

1. Introduction

This repository is targeted to cover the topic- **“Serverless full stack development guidelines with AWS Amplify framework”**. It is designed to help learners understand some basic steps to work on the mentioned topic.

We will have comprehensive walk-through and demonstration of the process of building a React-front-end based application with GraphQL, AppSync API, DynamoDB etc. features based back-end generated by AWS Amplify framework from its command line tool. The purpose is to guide the developers or learners through all the processes and generate a basic running web application.

AWS serverless services will be used to design this product's back-end features and React will be used for the front-end UI component.

With this single framework (AWS Amplify), even a developer having not much knowledge in AWS resources, can leverage his existing technological know-hows and expertise and can rapidly prototype and experiment highly scalable and robust application in incredibly less time span. The developer can focus on the front-end UI and business logic of the project but no need to think about database scaling, optimization, maintenance, disaster recovery etc. and also the back-end code is mostly auto generated, so it saves time and is less prone to errors that developers make while writing codes. There is no point in spending lot of time pondering about back-end infrastructure and security aspects of the application because they have been taken care of by AWS cloud service provider.

It is assumed that the learners may have:

- Programming knowledge of any language, however, React is primarily used for front-end UI.
- Very basic knowledge of serverless environment.
- Very basic knowledge of AWS serverless infrastructures, esp. S3, Lambda functions, DynamoDB, IAM, Cognito Userpool etc. as we will not need to provision them directly from AWS console.

2. Implementation

We will be talking about architecture design of the product that will be built through this step-by-step guideline, then setting-up the required tools and creating a example project- 'a2z-online'.

1.1 Architecture Design

The application 'a2z' is targeted to cover various online businesses in the future, however for now its scope is related to online restaurant business. Where food menus are displayed online and logged in users can make online order and payments, only the admin user is allowed to add or delete food menus.

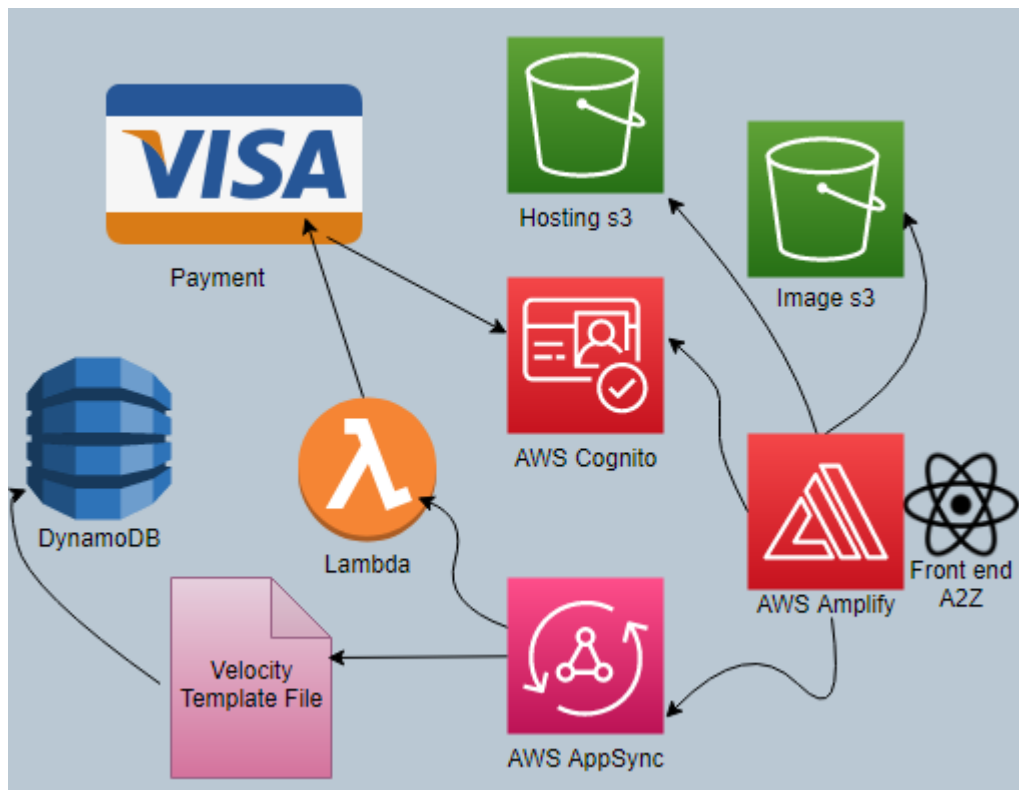


Figure 1. Architecture design for the product to be created

The front-end of this application is done in React and the serverless back-end is covered with AWS Amplify. The application is hosted in S3 bucket (Hosting s3) which is configured and enabled for static webhosting to work. Static web hosting in S3 bucket needs very simple and basic steps that almost removes the need for extra IT administration cost, too. With AWS Amplify, web hosting is done so quickly and without much efforts and with few commands from terminal, however for CI/CD pipelines, it is done from Amplify console directly.

Another S3 bucket, i.e., image S3, is used to store all images related to the application, such as food or menu or ingredients related images, and so on. Since, the images are meant to be viewed publicly, so this bucket is given public access.

In the above scenario, the front-end has AWS Amplify library installed and configured to be used from the CLI, where the front-end app sends information through the AWS Amplify library to back-end application, used with AWS AppSync GraphQL API, which does all the CRUD (create, read, update, delete) operations through it queries and mutations. GraphQL uses Velocity template language (VTL) to send queries and mutation to perform data storage, data edit in the DynamoDB.

The AppSync will invoke the serverless lambda functions which will work to detect the customer order and make payments. Once the payment is successful, it places an order and creates records in the DynamoDB. The entire process is very simple and concentrated on GraphQL AppSync API.

1.2 Tools

Node & NPM: Node.js and NPM need to be installed if they are not already installed where better compatibility for node -v should be at least 10.x and npm -v 1.x or greater.

AWS Account: If you are creating AWS Account for the first time you can get free tier option for 12 months. It is recommended that create an alias account with Administrative Access or use the account that is generated when amplify is being configured as below, but in any case, it is highly advised to turn on MFA to secure unauthorized login.

1.3 Creating Project (a2z-online)

Initialize a React project or clone the current repo: <https://github.com/khemrajneupane/a2z-online.git>.

