

A PROGRAMMER IN SERVERLESS LAND

A detailed how-to guide for
serverless applications



CONTENTS

A decorative graphic in the top right corner consisting of several overlapping, rounded rectangular bars in red, grey, yellow, and cyan, along with a small cyan circle.

3

Preface

5

01. First steps. First lambda.

19

02. Spinning up the Database.

33

03. Making friends with the database.

48

04. Building REST with ApiGateway.

65

05. We put the finishing touches on our serverless developer's bag of tricks.

79

Conclusion

Taking a serverless approach to development has long been very popular. According to various surveys, developers cite the following advantages of serverless technologies:

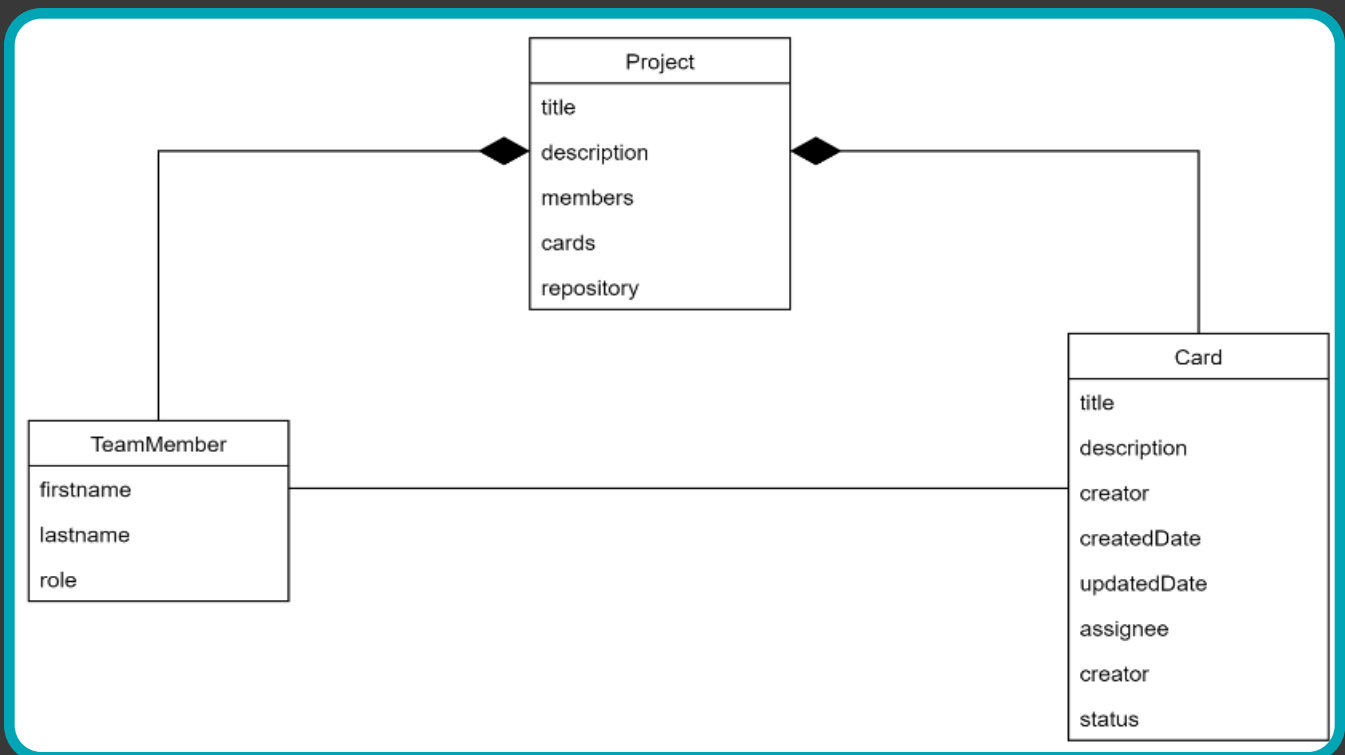
- Flexible scaling
- Speed of development
- Reduced time and cost of administering applications
- Quick releases

The benefits look enticing and promising. But does the approach deliver on the promise?

It's time to get acquainted with serverless technologies. We will analyze the serverless approach through the prism of the experience of creating "classic" applications. This means that there must be tests, the ability to run code locally, the ability to deploy and debug across multiple environments, logs, metrics, and so on. Let's skip past Hello World and take on a challenge with realistic scenarios.

Of course, there are a lot of different articles and instructions on the internet, but few of them take a holistic approach that takes you from bootstrapping your project all the way to post production support. Let's go!

Throughout the guide we apply the options and considerations we discuss to a sample project to create a REST API for a simple bug tracker. The bug tracker has projects, and people who work on these projects. There are tickets - tasks that are assigned to people. Tickets, of course, have an execution status. Each project has its own github repository. It looks something like this:



We will implement the following methods:

Get a project by ID with information about the people who work on it, as well as information about open pull requests.

- Update project information.
- Add a person to the project.
- Remove the person from the project.

Of course, we aren't cutting production code on a Friday night (never do that), nor are we learning coding basics.

We are focusing on the serverless approach and will show how to adapt familiar patterns to this approach. As such, topics beyond this focus, such as the structure of database queries for example, are less important than where and how and with what parameters such code is called. The specifics of database and 3rd party API integration will be assumed to work as per usual for the purposes of our discussion without delving into details.

So now, which serverless provider to choose? Recently, there are more and more of them, even Oracle has its own set of serverless services. That said, we will start with the flagship, AWS - it's stable, well documented, battle tested, and has a large and active developer community that can help to answer our questions when we run into trouble.

Our plan will proceed accordingly:

- Understanding lambda functions and making our first simple lambda function.
- Spinning up the data base.
- Integrating our Lambda function with the database
- Create a REST API and hook up our previously created functions up to it.
- Deploy to production.

CHAPTER 01

FIRST STEPS. FIRST LAMBDA

In this chapter we will create a framework for the application using an approach in which the lambda functions are independent and located in different directories.

We will:

- Implement a simple lambda function capable of returning a fixed object.
- Create the application using SAM using an IaC approach.
- For local development, we will install a plugin for VSCode with which we can run the lambda locally in debug mode.
- Deploy the lambda from the development environment or using console commands.

[Code can be found here.](#)

Where to begin?

First we need to prepare our environment:

Pick your favorite development environment (We will use VSCode)

Docker containerization system

AWS account with full privileges (we will touch on this issue in more detail later)

Let's start with the components that make up a classic serverless application - Amazon provides the AWS Lambda service. Lambda functions are functions that handle events from various sources, ranging from message queues to file update events from S3. The lambda function takes as input a json object as the input representing an event, does whatever it needs to, and then returns a response.

Choosing our programming language

And here we have our first serious decision: what language will we program in? It would seem that we can choose as we wish, but everything is not so simple, and in order to understand the nuances of this question, we are going to need to dive a bit into the documentation and consider how lambdas work.

The life cycle of a lambda consists of three phases:

- Init
- Invoke
- Shutdown

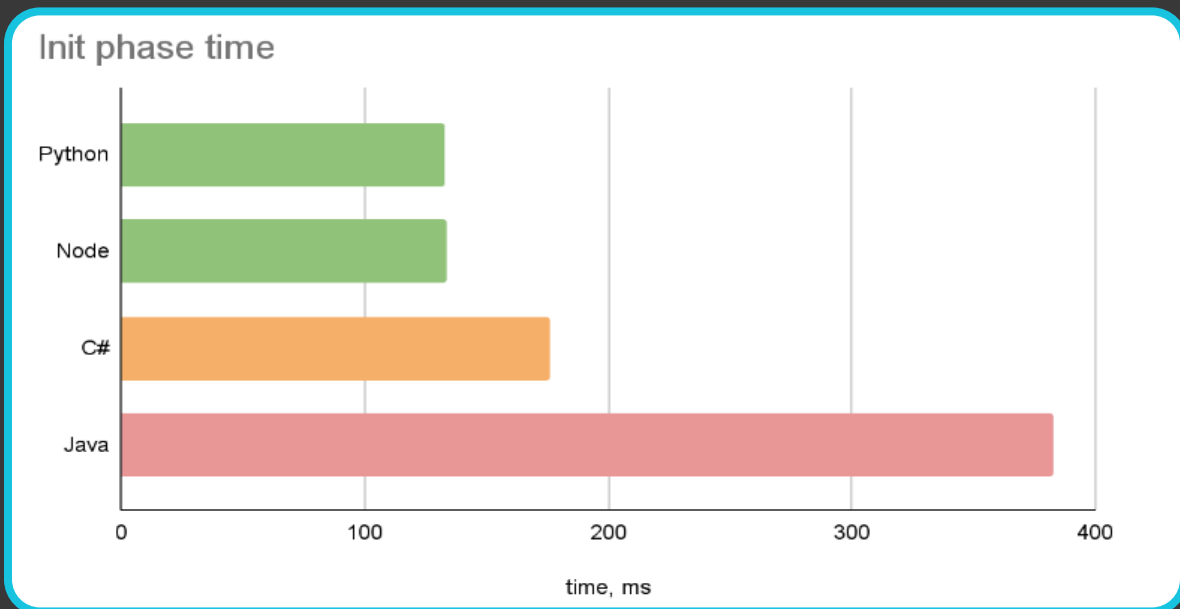
As the name implies, during the Init phase, the runtime environment where the lambda code will execute is stood up, the lambda function code is downloaded, and the constructor and initialization code is launched. This phase “raises” the lambda into the cloud. This init process happens once during the lifetime of a lambda function instance.

After the init phase comes the Invoke phase. In the Invoke phase, the lambda function handler code is directly called. Unlike the Init phase, which occurs once, this phase can be repeated many times depending on how many events are available for processing. Once the handler has returned, Amazon freezes the environment until there is a new event to handle. When a new event occurs, Amazon “unfreezes” the environment. If a critical error occurs while a phase is running, or if the runtime is out of bounds, then Amazon tries to recreate the runtime.

So what does all of this have to do with our choice of programming language? Well, our choice of language directly affects the Init phase. We have an uninstantiated lambda definition waiting for events to arrive. Then, our first event arrives. There is no instance of our lambda in the cloud to handle this event (this also happens when all existing instances are busy processing other events). AWS starts creating a new lambda instance for our event. The init phase “fires”. At this point, we are just waiting for the working environment for the lambda to show up. None of the useful work defined in our lambda has begun processing. After the init phase has completed we can move on to begin doing our real work. This situation is called a cold start - when there are no instances of functions available for processing and it takes time to instantiate these function instances (just like we spend time warming up a car engine in a Minnesota winter). The faster the Init phase passes, the faster the event processing will begin, the faster the client will receive a response, the faster the response time will be. Due to its nature, cold start is different for different programming languages.

For interpreted languages (Python, Nodejs) it is significantly faster than for compiled ones (java, c#). But compiled ones have an advantage in better use of resources, so lambdas have a higher “efficiency” if you need to use several cores or more of memory. Therefore, if you need to minimize cold start time, interpreted languages may be the correct choice. On the other hand, if you need to optimize for resource utilization, then compiled languages may be a more appropriate choice.

The results of a small test of cold-start time of a lambda “Hello World” handler written in different programming languages are shown below.



Based on this data and on our knowledge of Java and Javascript, we have opted to proceed with Node as our language of choice.

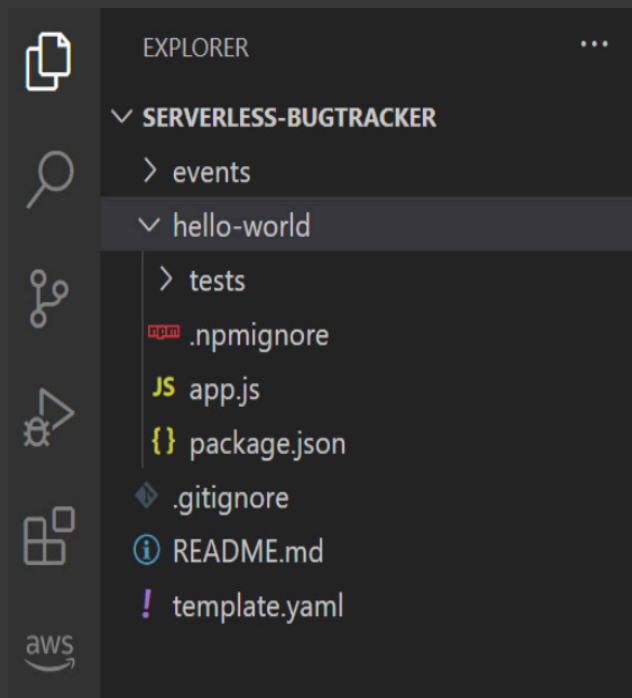
Project Structure

We will start with a lambda that will get information about our project. In principle, you can write code directly in the editor in the Amazon Console. It is enough to create a boilerplate “Hello World” lambda, then you can edit the code of this lambda function. Obviously, this option is doomed in any non-hello-world project, it will be not be impossible to build a normal CI / CD pipeline, and in general it will be inconvenient. You can create a lambda from scratch, or you can use a ready-made template as a basis. The SAM framework will help us with this, an extension of AWS CloudFormation (hereinafter referred to as CF) for Serverless applications. The SAM framework can create a lambda function hello-world template for us, help with deployment to the cloud, and more.

Here we have an example of the SAM framework hello-world template. The following console command will create a hello world project with a lambda function using Nodejs, and the created lambda function will be packed into a zip archive when built.

```
serverless-bugtracker> sam init --package-type Zip --runtime nodejs14.x
--app-template hello-world --name serverless-bugtracker
```

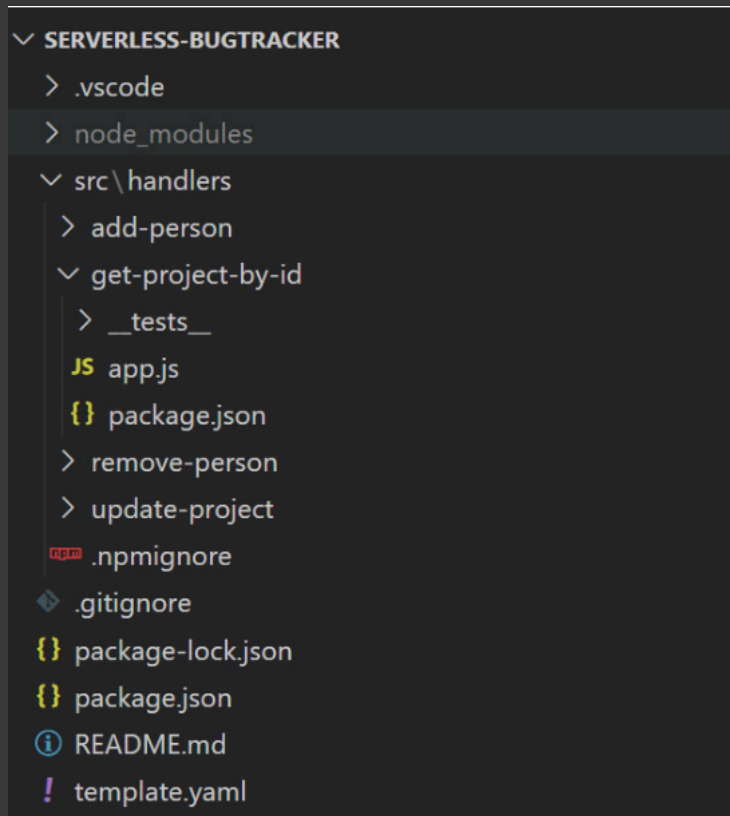
The structure of the first lambda function is as follows:



template.yaml — this is the SAM framework config describing the components that are involved in our project. The SAM template is compatible with the CF template. It adds ways to quickly and easily create serverless application-specific resource types (lambdas, api gateway, etc.). CF will deploy resources from this config that are specified in it. If 10 lambda functions, RDS, SQS are described in template.yaml, then it will create it all. But here it is necessary to observe the limits of reason, because even in the serverless world, you can create a serverless monolith, when dozens of different resources that may not be related to each other are in the same config. This approach makes the application more difficult to maintain.

An application with lambda functions is very similar to an application with a microservice architecture. Each lambda can be thought of as a separate microservice. Therefore, for maximum isolation and independence, we will place the function code in different folders, with their own separate independent package.json. We will leave the root package.json to run the tests.

Final project structure:



Let's pause for a minute and take a closer look at template.yaml. What parts does it consist of, and what are they for?

SAM Template

Let's start with the mundane bits, which contain information about the version of the CF template format and that inform us that we are dealing with a SAM template:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
```

There is a separate block for text a description and global settings:

```
Description: >
  serverless-bugtracker

  My bug tracker.Uses Amazon serverless stack

Globals:
  Function:
    Timeout: 3
```

One of the most important blocks is the resource description block, it describes all the resources for deployment: lambdas, databases, and other services. For example, the following is an example of a lambda function description.

Any resource in a template has a unique logical name. Using this name, you can access the properties of this resource. In our example, the logical name of the function is `GetProjectByIdFunction`.

The `Runtime` variable tells you which environment to use to run the code.

The `CodeUri` and `Handler` define the path to the code. The `Events` block describes the kinds of events handled by the function.

```
Resources:
  GetProjectByIdFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: src/handlers/get-project-by-id
      Handler: app.lambdaHandler
      Runtime: nodejs14.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /hello
            Method: get
```

And the last block that we need to concern ourselves with at this point is the `Outputs` block, which specifies the results of a deployment. It makes sense to add values to this section that may be required for further work or for integrations, for example, the URLs of the created APIs, the hostname of the instantiated database, and so on. Amazon shows these values in the console after deployment. Values are displayed in plain text, so you should never show passwords or secret keys in this block.

