Backstory

How the Serverless Framework Was Born

In 2014, Amazon Web Services released a groundbreaking service called AWS Lambda (https://aws.amazon.com/lambda/) that offers a new way to deploy your *web*, *mobile* or *IoT* application's code to the cloud. Instead of deploying the entire codebase of your app all at once, with Lambda you deploy each of your application's functions individually, to their own containers. Overall, Lambda is groundbreaking for three reasons:

- **Pay-Per-Use Pricing**

  AWS Lambda charges you only when your functions are run. No more monthly-based billing for servers, no more wasted dollars spent on under-utilized servers. There is no such thing as under-utilization with AWS Lambda.

- **Unprecedented Agility**

  Lambda offers ready to go containers and orchestration out-of-the-box, leaving you free to decide how you would like to containerize your logic. You can choose a monolithic, microservices, or nanoservices approach with Lambda. Read more about that in our Overview (http://docs.serverless.com/docs/introducing-serverless).

- **No Servers**

  Every Lambda function container auto-scales automatically when called concurrently. Since your functions can scale massively out-of-the-box, you no longer have to think about scaling/managing servers, Amazon deals with that. You focus only on building your product, wit.

However... while AWS Lambda offers a powerful new way of developing/running applications, when we began building our first project based entirely on AWS Lambda, we realized structure was badly needed. Managing all of the containers that Lambda introduces is a difficult task. Add to that multi-developer teams, multi-stage and multi-region support and you will quickly get into a messy situation.

Thus, the Serverless Framework (https://github.com/serverless/serverless) was born. The first and most powerful framework for building applications exclusively on AWS Lambda.

Suggest Edits (/docs/backstory/edit)

Overview

What the Framework Does

Serverless is an application framework for building serverless web, mobile and IoT applications exclusively on AWS Lambda (https://aws.amazon.com/lambda/). A Serverless app can simply be a couple of lambda functions to accomplish some tasks, or an entire back-end comprised of hundreds of lambda functions. Serverless currently supports `nodejs` and `python2.7` runtimes. Support for `Java` and other future runtimes that AWS Lambda will support will be coming soon.

Serverless comes in the form of a Node.js command line interface that provides structure, automation and optimization to help you build and maintain Serverless apps. The CLI allows you to control your Lambdas, API Gateway Endpoints as well as your AWS resources via AWS CloudFormation. Overall, we've made a strong effort to make not just a groundbreaking Serverless framework, but the best framework for building applications with AWS in general (that is also Serverless!). As a result, Serverless incorporates years of AWS expertise into its tooling, giving you best practices out-of-the-box. Serverless does not seek to conceal AWS in abstraction, but to put structure around the AWS SDK and CloudFormation, and approach Amazon Web Services and all that it offers from the focal point of Lambda. In the future, we believe that Lambda will be the focal point of AWS.

Lastly, we work full time on this and are funded by a top tier VC firm in Silicon Valley. We are here for the long-term to support developers building Serverless applications. Don't hesitate to email us (mailto:team@serverless.com) and ask us for help. Also, we're growing our team. If the Serverless architecture appeals to you or you just want to make awesome tools for other developers, please let us know (mailto:team@serverless.com).

Suggest Edits (/docs/introducing-serverless/edit)

Configuring AWS

Configuring AWS and Giving Serverless Access to Your Account

AWS gives you a ton of free resources whenever you create a new AWS account. This is called the free tier. It includes a massive allowance of free Lambda Requests, DynamoDB tables, S3 storage, and more. Before building Serverless apps, we strongly recommend starting with a fresh AWS account for maximum cost savings.

Creating an Administrative IAM User

The Serverless Framework is one of the first application frameworks to manage both your code and infrastructure. To manage your infrastructure on AWS, we're going to create an Admin user which can access and configure the services in your AWS account. To get you up and running quickly, we're going to create a AWS IAM User with Administrative Access to your AWS account.

> **⚠ Admin Access to Your AWS Account**
>
> In a production environment we recommend reducing the permissions to the IAM User which the Framework uses. Unfortunately, the Framework's functionality is growing so fast, we can't yet offer you a finite set of permissions it needs. In the interim, ensure that your AWS API Keys are kept in a safe, private location.

Now let's create an Admin IAM user:

- Create or login to your Amazon Web Services Account and go the the Identity & Access Management (IAM) Page.
- Click on **Users** and then Create New Users. Enter *serverless-admin* in the first field and click **Create**.
- View and copy the security credentials/API Keys in a safe place.
- In the User record in the AWS IAM Dashboard, look for **Managed Policies** on the **Permissions** tab and click **Attach Policy**. In the next screen, search for and select **AdministratorAccess** then click **Attach**.

When you go to create or install a Serverless Project, you will be prompted to enter your AWS Access Keys. Once you enter them, they will be persisted to your Project's folder in the `admin.env` file.

> **AWS Profiles**
>
> The Serverless Framework can also work with AWS Profiles already set on your computer. In the Project Create or Project Install screens, it will detect your AWS Profiles and display them. Select the one you want, and the AWS Access Keys for that profile will be persisted to your project.

Suggest Edits (/docs/configuring-aws/edit)

Installing Serverless

Installing the Serverless Framework and Creating Your First Serverless Project

Now that you've configured AWS, you're ready to start using the Serverless framework. To install, simply run the following command:

```
npm install serverless -g
```

After it installs, you can create a new project:

```
serverless project create
```

The Serverless CLI will ask for a few pieces of information about your project (name, domain, email...etc). Serverless uses this information to build up your stack with CloudFormation (http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html). This process takes around 3 mins.

> **Provisioning CloudFormation**
>
> If you would like to create the project structure without AWS resources being provisioned, use `serverless project create -c true` . You can inspect the generated CloudFormation and execute when ready. For more info, read the Project Create docs below (/project-create).

Now you have a barebones project that is pretty much useless. To make it a more useful, let's create a function. Make sure you're in the root directory of your newly created project, and then run:

```
serverless function create functions/function1
```

You'll be prompted to choose whether to create only a function, or create an endpoint or event along with your function. Choose "Create Endpoint" for this demo. This will create a `function1` function with one endpoint inside a folder called `functions`. You can create a function directly in the root of you project with `serverless function create function1`, but we recommend you group your functions in folders and subfolders.

Run `serverless dash deploy` to open up the interactive deployment dashboard. Use the down arrow key to select the function and press enter. Down arrow once more to select the endpoint and press the enter button again (so that both text items switch from grey to yellow).
Finally, move to "Deploy" and hit enter.

After deployment is complete, you will be given a URL. Visit this URL in your browser to see your new project deployed. Deployment couldn't get any simpler!

You now have a basic Serverless project, and you're ready to take a deep dive and explore the project structure.

Suggest Edits (/docs/installing-serverless/edit)

Project Structure

How Serverless Projects Look Like

A basic Serverless project contains the following directory structure:

```
s-project.json
s-resources-cf.json
admin.env
_meta
    |__resources
         |__s-resources-cf-dev-useast1.json
    |__variables
         |__s-variables-common.json
         |__s-variables-dev.json
         |__s-variables-dev-useast1.json
functions
    |__function1
         |__event.json
         |__handler.js
         |__s-function.json
```

Here's the same directory structure with some explanation:

```
s-project.json               // project and author data
s-resources-cf.json      // CloudFormation template for all stages/regions
admin.env                    // AWS Profiles - gitignored)
_meta                         // meta data that holds stage/regions config and variables - gitignored
    |__resources            // final CF templates for each stage/region
         |__s-resources-cf-dev-useast1.json
    |__variables             // variables specific to stages and regions
         |__s-variables-common.json
         |__s-variables-dev.json
         |__s-variables-dev-useast1.json
functions                     // folder to group your project functions
    |__function1           // your first function
         |__event.json      // sample event for testing function locally
         |__handler.js      // your function handler file
         |__s-function.json   // data for your lambda function, endpoints and event sources
```

Now let's dive deeper into the most critical pieces of a Serverless project:

Project

Each Serverless Project contains an `s-project.json` file that looks like this:

s-project.json ()

```
{
  "name": "projectName",
  "custom": {}, // For plugin authors to add any properties that they need
  "plugins": [] // List of plugins used by this project
}
```

Meta Data

Each Serverless project contains a `_meta` folder in its root directory. This folder holds user specific project data, like stages, regions, CloudFormation template files and variables (more on variables later). Since this folder contains sensitive information, it's gitignored by default, allowing you to share your Serverless projects with others, where they can add their own meta data.

Functions

Functions are the core of a Serverless project. These are the functions that get deployed to AWS Lambda. You can organize your project functions however you like. We recommend you group all of your project functions in a `functions` folder in the root of your project. You can make it even more organized with more nesting and subfolders inside that `functions` folder. For simple projects, you can put your functions directly in the root of your project. It's completely flexible.

