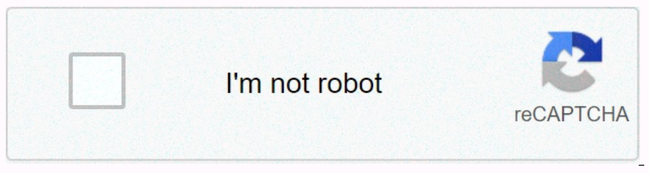

Continue

Serverless typescript template

On Amazon Web Services, the serverless computational par excellence is still Lambda, a must-have service when you talk about this paradigm. AWS Lambda allows you to use computational power without worrying about the management of the underlying servers, updates, software updates, or unexpected machine shutdowns. With the help of this paradigm, we can focus solely on our application. To date, AWS Lambda offers several runtime engines. Either way, it is possible to create your own if you need to use a programming language that is not yet supported by AWS. The big aspect is: it's up to us to choose the technology that best suits our needs when we write AWS Lambda Functions (FaaS): We can choose the programming language we're most familiar with so we can reach our goals faster. In this article, we will cover how to develop a serverless backend quickly and efficiently using TypeScript as the programming language. The program will utilize Express, a web application framework used to develop web services for Node.js. Node.js is the runtime environment – already supported by AWS – where Our TypeScript code, compiled in JavaScript, will run on, on Lambda. After the analysis, the proposed solution will be distributed on our AWS account. This project can be consulted and downloaded from GitHub! AWS services involved Let's start by proposing the architectural scheme with our serverless application. The code will be released on an AWS Lambda function, whose invocation will only be possible through an API Gateway. Obviously, a data source can't be missing in a bakend application. As we aim to take full advantage of the serverless paradigm, Aurora Serverless will be involved as a relational database to store and retrieve our data. In summary, we will use the following cloud services: AWS LambdaAPI GatewayAurora Serverless – Postgres EngineCloudFormation Diving into infrastructure management: the Serverless framework. Today, several tools are available for DevOps engineers to effectively manage infrastructures, such as AWS CLI, the console, or one of the frameworks available online, such as Troposphere, Terraform, or AWS SAM. As explained above, our goal is to build a backend program quickly: that's why we chose the Serverless framework. Serverless is a framework written in node.js so that we can manage the lifecycle of our serverless applications. It supports multiple cloud providers and features. In the case of AWS, Serverless will allow us to create and manage the resources we need in our account using the Cloudformation stack. Requirements Before you start working on the project, the following requirements must be met: Since we will write an application in TypeScript, node.js be the runtime environment on which we will run our code, compiled in JavaScript. To deploy the Cloudformation stack on aws, necessary to have the serverless framework installed on the machine. To do this, run the npm install -g serverless command and configure the AWS CLI credentials correctly. Hands on! At this point we are ready to start: download the project from our GitHub repository and check the structure. Let's start with cloning the project: git clone Enter the blog-serverless-backend-nodejs folder and run the command npm install to install all the necessary dependencies. Here are some clarification: aws-sdk: Software Development Kit for using AWS services. In our case, I will use it to integrate with Aurora Serverless and to retrieve the login credentials from Secrets Manager.sequelize: widely used and supported ORM for Node.js.serverless: Framework for the creation and management of Serverless infrastructure.express: Minimal and flexible web application framework for Node.js which offers a number of features for web application development.tsoa: Framework is used to write controllers for self-generation of Express routes. Thanks to that, it is also possible to generate OpenAPI specifications valid for both versions 2.0 and 3.0. It also offers automatic control for validation of the input of our requests, without having to create and maintain boiler plate code. You can find all these (and other) utility dependencies in the package.json file. In addition, it contains commands necessary to run and test our APIs locally, as well as to release them to your AWS account. By opening the project on the IDE, you will notice a scaffold similar to the following: Let's dive deep into the application logics. We're going to describe a very simple usage style: simple REST APIs that offer CRUD features to manage books in a library. The logic lies under the book catalog. Here is defined tsoa Controller, the service carries both the business logic and the sequel models. In the app.ts file, we instead register the Express panes (automatically generated by tsoa from the controllers), and then we export a module to be used as a handler for our Lambda function. Everything is made possible through the sentence: module.exports.handler = sls(app) Through this command we are invoking a Serverless framework function capable of creating a wrapper around Express routes. In this way, processing our AWS Lambda function will then know which Express route the rest API calls are going to. Finally, we just have to analyze the serverless.yml file: service: express-serverless provider: name: aws runtime: nodejs12.x region: eu-west-1 stage: $ {env:NODE_ENV} environment: NODE_ENV: $ {av: NODE_ENV} jegamRoleStatements: - Effect: 'Allow' Action: - 'secretsmanager:GetSecretValue' Resource: - '*' package: exclude: - node_modules/** - postpone postponing postpone defer postponement defer postponement src /** layer: nodeModules: path: layer compatibleRun: - nodejs12.x ..... features: app: handler: dist/app.handler lag: - {Ref: NodeModulesLambdaLayer} events: - http: path: / api method: SOME cors: true - http:// path: /api/{proxy+} method: ALL cors: true vpc: securityGroupIds: - {Ref: LambdaSecurityGroup} subnetIds: - subnet-1 - subnet-2 - subnet-3 plugins: - serverless-offline Note that in this file all the essential parts of the infrastructure are defined, such as the cloud provider on which we will deploy our Lambda function, its trades (the module we exported to the app.ts file) and Lambda Layers to pin the feature to. With this configuration, in fact, we will create a Lambda Layer that contains all node modules. By doing this, we significantly reduce the size of our function and achieve performance benefits during execution. Let's move on: how is the Lambda feature invoked? Through the http events defined in serverless file! This makes it possible to create an API Gateway that it will be possible to call our feature. The resource with path /api/{proxy+} will be the one that will proxy the router. We have thus created a single resource on the API Gateway page that allows us to start all the REST APIs. Local testing Not everyone knows that it is possible to test APIs without necessarily having to drop them on AWS Lambda every time you make changes to the source code. That's the big advantage of Node.js called serverless-offline! It is a serverless plugin that emulates AWS Lambda and API Gateway on your machine so you can speed up development activities. However, before you can use it, you must locally create a Postgres database and run the Sequelize migrations: From the root of the project, we execute the docker-compose command -d And then: npm run migrate-db-local At this point we can test APIs. Run the npm run start command to compile our TypeScript code in JavaScript and mimic Lambda via serverless offline plugin. As soon as the command is executed, you should see the following output at the terminal: At this point we are ready to test the API with the tool we are most familiar with (Postman, Curl). For example, using the curl command we can execute the command from the terminal: curl Deploy Go ahead and take the last step: the program deploy. If you run the 10pm execution deployment command, start the release process. The first time the following output is displayed: On the AWS account, a Cloudformation stack is generated that contains the resources we need. By accessing the console, we will find the just-created Lambda feature among the list of Lambda features available Before testing the APIs, it is necessary to run the Sequelize transfers to create tables on the Aurora Serverless database. Make sure that the can connect to the aws database. To do this, it will be necessary to create a bastion machine on our AWS account and divert traffic from ours to the bastion. Use the sshuttle command: sshuttle --dns -r ubuntu@EC2_BASTION_IP YOUR_VPC_CIDR --ssh-cmd 'ssh -i YOUR_PEM_KEY' At this point, start the migration by running the following command: npm run migrate-dev When the transfers on Aurora Serverless are completed, testing the APIs through API Gateway is the only thing left to do. The API Gateway endpoint is already printed in the output from the npm run deploy-dev command. Let's try it: curl The release process of our serverless infrastructure on the AWS account is complete! Conclusion To conclude, in this article we discussed how to create a serverless application on AWS Lambda using Node.js as the runtime engine. We decided to write the proposed project with TypeScript to have the benefit of using a transpiled, widely supported and known language. The choice of using node.js as a runtime engine provides a very limited cold start time that JavaScript is interpreted directly by the underline engine. Moreover, thanks to the configuration used, the source code is separate from the dependencies - stored on Lambda Layer, instead - drastically reduce the size of our source and improve boot performance. Satisfied with this solution? ☺ If you are interested in this topic, continue reading in our serverless section or check out this resource on how to build a Node.js /TypeScript REST API with Express.js. See you for 15 days on #Proud2beCloud for a new article! Article!