01 Getting Started

Issue 01

Date 2025-09-11

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: https://www.huaweicloud.com/intl/en-us/

Contents

1 Getting Started with Common Practices

Operation Guide

You can create a function and deploy code in either of the following ways:

- FunctionGraph console: It allows you to quickly create executable functions on the GUI, which is suitable for building lightweight applications and quick verification. For details, see **Creating a Function from Scratch and Executing the Function**.
- VS Code: Based on the Huawei Cloud FunctionGraph VS Code plug-in, you can easily create and execute functions in the local code editor. It is suitable for small teams and individual developers. For details, see **Local Debugging with VSCode**.
- Serverless Devs: You can manage multiple functions using the YAML configuration file based on the open-source CLI tool. It is suitable for complex projects with many functions or automatic deployment. For details, see **Introduction**.
- Serverless Framework: The Serverless Framework V3 uses the YAML configuration file to maintain the full lifecycle of functions, which is suitable for users familiar with the tool. For details, see **Quick Start**.

Best Practices

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

This section describes common practices of FunctionGraph. For details, see **FunctionGraph Best Practices**.

Table 1-1 Common practices

Practice	Description
Using FunctionGraph to Compress Images in OBS	Use FunctionGraph to compress images.
Using FunctionGraph to Watermark Images in OBS	Use FunctionGraph to watermark images.

Practice	Description
Building an HTTP Function Using an Existing Spring Boot Project	Deploy a Spring Boot application as an HTTP function on FunctionGraph.
Integrating with LTS to Analyze Logs in Real Time	• Use Log Tank Service (LTS) to collect, process, and convert task logs of servers, such as Elastic Cloud Servers (ECSs). • Obtain log data with an LTS trigger, analyze and process key information in logs using a custom function, and filter alarm logs. • Use Simple Message Notification (SMN) to push alarm messages to service personnel by SMS or email. • Store processed log data in a specified OBS bucket for processing.
Integrating with CTS to Analyze Login/Logout Security	• Use Cloud Trace Service (CTS) to collect real-time records of cloud resource operations. • Obtain operation records of subscribed cloud resources with a CTS trigger, analyze and process the records using a custom function, and report alarms. • Use SMN to push alarm messages to service personnel by SMS or email.
Processing IoT Data	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. 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.
Workflow + Function: Automatically Processing Data in OBS	Use FunctionGraph to process OBS data.

Practice	Description
Using FunctionGraph to Deploy Stable Diffusion for AI Drawing	Deploy Stable-Diffusion applications in the application center of FunctionGraph and provides multiple methods for customizing AI drawing applications.

2 Creating a Function from Scratch and Executing the Function

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

Prerequisites

1. View free quota.

   FunctionGraph offers a free tier every month, which you can share with your IAM users. For details, see **Free Tier**

   If you continue to use FunctionGraph after the free quota is used up, your account goes into arrears if the balance is less than the bill to be settled. To continue using your resources, top up your account in time. For details, see **Topping Up an Account**.

2. Grant the FunctionGraph operation permissions to the user.

   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 1: Create a Function

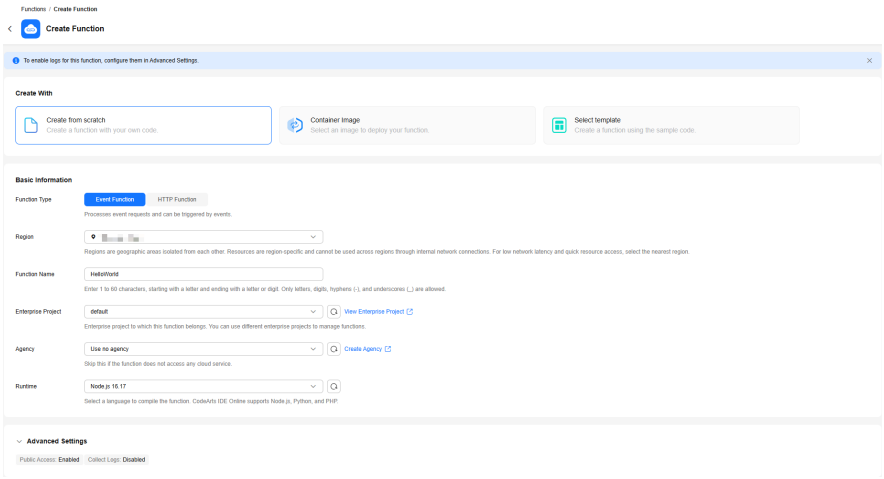
1. Log in to the **FunctionGraph console**. In the navigation pane, choose **Functions** > **Function List**.
2. Click **Create Function** in the upper right corner and choose **Create from scratch**.
3. Configure basic function information by referring to **Figure 2-1**. Then, click **Create Function**.

