

**FunctionGraph**

# Getting Started

**Issue**            01  
**Date**             2024-03-13



**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

---

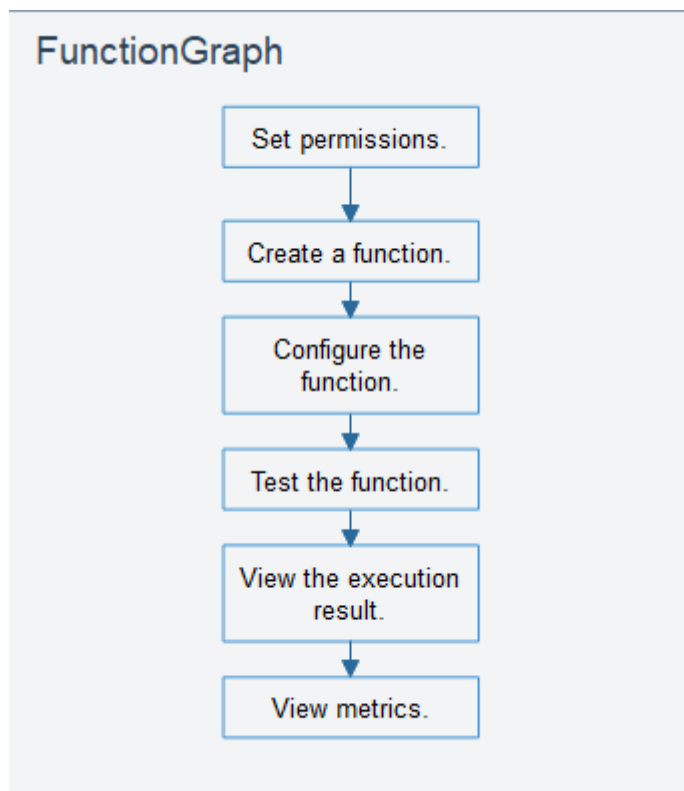
<b>1 Introduction.....</b>	<b>1</b>
<b>2 Creating a Function from Scratch.....</b>	<b>3</b>
<b>3 Creating a Function Using a Template.....</b>	<b>7</b>
<b>4 Deploying a Function Using a Container Image.....</b>	<b>10</b>
4.1 Developing an HTTP Function.....	10
4.2 Developing an Event Function.....	15
<b>5 Getting Started with Common Practices.....</b>	<b>20</b>

# 1 Introduction

## General Procedure

FunctionGraph allows you to run your code without provisioning or managing servers, while ensuring high availability and scalability. All you need to do is upload your code and set execution conditions, and FunctionGraph will take care of the rest. In addition, you pay only for what you use and you are not charged when your code is not running.

To quickly create a function using FunctionGraph, do as follows:



1. Set permissions: Ensure that you have the **FunctionGraph Administrator** permissions.

2. Create a function: Create a function from scratch or using the sample code or a container image.
3. Configure the function: Configure the code source or modify other parameters.
4. Test the function: Create a test event to debug the function.
5. View the execution result: On the function details page, view the execution result based on the configured test event.
6. View metrics: On the **Monitoring** tab page of the function details page, view function metrics.

# 2 Creating a Function from Scratch

---

## Introduction

This section describes how to quickly create and test a HelloWorld function on the FunctionGraph console.

## Step 1: Prepare the Environment

To perform the operations described in this section, ensure that you have the **FunctionGraph Administrator** permissions, that is, the full permissions for FunctionGraph. For more information, see [Permissions Management](#).

## Step 2: Create a Function

1. Log in to the [FunctionGraph console](#). In the navigation pane, choose **Functions > Function List**.
2. Click **Create Function** in the upper right corner and choose **Create from scratch**.
3. On the displayed page, set **Function Name** to **HelloWorld**, retain the default values for other parameters, and click **Create Function**. For details, see [Figure 2-1](#).

**Figure 2-1** Configuring basic information

**Basic Information**

\* Function Type [?](#)

Event Function HTTP Function

Processes event requests and can be triggered by APIG, OBS, and DIS events.

\* Region

Regions are geographic areas isolated from each other. Resources are region-specific and cannot be used across regions through internal network connections. For low network latency and quick resource access, select the nearest region.

\* Function Name

Enter 1 to 60 characters, starting with a letter and ending with a letter or digit. Only letters, digits, hyphens (-), and underscores (\_) are allowed.

