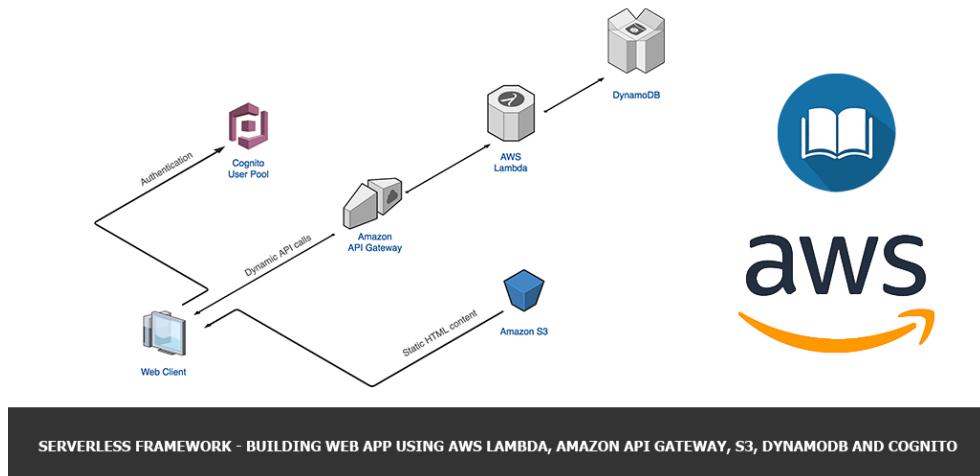


DEV-OPS-NOTES.COM

Dev & Ops tech blog

SERVERLESS FRAMEWORK – BUILDING WEB APP USING AWS LAMBDA, AMAZON API GATEWAY, S3, DYNAMODB AND COGNITO – PART 1

Voiced by [Amazon Polly](#) (<https://aws.amazon.com/polly/>)



SERVERLESS FRAMEWORK – BUILDING WEB APP USING AWS LAMBDA, AMAZON API GATEWAY, S3, DYNAMODB AND COGNITO

(<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-framework-Building-Web-App-using-AWS-Lambda-Amazon-API-Gateway-S3-DynamoDB-and-Cognito.png?ssl=1>)

Yesterday I decided to test [Serverless framework](#) (<https://serverless.com/>) and rewrite [AWS “Build a Serverless Web Application with AWS Lambda, Amazon API Gateway, Amazon S3, Amazon DynamoDB, and Amazon Cognito”](#) (<https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/>) tutorial.

In this tutorial we'll deploy the same [Wild Rides](http://www.wildrides.com/) (<http://www.wildrides.com/>) web application, but will do it in fully automated manner.

You can find full configuration and code in my [GitHub repo](#) (<https://github.com/andreivmaksimov/serverless-framework-aws-lambda-amazon-api-gateway-s3-dynamodb-and-cognito>). Use tag v1.0 for this article.

In [part 2](#) (<https://dev-ops-notes.com/cloud/serverless-framework-building-web-app-using-aws-lambda-amazon-api-gateway-s3-dynamodb-and-cognito-part-2/>) of this post you'll find how to replace API Gateway resources created in this article to Serverless framework events: .

APPLICATION ARCHITECTURE

Sure, to allow you to see all details in the same place, we need to copy some content from the original tutorial. So, our app will consist of:

- **Static Web Hosting** – Amazon S3 hosts static web resources including HTML, CSS, JavaScript, and image files which are loaded in the user's browser.
- **User Management** – Amazon Cognito provides user management and authentication functions to secure the backend

Lambda and API Gateway.

I'll keep the same modules structure for consistency:

- Static Web Hosting
- User Management
- Serverless Backend
- RESTful APIs
- Resource Termination and Next Steps

PROJECT SETUP

First of all, if you do not have Serverless framework installed, please, follow their [Quick Start \(<https://serverless.com/framework/docs/providers/aws/guide/quick-start/>\)](https://serverless.com/framework/docs/providers/aws/guide/quick-start/) guide. As soon as Serverless framework installed, we're ready to start. Let's create project directory and create a Serverless project template:

```
mkdir wild-rides-serverless-demo
cd wild-rides-serverless-demo
sls create -t aws-nodejs -n wild-rides-serverless-demo
```

At this moment of time you'll see two file inside our project directory:

- `handler.js` – this file contains demo Lambda function code
- `serverless.yaml` – this file contains Serverless project deployment configuration

Before continue this tutorial I strongly recommend to spend 30 minutes on looking through Serverless framework [AWS documentation \(<https://serverless.com/framework/docs/providers/aws/guide/>\)](https://serverless.com/framework/docs/providers/aws/guide/).

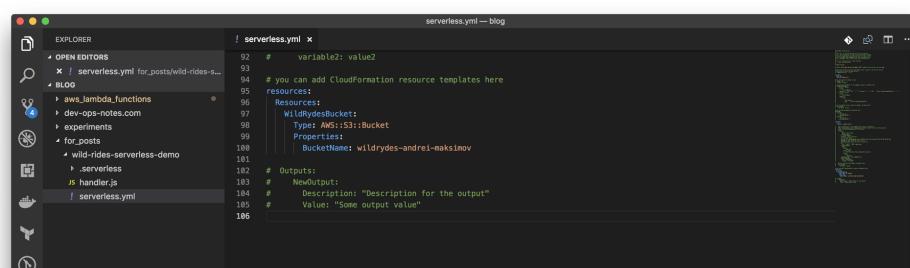
STATIC WEB HOSTING

In this module we'll configure Amazon Simple Storage Service (S3) to host the static resources for our web application. In subsequent modules we'll add dynamic functionality to these pages using JavaScript to call remote RESTful APIs built with AWS Lambda and Amazon API Gateway.

CREATE S3 BUCKET

To do so, first let's uncomment `resources:` section of the `serverless.yaml` file and change the name of [S3 bucket \(<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-s3-bucket.html>\)](#) to something like `wildrydes-firstname-lastname` (because S3 buckets must be globally unique). Also I changed `NewResource: CloudFormation` resource name to `WildRydesBucket`.

```
resources:
  Resources:
    WildRydesBucket:
      Type: AWS::S3::Bucket
      Properties:
        BucketName: wildrydes-andrei-maksimov
```



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1:bash
Serverless: Checking Stack create progress...
Serverless: Stack create finished...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading CloudFormation file to S3 (307 B)...
Serverless: Validating Template...
Serverless: Checking Stack update progress...
Serverless: Stack update finished...
Service Information
service: wild-rides-serverless-demo
region: us-east-1
stack: wild-rides-serverless-demo-dev
stage: dev
functions:
  None
endpoints:
  None
  functions:
    None
      function: wild-rides-serverless-demo-dev-hello
[amaksinov@awsmsk ~]$ Documents/projects/blog/_for_posts/wild-rides-serverless-demo]$
```

Ln 106, Col 1 Spaces: 2 UTF-8 LF YAML

(<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-S3-Bucket-Declaration.png?ssl=1>) Let's deploy this part of our stack:

```
sls deploy
```

This command will deploy demo lambda function, which was created by Serverless framework by default, and create our S3 bucket.

UPLOAD STATIC CONTENT

To upload Wild Rides website static content to our S3 bucket, we need to clone [aws-serverless-workshops](https://github.com/awslabs/aws-serverless-workshops) (<https://github.com/awslabs/aws-serverless-workshops>) repository:

```
git clone https://github.com/awslabs/aws-serverless-workshops/
```

Now we can copy website content:

```
aws s3 sync ./aws-serverless-workshops/WebApplication/1_StaticWebHosting/website s3://wildrydes-firstname-lastname
```

Right after that we may need to delete `aws-serverless-workshops` from our project:

```
rm -Rf ./aws-serverless-workshops
```

```

EXPLORER ! serverless.yml — blog
OPEN EDITORS ! serverless.yml for_posts/wild-rides-s...
BLOG
  aws_lambda_functions
  dev-ops-notes.com
  experiments
  for_posts
    wild-rides-serverless-demo
      .serverless
      aws-serverless-workshops
        JS handler.js
        ! serverless.yml
        ! serverless.yml
```

```

serverless.yml — blog
92 #   variable2: value2
93 # you can add CloudFormation resource templates here
95 # Resources:
96 #   WildRidesBucket:
97 #     Type: AWS::S3::Bucket
98 #     Properties:
99 #       BucketName: wildrydes-andrei-maksimov
100 # Outputs:
101 #   NewOutput:
102 #     Description: "Description for the output"
103 #     Value: "Some output value"
104 #   Resolving deltas: 100% (681/681), done.
105 #   Resolving deltas: 100% (681/681), done.
106 #   Resolving deltas: 100% (681/681), done.

