Hands-on tutorials

Build an iOS Application

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Build an iOS Application: Hands-on tutorials

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Build an iOS Application i
Overview 1
What you will accomplish 1
Prerequisites 1
Modules 2
Module 1: Create an iOS app 3
Overview 1
What you will accomplish 1
Implementation 3
Conclusion 12
Module 2: Initialize Amplify 13
Overview 1
What you will accomplish 1
Key concepts 13
Implementation 3
Conclusion 12
Module 3: Add Authentication 19
Overview 1
What you will accomplish 1
Key concepts 13
Implementation 3
Module 4: Add a GraphQL API service and a database 31
Overview 1
What you will accomplish 1
Key concepts 13
Implementation 3
Conclusion 12
Module 5: Add the Ability to Store Images 44
Overview 1
What you will accomplish 1
Key concepts 13
Implementation 3
Conclusion 12
Congratulations! 59

Build an iOS Application

AWS experience	Beginner
Time to complete	60 minutes
Cost to complete	Free Tier eligible
Service used	AWS Amplify
Last updated	January 24, 2024

Overview

In this tutorial, you will create a simple iOS application using AWS Amplify, a set of tools and serverless services in the cloud. As you complete each module, you will initialize a local app using the Amplify Command Line Interface (Amplify CLI), add user authentication, add a GraphQL API and a database to store your data, and update your app to store images.

What you will accomplish

- Manage a serverless cloud backend from the command line
- Add auth to your app to enable sign-in and sign-out
- Add a GraphQL API, database, and storage solution
- Share your backend between multiple projects

Prerequisites

- An AWS account with the correct permissions (an Administrator role or root account will also work, but we recommend a least-privileges approach).
- Node.js v 14.x or more recent.
- Xcode 15.x or more recent, available on the Mac App Store.
- AWS Command Line Interface AWS CLI 2.x or more recent.

Modules

This tutorial is divided into five short modules. You must complete each module in order before moving on to the next one.

1. Module 1: Create an iOS app (10 minutes): Create an iOS app and test it in the iPhone simulator.
2. Module 2: Initialize Amplify (10 minutes): Initialize a local app using AWS Amplify.
3. Module 3: Add Authentication (10 minutes): Add auth to your application.
4. Module 4: Add a GraphQL API service and a database (20 minutes): Create a GraphQL API.
5. Module 5: Add the Ability to Store Images (10 minutes): Add storage to your app.

You will be building this iOS application using the Terminal and Apple's Xcode IDE.

Module 1: Create an iOS app

Time to complete	> 10 minutes
Key concepts	**SwiftUI** – SwiftUI is a simple way to build user interfaces across all Apple platforms with the power of the Swift programming language.
Services used	AWS Amplify

Overview

In this tutorial, you will set up your AWS account and development environment. This will allow you to interact with your AWS account and programmatically provision any resources you need.

What you will accomplish

In this module, you will:

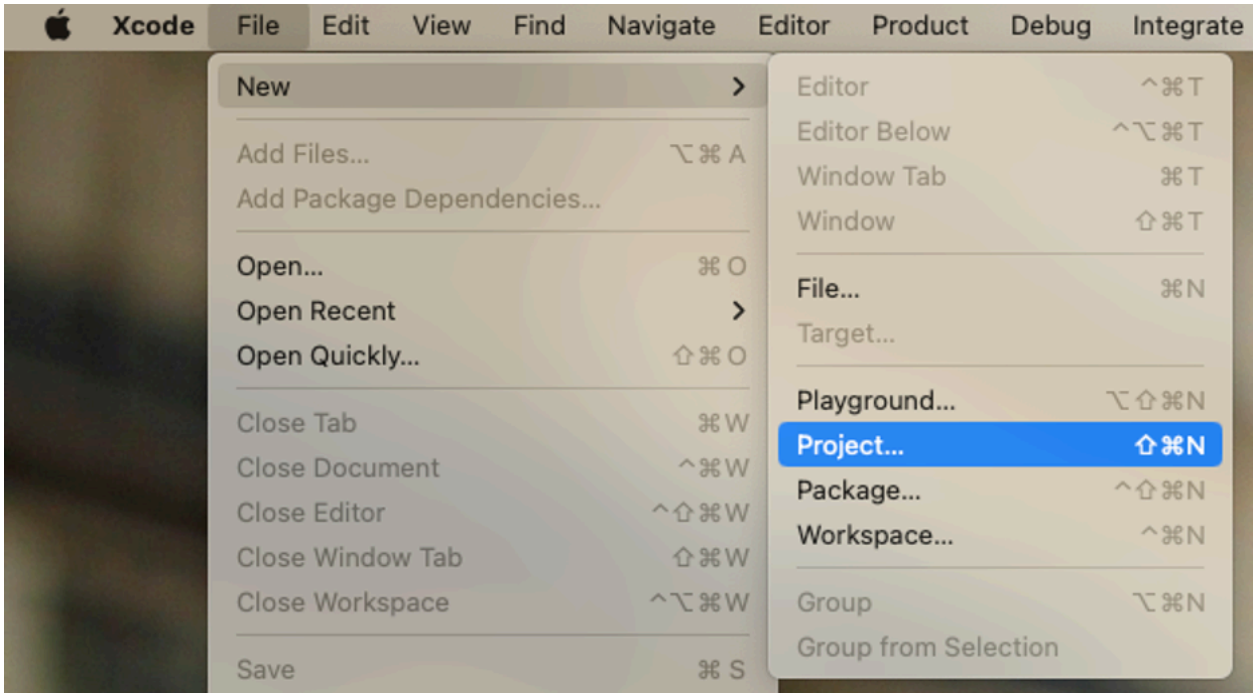
- Create an iOS application
- Update the main view
- Build and test your application

Implementation

Step 1: Create and Deploy an iOS App

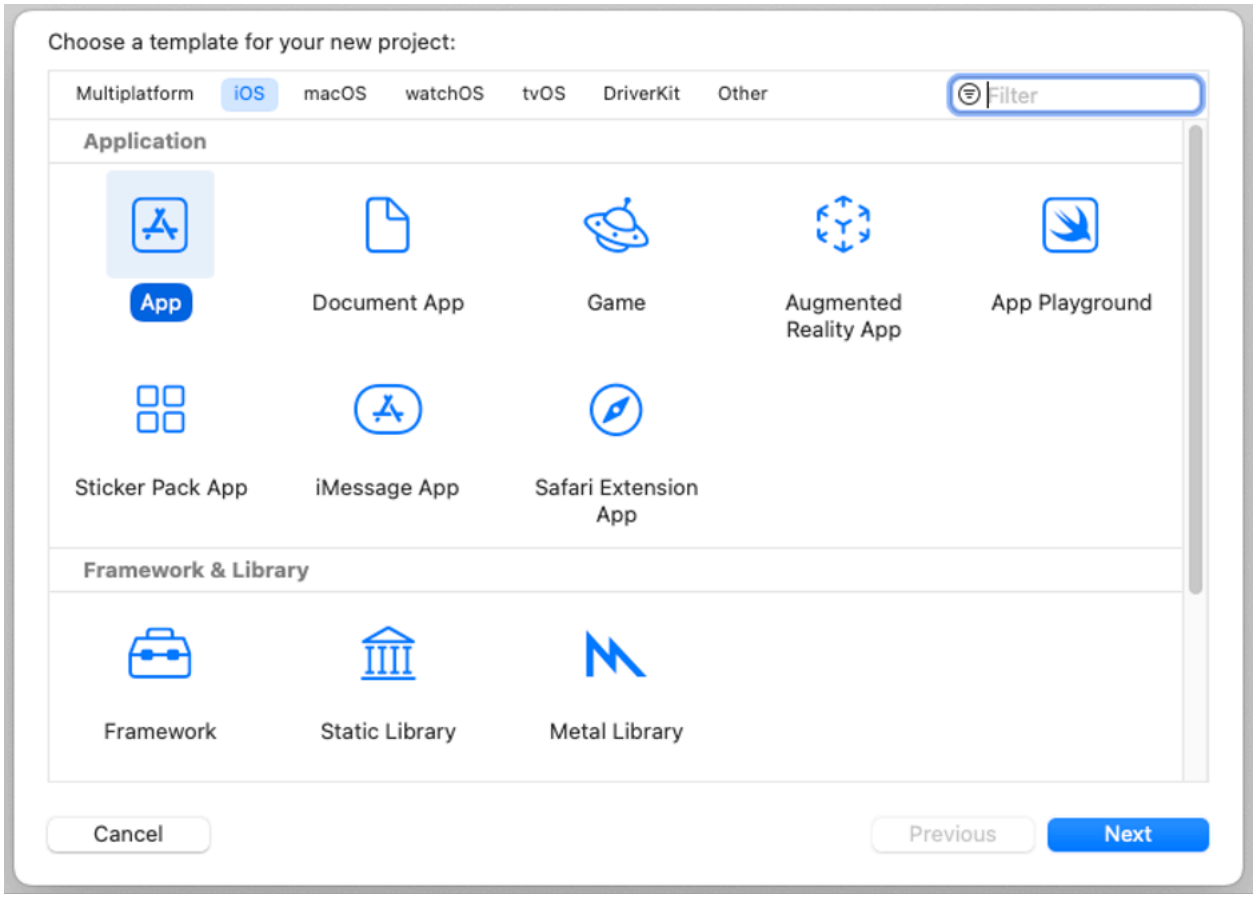
1. Start Xcode

   Start **Xcode** and create a new project by going to **File > New > Project...** or by pressing **Shift + Cmd + N**.



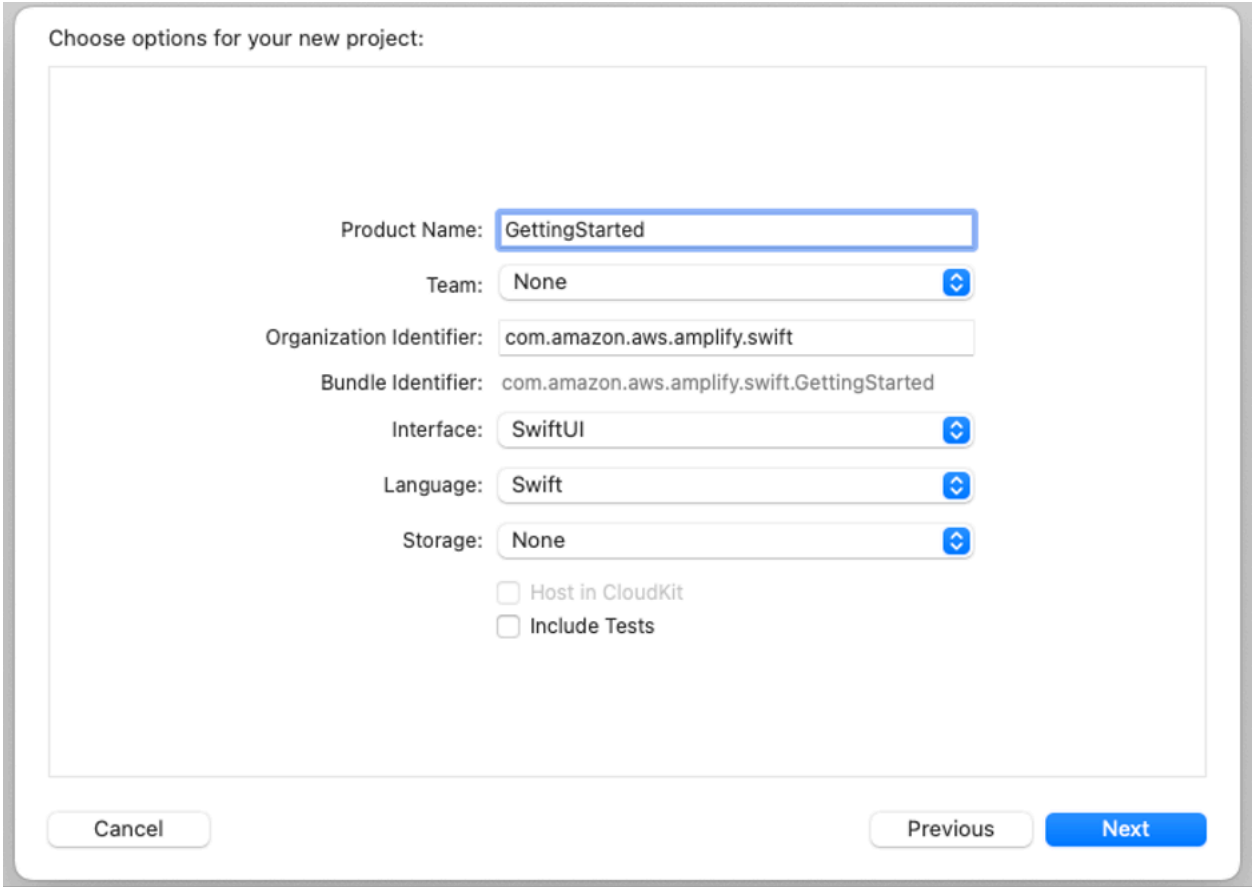
2. Choose your app template

   Choose **App** under **iOS**, **Application**, and then choose **Next**.



