What I learned after a couple of weeks of using AWS IoT Greengrass

13 May 2022 - 10 min. read

AWS Greengrass | Internet of Things (IoT)

In the last few days, I started looking at AWS Greengrass, one of the many IoT services offered by AWS, which was upgraded to v2 last year.

In this article, I'd like to talk about what I've learned, and my general takes, hoping to help anyone who will approach this service for the first time just like I did.

So, let's start with...

What is AWS Greengrass?

AWS Greengrass is a service that allows us to bring the computing experience we are used to when we work with AWS to our IoT edge devices. It consists of an open-source runtime that we need to install on our devices which offers several features, such as:

- Local processing with AWS Lambda functions
- Containers support
- Integration with AWS IoT Core (and device shadows)
- Local messaging

Like most other AWS services, Greengrass was created to solve multiple problems in a managed manner without forcing us developers to reinvent the wheel every time. When we build IoT solutions, we need to worry about scaling and keeping our devices' firmware up to date, and this is what Greengrass is trying to help us do.

AWS IoT Greengrass makes it easy to deploy and manage device software on millions of devices remotely. We can organize our devices in groups and deploy and manage device software and configuration to a subset of devices or all devices. AWS IoT Greengrass gives us the ability to send over-the-air updates on the software that relies on our machines without worrying about breaking the runtime layer and forcing us to go to the device place to reset it manually. This has been done by decoupling the many components that we want to run on our devices from the Core software that is tasked to orchestrate our components and manage firmware updated jobs.

With Greengrass, we can perform Machine Learning Inference, data aggregation, and streaming to multiple AWS cloud services (such as S3 or Amazon Kinesis) directly from our devices, therefore allowing us to have a grasp of what data goes to the cloud so we can optimize analytics, computing, and storage costs.

Let's think of a use case

Imagine you are building an IoT solution that consists of multiple devices installed in a factory. These devices periodically gather data and send it as is to a cloud backend. This data is then processed and stored, allowing your clients to know if their machines are performing as they should, if some of the parts need maintenance, and configure custom events that must be notified. Now imagine this solution deployed to thousands of factories, hence millions of devices that every few minutes send a payload to your backend: the costs of your application would grow with the number of devices installed. Furthermore, data received from a factory has no connection with data received from another because every factory behaves differently.

Now, how can we optimize this infrastructure? With Greengrass, we could configure a machine that works as a processing centralizer: it gathers data, aggregates it, and sends a single payload periodically with the global status of the plant. Of course, devices might still have to send some information (such as alerts or events) as-is to our backend, but moving some of the data processing to the edge would drastically improve the costs of our infrastructure. We'd have far fewer resources to provision on the cloud side as they would not grow with the number of devices but with the number of plants.

Time to get our hands dirty

My approach to learning Greengrass has been the same as I have used for other services: I usually start with a quick overview of the official documentation, then I start building something directly from the AWS console, and when I'm facing an obstacle I do punctual investigation on the web.

My take on the Greengrass official documentation is that it's very extensive, but sometimes notions are somewhat scattered. I often found myself looking for ways to fix a problem that I thought wasn't documented. After minutes (if not hours), I realized that the solution was right below my eyes but in a different documentation section. Greengrass is made by many strictly tight concepts, so you should seriously dive into the docs before starting building your solution.

What I want to do here is give you a way to quickly start a project and relieve you of the hassle of trying to figure out how the features of this service work together.

Setup your core device

The core device is a machine with the Greengrass runtime installed. There are many ways to do this; the path I chose first is the automatic provisioning of Greengrass in a Docker container, as it is the most straightforward way to do this task, and it doesn't require much configuration. All we need to do is create a ".env" file with our core device configuration that looks like this:

```
GGC_ROOT_PATH=/greengrass/v2
AWS_REGION=eu-west-1
PROVISION=true
THING_NAME=P2BCGreengrassCore
THING_GROUP_NAME=P2BCGreengrassCoreGroup
TES_ROLE_NAME=P2BCGreengrassV2TokenExchangeRole
TES_ROLE_ALIAS_NAME=P2BCGreengrassCoreTokenExchangeRoleAlias
COMPONENT_DEFAULT_USER=ggc_user:ggc_group
```

This ".env" file can then be passed in the docker-compose.yaml file as the "env_file" value; If we set the PROVISION key to true, Greengrass will take care of creating all the resources required on AWS IoT to set up a new device. But to do this, we need to give Greengrass some AWS credentials in the form of an access key, secret access key, and session token. I wasn't keen on this solution as I don't want to do this operation for

every core device I will set up in the future, and, above all, I want to be in charge of creating the roles and permissions needed.