For our purpose, it is better and easier to clone the repo and start the app.

```
git clone https://github.com/khemrajneupane/a2z-online.git
cd a2z-online
```

AWS CLI: It is installed globally with the following command:

```
npm install -g @aws-amplify/cli
```

Once the CLI is installed, let's configure Amplify:

```
amplify configure
```

```

PS C:\Users\Babita\Desktop\a2z-online> amplify configure
Follow these steps to set up access to your AWS account:

Sign in to your AWS administrator account:
https://console.aws.amazon.com/
Press Enter to continue

Specify the AWS Region
? region: eu-west-1
Specify the username of the new IAM user:
? user name: a2z-user
Complete the user creation using the AWS console
https://console.aws.amazon.com/iam/home?region=eu-west-1#/users\$policies&policies=arn:aws:iam::aws:policy%2FAdministratorAccess
Press Enter to continue

Enter the access key of the newly created user:
? accessKeyId: *****
? secretAccessKey: *****
This would update/create the AWS Profile in your local machine
? Profile Name: a2z-user

Successfully set up the new user.

```

Figure 2. Amplify Configure

The closest region is a better option to choose from the lists. The username is better to keep as meaningful or project specific as possible. In this case, the username is 'a2z-user' and the region is 'eu-west-1'.

Configuration command will ask to sign into the AWS Console

User name*

[+ Add another user](#)

Access type

Users will access AWS. Access keys and autogenerated

Access type* ☒ Programmatic access

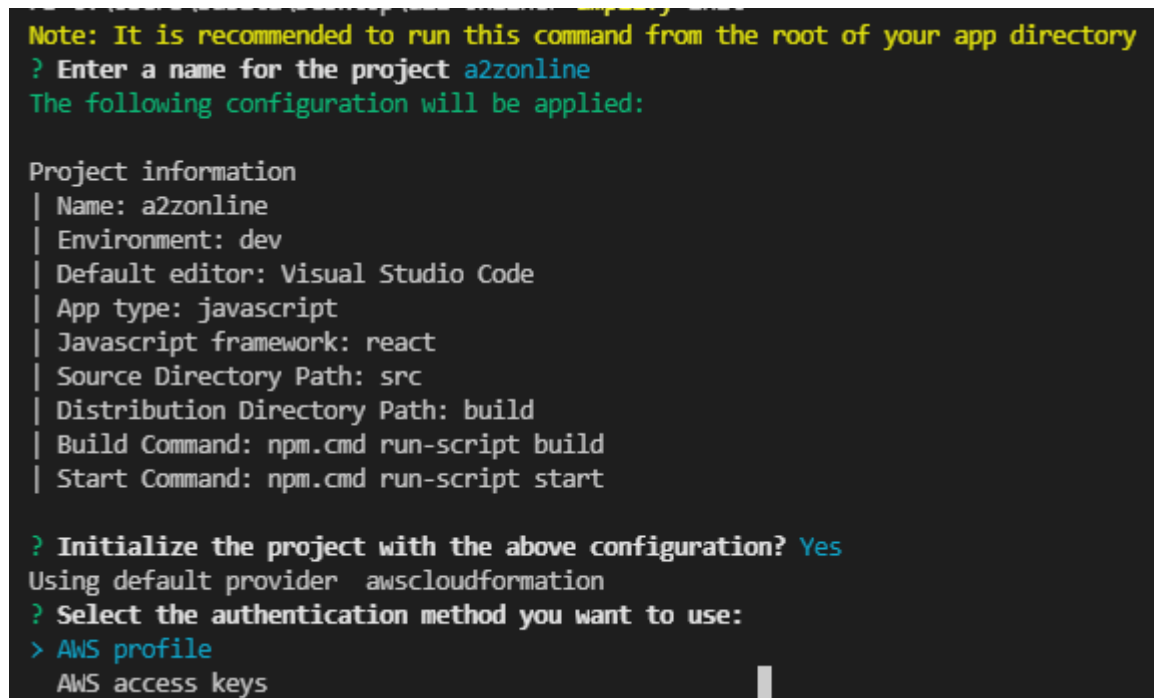
Figure 3. IAM user creation with programmatic access type.

and create an IAM user with the help of Amazon IAM service which takes care of managing users, setting permissions and so on. 'Programmatic access', is chosen so that our access key id and secret access key will be enabled and we will be able to use AWS API, CLI as required in this project.

Now we can initialize this project with AWS Amplify:

```
$ amplify init
```

This command will walk us through a number of processes to initializing amplify into our React project and generating information. Most of the processes are asking project information from us, however, if we don't provide any, Amplify is intelligent enough to prepare more suitable configuration.

A terminal window showing the AWS Amplify initialization process. The text is as follows:

```
Note: It is recommended to run this command from the root of your app directory
? Enter a name for the project a2zonline
The following configuration will be applied:

Project information
| Name: a2zonline
| Environment: dev
| Default editor: Visual Studio Code
| App type: javascript
| Javascript framework: react
| Source Directory Path: src
| Distribution Directory Path: build
| Build Command: npm.cmd run-script build
| Start Command: npm.cmd run-script start

? Initialize the project with the above configuration? Yes
Using default provider awscloudformation
? Select the authentication method you want to use:
> AWS profile
  AWS access keys
```

Figure 4. AWS Amplify initialization process

Usually, it creates a project name automatically by referring to the project directory name. Provide suitable environment, in our case, we can provide 'dev' as it is the development environment. It automatically recognizes most of the project specific information and the CLI uses this information while running commands. So, it is important to provide right information.

For authentication method, it is possible to choose AWS access keys or AWS profile, however, we have already set-up AWS-profile user in our local computer, so it is an easier and better option. If we want or need, can check all our saved aws credentials in local computer in '.aws' folder:

```
$ cd .aws
```

```
$ ls
```

```
amplify/ config credentials
```

```
$ vim config
```

Now, when we are asked to choose the profile, the CLI commands will display all the profile names that we have in our 'config' file. We choose 'a2z-user'. It starts adding back-end environment as 'dev' and start creating basic resources as below:

```
? Please choose the profile you want to use a2z-user
Adding backend environment dev to AWS Amplify Console app: d16eqzqntgetu5
- Initializing project in the cloud...

CREATE_IN_PROGRESS amplify-a2zonline-dev-160940 AWS::CloudFormation::Stack
03:00) User Initiated
CREATE_IN_PROGRESS DeploymentBucket AWS::S3::Bucket
03:00)
CREATE_IN_PROGRESS UnauthRole AWS::IAM::Role
03:00)
CREATE_IN_PROGRESS AuthRole AWS::IAM::Role
03:00)
CREATE_IN_PROGRESS AuthRole AWS::IAM::Role
03:00) Resource creation Initiated
CREATE_IN_PROGRESS UnauthRole AWS::IAM::Role
03:00) Resource creation Initiated
```