3. Name and configure the project

   Type a name for your project, for example, **Getting Started.** Make sure the **Interface** is **SwiftUI** and **Language** is **Swift**, then choose **Next.**



4. Finalize the project

   Select a directory and choose **Create** to create the project.

Step 2: Create the main view

1. Create Note file

   Create a new **Swift File** by clicking the plus (**+**) at the bottom of the navigation pane, or by pressing **Cmd + N**.

   Name the file **Note.swift**, and add the following content:

```
import Foundation
```

```
struct Note {
    let id: String
    let name: String
    let description: String?
    let image: String?

    init(
        id: String = UUID().uuidString,
        name: String,
        description: String? = nil,
        image: String? = nil
    ) {
        self.id = id
        self.name = name
        self.description = description
        self.image = image
    }
}
```

This struct holds information about notes, such as a name, description and image.

> **Note**
>
> Only the name is a mandatory parameter in its initializer.

2. Create view for Note objects

Next, create another file named **NoteView.swift** with the following content:

```
import SwiftUI

struct NoteView: View {
    @State var note: Note

    var body: some View {
        HStack(alignment: .center, spacing: 5.0) {
            VStack(alignment: .leading, spacing: 5.0) {
                Text(note.name)
                    .bold()
                if let description = note.description {
                    Text(description)
                }
```

```
            }

            if let image = note.image {
                Spacer()

                Divider()

                Image(systemName: image)
                    .resizable()
                    .aspectRatio(contentMode: .fill)
                    .frame(width: 30, height: 30)

            }
        }
    }
}
```

This defines a view that displays the information of a Note object, including creating an Image from its image property.

3. Create view for Notes array

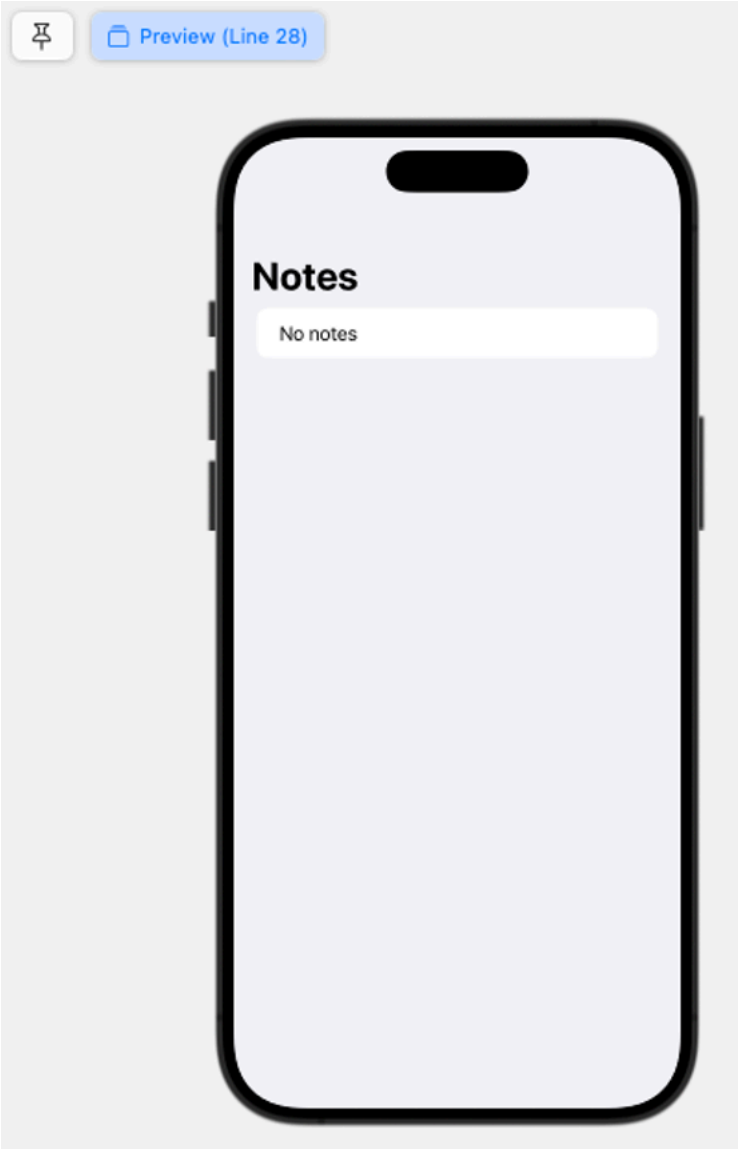
   Create a new **SwiftUI View** file named **NotesView.swift** with the following content:

```
import SwiftUI

struct NotesView: View {
    @State var notes: [Note] = []

    var body: some View {
        NavigationStack{
            List {
                if notes.isEmpty {
                    Text("No notes")
                }
                ForEach(notes, id: \.id) { note in
                    NoteView(note: note)
                }
            }
            .navigationTitle("Notes")
        }
    }
}
```

```
#Preview {
    NotesView()
}
```

This view will use the NoteView view to display all the notes in the notes array. If the array is empty, it will show a "No notes" message, as you can see in the **Canvas**.

> **Note**
>
> If you do not see the canvas, you can enable it by going to **Editor > Canvas**. If you see a **Preview paused** message, you can resume it by pressing the ↻ button next to it.

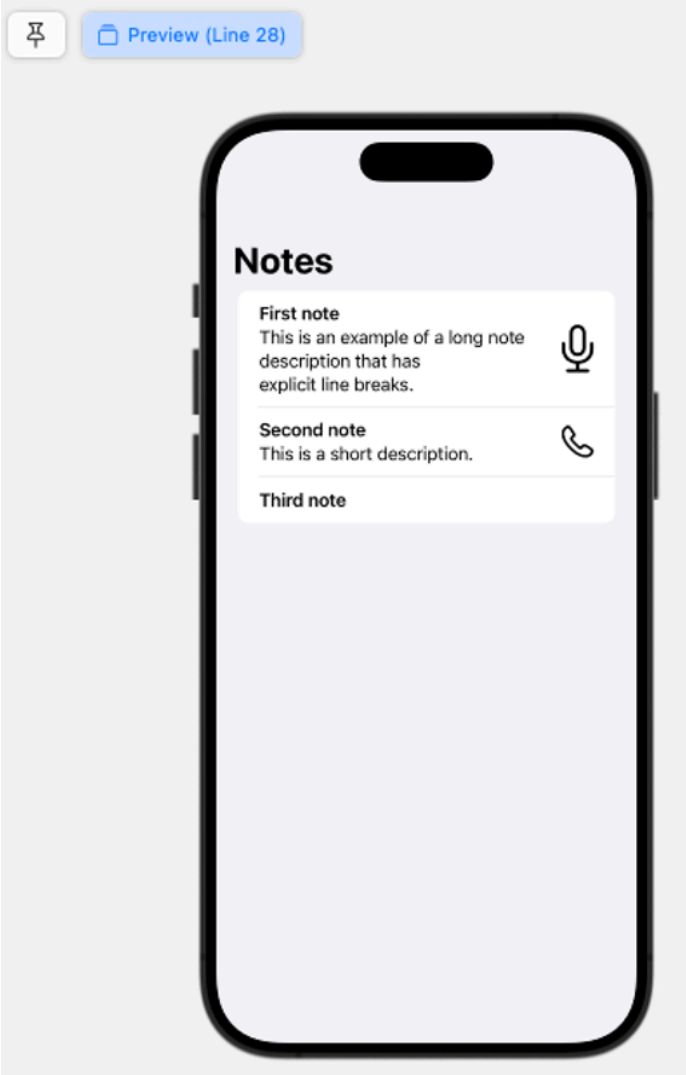


4. Set and view Notes arguments

   You can set the notes argument in the **#Preview** block to test how the view looks when the array is populated. For example:

```
#Preview {
    NotesView(notes: [
        Note(
            name: "First note",
            description: "This is an example of a long note description that has
 \nexplicit line breaks.",
            image: "mic"
```

```
        ),
        Note(
            name: "Second note",
            description: "This is a short description.",
            image: "phone"
        ),
        Note(
            name: "Third note"
        )
    ])
}
```



5. Configure the App instance

   Open the file that defines your App instance (for example, **GettingStartedApp.swift**) and replace the **ContentView() initialization** with **NotesView()**.

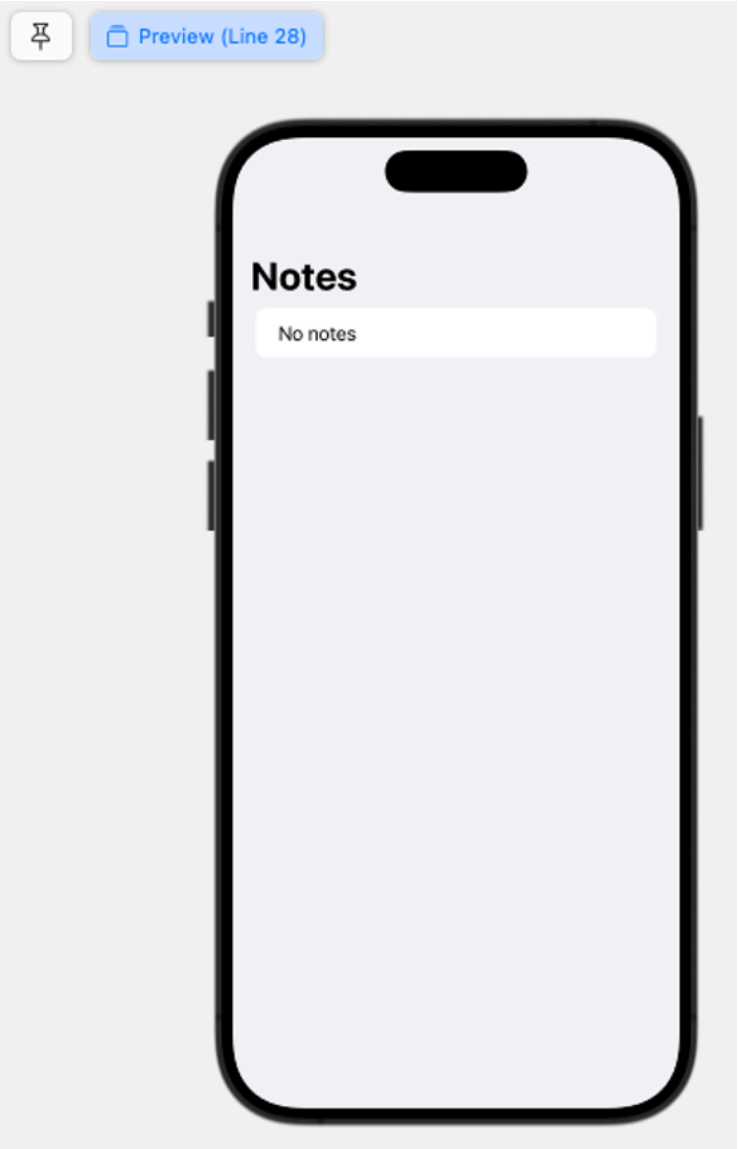
Delete the **ContentView.swift** file, we will not be using it for this tutorial.

```
import SwiftUI

@main
struct GettingStartedApp: App {
    var body: some Scene {
        WindowGroup {
            NotesView()
        }
    }
}
```

Step 3: Build and test

Build and launch the app in the simulator by pressing the ▶ button in the toolbar. Alternatively, you can also do it by going to **Product > Run**, or by pressing **Cmd + R**.

