods are invoked with the old item.
- If an item is modified, then the modify document store methods are invoked with both the old and new items.

One additional bit of functionality that the local document store provides, beyond the document store client functionality, is that the size of the store can be requested.

Key Value Store

The internal representation of a key-value store is a map of the key class to a string. The key class is the class specified in the annotation parameters, and the string is a JSON representation of an object. Additionally, a list of Java methods that are annotated with `@KeyValueStoreServerlessFunction` corresponding to this store is kept in the local representation.

In the cloud key-value store client items are serialised and deserialised from a Java Map. As in the local document store, the Jackson library is used to perform this serialisation and deserialisation and solves the same problems.

The local key-value store provides all the functionality of the `KeyValueStoreClient`. Items can be inserted, retrieved, and removed from the store. For all of these cases, the provided key is used to index the internal map. Depending on the operation, corresponding functions are called in the following manner:

- If an item is inserted, then the insert key-value store methods are invoked with the new item.
- If an item is removed, then the remove key-value store methods are invoked with the old item.

- If an item is modified, then the modify key-value store methods are invoked with both the old and new items.

As in the local document store client, the size of the store can be requested, for more testing functionality.

File Storage Bucket

A local file storage bucket uses a folder in the temporary directory of the operating system, the exact location corresponding to `tmp/nimbus/BUCKETNAME`, where BUCKETNAME is the name of the bucket being simulated. For unit tests, the static website functionality is not available. A list of methods corresponding to `FileStorageServerlessFunction` for this bucket is stored internally.

The local file storage bucket provides all the functionality of the `FileStorageClient`, so files can be saved, deleted, and listed. These functions just save and remove files from the temporary directory. When the local bucket is created the temp directory for the bucket is emptied so that an empty bucket is simulated and tests are independent. When an item is saved, the insert file storage methods are invoked, and when an item is deleted the delete file storage methods are invoked.

Relational Database

A local relational database is implemented using the H2 database engine [27]. This does not require any annotation processing as H2 detects it all automatically. When the user requests a connection to a database, the local database client returns a connection to an embedded in-memory database. This connection has a parameter that specifies not to delete the database when the connection is closed, only when the whole process is terminated.

4.3.3 Local Functions (Unit Tests)

All functions are stored in a map of an identifier to a basic `ServerlessMethod`. The serverless method class lets a user inspect the times the function has been invoked, plus the most recent invoke argument and return value. The identifier is a pair of the class the user's method is defined in plus the name of the method.

All internal representations of serverless functions extend the `ServerlessMethod` class and are responsible for keeping this up to date. No code generation is done here. Instead, the internal representations call the user's method directly. The internal methods usually perform this action before calling the user's code to replicate the serialisation and deserialisation of the generated classes. The main challenge here is detecting which parameters are present in the user's method and the order they come in so that it can be invoked correctly.

HTTP Functions

When an HTTP function is found it is added to a map of HTTP identifiers to the HTTP function, the *HTTP function map*. This identifier is simply the HTTP method and the path and is unique for all functions. In a unit test, an HTTP function is triggered by calling a method on the `LocalNimbusDeployment`, with an HTTP Request parameter. The request is then transformed into a string body, which is deserialised into the custom user type on the method, and the `HttpEvent` parameter. The user's function can then be located in the HTTP function map using these two parameters, and then invoked.

The `HttpEvent` provided to the HTTP function contains query string parameters and multi-value query string parameters, and similarly headers and multi-value headers. One challenge here was correctly extracting the query string parameters from the URL and then correctly populating the query string parameters in the HTTP event, and similarly for the headers. The single parameters are of type `Map<String, String>`, while the multi-value parameters are of type `Map<String, List<String> >`. For a URL `www.example.com?key=one,two`, the query parameters should have one entry `"key"` that has the value `"one,two"`, and the multi-value query parameters will have one entry `"key"` that has a list value with two entries, `"one"` and `"two"`. To perform this extraction, the path of the URL is taken and the substring after the first `'?'` taken. This substring is split on the `'&'` character to produce an array of strings. This array will have one entry for each query parameters. This array is iterated through, and each value is split on the `'='` character. The first half is taken as the key for the parameters, and the value is taken as the single parameter value. To convert this single parameter value into a multi-value parameter value, it is split on the `','` character and converted to a list. These values then used to populate both the single and multi-value parameters. The headers are processed in a similar manner, without the initial URL parsing.

WebSocket Functions

When a WebSocket function is found, it is added to a map of topic identifiers (string) to local WebSocket methods, the *WebSocket function map*. In unit tests, there are methods to invoke the connect and disconnect methods, and to invoke a custom user topic. If the custom user topic does not exist (i.e. not found in the WebSocket function map) then the default topic is attempted to be invoked. These methods are invoked with a mock WebSocket request, which is then transformed into the custom user type and the `WebSocketEvent`.

Document and Key-Value Store Functions

When a document or key-value store method is found, then it is added to an existing document or key-value store. The internal class that represents one of these methods has 3 exposed methods. These are to invoke the underlying user method either as an insert, modify, or delete. Inside of these methods is a check to see if the type of the user's method (i.e. insert, modify, or delete) matches the kind of trigger. If it does, then execution continues to execute the user's code; otherwise, it returns. In all cases, the number of times invoked is incremented. This is done to match the cloud behaviour as store cloud functions are invoked on any change to the store.

File Storage Bucket Functions

When a file storage function is found, then it is added to the corresponding local file storage bucket. If this bucket does not exist, then it is created. The internal class that represents a local file storage method has one exposed method which invokes the underlying user code. This exposed method has a parameter that specifies the file storage event (i.e. object created or deleted) that is used to check whether the event matches the underlying user method event and only calls the user method if it matches. A `FileStorageEvent` is created from the path of the file and the file size and then passed as a parameter to the user method.

Basic Functions

When a basic method is found, it is added to an internal map of function identifiers to local basic methods, the *basic function map*. The identifier is a pair of the class the user's method is defined in plus the name of the method. The internal representation of the method has one exposed method that then calls the user's method directly.

Notification Functions

When a notification method is found, it is added to the corresponding local notification topic.

A local notification topic keeps two lists of its current subscribers, either general endpoints, such as SMS or emails, or function endpoints. The found notification method is added to the list of function endpoints. A local notification topic also keeps track of all the messages that have been sent to general endpoints. This allows the user to check that SMS or email endpoints have been correctly notified. Due to the synchronous nature of the local testing, no internal state for messages needs to be kept; messages just need to be forwarded to all the endpoints. The notification topic has exposed methods that allow new general endpoints to be subscribed, unsubscribed, or all endpoints notified. When a new endpoint is subscribed, it is added to the list of general endpoints, and when an endpoint is unsubscribed, it is removed from the list. The notify method goes through the list of function endpoints and invokes the local notification method, and then goes through the list of general endpoints. When notifying a general endpoint, the message is added to the data structure that keeps track of general endpoint messages.

The internal notification methods are invoked with a string parameter which is then deserialised before calling the user method, along with creating a new `NotificationEvent`.

Queue Functions

When a queue method is found, it is added to the corresponding local queue.

A local queue keeps a list of its consumers. In the local deployment, these can only be queue methods. The queue provides methods to add items to the queue one at a time or in a batch. Due to the synchronous nature of the local deployment when an item is added one at a time, a consumer is chosen randomly and then immediately triggered. If a batch is added then a consumer is chosen randomly, then invoked with items up to its batch-size. If there are items remaining in the queue, this process is repeated, invoking consumers until the queue is empty.

Local queue methods can be invoked with any type of object. If the user method expects a list of items, then this object is either wrapped with a list or passed directly, depending on if the object is a list. If the user method expects single parameters, then the object is either passed directly or passed one by one depending on if the object is a list.

After Deployment Function

When an after deployment function is found, it is added to the internal list of after deployment functions. After all the local resources have been created, and file uploads have been processed, the after deployment functions are run. These methods are not tracked like the other serverless methods and therefore cannot be queried by the user. However, their return value is printed to the console like a cloud after deployment function.

The `@AfterDeployment` annotation has a parameter that allows a user to specify whether the method is used for testing. After deployment methods with this parameter set to false are added to the front of the internal list, and methods with this value set to true are added to the end of this list. This guarantees that testing methods will run after non-testing methods. This is useful when the non-testing methods build some structure and then testing methods populate the structure, and rely on it being there. For example, a non-testing method may create tables in a relational database, and then the testing method populates the tables with data.

Annotation	Permission Added to Environment	What Permission Allows	Creates Resources?
@UsesDocumentStore	StorePermission with class in annotation	Can use client initialised with same class	No
@UsesKeyValueStore	StorePermission with class in annotation	Can use client initialised with same class	No
@UsesRelationalDatabase	StorePermission with class in annotation	Can use client initialised with same class	No
@UsesFileStorage	FileStoragePermission with bucket name in annotation	Can use client initialised with same bucket name	Yes
@UsesNotificationTopic	NotificationTopicPermission with topic name in annotation	Can use client initialised with same topic name	Yes
@UsesQueue	QueuePermission with ID in annotation	Can use client initialised with same queue ID	No
@UsesBasic ServerlessFunction	BasicFunctionPermission with target class and method name in annotation	Can use client initialised with same target class and method name	No
@UsesServerless FunctionWebSocket	AlwaysTruePermission	Can instantiate any WebSocket client	No
@EnvironmentVariable	AlwaysTruePermission	Can instantiate any environment variable client	Adds Variables

Table 4.1: How the different resource permissions are processed locally

4.3.4 Function Environment

One feature that was desired was the ability to detect permissions locally so that permissions errors could be found before deploying to the cloud. Originally it was desired that this could be detected at compile time somehow, for example, by the annotation processor inspecting the serverless methods for the use of any clients. Unfortunately, the annotation processor does not have this capability so a compromise was reached where any permissions issue can be detected by running the code locally.

To detect permissions locally, when a client is run, it needs some way to know the permissions that the function running it has. There are two challenges associated with this. First is associating permissions with functions somehow, and the second is determining in a client what serverless function is calling it.

To solve the first challenge, the `FunctionEnvironment` class is used to represent the cloud environment a function will run in. It specifies the permissions the function has, as well as any environment variables. For each local function created, a new environment is created for that function and saved in a map of function identifier to the environment, the *function environment map*. The *function identifier* is a pair of the class the method is defined in and the name of the method.

When any local clients are used, the program stack is inspected to find the serverless method that is currently running, and then the corresponding environment obtained and checked for the client permission. The permission value is cached for that instance of the client so that the stack only has to be inspected once. More detail is provided in Subsection 4.3.5.

Table 4.1 details how each permission annotation is processed locally on `LocalNimbusDeployment` initialisation, and how it affects the corresponding `FunctionEnvironment`.

4.3.5 Local Clients

The local clients act as the interface between the user code and the local resources. As all the local resources implement the client interfaces, the local clients only have to acquire the corresponding local resource and then use the provided methods to provide the required functionality. To do this, they get an instance of the local Nimbus deployment and use the provided methods to get the local resource.

As mentioned in Subsection 4.3.4, one challenge with the clients is that they have to detect the permissions of the function they are running in. The *function environment map* at this point has been created which can be used to find the function environments by indexing on the class name and method name. The main challenge at this point is that the client has to determine which serverless function invoked it, through any level of method calls. The stack trace can be obtained, which contains an array of elements that represent all the method calls that have been chained to reach the current execution. One element of the array contains the class name and method name that was called, which can be used to create a function identifier to query the function environment map. If the identifier is contained in the map, then the corresponding element is a serverless function.

The program stack is thus queried downwards (most recent to least recent) until the first serverless function is found, or the entire stack has been queried. The corresponding environment is then obtained. Each client knows the specific permission that is required to use it; for example, a notification client knows that a `NotificationPermission` with a specific topic is required. Thus the client checks the environment for that specific permission. If the environment does contain the permission, then the client caches this value so that the stack trace does not have to be analysed again, and the client continues with its operation. If the environment does not contain the permission, then the client throws a permission exception that highlights to the user what permission is missing. This method also ensures that if there are nested function calls, which can be the case if a basic client is used, permissions are handled correctly.

The local clients check permissions on every method invocation, not on initialisation. This design was chosen as many serverless functions may be declared in a class, and clients may be declared in fields of the class. This allows only one client to be declared in the code, and each cloud function will create its own instance. If a cloud function doesn't have permission to use one of the clients declared in the class fields, then it would crash when invoked if permissions were checked on initialisation. This would occur even if the client was never actually used.

4.3.6 File Uploads (Local)

After resources, functions, and permissions have been processed, the file uploads occur. In the `LocalNimbusDeployment` instance, a list of file upload descriptions is stored, which describes the contents of all `@FileUpload` annotations. This list is created on `LocalNimbusDeployment` initialisation. The file upload description specifies the local file or directory, the target file or directory and whether or not to substitute variables.

Also inside of the `LocalNimbusDeployment` instance, the variables to substitute are stored as a map of strings to strings, the *variable substitution map*. This is a map of strings to strings. The key specifies the variable to replace, such as `"${NIMBUS_REST_API_URL}"`, and the value is the string to replace, such as `http://localhost:8080/function`. This map is updated as corresponding resources are created.