Each function can have several endpoints, and each endpoint can have several methods (ie. GET, POST...etc). These are the endpoints that get deployed to AWS API Gateway and they all point to the Function that they're defined within. Functions can also have several event sources (i.e. DynamoDB, S3..etc). You can configure your Lambda Function, its Endpoints and Events in the `s-function.json` file:

s-function.json ()

```
{
  "name": "functionName",
  "customName": false,  // Custom name for your deployed Lambda function
  "customRole": false,  // Custom IAM Role for your deployed Lambda function
  "handler": "function1/handler.handler", // path of the handler relative to the function root
  "runtime": "nodejs",
  "description": "some description for your lambda",
  "timeout": 6,
  "memorySize": 1024,
  "custom": {
    "excludePatterns": [] // an array of whatever you don't want to deploy with the function
  },
  "environment": { // env vars needed by your function. Makes use of Serverless variables
    "SOME_ENV_VAR": "${envVarValue}"
  },
  "events": [ // event sources for this lambda
    {
      "name" : "myEventSource", // unique name for this event source
      "type": "schedule", // type of event source
      "config": {
         "schedule": "rate(5 minutes)",
          "enabled": true
     }
    }
],
  "endpoints": [ // an array of endpoints that will invoke this lambda function
    {
      "path": "function1",
      "method": "GET",
      "authorizationType": "none",
      "apiKeyRequired": false,
      "requestParameters": {},
      "requestTemplates": {
        "application/json": ""
      },
      "responses": {
        "400": {
          "selectionPattern": "^\\[BadRequest\\].*", // selectionPattern is mapped to the Lambda Error Regex
          "statusCode": "400" // HTTP Status that is returned as part of the regex matching
        },
        "403": {
          "selectionPattern": "^\\[Forbidden\\].*",
          "statusCode": "403"
        },
        "404": {
          "selectionPattern": "^\\[NotFound\\].*",
          "statusCode": "404"
        },
        "default": {
          "statusCode": "200",
          "responseParameters": {},
          "responseModels": {},
          "responseTemplates": {
            "application/json": ""
          }
        }
```

```
      }
    }
  ],
  "vpc": {
    "securityGroupIds": [],
    "subnetIds": []
  }
}
```

For more information about the `s-function.json` file, check out the Function Configuration section (/docs/function-configuration).

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Suggest Edits (/docs/project-structure/edit)

Project Architectures

Determining How to Containerize Your Logic

One of the strengths of the Serverless framework is it gives you the freedom to containerize/isolate your logic any way you'd like. This is possible due to simple nesting of folders: `project/functions/subfolder/functions` . They allow the following architectures and patterns to be available to you in the framework:

Monolithic

If you choose, you can contain all of your logic into a single Lambda function. Then add multiple endpoints or events to that single Lambda function. In the Framework, you could do this by creating a single `function` .

> **Consider Using GraphQL**
>
> If you're making a REST API, consider using GraphQL (https://github.com/graphql/graphql-js) in front of your databases to reduce the number of endpoints you need. GraphQL may make Monolithic approaches viable again.
>
> Here's an example Serverless Blog REST API (https://github.com/serverless/serverless-graphql-blog) that has only one endpoint, powered by GraphQL.

Microservices

With this pattern, you can organize your functions in folders (e.g. `restApi` folder) and subfolders according to business logic. You can think of each subfolder as a `resource` (e.g. `restApi/users/`). It's a common pattern to have only a single Lambda function in a `resource` , which can handle all logic for whatever that `resource` is dedicated to.

- `restApi/users/usersAll`
- `restApi/posts/postsAll`
- `restApi/comments/commentsAll`

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

For example, a REST API could have a `users` resource containing one 'all' function that handles all actions for `users`. Assign endpoints for `create, read, update, delete` to the `all` function. API Gateway can pass the METHOD and PATH into your function via the `event` object, so you can determine inside your function how to handle/route incoming request.

Nanoservices

For the most agile solution, you can create a functions for every single endpoint and event you have.

For example, a REST API could have a `users` subfolder/resource containing multiple functions for `create, read, update, delete`.

- `restApi/users/createUser`
- `restApi/users/readUser`
- `restApi/users/updateUser`
- `restApi/users/deleteUser`

This way you can update/iterate on each endpoint or event individually, without affecting any other parts of your application. Like your `restApi/users/create` function.

Mixing

You can always mix the above approaches:

- `restApi/users/usersAll`
- `restApi/posts/createPost`
- `restApi/posts/readPost`
- `restApi/posts/updatePost`
- `restApi/posts/deletePost`

Suggest Edits (/docs/application-architectures/edit)

Templates & Variables

Templates offer reusable configuration syntax, Variables offer dynamic configuration values

---

Serverless Projects use configuration files written in JSON which can get big, and they sometimes need to include dynamic values that change for stages and regions (ie. ARNs).

To reduce redundancy in the config files, we created Project Templates. To allow for dynamic values in the config files, we created Project Variables.

> **Templates & Variables are for Configuration Only**
>
> Templates and variables are used for configuration of the project only. This information is not usable in your lambda functions. To set variables which can be used by your lambda functions, use environment variables.

Templates

Templates are variables containing objects, arrays or strings. These dramatically reduce redundancy in your configuration files, since you can use the same template in multiple places.

Here is an example of a template that can be used as API Gateway Mapping Template (http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-mapping-template-reference.html)

s-templates.json ()

```
{
  "apiGatewayRequestTemplate": {
    "application/json": {
      "body": "$input.json('$')",
      "pathParams" : "$input.params().path",
      "queryParams" : "$input.params().querystring"
    }
  }
}
```

Or, it can be a YAML file instead of JSON:

s-templates.yaml ()

```
  apiGatewayRequestTemplate:
    application/json:
      body: "$input.json('$')"
      pathParams: "$input.params().path"
      queryParams: "$input.params().querystring"
```

You can add specify templates in your `s-project.json, s-resources-cf.json, and s-function.json` configuration files by enclosing them in `$${}`, like this:

s-function.json ()

```
"requestTemplates": "$${apiGatewayRequestTemplate}"
```

To define templates in your project, add them to `s-templates.json` files to the root of your project, subfolders, or function folders.

```
project
|_ s-templates.json
    subfolder
    |_ s-templates.json
        function
        |_ s-templates.json
```

You can add them wherever you think is best. The reason you can add template files in multiple folders is because templates defined in parent folders can be extended by templates defined in subfolders.

Let's put the examples above together. Say we have two template files that extend the template "apiGatewayRequestTemplate":

project/s-templates.json ()

```
{
  "apiGatewayRequestTemplate": {
    "application/json": {
      "body": "$input.json('$')",
      "pathParams" : "$input.params().path",
      "queryParams" : "$input.params().querystring"
    }
  }
}
```

project/functions/function/s-templates.json ()

```
{
  "apiGatewayRequestTemplate": {
    "application/json": {
      "pathId": "$input.params('id')"
    }
  }
}
```

This template is used for an endpoint's API Gateway Mapping Template (http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-mapping-template-reference.html). In a Serverless project, this information is specified in the `s-function.json` containing the endpoint, like this:

project/functions/function/s-function.json ()

```
{
  "name": "show",
  "handler": "multi/show/handler.handler",
  "runtime": "nodejs",
  "timeout": 6,
  "memorySize": 256,
  "custom": {},
  "endpoints": [
    {
      "path": "users/show/{id}",
      "method": "GET",
      "authorizationType": "none",
      "apiKeyRequired": false,
      "requestParameters": {},
      "requestTemplates": "$${apiGatewayRequestTemplate}",
      "responses": {}
    }
  ]
}
```

The `apiGatewayRequestTemplate` template in the endpoint syntax above would be populated with the combination of both of the above templates.

Also, you can extend a template in a parent folder by templates in multiple subfolders at the same time. Each subfolders will get an aggregated template that contains unique values, depending on what values they extended the parent template with.

Lastly, you can put templates in your `s-resources-cf.json` file as well. Project Variables (described below) can also be included in templates. The Framework first populates all templates, then populates all variables.

> **Templates For Function Names**
>
> Create custom Function Name Templates by putting a `"functionName": "${project}-${stage}-${name}"` key in `s-templates.json` in the root of your project. `${name}` is a reserved Project Variable and is always populated with the name property in the current configuration file. Then in `s-function.json`, put this: `"customName": "$${functionName}"`

Variables

Variables hold strings or integers. They enable you to add dynamic values to your configuration files that change with each project stage and region. This is great for AWS Account specific data that changes across your stages and regions, like ARNs.

All of your variables are defined inside the `_meta/variables` folder (the `_meta` folder is gitgnored by default). Inside this variables folder you'll find a JSON file for each stage and each region for that stage.

Here's an example of defining a new variable in the `us-east-1` region in the `dev` stage. If you open the `s-variables-dev-useast1.json` file, you'll find the following default variables which are used by our framework:

s-variables-dev-useast1.json ()

```
{
  "region": "us-east-1",
  "resourcesStackName": "projectName-dev-r",
  "iamRoleArnLambda": "arn:aws:iam::AWSaccount:role/projectName-dev-r-IamRoleLambda-someRole"
}
```

As you can see, these values are specific to your project and your AWS account.

You can now reference this variable in any of your project's configuration files by enclosing it in `${}` (just like with templates, but with a single $ sign). For example, here is a variable being used in the CloudFormation syntax in your `s-resources-cf.json` file:

s-resources-cf.json ()

```
"IamPolicyLambda": {
        "Type": "AWS::IAM::Policy",
        "Properties": {
          "PolicyName": "${region}-lambda",
          ...
        }
}
```

You can use also use variables in `s-template.json` files. The framework first populates all templates, then populates all variables.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Suggest Edits (/docs/templates-variables/edit)

Function Configuration

All you need to know about configuring your functions, endpoints and event sources.

> **AWS Docs**
>
> The `s-function.json` file is just a reflection of AWS configurations. If some of the Function and Endpoint configurations feel alien to you, you need to familiarize yourself with AWS Lambda docs (including event sources) (http://docs.aws.amazon.com/lambda/latest/dg/welcome.html) and API Gateway docs (https://aws.amazon.com/api-gateway/).

Serverless Functions is the core of your project. All function configurations are in the `s-function.json` file, and it's the most complex Serverless JSON file. This file contains configuration for your Function, its Endpoints and Event Sources. Here's an example `s-function.json`:

s-function.json ()

```
{
  "name": "function1",
  "description": "My first lambda in project ${project} on stage ${stage}",
  "customName": false,
  "customRole": false,
  "handler": "function1/handler.handler",
  "timeout": 6,
  "memorySize": 1024,
  "custom": {
    "excludePatterns": []
  },
  "environment": {
    "SOME_ENV_VAR": "${envVarValue}"
  },
  "endpoints": [
    {
      "path": "fun",
      "method": "GET",
      "type": "AWS",
      "authorizationType": "none",
      "authorizationFunction": "",
      "apiKeyRequired": false,
      "requestParameters": {},
      "requestTemplates": {
        "application/json": ""
      },
      "responses": {
        "400": {
          "statusCode": "400"
        },
        "default": {
          "statusCode": "200",
          "responseParameters": {},
          "responseModels": {},
          "responseTemplates": {
            "application/json": ""
          }
        }
      }
    }
  ],
  "events": [
    {
      "name" : "myEventSource",
      "type": "schedule",
      "config": {
         "schedule": "rate(5 minutes)"
      }
    }
],
"vpc": {
    "securityGroupIds": [],
    "subnetIds": []
  }
}
```

The `s-function.json` file properties can be divided into three categories: **Function**, **Endpoint** and **Event** configurations.