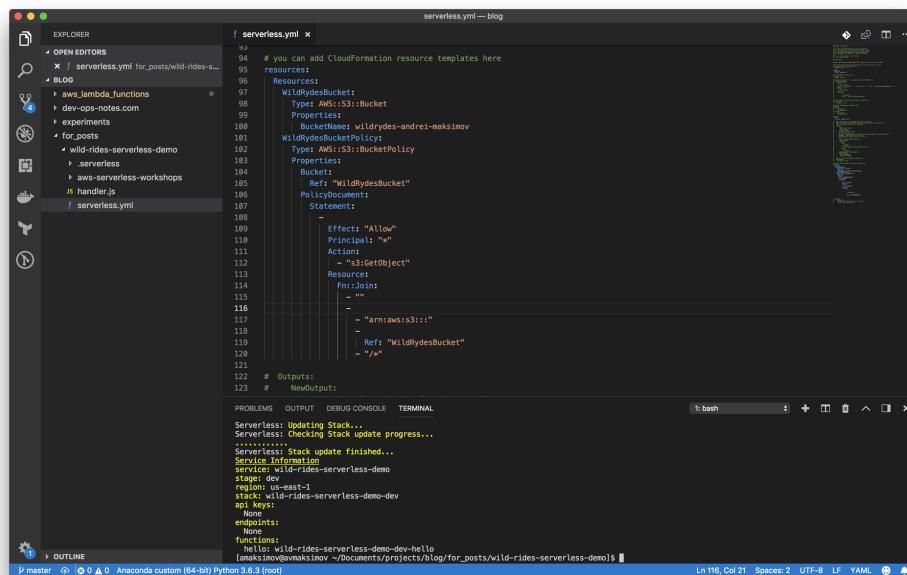
[amaksinov@awsmsk ~]$ Documents/projects/blog/_for_posts/wild-rides-serverless-demo]$ git clone https://github.com/awslabs/aws-serverless-workshops/
Cloning into 'aws-serverless-workshops'...
remote: Enumerating objects: 100%, 2623 objects
remote: Compressing objects: 100% (8/8), done.
remote: Total 2623 (delta 0), reused 1 (delta 0), pack-reused 2622
Resolving deltas: 100% (681/681), done.
[amaksinov@awsmsk ~]$ Documents/projects/blog/_for_posts/wild-rides-serverless-demo]$ aws s3 sync ./aws-serverless-workshops/WebApplication/1_StaticWe
hosting/wwwsite s3://wildrydes-andrei-maksimov
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1:bash
[amaksinov@awsmsk ~]\$ Documents/projects/blog/_for_posts/wild-rides-serverless-demo]\$

ADD A BUCKET POLICY TO ALLOW PUBLIC READS

Now we need to specify [S3 bucket policy](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-s3-policy.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-s3-policy.html>). To do so, add the following content to `resources:` section of `serverless.yaml` file:

```
WildRydesBucketPolicy:
  Type: AWS::S3::BucketPolicy
  Properties:
    Bucket:
      Ref: "WildRydesBucket"
    PolicyDocument:
      Statement:
        Effect: "Allow"
        Principal: "*"
        Action:
          - "s3:GetObject"
        Resource:
          Fn::Join:
            - ""
            - 
              - "arn:aws:s3:::"
            - 
              - Ref: "WildRydesBucket"
            - "//*"
```



(<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-S3-Bucket-Policy.png?ssl=1>)

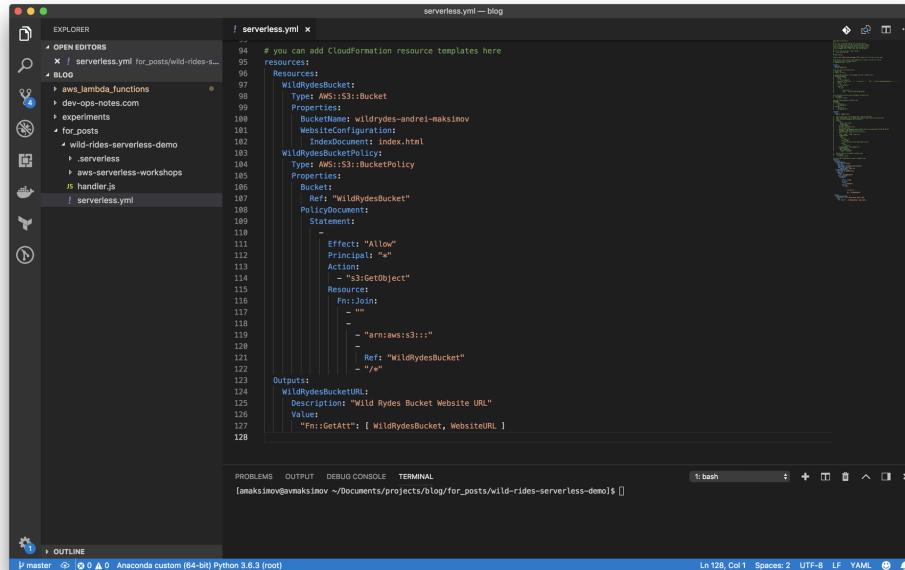
Deploy your changes:

```
sls deploy
```

```
WebsiteConfiguration:
IndexDocument: index.html
```

To display bucket website URL, uncomment the Outputs: section and add the following code there:

```
WildRydesBucketURL:
Description: "Wild Rydes Bucket Website URL"
Value:
"Fn::GetAtt": [ WildRydesBucket, WebsiteURL ]
```



```
serverless.yml
# you can add CloudFormation resource templates here
resources:
  Resources:
    WildRydesBucket:
      Type: AWS::S3::Bucket
      Properties:
        BucketName: wildrydes-andrei-maksimov
        WebsiteConfiguration:
          IndexDocument: index.html
        WildRydesBucketPolicy:
          Type: AWS::S3::BucketPolicy
          Properties:
            Bucket:
              Ref: "WildRydesBucket"
            PolicyDocument:
              Statement:
                - Effect: "Allow"
                  Principal: "*"
                  Action:
                    - "s3:GetObject"
                  Resources:
                    !Sub "arn:aws:s3:::${WildRydesBucket}/*"
                    - "arn:aws:s3:::${WildRydesBucket}"
                    - "arn:aws:s3:::${WildRydesBucket}/*"
            PolicyDocument:
              Statement:
                - Effect: "Allow"
                  Principal: "*"
                  Action:
                    - "s3:GetObject"
                  Resources:
                    !Sub "arn:aws:s3:::${WildRydesBucket}/*"
                    - "arn:aws:s3:::${WildRydesBucket}"
                    - "arn:aws:s3:::${WildRydesBucket}/*"
Outputs:
  WildRydesBucketURL:
    Description: "Wild Rydes Bucket Website URL"
    Value:
      "Fn::GetAtt": [ WildRydesBucket, WebsiteURL ]
```

(<https://i2.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-S3-Bucket-Website-Configuration.png?ssl=1>)

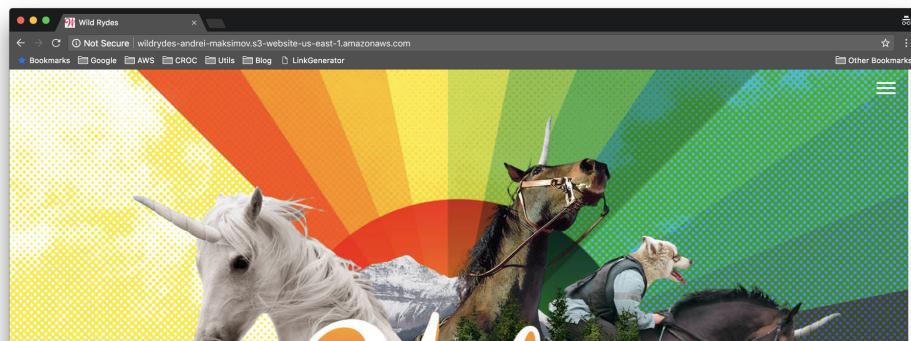
Let's redeploy the changes and test that static website is now accessible:

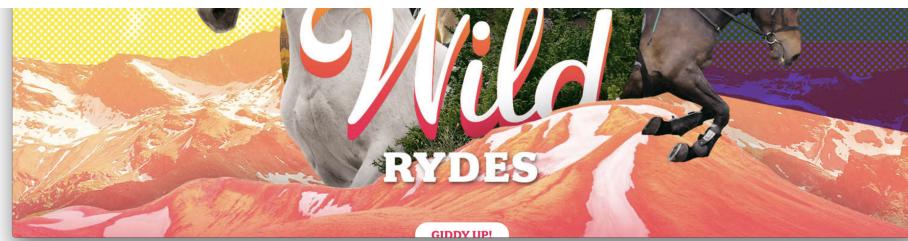
```
sls deploy
```

To display CloudFormation stack outputs run the following command:

```
sls info --verbose
```

You may open your website URL in the browser to check, that everything's deployed as expected:





<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-S3-Static-Website-Deployed.png?ssl=1>

USER MANAGEMENT

In this module we'll create an Amazon Cognito user pool to manage our users' accounts. We'll also deploy pages that enable customers to register as a new user, verify their email address, and sign into the site.

After users submit their registration, Amazon Cognito will send a confirmation email with a verification code to the address they provided. To confirm their account, users will return to your site and enter their email address and the verification code they received.

After users have a confirmed account (either using the email verification process or a manual confirmation through the console), they will be able to sign in. When users sign in, they enter their username (or email) and password. A JavaScript function then communicates with Amazon Cognito, authenticates using the Secure Remote Password protocol (SRP), and receives back a set of JSON Web Tokens (JWT). The JWTs contain claims about the identity of the user and will be used in the next module to authenticate against the RESTful API you build with Amazon API Gateway.

CREATE AN AMAZON COGNITO USER POOL

Amazon Cognito provides two different mechanisms for authenticating users:

- we can use Cognito User Pools to add sign-up and sign-in functionality to your application or use Cognito Identity Pools to authenticate users through social identity providers such as Facebook, Twitter, or Amazon, with SAML identity solutions
- we can use our own identity system.