Table 2-1 Parameters required for creating a function

Parameter	Description	Example Value
Function Type	Select **Event Function**. An event function is triggered by a specific event, which is usually a request event in JSON format.	Event Function
Region	Select the region where the function is located. Regions are geographic areas isolated from each other. Resources are region-specific and cannot be used across regions through internal network connections. Select a region near you to ensure the lowest latency possible.	CN East-Shanghai1
Function Name	Enter a function name. The naming rules are as follows: • Consists of 1 to 60 characters, and can contain letters, digits, hyphens (-), and underscores (_). • Start with a letter and end with a letter or digit.	HelloWorld
Enterprise Project	Select the enterprise project to which the function belongs. Enterprise projects let you manage cloud resources and users by project. The default value is **default**. You can select the created enterprise project. **If the Enterprise Management service is not enabled, this parameter is unavailable. For details, see Enabling the Enterprise Project Function.**	default
Agency	Select an agency for the function. An agency is used to authorize FunctionGraph to access other cloud services. If FunctionGraph does not access any cloud service, you do not need to select an agency. By default, **Use no agency** is used. You can select an existing agency. If no default agency is available, FunctionGraph allows you to quickly create a default agency named **fgs_default_agency**..	Use no agency

Parameter	Description	Example Value
Permission Policies	**This parameter is displayed only when an agency is selected.** For details about how to adjust the permission policies on the IAM console, see **Modifying an Agency**.	-
Runtime	Select a runtime to compile the function. For details about the runtimes supported see **Supported Runtimes**. • CloudIDE supports only online editing of Node.js, Python, PHP, and custom runtimes. • After a function is created, the runtime language cannot be changed.	Node.js 16.17
Advanced Settings	• **Public Access**: If this feature is enabled, functions can access services on the public network through the default NIC. The public network access bandwidth is shared among users and applies only to test scenarios. • **VPC Access**: If this feature is enabled, functions will use the NIC bound to the configured VPC for network access, and the default NIC of FunctionGraph will be disabled. That is, the **Public Access** parameter does not take effect. **To enable this feature, you need to configure an agency with VPC management permissions. If you select Use no agency in the Basic Information area, this feature cannot be enabled.** • **Collect Logs**: After it is enabled, logs generated during function execution will be reported to LTS. LTS will be billed on a pay-per-use basis. For details, see **LTS Pricing Details**. • **Static Encryption with KMS** (supported only in the **LA-Sao Paulo1** region): Select whether to use KMS static code encryption. DEW is billed by the pay-per-use mode. For details, see **DEW Billing**.	• **Public Access**: Enabled. • **VPC Access**: Disabled. • **Collect Logs**: Disabled. • **Static Encryption with KMS**: Disabled.

Figure 2-1 Creating a function



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.

