

**FunctionGraph**

# Best Practices

**Issue**            01  
**Date**             2024-08-21



**Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2024. All rights reserved.**

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

## **Trademarks and Permissions**



HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

## **Notice**

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

---

# Contents

---

<b>1 Compressing Images.....</b>	<b>1</b>
1.1 Introduction.....	1
1.2 Preparation.....	2
1.3 Building a Program.....	3
1.4 Adding an Event Source.....	5
1.5 Compressing Images.....	6
<b>2 Watermarking Images.....</b>	<b>8</b>
2.1 Introduction.....	8
2.2 Preparation.....	8
2.3 Building a Program.....	10
2.4 Adding an Event Source.....	12
2.5 Watermarking Images.....	13
<b>3 Integrating with CTS to Analyze Login/Logout Security.....</b>	<b>15</b>
3.1 Introduction.....	15
3.2 Preparation.....	16
3.3 Building a Program.....	17
3.4 Adding an Event Source.....	17
3.5 Processing Operation Records.....	18
<b>4 Building an HTTP Function with Spring Boot.....</b>	<b>19</b>
<b>5 Creating a FunctionGraph Backend API That Uses a Custom Authorizer.....</b>	<b>23</b>
5.1 Introduction.....	23
5.2 Resource Planning.....	23
5.3 Building a Program.....	24
5.4 Adding an Event Source.....	29
5.5 Debugging and Calling the API.....	30
<b>6 Building an HTTP Function with Go.....</b>	<b>32</b>

# 1 Compressing Images

---

## 1.1 Introduction

The best practice for FunctionGraph guides you through image compressing based on a function.

### Scenarios

- Upload images to a specified Object Storage Service (OBS) bucket.
- Compress each uploaded image.
- Upload the processed images to another specified OBS bucket.

#### NOTE

1. This tutorial uses two different OBS buckets.
2. The function you create must be in the same region (default region) as the OBS buckets.

### Procedure

- Create two buckets on the OBS console.
- Create a function with an OBS trigger.
- Upload an image to one of the buckets.
- The function is triggered to compress the image.
- The function uploads the processed image to the other bucket.

#### NOTE

After you complete this tutorial, your account will have the following resources:

1. Two OBS buckets (respectively used for storing uploaded and processed images)
2. A thumbnail image creation function (`fss_examples_image_thumbnail`)
3. An OBS trigger used for associating the function with the OBS buckets

## 1.2 Preparation

Before creating a function and adding an event source, you need to create two OBS buckets to respectively store uploaded and compressed images.

After creating the OBS buckets, you must create an agency to delegate FunctionGraph to access OBS resources.

### Creating OBS Buckets

#### Precautions

- The function and the source and destination buckets for storing images must be in the same region.
- Use two different OBS buckets. If only one bucket is used, the function will be executed infinitely. (When an image is uploaded to the bucket, the function is triggered to process the image and store the processed image into the bucket again. In this way, the function executes endlessly.)

#### Procedure

**Step 1** Log in to the [OBS console](#), and click **Create Bucket**.

**Step 2** On the **Create Bucket** page, set the bucket information.

- For **Region**, select a region.
- For **Bucket Name**, enter **your-bucket-input**.
- For **Data Redundancy Policy**, select **Single-AZ storage**.
- For **Default Storage Class**, select **Standard**.
- For **Bucket Policies**, select **Private**.
- **Server-Side Encryption**: Select **Disable**.
- For **Direct Reading**, select **Disable**.

Retain the default values for other parameters and click **Create Now**.

**Step 3** Repeat [Step 2](#) to create the destination bucket.

Name the destination bucket as **your-bucket-output**, and select the same region and storage class as those of the source bucket.

**Step 4** View **your-bucket-input** and **your-bucket-output** in the bucket list.

----End

### Creating an Agency

**Step 1** In the left navigation pane of the management console, choose **Management & Governance > Identity and Access Management** to go to the IAM console. Then choose **Agencies** in the navigation pane.

**Step 2** On the **Agencies** page, click **Create Agency**.

**Step 3** Set the agency information.

- For **Agency Name**, enter `serverless_trust`.
- For **Agency Type**, select **Cloud service**.
- For **Cloud Service**, select **FunctionGraph**.
- For **Validity Period**, select **Unlimited**.
- Enter a description.

**Step 4** Click **Next**. On the **Select Policy/Role** page, select **Tenant Administrator** and click **Next**.

 **NOTE**

Users with the **Tenant Administrator** permission can perform any operations on all cloud resources of the enterprise.

**Step 5** Select an authorization scope that meets your service requirements, and click **OK**.

----End

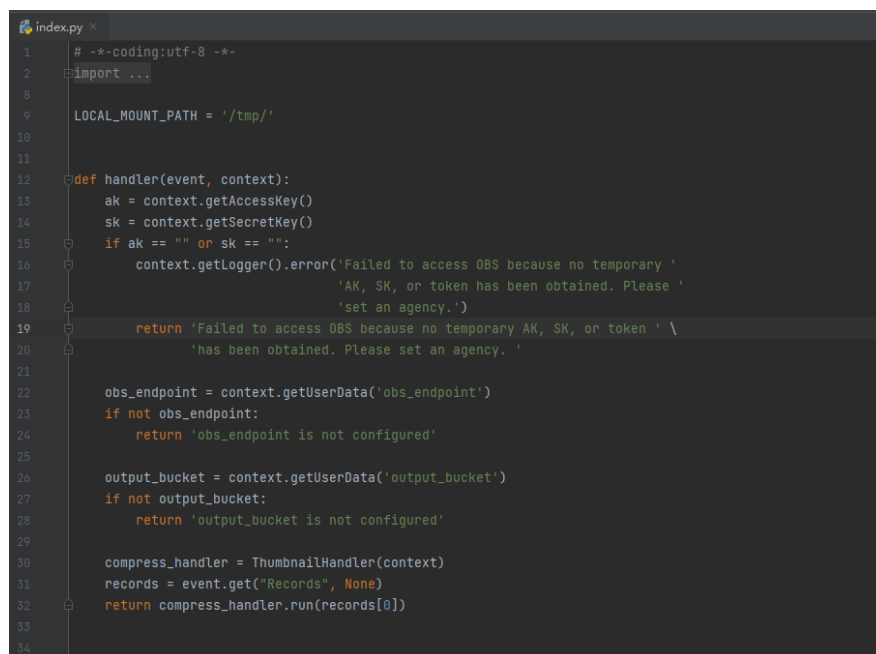
## 1.3 Building a Program

[Download fss\\_examples\\_image\\_thumbnail.zip \(SHA-256 verification package\)](#) to create an image compressing function from scratch.

### Creating a Deployment Package

This example uses a Python function to compress images. For details about function development, see [Developing Functions in Python](#). **Figure 1-1** shows the sample code directory. The service code is not described.