The iOS simulator will open and the app will run. As we are not setting a notes array, the default empty array is used and the "No notes" message is displayed.



Conclusion

You have successfully created an iOS app. You are ready to start building with Amplify!

Module 2: Initialize Amplify

Time to complete	10 minutes
Services used	AWS Amplify

Overview

Now that you have created an iOS application, you will want to continue development and add new features.

To start to use AWS Amplify in your application, you must install the Amplify command line, initialize the Amplify project directory, configure your project to use the Amplify libraries, and initialize Amplify libraries at runtime.

What you will accomplish

In this tutorial, you will:

- Initialize a new Amplify project
- Add Amplify libraries in your project
- Initialize Amplify libraries at runtime

Key concepts

Amplify CLI – Using the Amplify CLI you can create, manage, and remove AWS services directly from your terminal.

Amplify libraries – Using Amplify libraries you can interact with AWS services from a web or mobile application.

Implementation

Step 1: Install Amplify CLI

1. Install the CLI

To install AWS Amplify CLI, open Terminal, and enter the following command:

```
curl -sL https://aws-amplify.github.io/amplify-cli/install | bash
    && $SHELL
```

2. Configure the CLI

   Configure it to connect to your AWS account by running the following command:

   ```
   amplify configure
   ```

   Follow the steps as instructed. You can find a more detailed guide here .

Step 2: Initialize an Amplify backend

To create the basic structure of our backend, we first need to initialize the Amplify project directory and create our Cloud backend.

1. Initialize the backend

   Open the **Terminal** , navigate to the **root directory** of your project and run the following command:

   ```
   amplify init
   ```

2. Enter a name

   You will be asked to enter a name for the project. Keep the **default name** , and then **validate** that the information is correct. The following code is an example.

   ```
   ? Enter a name for the project (iOSGettingStarted): accept the default, press enter
   The following configuration will be applied:
   Project information
   | Name: iOSGettingStarted
   | Environment: dev
   | Default editor: Visual Studio Code
   | App type: ios
   ? Initialize the project with the above configuration? Yes, press enter
   Using default provider  awscloudformation, press enter
   ? Select the authentication method you want to use: AWS profile, press enter
   ```

```
? Please choose the profile you want to use: default, press enter
```

3. Follow the prompts

   Proceed with the remaining steps:

```
Initialize the project with the above configuration? (Y/n)      Y
Select the authentication method you want to use: (Use arrow keys) AWS profile
Please choose the profile you want to use      default
```

4. Verify initialization

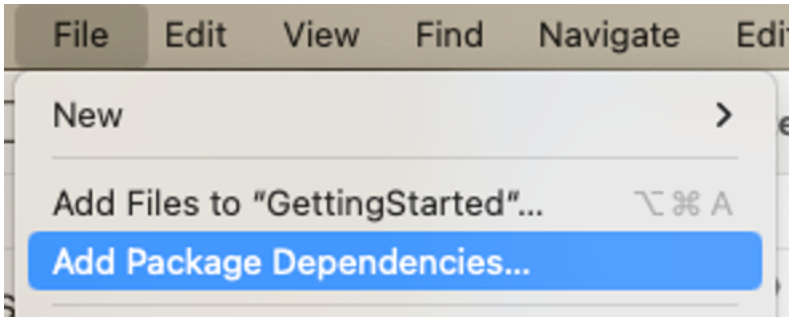
   This will provision the resources in the backend and might take a few minutes. Once it's done, you will see the following information:

```
Deployment state saved successfully.
# Initialized provider successfully.
# Initialized your environment successfully.
# Your project has been successfully initialized and connected
```

Step 3: Add Amplify libraries to your project

1. Add dependencies

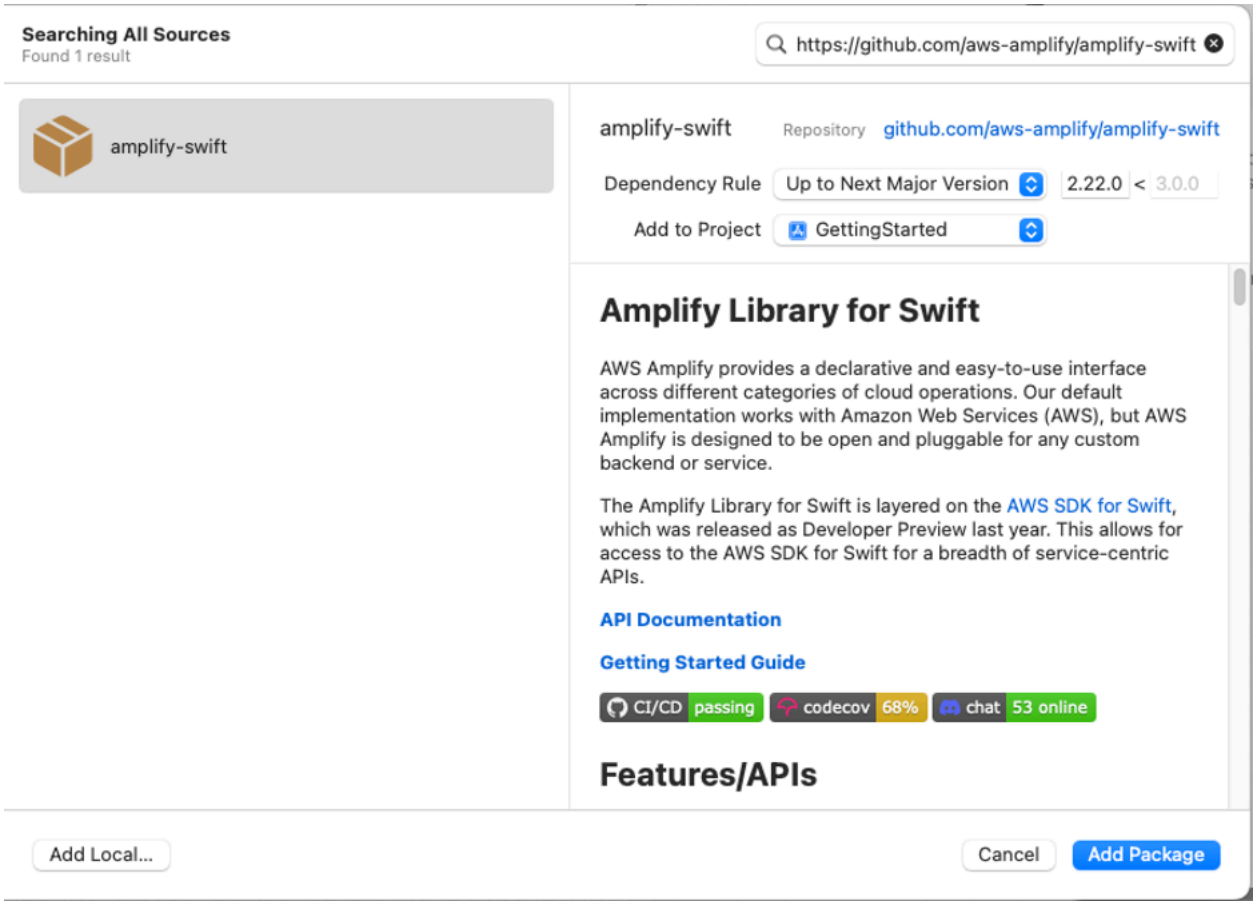
   Switch back to **Xcode**. Select **File** and choose **Add Package Dependencies...**



2. Search for the Amplify library

   Enter the **Amplify Libraries for Swift GitHub repo URL** (https://github.com/aws-amplify/amplify-swift) into the search bar, and press **Enter**.

   Make sure that **Up to Next Major Version** is selected from the **Dependency Rule** dropdown, and select **Add Package**.

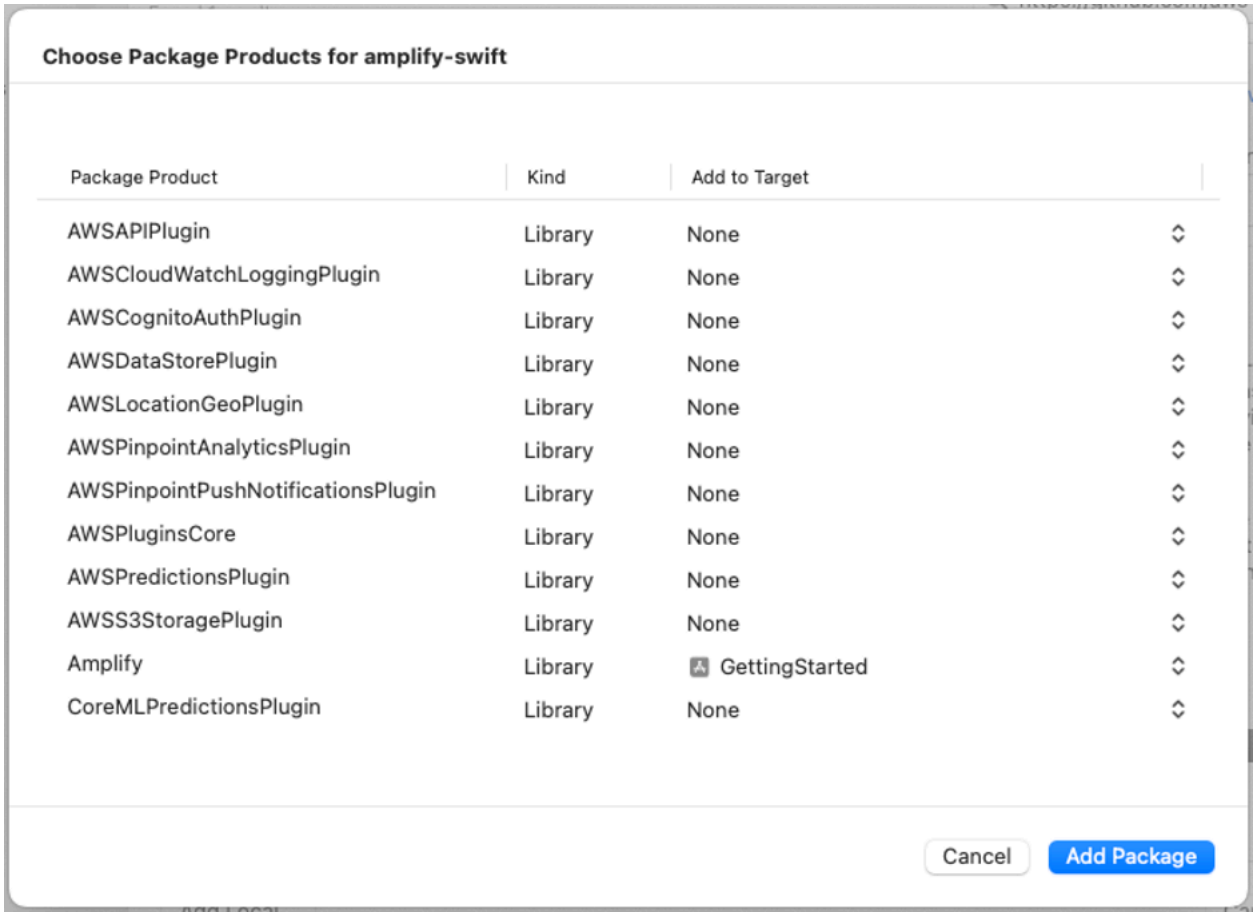


3. Select the library

   Once the libraries are fetched, you will be asked to select which ones you wish to add to your target.

   In the drop down, next to **Amplify**, choose **GettingStarted**.

   Select **None** in for the rest of the Package Products in the Add to Target section, and choose **Add Package**.



