Nimbus: Improving the Developer Experience for Serverless Applications

Robert Chatley
Imperial College London
London, United Kingdom
rbc@imperial.ac.uk

Thomas Allerton
Starling Bank
London, United Kingdom
thomasjallerton@gmail.com

ABSTRACT

We present Nimbus, a framework for writing and deploying Java applications on a Function-as-a-Service ("serverless") platform. Nimbus aims to soothe four main pain points experienced by developers working on serverless applications: that testing can be difficult, that deployment can be a slow and painful process, that it is challenging to avoid vendor lock-in, and that long cold start times can introduce unwelcome latency to function invocations.

Nimbus provides a number of features that aim to overcome these challenges when working with serverless applications. It uses an annotation-based configuration to avoid having to work with large configuration files. It aims to allow the code written to be cloud-agnostic. It provides an environment for local testing where the complete application can be run locally before deployment. Lastly, Nimbus provides mechanisms for optimising the contents and size of the artifacts that are deployed to the cloud, which helps to reduce both deployment times and cold start times.

Video: https://www.youtube.com/watch?v=0nYchh8jdY4

CCS CONCEPTS

• Computer systems organization → Cloud computing; **• Software and its engineering → Development frameworks and environments**.

KEYWORDS

serverless, developer tools, developer experience

ACM Reference Format:
Robert Chatley and Thomas Allerton. 2020. Nimbus: Improving the Developer Experience for Serverless Applications. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382135

1 INTRODUCTION

Function-as-a-Service technologies such as AWS Lambda are often gathered together under the term "serverless" and since 2014, each of the major cloud providers has developed a serverless offering. Rather than deploying and running monolithic services, or dedicated virtual machines, users are able to deploy individual functions, and pay only for the time that their code is actually executing. Serverless has become a popular technology for developers to use in building and deploying applications, and can significantly change how client/server applications are designed, developed and operated [2].

A serverless application architecture often leads to systems comprised of many separate functions (in our experience often tens or hundreds of functions). Much of the application infrastructure is handled by the cloud provider, with developers only required to write handlers to be triggered by events such as HTTP requests, file uploads, or messages appearing on a queue. While breaking down applications into very small components definitely has its benefits, a lot of the structure of the overall application can be hidden, and it becomes difficult to run the system as a whole without deploying it to the cloud. Rather than just being a hosting platform, the cloud infrastructure becomes an integral part of the application.

Development tools specifically targetting serverless are still in their infancy[3]. Multiple surveys have been performed asking developers working on serverless applications what they consider to be the most significant challenges [4, 5]. Some of the most common answers were: that testing can be difficult, that deployment can be a slow and painful process, that it is challenging to avoid vendor lock-in, and that long cold start times can introduce unwelcome latency to function invocations.

The authors have also experienced these problems first hand, notably during an industrial project that involved developing a Java REST API deployed on AWS Lambda. While working on this application, the most time-consuming activity was not getting the business logic right, but trying to successfully deploy code and have it interact correctly with the cloud environment. Often the application was deployed, a configuration error was revealed, fixed, and then another deployment performed. This process was repeated until everything was configured correctly. As each deployment typically took several minutes, this wasted time quickly accumulated. The errors that caused the issues ranged from typos in the large and complex deployment configuration files, to incorrect permissions set for the functions so that they could not interact with other cloud resources such as datastores.

Another pain point was the limit set by the cloud provider for the maximum file size that can be uploaded when deploying a function. It is not unreasonable for a provider to set such a limit, but as the application grew, with all the functions and their dependencies compiled together into one unit, the file size naturally grew larger and larger. Eventually, the limit was reached and the entire codebase had to be restructured to split it into a number of distinct modules (each built as its own Maven project and deployed separately) before further development could proceed.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382135

2 NIMBUS

Nimbus is a framework for writing serverless applications in Java and deploying them to AWS Lambda. The philosophy of Nimbus is that developers should be able to think about and work with their application as an integrated whole, even though the component parts will be deployed separately as a set of independent functions. Nimbus provides a number of features that aim to overcome the identified challenges when working with serverless applications. It uses an annotation-based configuration to avoid having to work with large configuration files. It aims to allow the code written to be cloud-agnostic, so that it could easily be deployed to a different cloud provider in future. It provides an environment for local testing where the complete application can be run locally before deployment. Lastly, Nimbus provides mechanisms for optimising the contents and size of the artifacts that are deployed to the cloud.

