**Configure Automation for Serverless Deployment**

Serverless deployments are a powerful way to manage and scale your applications without worrying about server provisioning and maintenance. In this article, I will guide you through the process of setting up and automating a serverless deployment on [AWS Lambda](https://hackernoon.com/tagged/aws?ref=hackernoon.com) using the Serverless Framework. An article will cover everything from defining your serverless configuration to setting up an API Gateway and handling CORS, as well as optimizing an AWS Lambda function by Webpack build.

1. **Provide a serverless.yml Configuration File**

The first step in automating a serverless deployment is to create serverless.yml configuration file. This file defines various aspects of a serverless service, such as the service name, provider details, plugins, and more. Below is an example of a serverless.yml file:

service: api-service-name

provider:

name: aws

region: eu-central-1

stage: prod

runtime: nodejs18.x

memorySize: 1024

ecr:

images:

api-image-name:

path: ./

environment:

NODE\_ENV: production

plugins:

- serverless-offline

custom:

serverless-offline:

useChildProcesses: **true**

configValidationMode: error

package:

excludeDevDependencies: **true**

individually: **true**

exclude:

- \_\_tests\_\_/**\*\***

- .gitignore

- package-lock.json

- .git/**\*\***

# ... (other exclusions)

functions:

main:

image:

name: api-image-name

command:

- dist/handler.handler

entryPoint:

- /lambda-entrypoint.sh

timeout: 25

memorySize: 512

events:

- http:

method: ANY

path: /{any+}

cors: **true**

This configuration file defines various settings for a service, such as the AWS region, runtime, memory size, and more.

1. **Initialise a Docker File for ECR Repository**

To deploy a serverless function, a Docker image should be created for it. Let's use the following Dockerfile as a starting point:

FROM --platform=linux/x86\_64 public.ecr.aws/lambda/nodejs:18

COPY . .

RUN npm install

RUN npm run build

CMD ["dist/handler.handler"]

This Dockerfile is tailored for AWS Lambda and ensures that your function is correctly packaged. package.json includes the several scripts that will be considered below.

{

"name": "application-name",

"version": "0.0.1",

"description": "API description",

"author": "John Doe, john@email.com",

"private": true,

"license": "UNLICENSED",

"scripts": {

"build": "NODE\_ENV=production nest build --webpack",

},

}

1. **Add Permissions for Lambda Function Role**

To ensure that a Lambda function has the necessary permissions, an IAM role with the required permissions should be defined. Here's an example of an IAM policy that allows Lambda to interact with networking resources:

{

"Effect": "Allow",

"Action": [

"ec2:CreateNetworkInterface",

"ec2:DescribeNetworkInterfaces",

"ec2:DeleteNetworkInterface"

],

"Resource": "\*"

}

This policy grants permissions for network interface operations, which may be necessary for a Lambda function.

4. **Automate Schema Processes with Resources in serverless.yml**

To automate schema deployment and definition, use the resources section in a serverless.yml:

resources:

Resources:

RequestModel:

Type: 'AWS::ApiGateway::Model'

Properties:

Name: RequestModel

RestApiId:

Ref: ApiGatewayRestApi

ContentType: application/json

Description: 'Request Model'

Schema: ${file(models/RequestSchema.json)}

ResponseModel:

Type: 'AWS::ApiGateway::Model'

Properties:

Name: ResponseModel

RestApiId:

Ref: ApiGatewayRestApi

ContentType: application/json

Description: 'Response Model'

Schema: ${file(models/ResponseSchema.json)}

This code automatically creates models for your API Gateway based on JSON schemas.