The file uploads are then run by going through each item in the file upload descriptions list. First, the description is checked to see if variable substitution takes place. If it does, then a new file is created inside of the temporary directory that has the replaced values, using the variable substitution map mentioned above. The substituted file or the file that did not need substituting is then copied into the local file storage bucket. If the file upload description specifies a directory, then the above method is recursively carried out for all items in the directory.

4.3.7 Local Deployment Servers

There are two kinds of servers that the framework can spin up: an HTTP server, and a WebSocket server.

HTTP Server

The HTTP server supports resources from a REST API, i.e. the HTTP serverless functions, and files hosted in file buckets configured as static websites. In the cloud, a REST API and every file storage bucket would have a different origin, so one challenge was deciding how to host all these servers locally in a way that emulates this behaviour. One way of doing this is by using a different port for each service. However, this could require attempting to host on many different ports if many are required. Instead, the user can configure one port that will host all endpoints. To separate each service into a distinct 'origin', each service has a different base path. For example, the REST API can be found at `http://localhost:8080/function/`, and different buckets can be found at `http://localhost:8080/BUCKET_NAME/`. This solution has the advantage of only one server needing to be spin up and allows the user to configure the port.

The Jetty [28] library is used to host the server. The library expects a handler to be written that then handles all requests. Handlers for each service (i.e. REST API and each bucket) are created as they are discovered as part of the `LocalNimbusDeployment` initialisation. A top-level handler was then written that maps the prefix part of the request path to the service handler, and Jetty is provided with this top level handler. For example, if the top level handler receives a request for `http://localhost:8080/function/`, it will extract `function`, look it up in an internal map of the prefix to a service handler, and then forward the request to the service handler. If it does not find a service handler, then it returns a not found error (404).

Service handlers contain a map of HTTP identifiers, a pair of the path and the HTTP method, to a resource that will process the request. The resource could be a file or a function. File resources simply read the file from the local storage bucket and return it with the specified content type. Function resources transform the request into one that the user function expects, i.e. with deserialised custom user type and a `HttpEvent` parameter, as well as serialising the function result into the HTTP response. This also has to handle the function returning a `HttpResponse` and take headers and status code from this result, similar to the cloud behaviour handled by the generated handler code. Service handlers have index and error files specified to them. When a request is made to the top-level resource, the resource specified by the index file processes the request, and if a resource cannot be found the resource specified by the error file processes the request.

The first HTTP function processed as part of the `LocalNimbusDeployment` initialisation will create the REST API service handler. As part of this step, the variable substitution map is updated with the local REST API URL. Whenever subsequent functions are processed, they add function resources to this handler, with the path and method specified by the `@HttpServerlessFunction` annotation. No index file or error file is specified for the REST API service.

When a `@FileStorageBucket` annotation is processed as part of the `LocalNimbusDeplo-yment` initialisation, if it is specified as a static website, a new service handler will be created for that bucket. As part of this step, the variable substitution map is updated with the bucket URL. The handler has the index and error files specified from the annotation parameters. When files are uploaded to the file storage bucket using the local client, file resources are added to the bucket handler. If a content type is specified, then this is set as the content type of the file resource, if not, then the content type is attempted to be inferred from the file. It is important to note that any `@FileUpload` files are handled in this case.

When the CORS issue between the different services was first discovered, some way of being able to catch this error locally was desired. As it is an issue that is only present when a browser is used, it is handled when testing with local servers. The resources that handle the requests have

allowed CORS headers and origins specified to them. As the browser cannot perform the checks, due to the origin being the same, the handlers have to perform the checks manually. When a request is made the service handlers check that all the headers and the origin are permitted. The headers are easy to check due to being directly in the request. However, the origin does not change between different services (i.e. one bucket or another) as it is hosted locally on one port. The `Referer` header is thus used to determine the origin, as this will specify the path part of the request. Due to the URL of the local services being different from the cloud services, the origin can be specified as a variable that is substituted with the local or cloud value. If the CORS check fails, then an unauthorised response is sent back, and a log line alerting the user to the CORS issue printed.

WebSocket Server

The WebSocket server only supports a WebSocket API defined by the WebSocket functions. As there is only one WebSocket API defined in a project, only one port is used as the local server.

To host the server the Jetty library is used. The library expects a handler to be written that then handles all requests. This top-level handler is created, with the connect, disconnect, default, and any other topics specified. When a WebSocket connection is made, the handler invokes the connect function. If this function throws an exception, then the session is not connected, as in the cloud environment. When a session is disconnected, the disconnect function is run. When a message is received from a session, the topic is extracted from the message, and the corresponding function found. If the function does not exist, then the default function is run.

When a WebSocket serverless function is processed as part of the initialisation of the `LocalNimbusDeployment`, the function is added to the handler with the topic specified by the parameter to the annotation. The handler does pattern matching on the topic to decide whether to add the function as a connect, disconnect, default, or normal topic. The variable substitution map is updated at this point with the WebSocket API URL.

4.4 Deployment Plugin

The purpose of the deployment plugin is to take the CloudFormation files generated by the annotation processor and deploy the code and described resources to the cloud. Nimbus is distributed as a maven dependency, and this plugin is distributed as a maven plugin. Thus the assumption is made that maven is the dependency manager being used by the user. First, the `nimbus-state` file is read, this was created by the annotation processor and contains information needed for deployment. Next, the code needs to be compiled into a shaded JAR, then uploaded to the cloud. Afterwards, the AWS CloudFormation stack update process is started which will create/update any resources required by the user, described in the CloudFormation files. Once this is done the file uploads are processed and the files uploaded to their corresponding bucket. Finally, any after deployment functions are invoked and run.

4.4.1 Creating Shaded JARs

Here the user has a choice. They can either use another packaging plugin, such as the maven-shade-plugin, that places all the class files of the project and dependencies into one JAR. Alternatively, the Nimbus deployment plugin can create multiple JAR files that contain the class files and dependencies of individual functions.

Considerations

The main advantage of the maven shade plugin is that all the required dependencies are guaranteed to be present in the shaded JAR. The disadvantages are that if there are many functions in the project, each with separate dependencies, then the function JAR can end up being very large. Large function JARs lead to slower cold start times, but most importantly many cloud providers have a JAR size limit (e.g. AWS has a 50 MB limit). Thus once this limit has been reached no more deployments can be done until the project is refactored, so that is built in separate chunks.

Determining Dependencies of Class Files

As part of the `nimbus-state` file, the class files of each serverless function are specified, along with any specific extra dependencies to include. The first step of assembly is determining the dependencies of each handler class file. These dependencies will then be packaged into the JAR of the function.

For the purpose of assembly, there are two types of dependencies. There are dependencies that come from the user's code, *user dependencies*, and there are dependencies from external libraries, *external dependencies*. As this is a maven plugin, it is assumed that all external dependencies can be found in the maven dependencies of the project.

Class files are used to process dependencies as the class file contains most of its dependencies as part of the constant pool. Thus to get the dependencies of a class file, the file is read and the constant pool obtained. Constant classes are obtained, as well as constant name and types, method types, and finally String and UTF-8 constants. The constant classes, constant name and types, and method types are guaranteed to correspond to a class, while the String and UTF-8 constants could be classes, files paths, or any other constants used. The String and UTF-8 constants are filtered down to only the class and file paths using a data structure defined in the next section.

Locating Class Files

To obtain the dependencies of a classpath, the class file for that class needs to be found so that the above processing can be done. If the classpath is a user dependency, the class file is located in the compiled sources folder of the project, and the classpath will directly relate to the file structure in the sources folder. For example, if the user writes a class with classpath `com.example.ExampleClass`, then that class will be found on the operating system at `sources/com/example/ExampleClass.class`. If the classpath corresponds to an external dependency, this is trickier as there is not a direct mapping of classpath to file location.

It is known that the external dependency will be located in the local maven repository. The local maven repository is comprised of maven artifacts, which correspond to the dependencies that the user marks in their maven POM file. A maven artifact is defined on a group ID (the organisation/individual, following reversed domain name conventions), an artifact ID (project name unique to the group), and a version number. The artifact is stored as a JAR file which contains all of its classes. The structure of the local repository that each artifact is placed in nested directories corresponding to its group ID, artifact ID and version. For example, the JAR file for an artifact with group ID `com.nimbusframework`, artifact ID `nimbus-core` and version `0.5` will be stored in `repo/com.nimbusframework.nimbus-core/0.5/nimbus-core-0.5.jar`.

Originally it was thought that it would be the case that any class stored in a maven artifact would have a classpath that would begin with the group ID and artifact ID. For example, a class path such as `com.nimbusframework.nimbuscore.SomeClass` would be stored in the local repository under the `repo/com/nimbusframework/nimbuscore/...` directories. To find the file, it was thought that the classpath could be split into the individual parts and then traverse down the directories until a match was found, for example, starting with `com/`, then `com/nimbusframework/` and so on. Immediately there are some challenges with this, for exam-

ple, the version is not accounted for and characters like `'-'` are removed from the classpath. There are fixes for this, mostly by looking at the exact dependencies of the project to determine the version and determining the closest match for missing characters. This still would not work as the assumption made before, that the prefix of the classpath will somewhat match the maven repository path, is completely wrong. After implementing the above fixes for the initial solution, many dependencies still could not be found. For example, the `com.amazonaws.adapters.types.TypeAdapter` class is found in `repo/com/amazonaws/ aws-java-sdk-core/aws-java-sdk-core.jar`. Instead, to actually determine the JAR file that an external classpath is found in, all the JARS have to be scanned through until the correct classpath is found. Obviously, this is inefficient, so is only done once, as described in the preprocessing section below.

Preprocessing

When the assembler is initialised a map of external classpaths to the JAR file they are located in is created, the *external dependency map*, to help find and determine external dependencies.

To create the external dependency map, first, the path of the local maven repository is obtained. Next, all the maven artifacts of the project are obtained, including transitive ones. The group ID, artifact ID, and artifact version are combined to obtain the location of the JAR file of that artifact in the local maven repository. This JAR file contains all the files of the artifact. All the entries of the JAR file are added to the external dependency map, with the entry key, which corresponds exactly to a classpath, mapped to the location of the JAR file.

Resource files are non-class files that need to be added to the JAR file, usually used for some form of configuration. These files are detected from the String and UTF-8 constants of the constant pool and specified as a file path. External dependency resources are found in the external dependency map, but local resources will be located in the local resources folder. This folder is scanned, and a map of the resource path to the operating system path is created, known as the *local resource map*. This processing is done at the start after the maven dependencies have been analysed.

Determining Dependencies of Functions

Once the external dependency and local resources maps have been created the dependencies of the handlers are determined. This starts with the top level class file of the serverless function. The dependencies of this class file are read, as described before, to obtain a set of classpaths and other files that the function depends on. Now the dependencies of each classpath in the set need to be obtained to determine transitive dependencies of the function.

For each classpath, it is first checked whether it is a local dependency by looking for the existence of the file locally in the build output directory. If it is, then this file is read directly, and a set of dependencies found. If it is not, then the external dependency map is checked for the location of the classpath. If it is contained in the map, then the JAR file is opened, and the corresponding entry found, again finding a new set of dependencies. If the classpath is not contained in the external dependency map, it is ignored, which is the case for many inbuilt Java classes. At each stage, a new set of dependencies are found, which are recursively processed until no new dependencies are found. Processing one dependency is relatively intensive as file I/O has to be done, so the dependencies of classpaths are cached so that this step can be skipped the next time the class is listed as a dependency. The caching applies for all function handlers of the project. This means that if functions share a large number of dependencies, after one has been processed, the others will be processed faster. The result of this stage is a set of dependencies for each handler file, known as the *file dependencies*.

JAR Creation

Now that the dependencies of each function handler file are known, the dependencies need to be added to a JAR for each handler file. One way to do this would be to process each handler file one at a time, locating the dependency and then adding it to the JAR file. This is very slow as JAR files will likely be opened multiple times, and if functions share dependencies, then individual class files will be opened multiple times. When the assembler was first written this one-at-a-time strategy was used and was very slow. The code was rewritten so that each dependency is only read from once to speed this up.

To read a dependency only once the file dependencies are converted to a map of an individual dependency to a list of target `JarOutputStreams`. This means that the dependency is read and then written to multiple JAR files. This solution can still jump back and forth between multiple JAR files, so to increase the efficiency the dependencies are grouped together based on the JAR they are located in. The resulting processing is then: JAR file is opened, all required entries are read and written to corresponding `JarOutputStream`, processing continues with the next JAR file. This grouping into JAR files is only done for external dependencies; local dependencies keep the map of dependency to a list of target `JarOutputStream`. This essentially means there are two maps used at this stage, the one for external dependencies that is grouped by JAR file, and one for local dependencies that is not grouped.

