Serverless Application Center

Advanced Guide

Product Documentation

Contents

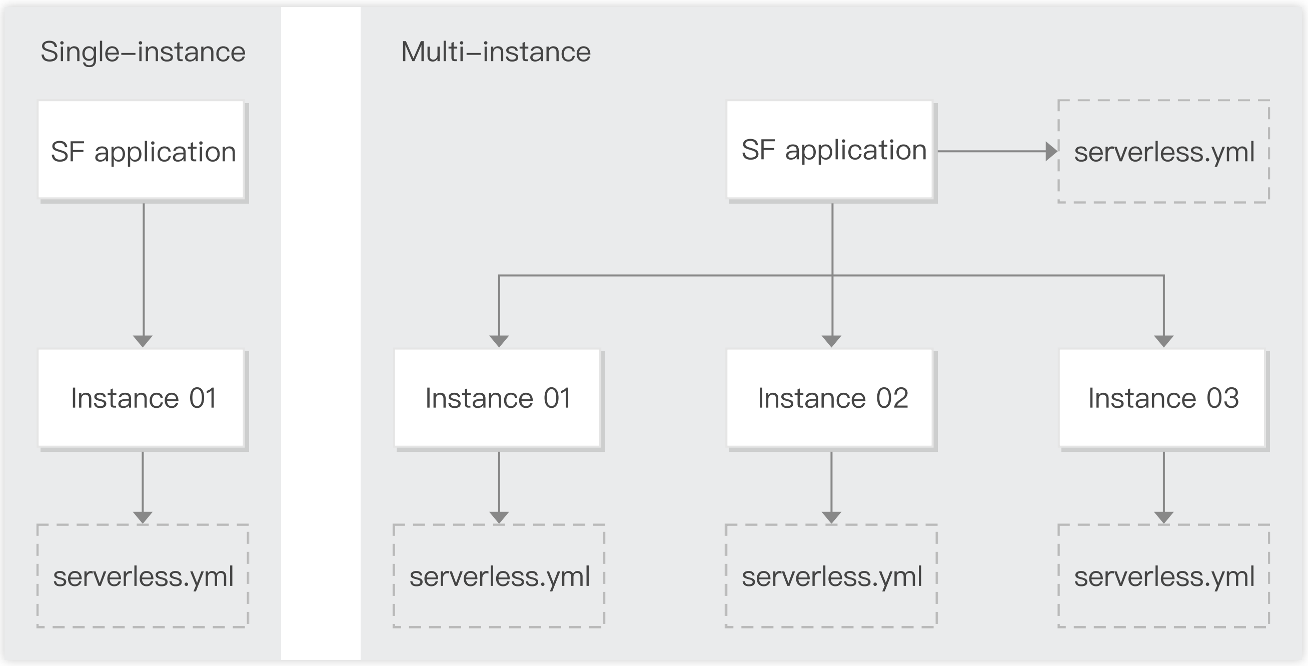
Advanced Guide

Application Management

Last updated：2024-12-02 10:48:10

Overview

Every time `sls deploy` is run, a serverless application will be deployed, which consists of one or multiple component instances, and each component corresponds to an instance.
Each instance involves a `serverless.yml` file, which defines certain parameters of the component. Such parameters are used to generate the instance information during deployment; for example, `region` defines the resource region.
The differences between the project organizations of a single-instance application and a multi-instance application are as shown below:



Single-Instance application

In the project of a single-instance application, only one component is imported, and only one component instance will be generated during deployment.
Generally, you do not need to manually name a single-instance application. If you want to customize the name, you can directly enter a name in `serverless.yml` of the component.

Multi-Instance application

In the project of a multi-instance application, multiple components are imported, and multiple component instances will be generated during deployment.
You need to enter a custom name for the multi-instance application to ensure that all components are managed under the same application. Generally, the application name is defined in `serverless.yml` in the project directory, so that all components can use the same application name.

serverless.yml file

The `serverless.yml` file defines the application organization parameters and the component's `inputs` parameters. During each deployment, resources will be created, updated, and orchestrated according to the configuration information in the `serverless.yml` file.
The following is a simple `serverless.yml` file:

```
# serverless.yml

# Application information
app: expressDemoApp # Application name, which is the component instance name by def
stage: ${env:STAGE} # Parameter used to isolate the development environment, which
# Component information
component: express # Name of the imported component, which is required. The `expres
name: expressDemo # Name of the instance created by the component, which is require

# Component configuration
inputs:
  src:
    src: ./
    exclude:
      - .env
  region: ap-guangzhou
  runtime: Nodejs10.15
  functionName: ${name}-${stage}-${app} # Function name
  apigatewayConf:
    protocols:
      - http
      - https
    environment: release
```

Configuration information in the `.yml` file:

Application information

Parameter	Description
org	Organization information, which is the `APPID` of your Tencent Cloud account by default. It is

	a reserved field and is not recommended to be used.
app	Application name, which is the same as the instance name in the component information by default. A single-instance application and a multi-instance application have different definitions of this parameter. For more information, please see Application Management.
stage	Environment information, which is `dev` by default. You can define different `stage` values to provide independent runtime environments for development, testing, and release of the serverless application, respectively. For more information, please see Project Development.

Component information

Parameter	Description
component	Name of the imported component. You can run `sls registry` to query components available for import.
name	Name of the created instance. An instance will be created when each component is deployed.