Step 4: Initialize Amplify at runtime

- Modify the file

  Open the **GettingStartedApp.swift file** and replace its content with the following information:

```
import Amplify
import SwiftUI

@main
struct GettingStartedApp: App {
    init() {
        do {
            try Amplify.configure()
            print("Initialized Amplify")
        } catch {
            print("Could not initialize Amplify: \(error)")
```

```
        }
    }

    var body: some Scene {
        WindowGroup {
            NotesView()
        }
    }
}
```

Step 5: Verify your setup

- Build the project

  To verify everything works as expected, build the project.

  Select the **Product** menu and then select **Build** , or press **Cmd + B** .

  There should be no error.

Conclusion

You have initialized the Amplify project and are now ready to start adding features! In the next module, we will add an entire user authentication flow with just a few lines of code.

Module 3: Add Authentication

Time to complete	5 minutes
Services used	AWS Amplify

Overview

The next feature you will be adding is user authentication. In this module, you will learn how to authenticate a user with the Amplify CLI and libraries, using Amazon Cognito, a managed user identity provider.

You will also learn how to use the Amazon Cognito hosted UI (user interface) to present an entire user authentication flow, allowing users to sign up, sign in, and reset their password with just a few lines of code.

Using a hosted UI means the application uses the Amazon Cognito web pages for the sign-in and sign-up UI flows. The user of the app is redirected to a web page hosted by Amazon Cognito and redirected back to the app after sign-in.

Amplify also offers a native UI component for authentication flows. You can follow these workshop instructions to learn more.

What you will accomplish

In this module, you will:

- Create and deploy an authentication service
- Configure your iOS app to include Amazon Cognito hosted UI authentication

Key concepts

Amplify libraries – Using Amplify libraries you can interact with AWS services from a web or mobile application.

Authentication – In software, authentication is the process of verifying and managing the identity of a user using an authentication service or API.

Implementation

Step 1: Create the authentication service

1. Add authorization

   To create the authentication service, open **Terminal** and run the following **command** in your project root directory:

   ```
   amplify add auth
   ```

2. Configure authorization options

   Select the following **options**, when prompted:

   ```
   Do you want to use the default authentication and security configuration?
        # Default configuration with Social Provider (Federation)
   How do you want users to be able to sign in?
        # Username
   Do you want to configure advanced settings?
        # No, I am done.
   What domain name prefix do you want to use?
        «default value»
   Enter your redirect signin URI:
        gettingstarted://
   Do you want to add another redirect signin URI? (y/N)
        N
   Enter your redirect signout URI:
        gettingstarted://
   Do you want to add another redirect signout URI? (y/N)
        N
   Select the social providers you want to configure for your user pool:
        «Do not select any and just press enter»
   ```

   > **Note**
   >
   > Do not forget to type the redirect URIs. They are needed for the redirection for Amazon Cognito Hosted UI to work.

3. Deploy the service

Run the following **command** to deploy the service:

```
amplify push
```

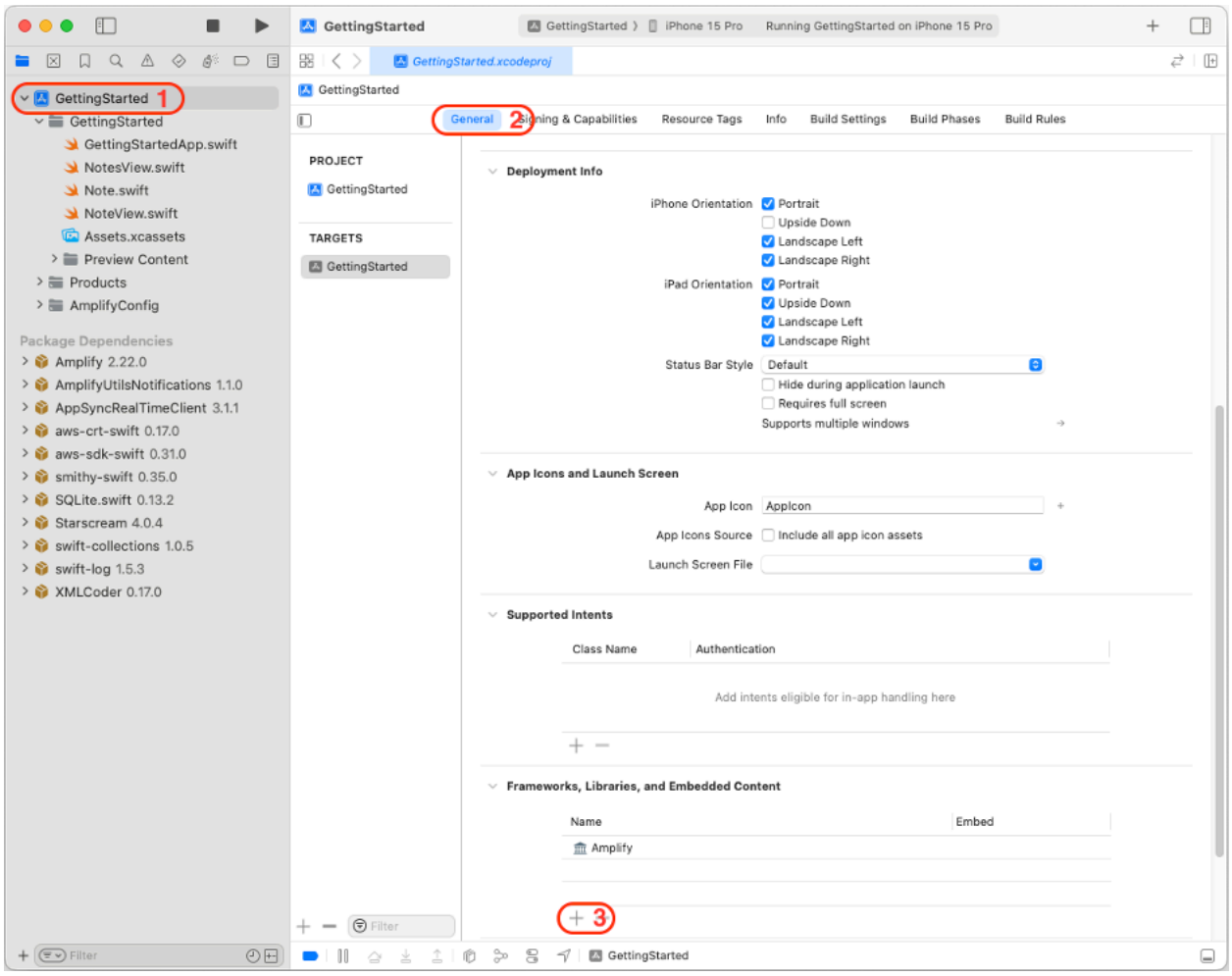
Press **Enter (Y)** when asked to continue.

Step 2: Add the Amplify Authentication library to the project

1. Open the general tab

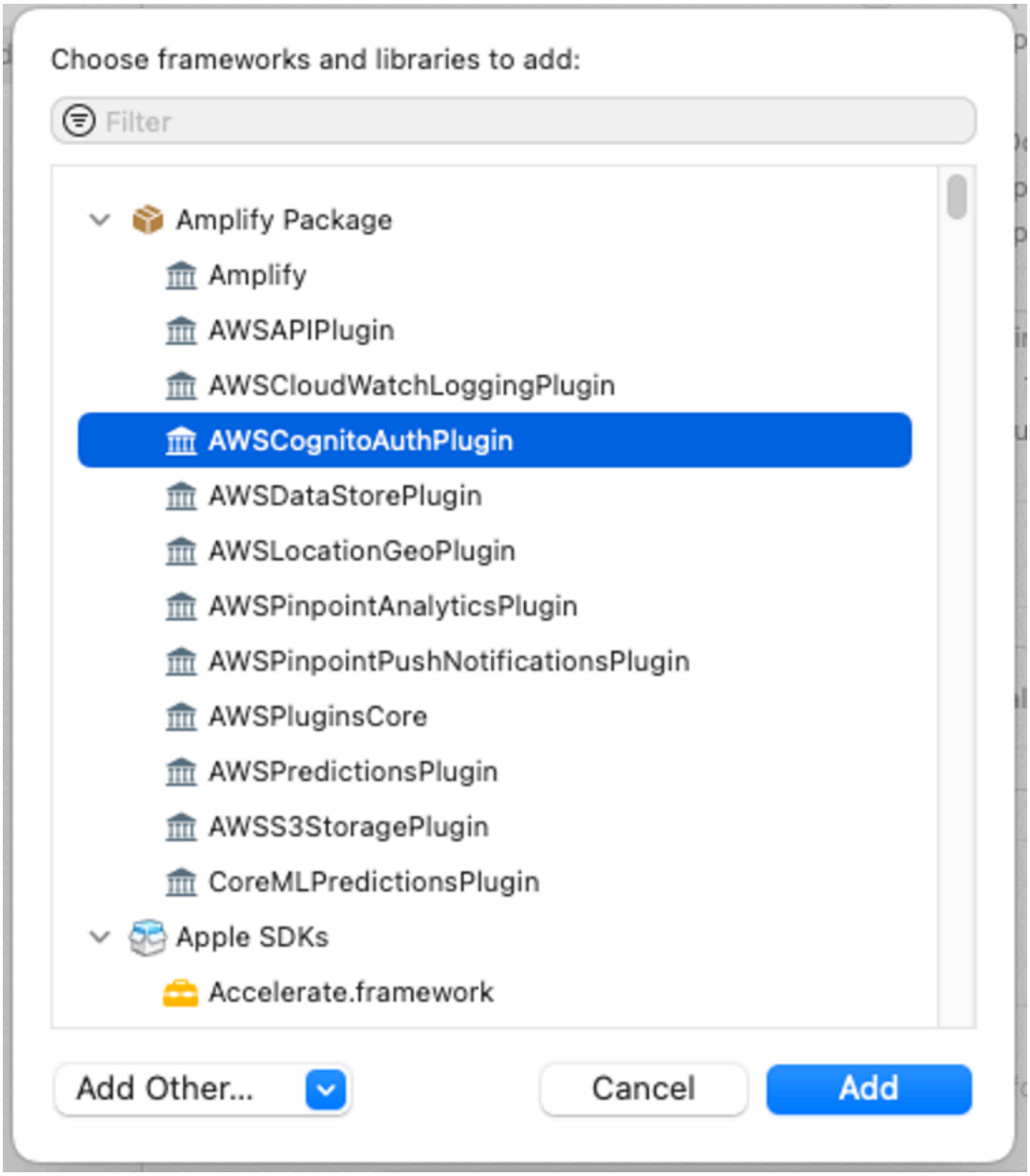
   To add the Amplify Authentication library to the dependencies of your project, navigate to the **General** tab of your Target application (Your Project > Targets > General).

   Select the **plus (+)** in the **Frameworks, Libraries, and Embedded Content** section.



2. Select the plugin

Select **AWSCognitoAuthPlugin**,and choose **Add**.



Step 3: Configure the Amplify Authentication library at runtime

- Configure authentication

  Navigate back to Xcode, and open the **GettingStartedApp.swift** file. To configure Amplify Authentication, you will need to:

  - Add the **import AWSCognitoAuthPlugin** statement.

- Create the **AWSCognitoAuthPlugin** plugin and register it with.