```
Outputs:
  GetProjectByIdFunctionApi:
    Description: "ApiGateway endpoint URL for Prod stage for
  GetProjectByIdFunction"
    Value: !Sub
    "https://${ServerlessRestApi}.execute-api.${AWS::Region}.amazonaws.com/Prod/
  hello/"
  GetProjectByIdFunction:
    Description: "GetProjectByIdFunction Lambda Function ARN"
    Value: !GetAtt GetProjectByIdFunction.Arn
  GetProjectByIdFunctionIamRole:
    Description: "Implicit IAM Role created for GetProjectByIdFunction
  function"
    Value: !GetAtt GetProjectByIdFunctionRole.Arn
```

As you can see from the snippets, SAM also creates an API whose requests are processed by lambda functions. In `Outputs`, in addition to the lambda function identifiers (all resources in Amazon have a unique ARN identifier), the URL for interacting with this API is returned. While I will concentrate on lambda functions, I will return to building the API later.

Running the lambda function

Our first lambda function will return fixed data by project ID:

```
exports.lambdaHandler = async (event, context) => {
  return {
    id: 123,
    title: 'My first project',
    description: 'First project to work with serverless. No cards. No
members.',
    cards: [],
    members: []
  };
};
```

Amazon supports several approaches to writing nodejs handlers:

- Using `async/await` constructs. In this case, the handler can return a value or a promise.
- Use of callback functions. In this case, a third callback argument is added to the handler which will be called upon completion.
- Representation of the handler in the form of an ES module (more recently, the nodejs environment began to support this feature)

We will use the first approach, since it is more convenient and familiar to us.

Cool, everything seems to be ready for the first launch. We want to be able to run lambda functions locally in debug mode, so we can quickly find and fix bugs, and check that everything works as it should. SAM has the ability to run lambdas locally. For local launch, a special docker image from Amazon is used. The container created in this way differs little from the runtime in the cloud.

Running code locally

There are several ways to run code locally, building and running with SAM and launching with VSCode.

Building and running with SAM

Before the code can be run locally in the console, it must be built. The application is built using the following command:

```
serverless-bugtracker> sam build
```

This command prepares artifacts for later deployment or launch. The build result for each lambda function is in a folder that contains all the files from the directory specified in the CodeUri in the template. If the CodeUri is empty, then all files from the root directory are copied. Also in this folder, all dependencies from package.json are installed, which is located in the folder specified in the CodeUri.

A nice aspect of the previously chosen project structure is that all artifacts will be independent and will not contain code related to other lambda functions.

There are other approaches you can take to the structure of the project. Even in the examples from Amazon, you can find the option where the files of all functions are in the same folder with one package.json, the CodeUri parameter also points to the same directory (or is absent altogether). Sample project with multiple features from Amazon:

```
▼ SAMPLE-PROJECT
  > __tests__
  > .aws-sam
  > events
  ▼ src\handlers
    JS get-all-items.js
    JS get-by-id.js
    JS put-item.js
  ◆ .gitignore
  ! buildspec.yml
  {} env.json
  {} package.json
  ⓘ README.md
  ! template.yaml
```

Function declaration in SAM template:

```
Resources:
  getAllItemsFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: src/handlers/get-all-items.getAllItemsHandler
      Runtime: nodejs14.x
      ...
  getByIdFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: src/handlers/get-by-id.getByIdHandler
      Runtime: nodejs14.x
      ...
  putItemFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: src/handlers/put-item.putItemHandler
      Runtime: nodejs14.x
      ...
```

With this alternate approach to project structure, the artifacts for different functions will be identical. For example, the first lambda works with the database, and the second one works with S3. The artifact of the first and second lambda will have both dependencies. With this approach, complete isolation is not obtained, moreover, the same dependencies are repeated many times.

The “sam” build command has a couple of useful options. The `-p` switch allows you to build in parallel. By default, all resources are collected sequentially. The `-c` switch allows the assembly to be cached. If there were no changes in the files, then the cached artifact will be reused without recompilation. If the project structure is used with a shared folder for all functions, then the `-c` key will not work, SAM will rebuild all the functions specified in the template each time, since it will always see changes. Using separate folders allows SAM to clearly identify changes and collect only the changed parts of the application. In general, these keys allow you to speed up the assembly and not do extra work.

By default, the build takes place in the `./aws-sam/build` folder. In addition to the function artifacts, there is also `template.yaml` in `./aws-sam/build`. This file is similar to the file from the root of the project, only all the functions in it are set to the build directory `./aws-sam/build`.

After running the build command:

```
serverless-bugtracker> sam build -c -p
```

In the console you can see hints from SAM on further actions:

```
Build Succeeded

Built Artifacts  : .aws-sam\build
Built Template   : .aws-sam\build\template.yaml

Commands you can use next
=====
[*] Invoke Function: sam local invoke
[*] Deploy: sam deploy --guided
```

As you can see from this tip, to run a function locally, you must use the local invoke command. To run, you must specify the function name (logical name from the template).

Optionally, you can specify the event that should be processed by the function. The -e switch is used to send an event. If it is not specified, the event will be empty. You can specify the path to a file with a json object representing this event, or specify - so that the event is read from stdin.

```
serverless-bugtracker> echo {"message": "Hey, are you there?" } | sam  
local invoke GetProjectByIdFunction --event -
```

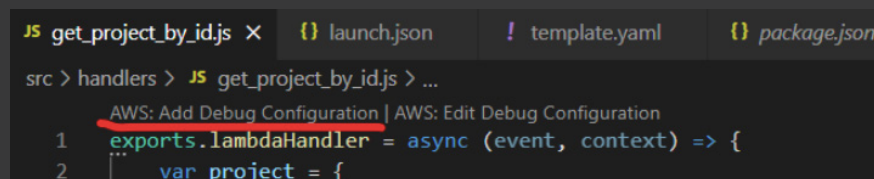
The first run usually takes a long time because the docker image is downloaded. At the end of the work in the console, you can see the result of the execution:

```
START RequestId: afebe87a-84a8-4b69-a430-08ec7f06828b Version: $LATEST  
END RequestId: afebe87a-84a8-4b69-a430-08ec7f06828b  
REPORT RequestId: afebe87a-84a8-4b69-a430-08ec7f06828b Init Duration:  
0.29 ms Duration: 98.90 ms Billed Duration: 100 ms Memory Size: 128  
MB Max Memory Used: 128 MB  
{  
  "id": 123,  
  "title": "My first project",  
  "description": "First project to work with serverless. No cards. No members.",  
  "cards": [],  
  "members": []  
}
```

Launching with VSCode

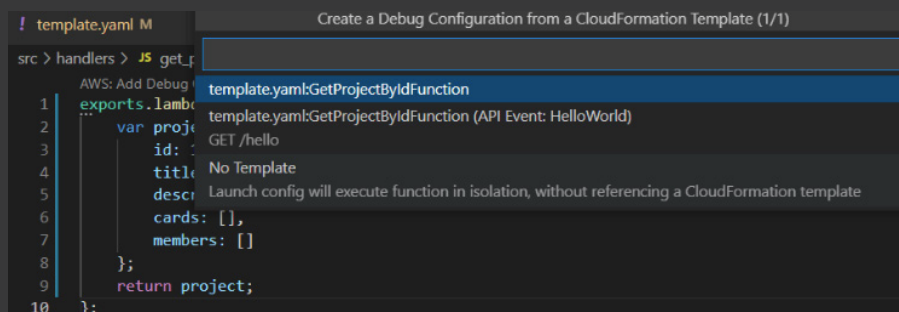
You can run a function locally directly from VSCode. To do this, we need a plugin for VSCode AWS Toolkit. This plugin will allow you to quickly deploy from your favorite development environment, set breakpoints and inspect your code. After installation, you need to configure it - add account credentials so that the plugin has access to the cloud.

After installing the plugin, the command to create a launch configuration appears above the function:



The screenshot shows the VS Code interface with a file explorer on the left containing 'get_project_by_id.js', 'launch.json', 'template.yaml', and 'package.json'. The main editor shows the code for 'get_project_by_id.js' with a red circle highlighting the 'exports.lambdaHandler' property. A context menu is open over this property, offering options: 'AWS: Add Debug Configuration' and 'AWS: Edit Debug Configuration'.

The plugin offers several modes:



The screenshot shows the VS Code interface with a file explorer on the left containing 'template.yaml'. The main editor shows the code for 'template.yaml' with a red circle highlighting the 'exports.lambdaHandler' property. A context menu is open over this property, offering options: 'template.yaml: GetProjectByIdFunction', 'template.yaml: GetProjectByIdFunction (API Event: HelloWorld)', 'GET /hello', 'No Template', 'Launch config will execute function in isolation, without referencing a CloudFormation template', 'cards: []', and 'members: []'.

Where the VSCode approach differs from using the SAM template is only in the type of event with which the function is called. In the first case it can be an arbitrary event, in the second case it is an http request. VSCode will create a `.vscode/launch.json` file containing the lambda function local launch setting. We will run the lambda in the first way, and will return to working with the API in subsequent articles. You will get the following launch config in VSCode:

```
{
  "configurations": [
    {
      "type": "aws-sam",
      "request": "direct-invoke",
      "name": "serverless-bugtracker:GetProjectByIdFunction",
      "invokeTarget": {
        "target": "template",
        "templatePath": "${workspaceFolder}/template.yaml",
        "logicalId": "GetProjectByIdFunction"
      },
      "lambda": {
        "payload": {
          "json": {
            "id": 123
          }
        },
        "environmentVariables": {}
      }
    }
  ]
}
```

The third option simply runs the code from the directory, in which case the code in the SAM template is ignored.

For the third option to work, you will need to install the typescript compiler using the following launch configuration:

```
./vscode/launch.json
```

In the payload section, you can specify a json object - an event that will be processed by the function.

```
{
  "type": "aws-sam",
  "request": "direct-invoke",
  "name": "get-project-by-id:app.lambdaHandler (nodejs14.x)",
  "invokeTarget": {
    "target": "code",
    "projectRoot":
      "${workspaceFolder}/src/handlers/get-project-by-id",
    "lambdaHandler": "app.lambdaHandler"
  },
  "lambda": {
    "runtime": "nodejs14.x",
    "payload": {
      "json": {
        "id": 123
      }
    },
    "environmentVariables": {}
  }
},
```

The result of running any of the options is the same - the application is launched locally in debug mode. The plugin uses the same "sam" build and "sam" local invoke commands, just with a different build directory.

Cloud Deployment

Now we will make the first deployment to the cloud. As with running locally, this can be done in two ways. The first way is to use the console, the second way is to use the plugin. We will evaluate both options.

Approach 1: Deployment using console commands

Let's try to make a deployment, armed with a console and SAM.

Before you can start deploying the application, you need to build it. Let's take a look at the above command.

For deployment, we need the "sam deploy" command. The sam deploy command archives and copies the collected artifacts to an S3 bucket and deploys applications to the cloud. All collected resources are created inside the stack. A CF stack (hereinafter simply a stack) is a set of resources in the cloud that have been deployed using a CF template. Each stack has a unique account name. For now, it is better to run the command with the --guided key. Then the interactive mode will start, where you will need to enter the name of the created stack, the region, the name of the S3 bucket for storing the code, and so on.

But the most interesting thing in this mode is that all selected options can be saved in a separate file with a specific profile (the default file is samconfig.toml, the default profile name is default). Then next time you won't have to specify any parameters other than the profile name.

Here everything is saved under the chapter1 profile.

samconfig.toml

```
version = 0.1
[chapter1]
[chapter1.deploy]
[chapter1.deploy.parameters]
stack_name = "serverless-bugtracker-ch1"
s3_bucket = "serverless-bugtracker-sam"
s3_prefix = "serverless-bugtracker-ch1"
region = "us-east-1"
confirm_changeset = true
capabilities = "CAPABILITY_IAM"
```

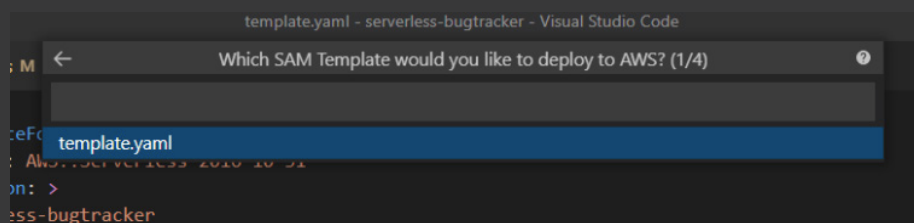
The next time you want to deploy, simply provide the profile name:

```
serverless-bugtracker> sam deploy --config-env=chapter1
```

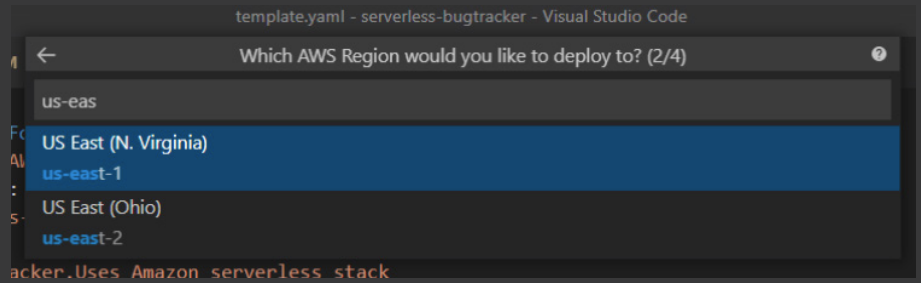
Approach 2: Plugin Deployment

When you right-click on the template.yaml file, a new item "Deploy SAM Application" will appear in the context menu. The plugin will ask:

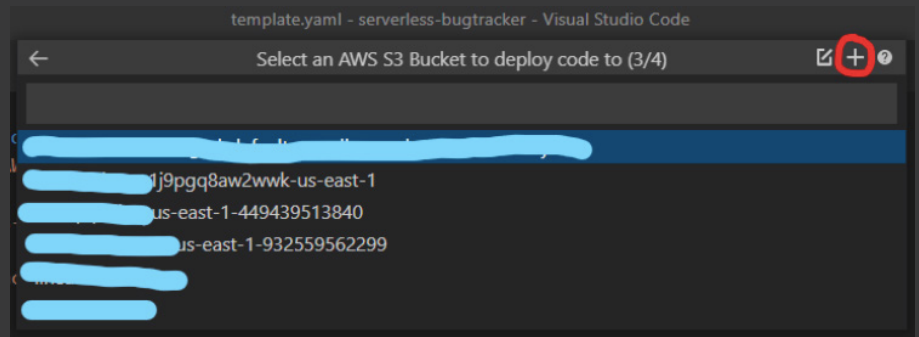
Which template file to expand (We have only one):



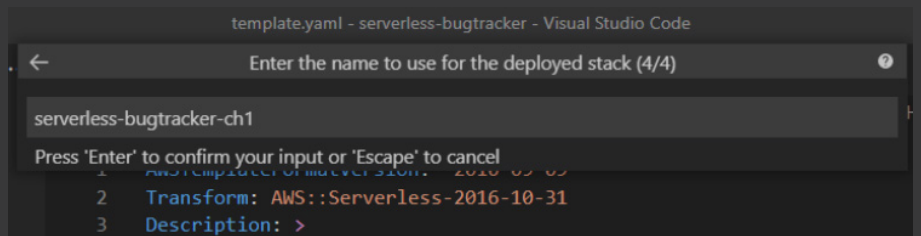
AWS Region:



s3 bucket that SAM will use. You can create a new one:



CF stack name:



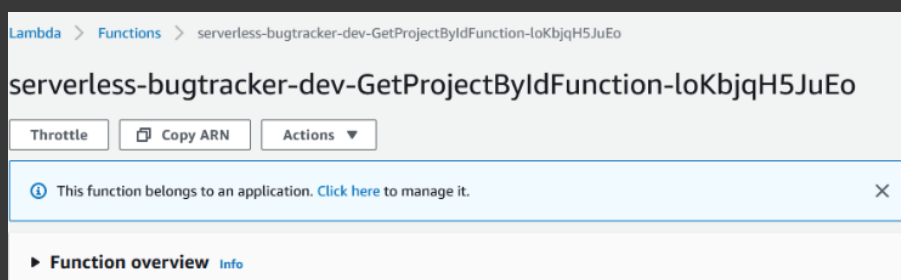
After going through all the steps, AWS Toolkit will run "sam build" and then "sam deploy". The only difference is that the build directory is different from ./aws-sam

Unfortunately, the plugin does not know how to remember the choice of parameter values, so each time you need to enter the same data. In this regard, launching through the console is more convenient.

In both cases, at the end in the console, we see the reassuring line "Successfully deployed SAM Application to CloudFormation Stack: serverless-bugtracker-ch1".

And boom! Our lambda is deployed to the cloud.

Amazon has provided a default name for our function:



In the SAM template, we don't provide a function name. Amazon takes the stack name, the logical name of the function in the template, and adds a random sequence of characters.

You can avoid this default naming convention if you use the `FunctionName` property in the SAM template, then the function name will be the one that you specify in this field.

The function can be tested using AWS Console. If you open a lambda function, it will have a special tab, test. Here I can send any json as an event (similar to the payload json I used in local launch):

Code **Test** Monitor Configuration Aliases Versions

Test event Format Save changes Test

Invoke your function with a test event. Choose a template that matches the service that triggers your function, or enter your event document in JSON.

New event
 Saved event

Template
hello-world

Name
MyEventName