Nimbus is implemented in Kotlin, and can be included in any Java project as a Maven dependency. It provides Maven integration to allow deployment tasks to be handled within a standard Java development environment. Java was targeted due to its popularity in enterprise development. We believe that the concepts used in Nimbus could translate to other languages, if equivalent tooling was developed, but it is not a goal of Nimbus to be language agnostic.

2.1 Annotation-based Configuration

Deploying a serverless application typically requires a configuration file that describes the cloud resources required. For example with AWS Lambda, this is typically done in the form of a CloudFormation template [1]. As the configuration is separate from the associated code, it can be difficult to determine from the code how a function will be deployed. When this configuration is written manually it is prone to errors, and many of these are not found until a deployment is attempted. An error in a CloudFormation template will cause an AWS stack update to be aborted, with any changes up to that point automatically rolled back. If the configuration 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 several minutes, the time taken to fix multiple errors can significantly increase the time taken to deploy an application successfully.

With Nimbus, instead of one large configuration file, functions and resources are defined by placing annotations on classes and methods. This drastically reduces the amount of configuration required to be written by the user. Additionally, use of annotations allows for type checking, meaning that a large class of errors can now be caught at compile time rather than at deployment time.

To deploy a particular Java method as a Lambda function, and configure an appropriate event on which it should be triggered, we simply apply the appropriate Nimbus annotation to the method concerned, and then at deployment time, Nimbus will generate the relevant CloudFormation template and use this to deploy our function and any associated resources.

Nimbus can create functions with many different types of event triggers. HTTP functions are triggered when an HTTP request is made to a particular endpoint. To create an HTTP function, we annotate a method with `@HttpServerlessFunction` and define a unique URL path and HTTP method.

The method can have at most two parameters. If header, path or query parameters need to be processed, one of these should be of type `HttpEvent`. The second method parameter available is one with a domain specific type which will be automatically deserialised from the JSON body of the request.

The example in Listing 1 shows configuring an HTTP API endpoint to receive POST requests to create Customer records in an associated data store. The code also uses the `@UsesDocumentStore` annotation which will cause Nimbus to set up an appropriate data store in the cloud (for example using DynamoDB in AWS) and set appropriate permissions to allow the function to write to the relevant store when it is invoked. Nimbus generates the CloudFormation template based on type safe annotations, which is much more reliable, and much more concise, than writing the configuration by hand.

```java
import com.nimbusframework.annotations.UsesDocumentStore;
import com.nimbusframework.annotations.HttpMethod;
import com.nimbusframework.annotations.HttpServerlessFunction;
import com.nimbusframework.clients.ClientBuilder;
import com.nimbusframework.clients.DocumentStoreClient;

public class CustomerApi {

  private DocumentStoreClient<Customer> customerStore =
      ClientBuilder.getDocumentStoreClient(Customer.class);

  @HttpServerlessFunction(method=HttpMethod.POST,path="customers")
  @UsesDocumentStore(dataModel=Customer.class)
  public boolean addCustomer(Customer newCustomer) {
    try {
      customerStore.put(newCustomer);
      return true;
    } catch (Exception e) {
      return false;
    }
  }
}
```

Listing 1: Configuring an API endpoint using annotations.

2.2 Cloud-agnostic Application Code

The example in Listing 1 shows that there is nothing AWS-specific in the application code. Nimbus provides a layer of adapters to connect to and interact with the various AWS-specific datastores (for example `DocumentStoreClient`). The annotation processor that runs at deployment time will generate AWS-specific CloudFormation templates, but this could easily be swapped out for a generator targetting another platform. At the time of writing, only AWS is supported, but we hope to look at generating configurations for Azure or IBM's Cloud Functions in the near future.

Another benefit of insulating the application code from directly referencing the cloud provider's services and APIs is that Nimbus can transparently switch out the cloud-based data stores for local alternatives when we want to run the application locally.