Your code should look like the following:

```
import Amplify
import AWSCognitoAuthPlugin
import SwiftUI

@main
struct GettingStartedApp: App {
    init() {
        do {
            try Amplify.add(plugin: AWSCognitoAuthPlugin())
            try Amplify.configure()
            print("Initialized Amplify")
        } catch {
            print("Could not initialize Amplify: \(error)")
        }
    }

    var body: some Scene {
        WindowGroup {
            NotesView()
        }
    }
}
```

Step 4: Create a class to support authentication operations

Create a **new Swift file** named **AuthenticationService.swift** with the following content:

```
import Amplify
import AuthenticationServices
import AWSCognitoAuthPlugin
import SwiftUI

@MainActor
class AuthenticationService: ObservableObject {
    @Published var isSignedIn = false

    func fetchSession() async {
```

```
        do {
            let result = try await Amplify.Auth.fetchAuthSession()
            isSignedIn = result.isSignedIn
            print("Fetch session completed. isSignedIn = \(isSignedIn)")
        } catch {
            print("Fetch Session failed with error: \(error)")
        }
    }

    func signIn(presentationAnchor: ASPresentationAnchor) async {
        do {
            let result = try await Amplify.Auth.signInWithWebUI(
                presentationAnchor: presentationAnchor,
                options: .preferPrivateSession()
            )
            isSignedIn = result.isSignedIn
            print("Sign In completed. isSignedIn = \(isSignedIn)")
        } catch {
            print("Sign In failed with error: \(error)")
        }
    }

    func signOut() async {
        guard let result = await Amplify.Auth.signOut() as? AWSCognitoSignOutResult
 else {
            return
        }

        switch result {
        case .complete, .partial:
            isSignedIn = false
        case .failed:
            break
        }

        print("Sign Out completed. isSignedIn = \(isSignedIn)")
    }
}
```

This class takes care of handling authentication by relying on Amplify's HostedUI capabilities, while also informing whether a user is signed in or not.

Step 5: Update the UI with authentication

1. Create authentication for signing in

   Create a **new Swift file** named **LandingView.swift** with the following content:

```
import AuthenticationServices
import SwiftUI

struct LandingView: View {
    @EnvironmentObject private var authenticationService: AuthenticationService
    @State private var isLoading = true

    var body: some View {
        ZStack {
            if isLoading {
                ProgressView()
            }

            Group {
                if authenticationService.isSignedIn {
                    NotesView()
                } else {
                    Button("Sign in") {
                        Task {
                            await authenticationService.signIn(presentationAnchor:
 window)
                        }
                    }
                }
            }
            .opacity(isLoading ? 0.5 : 1)
            .disabled(isLoading)
        }
        .task {
            isLoading = true
            await authenticationService.fetchSession()
            if !authenticationService.isSignedIn {
                await authenticationService.signIn(presentationAnchor: window)
            }
            isLoading = false
        }
    }
```

```
    private var window: ASPresentationAnchor {
        if let delegate = UIApplication.shared.connectedScenes.first?.delegate as?
 UIWindowSceneDelegate,
           let window = delegate.window as? UIWindow {
            return window
        }
        return ASPresentationAnchor()
    }
}
```

This view takes care of the following:

- Before the view first appears, it will fetch the current user status using the **authenticationService** property. This is done in the view's **task(priority:_:)** method.
- If the user is signed in, it will display the **NotesView**.
- If the user is not signed in, it will automatically invoke the HostedUI workflow. If the user does not sign in and closes that workflow, a "Sign In" button will be displayed that can be used to start the authentication workflow again.

We've marked the authenticationService variable with a **@EnvironmentObject** property wrapper annotation, meaning we need to set it using the **environmentObject(_:)** view modifier on an ancestor view.

Update the **GettingStartedApp.swift** file body to create this view instead, and to set the **AuthenticationService** object with the following:

```
var body: some Scene {
    WindowGroup {
        LandingView()
            .environmentObject(AuthenticationService())
    }
}
```

2. Create authentication for signing out

   Open the **NotesView.swift** file and replace its contents with the following:

```
struct NotesView: View {
    @EnvironmentObject private var authenticationService: AuthenticationService
```

```
    @State var notes: [Note] = []

    var body: some View {
        NavigationStack {
            List {
                if notes.isEmpty {
                    Text("No notes")
                }
                ForEach(notes, id: \.id) { note in
                    NoteView(note: note)
                }
            }
            .navigationTitle("Notes")
            .toolbar {
                Button("Sign Out") {
                    Task {
                        await authenticationService.signOut()
                    }
                }
            }
        }
    }
}
```

We've added a **Sign Out** button in the toolbar that calls **authenticationService.signOut()**. As this object has already been added when creating the LandingView ancestor, we don't need to do anything else.

Step 6: Build and test

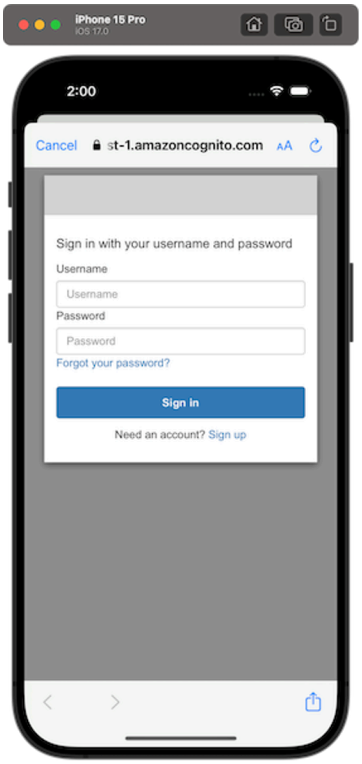
Build and launch the app in the simulator by pressing the ▶ button in the toolbar. Alternatively, you can also do it by going to **Product -> Run**, or by pressing **Cmd + R**.

The iOS simulator will open and the app should prompt you to sign in first.

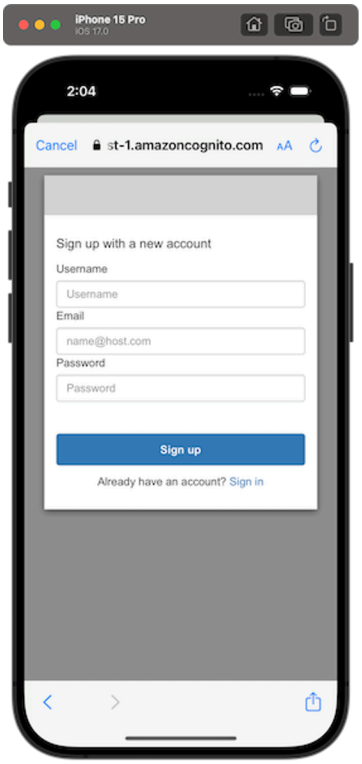
Once you've finished that process, the Notes view will display with a **Sign Out** button at the top right. If you choose it, you should land in the Sign in view again.

Landing view

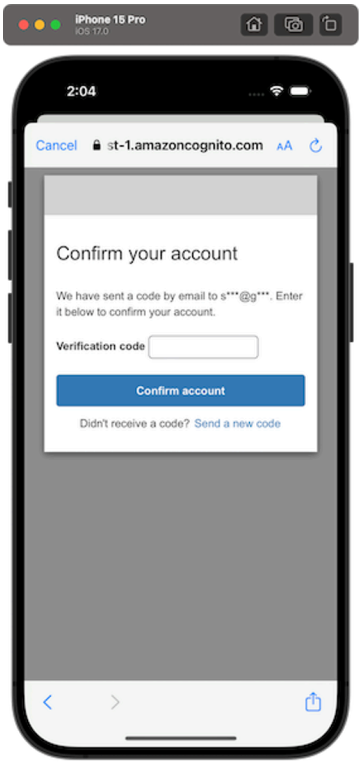
Hosted UI: Sign in



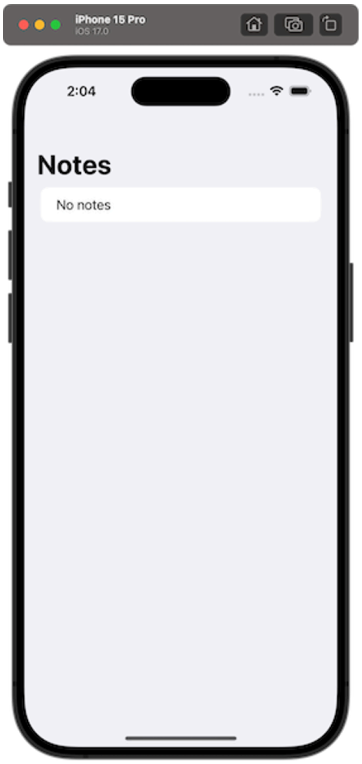
Hosted UI: Sign up



Hosted UI: Confirmation



List of notes



Module 4: Add a GraphQL API service and a database

Time to complete	20 minutes
Services used	AWS Amplify

Overview

Now that you have created and configured the app with user authentication, you will add an API and create, read, update, delete (CRUD) operations on a database.

In this module, you will add an API to our app using the Amplify CLI and libraries. The API you will be creating is a GraphQL API that uses AWS AppSync (a managed GraphQL service) which is backed by Amazon DynamoDB (a NoSQL database). For an introduction to GraphQL, visit this page.

The app you will be building is a note-taking app where users can create, delete, and list notes. This example gives you a good idea of how to build many popular types of CRUD+L (create, read, update, delete, and list) applications.

What you will accomplish

In this tutorial, you will:

- Create and deploy a GraphQL API
- Write frontend code to interact with the API

Key concepts

API – Provides a programming interface that allows communication and interactions between multiple software intermediaries.

GraphQL – A query language and server-side API implementation based on a typed representation of your application. This API representation is declared using a schema based on the GraphQL type system. To learn more about GraphQL, visit this page.

Implementation

Step 1: Create a GraphQL API service and a database

1. Add the Amplify API

   Open the **Terminal,** navigate to your **project root directory** , and run the following **command**:

   ```
   amplify add api
   ```

2. Configure the API

   When prompted, make the following selections:

   ```
   Select from one of the below mentioned services:
       # GraphQL
   Authorization modes:
       Choose the default authorization type for the API
           # Amazon Cognito User Pool
   Configure additional auth types?
       N
   ```

3. Confirm selections

   Validate the selected options, and choose **Continue.**

   ```
   Here is the GraphQL API that we will create. Select a setting to edit or continue
    (Use arrow keys)
     Name: gettingstarted
     Authorization modes: Amazon Cognito User Pool (default)
     Conflict detection (required for DataStore): Disabled
   # Continue
   ```

4. Edit the schema

   Select **Blank Schema** and **choose Y** when asked to edit the schema:

   ```
   Choose a schema template:
       # Blank Schema
   Do you want to edit the schema now?
       Y
   ```

5. Update the schema

   As we want to represent the model we previously defined in the **Note.swift** file, use the following schema and **save** the file:

   ```
   type Note
   @model
   @auth (rules: [ { allow: owner } ]) {
       id: ID!
       name: String!
       description: String
       image: String
   }
   ```

   The data model is made of one class named **Note** and four String properties: **id** and **name** are mandatory; **description** and **image** are optional.

   - The **@model** transformer indicates we want to create a database to store these data.
   - The **@auth** transformer adds authentication rules to allow access to these data. For this project, we want only the owner of Notes to have access to them.

   Delete the **Note.swift** file, we will re-generate the models in the next step.

Step 2: Generate client-side code

Amplify generates client-side code.

- Generate the code

  To generate the code, run the following **command** in your terminal:

  ```
  amplify codegen models
  ```

  This creates Swift files in the **amplify/generated/models** directory and automatically add them to your project.