```
1 {  
2   "key1": "value1",  
3   "key2": "value2",  
4   "key3": "value3"  
5 }
```

CHAPTER 02

SPINNING UP A DATABASE

In this chapter we will:

- Create a separate SAM template to deploy the shared global resources
- Using SAM, deploy the database in a separate VPC. The new resources will be deployed on the stack under the name serverless-bugtracker-global-resources
- Configure a cross-stack reference for parameter passing across the deployed stack.

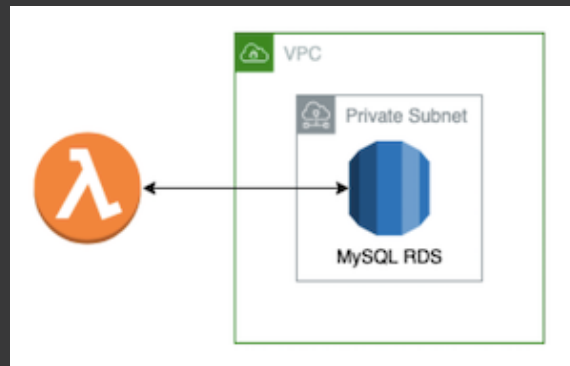
Our Bug Tracker REST API Example project requires a classic relational database, so we will stick with MySQL. Amazon provides the AWS RDS cloud service for working with databases.

[Code can be found here.](#)

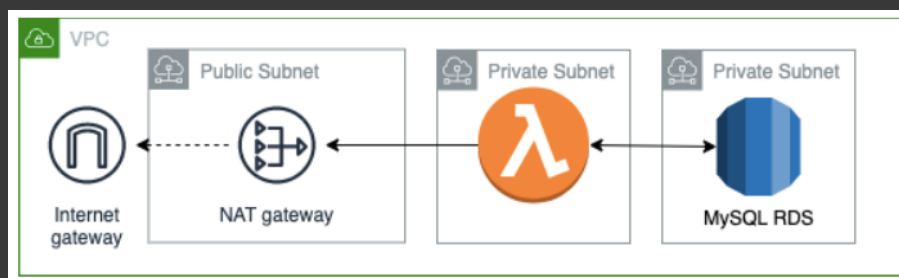
Database Application Architecture

In order to deploy a database in AWS RDS, you first need to create a VPC (Virtual Private Cloud). VPC — this is a way to organize a dedicated virtual network in AWS. We will get a virtual private network in which we will deploy all the resources of our project. After creating a new AWS account, there is already a default VPC that AWS has created for us. But it is good practice to create a separate VPC for the needs of the project, the so-called project VPC - this will allow you to safely and conveniently organize a completely isolated infrastructure for the project. So, for our project, we will create a separate project VPC, in which we will deploy all the resources necessary for our project.

AWS VPC has the concept of public and private subnets, (their intended purpose is easy to grasp from their names). In our case, it makes sense to deploy RDS on a private subnet, because in fact, apart from our lambda functions, no one else will access the database. Plus, this is a more secure and simple at first glance mechanism for organizing access to the database - that is, this is approximately the picture that was



And now the final result that was implemented:



As we can see, many resources have appeared that were not originally there. These are the reasons for the appearance of some of these resources:

- The Internet Gateway and NAT Gateway are needed so that our lambda can interact with the Internet. Since, after the lambda is launched in the VPC, without these resources it will not have access to the Internet.
- A public subnet is needed for NAT Gateway to work.
- An empty private subnet is needed for RDS: one of the requirements when creating an RDS is to have two subnets. According to the AWS documentation, the second subnet is used for storing backups and logs, and can be used for high availability purposes in case of Multi-AZ deployments.

How can all this be created?

The abundance of resources suggests that doing such things manually is something akin to signing your own death warrant.

In this case, a mix is obtained from a SAM application, which is deployed by two teams using the IaC approach, and “manual” resources. In our application, we will achieve maximum uniformity and automation so that in the future it will be easy and quick to set up CI / CD.

We have SAM. Will it be able to deploy such an infrastructure?

At the beginning of this guide, we mentioned that SAM is an extension of CF for Serverless applications. That's to say that SAM adds resources specific to the serverless approach without limiting the use of regular CF resources in the sam template. To deploy VPC, RDS, NAT, we will use standard resources from CF.

Organization of SAM templates in the application

We have decided on our tools. Now we need to decide how will we organize our SAM and CF templates Will we use one template for all resources?

This approach should only be seems to be temporary. With the growth of objects, it will become difficult to maintain a constantly growing file, so some kind of decomposition is required.

AWS provides several options for organizing resources:

- Use the AWS::Include macro.
- Use Nested Stacks.
- Use a separate independent stack for RDS and related resources.

We will compare approaches in order to choose the most convenient and suitable.

Option 1: Use the AWS::Include macro

This method is as easy to understand as possible; the macro allows you to insert a fragment into the configuration file.

Here is an example template.yaml with AWS::Include:
(./global-resources/template.yaml)

```
Resources:
  'Fn::Transform':
    Name: 'AWS::Include'
    Parameters:
      Location : global-resources/rds.yaml
```

(./global-resources/rds.yaml)

```
DB:
  Type: 'AWS::RDS::DBInstance'
  Properties:
    Engine: MySQL
    ...
```

At the time of deployment, CF will replace this macro with a fragment and then it will work with the changed configuration file.

This option is suitable when:

- You already want to make some kind of logical breakdown in a small template
- You want to reuse some fragments in one or more CF templates.

It is not suitable when:

- You plan to create a sprawling layered structure of nested resources (Amazon does not allow an Include to be included in another Include fragment).
- The number of resources in a template approaches AWS limits.

Option 2: Use Nested Stacks

Inside the root template, resources of the type `AWS::Serverless::Application` are created. Each such resource is a nested CF stack. Expanding the root stack will expand all of its nested stacks.

The structure is as follows:

(./global-resources/template.yaml)

```
...
Resources:
  RDS:
    Type: AWS::Serverless::Application
    Properties:
      Location: rds.yaml
```

(./global-resources/rds.yaml)

```
AWSTemplateFormatVersion: 2010-09-09
Description: "AWS CloudFormation for creating an Amazon RDS DB instance"

Resources:
  DB:
    Type: 'AWS::RDS::DBInstance'
    Properties:
      Engine: MySQL
    ...
```

Unlike `Include`, which only inserts a config fragment, `AWS::Serverless::Application` resources are full stacks that can be developed and deployed separately.

This approach allows applications to scale and create more resources than can be supported in a single file.

This option is quite versatile. You can break the application into arbitrarily complex parts and develop these parts separately from other stacks.

Option 3: Use a separate independent CF stack for RDS and related resources

The most obvious option is to make a separate independent stack. Separate file, separate stack, no links between configuration files.

However, because the Nested Stacks method covers all occasions, this method appears superfluous. When would this approach be useful?

Our application requires a datastore. To save money, we can use the same DB Instance for multiple environments by using different database names at the MySQL level. We can also use this instance for local development. Then Dev, QA, feature environments will live within one database instance.

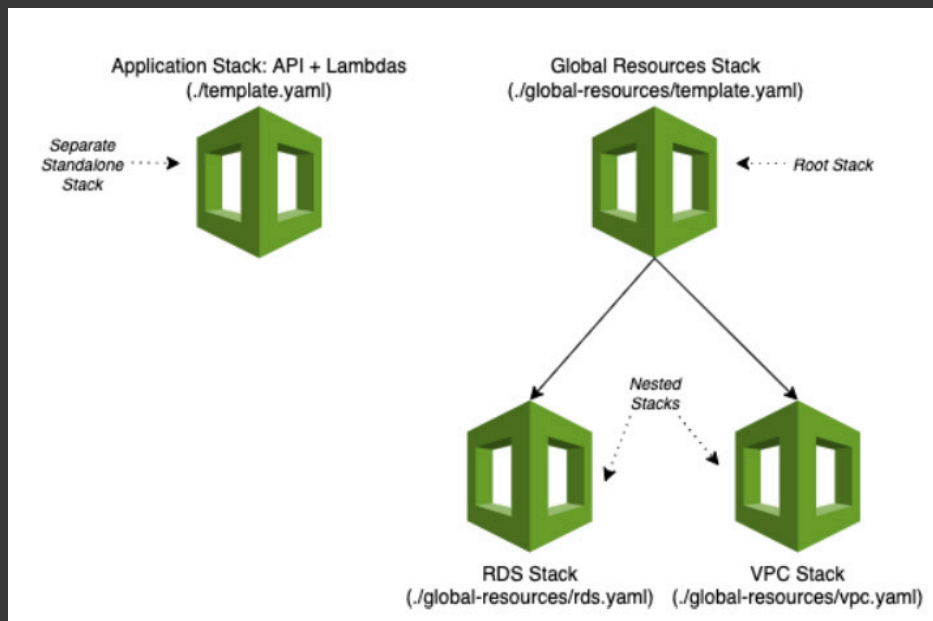
Using the first or second approach will deploy a separate DB Instance for each stack. Of course, the more developers in the team, the more feature environments will be required, and the more our AWS environment will cost each month.

Thus, the third option is applicable in the case of the existence of global resources within the project, when one resource is used in several environments.

Examples: a shared database, a shared VPC (in which all application elements are running), shared roles, etc.

Organizing SAM templates in the application

For our bug tracker REST API example, we see no reason to overpay for separate databases on each environment, so the database will be global. Accordingly, we settled on a combination of the second and third options: the third option is for organizing global resources that are common between environments, and the second is for decomposing one large stack into parts. Inside the stack with global resources, we have two components: RDS and VPC. For ease of support, we will place them in a separate nested stack.



The final structure of the files in which the application resources are located will be as follows:

```
serverless-bugtracker/
├── global-resources
│   ├── rds.yaml
│   ├── template.yaml
│   └── vpc.yaml
└── template.yaml
```

Where:

- ./global-resources/template.yaml - global resource template
- ./global-resources/rds.yaml - nested template for database settings
- ./global-resources/vpc.yaml - nested template for VPC settings and related resources
- ./template.yaml - template for serverless application resources (lambdas)

Deploying a database using a SAM template

VPC setup

We'll start by setting up the VPC and the components associated with it. To simplify the creation of a VPC and all related resources, the easiest way is to take an almost ready-to-use template from AWS as a starting point. - AWS CloudFormation VPC template - AWS CodeBuild. It implements a VPC with two public and two private subnets, Internet and NAT gateway, necessary routing tables, etc. But we have to slightly tweak the specified template to meet our needs - in particular, we need to remove the second public subnet and NAT Gateway for it, as well as some of the template parameters.

The template for creating a VPC is actually quite large, so we will not give all the contents here, instead small fragments will be provided. The final results can be viewed at the above link.

Some points of clarification:

For the nested vpc.yaml stack, we have made the RootStackName parameter. This value (the name of the parent stack) is actively involved in the composition of resource names. We did this on purpose so that the names can quickly and easily identify the application and the parent stack.

Lambda functions need to set up VPC parameters, but the corresponding resources are in a different stack. We cannot directly use resources from other stacks. For these purposes CF has a cross-stack reference mechanism that provides the ability to refer and use data between stacks.

To create a cross-stack reference, use the Export setting for the Outputs data. This setting specifies a region-unique name in the AWS account.

To import this value later, you need to know the Export name and the stack name. Creating a VPC and all related resources is implemented as a separate nested stack. The name of this stack is derived from the name of the parent stack, the logical resource name `AWS::Serverless::Application` in the parent template, and an additional Amazon-generated string. If we recreate the nested stack, then this generated string will be different.

It turns out that the nested stack name can change, unlike the parent stack name, which we set ourselves. It is more convenient to work with immutable names. Therefore, we have configured the export of variables from the parent template `./global-resources/template.yaml`.

VPC Setup

Thus, in order to make data from the VPC stack available to other 3rd party stacks, we need to:

1. Pass the desired values from `./global-resources/vpc.yaml` to `./global-resources/template.yaml`.
2. Set Export for Outputs values in `./global-resources/template.yaml`.

(`vpc.yaml`):

```
Outputs:
...
PrivateSubnet00:
  Description: A reference to the private subnet in the 1st Availability Zone
  Value: !Ref PrivateSubnet00
LambdaSecurityGroup:
  Description: "Security group for Lambda Functions"
  Value: !Ref LambdaSecurityGroup
```

Previously, we used the Outputs block in the template just to see the output values in the console. Now we use them to pass data from the nested stack to the parent.

The Ref function can do two things:

1. Return the parameter value by its logical name.
 2. Return resource id by its logical name
- (`./global-resources/template.yaml`).

```
Outputs:
  LambdaSubnet:
    Description: "Export the ID of created Subnet for Lambda Stack"
    Value:
      !GetAtt [VPC, Outputs.PrivateSubnet00]
    Export:
      Name: !Sub '${AWS::StackName}-LambdaSubnet'
  LambdaSecurityGroup:
    Description: "Export the ID of created SecurityGroup for Lambda Stack"
    Value:
      !GetAtt [VPC, Outputs.LambdaSecurityGroup]
    Export:
      Name: !Sub '${AWS::StackName}-LambdaSecurityGroup'
```

From the `vpc.yaml` nested stack, we returned the `PrivateSubnet00` and `LambdaSecurityGroup` IDs. Using CF's built-in `GetAtt` function, any resource property can be retrieved. Including the Outputs properties from nested stacks.

The built-in `Sub` function replaces `${MyVar}` variables in a string with the given values. In our case, we use the `AWS::StackName` global variable, which is replaced with the name of the stack where the deployment takes place.

RDC Setup

Now it's RDS's turn. As in the case of the VPC, we will pass the `RootStackName` parameter to name the resources

`./global-resources/rds.yaml:`

```
DB:
  Type: 'AWS::RDS::DBInstance'
  Properties:
    DBInstanceIdentifier:
      Fn::Join:
        - ""
        - - !Ref RootStackName
          - '-db'
    DBInstanceClass: !Ref DBInstanceClass
    AllocatedStorage: !Ref DBAllocatedStorage
    Engine: MySQL
    MasterUsername: MyName
    MasterUserPassword: MyPassword
```

In the `DBInstanceIdentifier` property, we set the instance name, which is assembled from the stack name. We also made the type of the instance and the size of the disk for the database a parameter. These are the parameters that will differ for instances on different environments, for example, production machines can be much more powerful, but at the same time a `db.t3.micro` instance may be enough for development and lower environments.

We already have a VPC, so we need to register it in the RDS settings along with subnet add-ons, this is required to run RDS on a private subnet. To do this, we need to pass parameters from `vpc.yaml` to `rds.yaml`. Since both of these stacks have the same parent, to transfer data between them, we just need to pass parameters through the parent stack to transfer data between them.

`./global-resources/template.yaml`

```
RDS:
  Type: AWS::Serverless::Application
  Properties:
    Location: rds.yaml
  Parameters:
    RootStackName: !Ref AWS::StackName
    DBInstanceClass: !Ref DBInstanceClass
    DBAllocatedStorage: !Ref DBAllocatedStorage
    PrivateSubnet00: !GetAtt VPC.Outputs.PrivateSubnet00
    PrivateSubnet01: !GetAtt VPC.Outputs.PrivateSubnet01
    DBSecurityGroup: !GetAtt VPC.Outputs.DBSecurityGroup
  DependsOn:
    - VPC
```

The “`DependsOn`” property allows you to set the order in which resources are created. Obviously, the RDS resource must be created after the VPC.

To run a database in a VPC, you need to configure DBSubnetGroup and VPCSecurityGroups.

./global-resources/rds.yaml

```
Parameters:
  PrivateSubnet00:
    Type: String
    Description: The shared value will be passed to this parameter by parent stack.
  PrivateSubnet01:
    Type: String
    Description: The shared value will be passed to this parameter by parent stack.
  DBSecurityGroup:
    Type: String
    Description: The shared value will be passed to this parameter by parent stack.
Resources:
  DBSubnetGroup:
    Type: "AWS::RDS::DBSubnetGroup"
    Properties:
      DBSubnetGroupName:
        Fn::Join:
          - ""
          - - !Ref RootStackName
            - '-dbsubnetgn'
      DBSubnetGroupDescription: Subnet Group for DB
      SubnetIds:
        - !Ref PrivateSubnet00
        - !Ref PrivateSubnet01
  DB:
    Type: 'AWS::RDS::DBInstance'
    ...
    Properties:
      ...
      DBSubnetGroupName: !Ref DBSubnetGroup
      VPCSecurityGroups:
        - !Ref DBSecurityGroup
```

Storing settings in the cloud

Another important consideration when creating an RDS is how to set the necessary login, password, and database name. As you can see, right now we just have the values hardwired into the template, which, of course, is unacceptable in a real normal project.

You could take advantage of the CF parameters that can be passed during stack creation where the parameters will be passed in the clear to the CF. But this approach is not suitable for passing secret parameters (passwords, keys). And it's inconvenient, because these parameters will have to be entered every time you deploy. You could, of course, set default values. But this will result in account names and passwords being presented in text form in the template code.

The best option is to use one of the AWS services for storing parameters. When using such services, our template will not have parameters in clear text.

Amazon provides several services for storing various parameters, including secret data (passwords, keys, tokens, etc.):

- Systems Manager Parameter Store (further SSM)
- Secrets Manager
- AppConfig

From the point of view of storage and use, Secrets Manager and SSM services have very similar basic functionality, but Secrets Manager, unlike SSM, has several additional features:

- Scheduled password generation.
- Password rotation for RDS, Redshift, etc.

Accordingly, it makes sense to use Secrets Manager if the above functionality is needed in the application. Also, worth considering, Secrets Manager costs a little more than SSM Parameter Store.

AppConfig is a service that allows you to work with the entire configuration at once (whereas the other services essentially allow you to work with parameters individually). For configuration changes, an analog of the CD (continuous delivery) process is provided when these parameters are gradually or immediately applied to the environment. This service is well suited for changing dynamic settings, while the first 2 services are tailored more to work with static, unchanging (or rarely changing) parameters. The ideal scenario for AppConfig is feature toggles.

For our bug tracker REST API example, we do not plan to do automatic password rotation given that our scenario is quite simple, and that the parameters will not change in the process of running the application on the fly, so we will use the SSM service for our application.

SSM stores any parameters as a regular key-value storage. There is a special secured data type for secret parameters. We will store the login in the usual String field, and we will put the password in the SecureString field. CF can be used to create parameters in SSM. But this only works for regular data types.

The secured field will have to be created by hand. For consistency, we will create all the parameters in SSM by hand.

As long as we have only one service/component in our account we can name our parameters whatever we want. But as soon as there are more applications, we will have to put things in order so instead we will do this proactively. It is good practice to store parameters in a hierarchical model, such as `/[environment]/[service]/[parameter]`. This will allow us to freely navigate dozens of different values for different applications and environments in the future. For now, we have 2 global settings (login and password) and one that will be different on different environments (database name on the server):

<input type="checkbox"/>	Name	Tier	Type	Last modified
<input type="checkbox"/>	/dev/serverless-bugtracker/db-name	Standard	String	Fri, 21 Jan 2022 14:29:45 GMT
<input type="checkbox"/>	/global/serverless-bugtracker/db-login	Standard	String	Wed, 26 Jan 2022 11:29:55 GMT
<input type="checkbox"/>	/global/serverless-bugtracker/db-password	Standard	SecureString	Wed, 26 Jan 2022 11:31:29 GMT

The only thing left is to pass the RDS parameters and settings to the appropriate properties.

There are two ways to work with parameters in a SAM/CF template:

- Use stack parameters.
- Use dynamic links.

Option 1 Use a section in the SAM template - Parameters.

In this block, you can describe the template parameters, the values of which we can set when deploying the stack.

To use a value from SSM as a parameter value, you must specify the `AWS::SSM::Parameter::Value` type:

Parameters:

DbLogin:

Description: **Required. Password stored in SSM**

Type: `AWS::SSM::Parameter::Value<String>`

Default: `/global/serverless-bugtracker/db-login`

CF will take the parameters from the SSM with the key name specified in the Default value. Now we can work with the DbLogin values in the template using the function Ref:

DB:

Type: `'AWS::RDS::DBInstance'`

Properties:

...

Engine: `MySQL`

MasterUsername: `!Ref DbLogin`

But this approach is not supported in CF with SecureString options so what are we going to do about the password?

Option 2. Use dynamic links

A dynamic link is a line of the form:

'{{resolve:ssm:parameter-name:version}}' - for regular parameters.

'{{resolve:ssm-secure:parameter-name:version}}' for encrypted parameters.

CF replaces these dynamic references to specific resources when deploying code. However, dynamic links have limitations - SSM-secure parameters are supported only by a limited list of services, including RDS, Redshift. For example, you can't pass parameters to lambda functions this way.

For RDS settings, when transferring passwords we can only use dynamic links, but when transferring a login both options are viable.

In order to avoid mixed metaphors in our configuration file we will use dynamic links in all cases.

However, if these values were used several times in our template, then the use of dynamic links would be less convenient. In the case of renaming the parameter name, you would have to change the code in several places.

./global-resources/rds.yaml:

```
Engine: MySQL
MasterUsername: '{{resolve:ssm:/global/serverless-bugtracker/db-login}}'
MasterUserPassword: '{{resolve:ssm-secure:/global/serverless-bugtracker/db-password}}'
```

Each entry in SSM has a version. You can refer to a specific version of the parameter from the change history, or you can always use the latest version. If the version is unspecified CF will take the latest one.

So, the VPC is ready, the database is ready.

Lets deploy our new stack in the cloud under the name: serverless-bugtracker-global-resources. AWS has created several stacks:

Stack name	Status	Created time	Description
serverless-bugtracker-global-resources-rds-1U24LUOLSC2PP <small>NESTED</small>	UPDATE_COMPLETE	2021-11-15 16:47:28 UTC+0600	AWS CloudFormation for creating an Amazon RDS DB instance
serverless-bugtracker-global-resources-vpc-1FEP5GBUQCOT9 <small>NESTED</small>	CREATE_COMPLETE	2021-11-15 16:43:21 UTC+0600	This template deploys a VPC, with a pair of public and private subnets spread across two Availability Zones. It deploys an internet gateway, with a default route on the public subnets. It deploys a NAT gateway, and default route for it in the private subnets.
serverless-bugtracker-global-resources	UPDATE_COMPLETE	2021-11-15 16:43:05 UTC+0600	Global Resources CloudFormation Template for BugTracker App global resources

If you look closely, you can see that Amazon adds a generated string to each nested stack. The stack is created in the cloud and now we can use the following variables in other templates:

serverless-bugtracker-global-resources-LambdaSecurityGroup

serverless-bugtracker-global-resources-LambdaSubnet

We will leave out the questions about creating the database structure, and how the data ends up there, after all, the article is not about that. These can be ordinary DDL scripts, or these data and tables can be created by hand. You can use special programs to manage the database structure: flyway or liquibase among others.

CHAPTER 03

HOW TO MAKE FRIENDS WITH THE DATABASE

In this chapter we will write the code that will interact with the database. For convenience, when deploying code in this chapter we will use a separate `serverless-bugtracker-ch3` stack. For this stack, we will add a new profile to `./samconfig.toml`.

We will have the following intermediate results:

- There will be a lambda function that subtracts an entry from RDS by ID.
- There will be a SAM template that allows you to deploy the entire application (lambda function + database + VPC settings).
- The Lambda will work both locally and in the cloud, without putting undue constraints on the developer.
- Common dependencies and auxiliary utilities will be separated into a separate layer, which is shared across all functions.
- We will move from initial stub code to integration with the database and have gained sufficient knowledge

[Code can be found here.](#)

Setting up work in VPC

In order for the lambda function to be able to access the database in RDS, we need to configure it to work in the same VPC where the database is already deployed - otherwise there will be no access to the database. By default, there is a separate VPC for lambda functions in the AWS cloud, and of course, it is not related to our project VPC in any way.

To run the lambda in our VPC, we need to add the `VpcConfig` property with two parameters pointing to the subnet in which the network interface for the lambda will be created, and to the corresponding security group. All necessary VPC resources have already been created, and a cross-stack reference for VPC parameters has also been configured.

The global resources are deployed on a stack named `serverless-bugtracker-global-resources`. We don't want to hard-code the name of this stack into our template, so in `./template.yaml` we'll make a new `GlobalResourceStack` parameter - the name of the stack with global resources. This is just in case we want to rename the stack with global resources, or if we want another DB environment for some experimentation.

The default value will be `serverless-bugtracker-global-resources`, so it will take less fiddling to deploy the main stack. Parameters are imported using the `ImportValue` function, which returns a value from another stack using cross-stack-reference.

`./template.yaml`:

```
Parameters:
  GlobalResourceStack:
    Description: Name of the global resources stack
    Type: String
    Default: serverless-bugtracker-global-resources
Resources:
  ...
  GetProjectByIdFunction:
    Type: AWS::Serverless::Function
    Properties:
  ...
  VpcConfig:
    SecurityGroupIds:
      - !ImportValue
        'Fn::Sub': '${GlobalResourceStack}-LambdaSecurityGroup'
    SubnetIds:
      - !ImportValue
        'Fn::Sub': '${GlobalResourceStack}-LambdaSubnet'
```

Integrating lambda functions with the database

To create a connection to the database, you need to know the login, password, database name, and RDS host name. Obviously, storing these parameters in a lambda function is a very bad decision. All values except host are already stored in SSM and you will need to pass this data to the function.

Passing the login and database name

As we discussed in Chapter 2, you can get regular (not secured) SSM parameters in a CF template through dynamic links or stack parameters.

For lambda functions, dynamic references do not work, so we will use stack parameters to pass values from SSM:

```
Parameters:
  ...
  DbLogin:
    Description: Required. Password stored in SSM
    Type: AWS::SSM::Parameter::Value<String>
    Default: /global/serverless-bugtracker/db-login
  DbName:
    Description: Database name
    Type: AWS::SSM::Parameter::Value<String>
    Default: /dev/serverless-bugtracker/db-name
```

Now these parameters can be passed to the function. The lambda function has a special Environment setting for passing parameters: `./template.yaml`

```
GetProjectByIdFunction:
  Type: AWS::Serverless::Function
  Properties:
    ...
    Environment:
      Variables:
        LOGIN: !Ref DbLogin
        DB_NAME: !Ref DbName
```

Passing the DB host name

In order to create a database connection, you need to know the RDS host. This value can be obtained from the resources that are specified in the `rds.yaml` template. We already set up a cross-stack reference for VPC parameters

Now we will do the same for the host. (`./global-resources/rds.yaml`):

```
Outputs:
  DBHost:
    Description: database host
    Value: !Sub "${DB.Endpoint.Address}"
```

Export the variable to `./global-resources/template.yaml`:

```
Outputs:
  DBHost:
    Description: database host
    Value: !GetAtt rds.Outputs.DBHost
    Export:
      Name: !Sub '${AWS::StackName}-DBHost'
```

Now you can use the variable in the lambda:

```
Environment:
  Variables:
    LOGIN: !Ref DbLogin
    HOST: !ImportValue
      'Fn::Sub': '${GlobalResourceStack}-DBHost'
```

All possible parameters have been passed through environment variables leaving the password to be passed to the lambda in a different fashion.

Supplying the password

Of all the parameters, only the password remains. Neither stack parameters nor dynamic references are suitable for passing secured data. So how can we proceed? There is one more option - to independently receive the parameter in the function code, using the client code.

To implement this option, you need a client for the SSM service:

```
serverless-bugtracker/src/handlers/get-project-by-id> npm install
@aws-sdk/client-ssm
```

Obviously, if we request a parameter on each request, then we will experience significant slowdown. In, in addition, there is a limit on the maximum number of requests allowed to the SSM service API (40 requests per second by default). Therefore, the parameter must either be read rarely, for example, once during the entire operation of the lambda function, or this value must be cached for a reasonably long period of time.

Another advantage of caching is the ability to detect changes in SSM. If we change a setting in SSM, then our functions will apply the new settings after some time without rebooting or updating the environment.

Since we are not planning to support changes in the settings, and in general the database connection parameters are unlikely to change, we will read the parameter once. In this case, we will increase the cold start time, but will not increase the processing time for subsequent requests:

```
./src/handlers/get-project-by-id/app.js
```

```
const { SSMClient, GetParameterCommand } = require("@aws-sdk/client-ssm");
const fetchParamsPromise =
  getParameterFromSsm("/global/serverless-bugtracker/db-password");
const client = new SSMClient({ region: "us-east-1" });

exports.lambdaHandler = async (event, context) => {
  try {
    const dbPassword = await fetchParamsPromise;
    //lambda handler code
    //...
  } catch (err) {
    console.log("Cannot retrieve project by id", err);
  }
};

async function getParameterFromSsm(parameter) {
  console.log("Fetching parameter from SSM...");
  try {
    const input = {
      Name: parameter,
      WithDecryption: true
    };
    const cmd = new GetParameterCommand(input);
    const result = await client.send(cmd);
    return result.Parameter.Value ?? result.Parameter;
  } catch(error) {
    console.log("Cannot fetch parameter", error);
    throw error;
  }
}
```

The `fetchParamsPromise` variable will be calculated once during the lifetime of the function instance, and the result of its execution will be used each time the instance processes another request.

All code outside of the main `lambdaHandler` runs in the `Init` phase of the lambda function.

In general, any initializations are best done outside the request handler for several reasons.

- First, the initialization phase is free. Amazon doesn't take this phase into account when it calculates fees for using lambda functions. Therefore, to save money, it makes sense to use the `Init` phase to the maximum by placing the initialization of variables and data preparation there when possible. That said, we must remember that the `Init` phase is not an infinite resource and should not exceed 10 seconds. Otherwise, Amazon will consider running your lambda an error and will try to create it again, but this time it will be charged.

- Second, during the `Init` phase, Amazon provides more computing power. This means that computation logic can run much faster if it is run in the initialization phase of the lambda function.

For our bug tracker REST API example, we create a promise in the initialization block that requests data from SSM.

Unfortunately, we can't wait for the result of execution in the initialization phase, due to the asynchrony of `ssm-client`. And in fact, the response is received when the event processing has begun. In our case, this would hardly have affected anything, because I / O operations work the same regardless of the phase in which they occur.

(Not long ago, AWS has started supporting `await`s declared at the top level (outside of `async` functions), but this only works for ES modules.)

Querying the database

Now we can start integrating with RDS. Since each instance of the lambda function will process exactly one request synchronously, there is no need to implement connection pools. In the lambda function handler, we will create exactly one connection.

If we create a connection to the database on each request, then the time of our function will be longer and this time will incur fees. Ideally, we can create a connection outside the function handler, but getting SSM parameters is possible only inside an `async` function (for non-es-modules), so the creation of the connection will have to be placed in the function handler. But we must take care that the connection is created once for the entire lifetime of the function instance.

All values passed in the template via `Environment.Variables` become available in the code as global variables, so there is all the data to create a connection:

`./src/handlers/get-project-by-id/app.js`

```
const mysql = require('mysql2/promise');
const { SSMClient, GetParameterCommand } = require("@aws-sdk/client-ssm");

const client = new SSMClient({ region: "us-east-1" });
const fetchParamsPromise =
  getParameterFromSsm("/global/serverless-bugtracker/db-password");

let dbConnection;

exports.lambdaHandler = async (event, context) => {
  try {
    if(!dbConnection) {
      const dbPassword = await fetchParamsPromise;
      dbConnection = await mysql.createConnection({
        database: process.env.DB_NAME,
        host: process.env.HOST,
        user: process.env.LOGIN,
        password: dbPassword
      });
    }

    const [rows, fields] = await dbConnection.execute("SELECT * FROM
`projects` where `id` = ?", [event.id]);
    //return result...
  } catch (err) {
    //error handling
  }
};
```

When implementing handlers, keep in mind that all asynchronous actions that are performed in a lambda function must be completed before the handler returns a result. In our case, until the last return. This is a consequence of how AWS treats the runtime after the event has finished processing. **./template.yaml**

As we mentioned in Chapter 1, after the Invoke phase has finished (after the handler has returned), AWS “freezes” the runtime if no further events occur. Incomplete actions can lead to interesting situations.

For example: suppose, in addition to the main logic, in our function it is requires us to update some counter in the database, and we do not expect this operation to end. We would perform an UPDATE but did not expect the result. It is possible that AWS will “freeze” our environment during this operation: the request may not have time to reach the database, and the lambda has already returned a response, thereby provoking a “freeze”. The subsequent event will unfreeze our runtime and our operation and only then will our counter finally be updated.

What does this imply? If you do not wait for asynchronous computations to complete, then the results of these computations may appear with a delay. Therefore, it is always necessary to complete such calculations before the final return of the lambda.

Launching the function

Our lambda is ready. Now we will check if it runs locally. We discussed launch methods in the first chapter. Since global variables are heavily used in the code of our function, a simple local launch will lead to errors. When running locally, SAM does not get parameters from SSM, so you need to fill in these global variables yourself.

If the launch is carried out using a plugin, then the settings are transferred through the VSCode configuration file. For these purposes, the `environmentVariables` section is provided in the settings, in which it is necessary to set the environment variables:

`./vscode/launch.json`

```
"lambda": {
  "payload": {
    "json": {
      "id": 1
    }
  },
  "environmentVariables": {
    "HOST": "SOMEHOST.us-east-1.rds.amazonaws.com",
    "LOGIN": "user",
    "DB_HOST": "bugtracker"
  }
}
```

Parameters are passed during console launch using a special file. The content of this file is a regular json object. The keys are the logical names of the functions from the template, and the values are an object with variables:

`./env.json`:

```
{
  "GetProjectByIdFunction": {
    "HOST": "SOMEHOST.us-east-1.rds.amazonaws.com",
    "LOGIN": "user",
    "DB_HOST": "bugtracker"
  }
}
```

This file must be passed at startup through the console

```
serverless-bugtracker> sam local invoke GetProjectByIdFunction --env-vars
./env.json
```

We run our lambda and get an error when connecting to the database:

```
Cannot retrieve project by id Error: connect ETIMEDOUT
```


A similar error occurs when any client tries to connect. The reason lies in the VPC. We set up lambda access to the database in the cloud, but did nothing to get access from the local machine. Considering that our database is running on a private subnet in AWS, in order to fix this problem, we need to provide access to the database from the local environment.

Globally, there are several possible options for organizing access to the database:

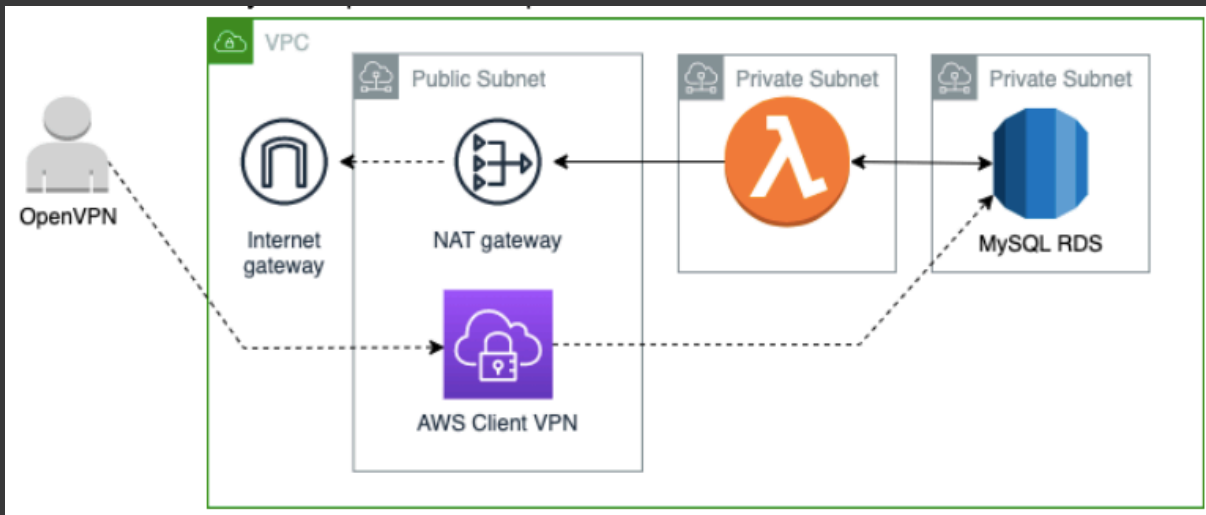
1. Make the database publicly available with a reasonable level of security. Yes, for this project we made it private for a number of reasons, but perhaps for your tasks, the option with a public database for which security measures have been taken is viable. For example, if you have a static IP, then by setting the Security Group you can restrict access to the database only from your IP. In this case, there will be no problems with access to the database when the lambda is launched locally.

2. Making the database private: 3 Ways:

01 Run a separate test database locally.

For example, in a separate docker container or directly on the local machine. That is, the database in RDS remains and will work for the lambda in AWS, and for local testing and development, a separate instance of the database will be used, running, for example, in a Docker container on the local development machine. In principle, there are no special problems with this option - it works fine and in fact does not require global changes. The only thing to do is to use the service's local MySQL address to run locally.

02 Create a VPN connection and connect to the database via VPN There are two ways to implement this option:



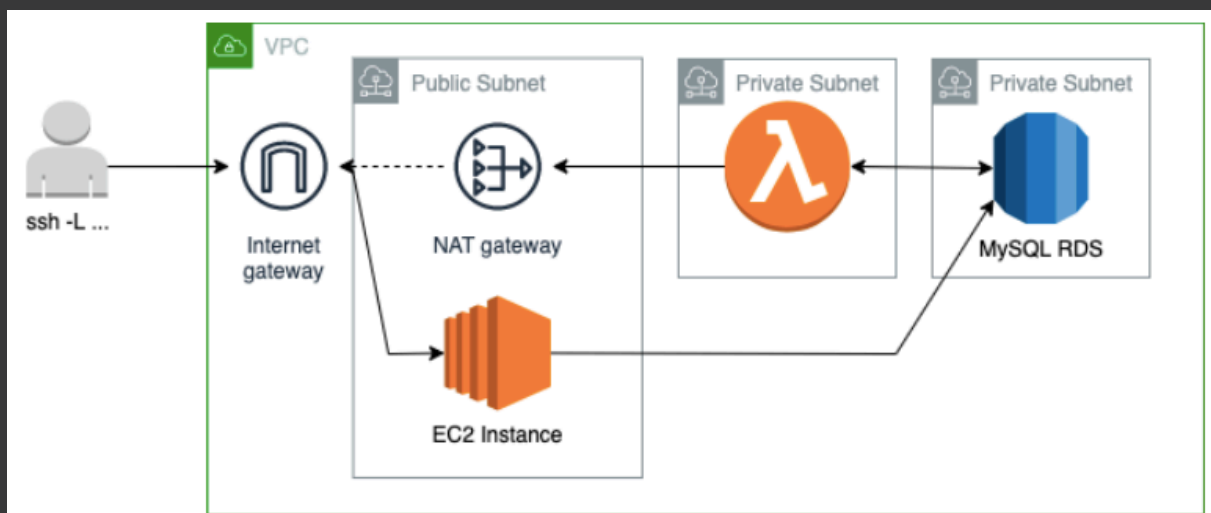
A. Manually install and configure OpenVPN on a separate EC2 instance. In this case, you need skills in configuring the VPN service and routing in AWS. Plus, do not forget about the cost of a constantly running EC2 instance.

B. Use a ready-made VPN service from AWS. The service is called AWS Client VPN and allows you to connect to a VPC using an OpenVPN client. Of the pitfalls when setting up this service, we can note the need to generate / add SSL certificates - you can do this yourself or use a ready-made certificate authority from AWS - ACM Private CA. And if in the case of the first option only the skills of working with OpenSSL are required, then the second option requires significant additional financial investments (~ \$400).

In general, this VPN option requires some technical knowledge and skills in working with AWS/VPN/SSL. At the same time, the cost of this option will be quite high - according to our calculations, current at the date of publication of the article, it is about \$80 per month if you generate certificates yourself, or about \$480 if you use the ACM Private CA service.

03 Make an SSH tunnel to RDS.

This option actually also implements a private tunnel to RDS, but does it through the functionality of the SSH protocol. This option will require the presence of an EC2 instance, through which traffic will actually be tunneled to RDS.



Each of these options has its pros and cons - for our bug tracker REST API example project we implemented an option with a VPN service from AWS and self-preparation of the necessary certificates for its operation. Yes, the cost of this solution is quite high, but in this case it is not a long-term solution, and we enjoyed the added benefit of getting acquainted with this service from AWS at the same time.

Which of the options is preferable for your case depends on the requirements / capabilities, etc., but the result in any case should allow you to fully work with the database when running the lambda locally.

Instructions, as well as code for deploying the AWS VPN service option, can be found in the project repository at the following link.

Take 2: Launching the function

Now we can run the lambda locally, connect to the database. However, deploying the lambda to the cloud we encounter a new error:

```
User:
arn:aws:sts::123456789:assumed-role/serverless-bugtracker-ch3-GetProjectByIdFunctionRo-1N2ELYNEXQPT/get-project-by-id is not authorized
to perform: ssm:GetParameter on resource ...
```

Our lambda function doesn't have enough rights to read the `/global/serverless-bugtracker/db-password` SSM parameter. During the local launch, everything worked because the rights of our account were used. Let's add the appropriate permissions:

```
Policies:
- Version: '2012-10-17'
  Statement:
  - Effect: Allow
    Action:
      - ssm:GetParameter
    Resource:
      - !Sub
        'arn:aws:ssm:${AWS::Region}:${AWS::AccountId}:parameter/global/serverless-bugtracker/db-password'
```

After these changes, the lambda can work both locally and when running in the cloud.

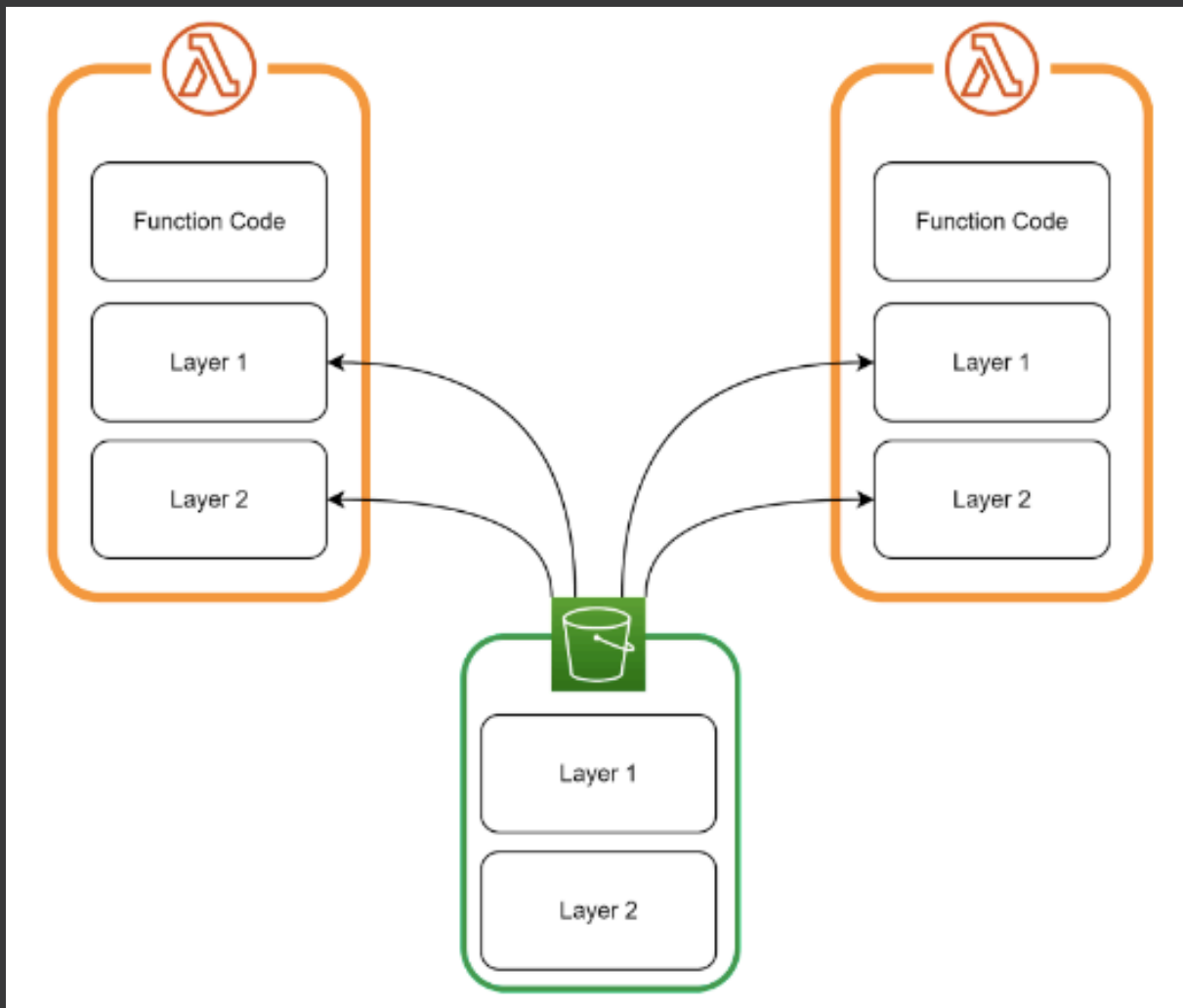
Mission accomplished.

General Dependencies

The first lambda is ready, now we can implement the remaining three. It doesn't seem like there should be any problems, but... It quickly becomes clear that all our functions have similar code, namely the code that receives SSM parameters. After all, each of them requires a password to create a connection to RDS. The situation is clear and the solution is obvious - to separate the common method into a separate file and change the functions so that they use the function from this file.

If you remember the structure that was chosen for the project and how the project assembly works, it will become clear that you need to work with shared code in a different way. After all, only the code that is located in the directory from the CodeUri gets into the artifact after assembly. Do we really need to copy the common code to all artifacts?

Amazon recommends using layers for lambda functions when they share common dependencies. A layer is an archive with code, data, settings. During the "sam" deploy command, the layer is packed into an archive and uploaded to S3. When a function runtime is created, the AWS Lambda service downloads the layers from S3 and unpacks them into the /opt folder. After that, you can work with the files in this folder.



So a layer allows the same data/files to be used in multiple functions. In addition to being a way to share code with layers, this tool allows you to speed up deployment by separating common dependencies into a separate layer. As a rule, dependencies change less often than function code. Highlighting common dependencies leads to a reduction in the size of the function artifact. Smaller - Less time spent loading data into S3, which translates into faster deployment.

Of course, in our bug tracker REST API example application with 4 functions, the difference will not be critical, but if it is supposed to deploy a large number of functions and the sizes of artifacts can be in the tens of megabytes, then the gain will be noticeable.

Layers support versioning. This means that I can set the function to use a specific version of the layer. In this case, updating the layer will not require updating the function.

We will create a layer that will contain an SSM utility class and two shared libraries (mysql2, aws-sam/client-ssm).

We will create a separate directory `src/layers/common-function-dependencies` and will transfer all common dependencies to the layer.

(`src/layers/common-function-dependencies/package.json`)

```
{
  "name": "common-function-dependencies",
  "version": "1.0.0",
  "dependencies": {
    "@aws-sdk/client-ssm": "^3.47.1",
    "mysql2": "^2.3.3"
  }
}
```

Utilities (`src/layers/common-function-dependencies/ssm_utils.js`)

```
const { SSMClient, GetParameterCommand } = require("@aws-sdk/client-ssm");
const util = require("util");
const client = new SSMClient({ region: "us-east-1" });

exports.getParameterFromSsm = async (parameter) => {
  console.log(util.format("Fetching %s from SSM...", parameter));
  ...
}
```

Now we will add the assembly of our layer to the SAM template.

The `ContentUri` property points to the directory where `package.json` is located. The `Retention Policy` property determines the behavior of SAM when the layer is updated. The `Delete` value determines the strategy for when old versions of layers are deleted on upgrade.

```
CommonFunctionDependencies:
  Type: AWS::Serverless::LayerVersion
  Properties:
    LayerName: common-function-dependencies
    Description: Common dependencies for all functions.
    ContentUri: ./src/layers/common-function-dependencies
    CompatibleRuntimes:
      - nodejs14.x
    RetentionPolicy: Delete
  Metadata:
    BuildMethod: nodejs14.x
```

The layer is ready, now it can be connected to the function.

```
Resources:
  GetProjectByIdFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: src/handlers/get-project-by-id
      Handler: app.lambdaHandler
      Runtime: nodejs14.x
      Layers:
        - !Ref CommonFunctionDependencies
```

When an environment is created to run a function, all layers for nodejs are unpacked into the `/opt/nodejs` folder. If several layers contain a file with the same name, then there will be a name conflict that you will have to resolve on your own.

```
sh-4.2# cd /opt/nodejs
sh-4.2# ls
node_modules package.json ssm_utils.js
```

The `/opt/nodejs/node_modules` path is already in the `NODE_PATH` value. Therefore, importing libraries will work from this directory automatically. But the utility file will have to be connected using the full path.

```
const { createConnection } = require('mysql2/promise');
const { getParameterFromSsm } = require('/opt/nodejs/ssm_utils');
```

This code will work locally if run using the SAM CLI or the VSCode plugin. However, such imports will not work in tests. We are using just for testing. To fix the missing dependencies issue, you'll have to tweak just a bit.

package.json:

```
"jest": {
  "moduleNameMapper": {
    "/opt/nodejs/(.*)": "$1"
  },
  "modulePaths": [
    "./src/layers/common-function-dependencies",
    "./src/layers/common-function-dependencies/node_modules"
  ]
},
```

Importing `/opt/nodejs/ssm_utils` in tests will import `ssm_utils` from the `./src/layers/common-function-dependencies` folder. Now you can safely write tests for functions. All mocks will work successfully.

`./src/handlers/get-project-by-id/__tests__/app.test.js`

```
jest.mock('mysql2/promise');
jest.mock('/opt/nodejs/ssm_utils');
const { createConnection } = require('mysql2/promise');
const app = require('../app.js');

beforeEach(() => {
  jest.clearAllMocks();
});
...
```

CHAPTER 04

BUILDING REST WITH APIGATEWAY

In this chapter, we will deal with issues related to building a real API and will also integrate with the github API.

We will collect all of our lambda functions into one API. This API, like our entire application, will be deployed using two sam commands.

The application will still run in debug mode and will be deployed from VSCode.

The API will work in the VPC where the database was previously deployed.

[Code can be found here.](#)

Our first function has been implemented, the database is deployed, our function is integrated with the database and run locally in debug mode and the entire infrastructure for a given environment can be spun up with the help of a few SAM commands. We have moved from initial stub code to integration with the database and have gained sufficient knowledge to implement our remaining three lambda functions. The settings of all of these functions are very similar, they all need the same SSM parameters, the same layer, the same VPC settings. In order not to produce duplication in the template, CF provides a special Globals block in which you can define some general settings for resources.

We'll use this section in the template to define general function settings:

```
./template.yaml
```

```
Globals:
  Function:
    Timeout: 3
    VpcConfig:
      SecurityGroupIds:
        - !ImportValue
          'Fn::Sub': '${GlobalResourceStack}-LambdaSecurityGroup'
      SubnetIds:
        - !ImportValue
          'Fn::Sub': '${GlobalResourceStack}-LambdaSubnet'
    Environment:
      Variables:
        LOGIN: !Ref DbLogin
        DB_NAME: !Ref DbName
        HOST: !ImportValue
          'Fn::Sub': '${GlobalResourceStack}-DBHost'
  Layers:
    - !Ref CommonFunctionDependencies
```

Now we have all the functions that we need. Nevertheless, it doesn't yet look like a complete application. The application does not have an API and our lambda functions don't handle http requests. So in this chapter, we will deal with issues related to building a real API and will also integrate with the github API.

Creating an API in a SAM Template

Our goal is to create a REST API. To process http requests, we will use the ApiGateway service from AWS. It allows you to create an API by integrating with lambda functions to process incoming requests. For each incoming http request, ApiGateway creates a special event which can be further processed by the function.

Our functions already have the necessary logic that allows you to work with the database. But this code works for an incoming event of a completely different format. We need to adapt our code to the event format accepted by ApiGateway. Similar actions will have to be carried out with the response. An example of an incoming event can be found in the events folder at the root of the project. SAM created it when we created the app. In fact, SAM has already made a separate API for us. The template for the lambda function has the Events property

```
./template.yaml
```

This setting describes the event sources that this function will process, the “Api” type defines the ApiGateway service as a source, the template specifies the path, requests for which will be processed by the function.

```
Resources:
  GetProjectByIdFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: src/handlers
      Handler: get_project_by_id.lambdaHandler
      Runtime: nodejs14.x
      FunctionName: get-project-by-id
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /hello
            Method: get
```

As you can see from the example, we did not explicitly specify the ApiGateway resource anywhere. SAM will create it itself. This kind of API creation is called implicit API creation. When building, SAM analyzes all functions in the template, collects all functions that have a filled Events section with resources of the API type, and collects all the paths for which these functions are responsible. It then creates a resource of type AWS::Serverless::Api named ServerlessRestApi. This API already has all the appropriate paths and the specified lambdas are assigned to these paths. The implicit API seems handy for simple, uncomplicated applications.

If a more flexible setting is needed, then an explicit resource definition would be a more convenient option. The minimally configured ApiGateway with the same name as the stack looks like this:

```
BugTrackerApi:
  Type: AWS::Serverless::Api
  Properties:
    StageName: prod
    OpenApiVersion: "3.0.3"
  GetProjectByIdFunction:
    Type: AWS::Serverless::Function
    Properties:
      ...
    Events:
      ApiEvent:
        Type: Api
        Properties:
          Path: /hello
          Method: get
          RestApiId: !Ref BugTrackerApi
```

Here you should stop and familiarize yourself with some of the configuration features of the API Gateway. First of all, we're interested in the StageName parameter, but in order to understand what it is and how to use it correctly, let's first pay attention to a slightly different issue, namely, the issue of organizing project environments.

Multi Stage vs Multi Stack

Earlier, we touched a little on the issue related to CF and environments. We will now return to this issue and try to expand it deeper.

Usually you need to have 3-4 standard fixed environments - Dev, QA, Stg, prod (often these environments are spun up in separate accounts). Additionally, it is often very useful to be able to deploy a feature environment for team members. There are two approaches to creating such environments:

All environments in one SAM/CF stack.

Separate stacks for each environment.

First, we will consider one stack for all environments.

How do we work with lambdas in this sort of scenario?

This is easy if we just need a straight copy of our lambdas in our new environment, but what if we need different versions of different lambdas in different environments for development or testing purposes? In this case, you can use additional features of the functions: versioning and aliases. When cloud-deployed lambda functions are published a version number is assigned and an incremented \$LATEST variable becomes available. In addition to versions, you will need aliases - the text name of the lambda function associated with a specific version

Suppose we have an expanded lambda function MyFunc. You can create an alias DEV_MY_FUNC corresponding to \$LATEST, an alias QA_MY_FUNC for functions version 5, and an alias STG_MY_FUNC for functions version 4. Now three different versions of the same lambda function can run in our cloud at the same time, despite the fact that there is only one resource in my SAM template with lambda function.

In ApiGateway, there are stages for implementations of several environments. Stages essentially mean the version of our API. There may be several such versions. For example, for my Dev/QA/Stg environments, my API will have stages Dev, QA, Stg respectively. Each stage generates separate urls:

https://GATEWAY_HOST/dev/*

https://GATEWAY_HOST/qa/*

https://GATEWAY_HOST/stg/*

As an example, I have a function MyFunc. I want this function to process requests along the /hello path.