**Request Schema**

The Request Schema defines the structure and constraints that a client's request must adhere to when interacting with your API. It helps in validating and ensuring that the incoming data is in the expected format. Below is the Request Schema described in JSON format:

{

"type": "object",

"required": ["request"],

"properties": {

"request": {

"type": "string"

}

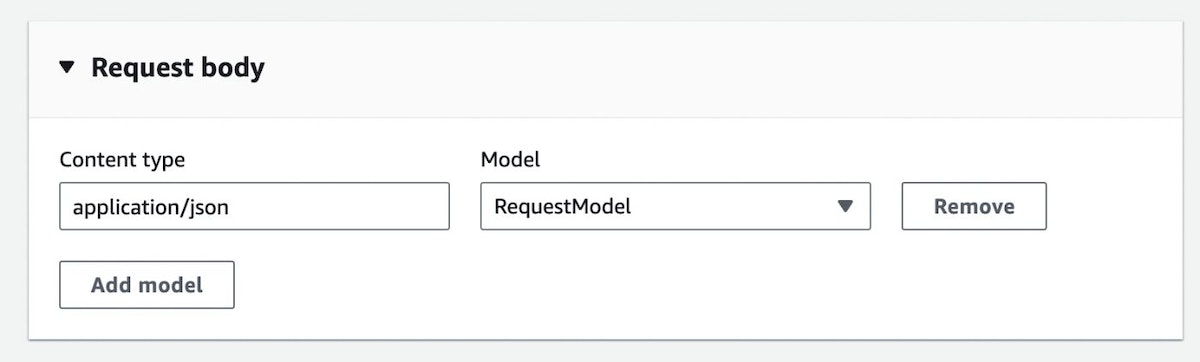
},

"title": "Request Schema"

}

**Key Components of the Request Schema:**

* **type**: This specifies that the top-level data structure is an object.
* **required**: It lists the properties that must be included in the request. In this case, the "request" property is required.
* **properties**: This section defines the structure of the request. It contains the "request" property, which is of type "string."
* **title**: The "title" field provides a human-readable name for the schema, which can be helpful for documentation and error reporting.



*Step 7. Application*

**Response Schema**

The Response Schema defines the structure and constraints that the API will adhere to when sending responses to clients. It ensures that the API's responses are consistent and in the expected format. Here is the Response Schema in JSON format:

{

"type": "object",

"required": ["response"],

"properties": {

"response": {

"type": "string"