Function Configurations

- **name**: the name of this function. This matches the name of the function folder. **The name of your function must be unique project wide**. Because this is how we identify the function.
- **description**: the description of the lambda function that will be visible in the AWS lambda console. Defaults to 'Serverless Lambda function for project ...' if not set.
- **customName**: the name of the lambda function. It's set to `false` by default. This means that you don't want to set a custom name for your lambda, and instead we'll name the lambda for you using a combination of project and function names.

> **name vs. customName**
>
> The `name` property is the function name within the context of your Serverless Project, while the `customName` property is the **actual lambda name that is deployed to AWS**. If you don't set a `customName`, we'll generate a lambda name for you that is just a combination of the project and function names.

- **customRole**: you can set a custom IAM role ARN in this property to override the default project IAM role. It's set to false by default.
- **handler**: The path to the lambda `handler.js` file relative to the root of the directory you want deployed along with your function.

> **The Handler Property Is Very Flexible**
>
> The handler property gives you the ability to share code between your functions. By default the handler property is `handler.handler`, that means it's only relative to the function folder, so only the function folder will be deployed to Lambda.
>
> If however you want to include the parent subfolder of a function, you should change the handler to be like this: `functionName/handler.handler`. As you can see, the path to the handler now includes the function folder, which means that the path is now relative to the parent subfolder, so in that case the parent subfolder will be deployed along with your function. So if you have a `lib` folder in that parent subfolder that is required by your function, it'll be deployed with your function.
>
> This also gives you the ability to handle npm dependencies however you like. If you have a `package.json` and `node_modules` in that parent subfolder, it'll be included in the deployed lambda. So the more parent folders you include in the handler path, the higher you go in the file tree.

- **timeout**: the maximum running time allowed for your lambda function. Default is 6 seconds. Maximum allowed timeout by AWS is 300 seconds.
- **memorySize**: the memory size of your lambda function. Default is 1024MB, but you can set it up to 1.5GB. This is a limit set by AWS lambda.
- **vpc**: an object containing configurations for using the function to access resources in AWS Virtual Private Cloud. This object contains two keys: `securityGroupIds` and `subnetIds`. The value of both of these keys are an array that contains the relevant security group and subnet Ids. For a detailed guide on using VPC with Lambda, checkout these notes (https://github.com/serverless/serverless/issues/629#issuecomment-184472421) shared by our awesome contributor Stacey Moore (https://github.com/staceymoore).
- **environment**: this is where you define environment variables needed for this function. Each env var is a key in this `environment` object, and the env var value is the value of this key. You can make use of Serverless variables in the `_meta` folder to reference sensitive information.

- **custom**: this property allows plugin authors to add more function configurations for their plugins. It has the following default property:
  - **excludePatterns**: this is where you set whatever you want to exclude during function deployment.

Endpoint Configurations

Each function (`s-function.json`) can have multiple endpoints. They are all defined in the `endpoints` property. It's an array that contains multiple Endpoint objects. Each of which have the following configurations:

- **path**: The path of the endpoint. This value gets added to the stage and AWS host to construct the full endpoint url (i.e. `https://fpq68h492f.execute-api.us-east-1.amazonaws.com/development/group1/endpoint-path1`)
- **method**: The HTTP method for this endpoint. Set to `GET` by default.
- **authorizationType**: The type of authorization for your endpoint. Default is `none` . If you want to use AWS IAM tokens, then specify `AWS_IAM` . If you want to use a custom authorizer function, specify `CUSTOM` and don't forget to set the *authorizerFunction* key below.
- **authorizerFunction**: The name of the function (from `s-function.json`) that you will be using as the custom auth function. Only used if *authorizationType* is set to `CUSTOM` .
- **apiKeyRequired**: Whether or not an API Key is required for your endpoint. Default is `false` .
- **requestParameters**: Sets the AWS request parameters for your endpoint. Request parameters are represented as a key/value map, the key must match the pattern of *integration.request.**{location}**.**{integrationName}***, and the value must match the pattern of *method.request.**{location}**.**{apiName}***. ***location*** is either *querystring*, *path*, or *header*. ***integrationName*** indicates the name that will be used to send this value to the backend lambda. ***apiName*** is the name that callers of you Api will use. For example, to set this to pass an Authorization header from the original api call to the backend Lambda, add the following key/value: `"integration.request.header.Authorization": "method.request.header.Authorization"`
- **requestTemplates**: Sets the AWS request templates for your endpoint.
- **responses**: Sets the AWS responses parameters and templates. By default there are two response objects in your `s-function.json` , "400" and "default".

```
* **selectionPattern**  The `selectionPattern` property of these objects maps to the Lambda Error Regex in the Integration Response.
```

Event Sources Configurations

Each function (`s-function.json`) can have multiple event sources. They are all defined in the `events` property. It's an array that contains multiple Event objects. Each of which have the following configurations:

- **name**: a name for your event source that is unique within your function. We use this name to reference the event source and construct the event source `sPath` .
- **type**: the type of your event source. We currently support 5 event sources: `dynamodbstream` , `kinesisstream` , `s3` , `sns` , and `schedule` .
- **config**: configuration object for your event source. The properties of this object depends on the event source `type` . Below are examples for each event source types and their configurations.

Event Sources Examples

s-function.json - DynamoDB Stream Event Source ()

```
"name" : "myDynamoDbTable",
"type": "dynamodbstream",
"config": {
  "streamArn": "${streamArnVariable}", // required!
  "startingPosition": "LATEST", // default is "TRIM_HORIZON" if not provided
  "batchSize": 50, // default is 100 if not provided
  "enabled": false // default is true if not provided
}
```

s-function.json - Kinesis Stream Event Source ()

```
"name" : "myKinesisStream",
"type": "kinesisstream",
"config": {
  "streamArn": "${streamArnVariable}", // required!
  "startingPosition": "LATEST", // default is "TRIM_HORIZON" if not provided
  "batchSize": 50, // default is 100 if not provided
  "enabled": false // default is true if not provided
}
```

s-function.json - S3 Event Source ()

```
"name" : "myS3Event",
"type": "s3",
"config": {
  "bucket": "${testEventBucket}", // required! - bucket name
  "bucketEvents": ["s3:ObjectCreated:*"], // required! - an array of events that should trigger the lambda
  "filterRules" : [
      {
          "name" : "prefix | suffix",
          "value" : "STRING_VALUE"
      }
   ] // optional, specify prefix or suffix for S3 event source
}
```

s-function.json - SNS Event Source ()

```
"name" : "mySNSEvent",
"type": "sns",
"config": {
  "topicName": "test-event-source" // required! - the topic name you want your lambda to subscribe to
}
```

s-function.json - Schedule Event Source ()

```
"name" : "mySchedule",
"type": "schedule",
"config": {
    "schedule": "rate(5 minutes)", // required! - could also take a cron expression: "cron(0 20 * * ? *)"
    "enabled": true // default is false if not provided
}
```

> **Real World Examples**
>
> If you'd like to see a real world working example of `s-function.json` with event sources, check out our official unit test function (https://github.com/serverless/serverless/blob/master/tests/test-prj/functions/function0/s-function.json).

Suggest Edits (/docs/function-configuration/edit)

Workflow

The recommended workflow for Serverless Projects.

Every Serverless Project uses resources from Amazon Web Services and divides these resources into three groups:

- AWS Lambdas
- AWS API Gateway REST API
- AWS Other Resources (IAM Roles, DynamoDB tables, S3 Buckets, etc.)

Serverless Projects don't have environments (they live exclusively on AWS). However, there is still need to separate and isolate the AWS resources a Project uses for development, testing and production purposes and Serverless does this through *Stages*. Stages are similar to environments, except they exist merely to separate and isolate your Project's AWS resources.

Each Serverless Project can have one or multiple Stages, and each Stage can have one or multiple Regions. Regions are based off of AWS Regions (https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/), like `us-east-1`. Some AWS resources come with their own "stage" concepts and Serverless Project Stages are designed to integrate with those, wherever possible. Here is how:

AWS Lambdas
Your Project will have one set of deployed Lambda Functions on AWS, which can be replicated across each Region your Project uses. Every Lambda Function can have multiple *versions* and *aliases*. When you deploy a Function in your Project to a Stage, it deploy a Lambda that will be immediately versioned and aliased under the name of that Stage.

AWS API Gateway REST API
If your Functions have Endpoint data in their `s-function.json` files, a REST API on AWS API Gateway will automatically be created for your Project. Projects can only have one REST API, which can be replicated across each Region your Project uses. Every API Gateway REST API can have multiple stages. When you deploy an Endpoint in your Project to a Project Stage, it builds the Endpoint

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

on your API Gateway REST API and then creates a deployment in that API Gateway stage.

AWS Other Resources
Your Project's other AWS resources are the only AWS resources that have separate deployments for each Stage. These separate Stage deployments can be replicated across each Region your Project uses as well.

Creating Your Project Stages

We recommend every Project have the following Stages.