Here we'll use a user pool as the backend for the provided registration and sign-in pages. First, let's create [Cognito User Pool](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-cognito-userpool.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-cognito-userpool.html>) by adding it's declaration to `resources:` section of our `serverless.yaml` file:

```
WildRydesCognitoUserPool:
  Type: AWS::Cognito::UserPool
  Properties:
    UserPoolName: WildRydes
```

```
serverless.yaml — blog
serverless.yaml x
1 OPEN EDITORS
2 ! serverless.yaml for_posts/wild-rides-s...
3 BLOG
4
5 ! serverless.yaml
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
```

```

131      "Fn::GetAtt": [ "WildRydesBucket", "WebsiteURL" ]
132
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
Serverless: Stack update finished...
Service Information
  Service: wild-rides-serverless-demo
    stage: dev
    region: us-east-1
    endpoint: wild-rides-serverless-demo-dev
    api keys:
      endpoints:
        None
      functions:
        hello: wild-rides-serverless-demo-dev-hello
[anaksinov@anaksinov ~]$

```

<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Configuration.png?ssl=1>

Let's deploy our changes:

sls deploy

ADD AN APP TO YOUR USER POOL

Next, we need to create [Cognito User Pool Client](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-cognito-userpoolclient.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-cognito-userpoolclient.html>). I do not understand, why they call it App Client in web console. To do so, as usual, we need to add new resource to resource: section in serverless.yaml file:

```

WildRydesCognitoUserPoolClient:
  Type: AWS::Cognito::UserPoolClient
  Properties:
    ClientName: WildRydesWebApp
    GenerateSecret: false
    UserPoolId:
      Ref: "WildRydesCognitoUserPool"

```

```

serverless.yml
serverless.yml — blog
110      Effect: "Allow"
111      Principal: "*"
112      Action:
113        - "s3:GetObject"
114      Resources:
115        Fn::ImportValue:
116          - "arnaws:s3:::"
117          - "arnaws:s3:::WildRydesBucket"
118
WildRydesCognitoUserPool:
119  Type: AWS::Cognito::UserPool
120  Properties:
121    UserPoolName: WildRydes
122    WildRydesCognitoUserPoolClient:
123      Type: AWS::Cognito::UserPoolClient
124      Properties:
125        ClientName: WildRydesWebApp
126        GenerateSecret: false
127        UserPoolId:
128          Ref: "WildRydesCognitoUserPool"
129
Outputs:
130   WildRydesBucketURL:
131     Description: "Wild Rydes Bucket Website URL"
132     Value:
133       "Fn::GetAtt": [ "WildRydesBucket", "WebsiteURL" ]
134
135
136
137
138
139
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
Serverless: Stack update finished...
Service Information
  Service: wild-rides-serverless-demo
    stage: dev
    region: us-east-1
    endpoint: wild-rides-serverless-demo-dev
    api keys:
      endpoints:
        None
      functions:
        hello: wild-rides-serverless-demo-dev-hello
Serverless: Removing old service artifacts from S3...
[anaksinov@anaksinov ~]$

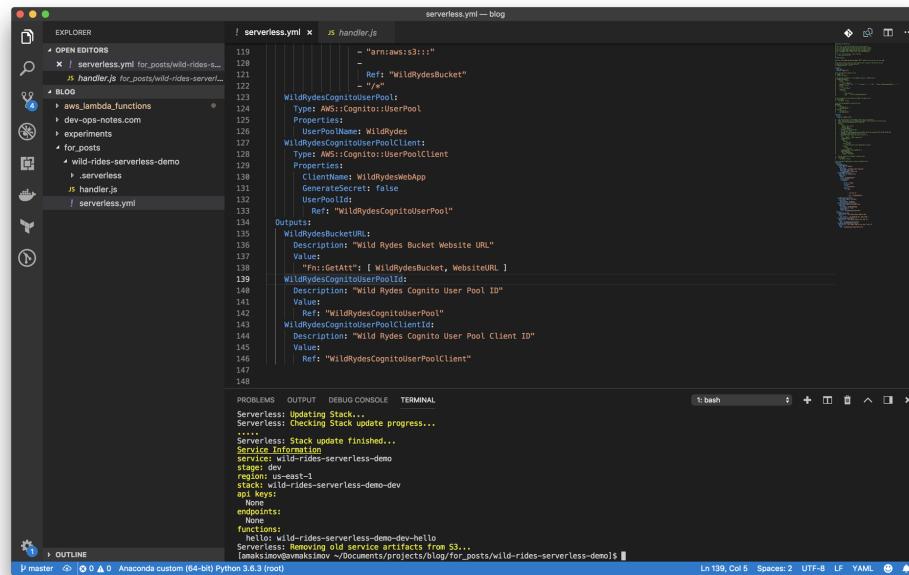
```

<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Configuration.png?ssl=1>

UPDATE THE CONFIG.JS FILE IN YOUR WEBSITE BUCKET

In this section of AWS tutorial they're asking us to make changes in `js/config.js` file. More over, we'll need `userPoolId` and `userPoolClientId`. As we're not using web console, let's request them as `Outputs:` in our `serverless.yaml` file:

```
WildRydesCognitoUserPoolId:
  Description: "Wild Rydes Cognito User Pool ID"
  Value:
    Ref: "WildRydesCognitoUserPool"
WildRydesCognitoUserPoolClientId:
  Description: "Wild Rydes Cognito User Pool Client ID"
  Value:
    Ref: "WildRydesCognitoUserPoolClient"
```



(<https://i2.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Configuration-2.png?ssl=1>)

Now we need to redeploy our changes:

```
sls deploy
```

And display IDs:

```
sls info --verbose
```

Now we're ready to make changes in the `js/config.js` file. First of all let's download it:

```
aws s3 cp s3://wildrydes-firstname-lastname/js/config.js ./
```

After that, fill `userPoolId`, `userPoolClientId` and `region` (all this information available from `sls info --verbose`

```

1 // Amazon Cognito
2   cognito: {
3     userPoolId: 'us-east-1_ZrpyYffN', // e.g. us-east-2_uxboGSpAb
4     userPoolClient: '54dr1gj9v6kpbign2vp7lmqbb', // e.g. 25ddkej4vhfsvrulhpf17n4v
5     region: 'us-east-1' // e.g. us-east-2
6   },
7   api: {
8     invokeUrl: '' // e.g. https://rc7nyt4tg1.execute-api.us-west-2.amazonaws.com/prod
9   }
10 }
11

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

region: us-east-1
stack: wild-rides-serverless-demo-dev
app: None
endpoints:
functions:
  hello: wild-rides-serverless-demo-dev-hello
Stack Outputs
WildrydesCognitoUrl: https://wildrydes-general-maksimov.s3-website-us-east-1.amazonaws.com
WildrydesCognitoUserPoolClient: 54dr1gj9v6kpbign2vp7lmqbb
HelloLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:121258976243:function:wild-rides-serverless-demo-dev-hello:4
WildrydesCognitoUserPoolId: us-east-1_ZrpyYffN
WildrydesCognitoUserPoolRegion: us-east-1

```

[maksimov@maksimov ~](master) Python 3.6.3 (root)

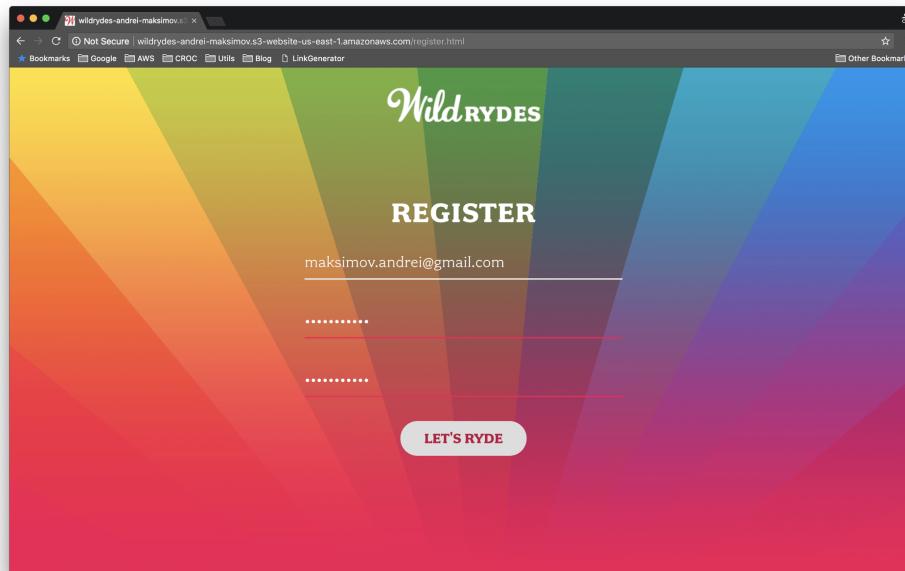
(<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Configuration-3.png?ssl=1>)

Save the file and upload it back:

```
aws s3 cp ./config.js s3://wildrydes-firstname-lastname/js/config.js
```

VALIDATE YOUR IMPLEMENTATION

Now we're ready to validate our Cognito configuration. I'll not copy-paste it from AWS tutorial. Here's the [link](https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/module-2/) (<https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/module-2/>). You need last step and manual user verification. It means, when you'll register new user:



(<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Configuration-4.png?ssl=1>)

The screenshot shows the AWS User Pools console for a pool named 'WildRydes'. The left sidebar has 'General settings' selected under 'Users and groups'. The main area shows a table with one row for the user 'maksimov.andrei@gmail.com'. The user is marked as 'Enabled' and 'UNCONFIRMED'. The 'Created' and 'Updated' columns both show 'Sep 9, 2018 2:55:05 PM'. Below the table are buttons for 'Import users', 'Create user', and a search bar.

(<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Testing-Registered-User.png?ssl=1>) And confirm registered user manually (click on the user and use **Confirm user** button):

This screenshot shows the detailed view for the user 'maksimov.andrei@gmail.com'. The left sidebar shows 'App integration' selected. In the center, there's a large button labeled 'Confirm user' which is highlighted in blue. Other buttons include 'Add to group', 'Enable SMS MFA', and 'Disable user'. Below the buttons, the user's status is shown as 'Enabled / UNCONFIRMED'. The 'Last Modified' and 'Created' fields are also visible. At the bottom, a table shows 'Device Key', 'Name', 'Last IP', 'Remembered', 'SDK', and 'Last Seen' with no devices found.

