Microservice architecture URLs / articles

Cloud native, lift and shift are the regular discussion points in digital transformation landscape to disrupt the value chain.

This article talks about moving microservices to serverless using strangler pattern and Sprint Cloud Function. Reference to links and a complete coverage.

<https://dzone.com/articles/from-java-microservices-to-lambda-functions-a-jour>

# From Spring Boot Microservices to Lambda Functions

### Get your microservices where they need to be.

[**[](https://dzone.com/users/1065393/foyst.html)**](https://dzone.com/users/1065393/foyst.html)**by**

[**Ben Foster**](https://dzone.com/users/1065393/foyst.html)

**·**

**Nov. 06, 21 ·**[**Java Zone**](https://dzone.com/java-jdk-development-tutorials-tools-news)**· Tutorial**

**Like**[**(8)**](https://dzone.com/articles/from-java-microservices-to-lambda-functions-a-jour)

**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/articles/from-java-microservices-to-lambda-functions-a-jour)

**20.92K Views**

Join the DZone community and get the full member experience.

[**JOIN FOR FREE**](https://dzone.com/static/registration.html)

 You may be one of many organisations (or an engineer in one) that operates Java microservices in the cloud with a desire to move towards a serverless architecture, but are unable to justify the steep migration path (e.g. decomposing your services into functions, rewriting in a more suitable language etc.) from those microservices to the likes of AWS Lambda.

But fear not! Because with the help of [spring-cloud-function](https://spring.io/projects/spring-cloud-function) you can repurpose your existing microservices into serverless functions in a gradual and controlled manner, with minimal effort or interruption of service.

In this article, I'll explain how you can achieve this utilising the [Strangler Fig Pattern](https://martinfowler.com/bliki/StranglerFigApplication.html) to quickly prove out the approach and see if it fits your needs. I've used AWS CDK, ECS, ALB and Lambda to demonstrate how you can move traffic from a Java microservice to multiple Lambda functions.

I've built a sample codebase and accompanying CDK code. I've used git branches to show how you go about transitioning over to Lamdba, which I'll be talking about through this post:

<https://github.com/foyst/spring-petclinic-rest-serverless>

(The above repo is freely available for people to base prototypes on, to quickly experiment with this approach in their organisations.)

It's based upon the Spring Petclinic REST Application example. I wanted to use something that was well known and understood, and representative of a real application scenario that demonstrates the potential of spring-cloud-function.

Note that the API and documentation for spring-cloud-function has changed over time, and I personally found it difficult to understand what the recommended approach is. So I hope this article also captures some of those pain points and provides others with a guide on how to implement it.

Along the journey I battled against the above and other gotchas, that can distract and take considerable time to overcome. To not distract from the main narrative I've moved these to the back of this post.

## Setting up AWS CDK

If you're following along with the GitHub repo above, make sure you're on the [master](https://github.com/foyst/spring-petclinic-rest-serverless) branch for the first part of this blog.

I'll not digress into how to set up CDK as there are plenty of resources out there. These are the ones which I found particularly useful:

<https://cdkworkshop.com/15-prerequisites/500-toolkit.html>

<https://cdkworkshop.com/20-typescript/20-create-project/100-cdk-init.html>

<https://docs.aws.amazon.com/cdk/api/latest/docs/aws-construct-library.html>

<https://gitter.im/awslabs/aws-cdk>

Using the above references, I created my CDK solution with cdk init sample-app --language typescript as a starting point.

I've put together a simple AWS architecture using CDK to demonstrate how to do this. I've kept to the sensible defaults CDK prescribes, its default configuration creates a VPC with 2 public and 2 private subnets. This diagram shows what's deployed by my GitHub repo:

Diagram

Description automatically generated

I have an RDS MySQL instance, which is used by both the ECS and Lambda applications. ECS is running the Petclinic Java microservice and Lambda is running its serverless counterparts. An Application Load Balancer is used to balance requests between the Fargate container and Lambda functions.