**Figure 1-1** Sample code directory



```
index.py
1 # -*-coding:utf-8 -*-
2 import ...
3
4 LOCAL_MOUNT_PATH = '/tmp/'
5
6
7
8
9
10
11
12 def handler(event, context):
13     ak = context.getAccessKey()
14     sk = context.getSecretKey()
15     if ak == "" or sk == "":
16         context.getLogger().error('Failed to access OBS because no temporary '
17                                   'AK, SK, or token has been obtained. Please '
18                                   'set an agency.')
19     return 'Failed to access OBS because no temporary AK, SK, or token ' \
20           'has been obtained. Please set an agency. '
21
22
23     obs_endpoint = context.getUserData('obs_endpoint')
24     if not obs_endpoint:
25         return 'obs_endpoint is not configured'
26
27     output_bucket = context.getUserData('output_bucket')
28     if not output_bucket:
29         return 'output_bucket is not configured'
30
31     compress_handler = ThumbnailHandler(context)
32     records = event.get("Records", None)
33     return compress_handler.run(records[0])
34
```

Under the directory, **index.py** is a handler file. The following code is a snippet of the handler file. Parameter **output\_bucket** is the address for storing compressed images and must be configured when you create a function.

```
def handler(event, context):
    ak = context.getAccessKey()
    sk = context.getSecretKey()
    if ak == "" or sk == "":
        context.getLogger().error('Failed to access OBS because no temporary '
                                   'AK, SK, or token has been obtained. Please '
                                   'set an agency.')
        return 'Failed to access OBS because no temporary AK, SK, or token ' \
              'has been obtained. Please set an agency.'

    obs_endpoint = context.getUserData('obs_endpoint')
    if not obs_endpoint:
        return 'obs_endpoint is not configured'

    output_bucket = context.getUserData('output_bucket')
    if not output_bucket:
        return 'output_bucket is not configured'

    compress_handler = ThumbnailHandler(context)
    records = event.get("Records", None)
    return compress_handler.run(records[0])
```

## Creating a Function

When creating a function, specify an agency with OBS access permissions so that FunctionGraph can invoke the OBS service.

**Step 1** Log in to the [FunctionGraph console](#), and choose **Functions > Function List** in the navigation pane.

**Step 2** Click **Create Function**.

**Step 3** Click **Create from scratch** and configure the function information.

After setting the basic information, click **Create Function**.

- **Function Type:** Select **Event Function**.
- For **Function Name**, enter **fss\_examples\_image\_thumbnail**.
- For **Agency**, select **serverless\_trust** created in [Creating an Agency](#).
- For **Runtime**, select **Python3.6**

**Step 4** On the **fss\_examples\_image\_thumbnail** details page, configure the following information:

1. On the **Code** tab, choose **Upload > Local ZIP**, upload the sample code **fss\_examples\_image\_thumbnail.zip**.
2. Choose **Configuration > Basic Settings**, set the following parameters, and click **Save**.
  - For **Memory**, select **256**.
  - For **Execution Timeout**, enter **40**.
  - For **Handler**, retain the default value **index.handler**.
  - For **App**, retain the default value **default**.
  - For **Description**, enter **Image compressing**.
3. Choose **Configuration > Environment Variables**, set environment variables, and click **Save**.

**output\_bucket:** the output bucket parameter defined in **index.py**. Set the value to **your-bucket-output**, the bucket created in [Creating OBS Buckets](#).

**obs\_endpoint:** the bucket address parameter defined in **index.py**. Set the value to **obs.region.myhuaweicloud.com**.

**Table 1-1** Environment variable description

Environment Variable	Description
obs_endpoint	OBS endpoint.
output_bucket	OBS bucket for storing output images.

----End

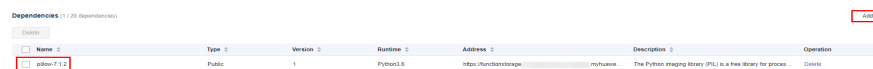
## Selecting a Dependency

The sample code depends on the Pillow package, which should be imported as dependencies. The procedure is as follows:

**Step 1** Go to the **fss\_examples\_image\_thumbnail** details page, click the **Code** tab, and click **Add** in the **Dependencies** area at the bottom.

**Step 2** Add public dependency **pillow-7.1.2**.

**Figure 1-2** Adding dependencies



----End

### NOTE

You do not need to configure the reference after adding a dependency, as it is preconfigured in the function code of the demo package.

## 1.4 Adding an Event Source

After creating the OBS buckets and function, you can add an event source to the function by creating an OBS trigger. Perform the following procedure:

**Step 1** On the **fss\_examples\_image\_thumbnail** page, choose **Configuration > Triggers**, and click **Create Trigger**.

**Step 2** Select **Object Storage Service (OBS)** for **Trigger Type**, and set the trigger information, as shown in [Figure 1-3](#).

- For **Bucket Name**, select **your-bucket-input** created in [Creating OBS Buckets](#).
- For **Events**, select **Put** and **Post**.



**Figure 1-3** Creating a trigger

Trigger Type: Object Storage Service (OBS)

\* Bucket Name: your-bucket-input [Create Bucket](#)

Cannot be the same as that of the current user's existing buckets. Cannot be the same as that of any other user's existing buckets. Cannot be edited after creation.

\* Events: Post, Put

Event Notification Name: obs-event-vbuh

Prefix: Enter a prefix, such as images/.

Suffix: Enter a suffix, such as JPG.

**Step 3** Click **OK**.

**NOTE**

After the OBS trigger is created, when an image is uploaded or updated to bucket **your-bucket-input**, an event is generated to trigger the function.

----End

## 1.5 Compressing Images

When an image is uploaded or updated to bucket **your-bucket-input**, an event is generated to trigger the function. The function compresses the image and stores the compressed one into bucket **your-bucket-output**.

### Uploading an Image to Generate an Event

Log in to the [OBS console](#), go to the object page of the **your-bucket-input** bucket, and upload the **image.jpg** image, as shown in [Figure 1-4](#).

**Figure 1-4** Uploading an image

<input type="checkbox"/>	Name	Storage Cl...	Size	Encrypted
<input type="checkbox"/>	<a href="#">image.jpg</a>	Standard	76.87 KB	No

**NOTE**

The size of the original **image.jpg** file exceeds 28 KB.

### Triggering the Function

After the image is uploaded to bucket **your-bucket-input**, OBS generates an event to trigger the image compressing function. The function compresses the image and stores the compressed one into bucket **your-bucket-output**. You can view running logs of the function on the **Logs** tab page.

Go to the **Objects** page of the **your-bucket-output** bucket and view the size of the compressed image.