- dev
- beta
- prod

If you are working on a team with multiple developers, we recommend giving each developer working on the Project their own Stage. In this case, your Project might have Stages like this:

- dev
- tom
- jeff
- beta
- prod

Deploying Your Functions

When you deploy your Function's Code (aka your AWS Lambda Function), it is given a `LATEST` version by default. However, pointing your Event Source Maps or your REST API Endpoints to `LATEST` is highly discouraged. This is because when multiple developers are working on the same Lambda function, they can trample each-other's work, if they have any resources that are pointing toward the `LATEST` version.

To avoid all of this, Serverless never uses `LATEST`. Every time you create or update your Lambda Function, it is automatically versioned and aliased to the Project Stage you specify. This avoids the "trampling" issue entirely. Further, it allows Serverless to include Environment Variables and Stage-specific IAM Roles within the Lambda Function, which AWS does not yet have support for.

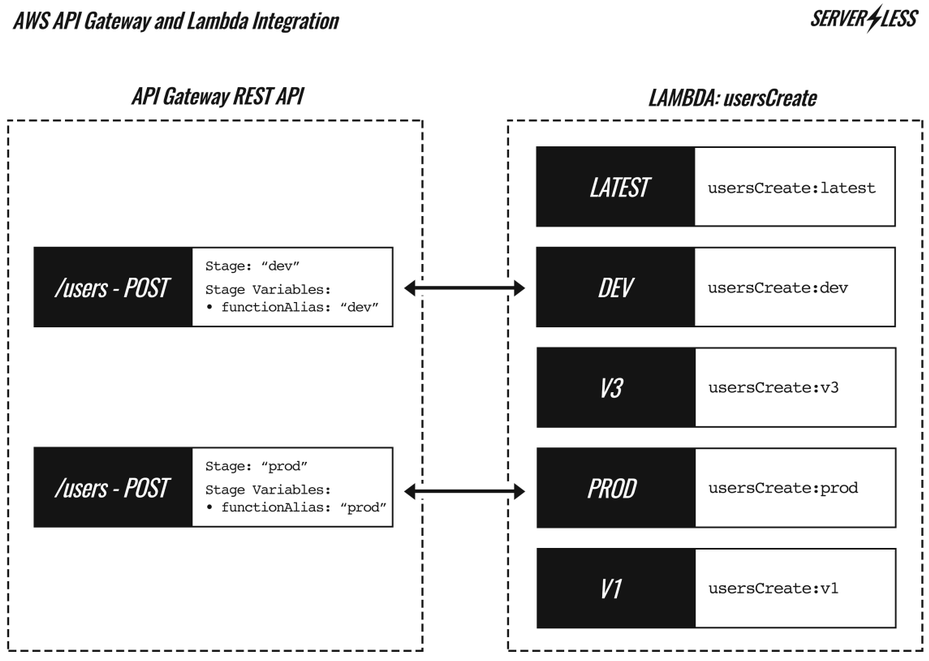
Here is what happens when you deploy your Function's Code:

- Your function code is copied to a distribution directory
- Your regular handler file is replaced by one that Serverless adds titled `_serverless_handler`, which contains your Function's Environment Variables in-lined in the code.
- Your Lambda code is compressed and uploaded to AWS in the region you chose w/ `publish: true` set. This publishes a new Lambda Function Version immediately.
- An update request is sent to your deployed Lambda Version to Alias it with the specified Stage.

- The IAM role specified in the s-project.json for the specified Stage is added to your Lambda on deployment.

Here is what happens when you deploy your Function's Endpoint:

- If no Stage is set, it deploys to your Project's “dev” Stage by default
- The Endpoint is constructed on your REST API in the specified Region
- A new Deployment is created.
- After Deployment is created, the Stage variable is reset to the Stage name.



(https://files.readme.io/WKs3MX5YTiqfcJTE79Nc_serverless_deployment_flow.png)

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Suggest Edits (/docs/workflow/edit)

Best Practices

Some best practices that we recommend while working with the Serverless Framework.

Don't give `AdministratorAccess` to AWS API keys

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

While the getting started guide says to create an administrative user with `AdministratorAccess` permissions, it only does this to get you going faster. As stated this should not be done in a production environment. Our recommendation is to give your users with API access key `PowerUserAccess` at a maximum. The fallout is you will not be able to execute a cloudformation json file from the command line that creates any IAM resources. This should be done from the AWS CloudFormation UI, behind a user that has 2FA enabled and a secure password. All the Serverless tooling has a `-c, --noExeCf` option that will simply update your CloudFormation file, which can then be executed in the UI.

Keep your lambda codebase as small as possible

The smaller the size of code, the quicker your container gets up and running. The less code in the execution path, the quicker your runtime VM returns a result. Both of these statements verified by AWS Lambda engineers.

Reuse your Lambda code

Organize your functions in subfolders and have them require a `lib` folder according to your business logic just like any package. Just make sure you set the `handler` property of the `s-function.json` file to be relative to the parent folder that you require so that it is deployed along with your functions.

Keep your CloudFormation resources organized

You can define CF resources in the `s-project.json` or `s-module.json`. If there's a resources that is used only by a single Module, it's best to define it in the `s-module.json` to keep your Project resources organized. And if you decide to share your Module with the community, they'll have that resource available out of the box.

Initialize external services outside of your Lambda code

When using services (like DynamoDB) make sure to initialize outside of your lambda code. Ex: module initializer (for Node), or to a static constructor (for Java). If you initiate a connection to DDB inside the Lambda function, that code will run on every invoke.

Suggest Edits (/docs/best-practices/edit)

CLI Overview

An overview of how the Serverless CLI works.

The Serverless Framework is mostly a command line utility that makes it easy to manage your serverless projects. We've designed the CLI to be as user-friendly as possible. A typical Serverless command consists of a **context** and an **action**, along with some **options** and **parameters** where applicable. For example, for the following command:

```
serverless project create -n project-name -r us-east-1
```

This command has a "project" **context**, and a "create" **action**. `-n project-name` and `-r us-east-1` are both options that the `serverless project create` command needs. It still needs more options, but the CLI is smart enough to know what's missing, and it'll prompt you for only what's missing.

> **The difference between a Parameter and an Option**
>
> A parameter doesn't require the dash that an option requires (ex. `-s <some-option>`), which makes it a little easier to type. But that also means that the ordering of the parameters matter if a command needs more than one parameter.
>
> We try to be consistent in our use of parameters and options. If there's a lot of data required from the user, we prefer setting them as options so that you wouldn't have to think about the ordering, but for simple commands, using parameters is better.

Here's a summary of all the available Serverless commands:

```
serverless project create
serverless project install
serverless project init
serverless project remove

serverless function run
serverless function create
serverless function deploy
serverless function logs
serverless function remove
serverless function rollback

serverless endpoint deploy
serverless endpoint remove

serverless event deploy
serverless event remove

serverless dash deploy
serverless dash summary

serverless stage create
serverless stage remove

serverless region create
serverless region remove

serverless resources deploy
serverless resources remove
serverless resources diff

serverless plugin create
```

> **A Nifty Shortcut**

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

> It's a bit tedious to type the `serverless` command every time you want to do something. You can use the `sls` or the `slss` shortcut instead. For example, you can create a new project by typing: `sls project create`. This applies to **all** other commands as well. Pretty neat, ha!

You can find detailed information about each Serverless command, its required options and parameters, along with some examples below. Keep reading!

> **Getting Help & Debugging**
>
> Whenever you feel confused, just run `serverless` in the terminal. This will generate helpful information about all the available commands. You can also add the `-h` or `--help` option to any command for information about that specific command.
>
> If you're facing some bugs, just add the `--debug` option to any command, which will run the command in **Debug Mode** that generates helpful information about what's going on with the code.
>
> You can also ask for help from our community by creating a new issue on our repo (https://github.com/serverless/serverless/issues/new), or asking in our Gitter chat room (https://gitter.im/serverless/serverless) (ping @ac360 (https://github.com/ac360) or @eahefnawy (https://github.com/eahefnawy) for support). Just make sure you provide your installed version of the Serverless Framework by running `serverless version`.

> **Turn Off CLI Interactivity**
>
> We've designed our CLI with best user experience in mind. So most commands are in interactive mode where you'll be prompted for any missing options. However, when you're using our CLI programmatically (ie. By using Continuous Integration tools), you'll need to turn off the interactive mode. You can do that by setting an environment variable called `CI` to true.

Suggest Edits (/docs/commands-overview/edit)

Project Create

Creates a new Serverless project.

```
serverless project create
```

Creates a new Serverless project in the current working directory with a default `dev` stage. It takes the following options:

- `-n <name>` the name of your project.
- `-b <bucket>` the domain of your project.
- `-p <awsProfile>` an AWS profile that is defined in `~/.aws/credentials` file.

- `-r <region>` a lambda supported region for your new project.
- `-s <stage>` the first stage for your new project.
- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

> **Manual steps required if -c specified**
>
> If you choose to not execute the CloudFormation file by specifying the `-c` flag, make sure to either manually run `sls resources deploy` or add the `iamRoleArnLambda` attribute to the `_meta/variables/s-variables-<stage>-<region>.json` file.

Examples

```
serverless project create
```

In this example, all options are missing, so you'll be prompted to enter each of the required options for best user experience.

```
serverless project create -n myProject -b com.my-project
```

In this example, you provided the name and the bucket of the project, so you'll only be prompted for the stage, region and profile options.

```
serverless project create -c true
```

In this example, you've instructed Serverless to **not** execute CloudFormation. So after you enter the project information, the CloudFormation stack won't be created, and you'll have to manually upload the relevant CF template file from the `_meta/resources` folder to AWS.

Suggest Edits (/docs/project-create/edit)

Project Install

Installs a published Serverless project

```
serverless project install
```

Installs a new Serverless project in the current working directory with a default `dev` stage. It takes the following parameters and options:

- `<npm-module-name>` (parameter): the name of the Serverless project you want to install that is published on npm.