Agency [?](#)

[Create Agency](#)

\* Enterprise Project [?](#)

[View Enterprise Project](#)

Runtime [?](#)

4. Configure the code source, copy the following code to the code window, and click **Deploy**.

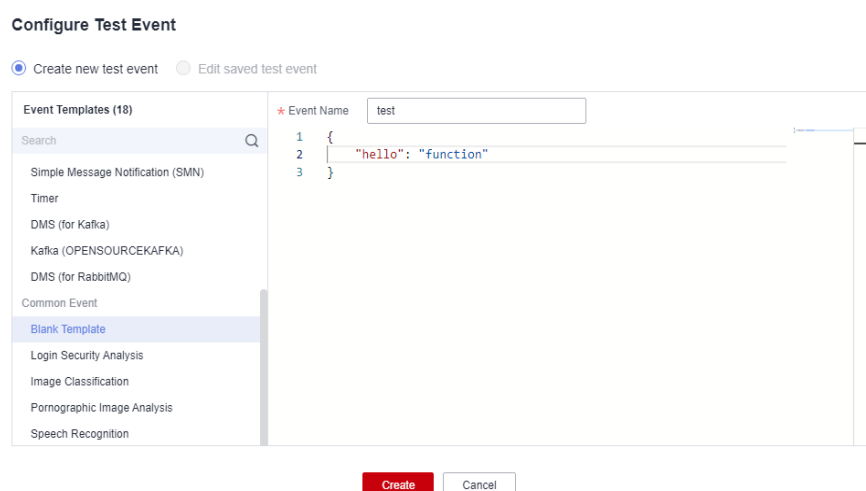
The sample code enables you to obtain test events and print test event information.

```
exports.handler = function (event, context, callback) {  
  const error = null;  
  const output = `Hello message: ${JSON.stringify(event)}`;  
  callback(error, output);  
}
```

### Step 3: Test the Function

1. On the function details page, click **Test**. In the displayed dialog box, create a test event.
2. Select **blank-template**, set **Event Name** to **test**, modify the test event as follows, and click **Create**.

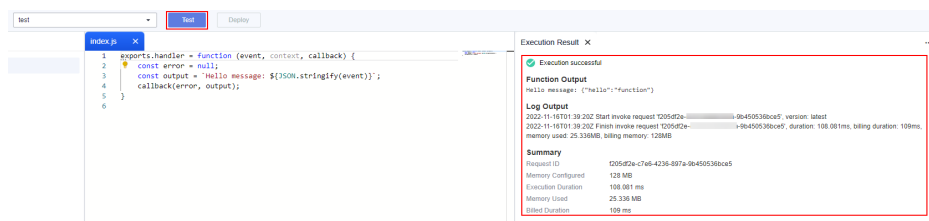
```
{  
  "hello": "function"  
}
```

**Figure 2-2** Configuring a test event

## Step 4: View the Execution Result

Click **Test** and view the execution result on the right.

- **Function Output:** displays the return result of the function.
- **Log Output:** displays the execution logs of the function.
- **Summary:** displays key information of the logs.

**Figure 2-3** Viewing the execution result

### NOTE

A maximum of 2 KB logs can be displayed. For more log information, see [Querying Function Logs](#).

## Step 5: View Monitoring Metrics

On the function details page, click the **Monitoring** tab.

- On the **Monitoring** tab page, choose **Metrics**, and select a time range (such as 5 minutes, 15 minutes, or 1 hour) to query the function.
- The following metrics are displayed: invocations, errors, duration (including the maximum, average, and minimum durations), and throttles.

## Step 6: Delete a Function

1. On the function details page, choose **Operation** > **Delete Function** in the upper right corner.



2. In the confirmation dialog box, enter **DELETE** and click **OK** to release resources in a timely manner.

# 3 Creating a Function Using a Template

## Introduction

FunctionGraph provides templates to automatically complete code and running environment configurations when you create a function, helping you quickly build applications.