So I decided to manually create the IoT thing, role, role alias, group, and certificates. The certificates will be used to authenticate the newly created device on AWS. You can find on this page a well-done guide that explains how to provision these resources directly from the command line: https://docs.aws.amazon.com/greengrass/v2/developerguide/run-greengrass-docker-manual-provisioning.html. The ".env" file now looks like this:

```
GGC_ROOT_PATH=/greengrass/v2
AWS_REGION=eu-west-1
PROVISION=false
COMPONENT_DEFAULT_USER=ggc_user:ggc_group
INIT_CONFIG=/tmp/config/config.yaml
```

The config.yaml contains all the references needed by the core device to find the location of the certificates and the endpoints of IoT Core. My config file looks like this:

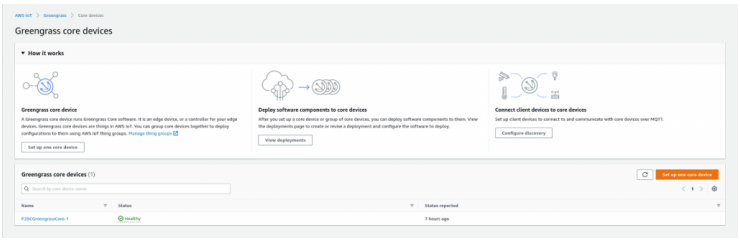
```
system:
  certificateFilePath: "/tmp/certs/device.pem.crt"
  privateKeyPath: "/tmp/certs/private.pem.key"
  rootCaPath: "/tmp/certs/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "P2BCGreengrassCore-1"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.5.3"
    configuration:
      awsRegion: "eu-west-1"
      iotRoleAlias: "P2BCGreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "xxxxxxxxxxxx-ats.iot.eu-west-1.amazonaws.com"
      iotCredEndpoint: "xxxxxxxxxxxx.credentials.iot.eu-west-1.amazon
aws.com"
```

The last thing we need to do is configure the docker-compose.yml file with the Docker image, the paths to the certificates and configurations directories, and where we want these volumes to be mounted. Here's my Docker compose file:

```
version: '3.7'

services:
  greengrass:
    init: true
    cap_add:
      - ALL
    build:
      context: .
    container_name: aws-iot-greengrass
    image: amazon/aws-iot-greengrass:latest
    volumes:
      - ./greengrass-v2-config:/tmp/config/:ro
      - ./greengrass-v2-certs:/tmp/certs:ro
    env_file: .env
    ports:
      - "8883:8883"
```

Now we can execute Docker compose, and our core device should go up and connect to AWS.



Deploy our first Lambda function

The next step is adding some capabilities to our newly created core device. In my case, I wanted to deploy a Lambda function as a Greengrass component. To do this, we need to create a new Lambda function in one of the runtimes accepted by Greengrass, which are:

- Python 3.8 (my choice)
- Python 3.7
- Python 2.7
- Java 8
- Node.js 12
- Node.js 10

The Lambda function I wanted to create is a simple function that interacts with the device's MQTT shadow, namely, performing a subscribe operation to one of the shadow's topics and publishing messages to another one. For this purpose, I included in the function code the awsiotsdk official library.

Remember to create a new version of your Lambda function after uploading your code because it is required for the next steps.

After our Lambda function has been created, we need to wrap it in a Greengrass component. The procedure is pretty straightforward: we select the function version we want to wrap and give it a name. Other configurations involve the component type and, optionally, some event sources.

There are two types of Lambda functions components that can be deployed to a Greengrass core device:

- Long-running (also called pinned) functions, best suited for tasks that need to be constantly running.
- On-demand functions, which start when they are invoked and stop when there are no tasks left to execute.

We can configure our Lambda function with event sources to be executed when a publish operation is performed on a local topic or an AWS IoT MQTT topic.

We can always come back to the function component configuration and modify its settings afterward if we need to make some tweaks.