}

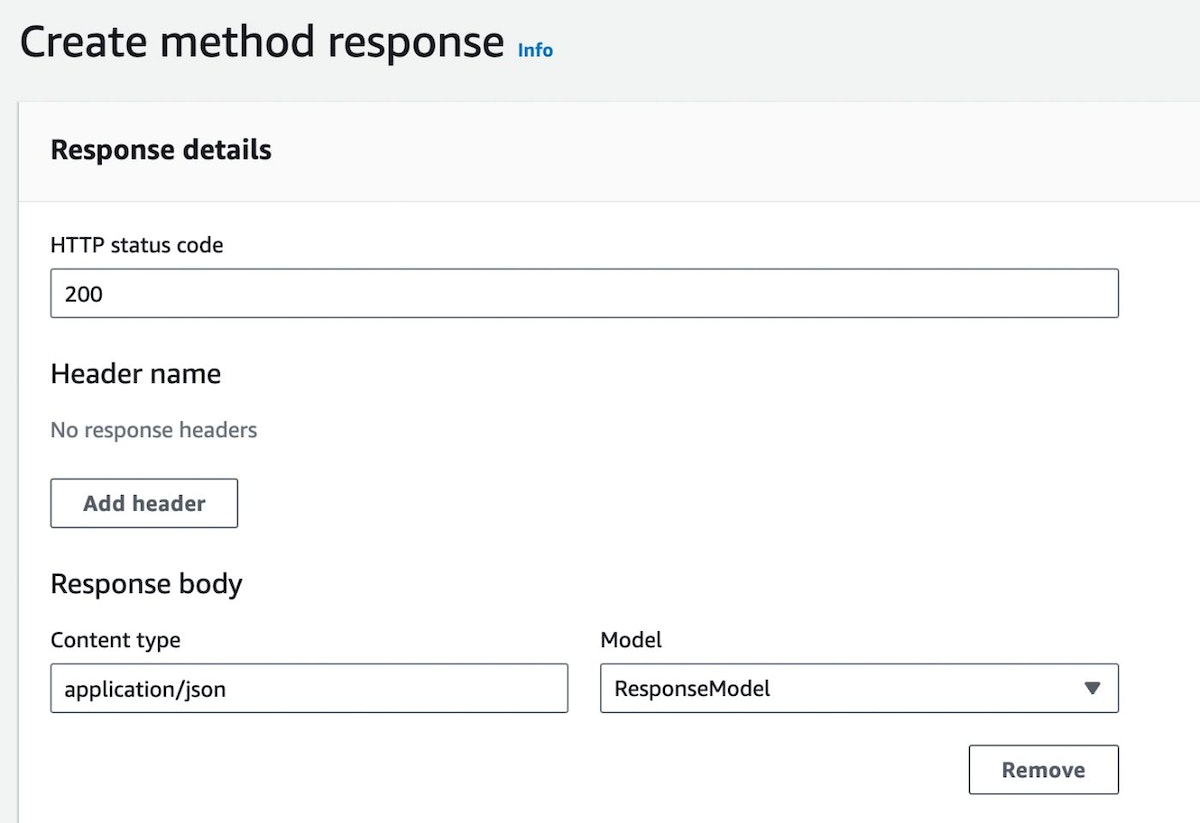
},

"title": "Response Schema"

}

**Key Components of the Response Schema:**

* **type**: Similar to the Request Schema, it specifies that the top-level data structure is an object.
* **required**: It lists the properties that must be included in the response. In this case, the "response" property is required.
* **properties**: This section defines the structure of the response. It contains the "response" property, which is also of type "string."
* **title**: As in the Request Schema, the "title" field provides a human-readable name for the schema, aiding in documentation and clarity.



*Step 7. Application*

**Schema Usage**

These schemas can be associated with specific API Gateway endpoints to validate incoming requests and outgoing responses. When a request is made to an endpoint, the API Gateway will use the Request Schema to validate the request payload. If the request doesn't conform to the schema, the API Gateway can reject it, ensuring data integrity.

Similarly, when the [API Gateway](https://hackernoon.com/how-to-build-multi-tenant-internal-services-in-aws-and-cdk-part-1-api-gateway-and-appsync?ref=hackernoon.com) sends a response, it uses the Response Schema to structure the response payload. This consistency in response format simplifies client-side code, as clients can expect a standardized response structure.

The Request and Response Schemas are essential tools in API development. They ensure that data is exchanged in a consistent and secure manner, making it easier to manage and maintain your serverless API.

1. **Set the serverless.yml Function Options**

Ensure that serverless.yml configuration specifies the necessary function options, including the image, timeout, memory size, and CORS settings:

provider:

ecr:

images:

api-image-name:

path: ./

functions:

main:

image:

name: api-image-name

command:

- dist/handler.handler

entryPoint:

- /lambda-entrypoint.sh

timeout: 25

memorySize: 512

events:

- http:

method: ANY

path: /{any+}

cors: **true**

These settings are crucial for the proper functioning of your Lambda function and API Gateway integration as an initial Lambda’s timeout is critical to run, and memory is valuable on application objectives.

1. **Deploy the Serverless Application**

To deploy a serverless application, use the following command:

sls deploy

This command will package and deploy a service, making it accessible on AWS Lambda.

1. **Open Lambda Function and Configure API Gateway**

After deployment, a Lambda function will be accessible through an API Gateway. Ensure that API Gateway has been modified for an API Gateway to handle the base path and methods an application requires for a Lambda interaction.