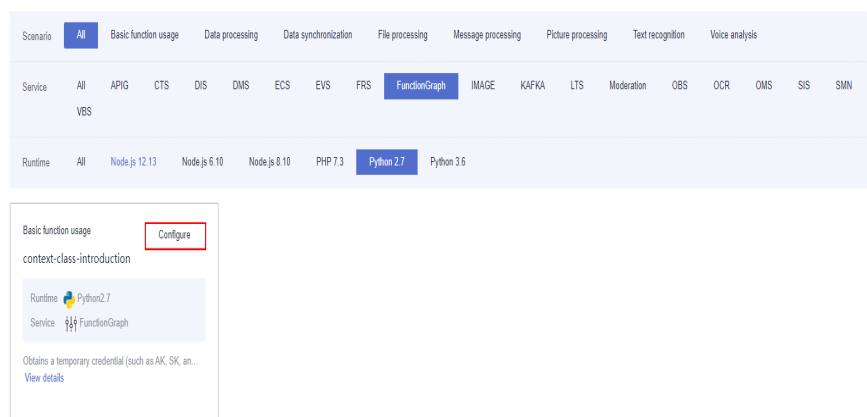
## Step 1: Prepare the Environment

To perform the operations described in this section, ensure that you have the **FunctionGraph Administrator** permissions, that is, the full permissions for FunctionGraph. For more information, see [Permissions Management](#).

## Step 2: Create a Function

1. Log in to the [FunctionGraph console](#). In the navigation pane, choose **Functions > Function List**.
2. Click **Create Function** in the upper right corner and choose **Select template**.
3. Select the template shown in [Figure 3-1](#) and click **Configure**.

**Figure 3-1** Selecting a template

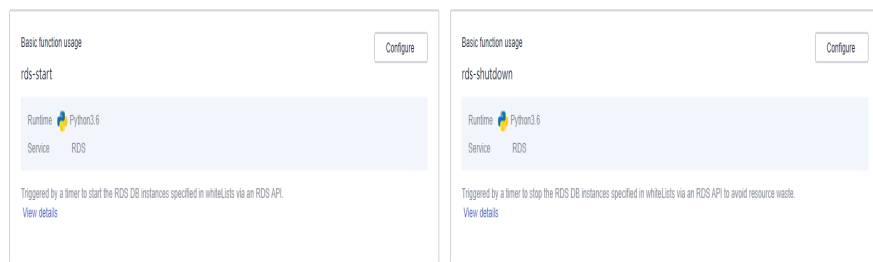


4. Set **Function Name** to **context**, select any agency from the **Agency** drop-down list, retain default values for other parameters, and click **Create Function**.

**NOTE**

- If no agency is configured, the following message will be displayed when the function is triggered:  
Failed to access other services because no temporary AK, SK, or token has been obtained. Please set an agency.
- Huawei Cloud provides function templates for periodically starting and stopping RDS DB instances, helping you efficiently manage resources and reduce maintenance costs.

**Figure 3-2** Templates for periodically starting and stopping Huawei Cloud RDS DB instances



**Figure 3-3** Setting basic information

**Basic Information**

Function Template  
context-class-introduction-python [Reselect](#)

\* Region  
  
Regions are geographic areas isolated from each other. Resources are region-specific and cannot be used across regions through internal network connections. For low network latency and quick resource access, select the nearest region.

\* Function Name  
  
Enter 1 to 60 characters, starting with a letter and ending with a letter or digit. Only letters, digits, hyphens (-), and underscores (\_) are allowed.

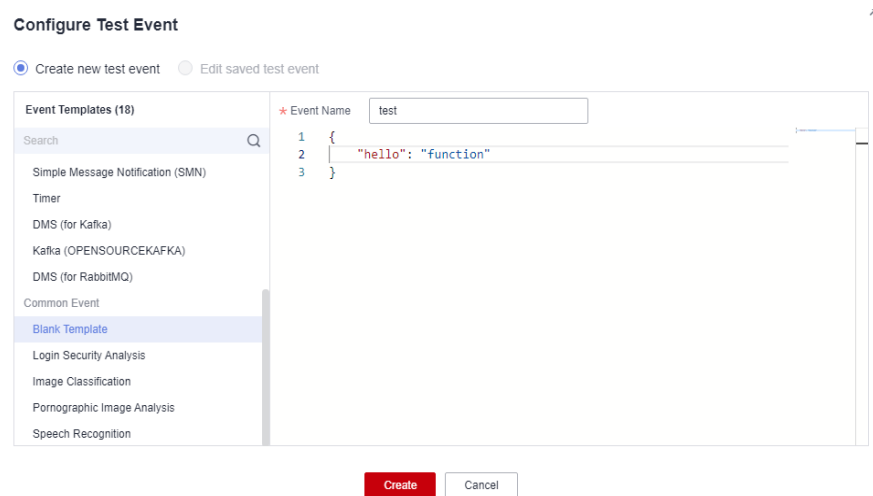
Agency [?](#)  
 [Create Agency](#)

\* Enterprise Project [?](#)  
 [View Enterprise Project](#)

Runtime [?](#)

**Step 3: Test the Function**

1. On the function details page, click **Test**. In the displayed dialog box, create a test event.
2. Select **blank-template**, set **Event Name** to **test**, and click **Create**.