Parameter information

The parameters under `inputs` are configuration parameters of the corresponding component. Different components have different parameters. To guarantee environment isolation and resource uniqueness, the component resource names are in the `${name}-${stage}-${app}` format by default.

Application Deployment

Single-Instance application

Do not set the application name (`app`) in the `serverless.yml` file, and an application with the same name as that of the instance (`name`) will be generated by default during deployment.

For example, if you create an SCF project, and the project directory is as shown below:

```
scfDemo
 |- index.js
 └── serverless.yml
```

Here, the `serverless.yml` file is configured as follows:

```
component: scf
name: myscf

inputs:
  src: ./
  runtime: CustomRuntime
  region: ap-guangzhou
  functionName: ${name}-${stage}-${app} # Function name

  events:
    - apigw:
        parameters:
          endpoints:
            - path: /
              method: GET
```

Run `sls deploy` in the `scfDemo` directory for deployment, and an application whose `app` is `myscf` will be generated by default, and the application will contain an SCF instance named `myscf` .
Generally, you can use the default application name for a single-instance application project. If you want to customize the name, you can directly enter a name in `serverless.yml` as follows:

```
app: scfApp # Set `app` to `scfApp`

component: scf
name: myscf

inputs:
  src: ./
  runtime: CustomRuntime
  region: ap-guangzhou

  events:
    - apigw:
        parameters:
          endpoints:
            - path: /
              method: GET
```

Run `sls deploy` in the `scfDemo` directory for deployment, and an application whose `app` is `scfApp` will be generated, and the application will contain an SCF instance named `myscf` .

Multi-Instance application

As the project contains multiple components, you need to unify the application name for all components. Generally, you should define a `serverless.yml` file in the root directory of the project to name the application.
For example, if you deploy a Vue.js + Express.js + PostgreSQL full-stack website, and the project directories are as shown below:

```
fullstack
 |- api
 |   |- sls.js
```

```
  |    |- ...
  |        └── serverless.yml
  |- db
  |        └── serverless.yml
  |- frontend
  |    |- ...
  |        └── serverless.yml
  |- vpc
  |        └── serverless.yml
  |- scripts
  └── serverless.yml
```

`app` is set in the `serverless.yml` file in the `fullstack` directory of the project:

```
# Project application information

app: fullstack
```

The component and parameter information is configured in the `serverless.yml` file in each component directory, such as the `api` directory:

```
# `api` configuration information

component: express
name: fullstack-api

inputs:
  src:
    src: ./
    exclude:
      - .env
  functionName: ${name}-${stage}-${app}
  region: ${env:REGION}
  runtime: Nodejs10.15
  functionConf:
    timeout: 30
    vpcConfig:
      vpcId: ${output:${stage}:${app}:fullstack-vpc.vpcId}
      subnetId: ${output:${stage}:${app}:fullstack-vpc.subnetId}
    environment:
      variables:
        PG_CONNECT_STRING: ${output:${stage}:${app}:fullstack-db.private.connection
  apigatewayConf:
    enableCORS: true
    protocols:
      - http
      - https
```

Note:

In the demo on the legacy version, the application name (`app`) is written into all components, which requires you to ensure that all components under the project have the same application name. We recommend you not use this method in the new version.

Project Development

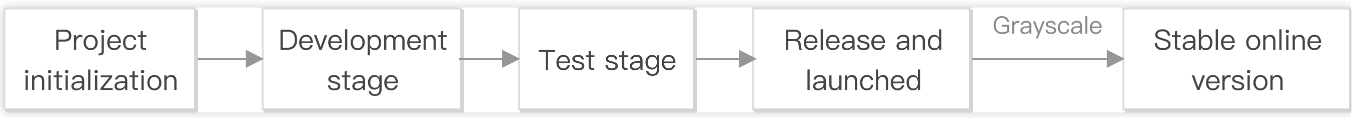
Last updated：2024-12-02 10:48:10

Prerequisites

You have understood how to quickly deploy a project.
You have understood serverless applications.
You have understood account and permission configuration.

Development Process

The development and launch process of a project is as shown below:



1. Project initialization: initialize the project; for example, select some development frameworks and templates to complete the basic construction.
2. Development: develop product features. This stage may involve collaboration among multiple developers, who will pull different feature branches for separated development and testing and finally merge them into the `dev` branch for joint testing.
3. Testing: test the product features by testing personnel.
4. Release and launch: publish and launch the tested product features. As a newly published version may be unstable, grayscale release will be used generally, and some rules will be configured to monitor the stability of the new version. After the new version becomes stable, all traffic will be switched to it.

Environment Isolation

During each stage of project development, an environment running independently is required to isolate the development operations.
Define `stage` in the `serverless.yml` file and write `stage` into the component's resource names as a parameter, so that resources named `instance name-{stage}-application name` will be generated during the deployment. In this way, you can generate different resources in different stages by defining different `stage` values so as to isolate the environments.

Take `serverless.yml` of the SCF component as an example:

```
# Application information
app: myApp
stage: dev # The environment is defined as `dev`

# Component information
component: scf
name: scfdemo

# Component parameters
inputs:
  name: ${name}-${stage}-${app} # Function name. The `${stage}` variable is used as
  src: ./
  handler: index.main_handler
  runtime: Nodejs10.15
  region: ap-guangzhou
  events:
    - apigw:
        parameters:
          endpoints:
            - path: /
              method: GET
```

Define the function `name` as `${name}-${stage}-${app}` .

Define `stage` as `dev` for the development and testing stages. After the deployment, the function will be named `scfdemo-dev-myApp` .