2.3 Local Testing

With existing platforms, the primary ways of testing functions are either by testing them in isolation using unit tests, or by deploying the system to the cloud in a development environment and testing it there. This is slow and also incurs additional hosting costs. Nimbus can deploy our full application locally to allow for local

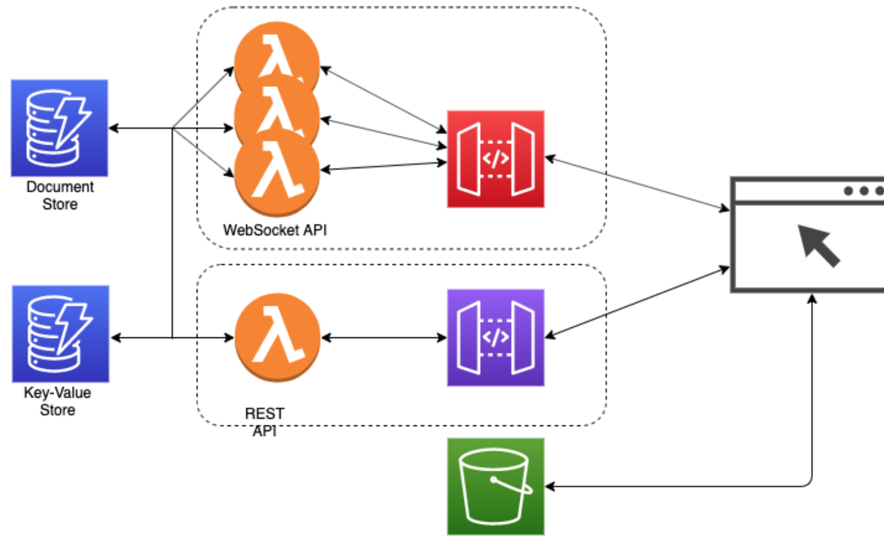

Figure 1: Example architecture for a serverless chat application.

integration testing and then deploy the same code to a production environment in the cloud without any further changes. With existing tools and frameworks (for example AWS SAM[1]) it is possible to run a function locally, but it is not possible to create a mirror of the complete environment, including datastores, queues, static web pages, websockets etc. Nimbus supports all of these locally.

Local deployment allows for a development experience much closer to that of traditional web-development, where we run the application locally, load it up in a browser, and explore directly to see how our new feature looks and feels. Working locally allows for much faster feedback than deploying to the cloud after every change. The local environment also supports automated tests which can be run on a developer's workstation, or as part of a build pipeline.

2.4 Optimising JAR Size

Nimbus includes a deployment plugin that can optimise the size of deployed JARs by analysing the code's dependencies and including in the JAR only the classes that are actually required. This optimised JAR size helps to reduce cold start times and to avoid the size limit on files that can be deployed.

Cold starts are a common source of latency in serverless systems. To handle a function invocation it may be necessary to spin up a new container, start the language runtime, and load the function, before it can be executed. The JVM is already slow to start compared with some other language runtimes, but the problem is exacerbated if we have to load large JARs. Building smaller JARs improves the cold start times [6]. In order to deploy a Java function to AWS Lambda the function must be compiled and packaged into a *shaded* JAR containing all the function's dependencies as well as our own code. The Maven Shade plugin[2] does this by combining all the JARs that our function depends on together into one big JAR, even if we only use a few classes. Nimbus analyses the code at a finer-grained level so that it can build a separate JAR for each serverless function, containing only the subset of classes that it actually needs in order to run. For example if we depend on Spring Framework, but only because we use the StringUtils class, Nimbus will include the Spring StringUtils class in our deployment JAR, but will omit any other parts of the Spring Framework. This analysis also works for transitive dependencies.

Additionally, Nimbus can analyse our source code so that even if we define all of our functions in one project (which is much more convenient for development and local testing), when we deploy, Nimbus can detect which functions have changed, and will build and deploy only those particular functions. This again helps a lot with the turn-around time for deployment, especially in big projects.

2.4.1 Evaluating effectiveness of JAR optimisation. To evaluate the effectiveness of Nimbus's JAR optimisation, several different functions with different types of dependencies were created and packaged both using the Maven Shade plugin, which creates one all encompassing JAR, and using Nimbus, which creates separate JARs for each function. The results are shown in Table 1.

Functions Packaged	Maven Shade Plugin	Nimbus Assembler
1 function, no deps	28,616 KB	589 KB
1 function, using SQS	28,616 KB	8070 KB
2 functions, one uses SQS, the other SNS	28,616 KB	8087.5 KB (per JAR)
2 functions, using small subsets of separate libs	34,199 KB	5198.5 KB (per JAR)

Table 1: Comparing JAR sizes produced by different tools

The Maven Shade Plugin always builds a large JAR when packaging a Nimbus project, as it needs to include the Nimbus framework. It pulls in the whole library, even if we only need the annotations and clients. The Nimbus assembler cuts this down as it only pulls in

[1] https://docs.aws.amazon.com/serverless-application-model
[2] https://maven.apache.org/plugins/maven-shade-plugin/

the parts of the Nimbus library that are required post-deployment, stripping out any unused clients for AWS services, the testing infrastructure, etc.