Issues with Reflection

At this point, JAR files for each of the function handlers can be created that contain only the direct class dependencies required for each function. This works when all dependencies are directly referenced in the class files. Where it doesn't work is when a dependency is created dynamically with reflection, for example, when a class is loaded from a String that is created via concatenation as in Listing 4.7. When the resource or classpaths are defined entirely in constants that are found in the constant pool, then the dependencies can be loaded in. The problem comes from the fact that the value cannot be determined at compile time.

```
1 public void loadClassFromProject(String className) throws ClassNotFoundException {
2     String package = "com.example.packageName";
3     Object clazz = Class.forName(package + "." + className);
4 }
```

Listing 4.7: Basic HTTP Function Example

There are two options the user has to help resolve this issue. If the dynamic dependencies are pulled in from the same artifact as the code that is failing, then the user can specify that any detected artifacts should be entirely packaged into the JAR. This results in bigger JARs but can still be smaller than when using the maven-shade-plugin as only artifacts used by the function are packaged. Alternatively, the serverless method can be annotated with `@ForceDependency` and a path specified. This path will then be highlighted as a dependency of the function in the nimbus-state file and will be packaged into the JAR along with any transitive dependencies.

One case where this dynamic class loading is done is when a JDBC driver is attempted to be loaded. Unfortunately, this affects any function attempting to use the `RelationalDatabaseClient`. When the annotation processor comes across an `@UsesRelationalDatabaseClient` annotation, the corresponding database driver is specified in the `nimbus-state` file as an extra dependency of the function. This solves the issue efficiently, without the user needing to specify the driver as a dependency manually. The assembler then adds the dependency to the file dependencies, and it is processed with everything else.

Issues with ClientBuilder

One interesting case to try to optimise for was the `ClientBuilder` class. As this class creates all the different types of clients, for each cloud resource, it was noticed that any function that used this

client would end up pulling many more dependencies than were actually needed. To solve this, when dependencies are being processed, it is checked if the dependency is the `ClientBuilder` class. If the dependency is not a `ClientBuidler`, then processing carries on as mentioned above, but if it is then no dependency processing is done on the class file. Instead, the dependencies are marked manually based on the permissions annotations on the function. As the permissions annotations have to be there for the client to work, they specify the exact clients and therefore, dependencies that are required by the function. The annotation processor marks the function as using the client and stores this information in the `nimbus-state` file. When the `ClientBuilder` class is found as a dependency, then the specified clients are processed further for their dependencies, and thus, unused clients are ignored.

Due to the possibility of users forgetting the permission annotation, it was desired that a more meaningful exception than `ClassNotFoundException` would be thrown if a client was attempted to be created and then used. Due to the requirement that clients can be created in any case, even without permissions, so long as they aren't used, the exception can't just be caught and then a more meaningful error thrown. To solve this, 'empty' clients were defined that have no dependencies and only throw a `PermissionException` if one of their methods is invoked. This alerts the user that if they want to use the client, the correct permission annotation must be added. If a `ClassNotFoundException` is caught when clients are attempted to be instantiated by the `ClientBuidler`, then the corresponding empty client is returned. Additionally, when dependencies are being processed, the empty clients are added as manual dependencies of the `ClientBuilder` class.

Summary

The overall processing steps are:

- Process maven dependencies
- Process local resources
- Determine the set of dependencies of each handler
- Transform the set of dependencies into a map of the path to a target JAR
- Use this map to create each JAR file

With all this done, the assembler can work for all functions; however, in some cases, additional configuration is needed. For this reason, the maven shade plugin is also supported, as it is guaranteed to work if the Maven POM file is configured correctly.

4.4.2 Stack Creation

In CloudFormation, each Lambda function specifies it's code location as a file in an S3 bucket. Now, if the CloudFormation stack has not been created, then this S3 bucket does not exist, and the full CloudFormation stack cannot be created as the functions would fail. This is the reason a create template and an update template are created. The create template just creates the S3 bucket resource, and the update bucket creates all the resources. If the stack doesn't exist in the cloud, then the create template is built, the code (JARs) uploaded, and finally, the update stack built. If the stack does exist, then the S3 bucket exists, and the code can be uploaded and then the update stack built, without the create stack needing to be built.

Each template is built using the AWS CloudFormation Java SDK. The template file is uploaded, and then progress is polled until the operation is completed.

Before each template can be uploaded, the version of the Lambda code being uploaded is substituted into the template file. If the maven-shade-plugin is used, then there will only be one

JAR, and if Nimbus does the assembly, there will be multiple JAR files. The version corresponds to the file path of the uploaded functions in the S3 bucket.

4.4.3 File Uploads (Cloud)

Once all the cloud resources have been created from the update template, any files specified by a `@FileUpload` annotation needs to be uploaded. This information is stored in the `nimbus-state` file as part of the *file upload description*. This specifies the local file path, target file path, and whether or not to substitute variables.

To obtain the values of the variables to substitute the stack outputs are queried and the results placed into a variable substitution map, similar to the local deployment variable substitution map. The substitutions and uploads are also done in the same manner as in the local deployment, except now the files are uploaded to S3 instead of to a local bucket.

4.4.4 After Deployment Functions

After any file uploads have been done, the after deployment functions are invoked. The after deployment functions are deployed as Lambda functions, and their names are specified in the `nimbus-state` file. The AWS Lambda SDK is then used to invoke the individual functions.

After these functions are invoked, the deployment is complete.

4.4.5 Deployment Optimisation

If Nimbus is used to assemble the project, this can increase the time taken to deploy due to having more file I/O, as well as uploading more data. Every time the project is assembled and deployed, the assembly and uploading have to be done for each function, regardless of if it actually changed. Thus an optimisation is only to assemble and upload functions that have changed.

A method of determining a function signature was defined so that the signature changes if the code of the function changes or any of its dependencies change. This signature is determined in a similar way as the root node value of a Merkle Tree, where every node's value is the hash of its children. Similarly, the signature of a function is a hash of its dependencies and the function class file itself. Specifically, a hash SHA-256 class file is created, and then the signatures of all dependencies of the class file concatenated together. The dependency signatures are sorted before being concatenated, to ensure that ordering doesn't change the signature, to produce the *dependencies hash*. The SHA hash and dependencies hash are then concatenated together. This long string could be used as the signature as it matches the behaviour required, however, to keep the signatures from being able to grow indefinitely, one final MD5 hash is taken of this long string to keep the length constant. Determining the dependencies of the function is done in the same way as the Nimbus assembler.

A file in the deployment S3 bucket is created that defines the most recent hash of each function. Before processing is done, the current signatures of the local functions are determined and then compared against the values in this file. Only functions that are not defined in the file, or have a different signature of the one in the file, are further processed. This can drastically improve the deployment time of projects when changes are being made to only a few of the functions.

Chapter 5

Building Applications with Nimbus

This chapter demonstrates how to build and deploy an application with Nimbus. The aim of this is to demonstrate the general workflow when writing an application using Nimbus.

5.1 The Application

The application will be a simple chat application that will have a web frontend that allows users to 'log in' and send messages to other online users. The sending of messages will be handled by a WebSocket backend API. In addition, registering new usernames will be processed by a REST API. All of the backend code will be written in Java using the Nimbus framework. Code snippets will be provided for clarity and to illustrate points, and the full code can be found in the examples GitHub repository[1]. Figure 5.1 shows an image of the application running in the cloud.

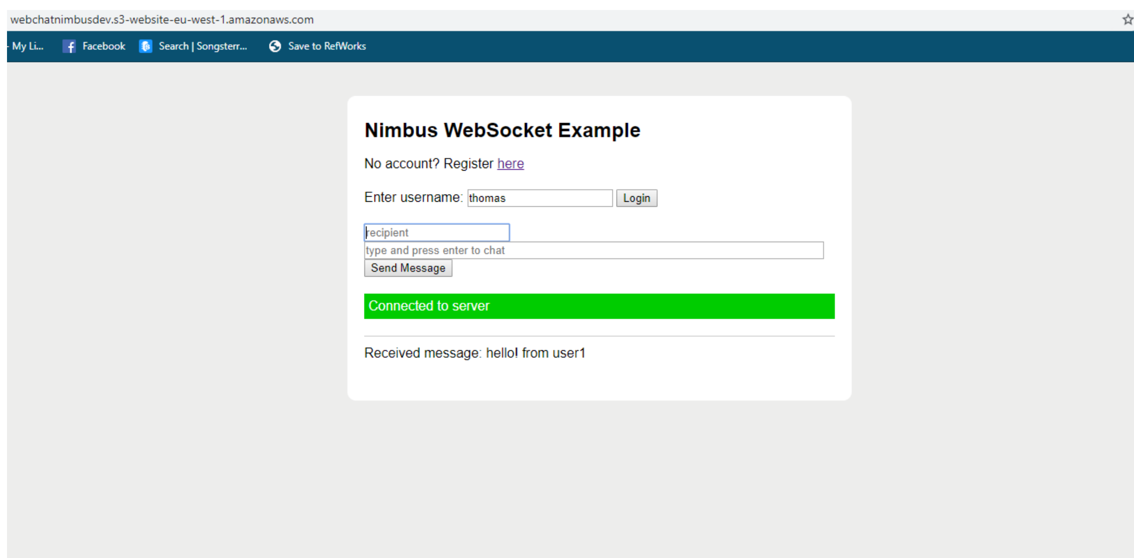

Figure 5.1: WebSocket chat application running in the cloud. A message 'hello!' has just been received from user1.

[1] https://github.com/thomasjallerton/nimbus-examples

5.1.1 Overall Architecture

The web frontend will be hosted in a file storage bucket. The WebSocket API will have three functions: one to handle connections, one for disconnections, and one for sending messages. The REST API will have one function that registers new users. Due to the stateless nature of the serverless functions, data stores are needed to persist data. A key-value store will be used to map WebSocket connection IDs to usernames, and a document store to store data on users, including their current active WebSocket. Figure 5.2 shows an architecture diagram for the application.

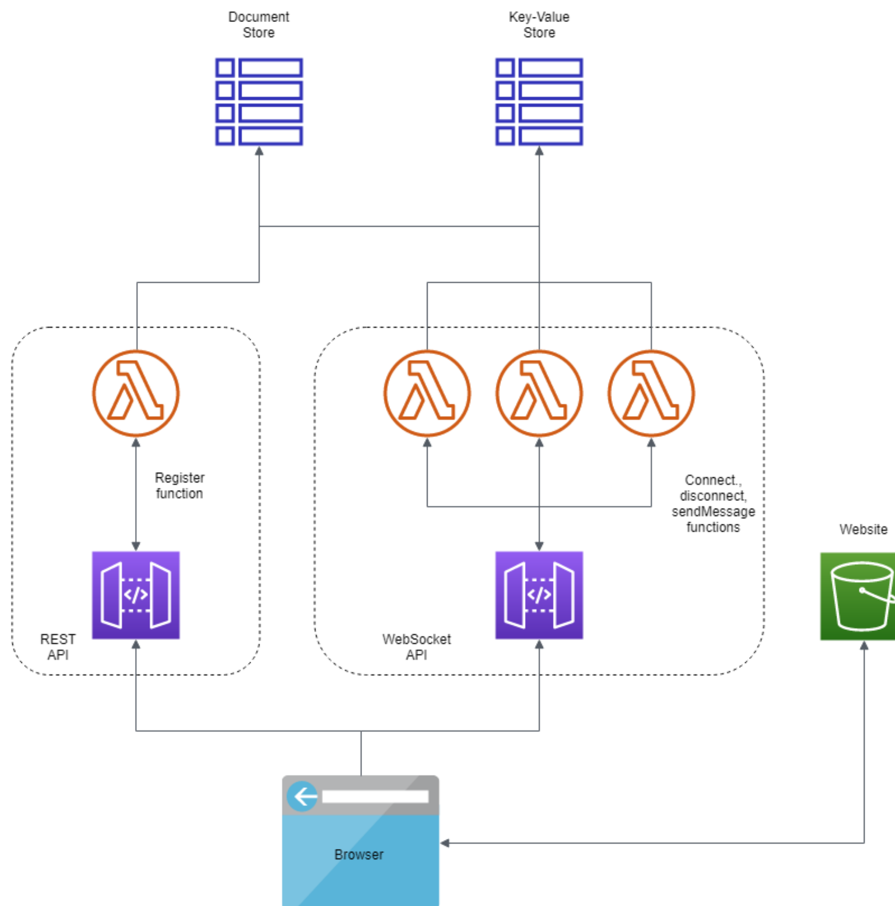

Figure 5.2: Architecture diagram for the WebSocket chat application.

5.1.2 The WebSocket API

Three WebSocket topic endpoints are needed for the API. These are `$connect`, to handle incoming WebSocket connections, `$disconnect`, to handle WebSocket disconnections (on a best-effort basis) and finally `sendMessage` to send messages between red, only sent between active users.

Listing 5.1 shows the configuration required to deploy the key-value store `ConnectionDetail`, and Listing 5.2 shows theguration required to deploy the document store `UserDetail`.

```
@KeyValueStore(
        keyType = String.class,
        stages = {DEV_STAGE, PRODUCTION_STAGE}
)
public class ConnectionDetail {
    @Attribute
    private String username = "";
    // Constructors, getters and setters
}
```

Listing 5.1: ConnectionDetail key-value store

```
@DocumentStore(stages = {DEV_STAGE, PRODUCTION_STAGE})
public class UserDetail {

    @Key
    private String username = "";

    @Attribute
    private String currentWebsocket = null;

    // Constructors, getters and setters
}
```

Listing 5.2: UserDetail document store

Now the actual WebSocket API can be written. On a connection, first the user is ensured to be valid, that is, they have an entry in the `UserDetail` store. The `UserDetail` store is then updated with the active connection ID, and then an entry is added to the `ConnectionDetail` store that points the connection ID to the username.