I used Secrets Manager to handle the generation of the RDS password, as this also allows you to pass the secret securely through to the ECS Container. For info on how to set up Secrets Manager secrets with RDS in CDK so you can do credentials rotation, I used <https://dev.to/michaelfecher/i-tell-you-a-secret-provide-database-credentials-to-an-ecs-fargate-task-in-aws-cdk-5f4>.

Initially, I tried to deploy a VPC without NAT gateways, to keep resources isolated and unnecessary costs down. But this is where I encountered my first Gotcha, due to changes in the way Fargate networking works as of version 1.4.0.

## Stage One - All requests to Java Container

So in the first instance, all of the requests are routed by default to the Petclinic Fargate Service:

Table

Description automatically generated

Later on, I'll use weighted target groups and path-based routing to demonstrate how you can use the Stangler Pattern to gradually migrate from microservices to Lambda functions in a controlled fashion.

To deploy the initial infrastructure with the RDS and ECS Services running, cd into the cdk folder and run the following command:

cdk deploy --require-approval=never --all

This will take some time (~30 mins), mainly due to the RDS Instance spinning up. Go put the kettle on...

## Transforming a Spring Boot Microservice to Serverless

Now for the meaty part. In my [GitHub repo](https://github.com/foyst/spring-petclinic-rest-serverless) you can switch over to the [1-spring-cloud-function](https://github.com/foyst/spring-petclinic-rest-serverless/tree/1-spring-cloud-function) branch which includes additional CDK and Java config for writing Spring Cloud Functions.

There's a number of good articles out there that demonstrate how to create serverless functions using Java and Spring, such as Baeldung with <https://www.baeldung.com/spring-cloud-function>. Where this blog hopefully differs is by showing you a worked example on how to decompose an existing Java microservice written in Spring Boot into Lambda functions.

Importantly, make sure you use the latest versions and documentation - a lot has changed, and there are so many search results pointing to outdated articles and docs that it can make it confusing to understand which are the latest. **The latest version of spring-cloud-function at the time of writing this is 3.2.0-M1**, and the documentation for this can be found here:

<https://docs.spring.io/spring-cloud-function/docs/3.2.0-M1/reference/html/spring-cloud-function.html#_introduction>

This too: <https://cloud.spring.io/spring-cloud-function/reference/html/>

And example functions can be found in <https://github.com/spring-cloud/spring-cloud-function/tree/main/spring-cloud-function-samples>

**So, can you do dual-track development of the existing application alongside splitting out into Lambda functions?**Yes, by pushing all application logic out of the REST and Lambda classes (delegating to a service layer or similar if you don't already have one) and having a separate Maven profile for lambda development. By [comparing the master and 1-spring-cloud-function branches](https://github.com/foyst/spring-petclinic-rest-serverless/compare/master...foyst:1-spring-cloud-function#diff-9c5fb3d1b7e3b0f54bc5c4182965c4fe1f9023d449017cece3005d3f90e8e4d8) you can see the additional changes made to the pom.xml, which includes this new "lambda" profile:

Text

Description automatically generated

The Maven lambda profile is aimed at developing the lambdas. It has the spring-cloud-function specific dependencies and plugins connected to it, which ensures none of those bleed into the existing core Java application. When you do development and want to build the original microservice jar, you can use existing Maven build commands as before. Whenever you want to build the Lambda jar just add the lambda profile, e.g. ./mvnw package -P lambda

For this example I've created a couple of functions, to demonstrate both how to define multiple functions within one jar, and how to isolate them in separate AWS Lambda functions. I've called them getAllOwners and getOwnerById which can be found in src/main/java/org/springframework/samples/petclinic/lambda/LambdaConfig.java:

@Bean

public Supplier<Collection<Owner>> getAllOwners() {

return () -> {

LOG.info("Lambda Request for all Owners");

return this.clinicService.findAllOwners();

        };

}

@Bean

public Function<Integer, Owner> getOwnerById() {

return (ownerId) -> {

LOG.info("Lambda Request for Owner with id: " + ownerId);

final Owner owner = this.clinicService.findOwnerById(ownerId);

return owner;

};

}

This is where I experienced my second gotcha. Spring Cloud Functions aspires to provide a cloud-agnostic interface that gives you all the necessary flexibility you may need, but sometimes you want control of the platform internals. In the above for example, you can't handle returning a 404 when a resource is not found because you don't have access to the payload that's returned to ALB/API Gateway.