Step 3: Deploy the API service and database

1. Deploy the backend database

   To deploy the backend API and database we have just created, in your terminal run the following **command**:

   ```
   amplify push
   ```

2. Push the deployment

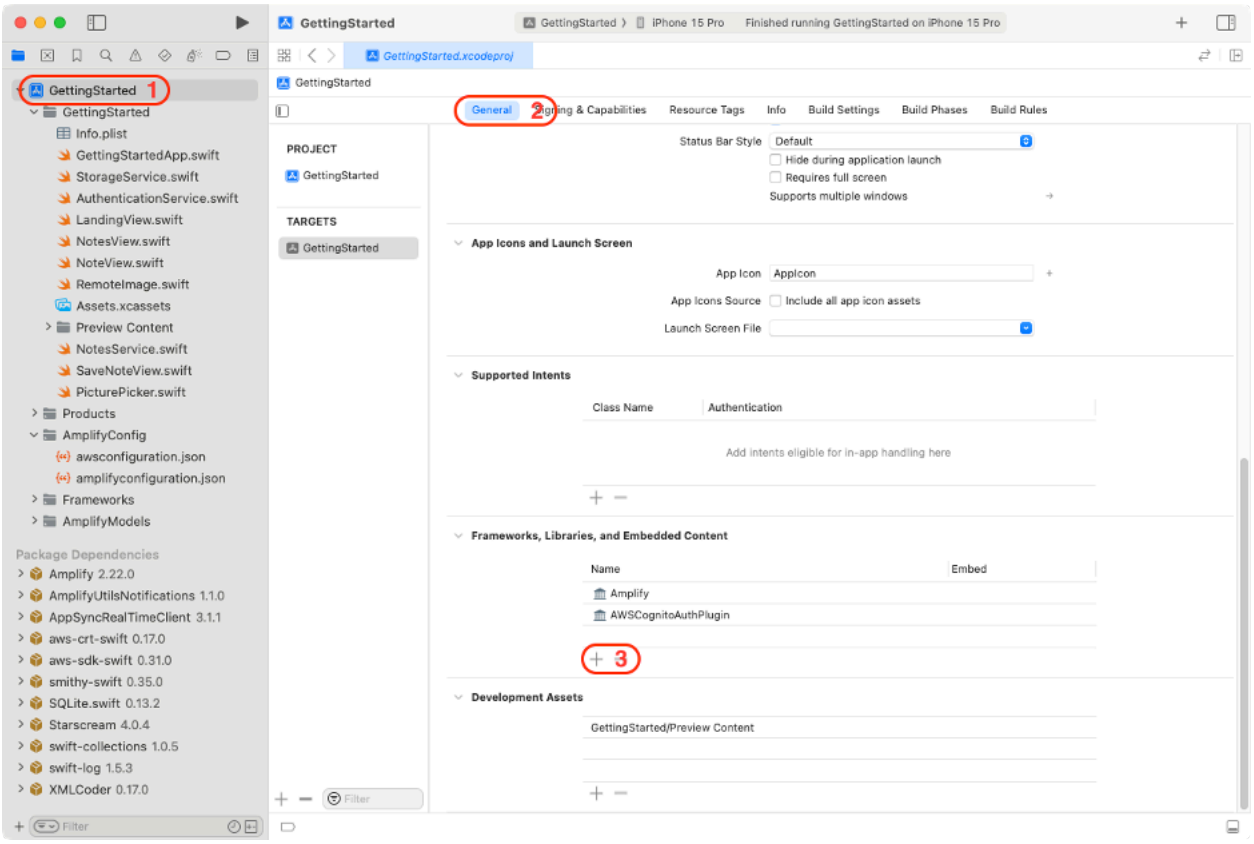
   When prompted, make the following selections:

   ```
   amplify push
   Are you sure you want to continue?
       Y
   Do you want to generate code for your newly created GraphQL API?
       N
   ```

Step 4: Add API client library to the project

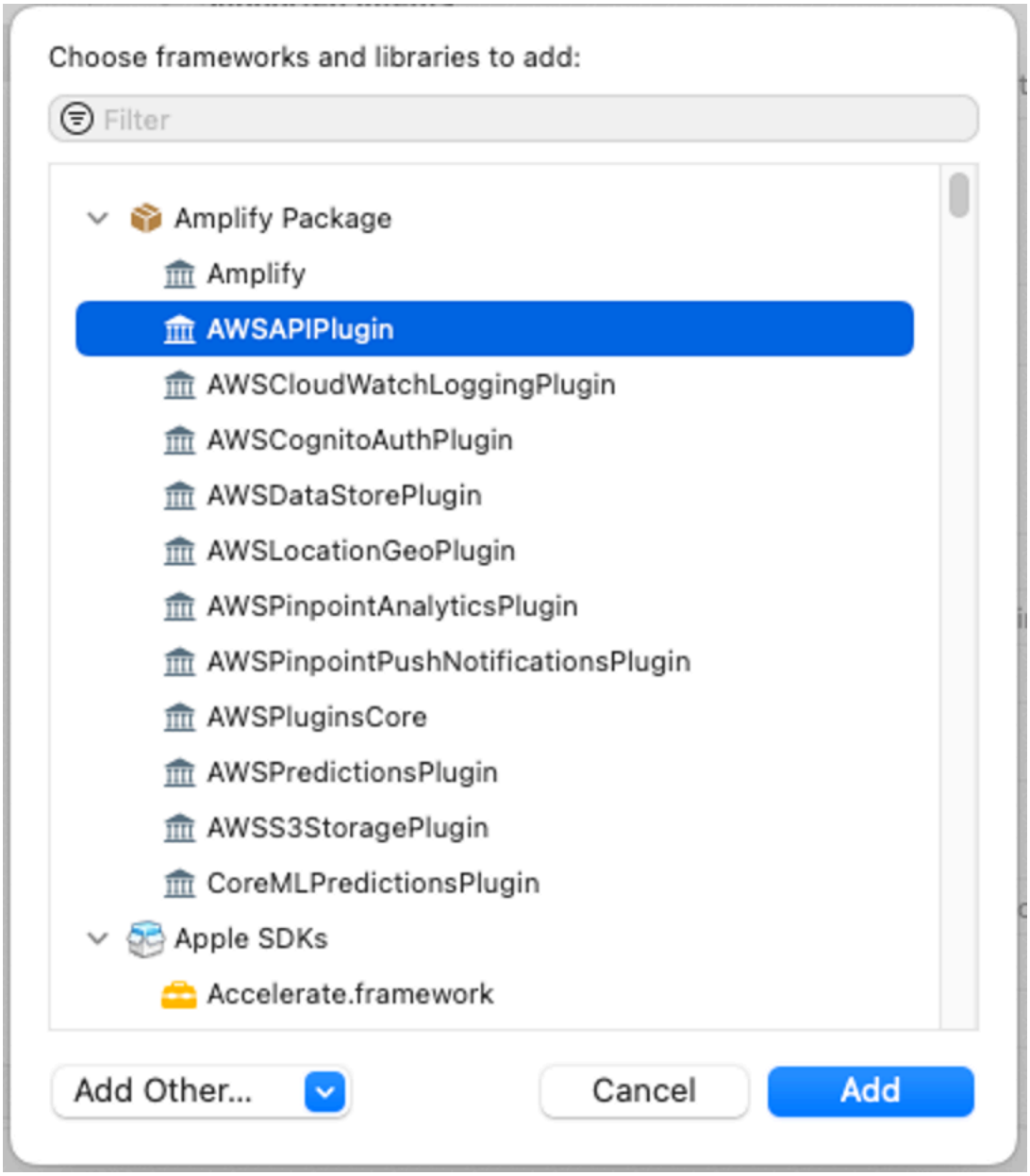
1. Open the general tab

   Navigate to the **General** tab of your Target application (Your Project > Targets > General), and select the **plus (+)** in the **Frameworks, Libraries, and Embedded Content** section.



2. Choose the plugin

   Choose the **AWSAPIPlugin** , and select **Add**.



3. Verify the dependency created

   You have now added **AWSAPIPlugin** as a dependency for your project.

Step 5: Configure the Amplify API library at runtime

- Configure the library

  Navigate back to **Xcode** , and open the **GettingStartedApp.swift** file.

  To configure Amplify API, you will need to:

  - Add the **import AWSAPIPlugin** statement.
  - Create the **AWSAPIPlugin** plugin and register it with Amplify.

  Your code should look like the following:

```
import Amplify
import AWSAPIPlugin
import AWSCognitoAuthPlugin
import SwiftUI

@main
struct GettingStartedApp: App {
    init() {
        do {
            try Amplify.add(plugin: AWSCognitoAuthPlugin())
            try Amplify.add(plugin: AWSAPIPlugin(modelRegistration:
 AmplifyModels()))
            try Amplify.configure()
            print("Initialized Amplify");
        } catch {
            print("Could not initialize Amplify: \(error)")
        }
    }

    var body: some Scene {
        WindowGroup {
            LandingView()
                .environmentObject(AuthenticationService())
        }
    }
}
```

Step 6: Create a class to support API CRUD operations

- Create a NotesService.swift file

  Create a **new Swift file** named **NotesService.swift** with the following code.

```
import Amplify
import SwiftUI

@MainActor
class NotesService: ObservableObject {
    @Published var notes: [Note] = []

    func fetchNotes() async {
        do {
            let result = try await Amplify.API.query(request: .list(Note.self))
            switch result {
            case .success(let notesList):
                print("Fetched \(notesList.count) notes")
                notes = notesList.elements
            case .failure(let error):
                print("Fetch Notes failed with error: \(error)")
            }
        } catch {
            print("Fetch Notes failed with error: \(error)")
        }
    }

    func save(_ note: Note) async {
        do {
            let result = try await Amplify.API.mutate(request: .create(note))
            switch result {
            case .success(let note):
                print("Save note completed")
                notes.append(note)
            case .failure(let error):
                print("Save Note failed with error: \(error)")
            }
        } catch {
            print("Save Note failed with error: \(error)")
        }
    }
```

```
    func delete(_ note: Note) async {
        do {
            let result = try await Amplify.API.mutate(request: .delete(note))
            switch result {
            case .success(let note):
                print("Delete note completed")
                notes.removeAll(where: { $0.id == note.id })
            case .failure(let error):
                print("Delete Note failed with error: \(error)")
            }
        } catch {
            print("Delete Note failed with error: \(error)")
        }
    }
}
```

This class allows to fetch all notes, save a new note, and delete an existing note, while also publishing the fetched notes in a notes array.

Step 7: Update the existing UI

- List notes

  Make the following changes to the **NotesView.swift** file:

  - Add a new **@EnvironmentObject private var notesService: NotesService** property
  - Delete the local **notes** array and instead use published **notesService.notes** when creating the List items in the ForEach loop.
  - Call **notesService.fetchNotes()** when the view appears. We can do this using the **task(priority:_:)** method.

  Your file should look like the following code.

```
struct NotesView: View {
    @EnvironmentObject private var authenticationService: AuthenticationService
    @EnvironmentObject private var notesService: NotesService

    var body: some View {
        NavigationStack{
            List {
```

```
                if notesService.notes.isEmpty {
                    Text("No notes")
                }
                ForEach(notesService.notes, id: \.id) { note in
                    NoteView(note: note)
                }
            }
            .navigationTitle("Notes")
            .toolbar {
                Button("Sign Out") {
                    Task {
                        await authenticationService.signOut()
                    }
                }
            }
        }
        .task {
            await notesService.fetchNotes()
        }
    }
}
```

Step 8: Build and test

1. Run the project

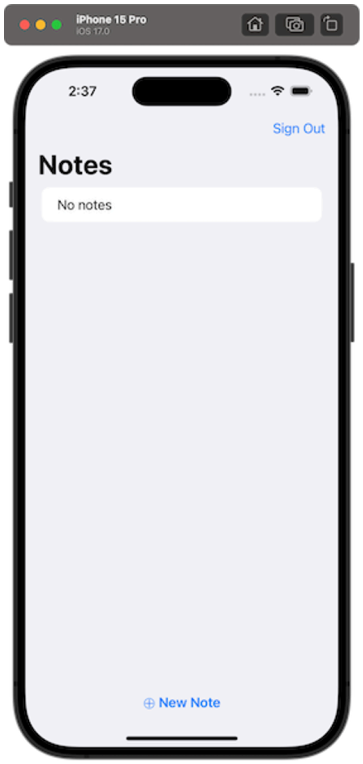
   To verify everything works as expected, build, and run the project.

   Choose the ► button in the toolbar. Alternatively, you can also do it by navigating to **Product -> Run** , or by pressing **Cmd + R** .

   The iOS simulator will open and the app should show you the Notes view, assuming you are still signed in.

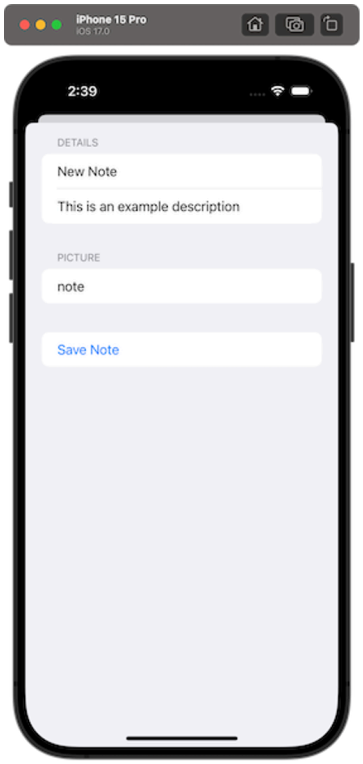
2. Create a new note

   Choose the " **# New Note** " button at the bottom to create a new list.