The `sendMessage` endpoint, shown in Listing 5.3, will send a message from one user to another if the recipient has an active connection. First, the sender and recipient details are obtained, and then the message is sent to the recipient's connection id. This requires a few extra models, a `WebSocketMessage` model that the backend receives, and a Message model that the backend sends to the recipient.

```
public class WebchatApi {

    private ServerlessFunctionWebSocketClient webSocketClient = ClientBuilder
            .getServerlessFunctionWebSocketClient();
    private DocumentStoreClient<UserDetail> userDetails = ClientBuilder
            .getDocumentStoreClient(UserDetail.class);
    private KeyValueStoreClient<String, ConnectionDetail> connectionDetails =
            ClientBuilder.getKeyValueStoreClient(
                String.class,
                ConnectionDetail.class);

    @WebSocketServerlessFunction(topic = "sendMessage", stages = {DEV_STAGE, PRODUCTION_STAGE})
    @UsesServerlessFunctionWebSocketClient(stages = {DEV_STAGE, PRODUCTION_STAGE})
    @UsesDocumentStore(dataModel = UserDetail.class, stages = {DEV_STAGE, PRODUCTION_STAGE})
    @UsesKeyValueStore(dataModel = ConnectionDetail.class, stages = {DEV_STAGE, PRODUCTION_STAGE})
    public void onMessage(WebSocketMessage message, WebSocketEvent event) {
        UserDetail userDetail = userDetails.get(message.getRecipient());
        ConnectionDetail connectionDetail = connectionDetails.get(
            event.getRequestContext().getConnectionId()
        );
        if (userDetail != null && connectionDetail != null) {
            webSocketClient.sendToConnectionConvertToJson(userDetail.getCurrentWebsocket(),
                    new Message(message.getMessage(), connectionDetail.getUsername()));
        }
    }
}
```

Listing 5.3: WebSocket function for sendMessage topic

The `$disconnect` endpoint is relatively simple. The corresponding entry is removed from the `ConnectionDetail`, and the user's active connection in `UserDetail` is set to null.

Two stages are defined, one for development and live environment testing, and one for actual end users to use. This allows for endpoints which can be used for debugging and testing, which are implemented as part of the REST API.

5.1.3 The REST API

The REST API will provide an endpoint to allow new users to register, as well as some endpoints that will help in testing and debugging the application in the development environment.

In this basic application, users only need to provide a username to register. This registration endpoint will be a POST request to the register path, with the username as a JSON String in the request body. This endpoint will only update the `UserDetail` store, adding a new entry. The code for the REST API is shown in Listing 5.4.

```java
public class WebchatRestApi {

    private DocumentStoreClient<UserDetail> userDetails = ClientBuilder.getDocumentStoreClient(
     UserDetail.class);
    private KeyValueStoreClient<String, ConnectionDetail> connectionDetails = ClientBuilder.
     getKeyValueStoreClient(String.class, ConnectionDetail.class);

    @HttpServerlessFunction(method = HttpMethod.POST, path = "register", stages = {DEV_STAGE,
     PRODUCTION_STAGE}, allowedCorsOrigin = "${WEBCHATNIMBUS_URL}")
    @UsesDocumentStore(dataModel = UserDetail.class, stages = {DEV_STAGE, PRODUCTION_STAGE})
    public void register(String username) {
        userDetails.put(new UserDetail(username, null));
    }

    //For development stage - Create users
    @AfterDeployment(stages = {DEV_STAGE})
    @UsesDocumentStore(dataModel = UserDetail.class, stages = {DEV_STAGE})
    public String setupBasicUsers() {
        UserDetail user1 = new UserDetail("user1", null);
        UserDetail user2 = new UserDetail("user2", null);

        userDetails.put(user1);
        userDetails.put(user2);
        return "Created test users";
    }
}
```

Listing 5.4: REST API for chat application

The register function implements the registration functionality of the API. The rest of the functions are there to aid in testing. `setupBasicUsers` creates two users after the application has been deployed that can be used immediately, with no need to register. `getUserDetails` and `getConnectionDetails` return the contents of the `UserDetail` and `ConnectionDetail` stores, respectively.

Note the `allowedCorsOrigin` parameter in the `HttpServerlessFunction` annotation of register. As the client, hosted on a file storage bucket, needs to access the REST API, which has a different origin, some Cross Origin Requests need to be enabled. Only the client should be able to register users from the browser; therefore, the file storage bucket URL is specified as an allowed origin.

5.1.4 The Client (Frontend)

The client will be hosted in a file storage bucket as a static website. Using Nimbus, a directory in the project is specified to be uploaded into the file storage bucket.

One feature of note is that when the URL to connect to the WebSocket API is declared a variable parameter can be supplied, as in Listing 5.5. In many typical applications, the backend APIs would have to be deployed first so that the URL can be inserted into the frontend. By using this variable, the actual URL is substituted when the file is uploaded to the file bucket. When deployed locally, this value is set to the local deployment URL.

```javascript
function connectToServer() {
    let wsUri = "${NIMBUS_WEBSOCKET_API_URL}?user=" + username.value;
    console.log(wsUri);
    websocket = new WebSocket(wsUri);
}
```

Listing 5.5: Connection code for javascript client

5.1.5 Testing

First, unit tests could be written to ensure that the business logic of the application works. For example, to ensure that on a connection the user is added to the `ConnectionDetail` store, as shown in Listing 5.6. Next, integration testing can be done to ensure that the interaction of resources works and that everything has the correct permissions. Alternatively, a full local deployment could be done and the application interacted with locally.

```
public class WebSocketApiTest {

    @Test
    public void onConnectAddsUserToConnectionDetail() {
        LocalNimbusDeployment localNimbusDeployment = LocalNimbusDeployment.getNewInstance("com.
    nimbusframework.examples.webchat");
        Map<String, String> queryStringParams = new HashMap<>();
        queryStringParams.put("user", "user1");
        Map<String, String> headers = new HashMap<>();
        localNimbusDeployment.connectToWebSockets(headers, queryStringParams);
        ServerlessMethod method = localNimbusDeployment.getMethod(WebchatApi.class, "onConnect");
        assertEquals(1, method.getTimesInvoked());
        LocalKeyValueStore<String, ConnectionDetail> connectionDetailStore = localNimbusDeployment.
    getKeyValueStore(ConnectionDetail.class);
        assertEquals(1, connectionDetailStore.size());
        Map<String, ConnectionDetail> connectionDetails = connectionDetailStore.getAll();
        for (ConnectionDetail connectionDetail : connectionDetails.values()) {
            assertEquals("user1", connectionDetail.getUsername());
        }
    }
}
```

Listing 5.6: Unit test for chat application

5.1.6 Cloud Deployment

After testing has been completed, and any errors fixed, the application can be deployed. The code is compiled, and then the deployment command, `mvn nimbus-deployment:deploy` is run. Once this has been completed, the deployment plugin will report the URL of the chat application, which can then be used in the cloud.

5.2 Remarks

The code written here is intuitive and straightforward, partly because much of the boilerplate code, such as the serialisation, deserialisation, and store interaction, is abstracted away. The local testing features also makes it very easy to quickly test the application, as well as being able to catch many common mistakes. For example, if the user forgot a permission annotation for the connect function, then the test in Listing 3.21 would have failed. Additionally, if the user forgot the CORs parameter to the HTTP function, then running the application locally would highlight this issue. This means that now, instead of performing multiple deployments, the application can be tested locally, and then only one deployment done.

Now when looking at the code, thanks to the annotations, it is obvious what the role of the functions are in the cloud. Document and key-value stores are simple to define, and the API intuitive to use as they can be thought of as sets and maps.

Chapter 6

Evaluation

6.1 Performance

A key aspect of the framework is that the increased flexibility and generality does not sacrifice the performance of the cloud functions. This is critical in two aspects: the function handlers and the clients.

6.1.1 Handler Performance

Equivalent functions were written using the Nimbus framework and the AWS Lambda SDK, for each of the serverless handlers, such as HTTP functions and notification functions. The function bodies were themselves empty, save for any deserialisation into a `DataModel` input type, and serialisation from a `ResponseModel` response type. As an example, the compared HTTP functions can be found in Listing 6.1 (AWS) and Listing 6.2 (Nimbus).

```
1  public class HttpFunction implements
2      RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent>
3  {
4      public APIGatewayProxyResponseEvent handleRequest(
5          APIGatewayProxyRequestEvent requestEvent, Context context)
6      {
7          try {
8              ObjectMapper mapper = new ObjectMapper();
9              DataModel dataModel = mapper.readValue(
10                 requestEvent.getBody(),
11                 DataModel.class
12             );
13             String response = new ObjectMapper()
14                 .writeValueAsString(new ResponseModel(true));
15             return new APIGatewayProxyResponseEvent()
16                 .withBody(response)
17                 .withStatusCode(200);
18         } catch (Exception e){
19             e.printStackTrace();
20             return new APIGatewayProxyResponseEvent().withStatusCode(500);
21         }
22     }
23 }
```

Listing 6.1: AWS HTTP Function

```
1  public class HttpFunction {
2
3      @HttpServerlessFunction(method = HttpMethod.POST, path = "evaluation")
4      public ResponseModel httpFunction(DataModel model, HttpEvent httpEvent)
5      {
6          return new ResponseModel(model.getUsername(), model.getFullName(),true);
7      }
8  }
```

Listing 6.2: Nimbus HTTP Function

The average response time of the function was measured for 10 invocations. This response time is the billable function time reported in the AWS, specifically in the CloudWatch logs of the corresponding Lambda function. Cold starts are many times slower than a typical invocation, and also are determined by factors other than the function code, like package size. Thus to avoid skewing the results, and introducing factors other than the handlers themselves, cold starts are ignored. In practice, this means that the function is triggered 11 times, with the result of the first invocation ignored. The results can be found in Table 6.1. Note that both document store functions and key-value store functions fall under the DynamoDB function type.

	Average Response Time (ms)	
Function Type	**AWS**	**Nimbus**
Basic	0.503	0.475
HTTP	2.416	2.415
WebSocket	6.599	6.613
DynamoDB	0.946	1.7
Notification	1.613	1.654
Queue	1.421	1.506
File Storage	2.357	2.31

Table 6.1: Average response time of different handlers.

Conclusion

It can be seen that the Nimbus handlers have similar performance for most function types, with the minor performance differences likely due to variance in the cloud environment.

The only significant difference is in the DynamoDB functions, with Nimbus being 79.70% slower. This is due to the extra work required by Nimbus to detect what fields of the `DataModel` class are annotated, and then convert the DynamoDB JSON format into a new Java object. In comparison, the AWS function only hard codes the deserialisation by extracting specific fields and then using those to create the Java object. In terms of billing, this performance difference is negligible, as the billing time slice is 100ms, and only in rare cases will this affect the amount charged.

The WebSocket handlers are much slower than all the other functions. In Subsection 2.3.1 it was mentioned that Java Lambda handlers could be written as a stream handler or as a method handler. It was also mentioned that stream handlers appear to be slower, which this is an example of. At the time of writing, AWS does not support method handlers for WebSocket functions, so stream handlers have to be written, and the deserialisation performed manually.

6.1.2 Client Performance

Basic functions were written for each type of Nimbus client, and then equivalent basic functions written using the AWS client. In the Nimbus functions, a client was initialised, and then each of the operations performed, with each operation being timed. In the AWS functions the AWS client was initialised, and then equivalent operations performed as the Nimbus counterpart, with each operation timed. As the client performance measured is independent of the handler, Nimbus was used to deploy all functions as it allows permissions to be configured much easier.

To measure the performance, the time taken to invoke a method was averaged, for a sample size of 10 invocations. The results can be found in Table 6.2.

Conclusion

In most cases, the Nimbus clients perform the same as the AWS clients, which is not surprising due to the Nimbus clients simply being an adapter for the AWS clients. The only notable difference in

Dynamo Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	6.25	8.17
INSERT	63.22	66.34
GET	8.21	8.67
DELETE	8.13	10.98

WebSocket Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	16.29	0.08
SEND	81.39	87.23

File Storage Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	7.36	5.97
SAVE	70.54	77.72
READ	8.86	11.27
DELETE	67.52	52.66

Basic Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	8.70	8.67
SYNC	67.07	62.34
ASYNC	31.72	37.16

Notification Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	4.44	5.75
SUBSCRIBE	174.32	169.72
NOTIFY	24.92	25.98
UNSUBSCRIBE	88.20	92.45

Environment Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	0.0002	0.0022
GET	0.0102	0.0116
CONTAINS	0.0078	0.0076

Database Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	0.00	0.00
CONNECTION	68.11	56.46

Queue Client Performance		
Operation	**AWS (ms)**	**Nimbus (ms)**
INITIALISATION	6.28	7.72
ADD	71.49	63.52

Table 6.2: The average response time of different clients for each method.

performance is the WebSocket client initialisation. The reason that Nimbus is much faster is that the client does not initialise the underlying AWS client until a method has been invoked, which is why the SEND method is consistently slower.