**Figure 1-5** Compressing the image

<input type="checkbox"/>	Name	Storage Cl...	Size	Encrypted
<input type="checkbox"/>	<a href="#">image-thumbnail.jpg</a>	Standard	3.68 KB	No

# 2 Watermarking Images

---

## 2.1 Introduction

The best practice for FunctionGraph guides you through image watermarking based on a function.

### Scenarios

- Upload images to a specified OBS bucket.
- Watermark each uploaded image.
- Upload the processed images to another specified OBS bucket.

#### NOTE

1. This tutorial uses two different OBS buckets.
2. The function you create must be in the same region (default region) as the OBS buckets.

### Procedure

- Create two buckets on the OBS console.
- Create a function with an OBS trigger.
- Upload an image to one of the buckets.
- The function is triggered to watermark the image.
- The function uploads the processed image to the other bucket.

#### NOTE

After you complete the operations in this tutorial, your account will have the following resources:

1. Two OBS buckets (respectively used for storing uploaded and processed images)
2. An image watermarking function
3. An OBS trigger used for associating the function with the OBS buckets

## 2.2 Preparation

Before creating a function and adding an event source, you need to create two OBS buckets to respectively store uploaded and watermarked images.

After creating the OBS buckets, you must create an agency to delegate FunctionGraph to access OBS resources.

## Creating OBS Buckets

### Precautions

- The function and the source and destination buckets for storing images must be in the same region.
- Use two different OBS buckets. If only one bucket is used, the function will be executed infinitely. (When an image is uploaded to the bucket, the function is triggered to process the image and store the processed image into the bucket again. In this way, the function executes endlessly.)

### Procedure

**Step 1** Log in to the [OBS console](#), and click **Create Bucket**.

**Step 2** On the **Create Bucket** page, set the bucket information.

- For **Region**, select a region.
- For **Data Redundancy Policy**, select **Single-AZ storage**.
- For **Bucket Name**, enter **hugb-bucket-input**.
- For **Default Storage Class**, select **Standard**.
- For **Bucket Policies**, select **Private**.
- For **Server-Side Encryption**: select **Disable**
- For **Direct Reading**, select **Disable**.

Click **Create Now**.

**Step 3** Repeat [Step 2](#) to create the destination bucket.

Name the destination bucket as **hugb-bucket-output**, and select the same region and storage class as those of the source bucket.

**Step 4** View **hugb-bucket-input** and **hugb-bucket-output** in the bucket list.

----End

## Creating an Agency

**Step 1** In the left navigation pane of the management console, choose **Management & Governance > Identity and Access Management** to go to the IAM console. Then choose **Agencies** in the navigation pane.

**Step 2** On the **Agencies** page, click **Create Agency**.

**Step 3** Set the agency information.

- For **Agency Name**, enter **serverless\_trust**.
- For **Agency Type**, select **Cloud service**.
- For **Cloud Service**, select **FunctionGraph**.

- For **Validity Period**, select **Unlimited**.
- Enter a description.

**Step 4** Click **Next**. On the **Select Policy/Role** page, select **Tenant Administrator** and click **Next**.

 **NOTE**

Users with the Tenant Administrator permission can perform any operations on all cloud resources of the enterprise.

**Step 5** Select an authorization scope that meets your service requirements, and click **OK**.

----End

## 2.3 Building a Program

[Download watermark.zip](#) to create an image watermarking function from scratch.

### Creating a Deployment Package

This example uses a Python function to watermark images. For details about function development, see [Developing Functions in Python](#). **Figure 2-1** shows the sample code directory. The service code is not described.

**Figure 2-1** Sample code directory



Under the directory, **index.py** is a handler file. The following code is a snippet of the handler file. Parameter **obs\_output\_bucket** is the address for storing watermarked images and must be configured when you create a function.

```
def handler(event, context):
    srcBucket, srcObjName = getObjInfoFromObsEvent(event)
    outputBucket = context.getUserData('obs_output_bucket')

    client = newObsClient(context)
    # download file uploaded by user from obs
    localFile = "/tmp/" + srcObjName
    downloadFile(client, srcBucket, srcObjName, localFile)

    outFileName, outFile = watermark_image(localFile, srcObjName)
    # Upload converted files to a new OBS bucket.
    uploadFileToObs(client, outputBucket, outFileName, outFile)

    return 'OK'
```

## Creating a Function

When creating a function, specify an agency with OBS access permissions so that FunctionGraph can invoke the OBS service.

**Step 1** Log in to the [FunctionGraph console](#), and choose **Functions > Function List** in the navigation pane.

**Step 2** Click **Create Function**.

**Step 3** Click **Create from scratch** and configure the function information.

After setting the basic information, click **Create**.

- **Function Type:** Select **Event Function**.
- For **Function Name**, enter **fss\_examples\_image\_watermark**.
- For **Agency**, select **serverless\_trust** created in [Creating an Agency](#).
- For **Runtime**, select **Python 3.6**.

**Step 4** Go to the **fss\_examples\_image\_watermark** details page, click the **Code** tab, click **Add** in the **Dependencies** area at the bottom, and add the public dependency **pillow-7.1.2**.

**Figure 2-2** Adding a dependency

Name	Type	Version	Runtime	Address	Description
pillow-7.1.2	Public	1	Python3.6	https://functiongraph-.../myhuaweicloud.co...	The Python imaging library (PIL) is a free library for processing off...

**Step 5** On the **fss\_examples\_image\_watermark** details page, configure the following information:

1. On the **Code** tab, choose **Upload > Local ZIP**, upload the sample code **watermark.zip**.
2. Choose **Configuration > Basic Settings**, set the following parameters, and click **Save**.
  - For **Memory**, select **128**.
  - For **Execution Timeout**, enter **3**.
  - For **Handler**, retain the default value **index.handler**.
  - For **App**, retain the default value **default**.
  - For **Description**, enter **Image watermarking**.
3. Choose **Configuration > Environment Variables**, set environment variables, and click **Save**. The following figure is for reference only. Replace the following values with the actual values.

**obs\_output\_bucket:** the output bucket parameter defined in **index.py**. Set the value to **hugb-bucket-output**, the bucket created in [Creating OBS Buckets](#) for storing watermarked images.

**obs\_region:** region where the OBS bucket **obs\_output\_bucket** resides, for example, **eu-west-101**.

**Figure 2-3** Adding environment variables

Key	Value
obs_output_bucket	hugb-bucket-output
obs_region	...

**Table 2-1** Environment variable description

Environment Variable	Description
obs_region	Region to which the OBS bucket belongs. The value must be the same as the region to which the function belongs.
obs_output_bucket	OBS bucket for storing watermarked images.