3. Enter details

   Enter details for the note and choose **Save Note**.

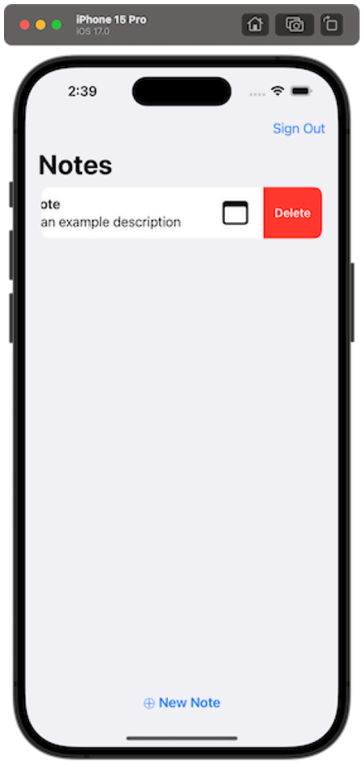


4. View note

View the note in the list.

5. Delete the note

   You can delete a note by swiping from the left of its row.



Conclusion

You have now added a GraphQL API and configured create, read, and delete functionality in your app. In the next module, we will add UI and behavior to manage pictures.

Module 5: Add the Ability to Store Images

Time to complete	5 minutes
Services used	AWS Amplify

Overview

Now that the notes app is working, you will add the ability to associate an image with each note.

In this module, you will use the Amplify CLI and libraries to create a storage service using Amazon S3. Then, you will update the iOS app to enable image uploading, fetching, and rendering.

What you will accomplish

In this tutorial, you will:

- Create a storage service
- Update your iOS app with logic to upload and download images
- Update the UI of your iOS app

Key concepts

Storage service – Storing and querying of files, such as images and videos, is a common requirement for applications. One option to do this is to Base64 encode the file and send it as a string to save in the database. This comes with disadvantages, such as the encoded file being larger than the original binary, the operation being computationally expensive, and the added complexity around encoding and decoding properly. Another option is to have a storage service specifically built and optimized for file storage.

Storage services like Amazon S3 exist to make this as easy, performant, and inexpensive as possible.

Implementation

Step 1: Create the storage service

1. Add storage

   Open the **Terminal,** navigate to your **project root directory**, and run the following **command**:

   ```
   amplify add storage
   ```

2. Configure options

   When prompted, make the following selections:

   ```
   Select from one of the below mentioned services:
       # Content (Images, audio, video, etc.)

   Provide a friendly name for your resource that will be used to label this category
    in the project
       image

   Provide bucket name
       «default value»

   Who should have access:
       # Auth users only

   What kind of access do you want for Authenticated users?
       # create/update
       # read
       # delete
       «i.e. select all options»

   Do you want to add a Lambda Trigger for your S3 Bucket?
       N
   ```

3. Deploy the service

   Finally, deploy the service by running the following **command**:

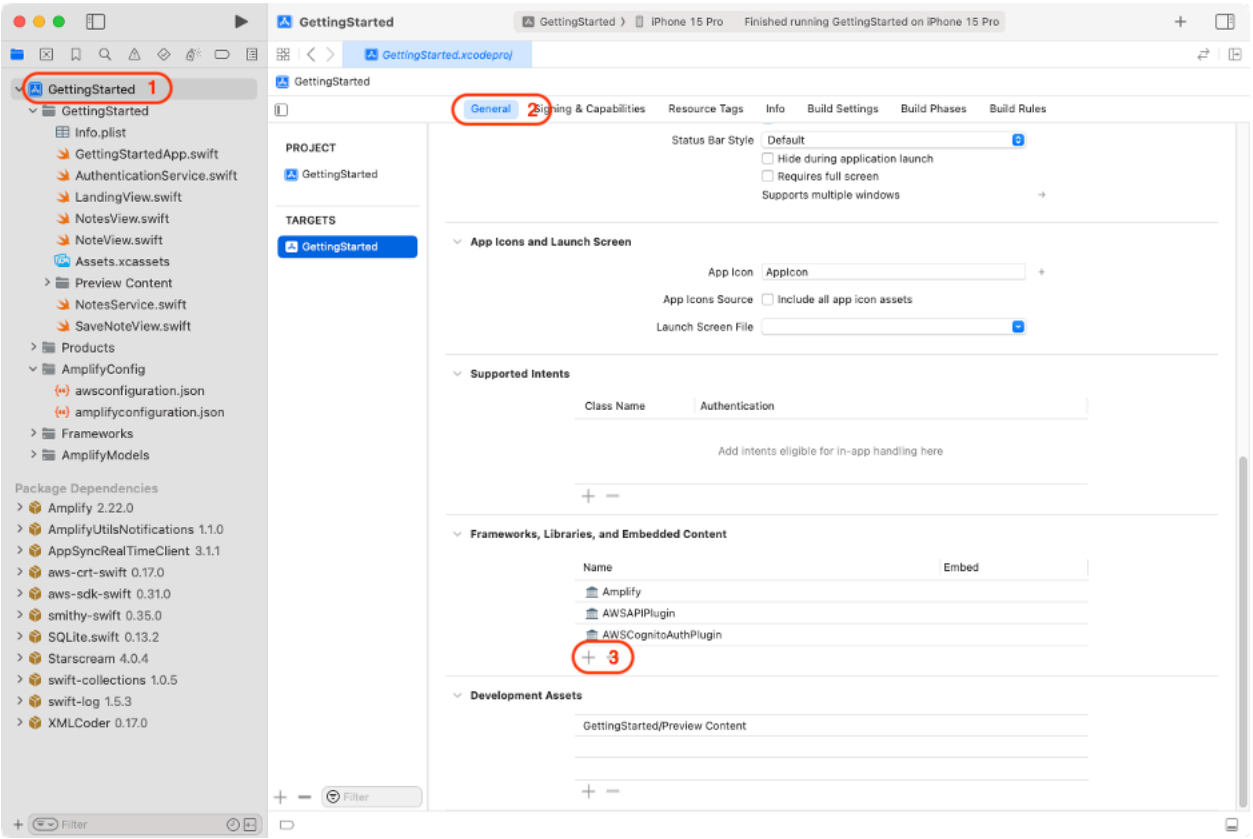
   ```
   amplify push
   ```

```
Are you sure you want to continue?
    Y
```

Step 2: Add the Amplify Storage library to the project

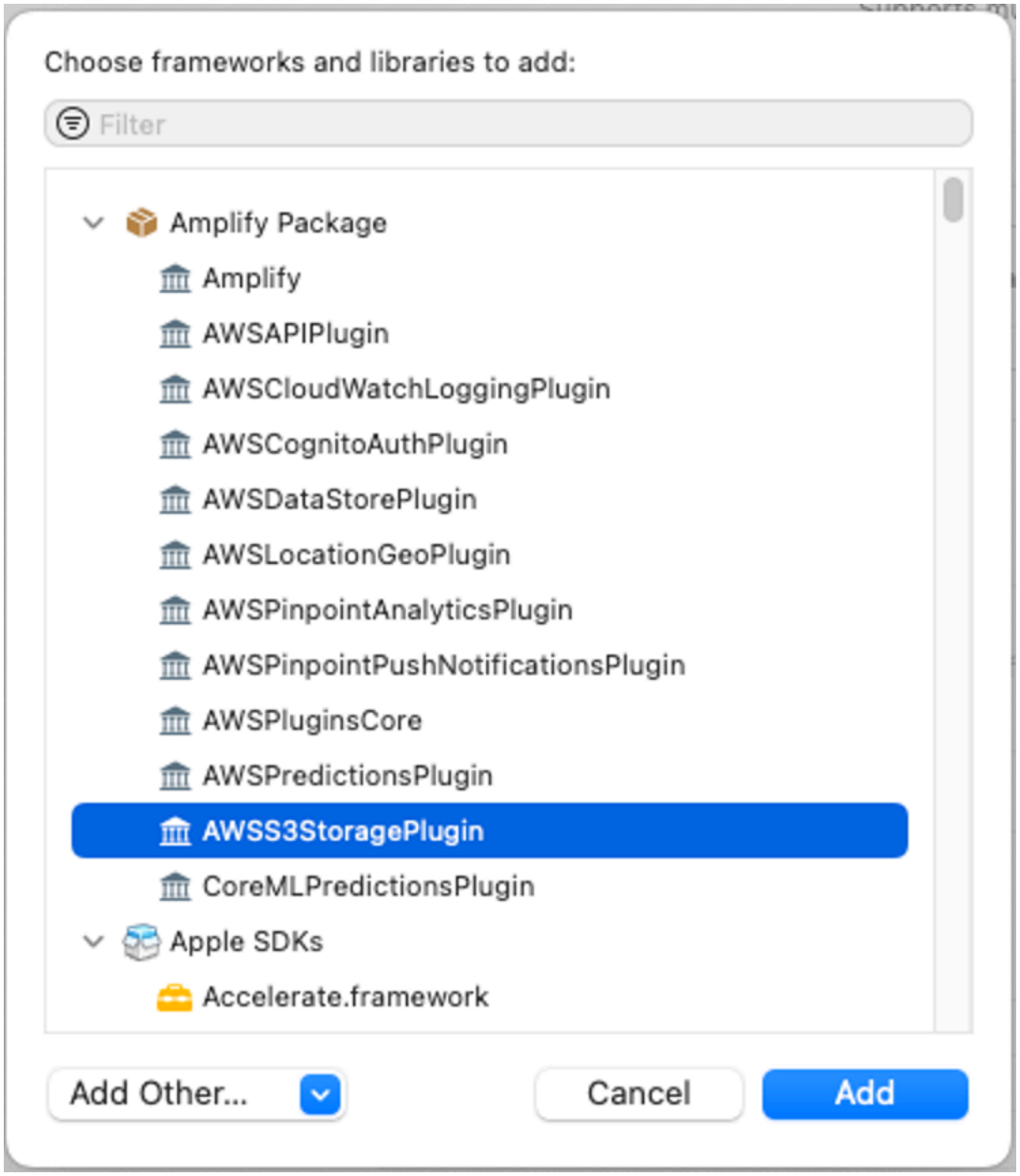
1. Open the general tab

   Navigate to the **General** tab of your Target application (Your Project > Targets > General), and select the **plus (+)** in the **Frameworks, Libraries, and Embedded Content** section.



2. Choose the plugin

   Choose **AWSS3StoragePlugin**, and select **Add**.



3. Verify dependency

   You will see **AWSS3StoragePlugin** as a dependency for your project.

Step 3: Configure the Amplify Storage library at runtime

- Modify the Xcode

  Navigate back to **Xcode** and open the **GettingStartedApp.swift** file. To configure Amplify API, you will need to:

  - Add the**import AWSS3StoragePlugin**statement.
  - Create the **AWSS3StoragePlugin**plugin, and register it with Amplify.