6.1.3 Performance Summary

The results have shown that Nimbus has similar performance, with at most a negligible difference, to using AWS code directly. Therefore the user does not sacrifice performance when using Nimbus.

6.2 Development Experience

In order to evaluate whether Nimbus has a positive effect on developers, different use cases and functionality are compared with existing solutions.

6.2.1 Amount of Code Written

One of the pain points of writing serverless applications before was the amount of boilerplate code required to do many tasks, especially serialisation and deserialisation. A straightforward evaluation method is comparing the amount of code written to do each task, for handlers and clients.

The lines of code for equivalent handlers were found, and lines of code for equivalent client operations. In the Nimbus handler case, there were no permission annotations, and the function annotations are counted. In the Nimbus client case, permission annotations are counted as lines of code, but function annotations are not, as the function annotations are counted in the handler case. Files generated by Nimbus are also not counted. The results of this are found in Table 6.3.

	Lines of Code	
	AWS	**Nimbus**
Function Handlers		
Basic	5	6
HTTP	13	6
WebSocket	24	6
Dynamo	13	6
Notification	11	6
Queue	11	6
File Storage	5	6
Clients		
Basic	17	9
WebSocket	11	8
Dynamo	14	9
Notification	6	7
Queue	4	5
File Storage	7	7
Database	3	3
Environment Variables	1	2
Total	145	92

Table 6.3: Lines of code for equivalent handlers and clients.

	Configuration Lines			
Project	**CloudFormation**	**Serverless**	**SAM**	**Nimbus**
A	1163	69	267	15
B	1013	164	65	20

Table 6.4: Lines of configuration needed to deploy cloud resources.

Conclusion

Nimbus, in most cases, reduces the amount of code needed to be written by a fair amount compared to writing AWS code. The only exceptions are basic and file storage handlers, and notification, queue, and file storage clients. The reason for these exceptions is that, the two AWS handlers are as simple as possible to start with, these AWS clients are simple to start with, and the annotations are counted in the Nimbus case. If the annotations are not counted, then Nimbus always performs the same or better than the AWS equivalent.

6.2.2 Amount of Configuration Written

Similar to the amount of code written, there is a certain amount of configuration that needs to be written to deploy the application. In AWS this is the CloudFormation file and in the Serverless framework the serverless.yml configuration file. In Nimbus, the configuration is done via the annotations. Two different project configurations are shown in Table 6.4 that compare the lines in Nimbus, Serverless, and AWS needed for configuration of a deployment. Project A has a function for each type of handler, and if that handler relies on an external resource, then that resource is also deployed. For example, a queue is deployed for a queue handler, a DynamoDB table for a document store function, and so on. Project B has one function that uses every single type of client. If the client needs an external resource, then that resource is also deployed. This includes a DynamoDB table and a relational database. Project B needs to have the permissions configured so that the function can interact with all the resources.

	Time Caught	
Error	**Serverless/AWS**	**Nimbus**
CloudFormation Dependency Error	During Deployment	Prevented
Invalid Value or Unsupported Resource Property	During Deployment	Prevented
Some AWS Limit Exceeded	During Deployment	During Deployment
Spelling Mistake When Referencing Resource	During Deployment	Prevented
Specified the Wrong Language	Function Runtime	Prevented
Specified the Wrong Handler Method	Function Runtime	Prevented
Invalid Permissions	Function Runtime	Using Nimbus Tests
CORs errors	Function Runtime	Using Nimbus Tests
Functions interacting with wrong resource	Function Runtime	Compile / Local Nimbus Tests
Deserialisation Errors	Custom tests are written	Natively in Nimbus Tests

Table 6.5: Some common errors and when they are caught using different frameworks.

Conclusion

The results of the CloudFormation, SAM and Serverless are more rough estimates as in these files formatting plays a significant role in the total number of lines. Even with this taken into account, Nimbus still drastically reduces the amount of configuration needed to deploy the required cloud resources.

6.2.3 Testing Cycle

Another pain point of developing and deploying serverless applications was the amount of time it took to find errors, simple or not. To compare this time to find errors, Table 6.5 shows some common errors when deploying code to AWS, and compares at what stage the error can be caught using Serverless/AWS and Nimbus.

For reference, a single deployment error can take minutes to discover as the resource creation can take minutes. This adds up as only one error is reported at a time so multiple attempts may be required. Additionally, on an error being found any updates done up to that point have to be rolled back, slowing down the progress. Function runtime errors are discovered after the deployment has successfully completed, so again can take minutes to find, assuming there are no deployment errors.

In comparison, Nimbus local deployments take seconds to start up, and Nimbus tests take seconds to run, assuming they have been written.

Conclusion

Many of the problematic errors that could take minutes to find and solve are now either eliminated entirely or can be found in seconds with Nimbus. In the cases where errors are prevented in Nimbus, the time saved by not having to figure out the root cause of the error, beyond just discovering that it exists, can also be considered. Another benefit of local deployment is that the user's code can be tested locally, to find any logic errors, in seconds compared to waiting minutes for the code deploy. When all of these factors are considered, Nimbus reduces the amount of time spent deploying and debugging the code.

There are still some errors that Nimbus can't find quicker than Serverless or AWS, like AWS Limits being reached. This could be that the user has reached the limit of relational database

instances. Nimbus can't find this error quicker due to it being due to an external factor which cannot be easily queried.

6.2.4 Vendor Lock-in & Functionality

Vendor Lock-In

The framework can be defined as cloud agnostic as when the user writes their code there is no need to have any knowledge of the underlying AWS resources and code, and the AWS portion could be swapped out for another cloud provider without any change to the user code. In this regard, the framework is a success; however, trade-offs had to be made in some cases that reduces functionality, especially with regards to clients.

Data Lock-In

Although the code itself can be said to be cloud agnostic, and avoid vendor lock-in, there is another form of vendor lock-in: data lock-in. Data lock-in is when data is stored in one cloud provider and is challenging to move to another cloud provider. In its current form, Nimbus does not protect the user from data lock-in as there is no easy way to transfer data to another cloud provider. In the future, Nimbus could support transitioning data and services across cloud providers in a way that ensures data is not lost.

Functionality

The clients written for the Nimbus framework aim to only interact with one resource and not do any management that would alter the CloudFormation stack. However, AWS clients support interacting with resources as well as managing the resource. AWS management is very AWS specific, so it would be challenging to implement in a way that would be cloud agnostic, and for this reason, the management portion of clients was ignored. Table 6.6 compares the functionality of clients, excluding the management functionality of the AWS clients, and any duplicate methods that perform the same operation but with different arguments.

Conclusion

In all cases, the Nimbus clients retain the core functionality of the AWS clients; however, in a couple of cases, some functionality is lost.

One of these cases is the queue client. The reason the delete message and change message visibility methods were not implemented was that it was deemed these features were unnecessary, as a queue in Nimbus will always have a function consumer, so the messages will always be consumed. One case where this reasoning falls apart is when the item on the queue causes the function to fail, and the queue has to repeatedly retry invoking the function until the item ends up on the dead letter queue. Support for delete could be useful and added in a future update. Receive message was not implemented as a function defined in the project will already be consuming items from the queue, and the processing of the queue item can then be done there.

Another case is the DyanmoDB client. Query and scan are operations that return multiple items from the table. Queries are performed on the primary keys and are thus quick, and scans are performed on the attributes and can be much slower in large tables. As the chosen APIs for DyanmoDB tables were document stores and key-value stores, similar to the Java `Set` and `Map` collections, it was decided for these implementations not to enable this query like behaviour. In the future, a new, more relational database table like interface could be implemented that would support the scan and query features. One problem with scans is at large scales they are not as

Queue Client	
AWS	**Nimbus**
sendMessage	addMessage
changeMessageVisibility	
deleteMessage	
recieveMessage	

Basic	
AWS	**Nimbus**
invoke	invokeSync
invoke (async params)	invokeAsync

DynamoDB Client	
AWS	**Nimbus**
insert	insert
get	get
delete	delete
query	
scan	

File Storage Client	
AWS	**Nimbus**
getObject	getFile
putObject	saveFile
deleteObject	deleteFile
listObjects	listFiles
copyObject	

Notification Client	
AWS	**Nimbus**
subscribe	createSubscription
publish	notify
unsubscribe	deleteSubscription
listSubscriptions	

WebSocket Client	
AWS	**Nimbus**
postToConnection	sendToConnection

Environment Variable Client	
AWS	**Nimbus**
getEnv	getEnv
containsKey	containsKey

Relational Database Client	
AWS	**Nimbus**
getConnection	getConnection

Table 6.6: Comparison of available methods in Nimbus and AWS clients. Methods that perform the same operation but with different arguments have been ignored.

efficient as relational database tables. If a user wants scan functionality then to discourage them from an efficient solution, the scan functionality of DyanmoDB could be hidden, and the user directed to a relational database solution.

6.2.5 Developer Experience Summary

This section has shown that using Nimbus will result in fewer lines in the project, especially configuration lines. This combined with errors either not being made or being reported much earlier in the testing cycle will result in faster development time, with very little loss of functionality. The result of all this should result in a better developer experience.

6.3 Local Testing

To evaluate the local testing, the behaviour of the local and cloud deployments are compared, with the aim for them to be the same. The deployment times are also compared to demonstrate that it is much faster to deploy locally.

6.3.1 Function Handlers

A handler for each type of serverless function was written. These handlers made assertions based on the input types to ensure that data had been deserialised correctly. Unit tests were then written to invoke each of these functions, and assertions made to assert that responses had been serialised correctly. These tests did not invoke the functions directly, except for in the case of the basic function, but by interacting with their associated resources. All the tests passed, showing that serialisation and deserialisation had been done as expected. The functions were then deployed to the cloud and then invoked in the same way before, by interacting with the associated resource. Manual verification was done to verify that the assert statements in the functions had passed and that any responses were deserialised correctly. All the functions passed this manual verification, which shows that the local functions correctly simulate the cloud functions with simple invocations.

One known complex function interaction is that when any document is changed, any document store functions associated with that store are invoked, regardless of the type of change. Similar behaviour is present in key-value stores and their functions. A store was updated in local tests and assertions were made to ensure that INSERT, MODIFY, and REMOVE functions were triggered. These tests passed for both document and key-value stores.

One challenging design decision was how to handle exceptions thrown by some of the serverless functions. In AWS, some functions are retried if an exception is thrown. This is useful in the cloud if the exception is erroneous, for example, a function timing out due to a cold start, or some other non-deterministic behaviour. Locally, these retries will just keep throwing the same exception, unless the user's code is non-deterministic. In the end, it was decided not to retry functions locally, and to just throw the exception up to the user so they can fix the issue.

6.3.2 Resources and Clients

Basic functions were written for each type of client. These functions performed all the operations on the client, similar to the client performance test. Assert statements were added in cases where a response was expected to verify the data. For example, after a get on a document store to check that the obtained item was the one expected. Local tests were written to check the environment before and after the clients had run. For example, the document store client test invoked the methods in the order of, insert, get, and then delete, all operating on the same object. The local tests then checked that the document store was empty to start with, and empty afterwards. All

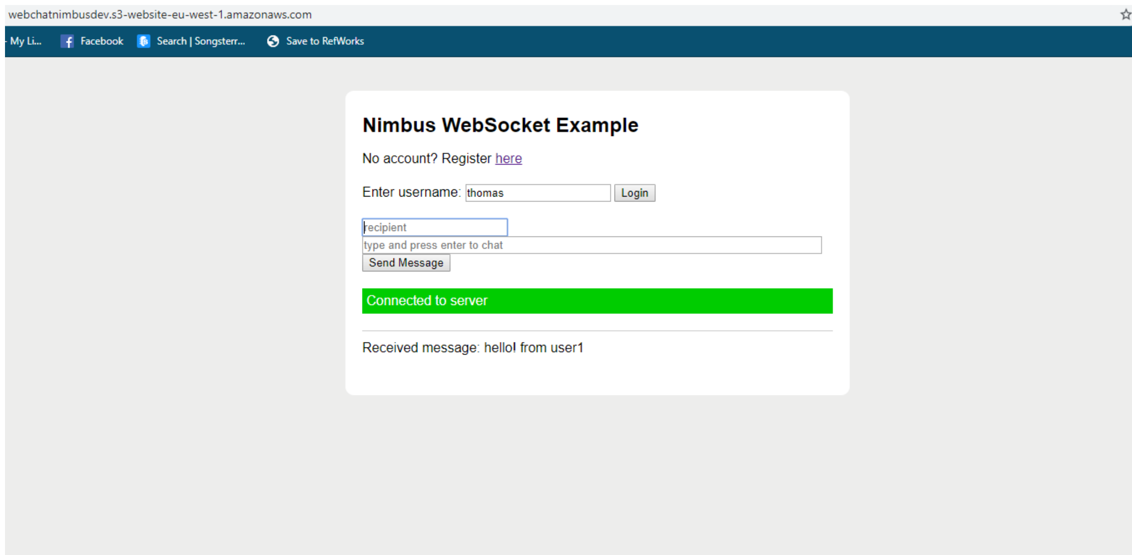

Figure 6.1: WebSocket chat application running in the cloud.