- `-n <name>` the name of your newly installed project. This will replace the npm name you provided earlier.
- `-b <bucket>` the bucket name of your installed project.
- `-p <awsProfile>` an AWS profile that is defined in `~/.aws/credentials` file.
- `-s <stage>` the first stage for your installed project.
- `-r <region>` a lambda supported region for your installed project.
- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

▼ GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
▶ DEEP DIVE ()
▶ CLI REFERENCE ()
▶ CREATING PLUGINS ()

Examples

```
serverless project install serverless-starter
```

In this example, you've passed the required parameter (the serverless project you want to install), but all options are missing, so you'll be prompted to enter each of the required options for best user experience. Just like with `serverless project create`.

Suggest Edits (/docs/project-install/edit)

Project Init

Initializes a new Serverless project

```
serverless project init
```

Must be run within a Serverless Project. Initializes a new Serverless project. It's useful when you `git clone` a Serverless project, and want to reconstruct the `_meta` folder and set up the project on AWS. It takes the following options:

- `-n <name>` the name of your project.
- `-b <bucket>` the bucket of your project.
- `-p <awsProfile>` an AWS profile that is defined in `~/.aws/credentials` file.
- `-s <stage>` the first stage for your new project.
- `-r <region>` a lambda supported region for your project.
- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Examples

```
serverless project init
```

After you `git clone` a Serverless Project and `cd` inside the root directory of the project. You can run `serverless project init` to reconstruct the `_meta` folder and set up your project on your own AWS account. It's very similar to `sls project create`, except that you're using a shared project with all the code written for you.

Suggest Edits (/docs/project-init/edit)

Project Remove

Removes and cleans up your Serverless Project from AWS.

```
serverless project remove
```

Must be run within a Serverless Project. Removes and cleans up your Serverless Project from AWS. It takes the following options:

- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

Examples

```
serverless project remove
```

In this example, the command will remove all stages, regions, CF resources, functions, endpoints and events from AWS. This command is useful when you want to clean up your deprecated projects form AWS.

```
serverless project remove -c
```

In this example, you've set the `-c` to true, so the command will remove your whole project from your account, but **it won't remove your CF resources**. It'll output a CF template in the `_meta/resources` folder that you can upload to the AWS console to manually remove your CF resources.

Suggest Edits (/docs/project-remove/edit)

Stage Create

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

creates a new stage for your serverless project

```
serverless stage create
```

Must be run inside a Serverless project. It creates a new stage and its first region for your project. It takes the following options:

- `-s <stage>` The name of your new stage.
- `-r <region>` A lambda supported region for your new stage.
- `-p <awsProfile>` An AWS profile to use for this stage.

Examples

```
serverless stage create
```

In this example, all options are missing, so you'll be prompted to enter each of the required options for best user experience.

```
serverless stage create -s production -r us-east-1 -p default
```

In this example, you provided all the required options. So your stage will be created right away without prompting for anything else.

Suggest Edits (/docs/stage-create/edit)

Stage Remove

Removes a stage from your Serverless Project

```
serverless stage remove
```

Must be run within a Serverless Project. Removes an existing stage from your Serverless Project and AWS account, along with the CF resources defined in that stage. It takes the following options:

- `-s <stage>` the stage you want to remove from your Serverless project.
- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

Examples

```
serverless stage remove -s production
```

In this example, the command will instantly remove the `prod` stage from your Serverless Project and AWS account along with any CF resources defined in that stage.

```
serverless stage remove -s prod -c
```

In this example, you've set the `-c` option to `true`, so the command will remove the `prod` stage from your Serverless Project and AWS account, but **it won't remove the CF resources for that stage**. Instead, it'll output a CF template in the `_meta/resources` folder that you can execute manually on the AWS console.

Suggest Edits (/docs/stage-remove/edit)

Region Create

Creates a new region for an existing stage.

```
serverless region create
```

Must be run inside a Serverless project. It creates a new region for an existing stage inside your project. It takes the following options:

- `-s <stage>` The name of the stage you want to add a region to.
- `-r <region>` A lambda supported region for your chosen stage.

Examples

```
serverless region create
```

In this example, all options are missing, so you'll be prompted to enter each of the required options for best user experience.

```
serverless region create -s prod -r us-east-1
```

In this example, you're creating a new `us-east-1` region in the `prod` stage. If the `production` stage doesn't exist in your project, or already has `us-east-1` region defined, you'll get an error.

Suggest Edits (/docs/region-create/edit)

Region Remove

Removes a region from a stage in your Serverless Project.

```
serverless region remove
```

Must be run within a Serverless Project. Removes an existing region from an existing stage in your Serverless Project, along with the CF resources defined in that region. It takes the following options:

- `-s <stage>` the stage that contains the region you want to remove from your Serverless Project.
- `-r <region>` the region you want to remove from your Serverless project.
- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

Examples

```
serverless region remove -s prod -r us-west-2
```

In this example, the command will instantly remove the `us-west-2` region from the `prod` stage in your Serverless Project. It will also clean up your AWS account from that region along with any CF resources defined in that region.

```
serverless region remove -s prod -r us-west-2 -c
```

In this example, you've set the `-c` option to `true`, so the command will remove the `us-west-2` region from the `prod` stage, but **it won't remove the CF resources for that region**. Instead, it'll output a CF template in the `_meta/resources` folder that you can execute manually on the AWS console.

Suggest Edits (/docs/region-remove/edit)

Function Create

Creates a New Function for Your Project

```
serverless function create
```

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Must be run inside a Serverless Project. It generates a basic scaffolding for a new Function. Where this function is located inside your project depends on the path you provide. This command takes the following parameters and options:

- `functionPath` The relative path to the function you want to create (relative to the project root). The command will create any folders required no matter how deep the path is to make that path valid.
- `-r <runtime>` Optional - The runtime of your new function. Default is `nodejs`. The only other supported runtime is `python2.7`

Examples

```
serverless function create function1
```

In this example, you'll create a function named `function1` in the root of the project.

```
serverless function create functions/function1
```

In this example, you'll create a function named `function1` inside a `functions` folder in the root of the project.

```
serverless function create functions/subfolder/function1
```

In this example, you'll create a function named `function1` inside the `subfolder` subfolder inside a `functions` folder in the root of the project. The command simply creates any subfolders you need to make your function path valid.

```
serverless function create
```

In this example, you did not provide a function path, so you'll be prompted to enter a function name. For best user experience, the exact location of your new function within your project will be based on the current working directory. If you run this command in the project root, you'll create the function in the root of the project, if you run it inside any folder, you'll create it inside that folder.

```
serverless function create -r python2.7
```

This example is identical to the previous example, but because you provided a runtime option with the value of `python2.7`, the created function will be a python function, and you'll have a `handler.py` instead of `handler.js` in the newly created function folder.

Suggest Edits (/docs/function-create/edit)

Function Deploy

Deploys your function to AWS.

```
serverless function deploy
```

Must be run inside a Serverless Project. It deploys your function to AWS. This command takes the following options and parameters:

- functionNames (parameter): The names of the functions you want to deploy. Can be one or many function names.
- `-s <stage>` The stage you want to deploy your functions to. Optional if your project has only one stage.
- `-r <region>` Optional - The AWS region you want to deploy your function to. If not provided, the function will be deployed to **all** the regions defined in your chosen stage by default.
- `-f <alias>` Optional - Sets an alias for your function.
- `-a <all>` Optional - Deploy all your functions.
- `-t <dontRemoveTemp>` Optional - Do not remove `_tmp` folder.

Examples

```
serverless function deploy
```

If you don't provide a function name like in this example, the command will behave depending on where in your Project you're running it. If you're running in your project root directory, it'll deploy all the functions in your project, if you're running in a subfolder, it'll only deploy all the functions inside that subfolder, if you're running in a Function, it'll deploy only this Function.

> **❗ Beware of the `handler` Property**
>
> We mentioned it in the Function Configuration Section (/v0.5.0/docs/function-configuration) already, but to emphasize we'll mention it again. The `handler` property of your function will determine the root of the deployed lambda package. So if your function require code from any parent folder, make sure you set the handler property path to be **relative to that parent folder**. By default, it's relative to the function folder only, so we're assuming you have a simple function that is not requiring any code from any parent folders.

```
serverless function deploy myFunction -s prod -r us-east-1
```

In this example, you'll instantly deploy the `myFunction` function. The function will be deployed to the `us-east-1` region in the `prod` stage.

```
serverless function deploy myFunction -f myAlias
```

In this example, you'll be prompted to choose a stage if your project has more than one stage. After that, the command will deploy the `myFunction` function all regions defined in the chosen stage while setting an alias `myAlias` to your function.

```
serverless function deploy myFunction myOtherFunction
```

In this example, you'll deploy the **two** functions `myFunction` and `myOtherFunction` because you provided two Funciton names. But first you'll be prompted to choose a stage if your Project has more than one stage, after that deployment to all regions in that stage will begin.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

> **Deploy Resources Before Functions**
>
> Function deployment requires the `iamRoleArnLambda` variable, which is generated only after you deploy your resources. By default, resources are deployed automatically whenever you create a new stage/region, but if you explicitly chose not to deploy your resources when you created your stage/region with the `-c` option, remember to deploy the resources manually with `sls resources deploy` to the same stage/region you want to deploy your functions to before you deploy your functions.

Suggest Edits (/docs/function-deploy/edit)

Function Remove

Removes deployed functions from your AWS account based on the provided stage/region.

```
serverless function remove
```

Must be run inside a Serverless Project. It removes deployed functions from your AWS account based on the provided stage/region. It takes the following options and parameters:

- functionNames (parameter): The names of the functions you want to remove. Can be one or many function names.
- `-s <stage>` the stage you want to remove functions from.
- `-r <region>` the region in your chosen stage you want to remove functions from.
- `-a <BOOLEAN>` **Optional** - removes all functions from your AWS account. Default is false (obviously! :)

Examples

```
serverless function remove myFunction
```

In this example, you'll be prompted to choose a stage and region to remove your functions from. After that the `myFunction` function will be removed from your AWS account.

```
serverless function remove myFunction myOtherFunction -s prod -r us-east-1
```

In this example, you'll instantly remove the functions `myFunction` and `myOtherFunction` from the `us-east-1` region in the `prod` stage.

```
serverless function remove --all -s prod -r us-east-1
```

In this example, you'll instantly remove all the functions in your project from the `us-east-1` region in the `prod` stage. Super dangerous!