**Figure 3-4** Configuring a test event

## Step 4: View the Execution Result

Click **Test** and view the execution result on the right.

- **Function Output:** displays the return result of the function.
- **Log Output:** displays the execution logs of the function.
- **Summary:** displays key information of the logs.

### NOTE

A maximum of 2 KB logs can be displayed. For more log information, see [Querying Function Logs](#).

## Step 5: View Monitoring Metrics

On the function details page, click the **Monitoring** tab.

- On the **Monitoring** tab page, choose **Metrics**, and select a time range (such as 5 minutes, 15 minutes, or 1 hour) to query the function.
- The following metrics are displayed: invocations, errors, duration (including the maximum, average, and minimum durations), and throttles.

## Step 6: Delete a Function

1. On the function details page, choose **Operation** > **Delete Function** in the upper right corner.
2. In the confirmation dialog box, enter **DELETE** and click **OK** to release resources in a timely manner.

# 4 Deploying a Function Using a Container Image

## 4.1 Developing an HTTP Function

### Introduction

When developing an HTTP function using a custom image, implement an HTTP server in the image and listen on **port 8000** for requests. **(Do not change port 8000 in the examples provided in this section.)** HTTP functions support only APIG triggers.

### Step 1: Prepare the Environment

To perform the operations described in this section, ensure that you have the **FunctionGraph Administrator** permissions, that is, the full permissions for FunctionGraph. For more information, see [Permissions Management](#).

### Step 2: Create an Image

Take the Linux x86 64-bit OS as an example.

1. Create a folder.  
`mkdir custom_container_http_example && cd custom_container_http_example`
2. Implement an HTTP server. Node.js is used as an example. For details about other languages, see [Creating an HTTP Function](#).

Create the **main.js** file to introduce the Express framework, receive POST requests, print the request body as standard output, and return "Hello FunctionGraph, method POST" to the client.

```
const express = require('express');

const PORT = 8000;

const app = express();
app.use(express.json());

app.post('/*', (req, res) => {
  console.log('receive', req.body);
```

```
res.send('Hello FunctionGraph, method POST');
});

app.listen(PORT, () => {
  console.log('Listening on http://localhost:${PORT}');
});
```

3. Create the **package.json** file for npm so that it can identify the project and process project dependencies.

```
{
  "name": "custom-container-http-example",
  "version": "1.0.0",
  "description": "An example of a custom container http function",
  "main": "main.js",
  "scripts": {},
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

- **name:** project name
- **version:** project version
- **main:** application entry file
- **dependencies:** all available dependencies of the project in npm

4. Create a Dockerfile.

```
FROM node:12.10.0

ENV HOME=/home/custom_container
ENV GROUP_ID=1003
ENV GROUP_NAME=custom_container
ENV USER_ID=1003
ENV USER_NAME=custom_container

RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u $
{USER_ID} -g ${GROUP_ID} ${USER_NAME}

COPY --chown=${USER_ID}:${GROUP_ID} main.js ${HOME}
COPY --chown=${USER_ID}:${GROUP_ID} package.json ${HOME}

RUN cd ${HOME} && npm install

RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}

RUN find ${HOME} -type d | xargs chmod 500
RUN find ${HOME} -type f | xargs chmod 500

USER ${USER_NAME}
WORKDIR /

EXPOSE 8000
ENTRYPOINT ["node", "main.js"]
```

- **FROM:** Specify base image **node:12.10.0**. The base image is mandatory and its value can be changed.
- **ENV:** Set environment variables **HOME (/home/custom\_container)**, **GROUP\_NAME** and **USER\_NAME (custom\_container)**, **USER\_ID** and **GROUP\_ID (1003)**. These environment variables are mandatory and their values can be changed.
- **RUN:** Use the format **RUN <Command>**. For example, **RUN mkdir -m 550 \${HOME}**, which means to create the **home** directory for user **\${USER\_NAME}** during container building.

- **USER:** Switch to user `${USER_NAME}`.
- **WORKDIR:** Switch the working directory to the `/` directory of user `${USER_NAME}`.
- **COPY:** Copy `main.js` and `package.json` to the **home** directory of user `${USER_NAME}` in the container.
- **EXPOSE:** Expose port 8000 of the container. Do not change this parameter.
- **ENTRYPOINT:** Run the `node main.js` command to start the container. Do not change this parameter.

#### NOTE

1. You can use any base image.
  2. In the cloud environment, UID 1003 and GID 1003 are used to start the container by default. The two IDs can be modified by choosing **Configuration > Basic Settings > Container Image Override** on the function details page. They cannot be `root` or a reserved ID.
  3. Do not change port 8000 in the example HTTP function.
5. Build an image.

In the following example, the image name is **custom\_container\_http\_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

```
docker build -t custom_container_http_example:latest .
```

### Step 3: Perform Local Verification

1. Start the Docker container.  