Thankfully, after posting a [Stack Overflow question](https://stackoverflow.com/questions/68465552/how-to-handle-a-404-response-with-spring-cloud-function-and-aws) and a [GitHub issue](https://github.com/spring-cloud/spring-cloud-function/issues/716), promptly followed by a swift solution and release (many thanks to Oleg Zhurakousky for the speedy turnaround!) you can now bypass the cloud-agnostic abstractions by returning an APIGatewayProxyResponseEvent, which gets returned to the ALB/API GW unmodified:

@Bean

public Function<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> getOwnerById() {

return (requestEvent) -> {

LOG.info("Lambda Request for Owner");

final Matcher matcher = ownerByIdPattern.matcher(requestEvent.getPath());

if (matcher.matches()) {

final Integer ownerId = Integer.valueOf(matcher.group(1));

final Owner owner = this.clinicService.findOwnerById(ownerId);

if (owner != null) {

return buildOwnerMessage(owner);

} else return ownerNotFound();

}

else return ownerNotFound();

};

}

private APIGatewayProxyResponseEvent buildOwnerMessage(Owner owner) {

final Map<String, String> headers = buildDefaultHeaders();

try {

APIGatewayProxyResponseEvent responseEvent = new APIGatewayProxyResponseEvent()

.withIsBase64Encoded(false)

.withBody(objectMapper.writeValueAsString(owner))

.withHeaders(headers)

.withStatusCode(200);

return responseEvent;

} catch (JsonProcessingException e) {

throw new RuntimeException(e);

}

}

private Map<String, String> buildDefaultHeaders() {

final Map<String, String> headers = new HashMap<>();

headers.put("Content-Type", "application/json");

return headers;

}

private APIGatewayProxyResponseEvent ownerNotFound() {

final Map<String, String> headers = buildDefaultHeaders();

APIGatewayProxyResponseEvent responseEvent = new APIGatewayProxyResponseEvent()

.withIsBase64Encoded(false)

.withHeaders(headers)

.withBody("")

.withStatusCode(404);

return responseEvent;

}

Using the APIGatewayProxyRequestEvent as a function input type does give you access to the full request, which you'll probably need when extracting resource paths from a request, accessing specific HTTP headers, or needing fine-grained control on handling request payloads.

For local testing of the functions, there's a few ways you can do it. Firstly you can use the spring-cloud-starter-function-web dependency, that allows you to test the lambda interfaces by calling a http endpoint with the same name as the lambda @Bean method name. For example you can curl localhost:8080/getOwnerById/3 to invoke the getOwnerById function.

Secondly, if you want to debug the full integration path that AWS Lambda hooks into, you can invoke it the same way Lambda does by creating a new instance of the FunctionInvoker class, and passing it the args you'd call it with in AWS. I've left an example of how to do this in src/test/java/org/springframework/samples/petclinic/lambda/LambdaConfigTests.java, which is how the function variants are tested within the [spring-cloud-function library itself](https://github.com/spring-cloud/spring-cloud-function/blob/main/spring-cloud-function-adapters/spring-cloud-function-adapter-aws/src/test/java/org/springframework/cloud/function/adapter/aws/FunctionInvokerTests.java).

When you've developed your Lambda functions, tested them, and are ready to build a jar to serve in AWS Lambda, you can run the following command:

./mvnw clean package -P lambda

You'll see in the next stage that this is run as part of the CDK deployment.