Define `stage` as `pro` for the release stage. After the deployment, the function will be named `scfdemo-pro-myApp` .

Manipulate different function resources in different stages so as to isolate the development from release.

Note:

You can directly define `stage` in the `serverless.yml` file or pass in the parameter through `sls deploy --stage dev` .

Permission Management

During project development, you need to assign permissions to different persons and manage their permissions; for example, you want a developer to access only a certain environment in a project. To this end, you can grant sub-accounts permissions to manipulate specified resources in Serverless Framework as instructed in Account and Permission Configuration.

Taking the `dev` environment of the `myApp` project as an example, the configuration is as follows:

```
{
    "version": "2.0",
    "statement": [
        {
            "action": [
                "sls:*"
            ],
            "resource": "qcs::sls:ap-guangzhou::appname/myApp/stagename/dev", # `ap
            "effect": "allow"
        }
    ]
}
```

Grayscale Release

Grayscale release (aka canary release) is a release method that can smoothly transition between black and white. To guarantee the stability of your business in the production environment, we recommend you use grayscale release to launch projects in the production environment.
Grayscale release for serverless applications is applicable only to the SCF component and relevant components involving SCF.
You can configure the traffic rule of the SCF function whose alias is `$default` (default traffic) to configure the traffic of two function versions, where one is the `$latest` version of the function, while the other is the last published version. For more information, please see Grayscale Release.

Serverless Framework Commands

From project development to release, you need to use relevant Serverless Framework commands. For more commands, please see List of Supported Commands.

```
# Initialize the project
sls

# Download the template project `scf-demo`. You can run `sls registry` to query the
sls init scf-demo

# Download the template project `scf-demo` and initialize it as `myapp`
sls init scf-demo --name my-app

# Deploy the application
sls deploy
```

```
# Deploy the application and specify `stage` as `dev`
sls deploy --stage  dev

# Deploy the application and print the deployment information
sls deploy --debug

# Deploy and publish the function version
sls deploy --inputs publish=trues

# Deploy and switch 20% traffic to the `$latest` version
sls deploy --inputs traffic=0.2
```

Project Practice

For more information, please see Developing and Launching Serverless Application.

Grayscale Release

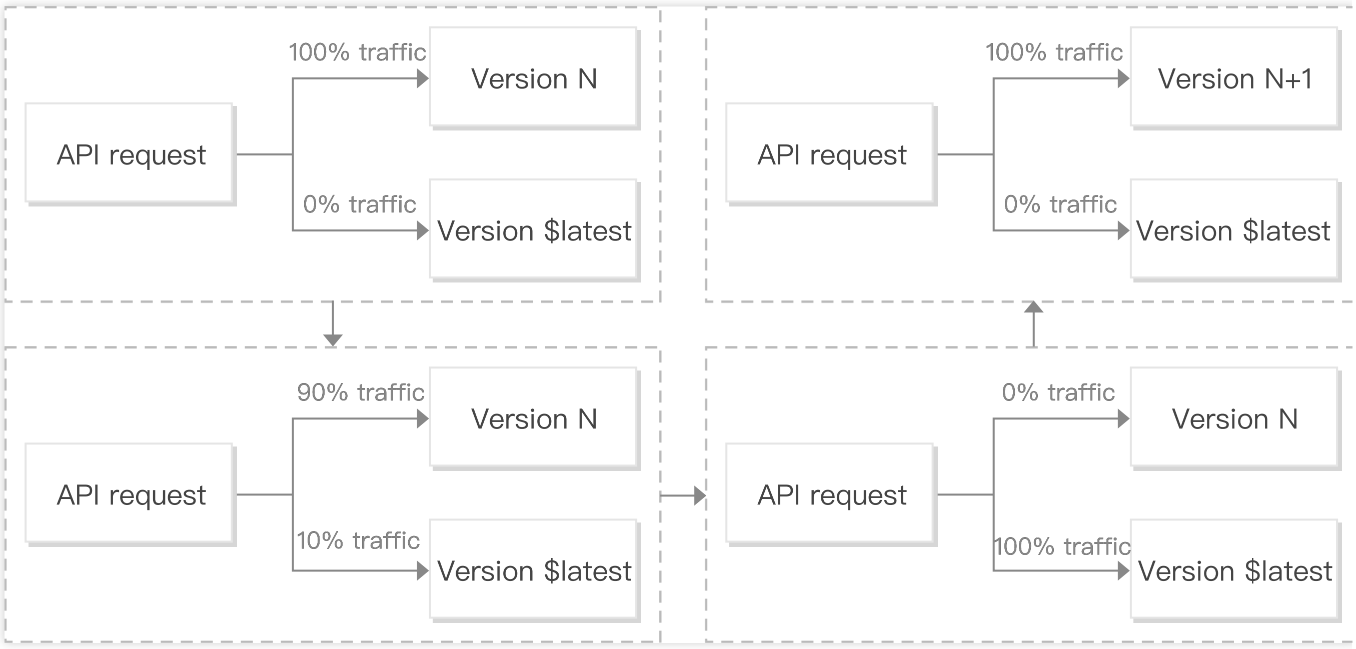
Last updated：2024-12-17 16:03:20

Overview

Grayscale release (aka canary release) is a release method that can smoothly transition between black and white. The grayscale release utilized for a serverless application is to configure traffic rules of the SCF function alias, i.e., configuring traffic rules for the two different versions of the function in the alias. Serverless Framework supports two ways to configure alias: **default alias** and **custom alias**.