Figure 5. Amplify CLI command triggering CloudFormation to create backend environment

Amplify will create many resources in the cloud under the hood by using the CloudFormation stack of the AWS Cloud. In the above process, Amplify will create IAM (Identity and Access Management) role for unauthenticated and authenticated users

<input type="checkbox"/>	Role name	Trusted entities
<input type="checkbox"/>	amplify-a2zonline-dev-163413-authRole	None
<input type="checkbox"/>	amplify-a2zonline-dev-163413-unauthRole	None

Figure 6. Amplify creating IAM roles for authenticated and unauthenticated users

along with S3 deployment bucket these resources can be checked in the CloudFormation stacks:

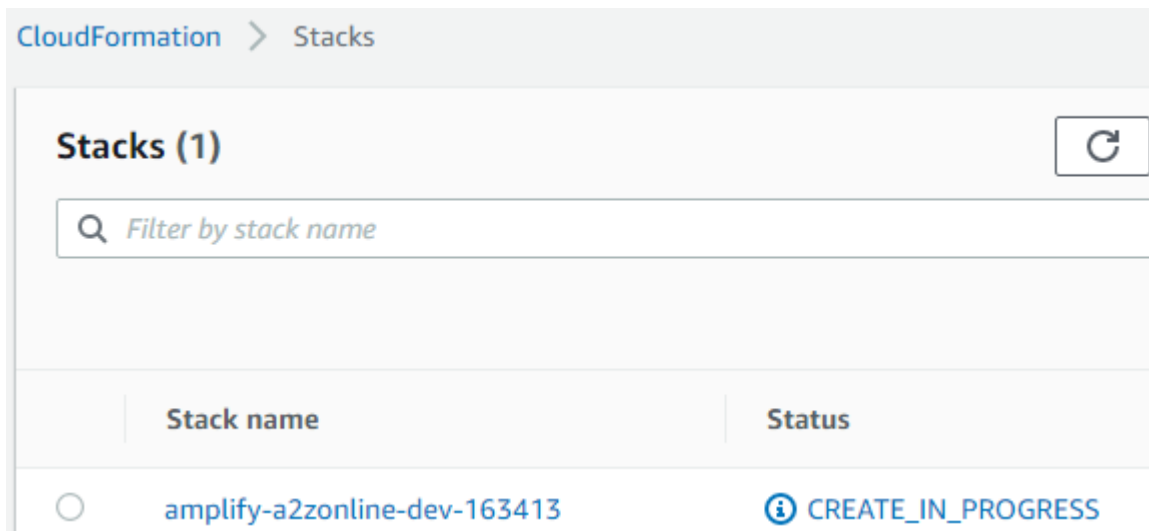


Figure 7. CloudFormation showing resources creation in progress

After the stack formation completes the status changes as in the figure below:

Stack name	Status
amplify-a2zonline-dev-163413	CREATE_COMPLETE

Figure 8. CloudFormation showing resources creation completion

The S3 bucket will also contain new data after the Amplify initialization completes:

Buckets (1) Info			Copy ARN	En
Buckets are containers for data stored in S3. Learn more				
<input type="text" value="Find buckets by name"/>				
Name	AWS Region	Access		
amplify-a2zonline-dev-163413-deployment	EU (Ireland) eu-west-1	Objects can be public		

Figure 9. Amplify initialization creates data in S3 bucket

We did not have to create all those resources but Amplify does that for us through our CLI commands which means a lot of time and burden of creating resources have been minimized. It would also be possible that if we had to create them on our own, we could make redundant resources, fail to provide suitable permissions and so on.

By now, if we check our 'a2z-online' project directory, we will notice that two files have been added: one in the root level, which is 'amplify' and the other is 'aws-exports.js' which is inside the 'src' file.

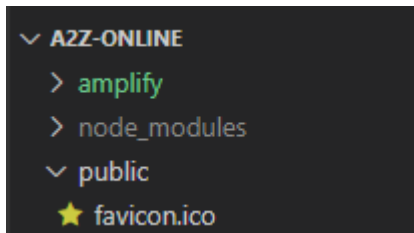


Figure 10. Amplify initialization also creates a folder containing different backend for different resources, in the project's directory 'A2Z-ONLINE'

amplify folder

It contains all the codes related to back-end and various configurations. The config file contains local AWS and environment configuration related json key-value pairs, along with project configuration information which we had provided while initializing the project earlier on.

```
{
  "projectName": "a2zonline",
  "version": "3.1",
  "frontend": "javascript",
  "javascript": {
    "framework": "react",
    "config": {
      "SourceDir": "src",
      "DistributionDir": "build",
      "BuildCommand": "npm.cmd run-script build",
      "StartCommand": "npm.cmd run-script start"
    }
  },
  "providers": [
    "awscloudformation"
  ]
}
```

Figure 11. Automatic generation of backend config. codes after Amplify initialization

The back-end folder will contain all the GraphQL schema for AppSync API which we will create later. It will also contain all the cloud formation .yaml files later when we command CLI for adding more resources.

aws-exports.js

This file is created inside the 'src' folder. It will contain all the key-value pairs generated automatically; we don't need to modify anything there.


```
const awsmobile = {
  "aws_project_region": "eu-west-1",
  "aws_cognito_identity_pool_id": "eu-west-1",
  "aws_cognito_region": "eu-west-1",
  "aws_user_pools_id": "eu-west-1_srIaHLfNd",
  "aws_user_pools_web_client_id": "55p1fhs1f",
  "oauth": {},
  "aws_user_files_s3_bucket": "a2zonline14fe",
  "aws_user_files_s3_bucket_region": "eu-west-1",
  "aws_appsync_graphqlEndpoint": "https://l",
  "aws_appsync_region": "eu-west-1",
  "aws_appsync_authenticationType": "AMAZON_COGNITO_USER_POOLS",
  "aws_appsync_apiKey": "da2-ey3hiagszrdahf5",
};

export default awsmobile;
```

Figure 12. Amplify generated aws-exports.js file inside src directory

1.4 Adding Authentication

By now, the project has been initialized and all the necessary configurations for Amplify back-end and react front-end is completed; next up we add authentication for login, logout and signup by using Amplify's ready-made Cognito Userpool authentication flow.

```
$ amplify add auth
```

There are default and federated login options to choose for this configuration. There are federated login possibilities with Facebook, Google, Apple, Amazon, SAML, OpenID. However, for the purpose of this application, we go for default configuration with username, password and phone number.

```
C:\Users\Babita\Desktop\a2z-online>amplify add auth
Using service: Cognito, provided by: awscloudformation

  The current configured provider is Amazon Cognito.

Do you want to use the default authentication and security
configuration? Default configuration
Warning: you will not be able to edit these selections.
How do you want users to be able to sign in? Username
Do you want to configure advanced settings? No, I am done.
Successfully added auth resource a2zonline31270649 locally
```

Figure 13. Amplify CLI adding authentication

Now, to see that our authentication works, we need to push this configuration which will create the necessary back-end in our local and provision it in the cloud.