In the target folder, you'll see alongside the existing jar that's used in our Docker microservice container that there's a new jar with an -aws suffix. **What's the difference between this jar and the original jar? Why do I need a separate variant?** Because AWS lambda doesn't support uber-jars, where jars are nested inside each other. To work in Lambda you have to generate a "shaded" jar, where all of the dependency classes are flattened into a single level within the jar archive. Additionally, by using the spring-boot-thin-layout plugin, you can reduce the size of the jar by removing unnecessary dependencies not required in Lambda, which can bring a small cold start performance improvement (the bigger the jar the longer it takes to load into the Lambda runtime):

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.boot.experimental</groupId>

<artifactId>spring-boot-thin-layout</artifactId>

<version>1.0.10.RELEASE</version>

</dependency>

</dependencies>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<configuration>

<createDependencyReducedPom>false</createDependencyReducedPom>

<shadedArtifactAttached>true</shadedArtifactAttached>

<shadedClassifierName>aws</shadedClassifierName>

</configuration>

</plugin>

## Stage Two - Balancing requests between Lambda and ECS

Once you're at the point where you have Lambda functions in your Java app ready to handle requests, we can configure the ALB to route requests between two target groups - one targeted at the incumbent Petclinic container, and the other at our new functions.

At this point, switch over to the [1-spring-cloud-function](https://github.com/foyst/spring-petclinic-rest-serverless/tree/1-spring-cloud-function) branch.

In here you'll see an additional lambda-stack.ts that contains the AWS Lambda configuration, and additional changes in lb-assoc-stack.ts, which creates a target group per Lambda function and uses weighted rules to balance traffic between Lambda and ECS:

Diagram

Description automatically generated

In this scenario I'm using the [ALB integration with Lambda](https://docs.aws.amazon.com/lambda/latest/dg/services-alb.html), to demonstrate that the approach is compatible with both ALB and API GW, and that both methods use the same approach from a code implementation perspective.

In [lambda-stack.ts](https://github.com/foyst/spring-petclinic-rest-serverless/blob/1-spring-cloud-function/cdk/lib/lambda-stack.ts), everything is included to build the Lambda functions. In CDK you can incorporate the building of your code into the deployment of your resources, so you can ensure you're working with the latest version of all your functions and it can all be managed within the CDK ecosystem.

I followed the these two articles to set up a Java Maven application build which delegates the building of the jar files to a Docker container.

<https://aws.amazon.com/blogs/devops/building-apps-with-aws-cdk/>

<https://aws.amazon.com/blogs/opensource/packaging-and-deploying-aws-lambda-functions-written-in-java-with-aws-cloud-development-kit/>

The Docker image you use and the commands you run to build your app are configurable so it's very flexible. AWS provides some Docker images, which ensures that artefacts that are built are compatible with the AWS Lambda runtimes provided.

Text

Description automatically generated

From the screenshot above you can see that the folder specified by the Code.fromAsset command is mounted at /asset-input, and AWS expects to extract a single archived file from the /asset-output/ folder. What you do in between is up to you. In the code above I trigger a Maven package, using my lambda profile I declared earlier (skipTests on your project at your own risk, it's purely for demonstration purposes here!).

This is where I encountered the third gotcha - when you see CDK TypeScript compilation issues, double-check your CDK versions are aligned between CDK modules.

Now on the [1-spring-cloud-function](https://github.com/foyst/spring-petclinic-rest-serverless/tree/1-spring-cloud-function) branch, rerun the CDK deploy command:

cdk deploy --require-approval=never --all

Rerunning this command will deploy the new lambda functions, and you should see the ALB listener rules change from this:

Graphical user interface

Description automatically generated

To this:

Graphical user interface, application

Description automatically generated

Another gotcha to be aware of with Lambda - at the time of writing it doesn't natively integrate with Secrets Manager, which means your secrets are set statically as environment variables, and are visible through the Lambda console. Ouch.

So at this point, we have an ALB configured to balance requests to 2 owners endpoints between the Petclinic container and new Lambda functions. Let's test this with a GET request for Owner information:

Graphical user interface, application

Description automatically generated

In doing this we're presented with a 502 Service Unavailable error. Not ideal but digging into the Lambda CloudWatch logs we can see the first challenge of deploying this lambda:



Further up the call stack we see this issue is affecting the creation of a rootRestController, which is a REST Controller within the petclinic application



The cause behind this with the Petclinic app is that there's a RootRestController bean that configures the REST capabilities in Spring, which requires Servlet-related beans that aren't available when you startup using the FunctionInvoker entry point.