(<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Testing-Confirm-Registered-User.png?ssl=1>)

After user confirmation, we're able to login using `/signin.html` page where we will be redirected to `/ride.html`:

The screenshot shows a web browser window for 'Wild Rydes'. The address bar shows the URL 'wildrydes-andrei-maksimov.s3-website-us-east-1.amazonaws.com/ride.html'. The page content includes a success message 'Successfully Authenticated!' and a map of a city with a pickup location selector. A sidebar on the right displays a token: 'Welcome! Click the map to set your pickup location. You are authenticated. Click to see your auth token.'



<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Testing-Login.png?ssl=1>

SERVERLESS SERVICE BACKEND

In this module we'll use AWS Lambda and Amazon DynamoDB to build a backend process for handling requests for your web application. The browser application that you deployed in the previous step allows users to request that a unicorn be sent to a location of their choice. In order to fulfill those requests, the JavaScript running in the browser will need to invoke a service running in the cloud.

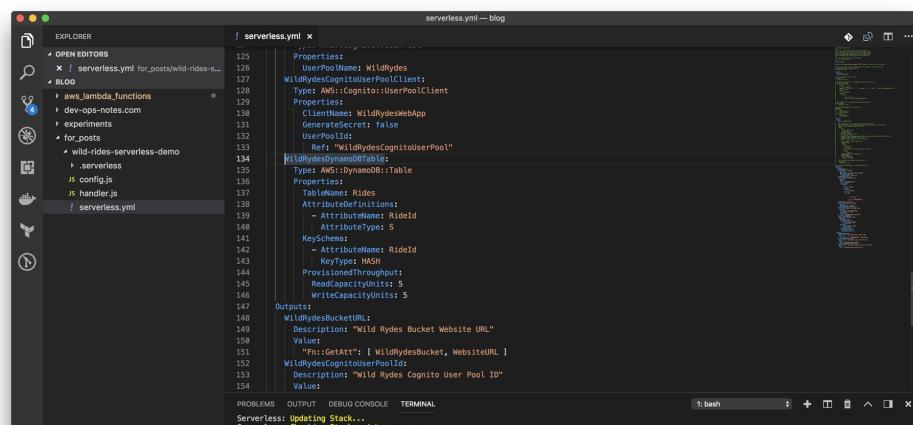
You'll implement a Lambda function that will be invoked each time a user requests a unicorn. The function will select a unicorn from the fleet, record the request in a DynamoDB table and then respond to the front-end application with details about the unicorn being dispatched.

The function is invoked from the browser using Amazon API Gateway.

CREATE AN AMAZON DYNAMODB TABLE

As in the original tutorial we'll call your table `Rides` and give it a partition key called `RideId` with type String. To do so, we need to add the [DynamoDB](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-dynamodb-table.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-dynamodb-table.html>) resource to `resources:` section of `serverless.yaml` file:

```
WildRydesDynamoDBTable:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: Rides
    AttributeDefinitions:
      - AttributeName: RideId
        AttributeType: S
    KeySchema:
      - AttributeName: RideId
        KeyType: HASH
    ProvisionedThroughput:
      ReadCapacityUnits: 5
      WriteCapacityUnits: 5
```



```
Serverless: Stack update finished...
Service Information
service: wild-rides-serverless-demo
region: us-east-1
stage: wild-rides-serverless-demo-dev
version: 1
last updated: None
endpoints: None
functions:
  hello: Wild-rides-serverless-demo-dev-hello
Serverless: Removing old service artifacts from S3...
[anaksinov@vmsimov ~]Documents/projects/blog/or_posts/wild-rides-serverless-demo$
```

<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-DynamoDB-Configuration.png?ssl=1>

And of cause we need to redeploy our stack:

```
sls deploy
```

Finally, we need to get DynamoDB **ARN**. And you already know how to do it. In the `outputs:` of `resources:` section in `serverless.yaml` file we need to add the following:

```
WildRydesDynamoDbARN:
  Description: "Wild Rydes DynamoDB ARN"
  Value:
    "Fn::GetAtt": [ WildRydesDynamoDBTable, Arn ]
```

```
serverless.yml — blog
serverless.yml x
OPEN EDITORS
  ✘ serverless.yml for_posts/wild-rides-s...
BLOG
  aws_lambda_functions
  dev-ops-notes.com
  experiments
  for_posts
    wild-rides-serverless-demo
      .serverless
      config.json
      handler.js
      ! serverless.yml
serverless.yml

  Outputs:
    WildRydesBucketURL:
      Description: Wild Rydes Bucket Website URL
      Value:
        "Fn::GetAtt": [ WildRydesBucket, WebsiteURL ]
    WildRydesCognitoUserPoolId:
      Description: Wild Rydes Cognito User Pool ID
      Value:
        "Fn::GetAtt": [ WildRydesCognitoUserPool, UserPoolId ]
    WildRydesCognitoUserPoolClient:
      Description: Wild Rydes Cognito User Pool Client ID
      Value:
        "Fn::GetAtt": [ WildRydesCognitoUserPoolClient, ClientId ]
    WildRydesDynamoDbArn:
      Description: Wild Rydes DynamoDB ARN
      Value:
        "Fn::GetAtt": [ WildRydesDynamoDBTable, Arn ]
```

<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-DynamoDB-Configuration-ARN.png?ssl=1>

CREATE AN IAM ROLE FOR YOUR LAMBDA FUNCTION

Every Lambda function has an IAM role associated with it. This role defines what other AWS services the function is allowed to interact with. For the purposes of this tutorial, you'll need to create an IAM role that grants your Lambda function permission to write logs to Amazon CloudWatch Logs and access to write items to your DynamoDB table.

To do so we need to create Lambda function IAM Role and assign it with a policy:

```

AssumeRolePolicyDocument:
  Version: '2012-10-17'
  Statement:
    - Effect: Allow
    Principal:
      Service:
        - lambda.amazonaws.com
    Action: sts:AssumeRole
Policies:
  - PolicyName: DynamoDBWriteAccess
  PolicyDocument:
    Version: '2012-10-17'
    Statement:
      - Effect: Allow
      Action:
        - logs>CreateLogGroup
        - logs>CreateLogStream
        - logs:PutLogEvents
    Resource:
      - 'Fn::Join':
        - ':'
        -
        - 'arn:aws:logs'
        - Ref: 'AWS::Region'
        - Ref: 'AWS::AccountId'
        - 'log-group:/aws/lambda/*:*:*'
  - Effect: Allow
  Action:
    - dynamodb:PutItem
  Resource:
    'Fn::GetAtt': [ WildRydesDynamoDBTable, Arn ]

```

```

serverless.yml x serverless.yml — blog
144   KeyType: HASH
145   ProvisionedThroughput:
146     ReadCapacityUnits: 5
147     WriteCapacityUnits: 5
148   WildRydesLambda:
149     Type: AWS::Lambda::Function
150     Properties:
151       RoleName: WildRydesLambda
152       AssumeRolePolicyDocument:
153         Version: '2012-10-17'
154       Statement:
155         - Effect: Allow
156           Principal:
157             Service:
158               - lambda.amazonaws.com
159           Action: sts:AssumeRole
160       Policies:
161         - PolicyName: DynamoDBWriteAccess
162         - PolicyDocument:
163           Version: '2012-10-17'
164           Statement:
165             - Effect: Allow
166               Action:
167                 - Logs>CreateLogGroup
168                 - Logs>CreateLogStream
169                 - Logs:PutLogEvents
170             Resource:
171               'Fn::Join':
172                 - ':'
173                 -
174                 - 'arn:aws:logs'
175                 - Ref: 'AWS::Region'
176                 - Ref: 'AWS::AccountId'
177                 - 'log-group:/aws/lambda/*:*:*'
178             - Effect: Allow
179               Action:
180                 - dynamodb:PutItem
181             Resource:
182               'Fn::GetAtt': [ WildRydesDynamoDBTable, Arn ]
183             Outputs:
184               WildRydesBucketURL:
185                 Description: Wild Rydes Bucket Website URL
186                 Value:
187                   "Fn::GetAtt": [ WildRydesBucket, WebsiteURL ]
188               WildRydesCognitoUserPoolId:
189                 Description: "Wild Rydes Cognito User Pool ID"

```

<https://i2.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Lambda-Function-Role-Policy.png?ssl=1>

CREATE A LAMBDA FUNCTION FOR HANDLING REQUESTS

Woohoo! It's time to create our Lambda function and grant it with appropriate privileges to have access to our DynamoDB! Let's start from the Lambda function itself. Yes, we're changing original tutorial order a little bit and create Lambda function first. In the next section, I'll show you, how you can easily attach necessary permissions.

AWS Lambda will run your code in response to events such as an HTTP request. In this step you'll build the core function that will process API requests from the web application to dispatch a unicorn. In the following steps you'll use Amazon API Gateway to create a RESTful API that will expose an HTTP endpoint that can be invoked from your users' browsers. You'll then connect the Lambda function you create in this step to that API in order to create a fully functional backend for your web application.