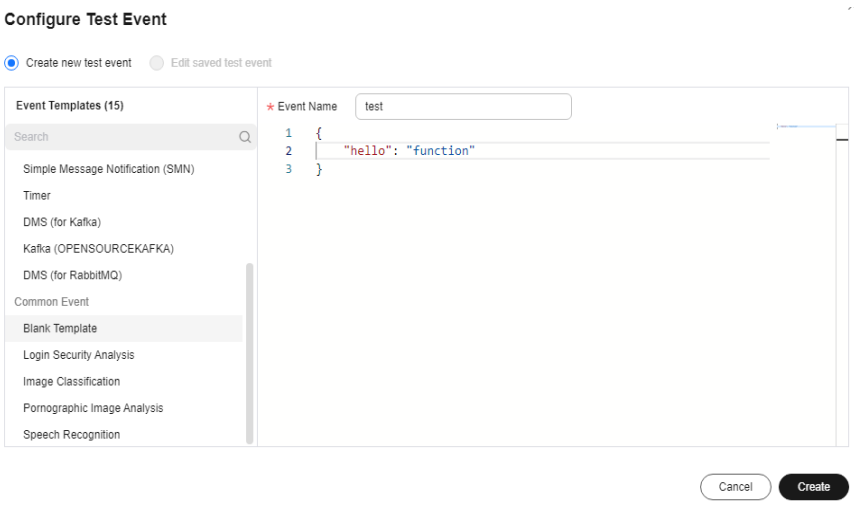
```
exports.handler = function (event, context, callback) {
    const error = null;
    const output = `Hello message: ${JSON.stringify(event)}`;
    callback(error, output);
}
```

Step 2: 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

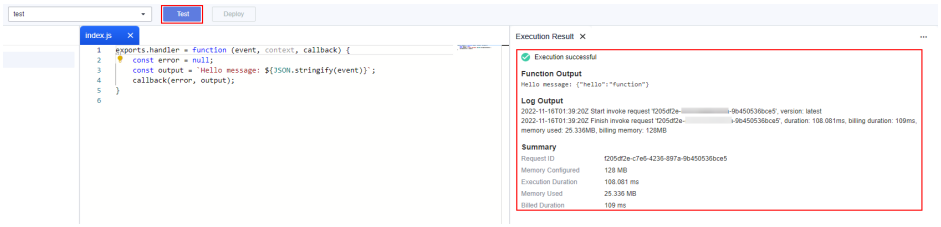


Step 3: View the Execution Result

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**.

Related Information

- For details about FunctionGraph concepts, see **Concepts**.
- For details about FunctionGraph pricing, see **FunctionGraph Pricing Details**.
- For details about the constraints and limitations of FunctionGraph, see **Notes and Constraints**.

3 Creating a Function Using a Template and Executing the Function

FunctionGraph provides multiple templates to automatically complete code and running environment configurations when you create a function, helping you quickly build applications. This section uses **context-class-introduction** as an example.

Prerequisites

1. View free quota.

   FunctionGraph offers a free tier every month, which you can share with your IAM users. For details, see **Free Tier**

   If you continue to use FunctionGraph after the free quota is used up, your account goes into arrears if the balance is less than the bill to be settled. To continue using your resources, top up your account in time. For details, see **Topping Up an Account**.

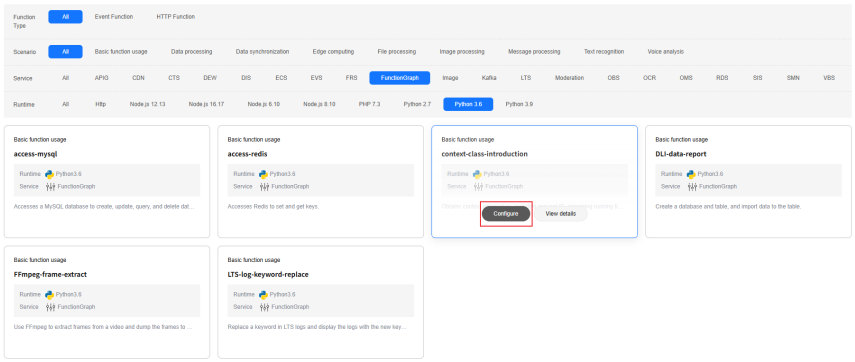
2. Grant the FunctionGraph operation permissions to the user.