----End

## 2.4 Adding an Event Source

After creating the OBS buckets and function, you can add an event source to the function by creating an OBS trigger. Perform the following procedure:

- Step 1** On the `fss_examples_image_watermark` page, click the **Triggers** tab and click **Create Trigger**.
- Step 2** Select **OBS** for **Trigger Type**, and set the trigger information, as shown in [Figure 2-4](#).

For **Bucket Name**, select `hugb-bucket-input` created in [Creating OBS Buckets](#).

For **Events**, select **Put** and **Post**.

**Figure 2-4** Creating an OBS trigger

**Create Trigger**

---

Trigger Type ? Object Storage Service (OBS)

\* Bucket Name hugb-bucket-input Create Bucket

OBS bucket to be used as an event source.  
Cannot be the same as that of the current user's existing buckets. Cannot be the same as that of any other user's existing buckets. Cannot be edited after creation.

\* Events ? Post Put

Event Notification Name obs-event-cg64

Prefix Enter a prefix, such as images/.

Prefix to limit notifications to those about objects whose names start with the matching characters.

Suffix Enter a suffix, such as JPG.

Suffix to limit notifications to those about objects whose names end with the matching characters.

**Recursive Invocation**

If your function writes objects to an OBS bucket, use different buckets for input and output. Using the same bucket will increase the risk of recursive invocation and result in high usage and costs.

I understand and agree.

**Step 3** Click **OK**.

 **NOTE**

After the OBS trigger is created, when an image is uploaded or updated to bucket **hugb-bucket-input**, an event is generated to trigger the function.

----End

## 2.5 Watermarking Images

When an image is uploaded or updated to bucket **hugb-bucket-input**, an event is generated to trigger the function. The function watermarks the image and stores the watermarked one into bucket **hugb-bucket-output**.

### Uploading an Image to Generate an Event

Log in to the [OBS console](#), go to the object page of the **hugb-bucket-input** bucket, and upload the **image.jpg** image, as shown in [Figure 2-5](#).

**Figure 2-5** Uploading an image



### Triggering the Function

After the image is uploaded to bucket **hugb-bucket-input**, OBS generates an event to trigger the image watermarking function. The function watermarks the image and stores the watermarked one into bucket **hugb-bucket-output**. You can view running logs of **fss\_examples\_image\_watermark** on the **Logs** tab page.

The **Objects** page of the bucket **hugb-bucket-output** displays the watermarked image **image.jpg**, as shown in [Figure 2-6](#). In the **Operation** column, click **Download** to download the image and view the watermarking effect, as shown in [Figure 2-7](#).

**Figure 2-6** Output image





**Figure 2-7** Watermarked image



# 3 Integrating with CTS to Analyze Login/Logout Security

## 3.1 Introduction

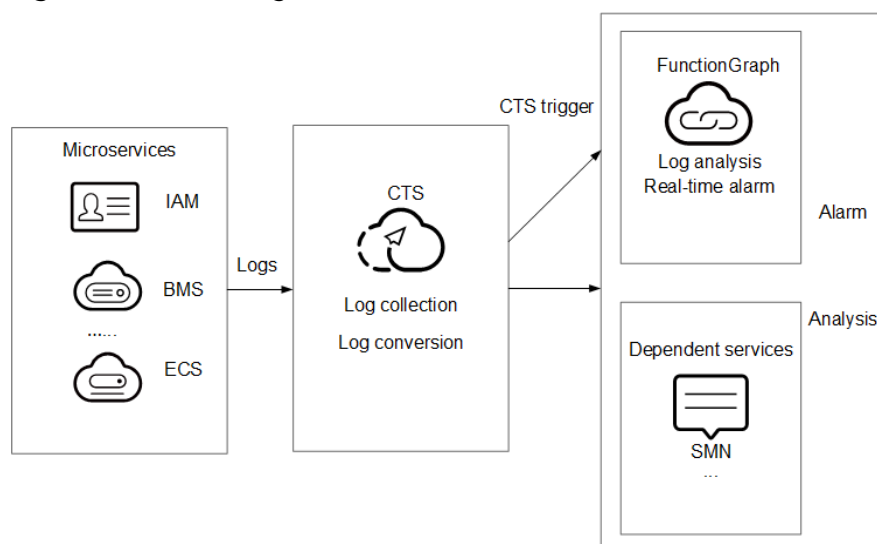
### Scenarios

Collect real-time records of operations on cloud resources.

Create a Cloud Trace Service (CTS) trigger to obtain records of subscribed cloud resource operations; analyze and process the operation records, and report alarms.

Use SMN to push alarm messages to service personnel by SMS message or email. The processing workflow is shown in [Figure 3-1](#).

**Figure 3-1** Processing workflow



### Values

- Quickly analyzes operation records collected by CTS and filters out operations from specified IP addresses.

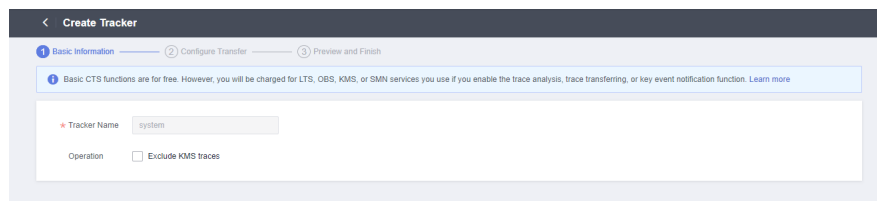
- Processes and analyzes data in response to log events in a serverless architecture, which features automatic scaling, no operation and maintenance, and pay-per-use billing.
- Sends alarm notifications through SMN.

## 3.2 Preparation

### Enabling CTS

Configure a tracker on CTS, as shown in [Figure 3-2](#). For details, see [Configuring a Tracker](#).

**Figure 3-2** Configuring a tracker



### Creating an Agency

**Step 1** Log in to the [IAM console](#), and choose **Agencies** in the navigation pane.

**Step 2** On the **Agencies** page, click **Create Agency**.

**Step 3** Set the agency information.

- For **Agency Name**, enter **serverless\_trust**.
- For **Agency Type**, select **Cloud service**.
- For **Cloud Service**, select **FunctionGraph**.
- For **Validity Period**, select **Unlimited**.
- For **Description**, enter a description.

**Step 4** Click **Next**. On the **Select Policy/Role** page, select **Tenant Administrator** and click **Next**.

#### NOTE

Users with the **Tenant Administrator** permission can perform any operations on all cloud resources of the enterprise.

**Step 5** Click **OK**.

----End

### Pushing Alarm Messages

- Create a topic named **cts\_test** on the SMN console. For details, see [Creating a Topic](#).
- Add subscriptions to the **cts\_test** topic to push alarm messages. For details, see [Adding a Subscription](#).

 NOTE

Alarm messages of a subscribed topic can be pushed through emails, SMS messages, and HTTP/HTTPS.

