



# Compute Dashboard

11.04.2020

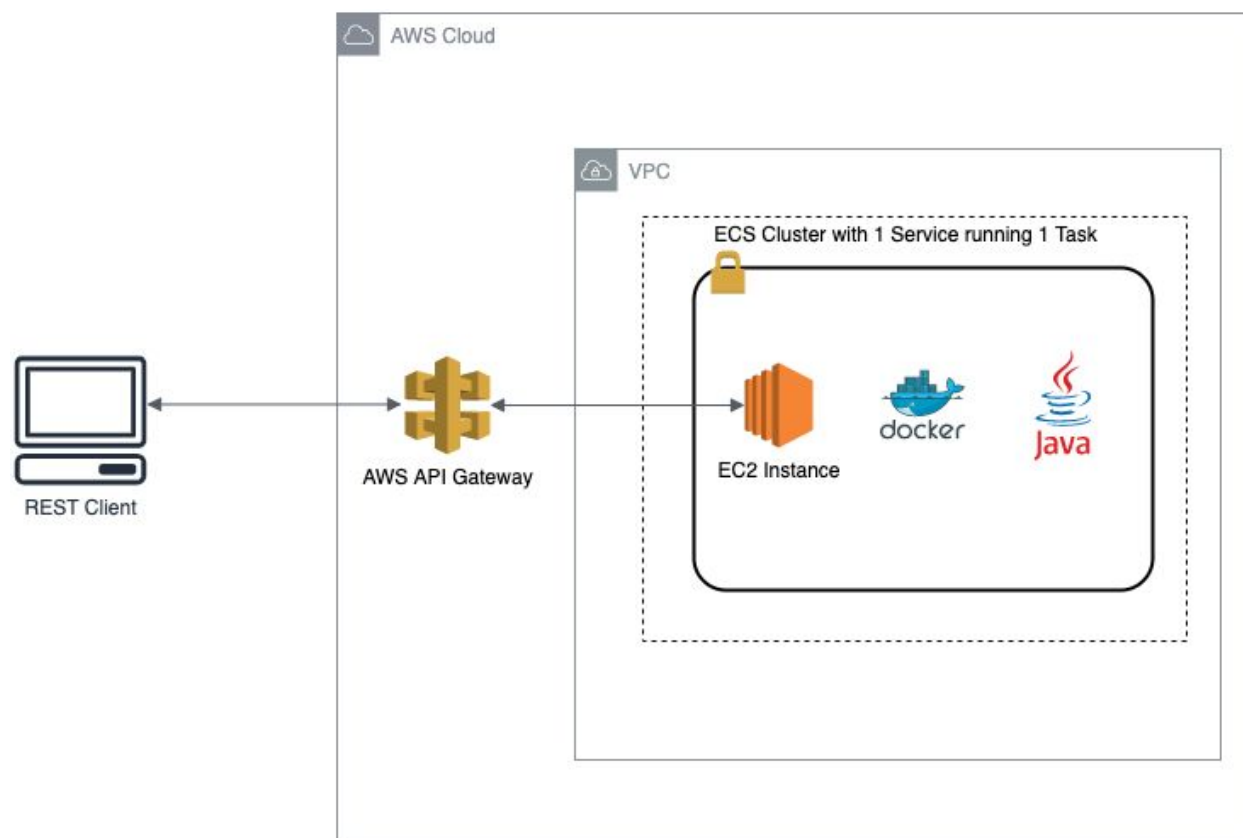
---

Timothy John Troup  
Somerled Solutions Ltd  
16 Angle Park Terrace  
Edinburgh EH11 2JX

## Overview

This document illustrates the overall system architecture of the Compute Dashboard deployed on AWS. The CI/CD pipeline that was created to build and deploy the application on AWS is also discussed. In addition, the technology stack that was selected to implement the Compute Dashboard API is also described together with some justification. In order to run and deploy this application on a local machine please refer to the README.md document provided.

## System Architecture



## ECS Cluster

The Compute Dashboard API is deployed as a Docker container running on an EC2 instance in an ECS cluster that is currently configured with 1 service running this 1 task. This was set up via the AWS web console but could be automated via Cloudformation:

<https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-task-definition.html>

The EC2ReadOnlyAccessPolicy was added to the `ecsInstanceRole` to allow the AWS SDK to make the API calls that it makes when we use it to describe the instances.

## API Gateway

API Gateway is placed in front of the ECS cluster. An OpenAPI document (discussed further in section OpenAPI 3.0 Document) was created and imported. A configuration was applied such that an endpoint is exposed that acts as a pass-through for the REST API running on the ECS cluster but requires an API key to be provided. This API key was generated using the AWS web console. A REST API integration was chosen over HTTP API as it supports API keys:

<https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html>

To productionise this deployment I would use Cloudformation or Terraform to capture this infrastructure as code. In the meantime the following documentation was followed to provision via the web console:

### Deployment:

<https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-deploy-api.html>

### IAM role to allow API Gateway service to write to cloudwatch logs:

<https://aws.amazon.com/premiumsupport/knowledge-center/api-gateway-cloudwatch-logs>

### Setting up a usage plan and API key:

<https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-create-usage-plans.html>

I would also consider placing an Application Load Balancer (ALB) between the API Gateway and put the EC2 instances in an auto-scaling group.

## Technology Stack and Implementation Details

### OpenAPI 3.0 Document

First an OpenAPI 3.0 Document was created to define the API that will be exposed to satisfy the requirements. To ensure that the code that is created meets this specification the `springdoc-openapi-ui` maven plugin is used at build time to generate an OpenAPI 3.0 definition from the code. This can then be cross checked with the OpenAPI 3.0 Document that was initially created.