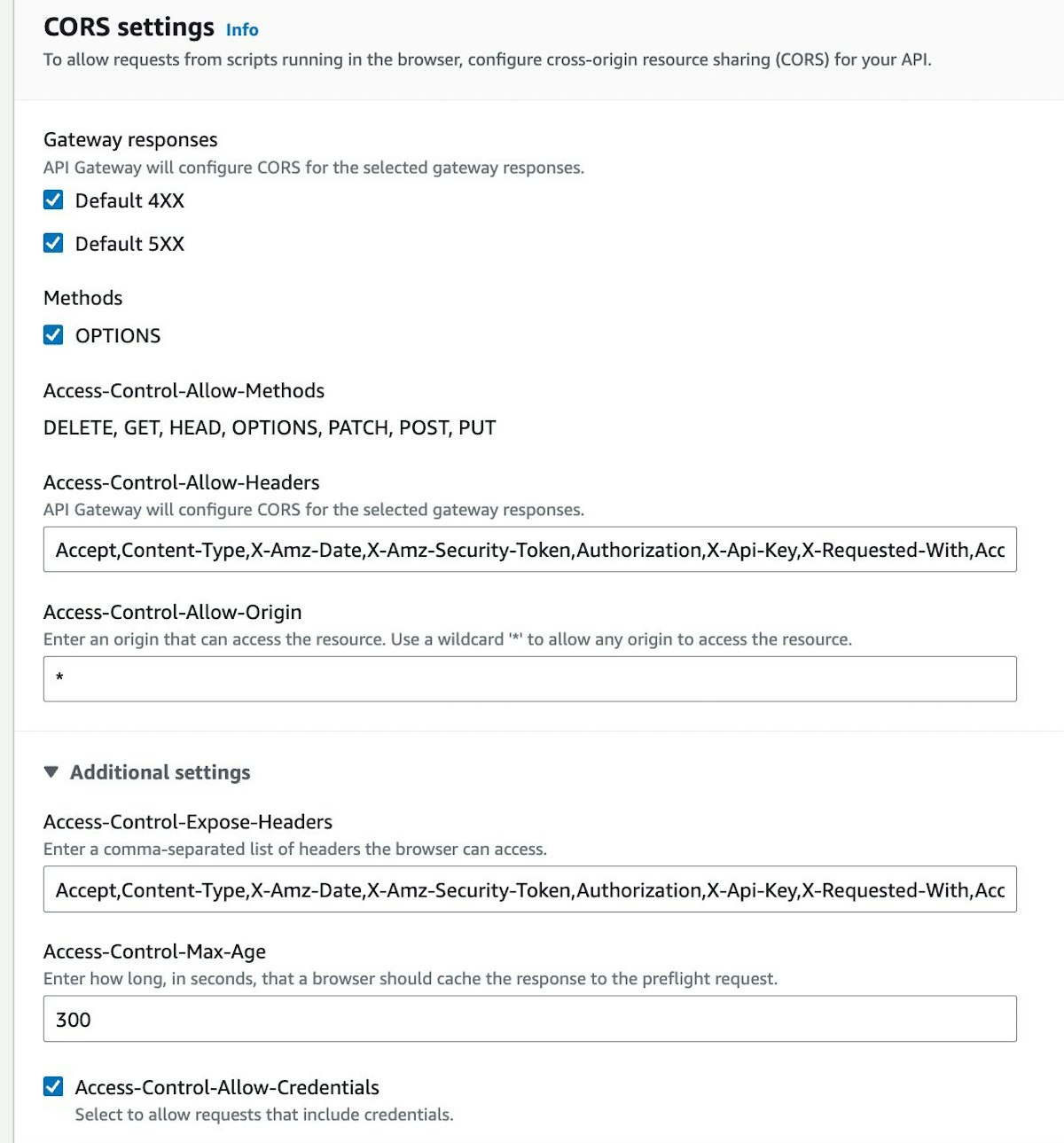
Additionally, define request and response schemas based on the JSON models you've deployed with a serverless.yml configuration.

1. **Set CORS Headers and Activate Credentials Checkbox**

Configure the API Gateway to allow Cross-Origin Resource Sharing (CORS) by setting appropriate headers. Common CORS headers to include are Accept, Content-Type, and others, depending on an application's requirements. Don't forget to activate the credentials checkbox if needed.

An example of the required CORS headers:

Accept,Content-Type,X-Amz-Date,X-Amz-Security-Token,Authorization,X-Api-Key,X-Requested-With,Accept,Access-Control-Allow-Credentials,Access-Control-Expose-Headers,Access-Control-Max-Age,Access-Control-Allow-Methods,Access-Control-Allow-Origin,Access-Control-Allow-Headers,Referer,User-Agent



**Integrating NestJS into a Serverless Deployment**

In this continuation of an article on configuring automation for serverless deployment, let’s delve into integrating a [NestJS application](https://hackernoon.com/creating-a-cli-app-with-nestjs-a-quick-and-easy-step-by-step-guide?ref=hackernoon.com) into the deployment pipeline. NestJS is a powerful framework for building scalable and maintainable serverless applications. Let’s provide detailed explanations of the key code parts, including the NestJS index file (main.ts) and the application module (app.module.ts).