In this example, when operation log events trigger the specified function, the function filters operations that are performed by users not in the IP address whitelist, and pushes alarm messages to the subscription endpoints.

## 3.3 Building a Program

[Download index.zip](#) to create an alarm log analysis function from scratch.

### Creating a Function

Create a function by uploading the [sample code package](#) to extract logs. Select the Python 2.7 runtime and the agency `serverless_trust` created in [Creating an Agency](#). For details about how to create a function, see [Creating an Event Function](#).

This function analyzes received operation records, filters logins or logouts from unauthorized IP addresses using a whitelist, and sends alarms under a specified SMN topic. This function can be used to build an account security monitoring service.

### Setting Environment Variables

On the **Configuration** tab page of the function details page, set the environment variables listed in [Table 3-1](#).

**Table 3-1** Environment variables

Environment Variable	Description
SMN_Topic	SMN topic.
RegionName	Region name.
IP	IP address whitelist.

Set the environment variables by following the procedure in [Environment Variables](#).

## 3.4 Adding an Event Source

Create a CTS trigger, as shown in [Figure 3-3](#).

**Figure 3-3** Creating a CTS trigger

CTS records the logins and logouts of users on IAM.

### 3.5 Processing Operation Records

The function runs in response to account logins and logouts to filter those not from the IP address whitelist, and sends a message or email through SMN, as shown in [Figure 3-4](#).

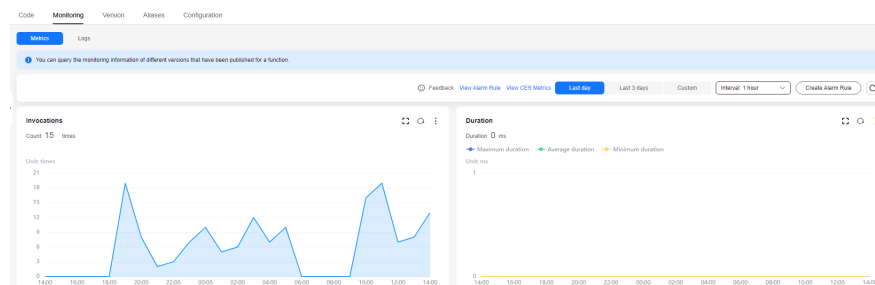
**Figure 3-4** Email notification

```
Illegal operation[ IP:10.65.56.139, Action:login]
-----
```

The email contains the unauthorized IP address and user operation (login or logout).

On the **Monitoring** tab page of the function, check the number of invocations, as shown in [Figure 3-5](#).

**Figure 3-5** Function metrics



# 4 Building an HTTP Function with Spring Boot

## Introduction

This chapter describes how to deploy services on FunctionGraph using Spring Boot. Usually, you may build Spring Boot applications using [SpringInitializer](#) or IntelliJ IDEA. This chapter uses the Spring.io project in <https://spring.io/guides/gs/rest-service/> as an example to deploy an HTTP function on FunctionGraph.

## Procedure

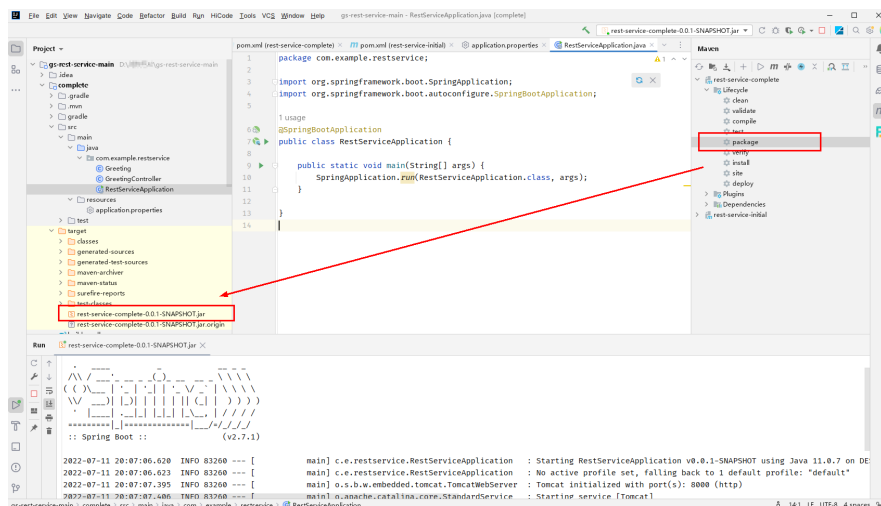
To deploy an existing project to FunctionGraph, change the listening port of the project to **8000**, and create a file named **bootstrap** in the same directory as the JAR file to include the command for executing the JAR file.

In this example, a Maven project created using IntelliJ IDEA is used.

### Building a Code Package

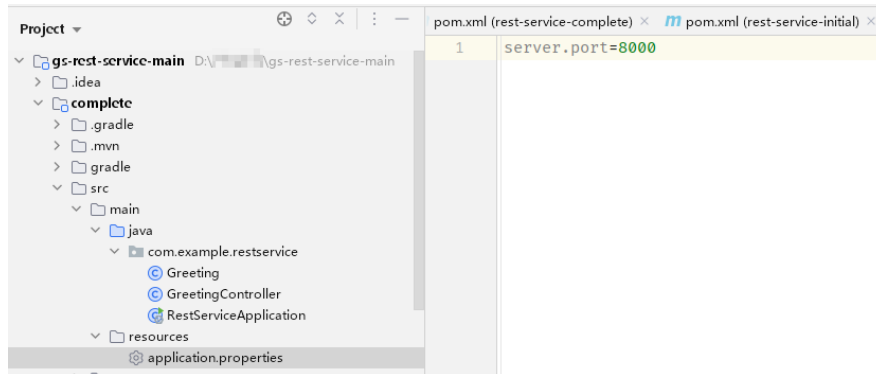
1. Open the Spring Boot project and click **package** in the **Maven** area to generate a JAR file.

Figure 4-1 Generating a JAR file



2. Set the web port to **8000 (do not change this port)** using the **application.properties** file or specify the port during startup. HTTP functions only support this port.

**Figure 4-2** Configuring port 8000



3. Create a file named **bootstrap** in the same directory as the JAR file, and enter the startup parameters.  

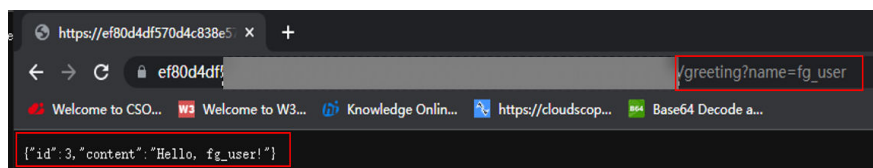
```
/opt/function/runtime/java11/rtsp/jre/bin/java -jar -Dfile.encoding=utf-8 /opt/function/code/rest-service-complete-0.0.1-SNAPSHOT.jar
```
4. Compress the JAR file and **bootstrap** file into a ZIP package.