But right now let's create Lambda function called `RequestUnicorn` that will process the API requests. First of all we need to declare it in `functions:` section in `serverless.yaml` file (replace `hello` function):

```
functions:
  RequestUnicorn:
    handler: handler.handler
    role: WildRidesLambdaRole
```

```
serverless.yml
functions:
  RequestUnicorn:
    handler: handler.handler
    role: WildRidesLambdaRole
```

(<https://i2.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Lambda-Function-Declaration.png?ssl=1>)

This will tell Serverless framework to create our function with appropriate name and specified entry point. Please, take a look under `handler:` declaration. First `handler` specifies the filename where to look our function content. The second `handler` is the name of exported function to call each time Lambda function is triggered.

Now we need to change the code of our lambda function. Open `handler.js` file and replace everything with a provided [requestUnicorn.js](https://github.com/awslabs/aws-serverless-workshops/blob/master/WebApplication/3_ServerlessBackend/requestUnicorn.js) code example as it is.

```
handler.js
const randomBytes = require('crypto').randomBytes;
```

```

 9      Name: 'Bucephalus',
10      Color: 'Golden',
11      Gender: 'Male',
12    },
13    {
14      Name: 'Shadowfax',
15      Color: 'White',
16      Gender: 'Male',
17    },
18    {
19      Name: 'Rocinante',
20      Color: 'Yellow',
21      Gender: 'Female',
22    },
23  ];
24
25 exports.handler = event, context, callback => {
26   if (!event.requestContext.authorizer) {
27     throwError('Authorization not configured', context.awsRequestId, callback);
28   }
29   return;
30
31   const rideId = token(String(randomBytes(10)));
32   console.log(`Received event ${rideId}`);
33
34   // Because we're using a Cognito User Pools authorizer, all of the claims
35   // included in the authentication token are provided in the request context.
36   // This includes the username as well as other attributes.
37   const username = event.requestContext.authorizer.claims['cognito:username'];
38
39   // The body field of the event in a proxy integration is a raw string.
40   // In order to extract meaningful values, we need to first parse this string
41   // into an object. A more robust implementation might inspect the Content-Type
42   // header first and use a different parsing strategy based on that value.
43   const requestBody = JSON.parse(event.body);
44
45   const pickupLocation = requestBody.PickupLocation;

```

Ln 119, Col 2 Spaces: 4 UTF-8 LF JavaScript

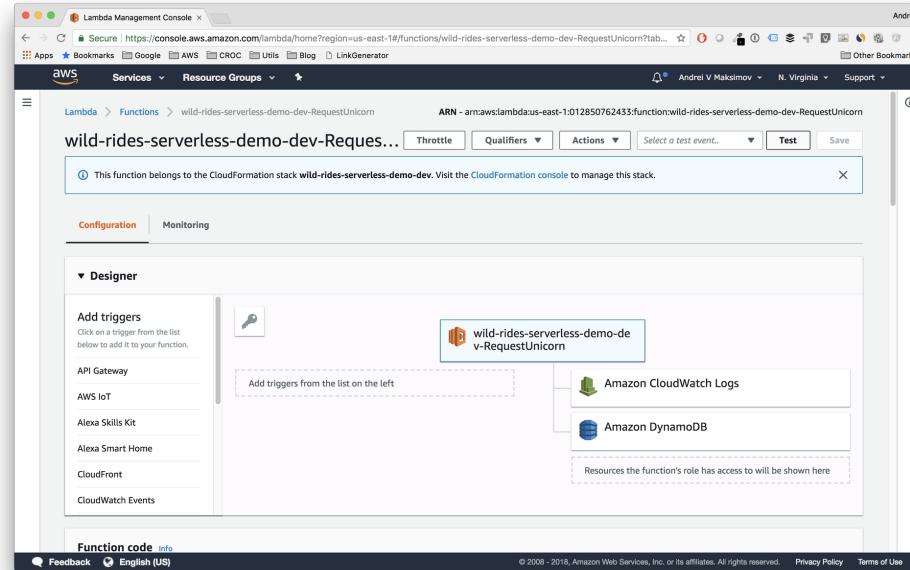
<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Lambda-Function-Code.png?ssl=1>

Now it is definitely a time to redeploy our stack:

sls deploy

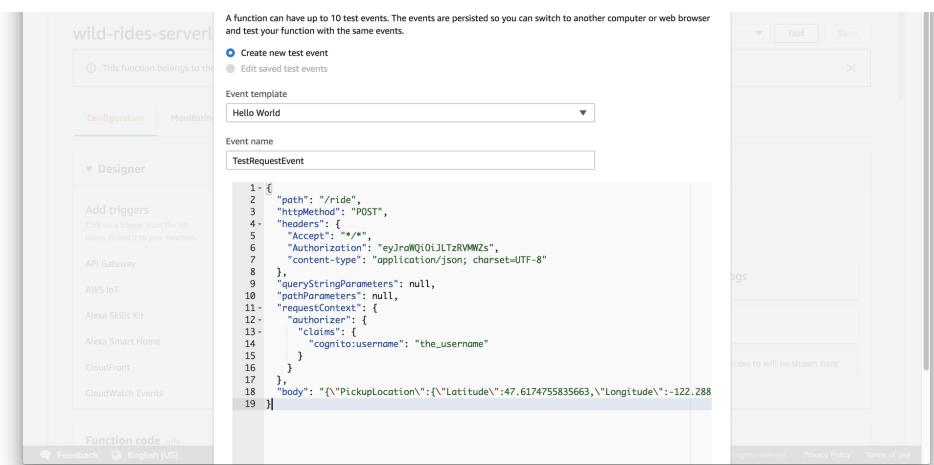
VALIDATE YOUR IMPLEMENTATION

To validate our current service implementation we need to follow official instructions from AWS tutorial. Let's open our Lambda function in AWS console. As you can see, it has permissions to access to CloudWatch and DynamoDB.



<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Lambda-Function-Details.png?ssl=1>

Let's create a test by clicking **Test** button. Fill the form as it shown below:



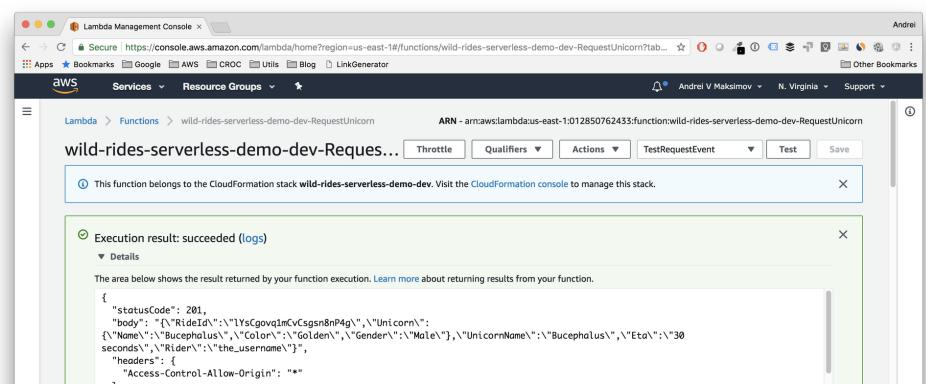
(<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Lambda-Function-Testing.png?ssl=1>)

Test name is `TestRequestEvent`. Test message body:

```
{
  "path": "/ride",
  "httpMethod": "POST",
  "headers": {
    "Accept": "*/*",
    "Authorization": "eyJraWQiOiJLTzRVMWZs",
    "content-type": "application/json; charset=UTF-8"
  },
  "queryStringParameters": null,
  "pathParameters": null,
  "requestContext": {
    "authorizer": {
      "claims": {
        "cognito:username": "the_username"
      }
    }
  },
  "body": "{\"PickupLocation\":{\"Latitude\":47.6174755835663,\"Longitude\":
\"-122.28837066650185}}"
}
```

Click **Create** button to create test.

Click **Test** button once more again to see successful test results:





<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Lambda-Function-Testing-Results.png?ssl=1>

RESTFUL APIs

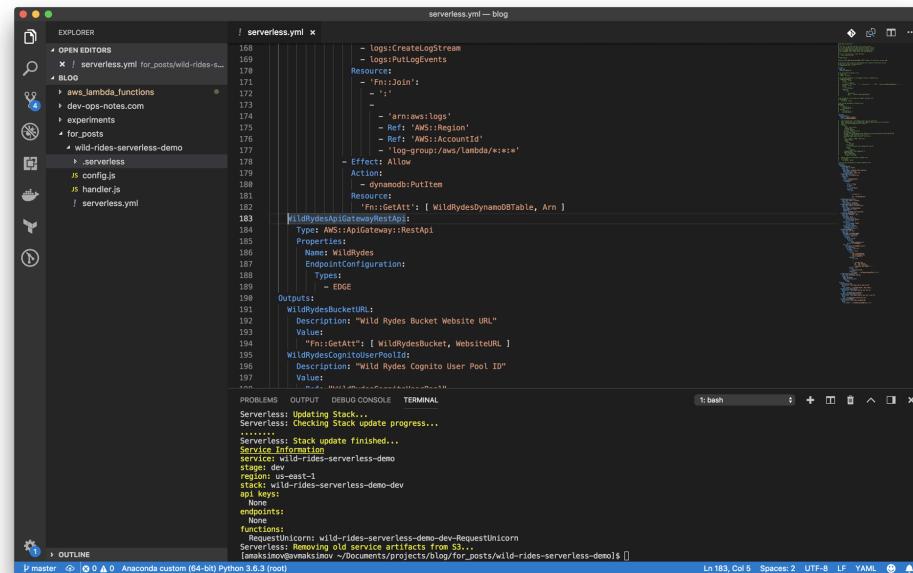
Now it's time to use [Amazon API Gateway](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-restapi.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-restapi.html>) to expose the Lambda function you built in the previous steps as a RESTful API. This API will be accessible on the public Internet. It will be secured using the Amazon Cognito user pool you created in the previously. Using this configuration you will then turn your statically hosted website into a dynamic web application by adding client-side JavaScript that makes AJAX calls to the exposed APIs.