1. **NestJS Index File (main.ts)**

Thus, to begin understanding the workings of the NestJS application, one should first delve into the main.ts file This file is the entry point for a NestJS application and includes important configurations for serverless deployment.

import { NestFactory } from '@nestjs/core';

import serverlessExpress from '@vendia/serverless-express';

import { Callback, Context, Handler } from 'aws-lambda';

import { ValidationPipe, VersioningType } from '@nestjs/common';

import { AppModule } from './app.module';

import { config as appConfig } from './config';

let server: Handler;

async function bootstrap(): Promise<Handler> {

const app = await NestFactory.create(AppModule);

// Apply global validation pipe

app.useGlobalPipes(new ValidationPipe());

// Set global prefix and enable API versioning

app.setGlobalPrefix(appConfig.app.baseUrl.prefix);

app.enableVersioning({

type: VersioningType.URI,

defaultVersion: appConfig.app.version.defaultVersion,

prefix: appConfig.app.version.prefix,

});

await app.init();

const expressApp = app.getHttpAdapter().getInstance();

return serverlessExpress({ app: expressApp });

}

export const handler: Handler = async (

event: any,

context: Context,

callback: Callback,

) => {

// Ensure Lambda doesn't wait for the event loop to be empty

context.callbackWaitsForEmptyEventLoop = false;

// Initialize the server on first execution

server = server ?? (await bootstrap());

// Execute the serverless function

return server(event, context, callback);

};

The bootstrap function is responsible for initializing the NestJS application and configuring it for serverless deployment. It sets up global middleware such as the validation pipe, API versioning, and prefixes. The handler function is the entry point application Lambda function. It ensures that Lambda doesn't wait for the event loop to be empty and initializes the server using the bootstrap function on the first execution.

1. **Application Module (app.module.ts)**

The app.module.ts file is the core of the NestJS application, where the modules, controllers, services, and other components of your application are defined. Below is an overview of your application module:

import { join } from 'path';

import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';

import { GraphQLModule } from '@nestjs/graphql';

import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';

import { APP\_GUARD } from '@nestjs/core';

import { ConfigModule, ConfigService } from '@nestjs/config';

import { S3 } from 'aws-sdk';

import { makeExecutableSchema } from '@graphql-tools/schema';

import { DynamicModule } from '@nestjs/common/interfaces';

import { GraphQLSchema } from 'graphql/index';

const graphqlModule = {

dev: GraphQLModule.forRoot<ApolloDriverConfig>({

driver: ApolloDriver,

autoSchemaFile: join(\_\_dirname, 'src/schema.gql'),

buildSchemaOptions: {

dateScalarMode: 'timestamp',

},

context: ({ request, reply }) => ({ request, reply }),

playground: true,

introspection: true,

}),

prod: (): DynamicModule => {

const s3 = new S3({

credentials: {

accessKeyId: process.env['AWS\_LLM\_ACCESS\_KEY\_ID'],

secretAccessKey: process.env['AWS\_LLM\_SECRET\_ACCESS\_KEY'],

},

});

const BUCKET\_NAME = 'api-bucket';

const SCHEMA\_FILE\_KEY = 'graphql/schema.gql'; // Adjust the key as needed

// Fetch the schema from S3 when the Lambda function starts

const schema: Promise<GraphQLSchema> = s3

.getObject({ Bucket: BUCKET\_NAME, Key: SCHEMA\_FILE\_KEY })

.promise()

.then((data) => data.Body.toString('utf-8'))

.then((schemaString) => makeExecutableSchema({ typeDefs: schemaString }));

return GraphQLModule.forRootAsync({

driver: ApolloDriver,

useFactory: async (configService: ConfigService) => {

const schemaFactory = await schema;

return {

schema: schemaFactory,

uploads: false, // Set to true if you need to handle file uploads

cache: 'bounded',

};