To avoid this issue, we can conditionally omit classes relating to the REST endpoints from being packaged in the jar within the lambda Maven profile:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<excludes>

<exclude>org/springframework/samples/petclinic/rest/\*\*</exclude>

<exclude>org/springframework/samples/petclinic/security/\*\*</exclude>

</excludes>

<testExcludes>

<exclude>org/springframework/samples/petclinic/rest/\*\*</exclude>

<exclude>org/springframework/samples/petclinic/security/\*\*</exclude>

</testExcludes>

</configuration>

</plugin>

I also needed to exclude classes from both the rest and security packages, as the security packages tried configuring REST security which relied on components no longer being initialised by the RootRestController behind the scenes.

This brings me on to my fifth gotcha - the Petclinic app uses custom serialisers that weren't being picked up. I scratched my head with this one as I wasn't able to override the ObjectMapper that was being autoconfigured by spring-cloud-function. However, the following changes fixed the automatic resolution of these serialisers:

* Upgrade spring-cloud-function-dependencies to 3.2.0-M1 (Bugs in previous versions prevented this from working correctly)
* Remove spring-cloud-function-compiler dependency (no longer a thing)
* No longer need to explicitly create a handler class for lambda (i.e. class that extends SpringBootRequestHandler), just use the org.springframework.cloud.function.adapter.aws.FunctionInvoker as the handler

Table

Description automatically generated

* If you have more than one function (which you will do as you begin to migrate more actions/endpoints to serverless), use the SPRING\_CLOUD\_FUNCTION\_DEFINITION environment variable to specify the bean that contains the specific function you want to invoke.

Graphical user interface, text, application, email

Description automatically generated

Redeploying with all of the above changes resulted in working Lambda functions. At this point we're able to send requests and they're picked up by either ECS or Lambda with the same result. Nice!

## Stage Three - Strangulating requests from ECS

Diagram

Description automatically generated

At this point, you can start to gradually phase out the use of your long-running Java microservices by porting functionality across to Lambdas. This approach allows you to build confidence in the Lambda operations by gradually weighting endpoints in favour of your Lambda function target groups.

You'll have probably noticed by this point that Lambda variants of Spring Boot are very slow to startup, which brings me to my sixth gotcha. This may be a dealbreaker for some situations but I'd encourage you to explore the points in my conclusion below before deciding on whether to adopt (or partially adopt)

When porting subsequent features or endpoints to Lambda functions, I'd suggest routing a small percentage of your traffic to the Lambda function as a canary test. So long as the error rate is within known tolerances you can gradually route more traffic to the function until the Lambda is serving 100% of requests. At this point, you can deregister the Fargate target group from that particular ALB path condition and repeat the process for other endpoints you want to migrate.

## Conclusion and Next Steps

This blog article aims to give you a guided walkthrough of taking an existing Spring Boot Java microservice, and by applying the Strangler Fig Pattern transition your workloads into Lambda functions in a gradual and controlled fashion. I hope you find this useful, and all feedback is greatly welcomed!

There's further considerations required to make this production-ready, such as performance tuning the Java Lambdas to reduce the cold start time, and removing DB connection pools that are superfluous in Lambdas and will cause additional load in your database(s). Here's some suggestions for these in the meantime, but I'm aiming to follow this up with an in-depth post to cover these in more detail:

* Analysing the impact of cold-starts on user experience
  + When your Java Lambda initially spins up, it takes a similar amount of time to start up as its microservice counterpart. Given the lower memory and CPU typically allocated to Lambdas, this can result in a cold boot taking anything up to 60 seconds, depending on how bulky your application is.
  + However, considering your particular load profiles, so long as there's a regular stream of requests that keep your Lambdas warm, it may be that cold-starts are rarely experienced and may be within acceptable tolerances
* Tweaking the resource allocations of your Lambdas
  + Allocating more memory (and CPU) to your functions can significantly improve the cold start-up time of Spring (see the graph below), but with a cost tradeoff. Each subsequent request becomes more expensive to serve but no faster, just to compensate for a slow cold start-up. If you have a function that's infrequently used (e.g. an internal bulk admin operation) this may be fine - for a function that's used repeatedly throughout the day, the cost can quickly be prohibitive.