```
MyFunc:
  Type: AWS::Serverless::Function
  Properties:
    ...
  Events:
    ApiEvent:
      Type: Api
      Properties:
        Path: /hello
        Method: get
        RestApiId: !Ref MyApiGateway
```

If you connect ApiGateway and a function, then it will process requests:

https://GATEWAY_HOST/dev/hello

https://GATEWAY_HOST/qa/hello

https://GATEWAY_HOST/stg/hello

With this setup, all of our environment APIs are bound to the same lambda function. If we want the dev stage to work with fresh code, and QA with the old stable one we will use aliases to achieve this.

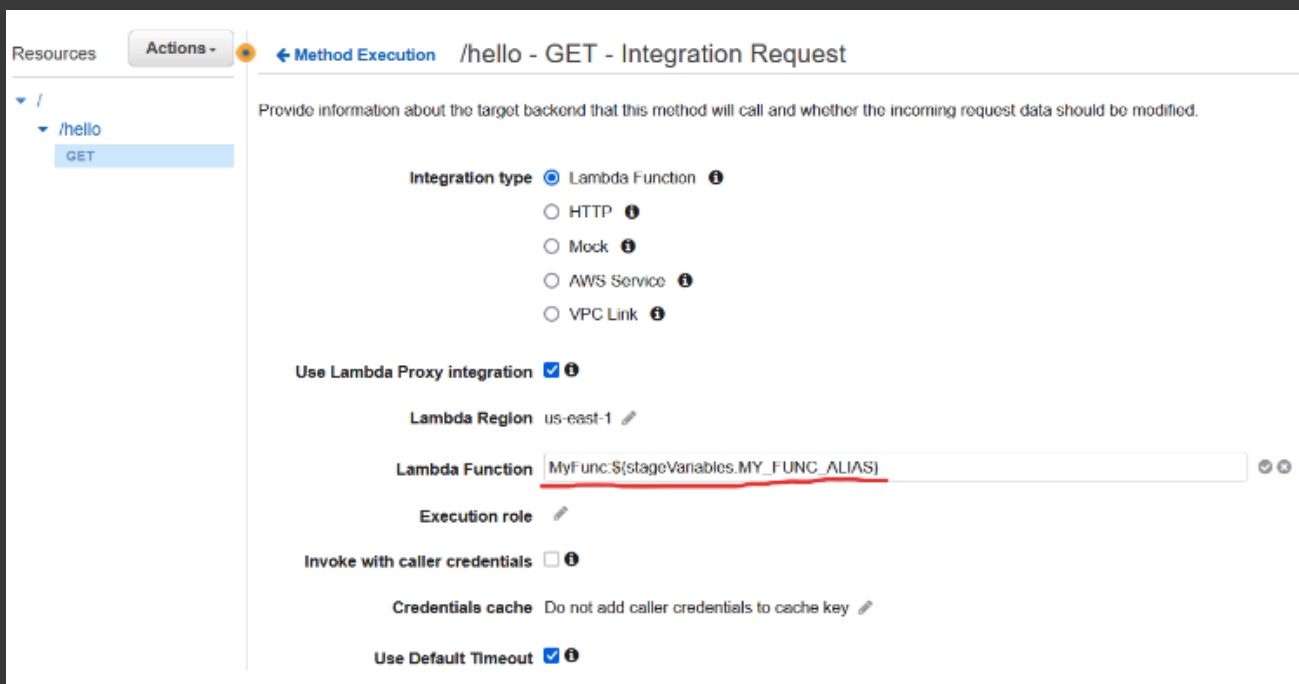
For each stage, it is possible to create variables. We will create a MY_FUNC_ALIAS variable. It will contain the name of our function alias, which matches the environment.

For the dev stage, this will be DEV_MY_FUNC, for qa - QA_MY_FUNC, for stg - STG_MY_FUNC.

Finally, we need to tell ApiGateway not to use the \$LATEST version of our function, but the version specified by the MY_FUNC_ALIAS variable.

Now it remains to parameterize our lambda function version in the ApiGateway settings defined by the variable MyFunc:\${stageVariables.MY_FUNC_ALIAS}.

If it is configured through the UI, then it looks like this:



Then automatically each stage will use the version of the lambda function specific to it.

For this whole mechanism to work properly, it is necessary to publish new versions of functions and update aliases after each implemented feature to the code that is in the main branch. These actions can be done programmatically using the AWS cli.

But what if we want each team member to be able to easily deploy a feature environment in the cloud for development? Each new environment is a new stage. All stages and their variables must be written somewhere in the SAM template (Ideally not by hand!) so that AWS creates ApiGateway with all configurations. The number of stages, aliases and versions will grow significantly.

Here is such a non-trivial scheme obtained for the approach with one stack for all environments.

Features of this approach:

You will have to use more complex resource configuration using CF. Working with stages can only be implemented using CF resources.

Complex CI / CD: actions are added to update versions and aliases after each feature made.

With all environments in “one file”, the probability of an error to break the “foreign” environment increases. Untested code may end up in higher environments.

Creating a feature environment turns into a pain, because you constantly need to modify the template (add new stages, remove old unused ones).

An alternative to this approach is to create a separate stack for each environment using the same template. In this case, the environments are completely isolated. No versions, aliases, stages are needed. The lambda on the stack for dev is different from the lambda for QA. These are different resources, different objects in the cloud with different ARNs. Changing one environment will not affect the other environment in any way. This approach will always work. And with any resources. In Chapter 2, we looked at the option of pushing global shared resources onto a separate stack. This approach goes well with the separation of environments at the stack level.

This option is easier to maintain and implement, so we'll stick with it. This means that our ApiGateway resource in the template should contain only one stage. Let's go ahead and define this to be our production environment.

Okay, we've got Stage figured out. What's the story with OpenApiVersion? Turns out that there is something that looks a fair bit like a bug in SAM, let's call it a conundrum. The issue is that when creating ApiGateway, in addition to the main stage (prod by default), sam also creates a mysterious stage called “Stage”. If you specify the OpenApiVersion field in the resource, this will not happen. This only works with newly created resources, this approach can not fix already corrupted resources.

Integration of lambda functions with the API

Now the resource created by ApiGateway can be connected to the lambda in the sam config. At the same time, we will make more correct REST urls:

```
GetProjectByIdFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: src/handlers
    Handler: get_project_by_id.lambdaHandler
    Runtime: nodejs14.x
    Events:
      ApiEvent:
        Type: Api
        Properties:
          Path: /projects/{projectId}
          Method: get
          RestApiId: !Ref BugTrackerApi
```

We have added the `projectId` variable to the path, which will automatically be included in the event object - http request. The `RestApiId` parameter contains the API resource ID. Similarly, we will correct other lambda functions so that they work with our ApiGateway.

The ApiGateway is configured. All that remains is to correct the code of the lambda functions. In the code of the handler of our lambda function the event parameter appears as the first argument. The format of this parameter differs depending on the data source.

For example, if we make a regular GET `/projects/1` request, then the event parameter will have the following fields (the pertinent subset of data is shown):

```
{
  "pathParameters": {
    "projectId": "1"
  },
  "httpMethod": "GET",
  ...
}
```

Example POST `/projects/1` request:

```
{
  "pathParameters": {
    "projectId": "1"
  },
  "httpMethod": "POST",
  "body": "{\"message\": \"hello world\"}",
  ...
}
```

Now it's clear where to get the identifier and query parameters from if I need them:

```
exports.lambdaHandler = async (event, context) => {
  const projectId = event.pathParameters.projectId;
  try {
    //some code here
    const [rows, fields] = await connection.execute("SELECT * FROM
`projects` where `id` = ?", [projectId]);
```

What about the response? In a lambda, you can return a response in absolutely any format, but not everything is suitable for ApiGateway. The service supports a response from lambda functions in the following format:

```
{
  "isBase64Encoded": false,
  "statusCode": 200,
  "headers": { "Content-Type": "application/json", ... },
  "multiValueHeaders": { "headerName": ["headerValue", "headerValue2",
...], ... },
  "body": "{\"status\": \"ok\"}"
}
```

The most important fields are statusCode and body. The first ApiGateway translates to http code. And the body content becomes the response that the client will see. Thus, if you want to return a cleanly formatted json in the response, then you need to pass in the body a string representing this json. isBase64Encoded is used when base64 encoded binary data is to be returned.

The updated function with support for the “correct answer” is as follows:

```
const [rows, fields] = await connection.execute("SELECT * FROM `projects` where `id` = ?",
[projectId]);
var response = {
  statusCode: 404,
  body: {
    msg: "entity not found"
  }
};
if(rows.length == 1) {
  return {
    statusCode: 200,
    body: JSON.stringify(rows[0])
  };
} else {
  throw "Entity not found";
}
```

The lambda function may return errors. If you make the handler in an asynchronous style (with async and promise, like we have done), then any unhandled exception will result in an error. Errors in the response from the lambda look like this:

```
{
  "errorType": "TypeError",
  "errorMessage": "Cannot read property 'projectId' of undefined",
  "trace": [
    "TypeError: Cannot read property 'projectId' of undefined",
    "    at Runtime.exports.lambdaHandler [as handler]
(/var/task/get_project_by_id.js:7:44)",
    "    at Runtime.handleOnce (/var/runtime/Runtime.js:66:25)"
  ]
}
```

The `errorMessage` field contains the error text. The `errorType` field is the error type, it can be a string if I pass a string as the first argument to the callback function. And the `trace` field contains the error stack trace, if any.

As you can see, this format is not compatible with ApiGateway. Any such errors in the lambda are treated by ApiGateway as internal errors. In general, if the lambda returns something incompatible in format, then we will see a 502 error in the response. To prevent such unpleasant situations from happening, all errors in the function must be caught and wrapped in the appropriate structure. This will allow you to set the necessary headers.

```
try {
  //db query here
  var response = {
    statusCode: 404,
    body: JSON.stringify({
      msg: "entity not found"
    })
  };
  if(rows.length == 1) {
    response.statusCode = 200;
    response.body = JSON.stringify(rows[0]);
  }
  return response;
} catch (err) {
  //exception
  console.log("Cannot retrieve project by id", err);
  return {
    statusCode: 500,
    body: JSON.stringify({
      msg: err.message
    })
  }
}
```


We can run the lambda function locally, which speeds up development and reduces the number of unforced errors. Moreover, we can also run a lambda function locally, complete with an http server that will act as an ApiGateway. As previously, we will describe 2 launch options: using the console command and the AWS Toolkit plugin for VSCode.

Option one:

The first way is to run the code locally using the sam cli. In this mode, the debugger is not connected and a separate web server is launched, which will work until it is explicitly stopped. This server supports hot re-loading, all changes in the code are immediately applied to the running application without restarting it. All server state is saved. The run command looks like this:

```
serverless-bugtracker> sam local start api ./env.json
```

We must not forget about the file with environment variables. This is the file I created to run the lambda function locally in the previous part. The same logic works for running the API. Therefore, this file can be used in both cases.

This mode is more for local development, debugging here is possible albeit painful. To do this, you need a third-party debugger and will need to run the lambda in a special debugging mode ... In our opinion it is more convenient to debug using the second method, which was actually created for this.

Option two:

The second way is to run the application in debug mode via the VSCode plugin. The API debug setting differs from the similar lambda function setting: `./vscode/launch.json`

```
{
  "type": "aws-sam",
  "request": "direct-invoke",
  "name": "API serverless-bugtracker:GetProjectByIdFunction",
  "invokeTarget": {
    "target": "api",
    "templatePath": "${workspaceFolder}/template.yaml",
    "logicalId": "GetProjectByIdFunction",
  },
  "api": {
    "path": "/projects/1",
    "httpMethod": "get",
    "payload": {
      "json": {}
    }
  },
  "sam": {
    "localArguments": ["--env-vars", "D:\\serverless-bugtracker\\env.json"]
  }
}
```

As you can see from the code, when starting the API we explicitly specify the URL for the request and the data that we want to send. We can still run the lambda function separately from everything in debug mode, we just have to update the payload section because the event format is now different from what was sent before.

For a lambda function it was possible to immediately determine the environment settings directly in the configuration, but for the API these settings have to be passed through a file. In the json section (as in the case of the lambda function), data is passed for POST / PUT requests. In the logs, you can see how the lambda function starts, ApiGateway answers. Under the hood, the plugin calls the same command that I described in the first option `sam local start-api`.

True, this command is launched from the bowels of VSCode, which is why you need to specify the full path to the file with variables. In this mode, VSCode immediately connects its debugger. After processing the request, the web server and the debugger are turned off. If you want to test the code with different parameters, you will have to run the application in debug mode each time.

It's time to deploy code to the cloud. In the AWS Console, you can find the URL where ApiGateway is available.

Integration with GitHub

To complete the necessary functionality, we just need to do the integration with GitHub in the `get projects` function. You will need to generate a special token in order to use the github api. As in the case of passwords from RDS, this value must be put into the SSM. Working with github-token is similar to working with a database password:

```
const { Octokit } = require("@octokit/rest");
//other params..
const githubTokenPromise = getParameterFromSsm("/global/serverless-bugtracker/github-token");
const githubPromise = githubTokenPromise.then(token => {
  return new Octokit({
    auth: token,
    request: {
      agent: new https.Agent({ keepAlive: true })
    }
  });
});

exports.lambdaHandler = async (event, context) => {
  //connect to db, etc...
  const [rows, fields] = /* get project */

  const project = rows[0];
  const { data: prInfo } = await listPullRequests(project['github_owner'],
project['github_repo']);

  //parse pull requests and prepare response...
}

async function listPullRequests(projectOwner, projectRepo) {
  //fetch PRs here...
}
```

Now with each request for a project we return a list of its PR from the github. Similar to the first function we will configure the remaining functions to work with our ApiGateway: add a URL and configure VPC subnets. Pay attention to the fact that the names of the SSM parameters with a password and a github token are hard-coded in the function code. These settings are global for all environments. But if you look in `template.yaml`, then you can find a parameter there, which, by its meaning, can take on different values for different environments. This is a parameter with the name of the database. Obviously, for dev and QA environments, this name should be different.

First, I need an additional parameter - the name of the environment:

```
Parameters:
  Env:
    Description: Environment name
    Type: String
    Default: dev
```

We have two options for proceeding further.

The first is to make a template string into which the Env value can be substituted. This method is suitable for simple cases where values can be easily calculated based on other parameters:

```
Globals:
  Function:
    Environment:
      Variables:
        DB_NAME: !Sub
          '{{resolve:ssm:${Env}/serverless-bugtracker/db-name}}'
```

The second is to make a special enumeration of all possible values depending on the selected environment. This option is suitable for cases where the values are not computable from the parameters. For example, we want to have lambda functions with different amounts of RAM depending on the environment:

```
Mappings:
  LambdaSizes:
    dev:
      "memorySize": "128"
    qa:
      "memorySize": "128"
    stg:
      "memorySize": "512"

Globals:
  Function:
    MemorySize: !FindInMap [LambdaSizes, !Ref Env, memorySize]
```

The Mappings block describes simple key-value structures. In our case, the key is the environment and the value is an object whose memorySize field is of interest. It is acceptable to have multiple properties on this object. The FindInMap function reads a value by key and property.

Privacy settings

Cool, we have ourselves a full-fledged REST API.

But the generated API is public. If we execute the request from the browser, then we can see the result of the operation. Different applications may have different non-functional requirements that relate to the openness of the application and access control (from the use of Bearer tokens and integration with AWS WAF to the use of the Cognito service). But we do not plan to go into all these mechanisms. We are interested in a basic way to restrict access to our ApiGateway at the network level, that is, the ability to work and access ApiGateway only from our VPC.

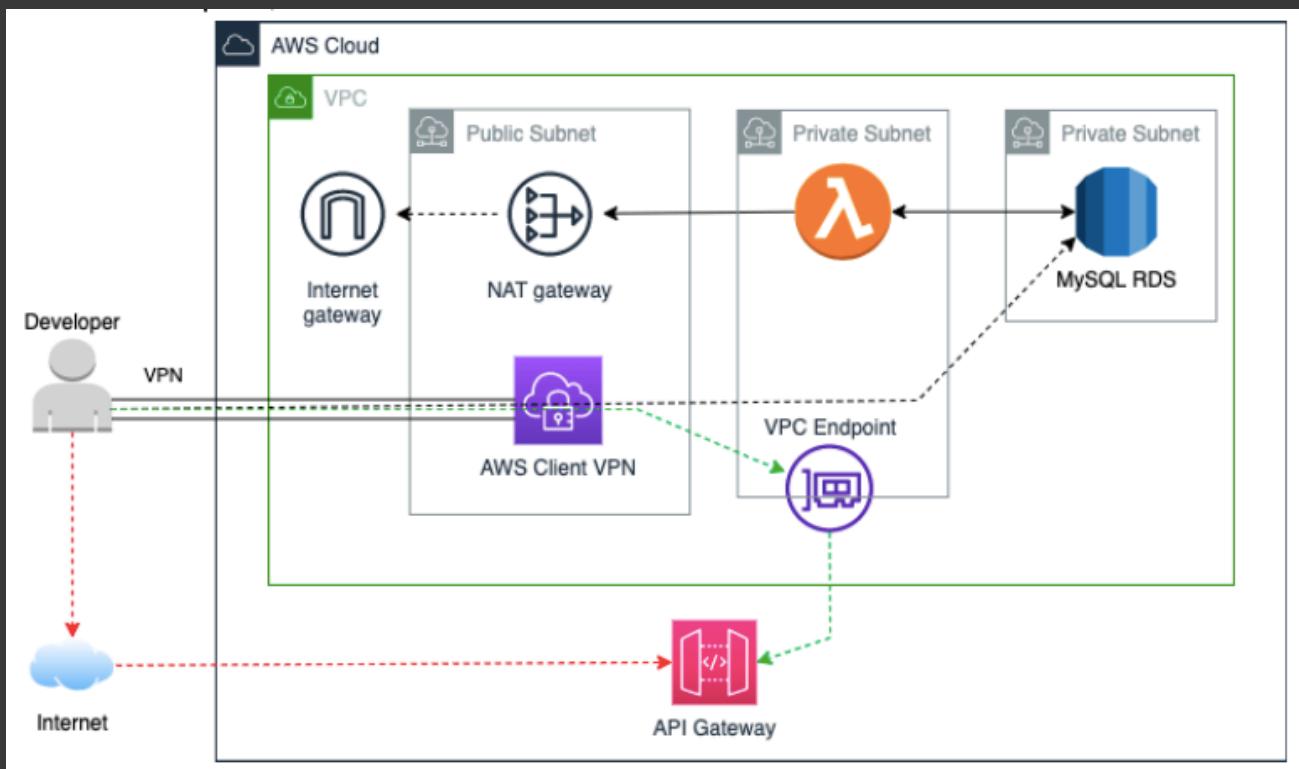
When creating a new API, the ApiGateway service provides the ability to specify the Endpoint Type parameter in the settings, which can be set to one of the following types:

edge-optimized - designed to provide global API access. To do this, CloudFront Points of Presence (CDN from AWS) are used - that is, requests to this type of ApiGateway will be directed to the geographically closest CloudFront point.

regional - assumes that access will be received from the region where the API was created.

private - an option that assumes access to the service only from the VPC.

Of these options we are interested in the Private type. To organize the Private access option in ApiGateway VPC Endpoints and Resource Policies are used. VPC Endpoints are a network interface that allows you to create a private entry point within a VPC to certain AWS services that do not support direct deployment to a VPC. For clarity and ease of understanding what VPC Endpoints are you can familiarize yourself with the diagram below - an example of organizing access to ApiGateway by default (red line) and through VPC Endpoint (green line). By default, traffic to ApiGateway goes through the public Internet, but if we use a VPC Endpoint, our traffic will not leave the AWS network:



To create a VPC Endpoint, we have added the necessary resource to the CF template of our VPC. This resource contains a number of interesting parameters, the most notable of which are:

PrivateDnsEnabled - this option transparently configures the DNS name for our private ApiGateway, that is, it will allow us to use the specified ApiGateway endpoint URL through the created VPC Endpoint.

ServiceName - allows us to specify which AWS service we need a VPC Endpoint for - you can get a complete list using the AWS CLI.

The following is an example VPC Endpoint resource code with dependent resources excluded:

./global-resources/vpc.yaml