### Creating an HTTP Function and Uploading Code

Create an HTTP function and upload the ZIP file. For details, see [Creating an HTTP Function](#).

### Verifying the Result

- Using a test event
  - a. On the function details page, select a version and click **Configure Test Event**.
  - b. On the **Configure Test Event** page, select the event template **apig-event-template**, and modify the **path** and **pathParameters** parameters in the template to construct a simple GET request.
  - c. Click **Create**.
  - d. Click **Test** to obtain the response.  
**When debugging a function, increase the memory size and timeout, for example, increase them to 512 MB and 5s.**
- Using an APIG trigger
  - a. Create an APIG trigger by referring to [Using an APIG Trigger](#). Set the authentication mode to **None** for debugging.
  - b. Copy the generated URL, add the request parameter **greeting?name=fg\_user** to the end of the URL (see [Figure 4-3](#)), and access the URL using a browser. The response shown in the following figure is displayed.

**Figure 4-3** Invoking the function

The default APIG trigger URL is in the format "*Domain name/Function name*". In this example, the URL is **https://your\_host.com/springboot\_demo**, where the function name **springboot\_demo** is the first part of the path. If you send a GET request for **https://your\_host.com/springboot\_demo/greeting**, the request address received by Spring Boot contains **springboot\_demo/greeting**. If you have uploaded an existing project, you cannot access your own services because the path contains a function name. To prevent this from happening, use either of the following methods to annotate or remove the function name:

- Method 1: Modify the mapping address in the code. For example, add the first part of the default path to the `GetMapping` or class annotation.

**Figure 4-4** Modifying the mapping address

```
@RestController
public class GreetingController {

    1 usage
    private static final String template = "Hello, %s!";
    1 usage
    private final AtomicLong counter = new AtomicLong();

    @GetMapping("/springboot_demo/greeting")
    public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {
        return new Greeting(counter.incrementAndGet(), String.format(template, name));
    }
}
```

- Method 2: Click the trigger name to go to the APIG console, and delete the function name in the path.

## FAQ

### 1. What Directories Are Accessible to My Code?

An uploaded code package is stored in the **/opt/function/code/** directory of the function (runtime environments, compute resources, or containers). However, the directory can only be read and cannot be written. If some data must be written to the function during code running and logged locally, or your dependency is written by default to the directory where the JAR file is located, use the **/tmp** directory.

### 2. How Are My Logs Collected and Output?

Function instances that have not received any requests during a specific period of time will be deleted together with their local logs. You will be unable to view the function logs during function running. Therefore, in



addition to writing logs to your local host, output logs to the console by setting the output target of Log4j to **System.out** or by using the **print** function.

Logs output to the console will be collected. If you have enabled LTS, the logs will also be stored in LTS for near real-time analysis.

Suggestion: [Enable LTS](#), and click **Go to LTS** to view and analyze logs on the **Real-Time Logs** tab page.

### 3. What Permissions Does My Code Have?

Similar to common event functions, code does not have the **root** permission. Code or commands requiring this permission cannot be executed in HTTP functions.

### 4. How Do I Package Spring Boot Projects of Multiple Modules?

Configure the following to package these Spring Boot projects.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <mainClass>com.example.YourServiceMainClass</mainClass>
      </configuration>
    </plugin>
  </plugins>
</build>
```

# 5 Creating a FunctionGraph Backend API That Uses a Custom Authorizer

---

## 5.1 Introduction

In addition to IAM and app authentication, APIG also supports custom authentication with your own system, which can better adapt to your business capabilities.

This chapter guides you through the process of creating a FunctionGraph API that uses a custom authorizer.

### Solution

- Log in to the FunctionGraph console, and create a function for custom authentication.
- Create a service function.
- Create an API group on the APIG console.
- Create an API and configure a custom authorizer and a FunctionGraph backend for it.
- Debug the API.

#### NOTE

After you complete the operations in this tutorial, your Huawei Cloud account will have the following resources:

1. An API group storing APIs
2. A custom authentication function
3. A service function
4. An API with a custom authorizer and a FunctionGraph backend

## 5.2 Resource Planning

Ensure that the following resources are in the same region.

**Table 5-1** Resource planning

Resource	Quantity
API group	1
Custom authentication function	1
Service function	1
API	1

## 5.3 Building a Program

### Creating an API group

Before creating a function and adding an event source, create an API group to store and manage APIs.

 **NOTE**

Before enabling APIG functions, buy a gateway by referring to section "Buying a Gateway".

**Step 1** Log in to the APIG console, choose **API Management > API Groups** in the navigation pane, and click **Create API Group** in the upper right.

**Step 2** Select **Create Directly**, set the group information, and click **OK**.

- **Name:** Enter a group name, for example, **APIGroup\_test**.
- **Description:** Enter a description about the group.

----End

### Creating a Custom Authentication Function

Frontend custom authentication means APIG uses a function to authenticate received API requests. To authenticate API requests by using your own system, create a frontend custom authorizer in APIG. Create a FunctionGraph function with the required authentication information. Then use it to authenticate APIs in APIG.

This section uses the header parameter **event["headers"]** as an example. For the description about request parameters, see [Request Parameter Code Example](#).

**Step 1** In the left navigation pane of the management console, choose **Compute > FunctionGraph** to go to the FunctionGraph console. Then choose **Functions > Function List** in the navigation pane.

**Step 2** Click **Create Function**.

**Step 3** Set the function information, and click **Create Function**.

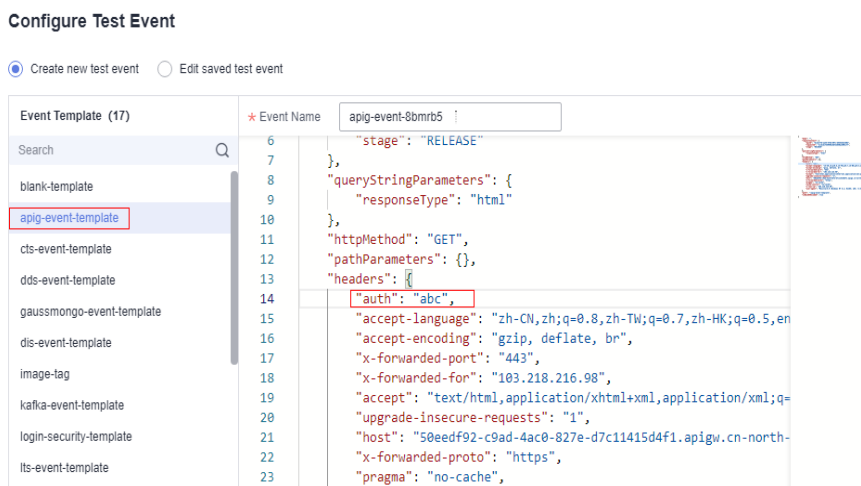
- **Template:** Select **Create from scratch**.

- **Function Type:** Select **Event Function**.
- **Function Name:** Enter a function name, for example, **apig-test**.
- **Agency:** Select **Use no agency**.
- **Runtime:** Select **Python 2.7**.

**Step 4** On the function details page that is displayed, click the **Code** tab and copy the **example request parameter code** to the online editor, and click **Deploy**.


**Step 5** Click **Configure Test Event**, and select an event template. Modify the template as required, and click **Create**. In this example, add **"auth":"abc"** to **"headers"**.

**Figure 5-1** Configuring a test event



**Step 6** Click **Test**. If the result is **Execution successful**, the function is successfully created.

**Figure 5-2** Viewing the execution result

 Execution successful

**Function Output**