The last thing we need to do is deploy our newly created component to our core device. To do so, visit the "Deployments" page of the Greengrass console and click on

the "Create" button. All deployments have a name, which is meant to describe what components are bundled inside of them, a target type that can either be a specific core thing or a thing group (this last one is useful if we want to create a single deployment for multiple devices) and a list of custom and public components. Since I needed to interact with the device's shadow, I had to add to the deployment the public component aws.greengrass.ShadowManager.

The Lambda function we have just created should appear in the list of custom components. Public components are provided by AWS and can be added to our deployment if we need to add specific capabilities to our core devices. We can configure each one after selecting which components we want to include in our deployment. The configuration of the components depends on the component type:

- The configuration of the ShadowManager public component has to contain which shadows we want to synchronize locally with AWS IoT. Below you can find the configuration I created:

```
{
        "strategy": {
                "type": "realTime"
        },
        "synchronize": {
                "shadowDocuments": [
                        {
                                "thingName": "P2BCGreengrassCore-1",
                                "classic": false,
                                "namedShadows": [
                                        "myShadow"
                                ]
                        }
                ]
        }
}
```

- The configuration of our Lambda function component contains the permissions we need to add to it. In our case, the function must have the capability to subscribe to a

shadow topic to receive notifications, and perform Get, Update, and Delete operations on the shadow.

```
{
        "accessControl": {
                "aws.greengrass.ShadowManager": {
                        "P2BCGreenGrassLambda:shadow:1": {
                                "policyDescription": "Allows access t
o shadows",
                                "operations": [
                                        "aws.greengrass#GetThingShado
w",
                                        "aws.greengrass#UpdateThingSh
adow",
                                        "aws.greengrass#DeleteThingSh
adow"
                                ],
                                "resources": [
"$aws/things/P2BCGreengrassCore-1/shadow/name/myShadow"
                                ]
                        }
                },
                "aws.greengrass.ipc.pubsub": {
                        "P2BCGreenGrassLambda:pubsub:1": {
                                "policyDescription": "Allows access t
o shadow pubsub topics",
                                "operations": [
                                        "aws.greengrass#SubscribeToTo
pic"
                                ],
                                "resources": [
"$aws/things/P2BCGreengrassCore-1/shadow/name/myShadow/update/delta"
                                ]
                        }
                }
```

```
        }
}
```

The last thing we can do to our deployment creation is deciding how the device should apply the update. What should the device do if the update operation fails? How much time does the device have to perform the update? Should we stop the deployment if some affected devices fail to update?

We are finally ready to deploy our components to the core devices we selected! Click on the "Deploy" button to start the update!

Debug issues

When I initially approached Greengrass, I had some issues configuring the deployment correctly (mostly the components' configuration), and I managed to debug the status of my core device by entering inside the Docker container and looking for the logs. Here's how I did it.

Look for the container ID by running this command:

```
docker ps
```

Copy the container ID and run this command:

```
docker exec -it  /bin/bash
```

You are now inside the Docker container. Now go to the logs directory that should be placed inside the */greengrass/v2/logs* directory. The log files that helped me the most in debugging my application were the greengrass.log file and the file with the same name as my Lambda function component (in my case *P2BCGreenGrassLambda.log*). The first contains the general status of the Greengrass runtime and the deployment status; the latter contains what I print to the stdout in my function code.

My considerations

AWS Greengrass is a powerful tool that relieves us from the burden of managing deployments of new versions of our software to the edge devices. It's not as intuitive and easy to use as many other services since many features are involved, all of which

are necessary to make things work. I spent quite a bit of time making a working proof of concept, but in the end, I can see its potential. The perfect use case for Greengrass is when we have many devices close to each other that are strictly related to one another and can be seen from the cloud as a single cluster of things. In this case, Greengrass could help us reduce the costs of our infrastructure (which is a critical point in IoT solutions), the network throughput, and the complexity of our code. Greengrass is still relatively young compared to other AWS services, and I'm sure more features and integrations will be added to it with time!

Mattia Costamagna

DevOps engineer and cloud-native developer @ beSharp. I love spending my free time reading novels and listening to 70s rock and blues music. Always in search of new technologies and frameworks to test and use. Craft beer is my fuel!