Suggest Edits (/docs/function-remove/edit)

Function Rollback

Rollback a deployed function to a previous version.

▼ GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
▶ DEEP DIVE ()
▶ CLI REFERENCE ()
▶ CREATING PLUGINS ()

```
serverless function rollback
```

Must be run inside a Serverless Project. This command lets you roll back a deployed function to a previous version. It takes the following options and parameters:

- functionName (parameter): The name of the function you want to rollback.
- `-s <stage>` The stage you want to rollback your function in.
- `-r <region>` The AWS region you want to rollback your function in. Optional if your stage has only one region.
- `-v <NUMBER>` The version you want to rollback your function to.
- `-m <NUMBER>` Optional - Maximum number of versions to show in prompt. Default is 50.

Examples

```
serverless function rollback myFunction
```

In this example, you're trying to rollback the `myFunction` function, but you didn't provide a region, stage or a version number, so you'll be prompted for all these required options. After you make your choices your deployed function will roll back to the chosen version.

```
serverless function rollback myFunction -s prod -r us-east-1 -v 4
```

In this example, you'll instantly rollback the `myFunction` function to version 4, in the `us-east-1` region in the `prod` stage.

Suggest Edits (/docs/function-rollback/edit)

Function Run

Runs your local or deployed function for testing.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

```
serverless function run
```

Must be run inside a Serverless Project. It runs your local or deployed function for testing using the `event.json` as a sample event. This command takes the following **parameters**:

- functionName (parameter): The unique name of the function you want to run.
- `-s <stage>` The stage you want to run your function in.
- `-r <region>` The region you want to run your function in. Optional if your Stage has only one region.
- `-l <BOOLEAN>` Show the log output. Optional.
- `-i, <invocationType>` AWS lambda InvocationType (http://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html#API_Invoke_RequestSyntax). Default value: `RequestResponse`.
- `-d` Executes the deployed function. Must be specified or other parameters (-s / -r) are ignored.

> **Stage/Region Options are just for Env Vars**
>
> The reason why you provide stage/region for function run regardless of whether you're running locally or remotely is to use the correct environment variable. So remember, if you provide a stage/region, that doesn't mean it'll run the function **remotely** in that stage/region. You have to pass the `-d` option to make that work.

Examples

```
serverless function run
```

If you don't provide a function name like in this example, the command will behave depending on where in your Project you're running it. If you're running outside of a function folder, it'll throw an error, if you're running in a function folder, you'll run this function.

```
serverless function run myFunction
```

In this example, you'll **locally** run the `myFunction` function.

```
serverless function run myFunction -s dev -d
```

In this example, you'll run the **deployed** `myFunction` function.

Suggest Edits (/docs/function-run/edit)

Function Logs

Fetches lambda function logs from CloudWatch

```
serverless function logs
```

Must be run inside a Serverless Project. It fetches the lambda function logs from CloudWatch. This command takes the following options and parameters:

- functionName (parameter): The name of the function you want get logs for. Not required if you're running from a function folder.
- `-s <stage>` The stage you want to get function logs from. Optional if your Project has only one stage.
- `-r <region>` Optional - The AWS region you want to get function logs from. Optional if your Stage has only one region.
- `-t <BOOLEAN>` Optional - Tail the log output. Default is false.
- `-d <STRING>` Optional - The duration of time in which the log history is shown. Example values: `10m`, `2h`, `1d`, `10minutes`, `1day`. Default: `5m`.
- `-f <STRING>` Optional - A log filter pattern.
- `-i <NUMBER>` Optional - Tail polling interval in milliseconds. Default: `1000`.

Examples

```
serverless function logs
```

If you don't provide a function name like in this example, the command will behave depending on where in your project you're running it. If you're running outside of a function folder, it'll throw an error, if you're running in a function folder, you'll get this function logs.

```
serverless function myFunction -s production -r us-east-1
```

In this example, you'll get the "myFunction" logs from the `us-east-1` region in the `prod` stage.

```
serverless function logs myFunction -s prod -r us-east-1 -t -d 24h -f error
```

In this example, you'll get the "myFunction" logs which has a `error` substring for the last 24 hours and will get new logs until you stop the command execution.

Suggest Edits (/docs/function-logs/edit)

Endpoint Deploy

Deploys an Endpoint to AWS API Gateway.

```
serverless endpoint deploy
```

Must be run inside a Serverless Project. It deploys your endpoints to AWS. This command takes the following options and parameters:

- endpointName (parameter): The name of your endpoint, which is the combination of your endpoint path and endpoint method (i.e. `users/create~GET`). This is how we identify endpoints in your project, because this combination is always unique.
- `-s <stage>` The stage you want to deploy your endpoint to. Optional if your project has only one stage.
- `-r <region>` Optional - The AWS region you want to deploy your endpoint to. If not provided, the endpoint will be deployed to **all** the region defined in your chosen stage by default.

Examples

```
serverless endpoint deploy
```

If you don't provide an Endpoint name like in this example, the command will behave depending on where in your Project you're running it. If you're running in your project root directory, it'll deploy all the endpoints in your project, if you're running in a subfolder, it'll only deploy all the endpoints inside that subfolder, if you're running in a function, you'll deploy only the endpoints of that function.

> **Deploy Functions Before Endpoints**
>
> If you try to deploy an endpoint that points to a lambda function that **wasn't yet deployed**, you'll get an error message from AWS. Once you deploy your Function at least once, you're free to deploy its endpoints anytime.

```
serverless endpoint deploy 'users/create~POST' -s prod -r us-east-1
```

In this example, you'll instantly deploy the endpoint with the path `users/create` and a method of `POST` . The endpoint will be deployed to the `us-east-1` region in the `prod` stage.

```
serverless endpoint deploy 'users/create~POST' users/list~GET
```

In this example, you'll deploy the **two** provided endpoints. But first you'll be prompted to choose a stage if your Project has more than one stage, after that deployment to all regions in that stage will begin.

Suggest Edits (/docs/endpoint-deploy/edit)

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Endpoint Remove

Removes deployed endpoints from your AWS account based on the provided stage/region.

```
serverless endpoint remove
```

Must be run inside a Serverless Project. It removes deployed endpoints from your AWS account based on the provided stage/region. It takes the following options and parameters:

- functionNames (parameter): The names of the endpoints you want to remove, which is just the combination of endpoints path and method.
- `-s <stage>` the stage you want to remove endpoints from.
- `-r <region>` the region in your chosen stage you want to remove endpoints from.
- `-a <BOOLEAN>` **Optional** - removes all endpoints from your AWS account. Default is false.

Examples

```
serverless endpoint remove user/create~POST
```

In this example, you'll be prompted to choose a stage and region to remove your endpoints from. After that the endpoint with the path `user/create` and method `POST` will be removed from your AWS account.

```
serverless endpoint remove user/create~POST user/list~GET -s prod -r us-east-1
```

In this example, you'll instantly remove the **two** provided endpoints from the `us-east-1` region in the `prod` stage.

```
serverless endpoint remove --all -s prod -r us-east-1
```

In this example, you'll instantly remove all the endpoints in your project from the `us-east-1` region in the `prod` stage. Super dangerous!

Suggest Edits (/docs/endpoint-remove/edit)

Event Deploy

Deploys event sources for your lambda.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

```
serverless event deploy
```

Must be run inside a Serverless Project. It deploys your events to AWS. This command takes the following options and parameters:

- eventNames (parameter): The names of the events you want to deploy.
- `-s <stage>` The stage you want to deploy your event to. Optional if your project has only one stage.
- `-r <region>` Optional - The AWS region you want to deploy your event to. If not provided, the endpoint will be deployed to **all** the region defined in your chosen stage by default.

> **Event Sources Permissions**
>
> For event sources that are following the **push model** (S3, SNS & Schedule), we create all the required permissions to invoke the lambda for you, so deploying will work the right away out of the box. But for the event sources that are following the **pull model** (DynamoDB & Kinesis Streams), you have to give your lambda function permission to access DynamoDB/Kinesis before deploying the event, otherwise deployment will fail.

Examples

```
serverless event deploy
```

If you don't provide an event path like in this example, the command will behave depending on where in your project you're running it. If you're running in your project root directory, it'll deploy all events in your project, if you're running in a subfolder, it'll only deploy all the events inside that subfolder, if you're running in a function, you'll deploy only the events of that function.

> **Deploy Functions Before Events**
>
> If you try to deploy an event that is associated to a lambda function that **wasn't yet deployed**, you'll get an error message from AWS. Once you deploy your Function at least once, you're free to deploy its events anytime.

```
serverless event deploy myEvent -s prod -r us-east-1
```

In this example, you'll instantly deploy the event named `myEvent` that is in the function `myFunction`. The event will be deployed to the `us-east-1` region in the `prod` stage.

```
serverless event deploy myEvent myOtherEvent
```

In this example, you'll deploy the **two** events `myEvent` and `myOtherEvent` because you provided two event names. But first you'll be prompted to choose a stage if your project has more than one stage, after that deployment to all regions in that stage will begin.