Chart, scatter chart

Description automatically generated

* AWS Lambda Provisioned Concurrency
  + The pricing of this can be prohibitive, but depending on the amount of concurrency required and when it's required (e.g. 2 concurrent lambdas in business hours for a back-office API) this may be suitable
    - Continue to assess this compared with just running ECS containers instead, and weigh it up against the other benefits of Lambda (e.g. less infrastructure to manage & secure) to ensure it's good value
* Using Function declarations instead of @Bean definitions (which can help to improve cold start time)
* Replacing DB Connection pooling (i.e. Hikari, Tomcat etc.) with a simple connection (that closes with the function invocation)
  + And combining this with AWS RDS Proxy to manage connection pooling at an infrastructure level
* Disabling Hibernate schema validation (move schema divergence checking out of your lambdas)
* Experimenting with Lambda resource allocation, to find the right balance between cold start time and cost
  + See [AWS Lambda Power Tuning](https://github.com/alexcasalboni/aws-lambda-power-tuning) - an open source library which provides an automated way to profile and analyse various resource configurations of your lambda

## Gotchas!

### ****First Gotcha**** - Fargate 1.4.0 Network Changes

As of Fargate 1.4.0, Fargate communications to other AWS Services such as ECR, S3 and Secrets Manager use the same network interface that hooks into your VPC. Because of this you have to ensure Fargate has routes available to AWS Services, otherwise you'll find that ECS is unable to pull container images or acquire secrets. You can either give the containers a public IP, route traffic to a NAT running in a public subnet, or create private VPC endpoints to each of the AWS services you require: <https://stackoverflow.com/questions/61265108/aws-ecs-fargate-resourceinitializationerror-unable-to-pull-secrets-or-registry>

### Second Gotcha - Spring Cloud Function Signature Challenges

**Update**: following Oleg's quick turnaround and release of spring-cloud-function 3.2.0-M1 a lot of what's below is moot, but I've kept it here for reference.

There have been developments to abstract the AWS specifics away from function definitions, and have the translation happen behind the scenes by the function adapters (AWS, GCP, Azure, etc.) into a more generic Map payload, rather than to specific AWS types (APIGatewayProxyResponseEvent, for example).

I was confused reading the Spring Cloud Function documentation and didn't find it clear how to make the transition from specific handlers to a generic one (FunctionInvoker). In fact, if you try to follow one of the many guides online (including the latest Spring docs) and use Function<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> the adapter ends up wrapping it in another APIGatewayProxyResponseEvent which is malformed. [AWSLambdaUtils](https://github.com/spring-cloud/spring-cloud-function/blob/b14c86a3d12b8ce8a7ed73b4a1c2f1e93500490f/spring-cloud-function-adapters/spring-cloud-function-adapter-aws/src/main/java/org/springframework/cloud/function/adapter/aws/AWSLambdaUtils.java" \t "_blank) tries to be helpful here but with the confusing documentation and behaviour, it just got in my way.

Feels like the approach is to abstract away from cloud implementation details in the spring-cloud-function core, and push all of that into the adapters. Problem with that is the adapters then have to map from the generic Message interface into an AWS response (APIGatewayProxyResponseEvent), which is great as it abstracts the cloud-platform implementation detail from you, but if you want that level of control there's no way to override this

The way I got clarity on the recommended approach was to **ignore all the docs and examples, and go with the**[**unit tests in the spring-cloud-function-adapter-aws**](https://github.com/spring-cloud/spring-cloud-function/blob/c2d0590ec32de34a006d732dbf2624aaff8d5e44/spring-cloud-function-adapters/spring-cloud-function-adapter-aws/src/test/java/org/springframework/cloud/function/adapter/aws/FunctionInvokerTests.java)**repo**. These demo the latest compatible ways of declaring functions.

I experimented with a few styles of function signatures to see which works...