WebSocket Chat Deployment Times (seconds)		
Nimbus maven-shade-plugin	**Nimbus Assembly**	**Nimbus Local**
234	383	1

Table 6.7: Deployment times of WebSocket chat application.

these tests passed locally, showing that the clients interacted successfully with local resources. The functions were then deployed to the cloud, and then the initial conditions verified manually. Once these matched the same as the local deployment, the functions were then run. The functions were manually verified to have passed the internal assertions, and then manual verification was done on the cloud environment. Again, it was found that the cloud deployment matched the behaviour of the local deployment.

One limitation of local deployment compared to the cloud deployment is that the cloud environment provides a web console to interact with all the resources. In the local deployment, the only way to interact with the resources is via the clients, which need to be run in a function or a unit test. Future work could be to implement a web console for the local deployment.

6.3.3 Deployment Times

The simple application written in Chapter 5 was taken, and the local and cloud deployments compared. Figure 6.2 shows the application running locally, and Figure 6.1 shows the application running in the cloud. Upon inspection, no discernible difference in behaviour was found between the two. The deployment times were then measured, and it was found that the local deployment took a fraction of the time to perform than the full deployment (Table 6.7). The local deployment time applies for unit tests as well as the full local deployments.

6.3.4 Supported Resources

A comparison was made between each of the frameworks that support local deployments to highlight the resources that can be deployed locally. Table 6.8 summarises the results.

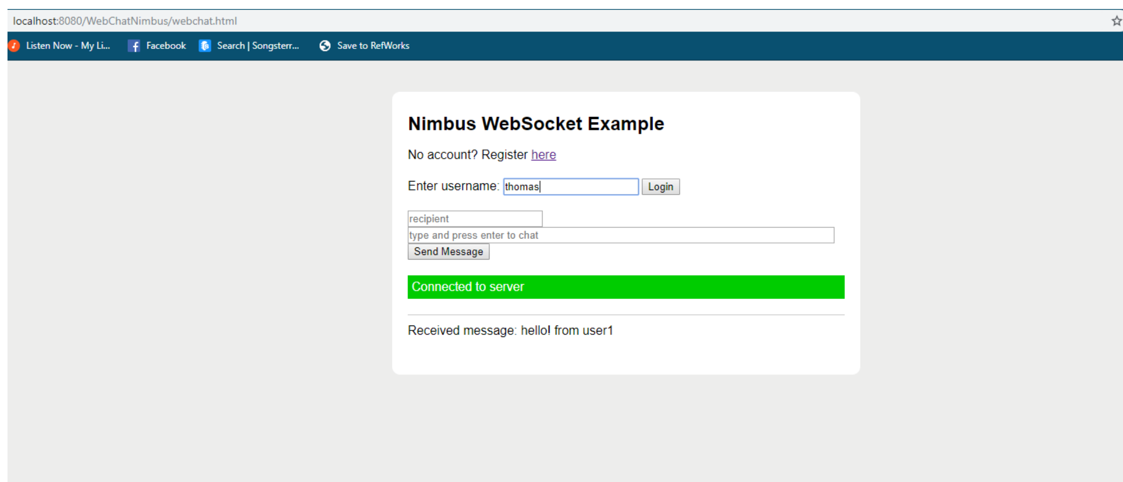

Figure 6.2: WebSocket chat application running locally.

	Can Deploy Locally		
Resource	**SAM**	**Spring Cloud**	**Nimbus**
HTTP Functions	Yes	Yes	Yes
WebSocket Functions	No	No	Yes
Basic Functions	Yes	Yes	Yes
File Storage Bucket	No	No	Yes
Document/Key-Value store	No	No	Yes
Notification Topic	No	No	Yes
Queue	No	No	Yes
Relational Database	No	No	Yes

Table 6.8: Resources that frameworks can deploy locally.

It can be seen that Nimbus supports local deployment of many more resources. In the other frameworks, actual cloud resources are connected to instead of local ones. While this is guaranteed to simulate the cloud environment correctly (as it is the cloud environment), the resources need to be deployed first. If a deployment needs to be done, then this takes away many of the benefits of local testing, such as the speed to deploy the environment. Additionally, if mistakes are made, it could be the user-facing environment that is connected to, which could lead to unwanted/dangerous modifications.

6.3.5 Conclusion

The local testing deployment successfully simulates the cloud in most known cases, and it has been shown that the local deployment can be performed much faster than a cloud deployment. Therefore the local deployment makes testing cloud interaction, and the user's code, faster than by deploying to the cloud. The reason that the Nimbus assembly is slower than the maven-shade-plugin is discussed in Subsection 6.4.2.

6.4 Deployment

6.4.1 Optimised Package Size

Different projects relying on different dependencies were set up to evaluate that using the Nimbus assembler can result in reductions of package size, as detailed in Table 6.9.

	maven shade plugin + serverless	**maven shade plugin + Nimbus**	**Nimbus assembly**		
Project Setup	**package size (KB)**	**package size (KB)**	**average package size (KB)**	**max package size (KB)**	**min package size (KB)**
One function, no dependencies	11	28616	589	589	589
One function, uses SQS	5291	28616	8070	8070	8070
Two functions, one uses SNS & other uses SQS	5621	28616	8087.5	8105	8070
Two functions, one uses Spring + Guava, other uses google phone lib + MySQL + ApacheCommons	8934	34199	5198.5	6450	3947

Table 6.9: Comparison of package sizes of different projects.

Remarks

When comparing the maven shade plugin with serverless to the Nimbus assembly plugin in small plugins, there is little benefit gained. The reason why this is due to the extra dependencies that using the Nimbus framework pulls in, like the Kotlin dependencies. This is obvious when compared with the maven shade plugin with the Nimbus framework, which pulls in all the maven dependencies of Nimbus into the JAR, resulting in massive initial size. When the assembly plugin is compared to the maven shade plugin with Nimbus, it almost seems like a requirement to reduce the package size to a reasonable level. The reason that the Nimbus dependency is so large is that it requires all the underlying AWS libraries, regardless of if the user will actually make use of them. Also included in the dependency are all the testing components, even though these will not be used in the cloud. Another way to solve this, without using the assembly plugin, would be to split the Nimbus dependency into separate parts, for example, a core part and a testing part, and the testing part could be marked as a test dependency only. In Nimbus, the AWS dependencies could be marked as provided elsewhere, so that the user has control of the exact dependencies of their application.

One undesired interaction was noticed when performing the examination. When just a DynamoDB client was being instantiated, it was noticed that the S3 dependencies were being packaged as well, even though they do not need to be. This was inspected further, and it was discovered that one of the classes that the DynamoDB client inherits from attempts to load in a class in the S3 library with reflection. This reflection is surrounded by a try-catch block and makes the assumption that if the class isn't there, then this is not an S3 client, and execution can continue as normal. This goes directly against the assumption that the Nimbus assembler makes, that if a class is attempted to be loaded in, then it is needed. Because of this assumption, that S3 class is loaded in, even though it isn't needed, and then from that much of the remaining S3 library is packaged into the JAR. This results in a larger packaged size than necessary.

WebSocket Chat Deployment Times (seconds)	
Nimbus maven-shade-plugin	**Nimbus Assembly**
234	383

One Update of WebSocket Chat (seconds)	
Nimbus maven-shade-plugin	**Nimbus Assembly**
182	91

Table 6.10: Deployment times on creation and updates.

Conclusion

When using the Nimbus framework, using the assembly plugin drastically reduces the size of the packaged JAR. When compared to the maven shade plugin, the Nimbus assembler is beneficial when functions start pulling in entirely separate dependencies. The two function example can be extrapolated into a project with a large number of dependencies that can focus on separate aspects, for example, some interaction with a database, some with a DynamoDB table, and some focusing on application logic. In this case, each function will have a reduced package size compared to the one large shaded JAR created by the maven shade plugin.

6.4.2 Effect of Updates

Service Interruption

When one JAR is used to deploy all the functions, each function is required to be updated. This creates new instances of the Lambdas to which any new requests will be directed, with each instance being affected by a cold start. An advantage of deploying multiple jars is that unchanged functions will not be affected by this change.

Deployment Times

The bottleneck of deployment with multiple JARs in testing was the network upload speed, used when uploading the functions. When this is the bottleneck, if the total amount to upload with function JARs is less than the maven shade plugin size, a deployment will take less time when using multiple JARs.

Table 6.10 shows the deployment times for the WebSocket chat application mentioned in Subsection 6.3.3, on initial creation and on one function being changed. Initial creation takes longer with the Nimbus assembly due to the total function JAR size being greater than the maven shade plugin JAR size. Upon updates, the Nimbus assembly plugin is faster due to the total JAR size being less than that of the maven-shade-plugin. Additionally, only the one function has to be updated by AWS, while when using the maven-shade plugin, all functions have to be updated by AWS.

Conclusion

For the first deployment of a project with multiple functions, deploying with just one JAR will likely be faster. This is due to there now being more data to upload in total, and the network operation being the bottleneck. However, for subsequent updates where only a small subset of functions of updates are done, packaging into multiple jars can result in a faster deployment.

Future work of the assembler could decide to package in a way that optimises the deployment time instead of the package size.

6.5 Open Source Evaluation

As Nimbus is provided as an open source framework that anyone can request to commit to on GitHub, to encourage people to commit the code design should be easily extensible.

Extending AWS Support

A contributor may want to add support for a new resource, a new type of function, or a new client with permissions. To add a new resource: new annotations must be defined, an intermediate representation class that extends `Resource` written, and a new resource creator written that extends `CloudResourceResouceCreator`. In this resource creator, the required method is implemented so that an instance of the intermediate representation is created and included in the CloudFormation documents. The intermediate representation class will need to implement the `toCloudFormation` method.

Adding a new function type follows the above pattern, except that there is a method that already exists to create a function resource, and the resource creator class will extend `FunctionResourc-eCreator`. Additional permissions may need to be added to the function, or an additional permissions resource created. All this will be done in the new resource creator class. Finally, a class that extends `ServerlessFunctionFileBuilder` needs to be created that will generate the function handler file.

To add a new client: AWS and local versions of the client need to be written, and an interface defined. Once these have been written, a new method needs to be added to the `ClientBuilder` class that will instantiate the correct client depending on if the code is running locally or in the cloud. To add permissions to a function that uses the client, a permissions resource creator is written that extends `UsesResourceHandler`. In this class, the permissions will be added to any function that is annotated with newly defined permissions annotations.

For all of the above, when a new resource creator is added, it needs to be registered in the annotation processor class, by adding the new class to the corresponding resource creator list. Additionally, any new annotations defined need to be registered in the annotation processor class.

Adding Cloud Providers

Adding a new cloud provider would require a significant effort in terms of the deployment aspect. However, it should require little code change and mainly additions. In cases where there is a one to one mapping of the new cloud provider resource to AWS resource, then the intermediate resource can be kept the same, and new code added that converts it to the CloudFormation equivalent. In some cases, new resources will need to be added. In theory, the resource creators could be kept the same, but in some cases, AWS specific resources will have to avoid being created, and new provider specific resources added.

In terms of the deployment plugin, the assembler can be kept the same. However, the code would need to be written to support the rest of the deployment for the new provider.

Documentation

There is clear documentation on how to use the resources on the framework website; however, at the time of writing, no Javadocs support has been added. This would allow contributors to look at

a class and quickly determine what role it plays, as well as help any average user of the framework to understand how to use it.

Conclusion

For the majority of AWS cases, there is a clear, demonstrated way to add new features. For a new cloud provider, there is a significant amount of initial effort to get a 'vertical slice' for the new provider; however, the existing code should not require much change. One current downside is the lack of Javadocs support, which can be added in future. Apart from this, the framework is easily extendable, which can also be seen from the fact that there are public forks [1] of the project.

6.6 Community Response

Nimbus was released in April 2019, with the website and documentation made public, as well as an article written on the Medium platform to advertise it to developers. The article, *"Nimbus Framework - Serverless applications, easier"*[2] has had over one hundred reads. Shortly after its release, an article on Nimbus appeared on the InfoQ news site, which has a very large readership among software developers[3]. Additionally, the project has accumulated 30 "stars" on GitHub.

Dustin Schultz, Lead Software Engineer and InfoQ Editor, in a personal communication, wrote:

"I found your framework quite interesting and quite helpful. I've been doing Java development for about 15 years now and I've run into exactly the problems that your framework is trying to solve. I can vividly remember creating a simple CRUD-based, function-as-a-service / serverless API at one of the companies I worked for and remembering how many moving parts there were to create just a simple serverless-based API. I also remember how painful the iterative development process was, not to mention the testing (or lack thereof). I had used a combination of Terraform and AWS SDKs to create my API but when it was all completed I wasn't really happy with how the whole development process had played out. As a team lead, I thought to myself that some of the junior members of my team would struggle to put all the pieces together, especially the ones that didn't have any devops or cloud experience. What I really liked about your framework is that it abstracted away a lot of these "moving parts". I loved the use of annotations as it had a very Spring-Framework-like feeling to it. I also thought the local deployment and testing was a great idea. With your framework, you can really improve the speed of iteration with local deployment and testing. My only criticism is the name. In terms of SEO, you're going to be competing with a lot of other things named Nimbus, especially a very popular browser extension for capturing webpage screenshots called Nimbus Capture. You've also got the Nimbus Project to compete with on Google results, which is in a similar space (cloud infrastructure). All in all though, great idea and great implementation. I'm excited to see where you take it. In fact, I was quite surprised that there wasn't already a project like yours in development from some of the major players like Spring."