In addition this maven dependency deploys a UI which enables endpoints to be manually invoked and the OpenAPI document to be downloaded:

<https://www.baeldung.com/spring-rest-openapi-documentation>

## Spring Boot

Spring Boot provides a plethora of tools and abstractions to facilitate the rapid development of REST APIs in Java. As such this was chosen as the main Java REST framework for this project.

<https://spring.io/projects/spring-boot>

## AWS SDK for Java

At the heart of the implementation lies the AWS SDK for Java. This provides an Amazon EC2 Client that enables EC2 instances to be managed programmatically including the retrieval of metadata for running instances:

<https://docs.aws.amazon.com/sdk-for-java/v2/developer-guide/examples-ec2-instances.html>

All calls being made to the Compute Dashboard API endpoint delegate the request through to the EC2Client provided.

AWS SDK for Java maven dependencies must be set up:

<https://docs.aws.amazon.com/sdk-for-java/v2/developer-guide/setup-project-maven.html>


The EC2Client can be configured such that it only retrieves a list of EC2 instances in a specified region:

<https://docs.aws.amazon.com/sdk-for-java/v2/developer-guide/java-dg-region-selection.html>

The data returned provides everything we need to satisfy the requirement of displaying the following attributes:

- name,
- instanceId (e.g. a-123456abcd),
- instanceType (e.g. t2.medium),
- state (e.g. running),
- availabilityZone (e.g. "us-east-1b"),
- public IP (e.g. "54.210.167.204"),
- private IPs (e.g. "10.20.30.40").

Inspection of the model returned shows that all required attributes are returned:



<https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/ec2/model/Instance.html>

Note that for AZ you need to look at the placement object and the name is stored under tags.

The results from the EC2Client are paged; you can get further results by passing the value returned from the result object's getNextToken method to your original request object's setNextToken method, then using the same request object in your next call to describeInstances.

## Model Mapping

We need a way to map the model object returned by the EC2Client to the model objects we expose through our endpoint. For this we chose to use Dozer as it reduces a lot of boilerplate code and has an integration with Spring Boot which provides a bean we can easily access via dependency injection which in turn enhances testability:

<https://dozermapper.github.io/gitbook/documentation/springBootIntegration.html>

<https://dozermapper.github.io/user-guide.pdf>

## Logging

Logback is provided out of the box with Spring Boot. Other logging frameworks can be configured but Logback provides what we require in an expedient way:

<https://docs.spring.io/spring-boot/docs/2.2.6.RELEASE/reference/html/spring-boot-features.html#boot-features-logging>

The EC2 instances are not configured to write the application logs out to CloudWatch. A further enhancement would be to use the awslog driver but for the time being SSH can be used to access the EC2 instance and query the logs which can be found under:

```
/var/lib/docker/overlay2/838b2c0bde203cec4236c735447af72f0020904e40db9e6aca0e8b67a6563e04/diff/logs/spring-boot-logger.log
```

## Testing

### Unit Test

Unit tests were written using Junit 5. Mockito was leveraged to mock out any dependencies and ensure we are testing the specific unit of code under test. AssertJ is also used as it provides a fluent API which makes tests easier to write and maintain.

## Mutation Testing

Traditional test coverage (i.e line, statement, branch, etc.) measures only which code is executed by your tests. It does not check that your tests are actually able to detect faults in the executed code. It is therefore only able to identify code that is definitely not tested.

The most extreme examples of the problem are tests with no assertions. Fortunately these are uncommon in most code bases. Much more common is code that is only partially tested by its suite. A suite that only partially tests code can still execute all its branches (examples).

As it is actually able to detect whether each statement is meaningfully tested, mutation testing is the gold standard against which all other types of coverage are measured.

<https://pitest.org/>

Mutation testing can be run via pitest by running the following:

```
mvn org.pitest:pitest-maven:mutationCoverage
```

A test report can be found under: target/pit-reports

## Manual Testing

As discussed previously, when the application is running locally the springdoc-openapi-ui maven plugin provides a UI which allows visual exploration of the API and to invoke endpoints manually with user defined payloads:

<http://localhost:8080/swagger-ui.html>

## Integration Testing

Spring Boot provides a testing framework which allows the application to be spun up but with dependencies stubbed out such that the application can be tested in isolation

<https://spring.io/guides/gs/testing-web/>

Wiremock was also configured but there was insufficient time available to write any tests using it. The general idea was to use wiremock to mock the AWS services that the AWS Java SDK EC2Client calls at runtime::

[https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API\\_DescribeInstances.html](https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_DescribeInstances.html)

## CICD Pipeline

The application code is under version control in an AWS CodeCommit repository. AWS CodeBuild has been configured to build the code and push the resulting docker image to Amazon ECR (see buildspec.yml). AWS CodePipeline has been configured to automatically pull the code from the codecommit and invoke CodeBuild each time code is pushed to the master branch.

<https://docs.aws.amazon.com/codebuild/latest/userguide/sample-docker.html>

Currently deployment to the ECS cluster requires a manual restart of the task but this could be automated via CodeDeploy.

## Idempotency

It is our understanding that the AWS Java SDK and the services it accesses are idempotent and by extension so is the REST API that this application exposes. Should the AWS services ever change to no longer be idempotent we can introduce a default sort order and thusly ensure our REST API remains idempotent.

The implementation provided always retrieves all results as the AWS SDK and services it calls does not support sorting. As a result the application must retrieve all results in order to be able to provide an accurate sorted collection.