Doesn't work:

@Bean

public Supplier<Message<APIGatewayProxyResponseEvent>> getAllOwners() {

Doesn't work (didn't correctly set the Content-Type header in the HTTP response)

@Bean

public Function<APIGatewayProxyRequestEvent, Collection<Owner>> getAllOwners()

You have to use the Message construct if you want control over the Content-Type header, otherwise the Message construct provides a contentType header which is incorrect. This then doesn't get picked up by the AWS adapter (resulting in a malformed response with a Content-Type of "application/octet-stream" and another "contentType" header of "application/json", which doesn't get picked up).

Works:

@Bean

public Supplier<Message<Collection<Owner>>> getAllOwners() {

final Map<String, Object> headers = new HashMap<>();

headers.put("Content-Type", "application/json");

...

return new GenericMessage<>(allOwners, headers);

Also works:

@Bean

public Function<APIGatewayProxyRequestEvent, Message<Collection<Owner>>> getAllOwners() {

final Map<String, Object> headers = new HashMap<>();

headers.put("Content-Type", "application/json");

...

return new GenericMessage<>(allOwners, headers);

There are some limitations with using the Message construct - you're not allowed to use null payloads when using GenericMessage, which makes it difficult to handle 404 situations.

Closest I could get was returning a Message<String> response, serialising the Owner object myself into a string using the ObjectMapper before returning it wrapped in a Message. That way when I needed to handle a 404 I just returned an empty string. Not pretty (or consistent with the ECS service) though:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

All of the above, however, is no longer an issue - as explained in the main article thread, you can now return an APIGatewayProxyResponseEvent and have full control over the payload that's returned to API GW/ALB.

### Third Gotcha - CDK Library Module Version Mismatches

NPM dependencies are fun... between working on the core infrastructure code and adding the Lambda functions, CDK had jumped from 1.109.0 to 1.114.0. When I introduced a new component of CDK, it installed the latest version, which then gave me confusing type incompatibility errors between LambdaTarget and IApplicationLoadBalancerTarget. Aligning all my CDK dependencies in package.json (i.e. "^1.109.0") and running an npm update brought everything back in sync.

### Fourth Gotcha - AWS Lambda Lacking Integration With Secrets Manager

There's a gotcha with Lambda and secrets currently - it can't be done natively. There is currently no way to natively pass secrets into a lambda like you can with ECS (see this [CDK issue](https://github.com/aws/aws-cdk/issues/12095)). You have to use [dynamic references in CloudFormation](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/dynamic-references.html), which are injected into your lambda configuration at deploy time. This, however, is far from ideal, as it exposes the secret values within the ECS console:

Graphical user interface, text, application

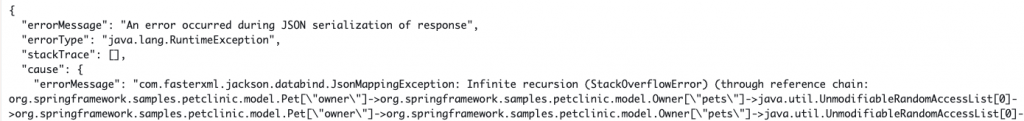
Description automatically generated

One way you can mitigate this is to make the lambda responsible for obtaining its own secrets, using the IAM role to restrict what secrets it's allowed to acquire. You can do this by hooking into Spring's startup lifecycle, by adding an event handler that can acquire the secrets, and set them in the application environment context before the database initialisation occurs.

### Fifth Gotcha - Custom Serialisers Not Loaded in Function

Those curious people will spot that if you explore the shaded lambda jar that's built, although the \*Controller classes have been omitted the (De)serialisers are still there. That's because the compiler plugin has detected dependencies of them in other classes (as opposed to the Controllers which are runtime dependencies and so wouldn't get picked up). This is fine...

Next challenge... the REST endpoints are using these custom (De)serialisers to massage the payloads and avoid a recursion loop between the entities in the object graph:



At this point, I realised maybe Baeldung's post is a little out of date (even though I've bumped up to the latest versions of spring-cloud-function dependencies). So let's see if I can find a more recent post and port my code changes to that.