Default Alias

The default alias is to configure the `$default` (default traffic) alias for the function. This alias always contains two function versions: one is the `$latest` version of the function, while the other is the last published version. During the deployment, the `traffic` parameter specifies the traffic percentage on the `$latest` version, while the rest traffic will be switched to the last published version of the current function by default.
Every time a feature is published, you can run `sls deploy` to deploy it onto the `$latest` version. You can switch some traffic to the `$latest` version to check the performance and gradually switch the rest traffic to it. When 100% traffic has been switched to it, it will be fixed, and all traffic will be switched to the fixed version.



Command description

Note:

The legacy command format `sls deploy --inputs.key=value` has been changed to `sls deploy --inputs key=value` since Serverless CLI v3.2.3. Legacy commands cannot be used in new versions of Serverless CLI. If you have upgraded Serverless CLI, please use the new commands.

Publishing function version

Publish all function versions under the project during deployment:

```
sls deploy --inputs publish=true
```

Setting function traffic

After deployment, switch 20% traffic to the `$latest` version.

```
sls deploy --inputs traffic=0.2
```

In Serverless Framework, the traffic rule of the SCF function whose alias is `$default` is modified to switch the traffic.
The objects to be configured are always the `$latest` version and the last published function version.
The value of `traffic` is configured as the traffic percentage of the `$latest` version. The traffic percentage of the last published SCF function version is 1 minus the traffic percentage of the `$latest` version (for example, if traffic=0.2, the traffic rule of `$default` will be `{$latest:0.2, last published function version: 0.8}`).
If no fixed versions are published for the function and only the `$latest` version exists, no matter how `traffic` is set, it will always be `$latest:1.0` .

Directions

You can publish a tested feature through grayscale release in the following steps:
1. Configure the production environment information into the .env file (`STAGE=prod` indicates the production environment):

```
TENCENT_SECRET_ID=xxxxxxxxxx
TENCENT_SECRET_KEY=xxxxxxxx
STAGE=prod
```

2. Deploy the function to the `$latest` production environment and switch 10% traffic to the `$latest` version (90% traffic will be switched to the last published function version N):

```
sls deploy --inputs traffic=0.1
```

3. Monitor the `$latest` version and switch 100% traffic to this version after it becomes stable:

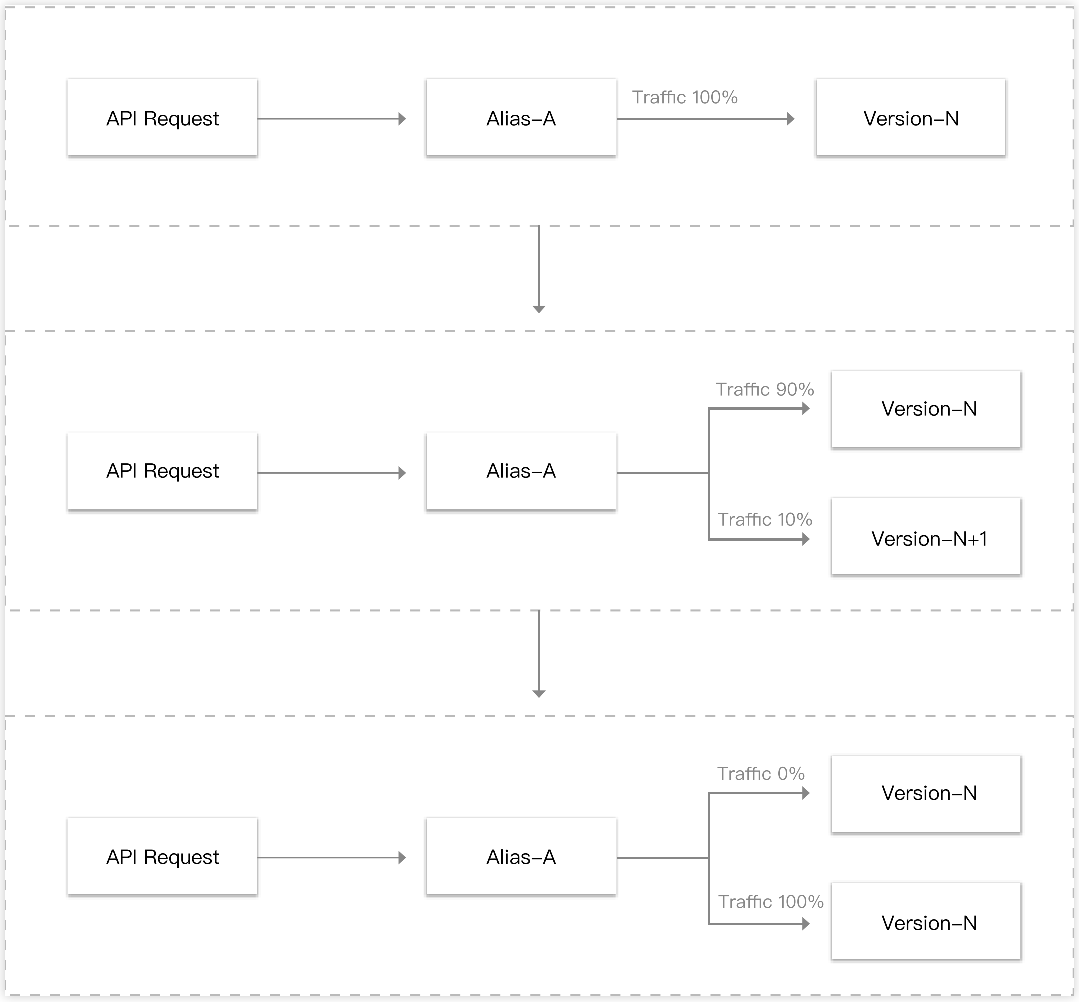
```
sls deploy --inputs traffic=1.0
```

4. After all traffic is successfully switched, the stable version needs to be marked, so that you can easily and quickly roll back to this version if a problem occurs in the production environment when a new feature is published. Deploy and publish the function version N+1 and switch 100% traffic to it:

```
sls deploy --inputs publish=true  traffic=0
```