},

});

},

};

@Module({

imports: [

AuthModule,

CustomerModule,

// ... other modules

ConfigModule.forRoot({

envFilePath:

process.env.NODE\_ENV === 'development'

? ['.env.development.local']

: void 0,

isGlobal: true,

}),

process.env.NODE\_ENV === 'production'

? graphqlModule.prod()

: graphqlModule.dev,

],

controllers: [AppController],

providers: [

{

provide: APP\_GUARD,

useClass: RoleAuthGuard,

},

AuthService,

CustomerService,

AppService,

PrismaService,

LocalStrategy,

JwtStrategy,

RefreshStrategy,

],

exports: [AuthService],

})

export class AppModule implements NestModule {

configure(consumer: MiddlewareConsumer) {

consumer.apply(ResponseHeadersMiddleware).forRoutes('\*');

}

}

Application module details:

* In the imports array, you import and configure various modules, including AuthModule, CustomerModule, and others. These modules contain controllers, services, and other components that make up your application.
* The providers array includes providers such as AuthService, CustomerService, and strategies like LocalStrategy, JwtStrategy, and RefreshStrategy. These providers handle authentication and business logic.
* The configure method is part of the NestModule interface and is used to apply middleware. In this example, ResponseHeadersMiddleware is applied to all routes.
* graphqlModule code fetches the production schema from S3, converts it to a string, and then uses makeExecutableSchema to create a GraphQL schema.
  + Finally, it returns the configuration for GraphQLModule with the dynamically fetched schema for production.

**Optimizing NestJS Builds with Webpack**

In this article, I will explore in detail the Webpack configuration used to optimize the build process of a NestJS application for production. The provided Webpack configuration is intended for use with the NODE\_ENV=production nest build --webpack CLI command. It leverages Webpack's capabilities to bundle, minify, and prepare your NestJS application for efficient deployment.

Let's break down the Webpack configuration step by step, explaining the purpose of each section:

const path = require('path');

const TerserPlugin = require('terser-webpack-plugin');

const nodeExternals = require('webpack-node-externals');

const CopyWebpackPlugin = require('copy-webpack-plugin');

module.exports = (options, webpack) => {

return {

...options,

entry: ['./src/main.ts'],

target: 'node',

mode: 'production',

externals: [...options.externals, nodeExternals()],

output: {

libraryTarget: 'commonjs',

filename: 'handler.js',

path: path.join(\_\_dirname, 'dist'),

},

module: {

...options.module,

},

plugins: [

...options.plugins,

],

resolve: {

extensions: ['.ts', '.js', '.gql'],

alias: {

src: path.resolve(\_\_dirname, 'src'),

},

},

optimization: {

minimizer: [

new TerserPlugin({

terserOptions: {

keep\_classnames: true,

},

}),

],

},

};

};

1. **Imports:**

The configuration begins by importing the necessary Node.js modules and Webpack plugins:

* path: The path module is used to work with file and directory paths.
* TerserPlugin: This plugin is responsible for minifying JavaScript code.
* nodeExternals: It helps exclude Node.js core modules and node\_modules dependencies from the bundle.

1. **Module Export:**

The Webpack configuration is wrapped in a function that takes two parameters, options and webpack. However, in this specific configuration, only options is used. The function returns an object representing the Webpack configuration.

1. **Entry Point:**

* entry: ['./src/main.ts']: This specifies the entry point of your application, which is typically the main TypeScript file (main.ts) in the src directory.

1. **Target:**

* target: 'node': Indicates that the target environment is Node.js. This ensures that Webpack understands that it is bundling code for a Node.js runtime.

1. **Mode:**

* mode: 'production': Sets the Webpack mode to production, enabling optimizations such as minification and tree shaking.

1. **Externals:**

* externals: [...options.externals, nodeExternals()]: Tells Webpack to exclude external dependencies from the bundle. nodeExternals() ensures that Node.js core modules and node\_modules dependencies are not bundled.

1. **Output:**

* libraryTarget: 'commonjs': Specifies the type of module system used in Node.js.
* filename: 'handler.js': Sets the name of the output file to handler.js.
* path: path.join(\_\_dirname, 'dist'): Defines the output directory where the bundled code will be placed.