   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 1: 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



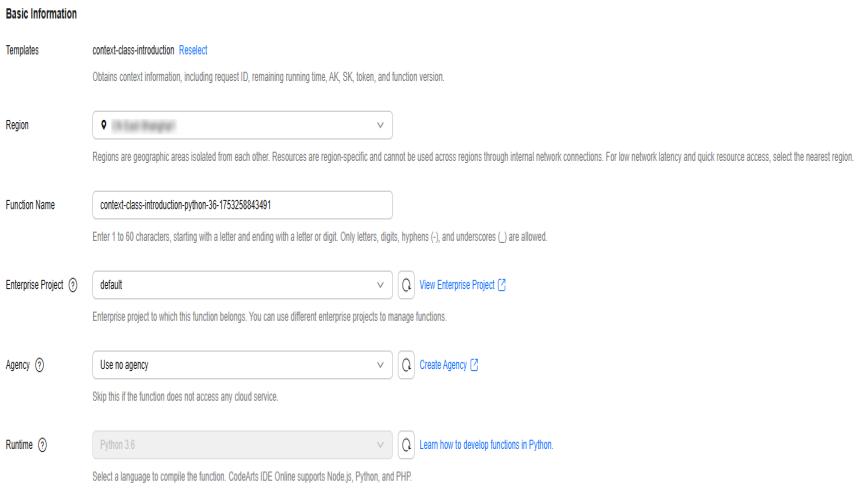
4. On the displayed page, set function parameters and click **Create Function**.
   - **Templates**: name of the selected template. To change the template, click **Reselect** on the right.
   - **Region**: The default value is used. You can select other regions.

     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 **context**.
   - **Enterprise Project**: The default value is **default**. You can select the created enterprise project.

     Enterprise projects let you manage cloud resources and users by project.
   - **Agency**: By default, no agency is used. You can select an existing agency.

     Specify an agency if you want to delegate FunctionGraph to access other cloud services, such as LTS and VPC.
   - **Runtime**: Select a runtime to compile the function. Default: **Python 3.6**. You can select another runtime.

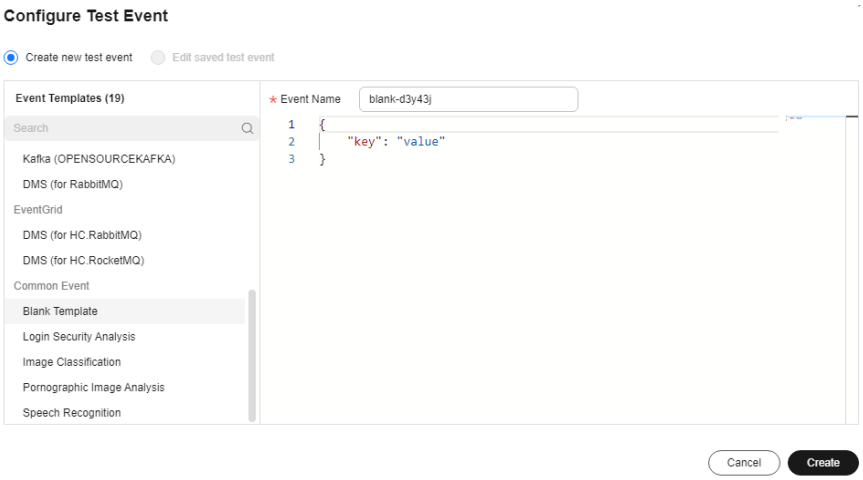
Figure 3-2 Setting basic information



Step 2: 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 3-3 Configuring a test event



Step 3: 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**.

Related Information

- For details about FunctionGraph concepts, see **Concepts**.
- For details about FunctionGraph pricing, see **FunctionGraph Pricing Details**.
- For details about the constraints and limitations of FunctionGraph, see **Notes and Constraints**.