The static website you deployed in the first steps already has a page configured to interact with the API you'll build in this module. The page at `/ride.html` has a simple map-based interface for requesting a unicorn ride. After authenticating using the `/signin.html` page, your users will be able to select their pickup location by clicking a point on the map and then requesting a ride by choosing the "Request Unicorn" button in the upper right corner.

CREATE A NEW REST API

All we need to do now – is to specify [Amazon API Gateway REST API](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-restapi.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-restapi.html>) resource:

```
WildRydesApiGatewayRestApi:
  Type: AWS::ApiGateway::RestApi
  Properties:
    Name: WildRydes
  EndpointConfiguration:
    Types:
      - EDGE
```



<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-1101-Configuration.png?ssl=1>

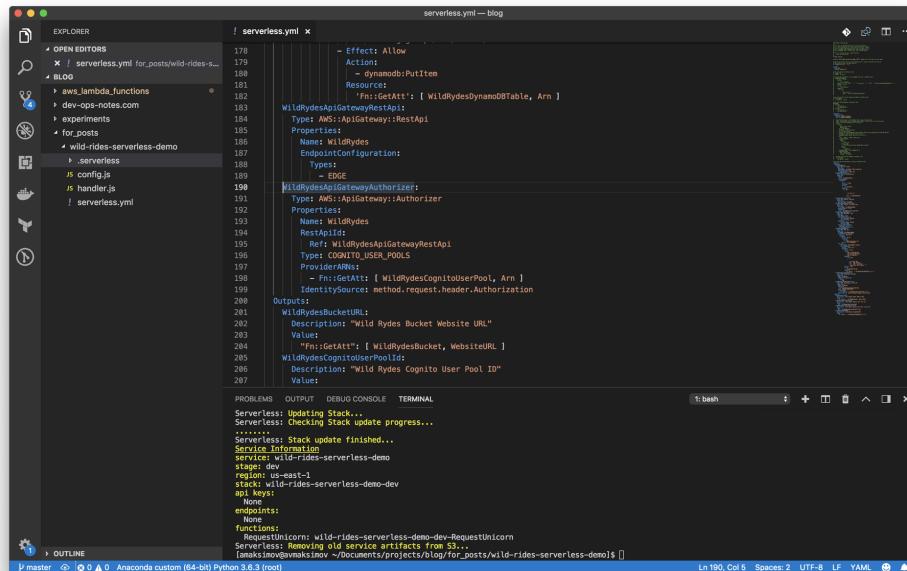
Redeploy your stack to get AWS Api Gateway up and running.

```
sls deploy
```

CREATE A COGNITO USER POOLS AUTHORIZER

Amazon API Gateway can use the JWT tokens returned by Cognito User Pools to authenticate API calls. In this step you'll configure an authorizer for your API to use the user pool you created earlier. First of all we need to create [ApiGatewayAuthorizer](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-authorizer.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-authorizer.html>) in our resources: section in `serverless.yaml` file:

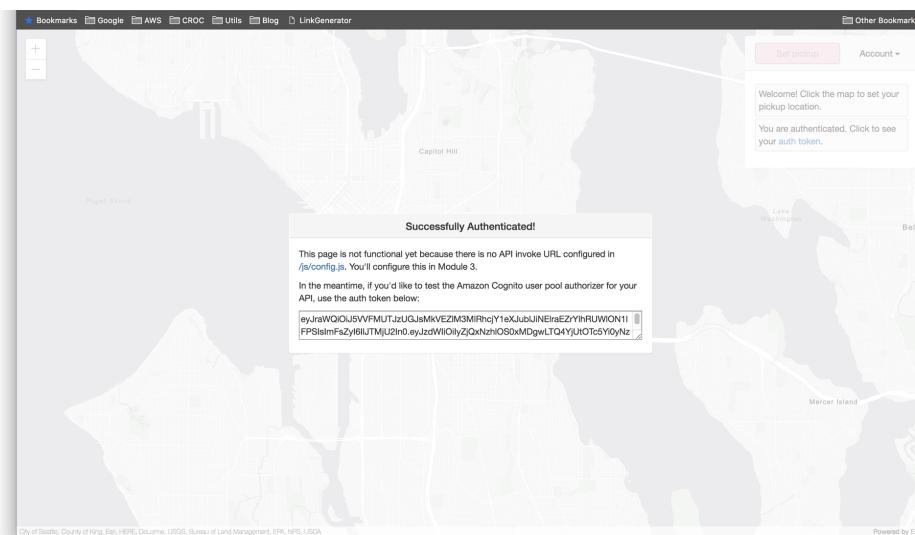
```
WildRydesApiGatewayAuthorizer:
  Type: AWS::ApiGateway::Authorizer
  Properties:
    Name: WildRydes
    RestApiId:
      Ref: WildRydesApiGatewayRestApi
    Type: COGNITO_USER_POOLS
    ProviderARNs:
      - Fn::GetAtt: [ WildRydesCognitoUserPool, Arn ]
    IdentitySource: method.request.header.Authorization
```



<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-Authorizer.png?ssl=1>

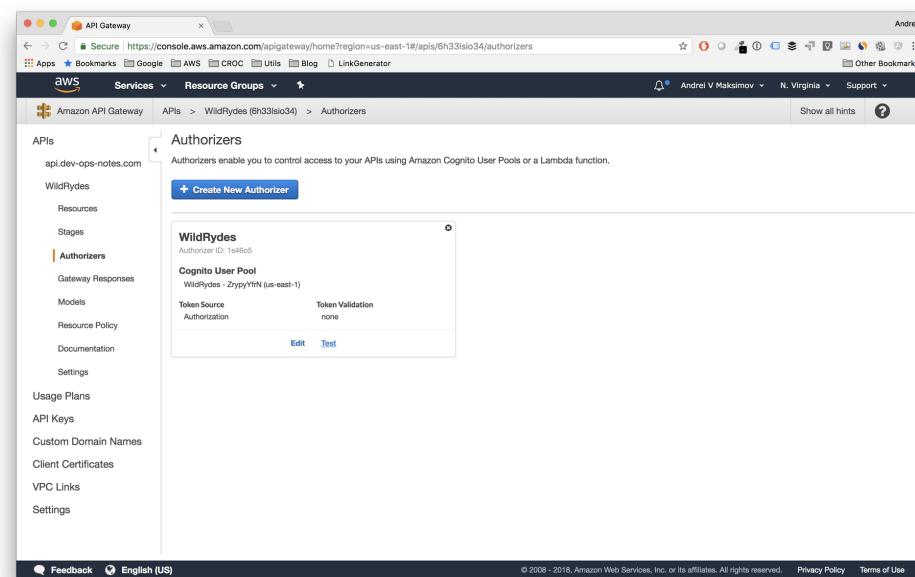
Redeploy your stack to setup Api Gateway Authorizer:

```
sls deploy
```



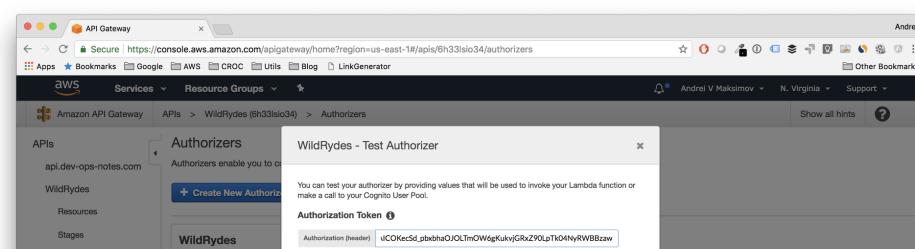
<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Cognito-User-Pool-Client-Testing-Login.png?ssl=1>

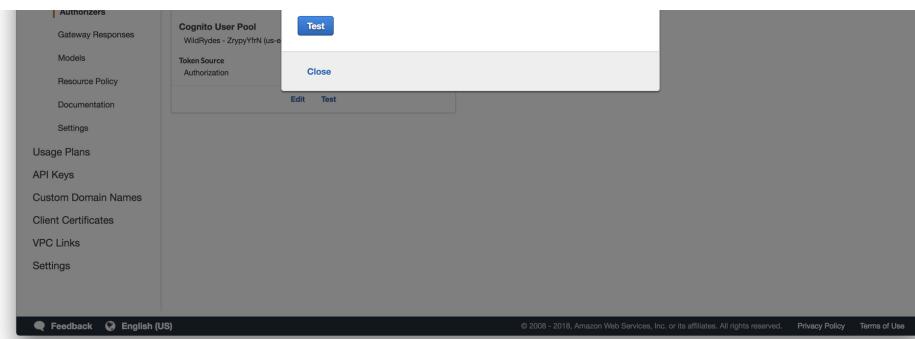
Go to **API Gateway** service in AWS console, open **WildRydes API Gateway** and select **Authorizers** in left menu. Click **Test** link:



<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-Authorizer-Test.png?ssl=1>