Custom Alias

A custom alias can be created through commands to configure the traffic ratio between two specified function versions.
When using a custom alias for grayscale release, first publish the new feature to a new version, then modify the alias configuration to switch part of the traffic to this version for observation, and finally switch all the traffic to this version gradually.
The custom alias enables flexible version switching. Its configuration method is more complicated than that of the default alias. It is suitable for business scenarios that require higher grayscale release capabilities. **Currently, custom alias is supported only for the SCF component.**



Command description

Publishing function version

Directly publish the version of the `my-function` function without deployment:

```
sls  publish-ver --inputs  function=my-function
```

Creating alias

Create the `routing-alias` alias for the `my-function` function, with the routing rule of 50% traffic for version 1 and 50% traffic for version 2:

```
sls  create-alias --inputs name=routing-alias  function=my-function  version=1
config='{"weights":{"2":0.5}}'
```

Updating alias

Update the flow rule of the `routing-alias` alias of the `my-function` function to 10% for version 1 and 90% for version 2:

```
sls update-alias --inputs name=routing-alias  function=my-function  version=1 confi
```

Listing alias

List the `routing-alias` alias of the `my-function` function:

```
sls list-alias --inputs function=my-function
```

Deleting alias

Delete the `routing-alias` alias of the `my-function` function:

```
sls delete-alias --inputs name=routing-alias  function=my-function
```

Directions

You can publish a tested feature through grayscale release in the following steps:

1. Configure the production environment information into the .env file (`STAGE=prod` indicates the production environment):

```
TENCENT_SECRET_ID=xxxxxxxxxx
TENCENT_SECRET_KEY=xxxxxxxx
STAGE=prod
```

2. Create the `alias-prod` alias and configure its traffic rule (suppose the current stable production version is N):

```
sls create-alias --inputs function=my-function name=alias-prod version=n config='{"
```

3. Configure the alias reference corresponding to the trigger in the `serverless.yml` of the `my-function` function:

```
  events: # Trigger
 - timer: # Timer trigger
     name: # Trigger name, which is `timer-${name}-${stage}` by default
     parameters:
       qualifier: alias-prod # Configure the alias as `alias-prod`
       cronExpression: '*/5 * * * *' # Trigger once every 5 seconds
       enable: true
       argument: argument # Additional parameter
```

4. Deploy the function to the `$latest` production environment and publish the new version (suppose the function name is `my-function` and the new version after release is N+1):

```
sls deploy
sls publish-ver --inputs function=my-function
```

5. Configure the traffic rule of the function alias to switch 10% traffic to the N+1 version (suppose the original production version is N and the function alias is `alias-prod`):

```
sls update-alias --inputs function=my-function name=alais-prod version=n config='{"
```

6. Continue to observe the monitor and switch 100% traffic to version N+1 after it becomes stable:

```
sls update-alias --inputs function=my-function name=alais-prod version=n config='{"
```

Layer Deployment

Last updated：2024-12-02 10:48:10

Overview

Due to the limits of SCF, only code packages below 50 MB in size can be uploaded currently. If your project is too large, you can put dependencies in layers instead of the deployment package to reduce the package size. For specific usages of layers, please see Operations.

Directions

Creating layer

You can create a layer and upload dependencies in the following two ways:
Create in the SSR console
Use the Layer component of Serverless Framework (for more information, please see Layer Component)

Using layer

You can use layer deployment in project configuration in the following two ways: console configuration and local configuration.

Configuring in console

For applications in the Node.js framework, Serverless Framework will automatically create a layer named `${appName}-layer` and upload the application dependency `node_modules` to this layer.
When importing an existing project, you can also choose to create a layer or use an existing layer for deployment. If you create a layer, Serverless Framework will automatically upload the application dependency `node_modules` to this layer.
Note:
The layer creation operation is supported only for the Node.js framework. When using a layer in other frameworks, please make sure that the layer has already been created and the relevant dependencies have been uploaded to the layer.

Configuring through Layer component

1. The Next.js component is used as an example here. Adjust the local project directory, add a `layer` folder, and create a **serverless.yml** file to configure the layer name and version. The `.yml` template is as follows:

```
app: appDemo
```

```
stage: dev

component: layer
name: layerDemo

inputs:
 name: test
 region: ap-guangzhou
 src: ../node_modules # Path of the file to be uploaded
 runtimes:
   - Nodejs10.14
```

For detailed configuration items, please see Layer Component Configuration.