```
{
  "body": "{\"status\": \"allow\", \"context\": {\"user\": \"success\"}}",
  "statusCode": 200
}
```

----End

## Creating a Custom Authorizer

Create a custom authorizer in APIG and connect it to the frontend custom authentication function.

**Step 1** In the left navigation pane of the management console, choose **Middleware > API Gateway** to go to the APIG console. In the navigation pane, choose **API Management > API Policies**. On the **Custom Authorizers** tab, click **Create Custom Authorizer**.

**Step 2** Configure basic information about the custom authorizer according to the following figure.

- **Name:** Enter a name, for example, **Authorizer\_test**.
- **Type:** Select **Frontend**.
- **Function URN:** Select **apig-test**.

**Figure 5-3** Creating a custom authorizer

Create Custom Authorizer

---

\* Name

\* Type  Frontend  Backend

\* Function URN  [Select](#)

\* Version/Alias

\* Max. Cache Age (s)

Identity Sources

Parameter Location	Parameter Name	Operation
+ Add Identity Source		

Send Request Body

User Data   
0/2,048

**i** The user data will be stored as plaintext. Be careful with the information that you include here.

**Step 3** Click **OK**.

----End

## Creating a Backend Service Function

APIG supports FunctionGraph backends. After you create a FunctionGraph backend API, APIG will trigger the relevant function, and the function execution result will be returned to APIG.

**Step 1** Create a service function by referring to [Creating a Custom Authentication Function](#). The function name must be unique.

**Step 2** On the **Code** tab of the function details page, copy the following code to the online editor, and click **Deploy**.

```
# -*- coding:utf-8 -*-
import json
def handler(event, context):
    body = "<html><title>Functiongraph Demo</title><body><p>Hello, FunctionGraph!</p></body></html>"
    print(body)
```

```
return {
  "statusCode":200,
  "body":body,
  "headers": {
    "Content-Type": "text/html",
  },
  "isBase64Encoded": False
}
```

----End

## Request Parameter Code Example

The following are the requirements you must meet when developing FunctionGraph functions. Python 2.7 is used as an example.

The function must have a clear API definition. Example:

### def handler (event, context)

- **handler**: name of the entry point function. The name must be consistent with that you define when creating a function.
- **event**: event parameter defined in JSON format for the function.
- **context**: runtime information provided for executing the function. For details, see [SDK APIs](#).

**event** supports three types of request parameters in the following formats:

- Header parameter: `event["headers"]["Parameter name"]`
- Query string: `event["queryStringParameters"]["Parameter name"]`
- Custom user data: `event["user_data"]`

The three types of request parameters obtained by the function are mapped to the custom authentication parameters defined in APIG.

- Header parameter: Corresponds to the identity source specified in **Header** for custom authentication. The parameter value is transferred when the API that uses custom authentication is called.
- Query string: Corresponds to the identity source specified in **Query** for custom authentication. The parameter value is transferred when the API that uses custom authentication is called.
- Custom user data: Corresponds to the user data for custom authentication. The parameter value is specified when the custom authorizer is created.
- The function response cannot be greater than 1 MB and must be in the following format:

```
{
  "statusCode":200,
  "body": "{\"status\": \"allow\", \"context\": {\"user\": \"abc\"}}"
```

The **body** field is a character string, which is JSON-decoded as follows:

```
{
  "status": "allow/deny",
  "context": {
    "user": "abc"
  }
}
```

The **status** field is mandatory and is used to identify the authentication result. The authentication result can only be **allow** or **deny**. **allow** indicates that the authentication is successful, and **deny** indicates that the authentication fails.

The **context** field is optional and can only be key-value pairs. The key value cannot be a JSON object or an array.

The **context** field contains custom user data. After successful authentication, the user data is mapped to the backend parameters. The parameter name in **context** is case-sensitive and must be the same as the system parameter name. The parameter name must start with a letter and can contain 1 to 32 characters, including letters, digits, hyphens (-), and underscores (\_).

### Example Header Parameter

```
# -*- coding:utf-8 -*-
import json
def handler(event, context):
    if event["headers"].get("auth")=='abc':
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status":"allow",
                "context":{
                    "user":"success"
                }
            })
        }
    else:
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status":"deny",
            })
        }
    return json.dumps(resp)
```

### Example Query String

```
# -*- coding:utf-8 -*-
import json
def handler(event, context):
    if event["queryStringParameters"].get("test")=='abc':
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status":"allow",
                "context":{
                    "user":"abcd"
                }
            })
        }
    else:
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status":"deny",
            })
        }
    return json.dumps(resp)
```

### Example User Data

```
# -*- coding:utf-8 -*-
import json
def handler(event, context):
    if event.get("user_data")=='abc':
```