[1] https://github.com/thomasjallerton/nimbus-core/network/members
[2] https://medium.com/@thomasjallerton/nimbus-framework-serverless-applications-easier
[3] https://www.infoq.com/news/2019/04/nimbus-serverless-java-framework/

Chapter 7

Conclusion

The result of this project is the Nimbus framework, explicitly designed to write and deploy Java serverless applications. The main features are detailed below. This chapter summarises what the project has achieved, and provides suggestions for future work.

7.1 Core Achievements

Reduced Configuration

By using an object-oriented model with annotations, Nimbus lets the user configure the application in a much more concise way. This allows the user to keep the code and configuration close, with no more massive configuration files.

Reduced Deployment Time

Many common errors are prevented by default, and thanks to the annotation processor, many errors are caught at runtime. Compared to errors being found minutes into a deployment as was the case before, Nimbus drastically reduces the overall deployment time. In addition, the optimised deployment on updates also helps to reduce the overall deployment time.

Reduced Testing Time

Thanks to the implementation of the local test component that simulates all the supported resources, applications can be thoroughly tested locally without having to wait minutes for a deployment. This means that mistakes can quickly be discovered and fixed, and removes the need for multiple full deployments to be done before the application works.

Optimised JAR Size

The framework includes a deployment plugin which optimises the size of function JARs by analysing their dependencies. This optimised JAR size helps to reduce cold start times and avoid the function size limit in large projects.

No Performance Trade-offs

In the evaluation, it was shown that these benefits do not come with any meaningful sacrifices in performance or functionality. Developers using the framework can then be confident that using the framework comes with very few drawbacks.

7.2 Future Work

In the future, it would be great to see more community support for the framework, with more cloud providers and resources being supported. Additionally, future work could be done to the deployment plugin to optimise package size or deployment time.

There are many quality of life features that could be added, like displaying what resources will be created and deployed before committing to a deployment, and Javadocs support for better documentation.

When developing REST APIs with Nimbus that use a document store or a key-value store it is very easy to forget that you are writing a serverless application and not a more traditional monolith server due to the APIs of the stores. These stores only emulate Java sets and maps, so some research could be done to figure out how to write lists, trees, or any other data structure in a similar fashion.

One issue mentioned with local testing is that there is no way to interact with resources directly, apart from HTTP endpoints and WebSocket endpoints. Future work could be to build a web console, similar to the AWS web console, that lets users interact with resources.

Currently, the functions are only deployed to FaaS offerings. If for some reason in the FaaS becomes much more expensive, or stops being supported, then the users of the framework have no choice but to keep using FaaS or rewrite the code for another service. Nimbus could start providing support for more cloud compute offerings, like containers and more traditional monolith servers. Ideally, this would require no, or very little code change, for the user. For example, a REST API could be containerised and then hosted on elastic container service (ECS) on AWS instead of API gateway. This also gives the user more control, for example, if the cold starts are too detrimental.

Permissions annotations are needed as the annotation processor cannot automatically infer what clients a serverless method uses. Research could be done to see if there is any simple way to add this functionality, ideally without making many assumptions about the environment. If this was done, then the permissions could be removed entirely, protecting the user from many easy mistakes.

7.3 Personal Reflections

While developing the framework, specifically when trying to support new resources, I was constantly reminded why I wanted the framework to exist. The development process for new resources was to create the code that generated the CloudFormation, then try to deploy it, and then fix any errors that occurred. Many of these errors were the ones mentioned earlier, like spelling mistakes or missing some parameters. In some cases, it was due to a misunderstanding of how the resource actually worked, and so the entire generation needed to change to add even more resources. For example, I think it took me around 6 hours to get a relational database to deploy in a way that I could connect to it from my local machine. The configuration never took that long to fix when an error was thrown, but because relational databases take so much longer than any other resources to create, the majority of the time was spent just waiting to see if the deployment had worked.

Now that I've used Nimbus to deploy some applications, it is hard to go back to using the other frameworks, just because of how easy it is to use. I dreaded part of the evaluation as I knew

I would have to go back and use the other frameworks, specifically having to write the massive configuration files. It feels good knowing that Nimbus is a framework that I would definitely use in the future, over existing solutions.

Appendix A

CloudFormation Template Examples

A.1 File Storage Bucket

```
"WebSocketChatServerWebChatNimbusFileBucket": {
  "Type": "AWS::S3::Bucket",
  "Properties": {
    "NotificationConfiguration": {
      "LambdaConfigurations": [
        {
          "Event": "s3:ObjectCreated:*",
          "Function": {
            "Fn::GetAtt": [
              "WebsiteonFileDeletedFunction",
              "Arn"
            ]
          }
        }
      ]
    },
    "AccessControl": "PublicRead",
    "WebsiteConfiguration": {
      "IndexDocument": "webchat.html",
      "ErrorDocument": "error.html"
    },
    "BucketName": "webchatnimbusdev"
  },
  "DependsOn": [
    "WebsiteonFileDeletedFunction",
    "WebsiteonFileDeletedFunctionPermissionApiGateway"
  ]
},
"WebSocketChatServerWebChatNimbusFileBucketPolicy": {
  "Type": "AWS::S3::BucketPolicy",
  "Properties": {
    "Bucket": {
      "Ref": "WebSocketChatServerWebChatNimbusFileBucket"
    },
    "PolicyDocument": {
      "Id": "WebSocketChatServerWebChatNimbusFileBucketPolicy",
      "Version": "2012-10-17",
      "Statement": [
        {
          "Sid": "PublicReadForWebSocketChatServerWebChatNimbusFileBucket",
          "Effect": "Allow",
          "Principal": "*",
          "Action": "s3:GetObject",
          "Resource": {
            "Fn::Join": [
              "",
              [
                "arn:aws:s3:::",
                "webchatnimbusdev/*"
              ]
            ]
          }
        }
      ]
    }
  }
```

```
}
```

Listing A.1: All resources required to create an S3 bucket

A.2 Relational Database

```
"cfexamplesdevVPC": {
  "Type": "AWS::EC2::VPC",
  "Properties": {
    "CidrBlock": "10.0.0.0/16",
    "EnableDnsHostnames": true,
    "EnableDnsSupport": true
  }
},
"PubliclyAccessibleSubnetA": {
 "Type": "AWS::EC2::Subnet",
  "Properties": {
    "CidrBlock": "10.0.1.0/24",
    "MapPublicIpOnLaunch": true,
    "VpcId": {
  "Ref": "cfexamplesdevVPC"
    },
    "AvailabilityZone": {
      "Fn::Join": [
        "",
        [
          {
            "Ref": "AWS::Region"
          },
          "a"
        ]
      ]
    }
  },
  "DependsOn": [
    "cfexamplesdevVPC"
      ]
},
"PubliclyAccessibleSubnetB": {
  "Type": "AWS::EC2::Subnet",
  "Properties": {
    "CidrBlock": "10.0.0.0/24",
    "MapPublicIpOnLaunch": true,
    "VpcId": {
      "Ref": "cfexamplesdevVPC"
        },
    "AvailabilityZone": {
      "Fn::Join": [
        "",
        [
          {
            "Ref": "AWS::Region"
          },
          "b"
        ]
      ]
    }
  },
 "DependsOn": [
 "cfexamplesdevVPC"
 ]
},
"PublicSubnetGroup": {
  "Type": "AWS::RDS::DBSubnetGroup",
  "Properties": {
"DBSubnetGroupDescription": "Publicly available database subnet - created by nimbus",
    "SubnetIds": [
  {
        "Ref": "PubliclyAccessibleSubnetA"
      },
      {
        "Ref": "PubliclyAccessibleSubnetB"
      }
    ]
  },
  "DependsOn": [
    "PubliclyAccessibleSubnetA",
    "PubliclyAccessibleSubnetB"
  ]
},
"PublicDBSecurityGroup": {
  "Type": "AWS::EC2::SecurityGroup",
  "DependsOn": [
    "cfexamplesdevVPC"
```

```
    ],
    "Properties": {
      "SecurityGroupIngress": [
        {
          "CidrIp": "0.0.0.0/0",
          "IpProtocol": "tcp",
          "FromPort": "3306",
          "ToPort": "3306"
        },
        {
          "CidrIpv6": "::/0",
          "IpProtocol": "tcp",
          "FromPort": "3306",
          "ToPort": "3306"
        }
      ],
      "SecurityGroupEgress": [
        {
          "CidrIp": "0.0.0.0/0",
          "IpProtocol": "tcp",
          "FromPort": "3306",
          "ToPort": "3306"
        },
        {
          "CidrIpv6": "::/0",
          "IpProtocol": "tcp",
          "FromPort": "3306",
          "ToPort": "3306"
        }
      ],
      "GroupDescription": "Allows all incoming for databases",
      "VpcId": {
        "Ref": "cfexamplesdevVPC"
      }
    }
  },
  "testdbRdsInstance": {
    "Type": "AWS::RDS::DBInstance",
    "Properties": {
      "AllocatedStorage": 20,
      "DBInstanceIdentifier": "testdbdev",
      "DBInstanceClass": "db.t2.micro",
      "Engine": "mysql",
      "PubliclyAccessible": true,
      "MasterUsername": "master",
      "MasterUserPassword": "287234jsdhf",
      "StorageType": "gp2",
      "VPCSecurityGroups": [
        {
          "Ref": "PublicDBSecurityGroup"
        }
      ],
      "DBSubnetGroupName": {
        "Ref": "PublicSubnetGroup"
      }
    },
    "DependsOn": [
      "PublicDBSecurityGroup",
      "PublicSubnetGroup"
    ]
  },
  "InternetGateway": {
    "Type": "AWS::EC2::InternetGateway",
    "Properties": {}
  },
  "cfexamplesdevVPCInternetGatewayAttachment": {
    "Type": "AWS::EC2::VPCGatewayAttachment",
    "Properties": {
      "InternetGatewayId": {
        "Ref": "InternetGateway"
      },
      "VpcId": {
        "Ref": "cfexamplesdevVPC"
      }
    },
    "DependsOn": [
      "cfexamplesdevVPC",
      "InternetGateway"
    ]
  },
  "RoutingTablecfexamplesdevVPC": {
    "Type": "AWS::EC2::RouteTable",
    "Properties": {
      "VpcId": {
        "Ref": "cfexamplesdevVPC"
      }
    },
```

```
166   "DependsOn": [
167     "cfexamplesdevVPC"
168   ]
169 },
170 "RoutingTablecfexamplesdevVPCToInternetGateway": {
171   "Type": "AWS::EC2::Route",
172   "Properties": {
173     "DestinationCidrBlock": "0.0.0.0/0",
174     "RouteTableId": {
175       "Ref": "RoutingTablecfexamplesdevVPC"
176     },
177     "GatewayId": {
178       "Ref": "InternetGateway"
179     }
180   },
181   "DependsOn": [
182     "RoutingTablecfexamplesdevVPC",
183     "InternetGateway"
184   ]
185 },
186 "AssociationRoutingTablecfexamplesdevVPCPubliclyAccessibleSubnetA": {
187   "Type": "AWS::EC2::SubnetRouteTableAssociation",
188   "Properties": {
189     "RouteTableId": {
190       "Ref": "RoutingTablecfexamplesdevVPC"
191     },
192     "SubnetId": {
193       "Ref": "PubliclyAccessibleSubnetA"
194     }
195   },
196   "DependsOn": [
197     "RoutingTablecfexamplesdevVPC",
198     "PubliclyAccessibleSubnetA"
199   ]
200 },
201 "AssociationRoutingTablecfexamplesdevVPCPubliclyAccessibleSubnetB": {
202   "Type": "AWS::EC2::SubnetRouteTableAssociation",
203   "Properties": {
204     "RouteTableId": {
205       "Ref": "RoutingTablecfexamplesdevVPC"
206     },
207     "SubnetId": {
208       "Ref": "PubliclyAccessibleSubnetB"
209     }
210   },
211   "DependsOn": [
212     "RoutingTablecfexamplesdevVPC",
213     "PubliclyAccessibleSubnetB"
214   ]
215 }
```

Listing A.2: All resources required to create a relational database

A.3 Key-Value and Document Stores

```
1  "UserDetaildev": {
2    "Type": "AWS::DynamoDB::Table",
3    "Properties": {
4      "TableName": "UserDetaildev",
5      "AttributeDefinitions": [
6        {
7          "AttributeName": "username",
8          "AttributeType": "S"
9        }
10     ],
11     "KeySchema": [
12       {
13         "AttributeName": "username",
14         "KeyType": "HASH"
15       }
16     ],
17     "ProvisionedThroughput": {
18       "ReadCapacityUnits": 2,
19       "WriteCapacityUnits": 2
20     }
21   }
22 }
```