The updated directory structure is as follows:

```
.
├── node_modules
├── src
│    ├── serverless.yml # Function configuration file
│    └── index.js # Entry function
├── layer
│    └── serverless.yml # Layer configuration file
└── .env # Environment variable file
```

2. Open the project configuration file, add the layer configuration item, and import the output of the Layer component as the input of the project configuration file. The template is as follows:

```
app: appDemo
stage: dev

component: nextjs
name: nextjsDemo

inputs:
src:
  src: ./
  exclude:
    - .env

region: ap-guangzhou
runtime: Nodejs10.15
apigatewayConf:
   protocols:
     - http
     - https
   environment: release
layers:
   - name: ${output:${stage}:${app}:layerDemo.name} # Layer name
   version: ${output:${stage}:${app}:layerDemo.version} # Version
```

For the import format, please see Variable Import Description.

3. In the project root directory, run `sls deploy` to complete layer creation and use the output of the Layer component as the input of the Next.js component to configure the layer.

Custom Domain Name and HTTPS Access Configuration

Last updated：2024-12-02 10:48:10

Operation Scenarios

After quickly constructing a Serverless website service through Serverless Component, if you want to configure a custom domain name and support HTTPS access, you can do so in the following two ways:

Prerequisites

A website service has been deployed, and the website hosting address at COS/API Gateway has been obtained. For the specific deployment method, please see Deploying Hexo Blog.
You already have a custom domain name (such as www.example.com). If the domain name is used for Mainland China services, ICP filing is required.
If you need HTTPS access, you can apply for a certificate and get the certificate ID (such as `certificateId` of `axE1bo3)` .

Method 1: Configuring Support for HTTPS Access to Custom Domain Name Through CDN

Before configuration, you need to make sure that you have completed identity verification for your account and activated the CDN service.

Adding configuration

In `serverless.yml` , add CDN custom domain name configuration:

```
# serverless.yml

component: website
name: myWebsite
app: websiteApp
stage: dev

inputs:
  src:
    src: ./public
    index: index.html
    error: index.html
  region: ap-guangzhou
  bucketName: my-hexo-bucket
  protocol: https
  # New CDN custom domain name configuration
  hosts:
    - host: www.example.com # Custom domain name to be configured
      https:
        switch: on
        http2: off
        certInfo:
          certId: 'abc'
          # certificate: 'xxx'
          # privateKey: 'xxx'
```

View full configuration item description >>

Deploying service

Deploy by running the `sls` command, and you can add the `--debug` parameter to view the information during the deployment process:

Note:

`sls` is short for the `serverless` command.

```
$ sls --debug
  DEBUG ─ Resolving the template's static variables.
  DEBUG ─ Collecting components from the template.
  DEBUG ─ Downloading any NPM components found in the template.
  DEBUG ─ Analyzing the template's components dependencies.
  DEBUG ─ Creating the template's components graph.
  DEBUG ─ Syncing template state.
  DEBUG ─ Executing the template's components graph.
  DEBUG ─ Preparing website Tencent COS bucket my-hexo-bucket-1250000000.
  DEBUG ─ Bucket "my-hexo-bucket-1250000000" in the "ap-guangzhou" region already e
  DEBUG ─ Setting ACL for "my-hexo-bucket-1250000000" bucket in the "ap-guangzhou"
  DEBUG ─ Ensuring no CORS are set for "my-hexo-bucket-1250000000" bucket in the "a
  DEBUG ─ Ensuring no Tags are set for "my-hexo-bucket-1250000000" bucket in the "a
  DEBUG ─ Configuring bucket my-hexo-bucket-1250000000 for website hosting.
  DEBUG ─ Uploading website files from /Users/tina/Documents/hexoblog/hexo/public t
  DEBUG ─ Starting upload to bucket my-hexo-bucket-1250000000 in region ap-guangzho
  DEBUG ─ Uploading directory /Users/tina/Documents/hexoblog/hexo/public to bucket
  DEBUG ─ The CDN domain www.example.com has existed.
  DEBUG ─ Updating...
  DEBUG ─ Waiting for CDN deploy success..
  DEBUG ─ CDN deploy success to host: www.example.com
  DEBUG ─ Setup https for www.example.com...
  DEBUG ─ Website deployed successfully to URL: https://my-hexo-bucket-1250000000.c
  myWebsite:
    url:  https://my-hexo-bucket-1250000000.cos-website.ap-guangzhou.myqcloud.com
    env:
    host:
      - https://www.example.com (CNAME: www.example.com.cdn.dnsv1.com)
  17s › myWebsite › done
```

Adding CNAME

After the deployment is completed, you can see a CNAME domain name suffixed with `.cdn.dnsv1.com` in the output on the command line. Set the corresponding CNAME at your DNS service provider as instructed in CNAME Configuration. After it takes effect, you can access the custom HTTPS domain name.

Method 2: Configuring Custom Domain Name Through API Gateway

Adding configuration

In `serverless.yml` , add API Gateway custom domain name configuration. This document uses the egg.js framework as an example, and the configuration is as follows:

```
# serverless.yml
restApi:
  component: "@serverless/tencent-apigateway"
  inputs:
    region: ap-shanghai
    protocols:
      - http
      - https
    serviceName: serverless
    environment: release
    endpoints:
      - path: /users
        method: POST
        function:
          functionName: myFunction # The function name to which the gateway connect
    # Add API Gateway custom domain name configuration
    customDomains:
      - domain: www.example.com
        certificateId: axE1bo3
        protocols:
          - https
```

View full configuration item description >>

Deploying service

Deploy by running the `sls` command, and you can add the `--debug` parameter to view the information during the deployment process:

Note:

`sls` is short for the `serverless` command.

```
$ sls --debug
  DEBUG ─ Resolving the template's static variables.
  DEBUG ─ Collecting components from the template.
  DEBUG ─ Downloading any NPM components found in the template.
  DEBUG ─ Analyzing the template's components dependencies.
  DEBUG ─ Creating the template's components graph.
  DEBUG ─ Syncing template state.
  DEBUG ─ Executing the template's components graph.
  DEBUG ─ Starting API-Gateway deployment with name restApi in the ap-shanghai regi
  DEBUG ─ Using last time deploy service id service-lqhc88sr
  DEBUG ─ Updating service with serviceId service-lqhc88sr.
  DEBUG ─ Endpoint POST /users already exists with id api-e902tx1q.
  DEBUG ─ Updating api with api id api-e902tx1q.
  DEBUG ─ Service with id api-e902tx1q updated.
  DEBUG ─ Deploying service with id service-lqhc88sr.
  DEBUG ─ Deployment successful for the api named restApi in the ap-shanghai region
  DEBUG ─ Start unbind all exist domain for service service-lqhc88sr
  DEBUG ─ Start bind custom domain for service service-lqhc88sr
  DEBUG ─ Custom domain for service service-lqhc88sr created successfullly.
  DEBUG ─ Please add CNAME record service-lqhc88sr-1250000000.sh.apigw.tencentcs.co
  restApi:
    protocols:
      - http
      - https
    subDomain:     service-lqhc88sr-1250000000.sh.apigw.tencentcs.com
    environment:   release
    region:        ap-shanghai
    serviceId:     service-lqhc88sr
    apis:
      -
        path:   /users
        method: POST
        apiId:  api-e902tx1q
    customDomains:
      - www.example.com (CNAME: service-lqhc88sr-1250000000.sh.apigw.tencentcs.com)
  8s › restApi › done
```

Adding CNAME record

After the deployment is completed, you can see a CNAME domain name suffixed with `.apigw.tencentcs.com` in the output on the command line. Once the corresponding CNAME is set and takes effects at your DNS service provider, you can access the custom HTTPS domain name.

Developing and Reusing Application Template

Last updated：2024-12-02 10:48:10

Overview

Serverless Cloud Framework provides multiple basic resource components, which you can mix and use to quickly create and deploy resources in the cloud. This document describes how to use existing components to build your own multi-component serverless application template.

Prerequisites

You have installed Serverless Cloud Framework on at least the 1.0.2 versions:

```
$ scf -v
```

Component Configuration Documentation

Basic Component List
Framework Component List

Directions

This document uses deploying a **framework project based on Layer and Egg** as an example to describe how to import multiple components into your project and quickly complete the deployment. The steps are as follows:

Step 1. Create a project

Create a project `app-demo` and enter this directory:

```
$ mkdir app-demo && cd app-demo
```

Step 2. Build an Egg project

1. In the `app-demo` directory, create a `src` folder and create an Egg project in it:

```
$ mkdir src && cd src
$ npm init egg --type=simple
$ npm i
```

2. In the `src` directory, write the configuration file `serverless.yml` :

```
$ touch serverless.yml
```

A sample `.yml` file for the Egg component is provided below. For more information on all configuration items, please see Egg.js Component Configuration.

```
# serverless.yml
app: app-demo # Application name. The `app`, `stage`, and `org` parameters must be
stage: dev
component: egg
name:  app-demo-egg # Name of the created instance, which is required

inputs:
  src:
    src: ./     # Project path for upload
    exclude:    # Exclude the `node_modules` and `.env` file
      - .env
      - node_modules
  region: ap-guangzhou
  functionName: eggDemo  # Function configuration
  runtime: Nodejs10.15
  apigatewayConf:
    protocols:           # API Gateway trigger configuration. A gateway will be cre
      - http
      - https
    environment: release
```

Note:

The **`app`** , **`stage`** , **and** **`org`** parameters must be the same for the resources created by each component under the same project.
The Egg component essentially creates an API Gateway trigger + SCF resource. Here, you can select different components according to your actual development needs, and the configuration methods are similar. For more information, please see Component Configuration Documentation.

Step 3. Create a layer

Go back to the root directory of `app-demo` , create a `layer` folder, and create a layer configuration file `serverless.yml` in it:

```
$ cd ..
$ mkdir layer && cd layer
$ touch serverless.yml
```

`serverless.yml` can be configured according to the following template (for more information on the configuration, please see Layer Component Configuration):

```
# serverless.yml
app: app-demo # Application name. The `app`, `stage`, and `org` parameters must be
stage: dev
component: layer
name:  app-demo-layer # Name of the created instance, which is required

inputs:
  region: ap-guangzhou
  src:
    src: ../src/node_modules # Path of the project you want to upload to the layer.
    targetDir: /node_modules # File compression directory after upload
  runtimes:
    - Nodejs10.15
```

Note:

The `app` , `stage` **, and** `org` parameters must be the same for the resources created by each component under the same project.

The Layer component also supports importing projects from COS buckets. For more information, please see Layer Component Configuration. When entering the `bucket` parameter, be sure not to include `-${appid}` , as the component will add it automatically.