4 Creating an HTTP Function Using a Container Image and Executing the Function

This section uses the creation of an HTTP function using a container image as an example to describe how to create and test a container image function.

In this example, 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.)** Note: You can use any base image.

Constraints

- HTTP functions support only APIC/APIG triggers.
- 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.
- **Do not change port 8000 in the example HTTP function.**
- If the basic image of the Alpine version is used, run the **addgroup** and **adduser** commands.

Prerequisites

1. View free quota.

   FunctionGraph offers a free tier every month, which you can share with your IAM users. For details, see **Free Tier**

   If you continue to use FunctionGraph after the free quota is used up, your account goes into arrears if the balance is less than the bill to be settled. To continue using your resources, top up your account in time. For details, see **Topping Up an Account**.

2. Grant the FunctionGraph operation permissions to the user.

   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 1: Create an Image

Take the Linux x86 64-bit OS as an example. (No system configuration is required.)

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 *FunctionGraph Developer Guide*.

   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 ${HOME}

EXPOSE 8000
ENTRYPOINT ["node", "/home/custom_container/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 *${USER_NAME}*.
- **WORKDIR**: Switch the working directory to the **${HOME}** 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.

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 2: 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 3: 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 4: 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. On the displayed page, select **Container Image** for creation mode.
3. Set the basic function information.
   - **Function Type**: Select **HTTP Function**.
   - **Region**: The default value is used. You can select other regions.

     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 **custom_container_http**.
   - **Enterprise Project**: The default value is **default**. You can select the created enterprise project.

     Enterprise projects let you manage cloud resources and users by project.
   - **Agency**: Select an agency with the **SWR Admin** permission. If no agency is available, create one by referring to **Creating an Agency**.
   - **Container Image**: Enter the image uploaded to SWR in **step 3**. Example: **swr.***{Region ID}***.myhuaweicloud.com/***{Organization name}***/***{Image name}***:***{Image tag}*
4. (Optional) Set container image overriding parameters.
   - **CMD**: container startup command. Example: **/bin/sh**. If no command is specified, the entrypoint or CMD in the image configuration will be used.
   - **Args**: container startup parameter. Example: **-args,value1**. If no argument is specified, CMD in the image configuration will be used.
   - **Working Dir**: working directory where a container runs. If no directory is specified, the directory in the image configuration will be used. The directory must start with a slash (/).

- **User ID**: Enter the user ID.
- **Group ID**: Enter the user group ID.