Listing A.3: Resource required to create a DynamoDB table

A.4 Functions

```
"WebchatApiregisterFunction": {
  "Type": "AWS::Lambda::Function",
  "DependsOn": [
    "IamRoleWebchregisterExecution"
  ],
  "Properties": {
    "Code": {
      "S3Bucket": {
        "Ref": "NimbusDeploymentBucket"
      },
      "S3Key": "nimbus/WebSocketChatServer/${HttpServerlessFunctionWebchatApiregister.jar}"
    },
    "FunctionName": "WebSocketChatServer-dev-WebchatApi-register",
    "Handler": "com.nimbusframework.examples.webchat.HttpServerlessFunctionWebchatApiregister::
     nimbusHandle",
    "MemorySize": 1024,
    "Role": {
      "Fn::GetAtt": [
        "IamRoleWebchregisterExecution",
        "Arn"
      ]
    },
    "Runtime": "java8",
    "Timeout": 10,
    "Environment": {
      "Variables": {
        "NIMBUS_STAGE": "dev"
      }
    }
  }
},
"IamRoleWebchregisterExecution": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": [
              "lambda.amazonaws.com"
            ]
          },
          "Action": [
            "sts:AssumeRole"
          ]
        }
      ]
    },
    "Policies": [
      {
        "PolicyName": "WebSocketChatServer-Webchregister-policy",
        "PolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {
              "Effect": "Allow",
              "Action": [
                "logs:CreateLogStream"
              ],
          "Resource": [
                {
                  "Fn::Sub": "arn:${AWS::Partition}:logs:${AWS::Region}:${AWS::AccountId}:log-group:/
     aws/lambda/WebSocketChatServer-dev-WebchatApi-register:*"
                }
              ]
            },
            {
              "Effect": "Allow",
              "Action": [
                "logs:PutLogEvents"
              ],
              "Resource": [
                {
                  "Fn::Sub": "arn:${AWS::Partition}:logs:${AWS::Region}:${AWS::AccountId}:log-group:/
     aws/lambda/WebSocketChatServer-dev-WebchatApi-register:*:*"
                }
              ]
            },
            {
              "Effect": "Allow",
              "Action": [
                "dynamodb:*"
              ],
              "Resource": [
                {
```

```
                  "Fn::GetAtt": [
                    "UserDetaildev",
                    "Arn"
                  ]
                }
              ]
            }
          ]
        }
      }
    ],
    "Path": "/",
    "RoleName": "WebSocketChatSer-dev-Webchregister"
  }
},
"WebchatApiregisterLogGroup": {
  "Type": "AWS::Logs::LogGroup",
  "Properties": {
    "LogGroupName": "/aws/lambda/WebSocketChatServer-dev-WebchatApi-register"
  }
}
```

Listing A.4: All resources required to create a Lambda function

A.5 Function Permission

This resource allows another resource to invoke the corresponding Lambda function.

```
"WebchatApigetConnectionDetailFunctionPermissionApiGateway": {
  "Type": "AWS::Lambda::Permission",
  "Properties": {
    "FunctionName": {
      "Fn::GetAtt": [
        "WebchatApigetConnectionDetailFunction",
        "Arn"
      ]
    },
    "Action": "lambda:InvokeFunction",
    "Principal": {
      "Fn::Join": [
        "",
        [
          "apigateway.",
          {
            "Ref": "AWS::URLSuffix"
      }
    ]
      ]
    },
    "SourceArn": {
      "Fn::Join": [
        "",
        [
          "arn:",
          {
            "Ref": "AWS::Partition"
          },
          ":execute-api:",
          {
            "Ref": "AWS::Region"
          },
          ":",
          {
            "Ref": "AWS::AccountId"
          },
          ":",
          {
            "Ref": "ApiGatewayRestApi"
          },
          "/*/*"
        ]
      ]
    }
  }
}
```

Listing A.5: Resource required to allow a resource to invoke a Lambda function

A.6 Function Event Source Mapping

This resource allows Lambda to poll a resource to then invoke a function.

```
"QueuehelloWorldNimbusSQSQueuemessage": {
  "Type": "AWS::Lambda::EventSourceMapping",
  "DependsOn": "IamRoleQueuehelloWorldExecution",
  "Properties": {
    "BatchSize": 1,
    "EventSourceArn": {
      "Fn::GetAtt": [
        "NimbusSQSQueuemessage",
        "Arn"
      ]
    },
    "FunctionName": {
      "Fn::GetAtt": [
        "QueueApihelloWorldFunction",
        "Arn"
      ]
    },
    "Enabled": "True"
  }
}
```

Listing A.6: Function Event Source Mapping Resource

A.7 REST API

```
"ApiGatewayRestApi": {
  "Type": "AWS::ApiGateway::RestApi",
  "Properties": {
    "Name": "cfexamples-dev-HTTP",
    "EndpointConfiguration": {
      "Types": [
    "EDGE"
      ]
    }
  }
},
"cfexamplesApiGatewayDeployment1559214782701": {
  "Type": "AWS::ApiGateway::Deployment",
  "Properties": {
    "Description": "Nimbus Deployment for project cfexamples",
    "RestApiId": {
      "Ref": "ApiGatewayRestApi"
    },
    "StageName": "dev"
  },
  "DependsOn": [
    "ApiGatewayMethodexamplepathGET"
  ]
},
"ApiGatewayResourceexample": {
  "Type": "AWS::ApiGateway::Resource",
  "Properties": {
    "ParentId": {
      "Fn::GetAtt": [
        "ApiGatewayRestApi",
        "RootResourceId"
      ]
    },
    "PathPart": "example",
    "RestApiId": {
      "Ref": "ApiGatewayRestApi"
    }
  }
},
"ApiGatewayResourceexamplepath": {
  "Type": "AWS::ApiGateway::Resource",
  "Properties": {
    "ParentId": {
      "Ref": "ApiGatewayResourceexample"
    },
    "PathPart": "path",
    "RestApiId": {
      "Ref": "ApiGatewayRestApi"
    }
  }
},
```

```
"ApiGatewayMethodexamplepathGET": {
  "Type": "AWS::ApiGateway::Method",
  "Properties": {
    "HttpMethod": "GET",
    "RequestParameters": {},
    "ResourceId": {
      "Ref": "ApiGatewayResourceexamplepath"
    },
    "RestApiId": {
      "Ref": "ApiGatewayRestApi"
    },
    "ApiKeyRequired": false,
    "AuthorizationType": "NONE",
    "Integration": {
      "IntegrationHttpMethod": "POST",
      "Type": "AWS_PROXY",
      "Uri": {
        "Fn::Join": [
          "",
          [
            "arn:",
            {
              "Ref": "AWS::Partition"
            },
            ":apigateway:",
            {
              "Ref": "AWS::Region"
            },
            ":lambda:path/2015-03-31/functions/",
            {
            "Fn::GetAtt": [
                "RESTAPIhelloWorldFunction",
                "Arn"
                ]
            },
            "/invocations"
          ]
        ]
      }
    },
    "MethodResponses": []
  }
},
```

Listing A.7: All resources required to create a simple REST API with one method that has a path "example/path"

A.8 WebSocket API

```
"WebsocketApiDeployment1559215340992": {
  "Type": "AWS::ApiGatewayV2::Deployment",
  "Properties": {
    "ApiId": {
      "Ref": "WebsocketApi"
    }
  },
  "DependsOn": [
    "messageRoute"
  ]
},
"WebsocketApiStage": {
  "Type": "AWS::ApiGatewayV2::Stage",
  "Properties": {
    "ApiId": {
      "Ref": "WebsocketApi"
    },
    "DeploymentId": {
      "Ref": "WebsocketApiDeployment1559215340992"
    },
    "StageName": "dev"
  }
},
"RESTAhelloWorldWebSocketIntegrationmessage": {
  "Type": "AWS::ApiGatewayV2::Integration",
  "Properties": {
    "ApiId": {
      "Ref": "WebsocketApi"
    },
    "IntegrationType": "AWS_PROXY",
    "IntegrationUri": {
  "Fn::Join": [
        "",
        [
          "arn:",
```

```
          {
            "Ref": "AWS::Partition"
          },
          ":apigateway:",
          {
        "Ref": "AWS::Region"
          },
          ":lambda:path/2015-03-31/functions/",
          {
            "Fn::GetAtt": [
              "RESTAPIhelloWorldFunction",
              "Arn"
            ]
          },
          "/invocations"
        ]
      ]
    }
  }
},
"messageRoute": {
  "Type": "AWS::ApiGatewayV2::Route",
  "Properties": {
    "ApiId": {
      "Ref": "WebsocketApi"
    },
    "RouteKey": "message",
    "AuthorizationType": "NONE",
    "Target": {
      "Fn::Join": [
        "/",
        [
          "integrations",
          {
            "Ref": "RESTAhelloWorldWebSocketIntegrationmessage"
          }
        ]
      ]
    }
  }
},
```

Listing A.8: All resources required to create a simple WebSocket API with one topic "message"

A.9 Queue

```
"NimbusSQSQueuemessage": {
  "Type": "AWS::SQS::Queue",
  "Properties": {
    "VisibilityTimeout": 60,
    "QueueName": "messagedev"
  }
}
```

Listing A.9: Resource required to deploy a queue

A.10 Notification

```
"SNSTopicmessage": {
  "Type": "AWS::SNS::Topic",
  "Properties": {
    "TopicName": "messagedev",
    "DisplayName": "",
    "Subscription": [
      {
        "Endpoint": {
          "Fn::GetAtt": [
            "QueueApihelloWorldFunction",
            "Arn"
          ]
        },
        "Protocol": "lambda"
      }
    ]
  }
}
```

Listing A.10: Resource required to deploy a notification topic

Bibliography

[1] "2018 Serverless Community Survey: Huge Growth in Serverless Usage." [Online]. Available: https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/

[2] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A Mixed-method Empirical Study of Function-as-a-Service Software Development in Industrial Practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, March 1, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218302735

[3] C. Smith, "AWS Lambda in Production: State of Serverless Report 2017," -11-21T14:25:41-08:00 2017. [Online]. Available: https://blog.newrelic.com/product-news/aws-lambda-state-of-serverless/

[4] G. Adzic and R. Chatley, "Serverless Computing: Economic and Architectural Impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 884–889.

[5] "Understanding Container Reuse in AWS Lambda," -12-31T00:48:29-08:00 2014. [Online]. Available: https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[6] "AWS Lambda – Serverless Compute - Amazon Web Services." [Online]. Available: https://aws.amazon.com/lambda/

[7] "Azure Functions – Serverless Architecture | Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-gb/services/functions/

[8] "Cloud Functions - Event-driven Serverless Computing | Cloud Functions." [Online]. Available: https://cloud.google.com/functions/

[9] "IBM Cloud Functions." [Online]. Available: https://console.bluemix.net/openwhisk/

[10] "Serverless - The Serverless Application Framework powered by AWS Lambda, API Gateway, and more." [Online]. Available: https://serverless.com/

[11] "Spring Cloud." [Online]. Available: http://spring.io/projects/spring-cloud

[12] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and A. Slominski, *Serverless Computing: Current Trends and Open Problems*, ser. Research Advances in Cloud Computing. Springer, 2017, pp. 1–20.

[13] H. Puripunpinyo and M. H. Samadzadeh, "Effect of Optimizing Java Deployment Artifacts on AWS Lambda." IEEE, May 2017, pp. 438–443. [Online]. Available: https://ieeexplore.ieee.org/document/8116416

[14] "AWS Cloudformation - Infrastructure as Code & aws Resource Provisioning." [Online]. Available: https://aws.amazon.com/cloudformation/

[15] "Snafu." [Online]. Available: https://github.com/serviceprototypinglab/snafu

[16] "AWS Serverless Application Model - Amazon Web Services." [Online]. Available: https://aws.amazon.com/serverless/sam/

[17] K. Kritikos and P. Skrzypek, “A Review of Serverless Frameworks.” IEEE, Dec 2018, pp. 161–168. [Online]. Available: https://ieeexplore.ieee.org/document/8605774

[18] “AWS Lambda Triggers.” [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html

[19] “Triggers and Bindings in Azure Functions.” [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings

[20] “Lesson: Annotations (The Java™ Tutorials; Learning the Java Language).” [Online]. Available: https://docs.oracle.com/javase/tutorial/java/annotations/

[21] H. Rocha and M. T. Valente, “How Annotations are Used in Java: An Empirical Study,” 2011.

[22] “Zappa: Serverless python web services.” [Online]. Available: https://github.com/Miserlou/Zappa

[23] “Eclipse Abstract Syntax Tree.” [Online]. Available: https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/

[24] C. Guindon, “Eclipse Desktop & Web IDEs | The Eclipse Foundation.” [Online]. Available: https://www.eclipse.org/ide/

[25] D. Landman, A. Serebrenik, and J. Vinju, “Challenges for Static Analysis of Java Reflection,” ser. ICSE ’17. IEEE Press, May 20, 2017, pp. 507–518. [Online]. Available: http://dl.acm.org/citation.cfm?id=3097429

[26] “Best Practices for Querying and Scanning Data - Amazon Dynamodb.” [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-scan.html

[27] “H2 Database Engine.” [Online]. Available: https://www.h2database.com/html/main.html

[28] “Jetty - Servlet Engine and HTTP Server.” [Online]. Available: https://www.eclipse.org/jetty/