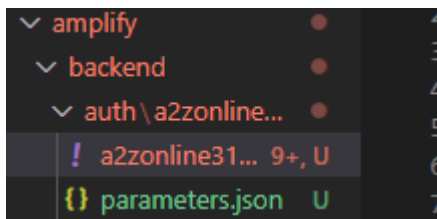


Figure 14. Amplify CLI creating backend code in the project folder, necessary for authentication

```
$ amplify push
```

This process will create many resources in the cloud. It will automatically create SNSRoles, Cognito UserPool and lambda functions which we don't require to do manually.

- CREATE_COMPLETE SNSRole AWS::IAM::Role Sun Oct 24 2021 15:42:3\ Updating resources in the cloud. This may take a few minutes...
- CREATE_IN_PROGRESS UserPool AWS::Cognito::UserPool Sun Oct 24 / Updating resources in the cloud. This may take a few minutes...
- CREATE_IN_PROGRESS UserPoolClientLambda AWS::Lambda::Function...

Then we make few imports and code changes in the front-end code in order to make the authentication form appear in the UI from where we can perform login, logout, signup operations.

```
import Amplify from 'aws-amplify';
import config from './aws-exports';
Amplify.configure(config);

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Figure 15. React front-end importing from 'aws-export' file to configure and use different AWS resources

After importing config from 'aws-exports' file which was created when amplify was initialized, now we can add authentication to any component in our app. For now, let's add it to one of the components as:

```

<AmplifyAuthenticator>
  <section>
    <header className="form-header">
      <h3>List all dishes</h3>
    </header>

    <button onClick={() => addDishes()}>Add Dishes</button>
    <button onClick={() => addManyDish()}>Add Many Dishes</button>
  </section>
</AmplifyAuthenticator>
<AmplifySignOut />

```

Figure 16. Using AmplifyAuthenticator and AmplifySignOut components in the React project

Then let's run the app with 'npm start'; we can see the login form:

Figure 17. Ready-to-use login UI created with Amplify authenticator

Since there is not any user registered in the Cognito UserPool, so we can create new account from the link and verify it with phone number or email then it will be registered like so:

Username	Enabled	Account status	Email	Email verified	Phone number verified	Updated
tester	Enabled	CONFIRMED	shekharneupane@gmail.com	true	false	Nov 7, 2021 1:50:56 PM

Figure 18. AWS Cognito UserPool registering new user

1.5 Creating S3 bucket

We need to store all our project and dish images or any other images.

`$ amplify add storage`

With this command, we will create s3 bucket to save our images. This bucket should be accessible for both guest and authenticated users. Even the users who are not logged in should be able to view the images or menus in the application. This bucket should store content types as images, audios, videos etc. For the purpose of this project, we have given this bucket a friendly name as 'DishImages'. Since, every bucket should have a unique name, so we accept the default name.

The authenticated user should be able to perform CRUD operations, but the guest users can only read. In order to choose between options for the authenticated user (a member of admin group in the Cognito user pool), we can press the space tab that will put asterisk for the options.

? Please select from one of the below mentioned services: Content (Images, audio, video, etc.)

? Please provide a friendly name for your resource that will be used to label this category in the project: s33bb10173

? Please provide bucket name: (a2zonline040ed514085a48aeaf9f4644b734a20d)

C:\Users\Babita\Desktop\A2z-online>amplify add storage

? Please select from one of the below mentioned services: Content (Images, audio, video, etc.)

? Please provide a friendly name for your resource that will be used to label this category in the project: DishImages

? Please provide bucket name: a2zonline14fee81cb4804213bd3e9f0674947f5d

? Who should have access: Auth and guest users

? What kind of access do you want for Authenticated users? create/update, read, delete

? What kind of access do you want for Guest users? read

? Do you want to add a Lambda Trigger for your S3 Bucket? No

1.6 Adding lambda functions

In our application, the lambda functions are required when dish order and dish payments are made. The user will press order button then two things will happen- the product order is registered, and payment is made, so we will create 'paymentLambda' and 'orderLambda' functions that work as pipeline resolvers in AppSync.

`$ amplify add function`

With the above command, we first create:

- 'paymentLambda' function, as serverless function capability
- NodeJS as runtime
- Default 'Hello World' template
- Choose more configuration and lambda function editing as 'no' because we want to do so later.

We will perform above process the same way for 'orderLambda' function, too. The above process will automatically create lambda functions inside <project-dir>/amplify/backend/function/paymentLambda/src.

```
exports.handler = async (event) => {
  // TODO implement
  const response = {
    statusCode: 200,
    // Uncomment below to enable CORS requests
    // headers: {
    //   "Access-Control-Allow-Origin": "*",
    //   "Access-Control-Allow-Headers": "*"
    // },
    body: JSON.stringify('Hello from Lambda!'),
  };
  return response;
};
```

Figure 19. Amplify add function CLI command creating basic Lambda function

This lambda handler simply returns HTTP status code 200 and string 'Hello from Lambda!'. Good thing about this amplify is also that we can mock or test this function locally without creating resources in the cloud as:

```
$ amplify mock function paymentLambda
```

```
C:\Users\Babita\Desktop\a2z-online>amplify mock function paymentLambda
? Provide the path to the event JSON object relative to C:\Users\Babita
end\function\paymentLambda src/event.json
Ensuring latest function changes are built...
Starting execution...
Result:
{
  "statusCode": 200,
  "body": {
    "key1": "hello world",
    "key2": "value2",
    "key3": "value3"
  }
}
Finished execution.
```

Figure 20. Amplify mocking Lambda function

or \$ amplify invoke paymentLambda however, this version of amplify command is deprecated but still outputs the results.

1.7 GraphQL API

We are going to create GraphQL API so that the front end will use this API to request data from cloud resources, through AppSync. With GraphQL, we will device required queries to handle exact data with AppSync in online and offline scenarios.