```
docker run -u 1003:1003 -p 8000:8000 custom_container_http_example:latest
```
2. Open a new Command Prompt, and send a message through port 8000. You can access all paths in the root directory in the template code. The following uses **helloworld** as an example.  

```
curl -XPOST -H 'Content-Type: application/json' -d '{"message":"HelloWorld"}' localhost:8000/helloworld
```

The following information is returned based on the module code:

```
Hello FunctionGraph, method POST
```

3. Check whether the following information is displayed:

```
receive {"message":"HelloWorld"}
```

```
[root@ecs-74d7 ~]# docker run -u 1003:1003 -p 8000:8000 custom_container_http_example:latest
Listening on http://localhost:8000
receive { message: 'HelloWorld' }
```

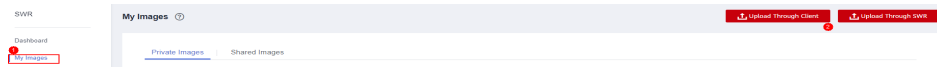
Alternatively, run the `docker logs` command to obtain container logs.

```
[root@ecs-74d7 custom_container_http_example]# docker logs 1354c3580638
Listening on http://localhost:8000
receive { message: 'HelloWorld' }
[root@ecs-74d7 custom_container_http_example]#
```

### Step 4: Upload the Image

1. Log in to the SoftWare Repository for Container (SWR) console. In the navigation pane, choose **My Images**.

2. Click **Upload Through Client** or **Upload Through SWR** in the upper right corner.
3. Upload the image as prompted.



4. View the image on the **My Images** page.

## Step 5: Create a Function

1. In the left navigation pane of the management console, choose **Compute > FunctionGraph**. On the FunctionGraph console, choose **Functions > Function List** from the navigation pane.
2. Click **Create Function** in the upper right corner and choose **Container Image**.
3. Set the basic information.
  - **Function Type:** Select **HTTP Function**.
  - **Function Name:** Enter **custom\_container\_http**.
  - **Container Image:** Select the image uploaded to SWR. Example: **swr. {Region ID}.myhuaweicloud.com/{Organization name}/{Image name}: {Image tag}**
  - **Agency:** Select an agency with the **SWR Admin** permission. If no agency is available, create one by referring to [Creating an Agency](#).
4. After the configuration is complete, click **Create Function**.

## Step 6: Test the Function

1. On the function details page, click **Test**. In the displayed dialog box, create a test event.
2. Select **apig-event-template**, set **Event Name** to **helloworld**, modify the test event as follows, and click **Create**.

```
{
  "body": "{\"message\": \"helloworld\"}",
  "requestContext": {
    "requestId": "11cdcdf33949dc6d722640a13091c77",
    "stage": "RELEASE"
  },
  "queryStringParameters": {
    "responseType": "html"
  },
  "httpMethod": "POST",
  "pathParameters": {},
  "headers": {
    "Content-Type": "application/json"
  },
  "path": "/helloworld",
  "isBase64Encoded": false
}
```


## Step 7: View the Execution Result

Click **Test** and view the execution result on the right.



**Figure 4-1** Execution result

Execution Result ✕

 Execution successful

**Function Output**

```
{
  "body": "SGVsbG8gRnVvY3Rpb25HcmFwaCwgblV0aG9kIFBPU1Q=",
  "headers": {
    "Content-Length": [
      "32"
    ],
    "Content-Type": [
      "text/html; charset=utf-8"
    ],
    "Date": [
      "Wed, 02 Nov 2022 11:06:38 GMT"
    ],
    "Etag": [
      "W/\\"20-uygbC2IEf2PxTTMC0H1BL5d/vwI\\"
    ],
    "X-Powered-By": [
      "Express"
    ]
  },
  "statusCode": 200,
  "isBase64Encoded": true
}
```

**Log Output**