```
resp = {
  'statusCode': 200,
  'body': json.dumps({
    "status": "allow",
    "context": {
      "user": "abcd"
    }
  })
}
else:
  resp = {
    'statusCode': 200,
    'body': json.dumps({
      "status": "deny",
    })
  }
}
return json.dumps(resp)
```

## 5.4 Adding an Event Source

### Creating an API

After creating an API group, custom authentication function, and backend function, create a FunctionGraph backend API that uses a custom authorizer by performing the following steps:

**Step 1** Log in to the APIG console, choose **API Management** > **APIs** in the navigation pane, and click **Create API** in the upper right.

**Step 2** Configure the basic information according to [Figure 5-4](#) and [Figure 5-5](#).

- **API Name:** Enter a name, for example, **API\_test**.
- **Group:** Select API group **APIGroup\_test**.
- **URL:** Set **Method** to **ANY**, **Protocol** to **HTTPS**, and **Path** to **/testAPI**.
- **Gateway Response:** Select **default**.
- **Authentication Mode:** Select **Custom**.
- **Custom Authorizer:** Select **Authorizer\_test**.

**Figure 5-4** Configuring frontend definition

Frontend Definition

\* API Name   
Enter a string of 3 to 255 characters starting with a letter. Only letters, digits, hyphens (-), underscores (\_), periods (.), slash (/), colons (:), and parentheses (()) are allowed.

\* Group

\* URL

Method	Protocol	Subdomain Name	Path
<input type="text" value="ANY"/>	<input type="text" value="HTTPS"/>	<input type="text" value="a5e...71db3b4f54e.apic.cn-e..."/>	<input type="text" value="/testAPI"/>

\* Gateway Response

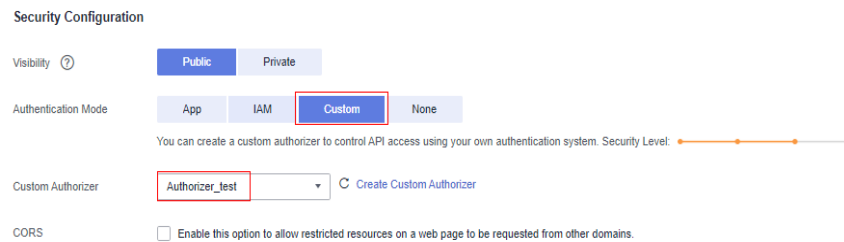
Matching  Exact match  Prefix match  
API requests will be forwarded to the specified path.

Tags

Description

Content Format Type

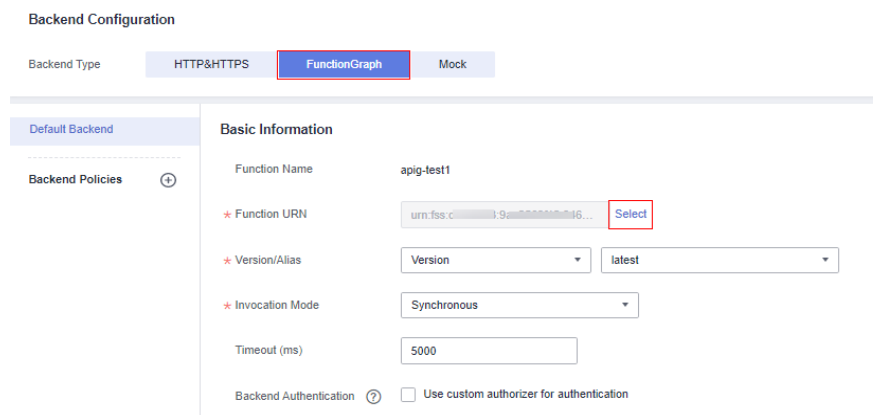


**Figure 5-5** Configuring security settings**NOTE**

For more parameters, see section "Creating an API".

**Step 3** Click **Next** to configure the backend service according to [Figure 5-6](#).

- **Backend Type:** Select **FunctionGraph**.
- **Function URN:** Select the created service function.
- **Version/Alias:** Select the **latest** version.
- **Invocation Mode:** Select **Synchronous**.

**Figure 5-6** Configuring the backend service

**Step 4** Click **Finish**.

**Step 5** Click **Publish** to publish the API in the RELEASE environment.

**Figure 5-7** Publishing an API

----End

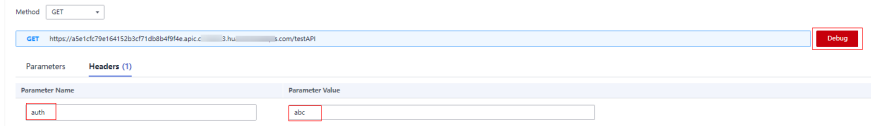
## 5.5 Debugging and Calling the API

APIG provides online debugging, enabling you to check an API after configuring it.

**Step 1** Log in to the APIG console. In the navigation pane, choose **API Management > APIs**. Then click **API\_test**, and click **Debug**.

**Step 2** Add a header parameter and click **Debug**.

- **Parameter Name:** Enter **auth**.
- **Parameter Value:** Enter **abc**.

**Figure 5-8** Adding a header

**Step 3** Check whether the API response contains the content you have defined in the service function. See [Figure 5-9](#).

**Figure 5-9** API response

```
HTTP/1.1 200 OK
Content-Length: 87
Connection: keep-alive
Content-Type: text/html; charset=UTF-8
Date: Tue, 07 Feb 2023 06:39:18 GMT
Server: api-gateway
Strict-Transport-Security: max-age=31536000; includeSubdomains;
X-ApiG-Latency: 2140
X-ApiG-Ratelimit-Api: remain:99,limit:100,time:1 minute
X-ApiG-Ratelimit-Api: remain:15601,limit:16000,time:1 second
X-ApiG-Ratelimit-Api-Allenv: remain:5999,limit:6000,time:1 second
X-ApiG-Ratelimit-Api-Allenv: remain:15601,limit:16000,time:1 second
X-ApiG-Ratelimit-User: remain:9870,limit:10000,time:1 second
X-ApiG-Upstream-Latency: 1539
X-Cff-Billing-Duration: 5
X-Cff-Invoke-Summary: {"funcDigest":"64999f78efbc98714f57b3f190573be","duration":4.952,"billingDuration":5,"memorySize":128,"memoryUsed":25.906,"podName":"pool22-300-128-fusion-67fc9b8d95-s6rsv"}
X-Cff-Request-Id: 495bcd5f5-d474-4aa5-ba04-c79f84d4367c
X-Content-Type-Options: nosniff
X-Download-Options: noopen
X-Frame-Options: SAMEORIGIN
X-Request-Id: dfa7d5925751f31f12221f45459a1312
X-Xss-Protection: 1; mode=block;

<html><title>Functiongraph Demo</title><body><p>Hello, FunctionGraph!</p></body></html>
```

----End

# 6 Building an HTTP Function with Go

---

## Introduction

This chapter describes how to deploy services on FunctionGraph using Go.

HTTP functions do not support direct code deployment using Go. This section uses binary conversion as an example to describe how to deploy Go programs on FunctionGraph.

## Procedure

### Building a code package

Create the source file **main.go**. The code is as follows:

```
// main.go
package main

import (
    "fmt"
    "net/http"

    "github.com/emicklei/go-restful"
)

func registerServer() {
    fmt.Println("Running a Go Http server at localhost:8000/")

    ws := new(restful.WebService)
    ws.Path("/")

    ws.Route(ws.GET("/hello").To(Hello))
    c := restful.DefaultContainer
    c.Add(ws)
    fmt.Println(http.ListenAndServe(":8000", c))
}

func Hello(req *restful.Request, resp *restful.Response) {
    resp.Write([]byte("nice to meet you"))
}

func main() {
    registerServer()
}

# bootstrap
/opt/function/code/go-http-demo
```



**Figure 6-2** Request result