```
$ amplify add api
```

```
? Please select from one of the below mentioned services: GraphQL
? Provide API name: a2zonline
? Choose the default authorization type for the API Amazon Cognito
User Pool
Use a Cognito user pool configured as a part of this project.
? Do you want to configure advanced settings for the GraphQL API No
, I am done.
? Do you have an annotated GraphQL schema? No
? Choose a schema template: Single object with fields (e.g., "Todo"
with ID, name, description)
```

Figure 21. Amplify adding GraphQL API

With the above configuration, we have created GraphQL API named 'a2zonline' with simple schema template for 'single object' which will be used to create our own schema for this project. Here, we have chosen default authorization type to be Amazon Cognito User Pool, that will help to identify our logged in user, however, we will require API keys based usages because not logged in users should also be able to use our application in limited manners. In the above configuration, we missed to add API keys as authorization type so, we are going to update this API:

```
$ amplify update api
```

With this command, we have options to either 'Walkthrough all configurations' or reset authorization types only. In this case, we have chosen to walkthrough all configs, as follow:

```
? Please select from one of the below mentioned services: GraphQL
? Select from the options below Walkthrough all configurations
? Choose the default authorization type for the API Amazon Cognito User Pool
Use a Cognito user pool configured as a part of this project.
? Do you want to configure advanced settings for the GraphQL API Yes, I want to m
ake some additional changes.
? Configure additional auth types? Yes
? Choose the additional authorization types you want to configure for the API API
key
API key configuration
? Enter a description for the API key: GuestUserAccessToMenus
? After how many days from now the API key should expire (1-365): 365
? Enable conflict detection? No
```

Figure 22. Amplify CLI command updating API configuration

Finally, if we check into amplify/backend folder, we will see api/a2zonline with the following basic schema template:

```

amplify > backend > api > a2zonline > schema.graphql
1  type Todo @model {
2    id: ID!
3    name: String!
4    description: String
5  }

```

Figure 23. Basic GraphQL schema

1.8 Mock API

As a good practice, it is better to run the mock test of the GraphQL API before we actually deploy or push to cloud for resources creation. So, we can save our resources without having to run them into our billable account. Running the mock version will allow us to test the API locally by running local graphical editor on a certain port. In order to perform the CRUD operations, it will use SQLite in-memory-database.

```
$ amplify mock api
```

```

? Choose the code generation language target javascript
? Enter the file name pattern of graphql queries, mutations and subscriptions src\graphql
  \**\*.js
? Do you want to generate/update all possible GraphQL operations - queries, mutations and
  subscriptions Yes
? Enter maximum statement depth [increase from default if your schema is deeply nested] 2

✓ Generated GraphQL operations successfully and saved at src\graphql
AppSync Mock endpoint is running at http://172.28.80.1:20002

```

Figure 24. Configuring for Amplify GraphQL API mocking

We have configured the mock environment as above and chosen language, queries, mutation and subscription file location with simple nesting. The GraphQL editor will be running in <http://localhost:20002/>, where we can perform all sorts of GraphQL operations for testing purposes.

```

mutation createToDo {
  createToDo(
    input: {
      description: "Hello from description",
      name: "Hello from name"
    }
  ) {
    id
    description
    name
  }
}

```

```

{
  "data": {
    "createToDo": {
      "id": "9fc98b5c-a200-4a7a-b5dd-6b56c4bd8e0d",
      "description": "Hello from description",
      "name": "Hello from name"
    }
  }
}

```

Figure 25. GraphQL mocking in <http://localhost:2000>

In order to run this mock environment, the above command will create local resources in our project: inside src/graphql, it creates all the required mutations, queries and subscription files along with adding GraphQL endpoint in the aws-exports.js file.

1.9 GraphQL Schema

We create GraphQL Schema to define our application's data model and enhance it by adding GraphQL directives to perform more actions. Out of 9 different kinds of directives, we will be using only the following which are defined in the Amplify documentation as following (Amplify Docs, Directives, 2021):

@model : Defines top level object types in your API that are backed by Amazon DynamoDB

@auth: Defines authorization rules for your @model types and fields

@connection: Defines 1:1, 1:M and N:M relationships between @model types

@function: Configures a Lambda function resolver for a field

@key Configures custom index structures for @model types.

We will create 4 types of schemas as 'Dish', 'DishOrder', 'Order' and custom mutation type 'processOrder':

1.9.1 Type Dish

```
type Dish @model(subscriptions:null)
# @searchable
@auth(rules: [
  { allow: groups, groups: ["Admin"]}
  { allow: private, operations: [read] }
  { allow: public, operations: [read] }]) {
  id: ID!
  name: String!
  image: String
  price: Float
  special_today: Boolean
  description: String
  orders: [DishOrder] @connection(keyName: "byDish", fields: ["id"])
```

Figure 26. The GraphQL schema 'Dish' type

In the above schema, the @auth directive contains many rules handled by Cognito User Pool:

- The admin or the users who belong to the 'Admin' group, has the full authority to perform CRUD operations on the Dish type.
- All the authenticated users have read rights on the Dish type and
- All the public users (not logged in) can have read access.

Likewise, @model directive is used to create tables for Dish type in the DynamoDB and create Velocity Template File that handles CRUD operations. Again, we are not allowing real-time updates yet, so subscription is set to null.

1.9.2 Type Order

This is the second main schema after Dish. It contains user info., ordered dish, date, price and id.

The auth rules implied:

- The admin or the users who belongs to the 'Admin' group has every rights
- The logged in user (identified user) can read his own orders only. The users logged in are identified with their registered emails in this case.

```
type Order @model(subscriptions: null)
  @auth(
    rules: [
      { allow: owner, identityClaim: "email", ownerField: "customer" }
      { allow: groups, groups: ["Admin"] }
    ]
  )
  @key(name: "byUser", fields: ["user"]) {
    id: ID!
    user: String!
    date: String
    total: Float
    dishes: [DishOrder] @connection(keyName: "byOrder", fields: ["id"])
  }
```

Figure 27. The GraphQL schema 'Order' type

We also add @key directive to identify the user who makes the order and also it refers to the 'user' fields to make search on this particular order.

1.9.3 Type Mutation

This is custom mutation type that is required to execute 2 lambda functions: 'paymentLambda' and 'orderLambda' and also to contain order fields. When the user selects the dishes, puts them in the basket and sets to checkout, the user's cart items, price, token from payment platform (Stripe in our case), address etc. are checked in the make order process and for successful and unsuccessful payments, it creates the status of 'DONE' or 'FAILED':