Suggest Edits (/docs/event-deploy/edit)

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Event Remove

Removes event sources from your AWS account based on the provided stage/region.

```
serverless event deploy
```

Must be run inside a Serverless Project. it removes event sources from your AWS account based on the provided stage/region.. This command takes the following options and parameters:

- eventNames (parameter): The names of the events you want to remove.
- `-s <stage>` The stage you want to remove your events from. Optional if your project has only one stage.
- `-r <region>` Optional - The AWS region you want to remove your events from. If not provided, the event will be removed from **all** the region defined in your chosen stage by default.
- `-a <BOOLEAN>` **Optional** - removes all events from your AWS account. Default is false.

Examples

```
serverless event remove myEvent
```

In this example, you'll be prompted to choose a stage and region to remove your events from. After that the `myEvent` event will be removed from your AWS account.

```
serverless event remove myEvent myOtherEvent -s prod -r us-east-1
```

In this example, you'll instantly remove the **two** events `myEvent` and `myOtherEvent` from the `us-east-1` region in the `prod` stage.

```
serverless event remove --all -s prod -r us-east-1
```

In this example, you'll instantly remove all the events in your project from the `us-east-1` region in the `prod` stage.

Suggest Edits (/docs/event-remove/edit)

Resources Deploy

Deploys all your project's and modules's CloudFormation resources.

```
serverless resources deploy
```

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Must be run inside a Serverless Project. It updates your AWS resources by collecting all the resources defined in your `project/s-resources-cf.json` file and populates all the referenced templates and variables. It takes the following options:

- `-s <stage>` the stage you want to deploy resources to
- `-r <region>` the region in your chosen stage you want to deploy resources to
- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

Examples

```
serverless resources deploy
```

In this example, you'll be prompted to choose a stage and region to deploy your resources to. After that it'll deploy the resources to the chosen stage and region.

```
serverless resources deploy -s dev -r us-east-1
```

In this example, you'll instantly deploy your resources to the `us-east-1` region in the `dev` stage.

```
serverless resources deploy -c -s dev -r us-east-1
```

In this example, you won't deploy the CF template file to AWS. It'll only generate a CF template file called `s-resources-cf-dev-useast1.json` inside the `_meta/resources` folder. You'll have to upload this file manually using the AWS console to deploy your resources.

Suggest Edits (/docs/resources-deploy/edit)

Resources Remove

Removes CloudFormation resources from a given stage/region in your Serverless Project

---

```
serverless resources remove
```

Must be run within a Serverless Project. Removes CloudFormation resources from a given stage/region in your Serverless project. It takes the following options:

- `-s <stage>` the stage that contains the region you want to remove resources from.
- `-r <region>` the region you want to remove resources from.
- `-c <BOOLEAN>` **Optional** - Doesn't execute CloudFormation if true. Default is **false**.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Examples

```
serverless resources remove -s prod -r us-west-2
```

In this example, the command will instantly remove CloudFormation resources from the `us-west-2` region in the `prod` stage.

```
serverless resources remove -s prod -r us-west-2 -c
```

In this example, you've set the `-c` option to `true`, so the command will only output a CF template in the `_meta/resources` folder that you can execute manually on the AWS console to remove the resources.

Suggest Edits (/docs/resources-remove/edit)

Resources Diff

Outputs the diff between your deployed resources and the resources currently defined in your project.

```
serverless resources diff
```

Must be run inside a Serverless Project. It outputs the different between your deployed resources and the resources currently defined in your project. It takes the following options:

- `-s <stage>` the stage you want to deploy resources to
- `-r <region>` the region in your chosen stage you want to deploy resources to
- `-j <BOOLEAN>` **Optional** - Output unformatted JSON. Default is false.

```
serverless resources diff
```

In this example, you'll be prompted to choose a stage and region to fetch your deployed resources. After that it'll output the difference between your deployed resources and the resources currently defined in your project.

```
serverless resources diff -s prod -r us-east-1 -j
```

In this example, you'll instantly see the difference between your deployed resources and the resources currently defined in your project without any prompts, and because you provided the `-j` option, you'll see an unformatted JSON output.

Suggest Edits (/docs/resources-diff/edit)

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

Dash Deploy

Prompts the deployment dashboard for functions and endpoints.

```
serverless dash deploy
```

Must be run inside a Serverless Project. An interactive CLI dashboard that makes it easy to select and deploy functions, endpoints and events concurrently. This command is intended to offer great user experience so it can only be used interactively.

This command will prompt you to choose a stage and region for your deployment, and it'll list all the functions and their endpoints and events. You just select whatever you want to deploy, and hit `Deploy` . If you run this command in the root directory of your project, it'll list all functions, endpoints and events in your Project. If you run it inside a subfolder, it'll only list the functions, endpoints and events inside that subfolder, and if you run it inside a function, it'll only list that function, its endpoints and its events.

Suggest Edits (/docs/dash-deploy/edit)

Dash Summary

Displays a summary of your Serverless Project state.

```
serverless dash summary
```

Must be run inside a Serverless project. It displays a summary of your Serverless project, number of stages, regions, functions, endpoints and events. This simple command doesn't take any options or parameters.

Suggest Edits (/docs/dash-summary/edit)

Plugins Overview

An overview of Serverless Plugins, and how to extend the framework functionality.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

One of our core principles is to make the Framework completely extensible. To stick to that principle, we have designed the entire framework so developers can modify or extend everything that it does.

Like you've seen in the CLI Reference, all of the functionality in the Serverless Framework is divided into Actions. For example, the functionality to create a new Serverless project exists in an action, and creating a new stage in that project exists in a separate action. If an operation is long or complex, like deploying a Lambda function, it is divided into multiple actions so developers can modify parts of the process, instead of the whole process. Actions can also call other actions. For example, the ProjectCreate action calls StageCreate and RegionCreate actions.

Even better, Serverless also features Hooks which allow you to add functions that run before and after an individual action. Like we said, extensibility is one of our core principles. To add custom actions and hooks, make a **Serverless Plugin**. Plugins and actions are basically the same thing, but we refer to the core functionality as actions, and custom functionality as plugins.

The following sections of the docs will walk you through how to create a custom action, or a plugin, and make use of the Serverless API. If you're curious how action/plugin files look like, take a look at our Actions folder in our codebase (https://github.com/serverless/serverless/tree/master/lib/actions). It includes all the functionality that was described in the CLI Reference section. Once you take a look at some action files, you'll notice a pattern for creating actions/plugins. The plugin that you will create will look very similar. So always refer to those default action files while developing your plugin as an example if you ever get stuck.

Each of these classes contain lots of methods to help you manipulate your project. Usually you won't have to init these classes yourself, but an instance will be returned to you by using certain methods (i.e. `_this.S.getProject()` returns a Project class instance). In the following sections we'll explore each of these classes and methods in detail, starting with the main Serverless class.

Installing Plugins

You can extend the Serverless Framework through Serverless Plugins that have been authored by our community. They are packaged as *npm* modules.

Right now, the available plugins are listed in the Serverless Framework's README file (we are working on making discovery easier). To install them, find their npm names and follow these steps:

- Go to the root of your Serverless Project
- Run `npm install <plugin> --save`
- In your Project's `s-project.json`, in the `plugins` property, add the npm name of your recently added plugin to the array, like this:

```
plugins: [
   "serverless-optimizer-plugin"
]
```

Suggest Edits (/docs/plugins/edit)

Your First Plugin

Creating and exploring your first Serverless Plugin.

The first step for creating a new plugin is to create the initial boilerplate by running the following command in the root directory of your project:

```
serverless plugin create
```

This command will ask you for a plugin name, and then create a new folder in the root directory of your Serverless project called `plugins` if it doesn't already exist. It will also create a subfolder inside that `plugins` folder with the name you provided, along with some boilerplate files for the plugin. The most important file that will be generated is the `index.js`. This is where you'll be developing your plugin. If you open this `index.js` file you'll find some starter code and helpful comments to get you started. You can check the full file with all the comments by clicking here (https://github.com/serverless/serverless/blob/master/lib/templates/plugin/index.js). Here's a simpler version:

index.js ()

```
'use strict';


const path  = require('path'),
  fs        = require('fs'),
  BbPromise = require('bluebird'); // Serverless uses Bluebird Promises and we recommend you do to because they provide more than your average Promise :)

module.exports = function(S) { // Always pass in the ServerlessPlugin Class

  /**
   * Adding/Manipulating Serverless classes
   * - You can add or manipulate Serverless classes like this
   */

  S.classes.Project.newStaticMethod     = function() { console.log("A new method!"); };
  S.classes.Project.prototype.newMethod = function() { S.classes.Project.newStaticMethod(); };

  /**
   * Extending the Plugin Class
   * - Here is how you can add custom Actions and Hooks to Serverless.
   * - This class is only required if you want to add Actions and Hooks.
   */

  class PluginBoilerplate extends S.classes.Plugin {

    constructor() {
      super();
      this.name = 'myPlugin';
    }

    registerActions() {

      S.addAction(this._customAction.bind(this), {
        handler:       'customAction',
        description:   'A custom action from a custom plugin',
        context:       'custom',
        contextAction: 'run',
        options:       [{
          option:      'option',
          shortcut:    'o',
          description: 'test option 1'
        }],
        parameters: [
          {
            parameter: 'paths',
            description: 'One or multiple paths to your function',
            position: '0->'
          }
        ]
      });

      return BbPromise.resolve();
    }

    registerHooks() {
```

```
    S.addHook(this._hookPre.bind(this), {
      action: 'functionRun',
      event:  'pre'
    });

    S.addHook(this._hookPost.bind(this), {
      action: 'functionRun',
      event:  'post'
    });

    return BbPromise.resolve();
  }

  _customAction(evt) {

    let _this = this;

    return new BbPromise(function (resolve, reject) {

      // console.log(evt)           // Contains Action Specific data
      // console.log(_this.S)       // Contains Project Specific data
      // console.log(_this.S.state) // Contains tons of useful methods for you to use in your plugin.  It's the official API for plugin developers.

      console.log('------------------');
      console.log('YOU JUST RAN YOUR CUSTOM ACTION, NICE!');
      console.log('------------------');

      return resolve(evt);

    });
  }

  _hookPre(evt) {

    let _this = this;

    return new BbPromise(function (resolve, reject) {

      console.log('------------------');
      console.log('YOUR SERVERLESS PLUGIN\'S CUSTOM "PRE" HOOK HAS RUN BEFORE "FunctionRun"');
      console.log('------------------');

      return resolve(evt);

    });
  }

  _hookPost(evt) {

    let _this = this;

    return new BbPromise(function (resolve, reject) {

      console.log('------------------');
      console.log('YOUR SERVERLESS PLUGIN\'S CUSTOM "POST" HOOK HAS RUN AFTER "FunctionRun"');
      console.log('------------------');
```

```
        return resolve(evt);

      });
    }
  }
  return PluginBoilerplate;

};
```

As you can see, we're passing the Serverless class instance `s` to our plugin. This instance is the starting point for the entire Serverless API, which gives you all the power you need to start writing your plugin and integrate it with your project. Keep reading!