  Your code should look like the following.

```
import Amplify
import AWSAPIPlugin
import AWSCognitoAuthPlugin
import AWSS3StoragePlugin
import SwiftUI

@main
struct GettingStartedApp: App {
    init() {
        do {
            try Amplify.add(plugin: AWSCognitoAuthPlugin())
            try Amplify.add(plugin: AWSAPIPlugin(modelRegistration:
 AmplifyModels()))
            try Amplify.add(plugin: AWSS3StoragePlugin())
            try Amplify.configure()
            print("Initialized Amplify");
        } catch {
            print("Could not initialize Amplify: \(error)")
        }
    }

    var body: some Scene {
        WindowGroup {
            LandingView()
                .environmentObject(NotesService())
                .environmentObject(AuthenticationService())
        }
    }
}
```

Step 4: Create a class to support image CRUD operations

- Create a StorageSwift file

  Create a **new Swift file** named **StorageService.swift** with the following content:

```
import Amplify
import Foundation

class StorageService: ObservableObject {
    func upload(_ data: Data, name: String) async {
        let task = Amplify.Storage.uploadData(
            key: name,
            data: data,
            options: .init(accessLevel: .private)
        )
        do {
            let result = try await task.value
            print("Upload data completed with result: \(result)")
        } catch {
            print("Upload data failed with error: \(error)")
        }
    }

    func download(withName name: String) async -> Data? {
        let task = Amplify.Storage.downloadData(
            key: name,
            options: .init(accessLevel: .private)
        )
        do {
            let result = try await task.value
            print("Download data completed")
            return result
        } catch {
            print("Download data failed with error: \(error)")
            return nil
        }
    }

    func remove(withName name: String) async {
        do {
            let result = try await Amplify.Storage.remove(
                key: name,
```

```
                options: .init(accessLevel: .private)
            )
            print("Remove completed with result: \(result)")
        } catch {
            print("Remove failed with error: \(error)")
        }
    }
}
```

The methods in this class simply call their Amplify counterpart. Amplify Storage has three file protection levels:

- **Public**: Accessible by all users
- **Protected**: Readable by all users, but only writable by the creating user
- **Private**: Readable and writable only by the creating user

For this app, we want the images to only be available to the note owner, so we set the **accessLevel: .private property** in each operation's options.

Step 5: Update the existing UI

1. Create a RemoteImage file

   Create a **new Swift file** named **RemoteImage.swift** with the following content:

```
import SwiftUI

struct RemoteImage: View {
    @EnvironmentObject private var storageService: StorageService
    @State private var image: UIImage? = nil
    @State private var isLoading = true
    var name: String

    var body: some View {
        content
            .task {
                if let data = await storageService.download(withName: name) {
                    image = UIImage(data: data)
                }
                isLoading = false
```

```
            }
    }

    @ViewBuilder
    private var content: some View {
        if isLoading {
            ProgressView()
        } else if let image {
            Image(uiImage: image)
                .resizable()
                .aspectRatio(contentMode: .fill)
        } else {
            EmptyView()
        }
    }
}
```

This view will attempt to download the data using the storage service and the provided name, while displaying a loading view while the operation is in progress. If the data cannot be downloaded, it shows an empty view.

2. Update the NoteView file

   Next, update **NoteView.swift** to use this new view when displaying the image:

```
    if let image = note.image {
        Spacer()
        RemoteImage(name: image)
            .frame(width: 30, height: 30)
    }
```

3. Update the GettingStartedApp file

   Finally, update the **GettingStartedApp.swift's** body to set the **StorageService** object:

```
    var body: some Scene {
        WindowGroup {
            LandingView()
                .environmentObject(NotesService())
                .environmentObject(AuthenticationService())
                .environmentObject(StorageService())
        }
```

```
    }
```

4. Create a PicturePicker file

   In order to allow the user to select a picture from their library, create a **new Swift file** named **PicturePicker.swift** with the following content:

```
import PhotosUI
import SwiftUI

struct PicturePicker: View {
    @State private var selectedPhoto: PhotosPickerItem? = nil
    @Binding var selectedData: Data?

    var body: some View {
        VStack {
            if let selectedData, let image = UIImage(data: selectedData) {
                Image(uiImage: image)
                    .resizable()
                    .frame(width: 100, height: 100)
                    .clipShape(Circle())
                    .overlay(Circle().stroke(Color.white, lineWidth: 4))
                    .shadow(radius: 10)
            }

            PhotosPicker(title, selection: $selectedPhoto)
        }
        .onChange(of: selectedPhoto) {
            Task {
                selectedData = try? await selectedPhoto?.loadTransferable(type:
 Data.self)
            }
        }
    }

    private var title: String {
        return selectedPhoto == nil ? "Choose a picture" : "Change picture"
    }
}
```

5. Update the SaveNoteView file

   Make the following changes to the **SaveNoteView.swift** files:

- Add a new **@EnvironmentObject private var storageService: StorageService** property.
- Replace the type of the image property to **Data** instead of **String**.
- Display **PicturePicker(selectedData: $image)** on the **Picture** section instead of a text field.
- Modify the **Save Note** button's action to also save the image using storageService. Keep in mind that the note's image value should match the name you give to the stored image.

You file should look like the following:

```
struct SaveNoteView: View {
    @Environment(\.dismiss) private var dismiss
    @EnvironmentObject private var notesService: NotesService
    @EnvironmentObject private var storageService: StorageService
    @State private var name = ""
    @State private var description = ""
    @State private var image: Data? = nil

    var body: some View {
        Form {
            Section("Details") {
                TextField("Name", text: $name)
                TextField("Description", text: $description)
            }

            Section("Picture") {
                PicturePicker(selectedData: $image)
            }

            Button("Save Note") {
                let imageName = image != nil ? UUID().uuidString : nil
                let note = Note(
                    name: name,
                    description: description.isEmpty ? nil : description,
                    image: imageName
                )

                Task {
                    if let image, let imageName {
                        await storageService.upload(image, name: imageName)
                    }
                    await notesService.save(note)
```

```
                    dismiss()
                }
            }
        }
    }
}
```

6. Configure image deletion

   To delete images that are associated with a note that is deleted, update the **NotesView.swift** file:

   - Add a new **@EnvironmentObject private var storageService: StorageService** property
   - Call **storageService.remove(withName:)** inside the **onDelete** callback after calling **notesService.delete(_:)**.

   Your file should look like the following:

```
struct NotesView: View {
    @EnvironmentObject private var authenticationService: AuthenticationService
    @EnvironmentObject private var notesService: NotesService
    @EnvironmentObject private var storageService: StorageService
    @State private var isSavingNote = false

    var body: some View {
        NavigationStack{
            List {
                if notesService.notes.isEmpty {
                    Text("No notes")
                }
                ForEach(notesService.notes, id: \.id) { note in
                    NoteView(note: note)
                }
                .onDelete { indices in
                    for index in indices {
                        let note = notesService.notes[index]
                        Task {
                            await notesService.delete(note)
                            if let image = note.image {
                                await storageService.remove(withName: image)
                            }
```

```
                        }
                    }
                }
            }
            .navigationTitle("Notes")
            .toolbar {
                Button("Sign Out") {
                    Task {
                        await authenticationService.signOut()
                    }
                }
            }
            .toolbar {
                ToolbarItem(placement: .bottomBar) {
                    Button("# New Note") {
                        isSavingNote = true
                    }
                    .bold()
                }
            }
            .sheet(isPresented: $isSavingNote) {
                SaveNoteView()
            }
        }
        .task {
            await notesService.fetchNotes()
        }
    }
}
```

Step 6: Build and test

- Run the project

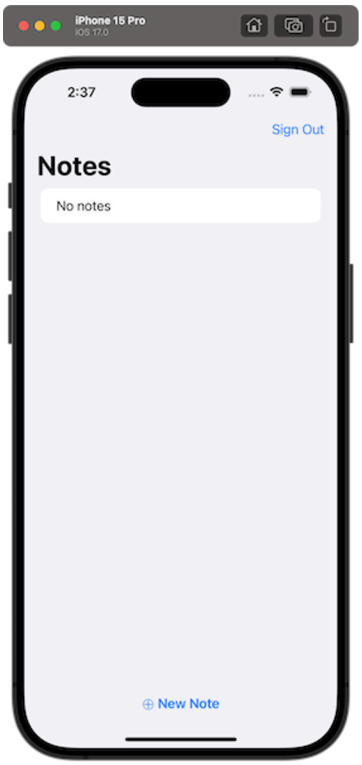
  To verify everything works as expected, build, and run the project.

  Choose the ▶ button in the toolbar. Alternatively, you can also do it by going to **Product -> Run**, or by pressing **Cmd + R**.

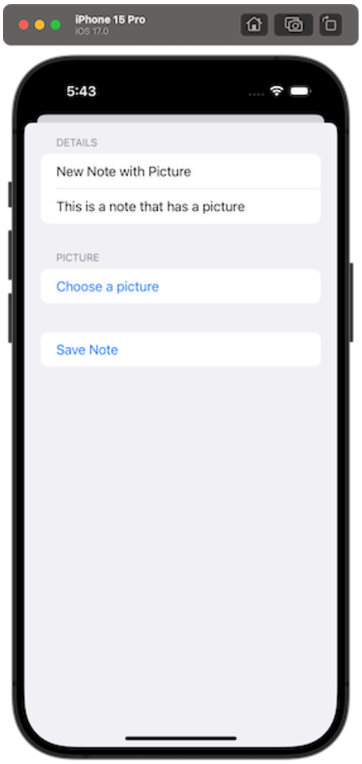
  The iOS simulator will open and the app should show you the Notes view, assuming you are still signed in.

You can tap on the "**# New Note**" button at the bottom to create a new list, and now you should be able to select a picture from the device's photo library.

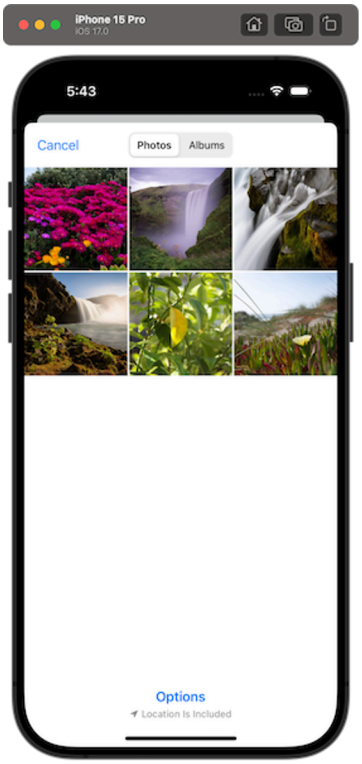
List of notes



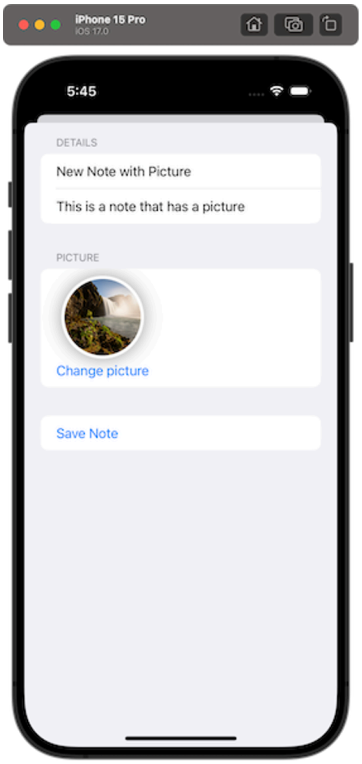
Create a note



Select a picture



Note with picture



(Optional) Step 7: Share your backend among multiple projects

Amplify makes it easy to share a single backend among multiple frontend applications.

1. Synchronize your local project

   Open the **Terminal**, navigate to your other **project directory**, and run the following **command**:

   ```
   amplify pull
   ```

2. Configure options

   When prompted, make the following selections:

   ```
   Select the authentication method you want to use
       # AWS profile
   Please choose the profile you want to use (Use arrow keys)
       # default
   Which app are you working on?
       # GettingStarted («id»)
   Choose your default editor:
       # «Choose your desired editor»
   ```

```
Choose the type of app that you're building …
    # «Choose your desired app type, and any subsequent configuration related to
 it»
Do you plan on modifying this backend?
    # N
```

(Optional) Clean up resources

When creating a backend for a test or a prototype, or just for learning purposes like this tutorial, you should delete the Cloud resources you created.

- Delete the project

  Open **Terminal**, navigate to your **project root folder**, and run the following **command**:

  ```
  amplify delete
  ```

Conclusion

You have built an iOS application using AWS Amplify! You have added authentication to your app allowing users to sign up, sign in, and manage their account. The app also has a scalable GraphQL API configured with an Amazon DynamoDB database which users can use to create and delete notes. You have also added file storage using Amazon S3, which users can use to upload images and view them in their app.

To conclude this guide, you can find instructions to reuse or delete the backend you have been using in this tutorial.

Congratulations!

You successfully built a web application on AWS! As a great next step, dive deeper into specific AWS technologies and take your application to the next level.