```
2022-11-02T11:06:38Z Start invoke request '7309717e-f597-4368-a7fe-89ac3d9b5df5', version: latest
receive { message: 'hello world' }
2022-11-02T11:06:38Z Finish invoke request '7309717e-f597-4368-a7fe-89ac3d9b5df5', duration: 31.563ms, billing duration: 32ms, memory
used: 10.566MB, billing memory: 128MB
```

**Summary**

Request ID	7309717e-f597-4368-a7fe-89ac3d9b5df5
Memory Configured	128 MB
Execution Duration	34.247 ms
Memory Used	10.566 MB
Billed Duration	35 ms

- **Function Output:** displays the return result of the function.
- **Log Output:** displays the execution logs of the function.
- **Summary:** displays key information of the logs.

 **NOTE**

A maximum of 2 KB logs can be displayed. For more log information, see [Querying Function Logs](#).

## Step 8: View Monitoring Metrics

On the function details page, click the **Monitoring** tab.

- On the **Monitoring** tab page, choose **Metrics**, and select a time range (such as 5 minutes, 15 minutes, or 1 hour) to query the function.
- The following metrics are displayed: invocations, errors, duration (including the maximum, average, and minimum durations), and throttles.

## Step 9: Delete the Function

1. On the function details page, choose **Operation** > **Delete Function** in the upper right corner.
2. In the confirmation dialog box, enter **DELETE** and click **OK** to release resources in a timely manner.

## 4.2 Developing an Event Function

### Introduction

When developing an event function using a custom image, implement an HTTP server in the image and listen on port 8000 for requests. By default, the request path `/init` is the function initialization entry. Implement it as required. The request path `/invoke` is the function execution entry where trigger events are processed. For details about request parameters, see [Supported Event Sources](#).

### Step 1: Prepare the Environment

To perform the operations described in this section, ensure that you have the **FunctionGraph Administrator** permissions, that is, the full permissions for FunctionGraph. For more information, see [Permissions Management](#).

### Step 2: Create an Image

Take the Linux x86 64-bit OS as an example.

1. Create a folder.  
`mkdir custom_container_event_example && cd custom_container_event_example`
2. Implement an HTTP server to process **init** and **invoke** requests and give a response. Node.js is used as an example.

Create the **main.js** file to introduce the Express framework and implement a function handler (method **POST** and path `/invoke` and an initializer (method **POST** and path `/init`).

```
const express = require('express');

const PORT = 8000;

const app = express();
app.use(express.json());

app.post('/init', (req, res) => {
  console.log('receive', req.body);
  res.send('Hello init\n');
});

app.post('/invoke', (req, res) => {
  console.log('receive', req.body);
  res.send('Hello invoke\n');
});

app.listen(PORT, () => {
  console.log(`Listening on http://localhost:${PORT}`);
});
```

3. Create the **package.json** file for npm so that it can identify the project and process project dependencies.

```
{
  "name": "custom-container-event-example",
  "version": "1.0.0",
  "description": "An example of a custom container event function",
  "main": "main.js",
  "scripts": {},
  "keywords": [],
```

```
"author": "",
"license": "ISC",
"dependencies": {
  "express": "^4.17.1"
}
}
```

- **name:** project name
- **version:** project version
- **main:** application entry file
- **dependencies:** all available dependencies of the project in npm

#### 4. Create a Dockerfile.

```
FROM node:12.10.0

ENV HOME=/home/custom_container
ENV GROUP_ID=1003
ENV GROUP_NAME=custom_container
ENV USER_ID=1003
ENV USER_NAME=custom_container

RUN mkdir -m 550 ${HOME} && groupadd -g ${GROUP_ID} ${GROUP_NAME} && useradd -u ${USER_ID} -g ${GROUP_ID} ${USER_NAME}

COPY --chown=${USER_ID}:${GROUP_ID} main.js ${HOME}
COPY --chown=${USER_ID}:${GROUP_ID} package.json ${HOME}

RUN cd ${HOME} && npm install

RUN chown -R ${USER_ID}:${GROUP_ID} ${HOME}

RUN find ${HOME} -type d | xargs chmod 500
RUN find ${HOME} -type f | xargs chmod 500