Thus, when using the Nimbus framework, using the Nimbus assembler drastically reduces the size of the packaged JAR. When compared to the Maven Shade Plugin, the Nimbus assembler is also beneficial when different functions pull in separate dependencies. The examples in Table 1 where we package two functions show a significant reduction in the size of each JAR. Remember that with Nimbus a change to one function means that only one JAR has to be built and deployed. Unchanged functions do not need to be redeployed, which is another defence against cold starts. This could be particularly effective in a project with a large number of different functions with varying dependencies – for example, if some interact with a relational database, some with a DynamoDB table, and some focus on application logic. In this case, when packaged with Nimbus Assembler each function will have a reduced package size compared to the one large shaded JAR created by the Maven Shade Plugin.

3 DEVELOPING WITH NIMBUS

To show how a more complex system can be built and deployed with Nimbus we have developed a simple chat application. The chat app has simple a web frontend (HTML and JavaScript) allowing users to log in and send messages to other online users. Sending messages is handled by a WebSocket-based backend API. Registering new users is handled by a REST API, invoked when an HTML form on the frontend is submitted.

Figure 1 shows an architecture diagram for the chat application. When deployed to the cloud, the web frontend is hosted in and served from a file storage bucket (S3). The WebSocket API comprises three functions: one to handle connections, one for disconnections, and one for sending messages. Each of these is deployed as a separate lambda function. The REST API has just one function that registers new users, called via HTTP POST requests. Due to the stateless nature of serverless functions, data stores are needed to persist data between requests. A key-value store is used to map WebSocket connection IDs to usernames, and a document store to store data on users, including their current active WebSocket. 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 active users.

All of this can be configured using Nimbus annotations in the Java code, and the full application can be run and tested locally, before deploying it into the cloud. The full code for the chat application can be found in the Nimbus examples GitHub repository[3].

4 COMMUNITY RESPONSE

Nimbus was released in April 2019, with the website, source code and documentation made public[4], as well as an article written on the Medium platform to advertise it to developers. Subsequently the project accumulated over 30 stars on GitHub. Shortly after its release, an article on Nimbus appeared on the InfoQ news site [7], which has a very large readership among software developers.

Dustin Schultz, Lead Software Engineer and InfoQ Editor, in a personal communication, wrote: "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 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 ... I was quite surprised that there wasn't already a project like [Nimbus] in development from some of the major players like Spring."

5 CONCLUSIONS AND FUTURE WORK

Nimbus provides a smoother developer experience when developing applications targetting a serverless platform. When developing applications with Nimbus that provide REST APIs and 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 monolithic server. In general this is a good thing, as it allows the developer to work quickly, and to concentrate on the business logic of their application, rather than the details of how it will be deployed. However there is a potential danger that the local infrastructure provides only a relatively simple simulation of the cloud infrastructure and storage services. If developers want to fine tune specific details then they might need more control than Nimbus currently offers, especially given its aims of being cloud agnostic.

The next obvious step for development is to allow additional cloud platforms beyond AWS to be targeted by the same application, by generating different configuration and interacting with the appropriate APIs to trigger deployment.

REFERENCES

[1] 2019. AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning. https://aws.amazon.com/cloudformation/

[2] Gojko Adzic and Robert Chatley. 2017. Serverless Computing: Economic and Architectural Impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 884–889.

[3] Kyriakos Kritikos and Pawel Skrzypek. Dec 2018. A Review of Serverless Frameworks. IEEE, 161–168. https://doi.org/10.1109/UCC-Companion.2018.00051

[4] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2019. A Mixed-method Empirical Study of Function-as-a-Service Software Development in Industrial Practice. *Journal of Systems and Software* 149 (March 1, 2019), 340–359. https://doi.org/10.1016/j.jss.2018.12.013

[5] Andrea Passwater. 2018. 2018 Serverless Community Survey: Huge Growth in Serverless Usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/

[6] Hussachai Puripunpinyo and M. H. Samadzadeh. May 2017. Effect of Optimizing Java Deployment Artifacts on AWS Lambda. IEEE, 438–443. https://doi.org/10.1109/INFCOMW.2017.8116416

[7] Dustin Schultz. 2019. Nimbus: New Framework for Building Java Serverless Applications. https://www.infoq.com/news/2019/04/nimbus-serverless-java-framework/

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

[4] https://www.nimbusframework.com/