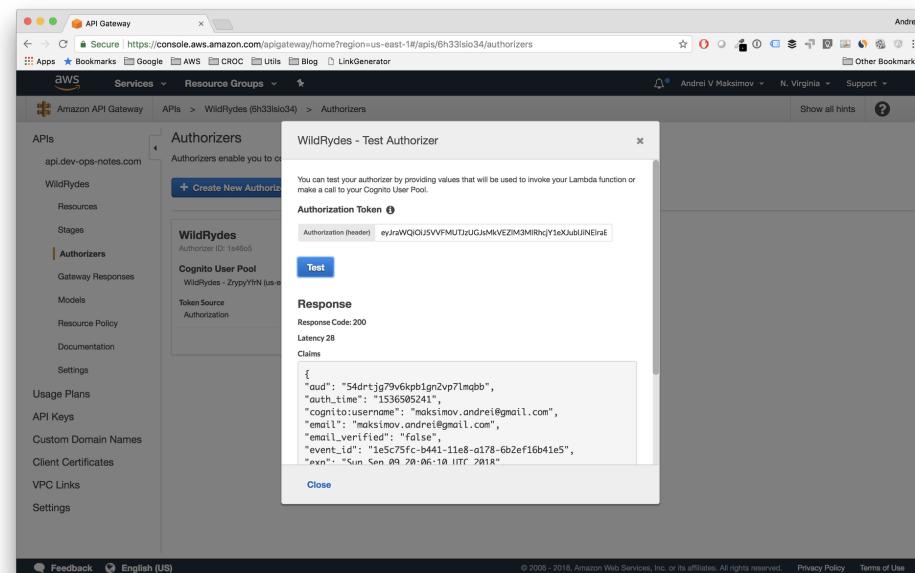
Paste your Authorization Token in the field and press **Test** button:





(<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-Authorizer-Test-Token.png?ssl=1>)

If you did everything correctly, you'll see successful response:



(<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-Authorizer-Test-Result.png?ssl=1>)

CREATE A NEW RESOURCE AND METHOD

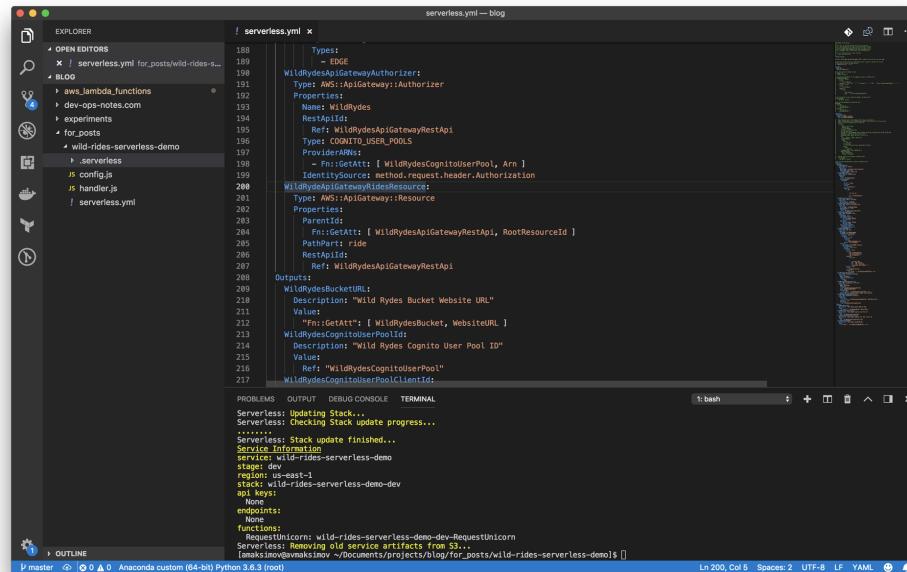
Next we need to create a new [API Gateway Resource](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-resource.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-resource.html>) called `/ride` within your API. Then create a [POST API Gateway Method](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-method.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-method.html>) for that resource and configure it to use a Lambda proxy integration backed by the RequestUnicorn Lambda function.

No problem, here's its declaration:

```

Fn::GetAtt: [ WildRydesApiGatewayRestApi, RootResourceId ]
PathPart: ride
RestApiId:
Ref: WildRydesApiGatewayRestApi

```



(<https://i2.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-Resource.png?ssl=1>)

In the official tutorial we may click **Enable CORS** checkbox to Enable CORS for a needed resource. When you're dealing with automation it is not always so easy. It means you need to implement “Enable CORS” functionality for an API Gateway Resource yourself. Somebody on StackOverflow [already did it for us](#) (<https://stackoverflow.com/questions/40292888/enable-cors-for-api-gateway-in-cloudformation-template>), so we thankfully will take resource description example and adopt it for our needs:

```
--  
RestApiId:  
  Ref: WildRydesApiGatewayRestApi  
ResourceId:  
  Ref: WildRydeApiGatewayRidesResource  
HttpMethod: OPTIONS  
Integration:  
  IntegrationResponses:  
    - StatusCode: 200  
      ResponseParameters:  
        method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-  
Date,Authorization,X-Api-Key,X-Amz-Security-Token'"  
        method.response.header.Access-Control-Allow-Methods: "'POST,OPTIONS'"  
        method.response.header.Access-Control-Allow-Origin: "'*'"  
      ResponseTemplates:  
        application/json: ''  
    PassthroughBehavior: WHEN_NO_MATCH  
    RequestTemplates:  
      application/json: '{"statusCode": 200}'  
    Type: MOCK  
  MethodResponses:  
    - StatusCode: 200  
      ResponseModels:  
        application/json: 'Empty'  
      ResponseParameters:  
        method.response.header.Access-Control-Allow-Headers: false  
        method.response.header.Access-Control-Allow-Methods: false  
        method.response.header.Access-Control-Allow-Origin: false
```

Redeploy your stack to check that everything's working as expected:

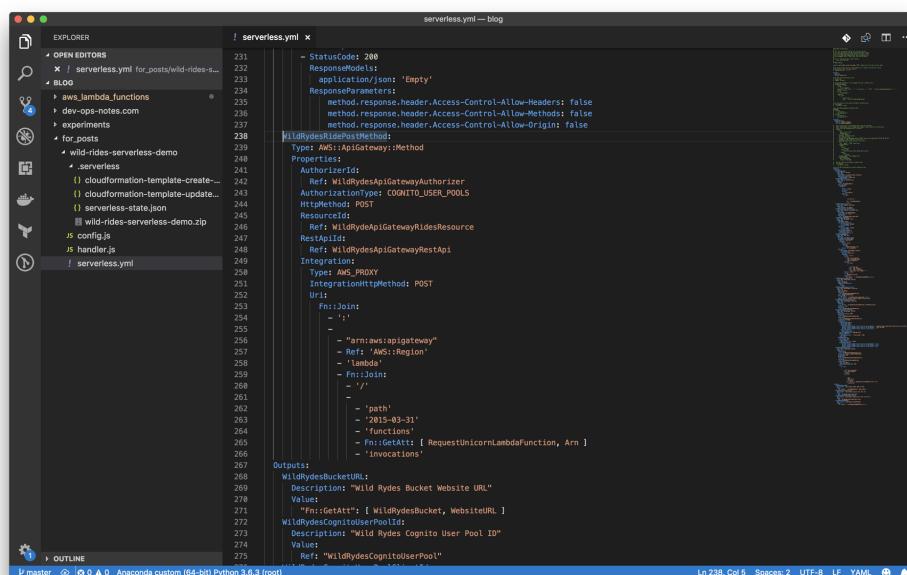
```
sls deploy
```

Now we need to create the actual POST method:

```

Ref: WildRydesApiGatewayAuthorizer
AuthorizationType: COGNITO_USER_POOLS
HttpMethod: POST
ResourceId:
Ref: WildRydeApiGatewayRidesResource
RestApiId:
Ref: WildRydesApiGatewayRestApi
Integration:
Type: AWS_PROXY
IntegrationHttpMethod: POST
Uri:
Fn::Join:
  - ':'
  -
  - "arn:aws:apigateway"
  - Ref: 'AWS::Region'
  - 'lambda'
  - Fn::Join:
    - '/'
    -
    - 'path'
    - '2015-03-31'
    - 'functions'
  - Fn::GetAtt: [ RequestUnicornLambdaFunction, Arn ]
  - 'invocations'

```



(<https://i1.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-Method-With-Lambda-Integration.png?ssl=1>)

This will create protected by API Gateway authorization POST method, which will call Lambda function for authorized users.

CREATE YOUR API DEPLOYMENT

Mostly we're done. All we need to do is to create [API Gateway Deployment](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-deployment.html) (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigateway-deployment.html>) to publish our API. Here it is:

```
RestApiId:          Ref: WildRydesApiGatewayRestApi  
StageName: ${opt:stage, 'dev'}
```

The screenshot shows a terminal window with the following command history:

```
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
Serverless: Stack update finished.
  Service Information
    Service: wild-rides-serverless-demo
      stage: dev
      region: us-east-1
      api: wild-rides-serverless-demo-dev
      api keys:
      endpoints:
        None
      functions:
        Request Unicorn: wild-rides-serverless-demo-dev-RequestUnicorn
  Serverless: Removing old service artifacts from S3...
  [~/Documents/projects/blog/_posts/wild-rides-serverless-demo]$
```

(<https://i2.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-API-Gateway-Deployment.png?ssl=1>)

Here we're using `${opt:stage, 'dev'}` to dynamically specify stage name. See [Variables](https://serverless.com/framework/docs/providers/aws/guide/variables/) (<https://serverless.com/framework/docs/providers/aws/guide/variables/>) for more info.