5. After the configuration is complete, click **Create Function**.

Step 5: Test the Function

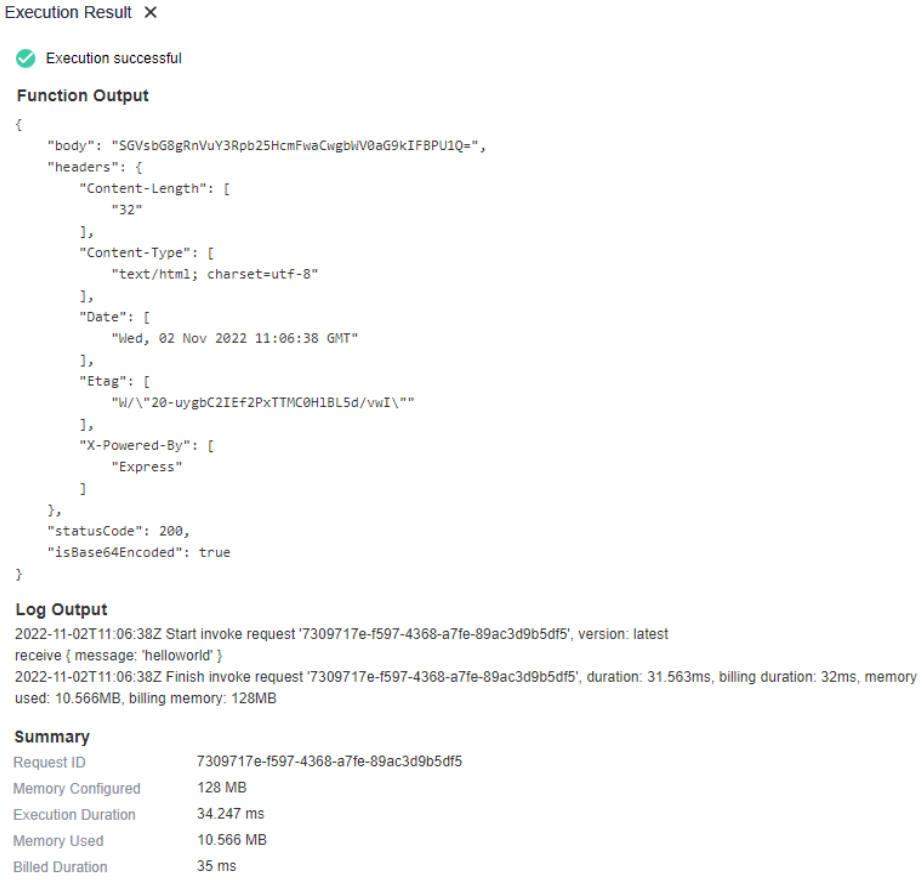
1. On the function details page, click **Test**. In the displayed dialog box, create a test event.
2. Select **apig-event-template**, set **Event Name** to **helloworld**, modify the test event as follows, and click **Create**.

```
{
    "body": "{\"message\": \"helloworld\"}",
    "requestContext": {
        "requestId": "11cdcdcf33949dc6d722640a13091c77",
        "stage": "RELEASE"
    },
    "queryStringParameters": {
        "responseType": "html"
    },
    "httpMethod": "POST",
    "pathParameters": {},
    "headers": {
        "Content-Type": "application/json"
    },
    "path": "/helloworld",
    "isBase64Encoded": false
}
```

Step 6: View the Execution Result

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

Figure 4-1 Execution result



- **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**.

Related Information

- For details about FunctionGraph concepts, see **Concepts**.
- For details about FunctionGraph pricing, see **FunctionGraph Pricing Details**.
- For details about the constraints and limitations of FunctionGraph, see **Notes and Constraints**.

5 Creating an Event Function Using a Container Image and Executing the Function

This section uses the creation of an event function using a container image as an example to describe how to create and test a container image function.

You need to implement an HTTP server in the image and listen to port **8000** to receive 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**. Note: You can use any base image.

Constraints

- 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.
- If the base image of the Alpine version is used, run the **addgroup** and **adduser** commands.

Prerequisites

1. View free quota.

   FunctionGraph offers a free tier every month, which you can share with your IAM users. For details, see **Free Tier**

   If you continue to use FunctionGraph after the free quota is used up, your account goes into arrears if the balance is less than the bill to be settled. To continue using your resources, top up your account in time. For details, see **Topping Up an Account**.

2. Grant the FunctionGraph operation permissions to the user.

   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 1: Create an Image

Take the Linux x86 64-bit OS as an example. (No system configuration is required.)

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 ${HOME}

EXPOSE 8000
ENTRYPOINT ["node", "/home/custom_container/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 *${USER_NAME}*.
- **WORKDIR**: Switch the working directory to the **${HOME}** 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.

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 2: 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 3: 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 4: 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. On the displayed page, select **Container Image** for creation mode.
3. Set the basic function information.
   - **Function Type**: Select **Event Function**.
   - **Region**: The default value is used. You can select other regions.

     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 **custom_container_http**.
   - **Enterprise Project**: The default value is **default**. You can select the created enterprise project.

     Enterprise projects let you manage cloud resources and users by project.
   - **Agency**: Select an agency with the **SWR Admin** permission. If no agency is available, create one by referring to **Creating an Agency**.