```
enum Status {
  DONE
  FAILED
}

input DishCart {
  id: ID!
  title: String
  image: String
  price: Float
  amount: Int
}

input OrderFields {
  id: ID!
  cart: [DishCart]
  total: Float!
  token: String!
  address: String
}

type Mutation {
  makeOrders(input: OrderFields!): Status
    @function(name: "paymentLambda-${env}")
    @function(name: "orderLambda-${env}")
}
```

Figure 28. GraphQL custom 'Mutation' type

'paymentLambda' function will be executed and once the payment is successful, it will invoke 'orderLambda' and keeps the records in the DynamoDB.

In other words, when paymentLambda function executes first, it does two things:

- Sends the order charging bill information to Stripe as:

```
- await stripe.charges.create({
-   amount: total,
-   currency: "eur",
-   source: token,
-   description: `Order ${new Date()} by user: ${email}`
- });
```

- And returns the following fields:

```
- return { id, cart, total, address, username, email };
```

While the orderLambda function will receive the above return values as:

```
const payload = event.prev.result;
```

From this payload, it will extract the details to register the order in OrderTable and also in DishOrder bridge table.

1.9.4 Type DishOrder

This is a N:M (many-to-many) relationship scenario where every order for dishes is presented in this third table. This type is required because a user can have many orders for many dishes.

```
type DishOrder @model(queries: null, subscriptions: null)
  @key(name: "byDish", fields: ["dish_id","order_id"])
  @key(name: "byOrder", fields: ["order_id","dish_id"])
  @auth(rules: [
    { allow: owner,identityClaim: "email",ownerField: "customer" }
    { allow: groups,groups:["Admin"]})] {
    id: ID!
    dish_id: ID!
    order_id: ID!
    dish: Dish @connection(fields: ["dish_id"])
    order: Order @connection(fields: ["order_id"])
  }
```

Figure 29. GraphQL custom DishOrder type to create bridge table

When an order is made, we need this table to keep the record of the orders and dish by their ids. This is a bridge table which is referred in Dish type and Order type as:

```
orders: [DishOrder] @connection(keyName: "byDish", fields: ["id"])
dishes: [DishOrder] @connection(keyName: "byOrder", fields: ["id"])
```

Likewise, secondary indexes 'byDish' and 'byOrder' are used to get all dishes and orders. When we check the status as '*amplify status*' CLI command, we will get a table of all the categories, resource names, operation which helps us to understand which resources have been created, updated, deleted etc. Now that most of our resources have been created, we can finally push it to the cloud.

```
$ amplify push
```

Since we last created GraphQL API, we have not pushed the code to cloud so now, it will automatically create 'graphql' folder inside 'src' folder that contains all GraphQL mutation.js, queries.js and schema.json files along with the greatest number of other resources in the cloud. When the process finishes, we can look into the aws-exports.js file and see that it has created a lot of key-values json object some of which will be used in the project later.

3. React front-end

In this section, we will delve deep into the front-end part of the React project to understand how the CRUD operations are being called from different components.

2.1 Submitting food menu from admin page:

Admin page is designed to create dish entries in the database. This page requires login as admin because we have used AmplifyAuthenticator component imported from aws-amplify/ui-react. Any page that requires authentication can be used with this UI component. If the not logged in user tries to access that page, will be shown the built-in login form.

With the following GraphQL function we create dish object:

```
const dishDetails = ({ name: name, description: description, image: image, price: price, special_today:special_today });
await API.graphql(graphqlOperation(createDish, { input: dishDetails }));
```

AWS Amplify library can be imported to call the API to perform GraphQL operations to createDish mutation type.

Since, only the user from Admin group is authorized to perform CRUD operation on foodmenu, hence when we test submitting foodmenu as a normal logged in user, we get the following error:

```
error creating Dishes:
▼ {data: {...}, errors: Array(1)} ⓘ
  ► data: {createDish: null}
  ▼ errors: Array(1)
    ▼ 0:
      data: null
      errorInfo: null
      errorType: "Unauthorized"
      ► locations: [{...}]
      message: "Not Authorized to access createDish on type Dish"
      ► path: ['createDish']
      ► [[Prototype]]: Object
      length: 1
      ► [[Prototype]]: Array(0)
      ► [[Prototype]]: Object
```

Figure 30. Dish creation not authorized example

From amplify console, we need to add this normal user to the Admin group. In our case, we have 'testuser' as user member of Admin group.

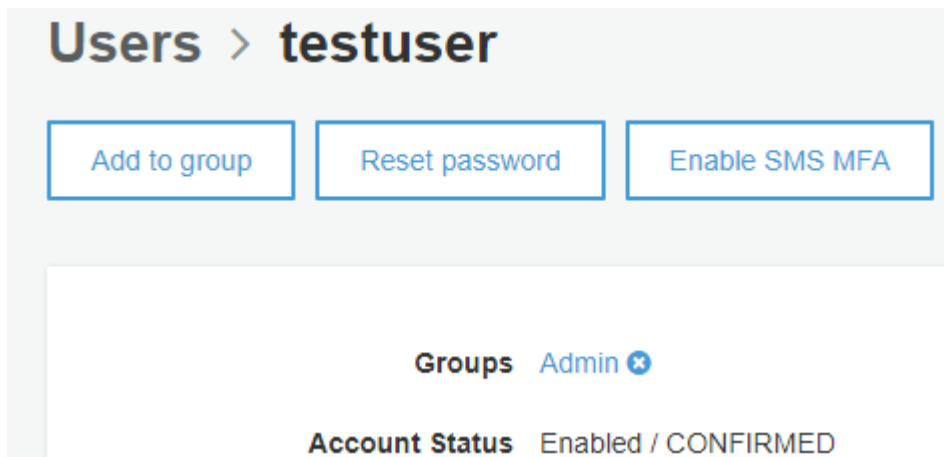


Figure 31. Adding the user to Admin group to enable dish creation

Since we have added the user to the Admin group, we can add new foodmenu item from admin page. This time, the item is saved in DynamoDB and also in S3 bucket.

id - Partition key	75e7f213-c752-4451-b408- New
special_today	<input checked="" type="radio"/> True <input type="radio"/> False
__typename	Dish
image	https://a2zonline14fee81cb4804213bd3e9f0674947f5d140530-dev.s3.eu-west-
updatedAt	2021-12-26T13:59:41.258Z
createdAt	2021-12-26T13:59:41.258Z
price	65
description	In Nepal, dhindo is a famous meal. It is prepared by gradually adding flour to
name	Nepali Kodo-Dhindo

Figure 32. Item created in DynamoDB by the user member of Admin group

In S3 bucket, the admin created item is located in:

Amazon S3 > a2zonline14fee81cb4804213bd3e9f0674947f5d140530-dev > public/ > images/

Figure 33. Item created in S3 bucket by the user member of Admin group

However, this directory has no public access:

```
▼<Error>
  <Code>AccessDenied</Code>
  <Message>Access Denied</Message>
  <RequestId>CZQKYZ85YKNEEZBX</RequestId>
  <HostId>dvNmBw4rJS19B/CxI/JB1ahdERcNfHhThcv2M8cXG5g5g5qfUTGFmJSS7vsNwOEvjN9VPVM96Xs=</HostId>
</Error>
```

So, we can edit the bucket policy and allow public access to images folder:

Policy

```
1 ▼ {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Sid": "PublicRead",
6       "Effect": "Allow",
7       "Principal": "*",
8       "Action": ["s3:GetObject", "s3:GetObjectVersion"],
9       "Resource": ["arn:aws:s3:::a2zonline14fee81cb4804213bd3e9f0674947f5d140530-dev/public/images/*"]
10    }
11  ]
12 }
```

Figure 34. Policy creating public access to the S3 bucket

After the access rights are fixed, we can easily view them in the React front-end home page:

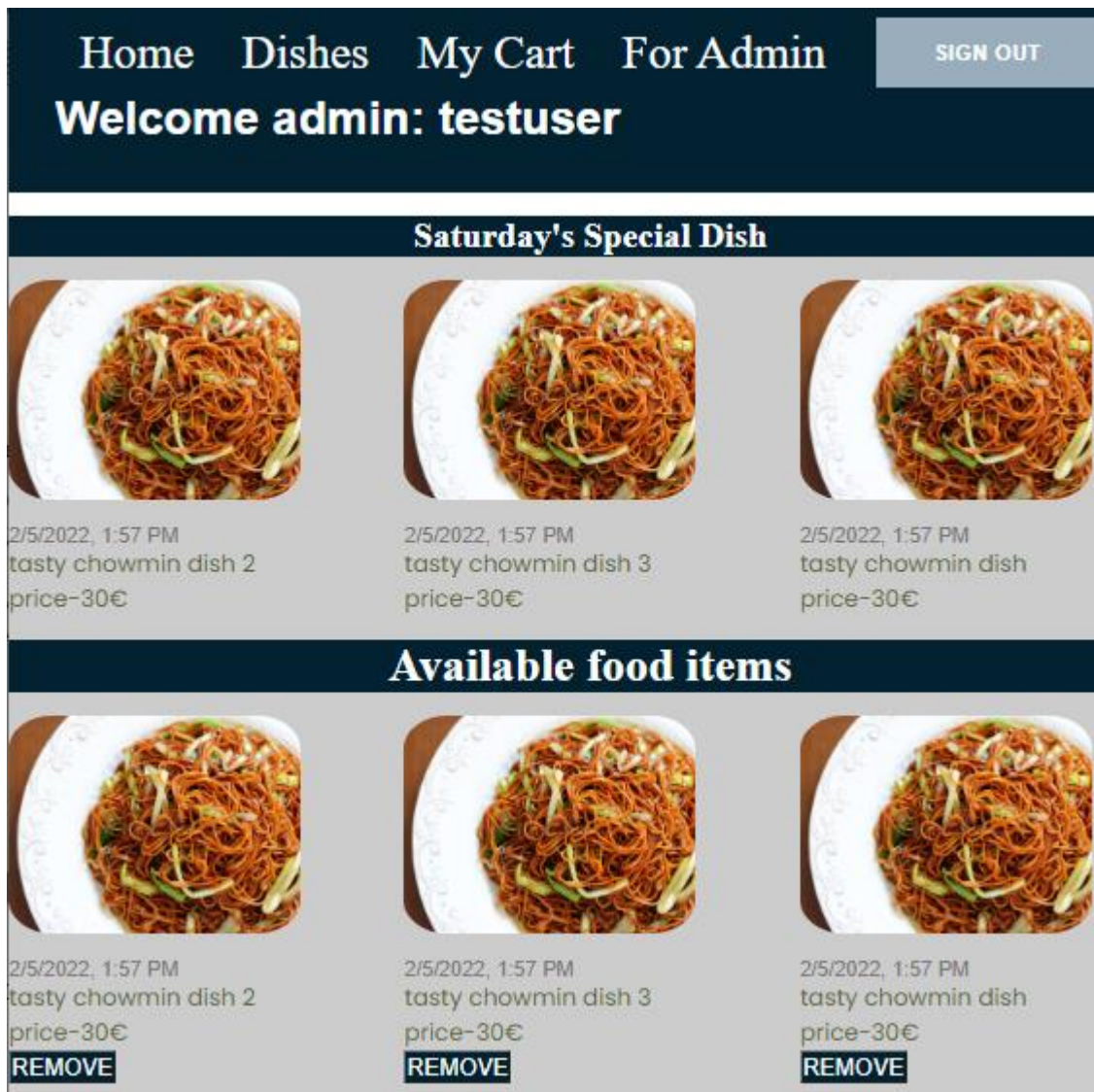


Figure 35. React front-end UI view, after the images are added in S3 and given public access policy

We can now select items and put them into cart. However, payment part is still not final. At this stage, if we try hitting payment button, we will get the following errors:

```

▼ {data: {...}, errors: Array(1)} ⓘ
  ▶ data: {ordering: null}
  ▼ errors: Array(1)
    ▼ 0:
      data: null
      errorInfo: null
      errorType: "Lambda:Unhandled"
      ▼ locations: Array(1)
        ▶ 0: {line: 2, column: 3, sourceName: null}
          length: 1
          ▶ [[Prototype]]: Array(0)
          message: "Error: Cannot find module 'stripe'\nRequire stack:\n- /var/task/index.js\n- /var/runtime/UserFunction.js\n-
          ▶ path: ['ordering']
          ▶ [[Prototype]]: Object
          length: 1

```

Figure 36. Stripe payment error example

This is because we still need to import stripe into this file:

a2z-online/amplify/backend/function/paymentLambda/src

We still see the error in console that clarifies that our function 'paymentLambda' has no permission to perform Cognito AdminGetUser. Hence, we can allow this function to access Cognito users.

From amplify console, we can go to functions and choose paymentLambda and view in lambda.

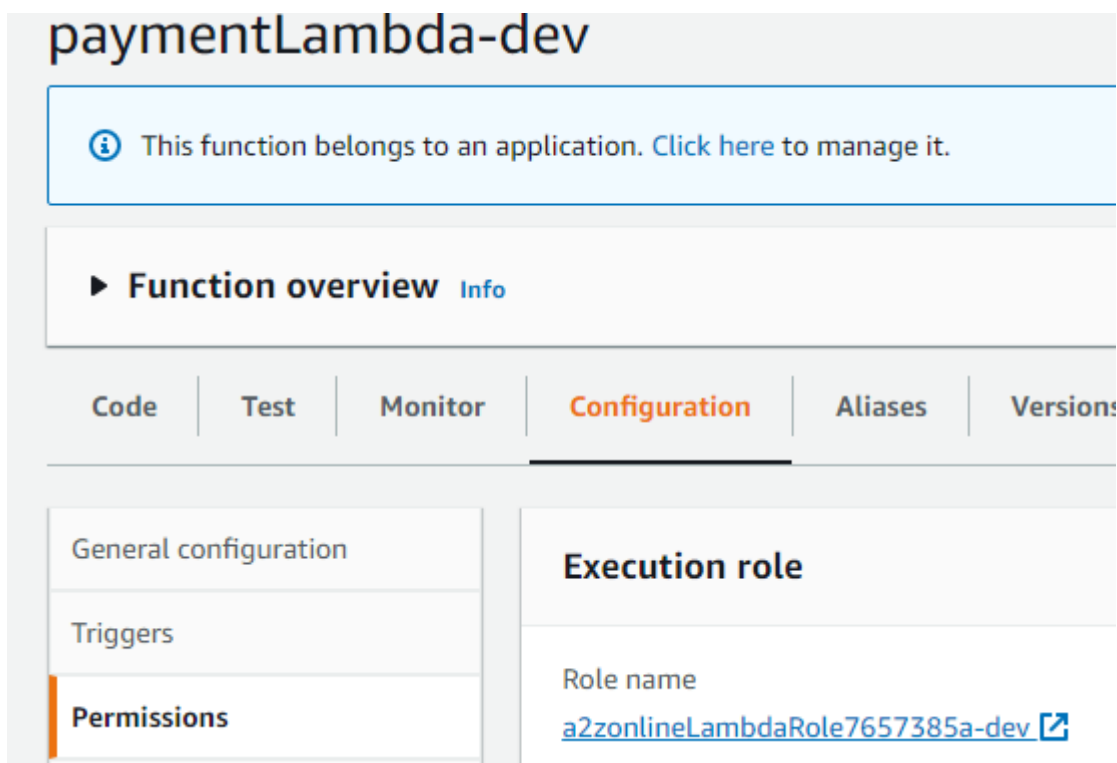


Figure 37. Configuring 'paymentLambda' function to perform Cognito AdminGetUser

From Configuration tab, choose Permission, which will show you the function execution role name. Follow this name link which will open in IAM console where we need to add inline policy for Cognito User Pool.



Figure 38. The process of adding Cognito User Pool permission to Lambda function

We need to select reading access levels for 'AdminGetUser'.

Likewise, we have 'orderLambda' function that needs to have access to the DynamoDB. Hence, we can attach one more IAM policy to this function with DynamoDB full access.

2.2 Cart

Items can be picked from home page or all dishes page and added to the cart for payment. The cart is an array that contains cart items as:

```
const cartItems = [...cart, { id, title, image, price, amount: 1 }];
```

The cart contains checkout form component which accepts the orderdetails containing token. If the token is present, then the react useEffect hook executes the checkout function.

```
useEffect(() => {
  if (orderDetails.token) {
    checkout(orderDetails);
    clearCart();
    history.push("/");
  }
}, [checkout, clearCart, history, orderDetails]);
```

2.3 Stripe Payment:

Stripe payment is very easy to integrate into React and Node project with its simple-to-use API and rich documentation. It has free test mode that is so useful to test the payment and transaction without having to make real bank transactions. We can create a free demo account and start using its public and secret keys.

The project follows stripe setup and code templates from the stripe documentation that can be followed herein:

<https://stripe.com/docs/payments/accept-a-payment-charges?platform=web#add-stripe.js-and-elements-to-your-page>.

In this project, we have used CardElement imported from react-stripe library to build the payment form with some custom styles and error messages. The Checkout component is responsible for sending the order details to the stripe dashboard. Stripe element is wrapped inside the

AmplifyAuthenticator component so that the logged in user can make this order. The Elements provider allows you to use Element components and access the Stripe object. To use Elements provider, call loadStripe from @stripe/stripe-js with your publishable key. The loadStripe function asynchronously loads the Stripe.js script and initializes a Stripe object. (Stripe docs, 2022)

```
const stripeKey= process.env.REACT_APP_STRIPE_KEY;
const stripePromise = loadStripe(stripeKey);
<AmplifyAuthenticator>
  <Elements stripe={stripePromise}>
    <section>
      <h2>Checkout Now</h2>
      <CheckoutForm />
    </section>
  </Elements>
</AmplifyAuthenticator>
```

Once the payment is successful, we can log in to the stripe dashboard and make sure it is registered there.

Payments

Filter Export + Create

All Succeeded Refunded Uncaptured

<input type="checkbox"/>	AMOUNT			DESCRIPTION	CUSTOMER	DATE
<input type="checkbox"/>	\$68.00	USD	Succeeded ✓	Order Sat Jan 29 2022 13:56:53 GMT+0000 (Coordinated Universal Time) by user: khemrjneupane@gmail.com		Jan 29, 3:56 PM
<input type="checkbox"/>	\$57.00	USD	Succeeded ✓	Order Sat Jan 29 2022 13:36:41 GMT+0000 (Coordinated Universal Time) by user: khemrjneupane@gmail.com		Jan 29, 3:36 PM
<input type="checkbox"/>	\$160.00	USD	Succeeded ✓	Order Sat Jan 29 2022 13:25:49 GMT+0000 (Coordinated Universal Time) by user: khemrjneupane@gmail.com		Jan 29, 3:25 PM
<input type="checkbox"/>	\$120.00	USD	Succeeded ✓	Order Sat Jan 29 2022 13:21:05 GMT+0000 (Coordinated Universal Time) by user: khemrjneupane@gmail.com		Jan 29, 3:21 PM
<input type="checkbox"/>	\$51.00	USD	Succeeded ✓	Order Sat Jan 29 2022 12:27:13 GMT+0000 (Coordinated Universal Time) by user:		Jan 29, 2:27 PM

Figure 39. Successful Stripe payment example from Stripe payment dashboard