USER ${USER_NAME}
WORKDIR /

EXPOSE 8000
ENTRYPOINT ["node", "main.js"]
```

- **FROM:** Specify base image **node:12.10.0**. The base image is mandatory and its value can be changed.
- **ENV:** Set environment variables **HOME (/home/custom\_container)**, **GROUP\_NAME** and **USER\_NAME (custom\_container)**, **USER\_ID** and **GROUP\_ID (1003)**. These environment variables are mandatory and their values can be changed.
- **RUN:** Use the format **RUN <Command>**. For example, **RUN mkdir -m 550 \${HOME}**, which means to create the **home** directory for user **\${USER\_NAME}** during container building.
- **USER:** Switch to user **\${USER\_NAME}**.
- **WORKDIR:** Switch the working directory to the **/** directory of user **\${USER\_NAME}**.
- **COPY:** Copy **main.js** and **package.json** to the **home** directory of user **\${USER\_NAME}** in the container.
- **EXPOSE:** Expose port 8000 of the container. Do not change this parameter.
- **ENTRYPOINT:** Run the **node /home/tester/main.js** command to start the container.

 NOTE

1. You can use any base image.
2. In the cloud environment, UID 1003 and GID 1003 are used to start the container by default. The two IDs can be modified by choosing **Configuration > Basic Settings > Container Image Override** on the function details page. They cannot be **root** or a reserved ID.
5. Build an image.

In the following example, the image name is **custom\_container\_event\_example**, the tag is **latest**, and the period (.) indicates the directory where the Dockerfile is located. Run the image build command to pack all files in the directory and send the package to a container engine to build an image.

```
docker build -t custom_container_event_example:latest .
```

### Step 3: Perform Local Verification

1. Start the Docker container.  

```
docker run -u 1003:1003 -p 8000:8000 custom_container_event_example:latest
```
2. Open a new Command Prompt, and send a message through port 8000 to access the **/init** directory specified in the template code.  

```
curl -XPOST -H 'Content-Type: application/json' localhost:8000/init
```

The following information is returned based on the module code:

```
Hello init
```
3. Open a new Command Prompt, and send a message through port 8000 to access the **/invoke** directory specified in the template code.  

```
curl -XPOST -H 'Content-Type: application/json' -d '{"message":"HelloWorld"}' localhost:8000/invoke
```

The following information is returned based on the module code:

```
Hello invoke
```
4. Check whether the following information is displayed:  

```
Listening on http://localhost:8000  
receive {}  
receive { message: 'HelloWorld' }
```

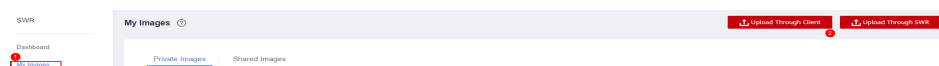
```
[root@ecs-74d7 ~]# docker run -u 1003:1003 -p 8000:8000 custom_container_event_example:latest  
Listening on http://localhost:8000  
receive {}  
receive { message: 'HelloWorld' }
```

Alternatively, run the **docker logs** command to obtain container logs.

```
[root@ecs-74d7 custom_container_event_example]# docker logs 5560e1ec09d3  
Listening on http://localhost:8000  
receive {}  
receive { message: 'HelloWorld' }  
[root@ecs-74d7 custom_container_event_example]#
```

### Step 4: Upload the Image

1. Log in to the SWR console. In the navigation pane, choose **My Images**.
2. Click **Upload Through Client** or **Upload Through SWR** in the upper right corner.
3. Upload the image as prompted.



4. View the image on the **My Images** page.

## Step 5: Create a Function

1. In the left navigation pane of the management console, choose **Compute > FunctionGraph**. On the FunctionGraph console, choose **Functions > Function List** from the navigation pane.
2. Click **Create Function** in the upper right corner and choose **Container Image**.
3. Set the basic information.
  - **Function Type**: Select **Event Function**.
  - **Function Name**: Enter **custom\_container\_event**.
  - **Container Image**: Enter the image uploaded to SWR. Example: **swr.{Region ID}.myhuaweicloud.com/{Organization name}/{Image name}:{Image tag}**
  - **Agency**: Select an agency with the **SWR Admin** permission. If no agency is available, create one by referring to [Creating an Agency](#)
4. After the configuration is complete, click **Create Function**.
5. On the function details page, choose **Configuration > Advanced Settings**, and enable **Initialization**. The **init** API will be called to initialize the function.

## Step 6: Test the Function

1. On the function details page, click **Test**. In the displayed dialog box, create a test event.
2. Select **blank-template**, set **Event Name** to **helloworld**, modify the test event as follows, and click **Create**.