Suggest Edits (/docs/serverless/edit)

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

The Serverless API

Exploring the Serverless API, its classes and methods.

To start manipulating your project and add extra functionality, you'll need to use the Serverless API. Within your action method, you have access to the Serverless instance. This instance is the starting point of the whole Serverless API. It gives you access to all of the Serverless classes, each of which has tons of helpful methods for manipulating Serverless projects. This Serverless instance itself has some methods to get you started with the classes quickly. On top of that, the Serverless instance also gives you access to all of the useful utility functions that we've written.

The Serverless instance is directly passed to your plugin. So you can use it right away like this like this...

```
_customAction(evt) {
    let Serverless = S;
}
```

Serverless Classes

Below is a list of all of our classes. Using these classes and their methods together will give you complete control over your project, allowing you to extend the framework core functionality very easily. Because we're making rapid changes and moving fast, we're keeping all the docs for the API methods inline in classes files, that makes it easier to maintain and encourages users to be familiar with our codebase. So to learn more on each class, checkout its file:

- **Project:** This class represents a single Serverless project. This is your entry point to all of the other classes. For a full API Reference on this class constructor and methods, checkout the Project class file (https://github.com/serverless/serverless/blob/master/lib/Project.js).
- **Function:** This class represents a single Serverless function. For a full API Reference on this class constructor and methods, checkout the Function class file (https://github.com/serverless/serverless/blob/master/lib/Function.js).
- **Endpoint:** This class represents a single Serverless endpoint. For a full API Reference on this class constructor and methods, checkout the Endpoint class file (https://github.com/serverless/serverless/blob/master/lib/Endpoint.js).

- **Event:** This class represents a single Serverless event. For a full API Reference on this class constructor and methods, checkout the Event class file (https://github.com/serverless/serverless/blob/master/lib/Event.js).
- **Stage:** This class represents a single Serverless stage. For a full API Reference on this class constructor and methods, checkout the Stage class file (https://github.com/serverless/serverless/blob/master/lib/Stage.js).
- **Region:** This class represents a single Serverless region. For a full API Reference on this class constructor and methods, checkout the Region class file (https://github.com/serverless/serverless/blob/master/lib/Region.js).
- **Resources:** This class represents your project CloudFormation resources. For a full API Reference on this class constructor and methods, checkout the Resources class file (https://github.com/serverless/serverless/blob/master/lib/Resources.js).
- **Variables:** This class represent your project variables. For a full API Reference on this class constructor and methods, checkout the Variables class file (https://github.com/serverless/serverless/blob/master/lib/Variables.js).
- **Templates:** This class represents your project templates. For a full API Reference on this class constructor and methods, checkout the Templates class file (https://github.com/serverless/serverless/blob/master/lib/Templates.js).
- **ProviderAws:** This class represents AWS as a provider. For a full API Reference on this class constructor and methods, checkout the ProviderAws class file (https://github.com/serverless/serverless/blob/master/lib/ProviderAws.js).

This Serverless instance has a property called `classes`, which is just an object that contains each of our classes. So you can init each class like this:

```
let Project = new S.classes.Project(...);
let Function = new S.classes.Function(...);
// and so on
```

Of course each class is constructed differently and requires different parameters. Checkout each class file constructor for better understanding of what's needed.

Serverless Methods

Most of the time, you won't need to initialize the Serverless classes mentioned earlier, instead, we're providing some helpful methods in the Serverless instance that makes it easy to get started manipulating Serverless projects.

getProject()

```
let Project = S.getProject();
```

Returns a **Project class instance** that contains all your project data. You can then use all the methods of the Project class for more power.

getProvider()

```
let aws = S.getProvider();
```

Returns a **Provider class instance** that contains powerful methods to interact with AWS.

GETTING STARTED ()
Backstory (/docs/backsto...
Overview (/docs/introdu...
Configuring AWS (/docs/...
Installing Serverless (/do...
DEEP DIVE ()
CLI REFERENCE ()
CREATING PLUGINS ()

updateConfig(config)

```
S.updateConfig({ projectPath: 'path/to/project' });
```

Updates the Serverless Instances configuration. Useful when you want to set a project to the Serverless instance by providing a `projectPath`.

Serverless Utilities

To give you even more power, we've included all of the utility functions we're using in our framework in the Serverless instance, giving you access to some common functionalities that are otherwise tedious to implement. You can access the utility functions through the `S.utils` property of the Serverless instance.

Below is a simple example utility function that checks whether a directory exists or not and returns a **Boolean**. To learn more about all of our utility functions, checkout the `utils` file. It's pretty well documented :)

```
S.utils.dirExistsSync('path/to/dir');
```

Suggest Edits (/docs/state/edit)

Putting It All Together

Some demos on how to use the Serverless API to accomplish common tasks.

Now that you've learned about creating plugins and the Serverless API, let's take a closer look at what your plugin can accomplish and what you have access to by using the Serverless API. We'll demonstrate the most common tasks and how using a combination of Serverless classes and methods will make that very easy.

It's recommended that you take a look at the classes files mentioned earlier and read the inline docs to get a basic idea of what each method does, since we'll be using these methods in this section.

Project, Functions and Other Assets

Here's a demo of how you can play around with your project assets (functions, endpoints, events, resources...etc)

index.js ()

```
// returns a Project instance
let Project = S.getProject();

// returns an array of Function instances. Which are all the functions that exist in your project
let allFunctions = Project.getAllFunctions();

// returns a Function instance
let specificFunction = Project.getFunction('myFuncName');

// returns `myFuncName`
specificFunction.getName();

// returns an array of Endpoint instances, which are all the endpoints of this function.
specificFunction.getAllEndpoints();

// returns an array of Event instances, which are all the events of this function.
specificFunction.getAllEvents();

// returns the name of the deployed lambda
let options = {
    stage: "dev",
    region: "us-east-1"
}
specificFunction.getDeployedName(options);

// returns an array of all the Endpoint instances in your project
let AllEndpoints = Project.getAllEndpoints();

// returns an array of all the Event instances in your project
let AllEvents = Project.getAllEvents();

// returns a Resources instance
let projectResources = Project.getResources();
```

Stages, Regions, and Variables

You can play around with project stages, regions and variables like this...

index.js ()

```
// getting the project
let Project = S.getProject();

// returns an array of Stage instances
let allStages = Project.getAllStages();

// returns a Stage instance
let specificStage = Project.getStage('dev');

// returns an array of Region instances that exist in that specific stage
let regionsInStage = specificStage.getAllRegions();

// or as a shortcut you can get stage regions using the Project instance instead
let regionsInStage = Project.getAllRegions('dev');

// returns a Variables instance which includes all the variables of a specific stage
let varsInStage = specificStage.getVariables();

// returns a specific Region instance in a specific stage
let region = specificStage.getRegion('us-east-1');

// returns a Variables instance which include all the variables of a specific region
let varsInRegion = region.getVariables();

// add a new region to a stage
let newRegion  = new S.classes.Region({ name: 'us-west-2' }, specificStage);
specificStage.setRegion(newRegion);

// add new variables to a region
let region = Project.getRegion('dev', 'us-east-1');
let newVars = {
    variableOne: "someValue",
     variableTwo: "someOtherValue"
}
region.addVariables(newVars);
```

Getting Populated Data

Sometimes you need project data to be populated with any referenced variables according to stage/region instead of returning the variable syntax to you (ie. `"name":"${myVar}"`). You can populate the referenced variables in your assets data like this...

index.js ()

```
// get the project as usual, then get a function and get its data populated with any referenced variables
let Project = S.getProject();
let myFunc = Project.getFunction('myFunc');
let options = {
    stage: "dev",
    region: "us-east-1"
}
let populatedFunc = myFunc.toObjectPopulated(options);

// you can also do that with any other asset, not just Functions, for example, an Endpoint...
let Endpoint = Project.getEndpoint('users/create~GET');

let options = {
    stage: "dev",
    region: "us-east-1"
}
let populatedEndpoint = Endpoint.toObjectPopulated(options);
```

▼ GETTING STARTED ()

Backstory (/docs/backsto...

Overview (/docs/introdu...

Configuring AWS (/docs/...

Installing Serverless (/do...

▸ DEEP DIVE ()

▸ CLI REFERENCE ()

▸ CREATING PLUGINS ()

Updating and Saving Data

To update and save data in the file system, you first need to get an object literal from an instance, then manipulate that object however you like, then update the instance using that new updated object. Then, you can save that new instance...

index.js ()

```
let Project = S.getProject();

// updating function data for demo. You can follow the same steps for other assets too (ie. Endpoints...etc)
let myFunc = Project.getFunction('myFunc');

// convert to object
myFuncObj = myFunc.toObject();

// make changes!
myFuncObj.timeout = 10;

// update the instance
myFunc.fromObject(myFuncObj);

// persist to file system
myFunc.save()
```

Suggest Edits (/docs/project/edit)