1. **Module:**

* ...options.module: Spreads the module configuration from options. This allows you to inherit module configurations from the NestJS application.

1. **Resolve:**

* extensions: ['.ts', '.js', '.gql']: Specifies the file extensions that Webpack should resolve when importing modules.
* alias: Defines an alias for the src directory, simplifying the import of modules with relative paths.

1. **Optimisation:**

The minimizer array contains an instance of the TerserPlugin. This plugin is responsible for minifying JavaScript code and the keep\_classnames: true option preserves class names during the minification process.

With this detailed explanation of the provided Webpack configuration, you can now understand how it optimizes the NestJS build for production. This configuration ensures that a NestJS application is bundled efficiently and ready for deployment as a serverless function, making it well-suited for production use.

**Conclusion**

In this comprehensive guide, the serverless deployment process for a NestJS application has been explored. It has covered various aspects, from configuring automation for serverless deployment to integrating a NestJS application into the deployment pipeline. Additionally, an article has delved into the details of a Webpack configuration optimized for NestJS builds.

**Let's summarise the key points:**

1. **Serverless Deployment Configuration:**

I’ve begun by setting up a serverless deployment using the serverless.yml configuration file. This configuration provides details about the service, runtime, memory allocation, and more. It also includes plugins for local development, package settings, and resources like API Gateway models.

1. **Docker Image for ECR Repo:**

To streamline the deployment process, I’ve created a Docker image for the Elastic Container Registry (ECR) repository. This image is based on the public.ecr.aws/lambda/nodejs:18 platform and includes necessary dependencies, building, and initialization steps.

1. **Lambda Function Permissions:**

I’ve added permissions for the Lambda function role to interact with [AWS resources](https://hackernoon.com/connecting-securely-to-private-aws-resources-by-using-ssh-tunnels-and-bastion-hosts-cei31i4?ref=hackernoon.com), like creating and managing network interfaces. These permissions are crucial for the function to operate correctly.

1. **Schema Management:**

I’ve highlighted the importance of automating schema processes using resources defined in the serverless.yml file. This includes request and response model definitions, which are essential for the API Gateway to validate incoming requests and responses.

1. **Serverless Function Options:**

The serverless.yml configuration also defines function options, including image settings, command, entry point, timeout, memory size, and event triggers. These settings dictate how the Lambda function operates and responds to incoming requests.

1. **NestJS Integration:**

I’ve continued by detailing the integration of a NestJS application into the serverless deployment process. This included the main.ts entry file, which sets up the NestJS application and handles Lambda function execution.

1. **Application Module:**

The app.module.ts file defines the core structure of the Nest.js application, including module imports, providers, and middleware configuration. I also covered the setup of GraphQL modules for development and production environments.

1. **Webpack Configuration:**

I’ve concluded the article by providing a Webpack configuration optimized for production builds of NestJS applications. This configuration bundles and minifies the code and prepares it for deployment.

This schematic encapsulates a robust, scalable, and efficient serverless architecture ideal for modern applications. It leverages AWS Lambda for computing, orchestrated by Amazon API Gateway for client interactions, and is bolstered by the resilience of Amazon RDS for relational data persistence. The inclusion of Amazon S3 provides scalable object storage, while Amazon ECR hosts container images for deployment. Amazon VPC ensures secure network isolation, and IAM roles enforce granular access control. NestJS underpins this architecture as the core framework, streamlining development and upkeep, while Prisma amplifies data management capabilities at the Lambda function tier. The schematic focuses on the overarching structure and may not depict the intricacies of NestJS and Prisma dependencies. Together, this architecture promises to deliver high performance, manageability, and cost-effectiveness while providing a streamlined development experience.

In summary, serverless deployment of a [NestJS application](https://hackernoon.com/using-a-nestjs-application-with-dynamodb-and-serverless-framework-on-aws?ref=hackernoon.com) involves careful configuration, resource management, and integration of various components. By following the steps and configurations described in this guide, you can build and deploy scalable, efficient, and maintainable serverless applications on AWS Lambda.