```
{  
  "message": "HelloWorld"  
}
```

## Step 7: View the Execution Result

Click **Test** and view the execution result on the right.

**Figure 4-2** Execution result

Execution Result ✕

 Execution successful

**Function Output**

```
{
  "body": "SGVsbG8gRnVvY3Rpb25HcmFwaCwgblV0aG9kIFBPU1Q=",
  "headers": {
    "Content-Length": [
      "32"
    ],
    "Content-Type": [
      "text/html; charset=utf-8"
    ],
    "Date": [
      "Wed, 02 Nov 2022 11:06:38 GMT"
    ],
    "Etag": [
      "W/\"20-uygbC2IEf2PxTTMC0H1BL5d/vwI\""
    ],
    "X-Powered-By": [
      "Express"
    ]
  },
  "statusCode": 200,
  "isBase64Encoded": true
}
```

**Log Output**

```
2022-11-02T11:06:38Z Start invoke request '7309717e-f597-4368-a7fe-89ac3d9b5df5', version: latest
receive { message: 'hello world' }
2022-11-02T11:06:38Z Finish invoke request '7309717e-f597-4368-a7fe-89ac3d9b5df5', duration: 31.563ms, billing duration: 32ms, memory
used: 10.566MB, billing memory: 128MB
```

**Summary**

Request ID	7309717e-f597-4368-a7fe-89ac3d9b5df5
Memory Configured	128 MB
Execution Duration	34.247 ms
Memory Used	10.566 MB
Billed Duration	35 ms

- **Function Output:** displays the return result of the function.
- **Log Output:** displays the execution logs of the function.
- **Summary:** displays key information of the logs.

 **NOTE**

A maximum of 2 KB logs can be displayed. For more log information, see [Querying Function Logs](#).

## Step 8: View Monitoring Metrics

On the function details page, click the **Monitoring** tab.

- On the **Monitoring** tab page, choose **Metrics**, and select a time range (such as 5 minutes, 15 minutes, or 1 hour) to query the function.
- The following metrics are displayed: invocations, errors, duration (including the maximum, average, and minimum durations), and throttles.

## Step 9: Delete the Function

1. On the function details page, choose **Operation** > **Delete Function** in the upper right corner.
2. In the confirmation dialog box, enter **DELETE** and click **OK** to release resources in a timely manner.

# 5 Getting Started with Common Practices

After understanding basic operations such as creating a function, you can follow FunctionGraph's common practices to implement your services.

This section describes the common practices that can help you better use FunctionGraph.

**Table 5-1** Common practices

Practice	Description
<a href="#">Compressing Images</a>	Use FunctionGraph to compress images.
<a href="#">Watermarking Images</a>	Use FunctionGraph to watermark images.
<a href="#">Integrating with LTS to Analyze Logs in Real Time</a>	<ul style="list-style-type: none"><li>• Use Log Tank Service (LTS) to collect, process, and convert task logs of servers, such as Elastic Cloud Servers (ECSs).</li><li>• Obtain log data with an LTS trigger, analyze and process key information in logs using a custom function, and filter alarm logs.</li><li>• Use Simple Message Notification (SMN) to push alarm messages to service personnel by SMS or email.</li><li>• Store processed log data in a specified OBS bucket for processing.</li></ul>

Practice	Description
<b>Integrating with CTS to Analyze Login/Logout Security</b>	<ul style="list-style-type: none"><li>• Use Cloud Trace Service (CTS) to collect real-time records of cloud resource operations.</li><li>• Obtain operation records of subscribed cloud resources with a CTS trigger, analyze and process the records using a custom function, and report alarms.</li><li>• Use SMN to push alarm messages to service personnel by SMS or email.</li></ul>
<b>Periodically Starting or Stopping Huawei Cloud ECSs</b>	<p>To start or stop your ECSs at specified time, use FunctionGraph to call the corresponding ECS APIs.</p> <ul style="list-style-type: none"><li>• Nodes to start: VMs that need to be started periodically.</li><li>• Nodes to stop: VMs that need to be stopped periodically.</li></ul>
<b>Processing IoT Data</b>	<ul style="list-style-type: none"><li>• Use FunctionGraph and IoT Device Access (IoTDA) to process status data reported by IoT devices. IoT devices are managed on the IoTDA platform. Their data is transferred from IoTDA to trigger the FunctionGraph functions you have compiled for processing.</li><li>• This combination is suitable for processing device data and storing them in OBS, structuring and cleansing data and storing them in a database, and sending event notifications for device status changes.</li></ul>
<b>Workflow + Function: Automatically Processing Data in OBS</b>	Use FunctionGraph to process OBS data.