```

ApiGwVpcEndpoint:
  Type: AWS::EC2::VPCEndpoint
  Properties:
    PolicyDocument:
      Version: 2012-10-17
      Statement:
        - Effect: Allow
          Principal: '*'
          Action: '*'
          Resource: '*'
    SubnetIds:
      - !Ref PrivateSubnet00
      - !Ref PrivateSubnet01
    VpcEndpointType: Interface
    PrivateDnsEnabled: true
    SecurityGroupIds:
      - !Ref ApiGwSecurityGroup
    ServiceName: !Sub 'com.amazonaws.${AWS::Region}.execute-api'
    VpcId: !Ref VPC

```

In the above Resource Policies rules for accessing ApiGateway are set. Below is the ApiGateway resource after the changes made to work in the Private version:

```

BugTrackerApi:
  Type: AWS::Serverless::Api
  Properties:
    ...
    EndpointConfiguration:
      Type: PRIVATE
      VPCEndpointIds:
        - !ImportValue
          'Fn::Sub': '${GlobalResourceStack}-VPCeApiGW'
    Auth:
      ResourcePolicy:
        IntrinsicVpceWhitelist:
          - !ImportValue
            'Fn::Sub': '${GlobalResourceStack}-VPCeApiGW'

```

In the added settings, you can see the configuration of the private Endpoint, where the type of Endpoint and the ID of the VPC Endpoint used are set, just below in the Auth section, the Resource Policy is configured, which we used for ApiGateway in our project.

To configure the Resource Policy in the description of the SAM resource for ApiGateway, there are already ready-made policy templates for different criteria for organizing access rules, in our case we took the `IntrinsicVpceWhitelist` template as a basis. This template defines an access policy that allows only a specific VPC Endpoint to interact with ApiGateway. That is, to configure Resource Policies, you can use one of the ready-made templates (in the form of allow / block lists for IP Range, AWS Account ID, VPC or VPC-endpoint), or completely independently specify the necessary rules in a form similar to IAM- policy(`CustomStatements`). In the case of self-configuring the Resource Policy, at first glance it may seem that these are the same IAM policies, but there are a number of differences here. The main difference is the object on which the policy is applied: in the case of IAM, it is either a user, group, or role; and in the case of Resource Policy, the object is the service itself. As indicated above, we have used one of the ready-made presets to configure the Resource Policy (the allowed list of VPC-endpoint is `IntrinsicVpceWhitelist`).

Thus, we can summarize the organization of secure access to ApiGateway: We created an Interface VPC Endpoint, configured a private Endpoint Type and set the necessary rules for access in the Resource Policy. As a result, we have private access to our ApiGateway. At the same time, it is important to understand that the Endpoint Type can be dynamically changed if necessary.

Another important point that you need to pay attention to: after any change in the Resource Policy configuration, you must definitely make a Deploy API, because changes to the Resource Policy are not automatically applied. This can be done in the UI using the corresponding Deploy API menu item or using the command:

```
aws apigateway create-deployment --rest-api-id xxxxxxxx --stage-name prod
--description 'Deployment to the prod stage'
```

The `rest-api-id` parameter is the ID of the resource created by ApiGateway. You can get it both in the AWS console of the service, and using the following CLI command:

```
aws apigateway get-rest-apis --query 'items[*].[name,id]'
```

This command will display information (name and search ID) on existing ApiGateways. An example command output can be seen below:

```
[
  [
    "myapi1",
    "bay55rt458"
  ],
  [
    "serverless-bugtracker-ch3",
    "q0cr55puj4"
  ]
]
```

In addition, after setting up private access, a reasonable question may arise: how can we get access to ApiGateway ourselves if we make it private and access to it is possible only from the VPC? In the third chapter we already deployed a client VPN from AWS, which allowed us to access RDS and it turns out that through it we can reach ApiGateway - no additional steps are required.

And the last organizational point worth focusing on: through one VPC Endpoint, you can access several ApiGateways, in connection with this, the CF description of the VPC Endpoint resource was added to the CF stack for global resources. This means that, for example, within the lower environments, each separate Application stack (for example, dev/stg/qa) will use one global VPC Endpoint created in the stack with global resources.

The graphic features the number '055' in a large, light gray, sans-serif font in the upper left. To the right, there are several colorful brush strokes: a long red one, a shorter yellow one, a small cyan circle, and a gray one, all arranged diagonally.

CHAPTER 05

WE PUT THE FINISHING TOUCHES ON OUR SERVERLESS DEVELOPER'S BAG OF TRICKS

In this final part, we will address remaining questions that may continue to pose difficulty.

A lot has already been studied to start collecting a must-have bag of tricks for serverless development. In this part we will put the finishing touches on our toolkit.

Based on the previous parts, the first and one of the most significant tools in our bag of tricks will be the SAM framework. This tool has allowed us to build all of the above and thanks to it our application is successfully launched both in the cloud and locally.

[Code can be found here.](#)

Implementation of CI/CD

The `sam build` and `sam deploy` commands are used to deploy the application. Each application should be able to automatically deploy a new one (on commit or on demand).

We want to implement a complete CI/CD pipeline for our Jenkins CI application. Not so long ago, AWS added a number of commands to the SAM framework to facilitate the task of organizing a full-fledged CI / CD approach - we are talking about the following commands:

```
sam pipeline bootstrap
sam pipeline init
```

These commands make it easy to create CI/CD pipeline files for a number of popular CI/CD tools.

The `sam pipeline bootstrap` command is used to prepare the resources necessary for CI / CD work in AWS. This is a preparatory command that will interactively request a series of data (number of stages, AWS accounts, etc.) and based on them, create the necessary resources in AWS. As a result of this command, a Cloud Formation stack will be deployed in the AWS account, which will create the resources necessary for a full-fledged pipeline: IAM roles / policies, S3 bucket.

In turn, the `sam pipeline init` command is used to directly create a configuration pipeline file in the format of the CI / CD service you are using. This command is based on the resources already created for deployment and depending on the CI / CD selected from the supported list - service - will generate the corresponding pipeline file.

The following services are currently supported:

- Jenkins
- GitLab CI/CD
- GitHub Actions
- Bitbucket Pipelines
- AWS CodePipeline

The result of this command will be a configuration pipeline file for the selected CI/CD tool. Below is a code snippet for Jenkins CI:

```
...
  stage('deploy-prod') {
    when {
      branch env.MAIN_BRANCH
    }
    agent {
      docker {
        image 'public.ecr.aws/sam/build-provided'
      }
    }
    steps {
      withAWS(
        credentials: env.PIPELINE_USER_CREDENTIAL_ID,
        region: env.PROD_REGION,
        role: env.PROD_PIPELINE_EXECUTION_ROLE,
        roleSessionName: 'prod-deployment') {
        sh '''
        cd global-resources/
        sam deploy --stack-name ${PROD_STACK_NAME} \
          --template template.yaml \
          --capabilities CAPABILITY_IAM \
          --region ${PROD_REGION} \
          --s3-bucket ${PROD_ARTIFACTS_BUCKET} \
          --no-fail-on-empty-changeset \
          --role-arn ${PROD_CLOUDFORMATION_EXECUTION_ROLE}
        ...
        }
      }
    }
  }
...

```

In this case, you can see that the created pipeline file uses the usual shell commands available when installing the SAM cli utilities. Nothing prevents you from creating a pipeline configuration file yourself using the `sam build` and `sam deploy` commands. In order for the above option to work correctly, you need a Jenkins plugin Pipeline: AWS Steps.

The command is launched in a docker instance, which contains all the components necessary for building and deploying the application. The code in the `withAWS` block is executed in the context of the specified AWS profile, with the appropriate permissions. The variable `MAIN_BRANCH`, as you might guess, defines the main branch of the repository, and in this case, the `deploy-prod` stage will only run if the main branch is used.

Thus, the `sam pipeline` command allows you to quickly deploy the first version of the CI / CD pipeline for a serverless application. This tool will come in handy for a quick start or for prototyping and will fit perfectly in our aforementioned toolkit.

CPU and disk size settings

We already have 4 lambda functions. When working with lambda functions, we do not work directly with the infrastructure. If these were EC2 instances, then we could choose the size of the disk, the type of instance (which would directly affect the performance of virtual cores and the amount of available RAM). But what do lambda functions have? What if our lambda needs to use threads efficiently, how can we make use of more advanced virtual cores?

In AWS, there is only one setting that can affect the performance of the lambda function environment - MemorySize (In the last part, we added a setting to allocate different amounts of memory for different environments). On the one hand, everything is greatly simplified, on the other hand, there is a direct and non-obvious dependence of the CPU on the amount of RAM.

At the time of this writing, the maximum amount of memory for a function is 10GB, the minimum is 128MB. For every 1769MB, AWS “adds” 1 vCPU to the environment. So we can have 1 to 6 virtual cores. The performance of the core also depends on the amount of memory, so, 1 environment core with 128 MB of memory is 2 times “weaker” than 1 environment core with 256 MB of memory.

As memory increases our lambda gets faster, but so does the cost of running it. Using the AWS Lambda Power Tuning tool we can find the sweet spot between performance and cost for our particular function. This utility runs a lambda function in different configurations, measures the time and cost of work and shows it on a graph.

If the application processes a lot of requests, then this tool will allow you to determine the optimal configuration of the lambda function, which would give acceptable performance at the lowest cost. This tool is definitely worth adding to your toolkit.

In addition to RAM, you can adjust the size of the /tmp directory that is available to the function. Configuration is done using the EphemeralStorage property. The minimum size (and default) is 512 MB, the maximum size is 10 GB.

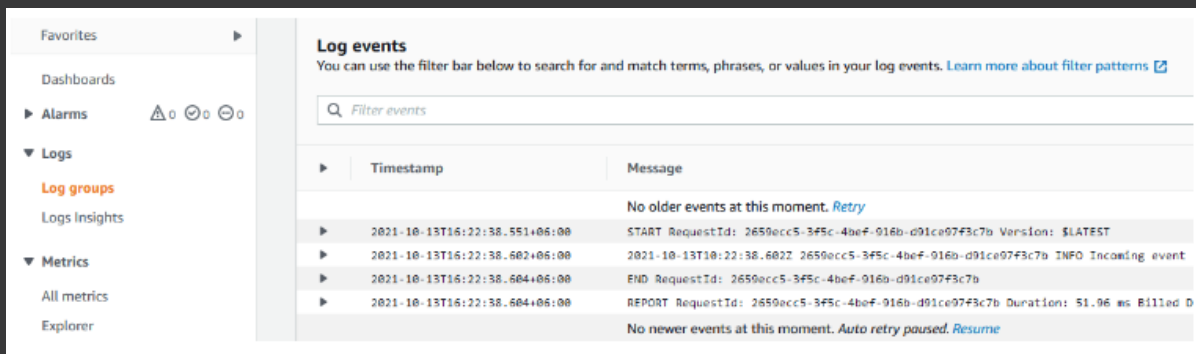
Log setup

Let's say an error occurs. How are we going to find out about it? Where can you find the reasons? In a classic app, we have logs. The logs are displayed in some service (for example, in Kibana), where you can filter the results, search by keywords.

It's time to deal with the logs in our serverless application. We have lambda functions that contain the business logic of the application, there is an ApiGateway that accepts requests. We need ways to work with the logs of these components.

Lambda function logs

Let's start with functions. AWS has a service Cloudwatch Logs, which is used to store and analyze application logs. For each lambda function, SAM creates a separate log group in this service. Each log group consists of several log streams. Each such thread refers to one instance of the lambda function. All data from stdout and stderr goes there. If we go to the AWS Console in the Cloudwatch service, then our logs there look like this:



The first thing to notice is that the logs contain entries starting with START, END, REPORT. These auxiliary records generated by AWS give an idea of when the lambda started executing the request, when it finished and the resulting metrics - execution time, billed time, memory used, allocated memory, etc. The second thing you should pay attention to is that each request has its own unique RequestId. Thanks to this we can find all the logs for a particular request and see the entire execution chain.

Viewing streams in a log group is inconvenient. Several dozen function instances can be created and killed in a day, and in this case it is very laborious to look for errors. Another tool in the same service comes to the rescue - Logs Insights. It is a tool that allows you to search for logs across all streams of different log groups with filtering and sorting. The query language, of course, will require some getting used to and you will have to look into the documentation more than once, but auto-completion makes the situation a little easier. We'll try to filter all logs with INFO level. It turns out that there is no field in the entry log by which we can filter such logs. The logs are not structured and are just a text string. The INFO string is in the @message field:

```
fields @timestamp, @message
| filter @message like 'INFO'
| sort @timestamp desc
| limit 100
```

Logs Insights is much more convenient to use than the ability to view “raw” log records. Yes, in terms of functionality it loses to Kibana and ElasticSearch. But the minimum set of functionality is enough to explore problems and find solutions. For those cases where a more advanced query language is required, you can instantiate ELK in the cloud and send logs from functions there.

Logs Insights or its equivalent should be in the arsenal of any serverless developer. If you only need logs for a specific function, then sam can also help. The sam logs command allows you to get the logs of a specific lambda function. It is possible to filter records by date and text:

```
D:\serverless-bugtracker> sam logs -n GetProjectByIdFunction --stack-name
serverless-bugtracker-ch5 -s '10days ago'
```

You can also set up continuous retrieval of logs:

```
D:\serverless-bugtracker> sam logs -n GetProjectByIdFunction --stack-name
serverless-bugtracker-ch5 --tail
```

Unstructured logs are inconvenient to work with. Filtering is particularly problematic. For example, I’d like to be able to search for all logs related to a particular projectId. In the current implementation, this will be difficult to do.

With Cloudwatch, structured logging can be implemented at the code level. If you pass an object to console.log(), then AWS will automatically split this object into parts and allow you to search through the fields of this object. If you do not want to make such band-aids yourself (wrap each line in an object with a field), then you can use special libraries for logs.

For example, there is winston-cloudwatch, but it uses the Cloudwatch API to publish logs from any external resources. You will have to explicitly set the group log name, configure the AWS account settings. This does not appear ideal in our case.

Ideally, you just need a wrapper around console.log (info, warn, error), which will help wrap text and tags into objects. The lambda-log library handles this role perfectly, for example, no account settings, just a convenient wrapper for outputting logs to the console, the output logs are structured. Also, this library allows you to work with the context of log entries:

```
const log = require('lambda-log');

exports.lambdaHandler = async (event, context) => {
  const projectId = event.pathParameters.projectId;
  log.options.tags.push(projectId);

  log.info("Incoming event", event);
  //code here...
};
```

The following object will be in the logs:

```
{
  "_logLevel": "info",
  "msg": "Incoming event",
  "body": null,
  "headers": {
    ...
  },
  "httpMethod": "GET",
  "isBase64Encoded": false,
  "multiValueHeaders": {
    ...
  },
  "multiValueQueryStringParameters": null,
  "path": "/projects/1",
  "pathParameters": {
    "projectId": "1"
  },
  "queryStringParameters": null,
  "requestContext": {
    ...
  },
  "resource": "/projects/{projectId}",
  "stageVariables": null,
  "version": "1.0",
  "_tags": ["1"]
}
```

Our project ID is now in the tags section, now we can search for this field in Cloudwatch. In the future, if a lot of data is stored in tags, then each value can be equipped with a prefix.

Logs in ApiGateway

Sometimes there are situations when requests do not reach functions. In such a situation, the logs configured in ApiGateway will help.

There are 2 types of logs in ApiGateway
 execution logs
 access logs

Execution Logs

In the first type of logs, you can find information about the request / response, about which authentication methods are used. These logs turn out to be very voluminous, for one request we can get 10 response lines. But it can be useful for deep investigation of some problems, because it gives information about how the request is processed in the service itself.

You can enable execution logs as follows:

```

BugTrackerApi:
  Type: AWS::Serverless::Api
  Properties:
    Name: bugtracker-api
    Description: my first serverless api
    StageName: dev
    OpenApiVersion: "3.0.3"
    MethodSettings:
      - LoggingLevel: INFO
        ResourcePath: '/*' # allows for logging on any resource
        DataTraceEnabled: false #true if request/response needs
        HttpMethod: '*' # allows for logging on any method
  
```

AWS for each ApiGateway automatically creates a separate log group for execution logs. An example of log for one request:

```

2021-10-13T18:04:36.471+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) Extended Request Id: HJS10FmioAMF3hg-
2021-10-13T18:04:36.471+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) Verifying Usage Plan for request: e1425e3a-6b83-488a-a9e5-45256367f196. API Key: API Stage: vml...
2021-10-13T18:04:36.474+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) API Key authorized because method 'GET /projects/{projectId}' does not require API Key. Request...
2021-10-13T18:04:36.474+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) Usage Plan check succeeded for API Key and API Stage vmipnq141b/dev
2021-10-13T18:04:36.474+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) Starting execution for request: e1425e3a-6b83-488a-a9e5-45256367f196
2021-10-13T18:04:36.474+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) HTTP Method: GET, Resource Path: /projects/123
2021-10-13T18:04:36.750+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) Successfully completed execution
2021-10-13T18:04:36.750+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) Method completed with status: 200
2021-10-13T18:04:36.750+06:00 (e1425e3a-6b83-488a-a9e5-45256367f196) AWS Integration Endpoint RequestId : 546180eb-aa55-4d51-bb9b-e028b117bfd7
  
```

Just one line contains information about the request, response status, and other additional parameters.

We will set up access logs the format of which AWS allows you to customize.

Access Logs

In addition to the format (Common Log Format, CSV, XML, JSON), you can customize the list of fields to display. The context of the event log contains a large amount of data, there are several dozen different fields that can be displayed in the logs. We'll stick with the standard ones:

```
BugTrackerApi:
  Type: AWS::Serverless::Api
  Properties:
    ...
    AccessLogSetting:
      DestinationArn: !GetAtt AccessLogGroup.Arn
      Format:
        '{"requestTime":"$context.requestTime","requestId":"$context.requestId","httpMethod":"$context.httpMethod","path":"$context.path","resourcePath":"$context.resourcePath","status":$context.status,"responseLatency":$context.responseLatency}'

AccessLogGroup:
  Type: AWS::Logs::LogGroup
  Properties:
    RetentionInDays: 14
```

Now we can update the rest of the lambda functions: add logs, integrate with the new event format.

Tracing

A finished serverless application usually consists of many features. These functions call other services, other functions. The resulting chain of calls from receiving a request to returning a result can be quite complex. In such a distributed environment, in addition to logs, tools are needed to help evaluate application performance and detect paths that lead to errors. Of course, all this information can be obtained from the logs, but the convenience of using this data leaves much to be desired, because you will have to build a chain of calls yourself each time in your head. A service for tracing such as AWS X-Ray comes to the rescue.

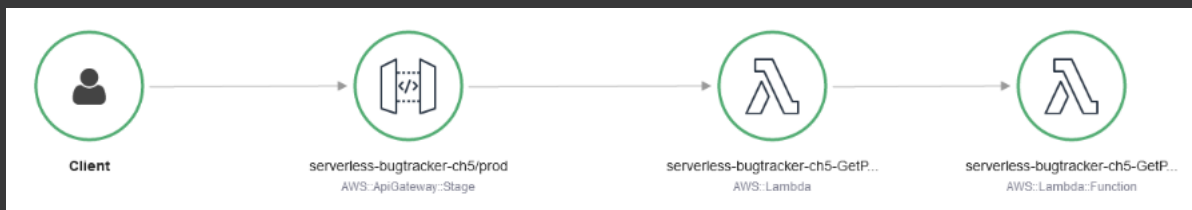
AWS X-Ray allows you to build service maps, track the execution logic for incoming requests, and show errors and failures in the application. This service marks some requests with a special Traceld. If other services are called during the processing of a request, then the generated requests are also marked with this identifier. Thus, it allows you to build a chain of calls, track the execution of a specific request, and profile the application.

In order to start using this service, you need to enable x-ray tracing in ApiGateway and lambda functions:

```
Resources:
  BugTrackerApi:
    Type: AWS::Serverless::Api
    Properties:
      Description: my first serverless api
      StageName: prod
      OpenApiVersion: "3.0.3"
      TracingEnabled: true
      ...

  GetProjectByIdFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: src/handlers
      Handler: get_project_by_id.lambdaHandler
      Runtime: nodejs14.x
      Tracing: Active
      ...
```

Now we will make a couple of calls to our API and go to the AWS Console in the XRay section. AWS produces this output trace:



In addition to the map, XRay provides information about the duration of steps. My map is very simple, you can only see the initialization and operation time of the lambda function:

Name	Res.	Duration	Status	0.0ms	200ms	400ms	600ms	800ms	1.0s	1.2s	1.4s	1.6s	1.8s	2.0s
▼ serverless-bugtracker-ch5/prod AWS: ApiGateway: Stage														
serverless-bugtracker-ch5/prod	200	2.1 sec	✓	[Progress bar]										
Lambda	200	2.0 sec	✓	[Progress bar]										
▼ serverless-bugtracker-ch5-GetProjectByIdFunction-fKXQwEajjnYI AWS: Lambda														
serverless-bugtracker-ch5-GetProjectByIdFunction	200	2.0 sec	✓	[Progress bar]										
▼ serverless-bugtracker-ch5-GetProjectByIdFunction-fKXQwEajjnYI AWS: Lambda: Function														
serverless-bugtracker-ch5-GetProjectByIdFunction	-	988 ms	✓	[Progress bar]										
Initialization	-	750 ms	✓	[Progress bar]										
Invocation	-	987 ms	✓	[Progress bar]										
Overhead	-	0.9 ms	✓	[Progress bar]										

Unfortunately, we don't see the database or github but they are necessary for more detailed profiling. Let's correct this problem. To add these services, you will need to use `aws-xray-sdk` in your function code. To profile the work with the database:

```
const captureMySQL = require('aws-xray-sdk-mysql');
const mysql = captureMySQL(require('mysql2/promise'));
```

After these changes, XRay will automatically receive information about queries in MySQL. If desired, you can add the SQL query itself to the context.

The Github API is not a standard service for the sdk, so these requests will not show up in the map or trace chains. To improve tracing for the GitHub API, we'll take advantage of the fact that XRay can track http/https requests.

```
AWSXRay.captureHTTPsGlobal(require('https'));
```

Now requests to the GitHub API will appear on the map.

Another useful feature in the XRay sdk is creating custom segments. They are not displayed on the service map, but the duration is considered for them and with this you can do profiling of important parts of the code. We'll make a segment that includes a GitHub API request and processes it:

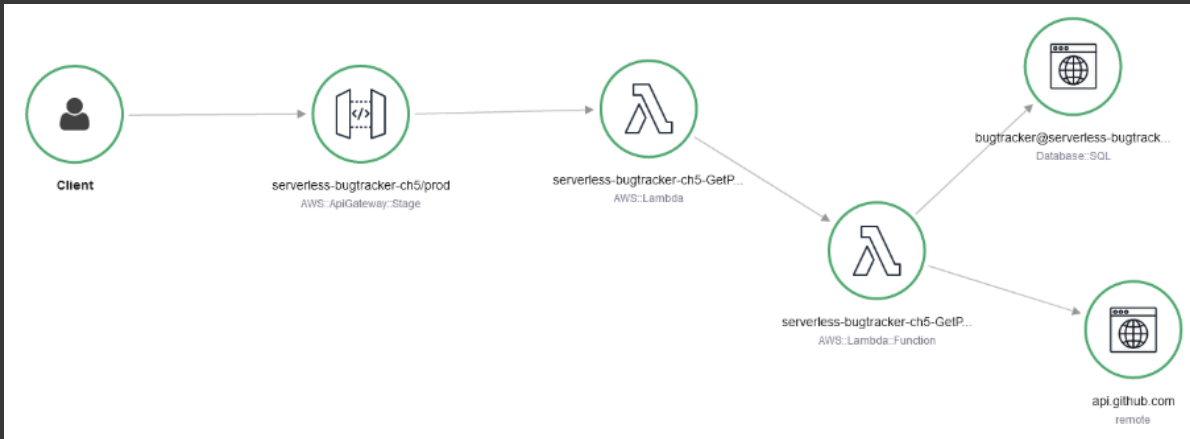
```
var AWSXRay = require('aws-xray-sdk-core');
AWSXRay.captureHTTPsGlobal(require('https'));

...

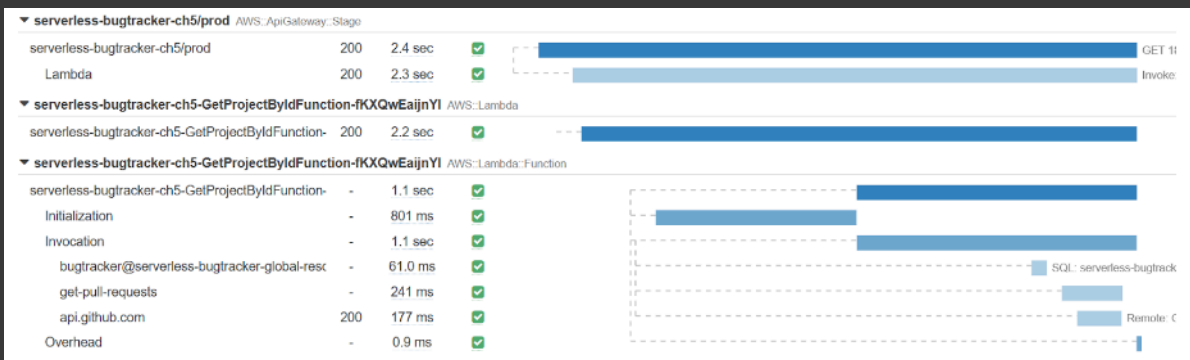
const xraySegment = AWSXRay.getSegment();
var subsegment = xraySegment.addNewSubsegment('get-pull-requests');
const { data: prInfo } = await listPullRequests(owner, project['github_repo']);
subsegment.close();
```

SSM can be integrated with sdk, but it is not necessary. This is because the request in SSM occurs once during the lifetime of the instance at the time of initialization. Tracing works only when the event handling phase occurs.

After all the changes, the call map looks like this:



And here is the updated list of segments:



As you can see, the tool is useful and necessary in distributed systems. But there is a fly in the ointment. Now my code doesn't work locally. A real lambda function is run locally and the XRay context is not created as it should. Because of this, an error occurs and the lambda function does not start: 'Missing AWS Lambda trace data for X-Ray. Ensure Active Tracing is enabled and no subsegments are created outside the function handler'.

To run locally, you can hide these errors. Amazon provides a special variable, `AWS_SAM_LOCAL`, to define how to run locally. To ignore these errors, we will use a special mode in the Xray client:

```
if (process.env.AWS_SAM_LOCAL) {
  AWSXRay.setContextMissingStrategy("IGNORE_ERROR");
}
```

The deployed application will work with tracing, while tracing will be ignored in local mode.

Tracing services play an important role in monitoring and debugging distributed systems, and this role increases with the increase in the number of application components. Such a tool must be in a serverless developer's toolkit.

Limiting Lambda function concurrency

As mentioned earlier, for each event from ApiGateway, AWS will look for a function instance ready for processing. If there is no ready instance, then AWS will start creating it. Thus, if there are no ready-made instances and 10 events arrive at once, then 10 instances of lambda functions will be created. This is very convenient, the provider will be able to handle peak loads and scale automatically. But what will this do to our database?

It is obvious that such uncontrolled “reproduction” will eventually lead to a database failure or very large delays. A similar problem can be seen with any constrained resource, access to which is limited by some kind of quotas. What can be done here?

For each function, you can set the maximum number of instances that can run at the same time. We can set this parameter via global settings for all functions:

```
Globals:
  Function:
    ReservedConcurrentExecutions: 5
```

By introducing this limit, we can block the creation of new instances beyond this limit. Events that lack a handler will remain unhandled. In the case of API Gateway, the user will receive an error. This mechanism allows you to control the rate of event processing for applications. Whether it is correct to handle such loads in this way is another question, and it is solved differently in each individual case.

If we set `ReservedConcurrentExecutions` to 0, then our lambda will stop processing events altogether, because the provider will not be able to create new instances. This can be useful in situations where you need to disable some functionality without updating the entire application. For example, a bug in a function or a database problem, we can disable the lambda that can cause these problems.

Provisioning lambda function concurrency

In the first part we touched on the problem of cold start a bit. To reduce the cold start time, we chose node.js, since interpreted programming languages have less initialization time. But nevertheless, even with the use of a certain language, this problem may not disappear. The more logic in the initialization block, the longer the waiting time for the first response. What to do? What else can be done to reduce this time, or to minimize the impact of this cold start problem?

In the initialization phase, the code is downloaded and the environment is prepared. After the environment is ready, the actual initialization of the code takes place.

As developers we can influence how our code is initialized. Obviously, the use of “heavy” libraries and frameworks will increase this time. Therefore, if the application can do without any frameworks,

then it is better not to add them. For example, you can do just fine without Express.js when writing lambda functions. There is no need to add the entire aws-sdk to the dependencies if only the SSM client is needed from this library.

Similarly, as developers we also have control over the size of our codebase. Here the recommendation is very similar to the previous ones. Frameworks and libraries, in addition to loading time, also affect the size of the artifact. AWS-SDK is 70MB while aws-ssm client is only about 5MB (according to npm).

Another way to reduce the codebase is to use source compression tools like webpack. This option is the simplest and most efficient if no layers with dependencies are applied, and all libraries are inside the lambda function.

Arranging compression across layers will require more effort. Dependencies from layers can be used in many functions. You will have to fix the library methods used through the proxy modules. Unfortunately, this solution is very inconvenient and cumbersome.

All these methods will help reduce the cold start time. But it is difficult to completely overcome this phenomenon. Yes, we can optimize our dependencies, but they will still contribute something to the overall initialization time.

How can we resolve this issue?

AWS has the ability to keep a certain number of “live” function instances in the cloud. Thus, when a new event appears for processing, the function is immediately ready for work, it does not need time to initialize. This functionality in AWS for lambda functions is called Provisioned Concurrency. We pay money to have the cloud provider keep a number of features “warmed up”.

For these “warmed up” functions, the cold start problem becomes irrelevant. If the number of events exceeds the number of existing warmed features, normal AWS rules are enabled. New instances are created according to the usual rules (with cold start)

Setting the ProvisionedConcurrency for the lambda function is done in the SAM template:

```
Globals:
  Function:
    ProvisionedConcurrencyConfig:
      ProvisionedConcurrentExecutions: 5
```

In a sense, you can consider the possibility of maintaining already “warmed up” function instances as a silver bullet to combat cold start. This solution is especially advantageous in the case of small loads. For applications with a variable load however, this solution may come with additional costs.



CONCLUSION

The application is ready and it's time to take stock of the whole adventure. Developing a serverless application was fun and interesting. We touched on the basic set: api gateway, lambda functions, RDS, sam. During the development process, we tried to cover the same points that we often encounter when developing classic server applications: testing, debugging, deployment, and so on.

Our bag of tricks now consists of the following:

SAM framework for building and managing the application.

AWS XRay as a tracing solution.

Logs Insights, sam logs for working with logs.

lambda-log for creating structured logs.

Conclusion

In the preface, we wrote about the pros that are usually attributed to serverless development. In the process of working on the application, we can definitely confirm two of the four:
speed of development.
fast releases.

Often, at the start of projects, the first weeks are intensive work on creating environments, writing deployment scripts, and working out CI / CD. Quite a lot of work at this time falls on DevOps engineers. With serverless applications, this first step can be completed faster.

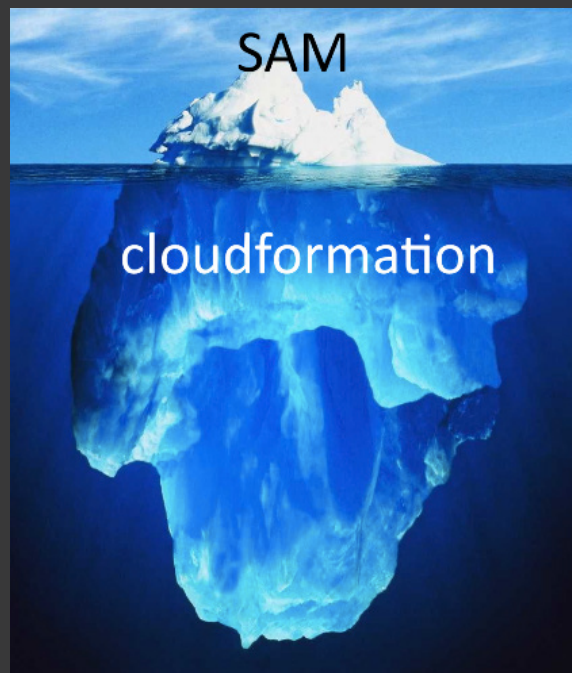
Of course, DevOps help will most likely be required to set up VPCs, VPNs, and integrate them with AWS services. As for the application itself, the developer is quite able to create deployment scripts, configure SSM, set build parameters, and organize dev/qa/stg/feature environments. SAM has proven to be a very handy deployment tool.

The releases themselves happen quickly, there is no need to update the entire application, instead you update only the functions that have changed.

“Flexible scaling” and “reducing the time or cost of administering applications” were not fully experienced. Our project did not have a real trial operation, despite this, we managed to test the operation of the application with variable load.

Performance tuning is greatly simplified, there are 3 parameters that can be controlled to increase / decrease the performance of the application: the amount of memory in the function itself, and the Reserved-Concurrency and ProvisionedConcurrency values.

But not everything can be perfect and sunny. There are moments that require a lot of attention.



If there were practically no problems with serverless components, then there were difficulties at the junction of SAM and CF. In CF, the number of resources and components is much larger and these resources are more difficult to configure. Therefore, it felt like more time was spent on CF resources and settings. Sometimes there was a lack of hints in the AWS documentation on how to do certain things.

Frankly, in the process of working on the application, we reworked the project structure several times. This is because there were no clear guidelines or recommendations from AWS on how best to arrange files in a project. Initially, all our js files were in one folder and everything seemed simple and clear until we looked into the assembly folder. We even took the time to learn how to use webpack to compress our sources. But then the understanding came that such a project structure is not quite suitable for a serverless application.

And so, this adventure has come to an end. We hope that it will be possible to make some kind of continuation.

There are many more interesting things in “serverless land”:
working with long running processes (Lambda functions are limited to 15 minutes).
building chains from lambda functions using step functions.
using other frameworks for deployment (serverless, aws cdk).
other cloud providers.



THANKS FOR READING!

For more info about the work we do, and the training we provide our teams, check out:

[Our Work](#)

[Blog](#)