But what all of this lacks is an example of how to decompose an existing microservice into one or more serverless functions.

By following the Spring documentation and upgrading to the latest version I was able to resolve the issue with the custom Serialisers not being utilised OOTB.

### Sixth Gotcha - Cold Startups Are... Slow

Spring Boot can be notoriously slow at starting up, depending on the size of your applications and the amount of autoconfiguring it has to do. When running in AWS Lambda it still has to go through the same bootstrapping process, but given the typically lower resources allocated to functions (MBs as opposed to GBs) this can draw out the startup of a lambda to upwards of a minute.

This doesn't affect every request - once a Lambda has warmed up its instance becomes cached and stays in a warm (but paused state) until the next request comes in, or AWS Lambda cleans up infrequently used functions. This means subsequent function requests typically respond as quickly as their microservice counterparts. But any requests that don't hit a warm function, or any concurrent requests that trigger a new Lambda function will incur this cold start penalty.

This may be acceptable for some workloads, for example, requests that are triggered in a fire and forget fashion, infrequent (but time-consuming) operations, or endpoints that have a steady stream of requests with little or no concurrent requests. I would advise you as part of prototyping this approach to look at the recommendations in the Conclusion section and, using the Fig Strangler Pattern approach documented above, try it out on a subset of requests to understand how it'd affect your workloads.

<https://www.apiscene.io/lifecycle/rest-grpc-api-communication-in-microservices>/

Another one about REST and gRPC where to use and where not to.

<https://www.infoq.com/news/2021/09/practical-api-design-netflix>

<https://grpc.io/docs/what-is-grpc/core-concepts/>

**Subject:**Distributed transaction patterns for microservices compared | Red Hat Developer

<https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#the_modular_monolith>

I got a very good article on the subject we were discussing in the morning. Please have a look.

<https://dzone.com/articles/memory-wasted-by-spring-boot-application>

Please check this one and point me few thoughts which we should be watching and be careful about programming. I will also try to check the tools sets …

<https://dzone.com/articles/is-todays-microservice-more-bloated-than-yesterday>

My favourite

<https://eng.uber.com/microservice-architecture/>

Saga pattern

<https://blog.mia-platform.eu/en/saga-pattern-how-to-manage-distributed-transactions-with-microservices>

Design for failure

<https://levelup.gitconnected.com/design-for-failure-distributed-transaction-in-microservices-f026b25ba847>

"7 Microservices Best Practices for Developers - DZone Microservices" <https://dzone.com/articles/7-microservices-best-practices-for-developers>

"Is Today’s Microservice More Bloated than Yesterday’s Monolith? - DZone Java" <https://dzone.com/articles/is-todays-microservice-more-bloated-than-yesterday>

Nano service era

<https://www-nextplatform-com.cdn.ampproject.org/v/s/www.nextplatform.com/2021/09/13/forget-microservices-a-nic-cpu-co-design-for-the-nanoservices-era/amp/?amp_gsa=1&amp_js_v=a6&usqp=mq331AQIKAGwASCAAgM%3D#amp_tf=From%20%251%24s&aoh=16316879629317&csi=0&referrer=https%3A%2F%2Fwww.google.com&ampshare=https%3A%2F%2Fwww.nextplatform.com%2F2021%2F09%2F13%2Fforget-microservices-a-nic-cpu-co-design-for-the-nanoservices-era%2F>

Service Mesh

<https://www.infoq.com/articles/service-mesh-ultimate-guide-2e/>

<https://medium.com/walmartglobaltech/microservices-with-orchestration-and-choreography-d09941b1b4e6>

Sanjay, Jayraj ...this is a good article. Please go through and get the perspective ..map it to situation and think of set of questions to be asked. Our job is ask the right questions, deliberate, share views .... decision will be their and then clearly document and getting approval. Stick to the discipline please.

<https://www.geeksforgeeks.org/microservice-architecture-introduction-challeneges-best-practices>/

<https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#the_modular_monolith>

I got a very good article on the subject we were discussing in the morning. Please have a look.

<https://blog.dreamfactory.com/microservices-examples>/

<https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#the_modular_monolith>