Step 4. Organize the resource relationship

In the same application, you can organize the creation order of resources according to their dependency relationship. Taking this project as an example, you need to create a layer first and then use the layer in the Egg.js project; therefore, you should ensure that the resource creation order is * *layer > Egg.js application**. The specific steps are as follows:

Modify the `.yml` configuration file of the Egg.js project, configure the layer configuration according to the following syntax, and import the deployment output of the Layer component as the deployment input of the Egg.js project to ensure that the Layer component is created before the Egg.js project:

```
$ cd ../src
```

In `serverless.yml` , add layer configuration in the `inputs` section:

```
inputs:
  src:
    src: ./
    exclude:
      - .env
      - node_modules
  region: ap-guangzhou
  functionName: eggDemo
  runtime: Nodejs10.15
  layers:   # Add the layer configuration
    - name: ${output:${stage}:${app}:app-demo-layer.name} # Layer name
      version: ${output:${stage}:${app}:app-demo-layer.version} # Version
  apigatewayConf:
    protocols:
      - http
      - https
    environment: release
```

For the variable import format, please see Variable Import Description.

At this point, the serverless application has been built, and the project directory structure is as follows:

```
./app-demo
├── layer
│   └── serverless.yml # Layer configuration file
├── src
│   ├── serverless.yml # Egg component configuration file
│   ├── node_modules # Project dependency file
│   ├── ...
│   └── app # Project routing file
└── .env # Environment variable file
```

Step 5. Deploy the application

In the project root directory, run `scf deploy` to complete layer creation and use the output of the Layer component as the input of the Egg.js component to cloudify the Egg.js framework.

```
 $ scf deploy

serverless-cloud-framework

app-demo-layer:
  region:         ap-guangzhou
  name:           layer_component_xxx
  bucket:         scf-layer-ap-guangzhou-code
  object:         layer_component_xxx.zip
  description:    Layer created by serverless component
  runtimes:
    - Nodejs10.15
  version:        3
  vendorMessage: null

app-demo-egg:
  region:        ap-guangzhou
  scf:
    functionName: eggDemo
    runtime:      Nodejs10.15
    namespace:    default
    lastVersion:  $LATEST
    traffic:      1
  apigw:
    serviceId:   service-xxxx
    subDomain:   service-xxx.gz.apigw.tencentcs.com
    environment: release
    url:         https://service-xxx.gz.apigw.tencentcs.com/release/
  vendorMessage: null

76s › app-demo › "deploy" ran for 2 apps successfully.
```

You can click the URL output by `apigw` to access the created application, run `scf info` to view the status of the deployed instance, or run `scf remove` to quickly remove the application.

Step 6. Publish the application template

After the serverless project template is built, Serverless Cloud Framework allows you to publish it in the Serverless Registry for use by your team and others.

1. Create a configuration file

In the root directory, create a `serverless.template.yml` file, and the project directory structure is as follows:

```
./app-demo
├── layer
│   └── serverless.yml # Layer configuration file
├── src
│   ├── serverless.yml # Egg component configuration file
│   ├── node_modules # Project dependency file
│   ├── ...
│   └── app # Project routing file
├── .env # Environment variable file
└── serverless.template.yml # Template project description file
```

2. Configure and publish the project template file

```
 # serverless.template.yml
name: app-demo # Project template name, which must be unique
displayName: Egg.js project template created based on layer # Name of the project t
author: Tencent Cloud, Inc. # Author name
org: Tencent Cloud, Inc. # Organization name, which is optional
```

```
type: template # Project type, which can be either `template` or `component`. It is
description: Deploy an egg application with layer. # Describe your project template
description-i18n:
  zh-cn: Egg.js project template created based on layer # Description
keywords: tencent, serverless, eggjs, layer # Keywords
repo:  # Source code repo, which is usually your GitHub repo
readme:  # Detailed description file, which is usually your GitHub repo README file
license: MIT # Copyright notice
src: # Describe the files in the project to be published as a template
  src: ./ # Specify a relative directory, the files under which will be published a
  exclude: # Describe the files in the specified directory to be excluded
    # The following files are usually excluded
    # 1. Files containing `secrets`
    # 2. Files managed by `.git` git source code
    # 3. Third-party dependencies such as `node_modules`
    - .env
    - '**/node_modules'
    - '**/package-lock.json'
```

After the `serverless.template.yml` file is configured, you can use the `scf publish` command to publish the project to the Registry as a template.

```
$ scf publish

serverless ⚡registry
Publishing "app-demo@0.0.0"...

Serverless › Successfully published app-demo
```

3. Reuse the template

After your template is published, others can quickly download it and reuse the project by running the `scf init` command.

```
$ scf init app-demo --name example
$ cd example
$ npm install
```

Variable Import Description

`serverless.yml` supports multiple ways to import variables:

Import basic Serverless parameters

In the `inputs` field, you can directly import basic Serverless parameter configuration information through the

`${org}` and `${app}` syntax.

Import environment variables In `serverless.yml` , you can directly import the environment variable configuration (including the environment variable configuration in the `.env` file and variable parameters manually configured in the environment) through the `${env}` syntax.

For example, you can import the environment variable `REGION` through `${env:REGION}` .

Import the output results of other components

If you want to import the output information of other component instances into the current component configuration file, you can configure it by using the following syntax: `${output:[app]:[stage]:[instance name].[output]}`

Sample `.yml` file:

```
app: demo
component: scf
name: rest-api
stage: dev

inputs:
  name: ${stage}-${app}-${name} # The final name is "acme-prod-ecommerce-rest-api"
  region: ${env:REGION} # `REGION=` information specified in the environment variab
  vpcName: ${output:prod:my-app:vpc.name} # Get the output information of other com
  vpcName: ${output:${stage}:${app}:vpc.name} # The above methods can also be used
```