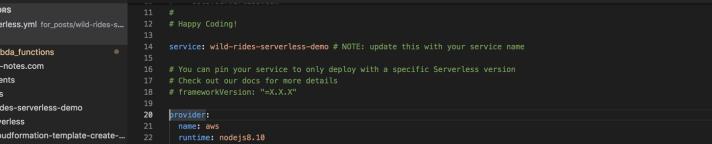
Let's redeploy our stack to check that everything's working as expected:

```
sls deploy
```

GLOBAL STAGE DECLARATION

Definitely we want to add global stage declaration, but not only for a single resource. To do so, add `stage:` parameter declared in the same way to `provider:` section:

```
provider:  
  name: aws  
  runtime: nodejs8.10  
  stage: ${opt:stage, 'dev'}
```



```
serverless.yml x

11 #
12 # Happy Coding!
13 #
14 services: wild-rides-serverless-demo # NOTE: update this with your service name
15 #
16 # You can pin your service to only deploy with a specific Serverless version
17 # Check out our docs for more details
18 # frameworkVersion: "X.X.X"
19 #
20 providers:
21   name: aws
22   runtime: nodejs8.10
23   stage: ${opt:stage, 'dev'}
24   #
25   # you can overwrite defaults here
26   # stage: dev
27   # region: us-east-1
28   #
29   # you can add statements to the Lambda function's IAM Role here
30   iamRoleStatements:
31     Effect: "Allow"
32     Actions:
33       - "s3:ListBucket"
34     Resource: ["Fn::Join": "", ["arn:aws:s3:::", {"Ref": "ServerlessDeploymentBucket"}] ]
35     Effect: "Allow"
36     Actions:
37       - "s3:PutObject"
38     Resource:
39       Fn::Join:
40         - ""
```

(<https://i0.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Global-Stage-Declaration.png?ssl=1>)

UPDATE THE WEBSITE CONFIG

Most of the work done. At this section we'll complete our website configuration (`config.js` file which is still in our project folder). But first of all let's get API Gateway Deployment URL. Specify this declaration of your outputs: of resources: section:

```
WildRydesApiGatewayUrl:
  Description: "Wild Rydes Api Gateway URL"
  Value:
    "Fn::Join":
      - ""
      - "https://"
      - Ref: "WildRydesApiGatewayRestApi"
      - ".execute-api."
      - Ref: "AWS::Region"
      - ".amazonaws.com"
      - "/${opt:stage, 'dev'}"
```

Now to get the url you can do:

```
sls deploy  
sls info --verbose
```

Next, we need to copy `WildRydesApiGatewayUrl` value to `invokeUrl:` of `config.js` file:

The screenshot shows the VS Code interface with the following details:

- EXPLORER**: Shows files like `serverless.yml`, `config.js`, and `handler.js`.
- config.js** is the active file, containing the following code:

```
1 window_config = {
2   cognito: {
3     userPoolId: 'us-east-1_2zrypyffn', // e.g. us-east-2_ixbdGspAb
4     userPoolClientID: '540rtjg7wvkbpbgnvp7mqbb', // e.g. 25d0knj4v6hfsfvruhpf17n4hv
5     region: 'us-east-1' // e.g. us-east-2
6   },
7   api: {
8     invokeUrl: 'https://6h33lsl034.execute-api.us-east-1.amazonaws.com/dev' // e.g. https://rcnyt4tql.execute-api.us-wes
9   }
10 };
11
```

- PROBLEMS**: Shows no errors.
- OUTPUT**: Shows deployment logs:

- app keys:
 - None
- endpoints:
 - None
- functions:
 - RequestUnicorn: wild-rides-serverless-demo-dev-RequestUnicorn
- Stack Outputs:
 - WildRidesDynamoDBARN: arn:aws:dynamodb:us-east-1:81289762433:t0Le/rides
 - WildRidesLambdaFunctionARN: arn:aws:lambda:us-east-1:81289762433:function:wild-rides-serverless-demo-dev-RequestUnicorn:3
 - WildRidesBucketURL: http://wildrides-andre-maksimov.s3-website-us-east-1.amazonaws.com
 - WildRidesCognitoUserPoolClientID: 540rtjg7wvkbpbgnvp7mqbb
 - WildRidesCognitoUserPoolID: us-east-1_2zrypyffn
 - ServerlessDeploymentBucketName: wild-rides-serverless-de-serverlessdeploymentbuck-ayud2B5dtv5
 - WildRidesCognitoUserPoolID: us-east-1_2zrypyffn

- TERMINAL**: Shows the command `[anakinm@anakinm ~]$ cd /Documents/projects/blog/for_posts/wild-rides-serverless-demo $`.

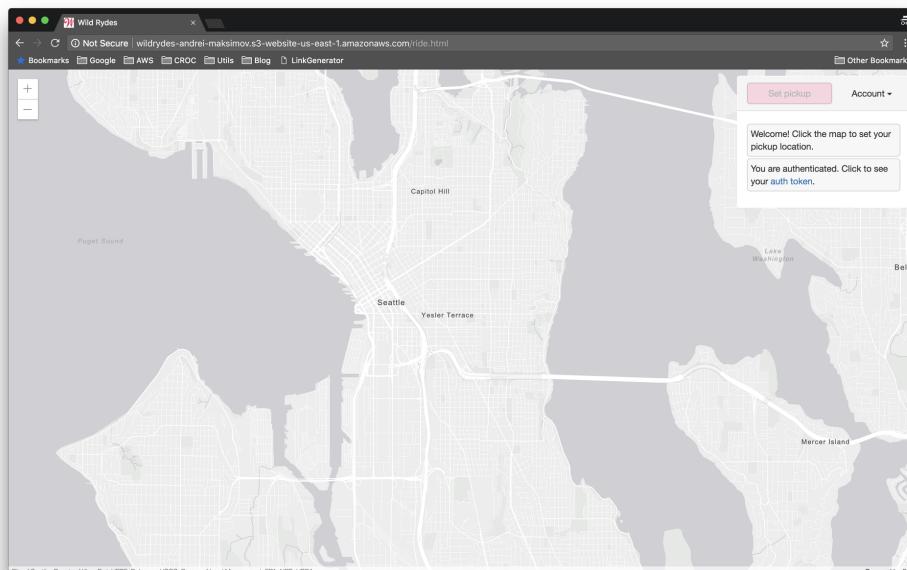
(<https://i2.wp.com/dev-ops-notes.com/wp-content/uploads/sites/2/2018/09/Serverless-Framework-Final-Website-Configuration.png?ssl=1>)

Upload config.js file to its location in S3 bucket and remove the file:

```
aws s3 cp ./config.js s3://wildrydes-firstname-lastname/js/config.js  
rm ./config.js
```

VALIDATE YOUR IMPLEMENTATION

All we need to do here, is to visit [/ride.html](#) and click anywhere on the map to set a pickup Unicorn location.



Choose **Request Unicorn**. You should see a notification in the right sidebar that a unicorn is on its way and then see a unicorn icon fly to your pickup location.

RESOURCE CLEANUP

To cleanup everything you need to call

```
aws s3 rm s3://wildrydes-firstname-lastname --recursive  
sls remove
```

Congratulations! Hope, you've find this tutorial useful. Please, feel free to ask questions in the comments section! Good luck!



(



Start the discussion...

LOG IN WITH



OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

[Subscribe](#) [Add Disqus to your site](#) [Add Disqus](#) [Disqus' Privacy Policy](#) [Privacy Policy](#) [Privacy](#)

(https://www.addtoany.com/add_to/facebook?linkurl=https%3A%2F%2Fdev-ops-notes.com%2Faws%2Fserverless-framework-building-web-app-using-aws-lambda-amazon-api-gateway-s3-dynamodb-and-cognito%2F%3Futm_campaign%3DServerless%252BDigest%26utm_medium%3Dweb%26utm_source%3DServerless_Digest_50&linkname=Serverless%20framework%20-%20Building%20Web%20App%20using%20AWS%20Lambda%2C%20Amazon%20API%20Gateway%2C%20S3%2C%20DynamoDB%20and%20Cognito%20-%20Part%201%20-%20Dev-Ops-Notes.com) (https://www.addtoany.com/add_to/twitter?linkurl=https%3A%2F%2Fdev-ops-notes.com%2Faws%2Fserverless-framework-building-web-app-using-aws-lambda-amazon-api-gateway-s3-dynamodb-and-cognito%2F%3Futm_campaign%3DServerless%252BDigest%26utm_medium%3Dweb%26utm_source%3DServerless_Digest_50&linkname=Serverless%20framework%20-%20Building%20Web%20App%20using%20AWS%20Lambda%2C%20Amazon%20API%20Gateway%2C%20S3%2C%20DynamoDB%20and%20Cognito%20-%20Part%201%20-%20Dev-Ops-Notes.com) (https://www.addtoany.com/add_to/linkedin?linkurl=https%3A%2F%2Fdev-ops-notes.com%2Faws%2Fserverless-framework-building-web-app-using-aws-lambda-amazon-api-gateway-s3-dynamodb-and-cognito%2F%3Futm_campaign%3DServerless%252BDigest%26utm_medium%3Dweb%26utm_source%3DServerless_Digest_50&linkname=Serverless%20framework%20-%20Building%20Web%20App%20using%20AWS%20Lambda%2C%20Amazon%20API%20Gateway%2C%20S3%2C%20DynamoDB%20and%20Cognito%20-%20Part%201%20-%20Dev-Ops-Notes.com) (https://www.addtoany.com/add_to/google_plus?linkurl=https%3A%2F%2Fdev-ops-notes.com%2Faws%2Fserverless-framework-building-web-app-using-aws-lambda-amazon-api-gateway-s3-dynamodb-and-cognito%2F%3Futm_campaign%3DServerless%252BDigest%26utm_medium%3Dweb%26utm_source%3DServerless_Digest_50&linkname=Serverless%20framework%20-%20Building%20Web%20App%20using%20AWS%20Lambda%2C%20Amazon%20API%20Gateway%2C%20S3%2C%20DynamoDB%20and%20Cognito%20-%20Part%201%20-%20Dev-Ops-Notes.com) (<https://www.addtoany.com/share>)