- **Container Image**: Enter the image uploaded to SWR in **step 3**. Example: **swr.***{Region ID}***.myhuaweicloud.com/***{Organization name}***/***{Image name}***:***{Image tag}*

4. (Optional) Set container image overriding parameters.
   - **CMD**: container startup command. Example: **/bin/sh**. If no command is specified, the entrypoint or CMD in the image configuration will be used.
   - **Args**: container startup parameter. Example: **-args,value1**. If no argument is specified, CMD in the image configuration will be used.
   - **Working Dir**: working directory where a container runs. If no directory is specified, the directory in the image configuration will be used. The directory must start with a slash (/).
   - **User ID**: Enter the user ID.
   - **Group ID**: Enter the user group ID.
5. After the configuration is complete, click **Create Function**.
6. On the function details page, choose **Configuration** > **Lifecycle**, and enable **Initialization**. The **init** API will be called to initialize the function.

Step 5: Test the Function

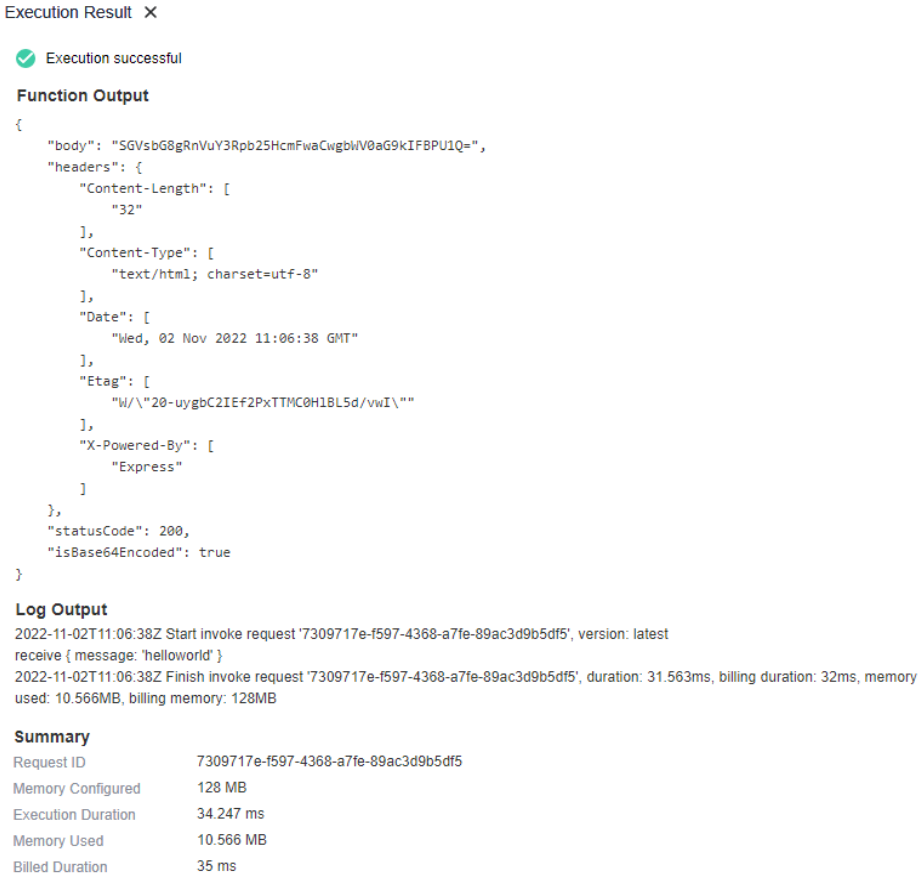
1. On the function details page, click **Test**. In the displayed dialog box, create a test event.
2. Select **blank-template**, set **Event Name** to **helloworld**, modify the test event as follows, and click **Create**.

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

Step 6: View the Execution Result

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

Figure 5-1 Execution result



- **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**.

Related Information

- For details about FunctionGraph concepts, see **Concepts**.
- For details about FunctionGraph pricing, see **FunctionGraph Pricing Details**.
- For details about the constraints and limitations of FunctionGraph, see **Notes and Constraints**.

