Swift OpenAPI Generator Essentials

Generating a client in a Swift package

This tutorial guides you through building *GreetingServiceClient*—an API client for a fictitious service that returns a personalized greeting.

```
% curl 'localhost:8080/api/greet?name=Jane'
{
  "message" : "Hello, Jane"
}
```

The API for the service is defined using OpenAPI and you'll create a Swift client for this service, from scratch!

Your Swift package will make use of the Swift OpenAPI Generator plugin to generate the code you'll use to call this API.

> **Tip**
>
> The hello-world-urlsession-client-example package contains the result of this tutorial, in case you're looking for a working example.

20mins

Estimated Time Swift 5.9

Section 1

(Optional) Downloading and running the server locally for testing

In the next section of the guide we will create a client for this service. In order to execute requests, you can download an example server implementation and run it locally.

Step 1

Clone the Git repository and change the current working directory to the nested example package.

console

```
% git clone https://github.com/apple/swift-openapi-generator
% cd swift-openapi-generator/Examples/hello-world-vapor-server-example

% swift run HelloWorldVaporServer
..
Build complete! (37.91s)
2023-12-12T09:06:32+0100 notice codes.vapor.application : [Vapor] Server starting

% curl 'localhost:8080/api/greet?name=Jane'
{
  "message" : "Hello, Jane"
}
```

Step 2

Build and run the service.

Step 3

While keeping the server running, in a separate Terminal window, test that the server is working by using `curl` from the command line.

Section 2

Creating a new Swift package

Create a brand-new Swift package for the client.

console

No Pre

```
% mkdir GreetingServiceClient
```

Step 1

Create a new directory.

Step 2

Create a new package using the Swift Package Manager CLI.

Step 3

Open the new package in Xcode using the `open` command.

Section 3

Configuring your project to use Swift OpenAPI Generator

Let's extend this sample package to call our `GreetingService` API.

Step 1

Copy in the OpenAPI document into the `Sources` directory.

Sources/openapi.yaml

No Pre

```yaml
openapi: '3.1.0'
info:
  title: GreetingService
  version: 1.0.0
servers:
  - url: https://example.com/api
    description: Example service deployment.
paths:
  /greet:
    get:
      operationId: getGreeting
      parameters:
        - name: name
          required: false
          in: query
          description: The name used in the returned greeting.
          schema:
            type: string
      responses:
        '200':
          description: A success response with a greeting.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Greeting'
components:
  schemas:
    Greeting:
      type: object
      properties:
        message:
          type: string
      required:
```

Step 2

If you launched a local server in the previous section, add a localhost server entry to the OpenAPI document.

This will make it easy to call the local server from generated code in the next section.

Step 3

We also need a config file that controls the behavior of the Swift OpenAPI Generator plugin. Create a file in the same directory called `openapi-generator-config.yaml`, with the following contents.

Step 4

Update the `Package.swift` by specifying the minimum platforms the package supports.

Step 5

Then, we'll add the package dependencies.

We added dependencies on the generator package plugin, which generates code at build time; the runtime package, which allows us to make use of the generated code; and, because Swift OpenAPI Generator has been built with an extensible transport abstraction, a concrete transport implementation library. Here we've selected the transport implementation that uses Foundation's URLSession.

Step 6

Now we can update our target to make use of the Swift OpenAPI Generator plugin.

Step 7

Finally, we need to declare the runtime dependencies for our target.

Step 8

Build the project now to ensure it's configured correctly.

The Swift OpenAPI Generator build plugin gets built and generates a client for the `Greeting Service` behind the scenes, making it available to use in the next section.

Section 4

Using the generated code in your target

Now we're ready to use the code that the plugin generated behind the scenes to fetch some personalized greetings!

main.swift

No Pre

```swift
// The Swift Programming Language
// https://docs.swift.org/swift-book

print("Hello, world!")
```

Step 1

Navigate to `Sources/main.swift` which is the entry point for our program.

Currently it just prints "Hello, world!" to the console. We'll make changes to this file to make use of the code that was generated by the plugin to call the `GreetingService` API.

Step 2

First we'll need to import our two runtime dependencies.

Step 3

Next we'll create an instance of our client.

Note: `Servers.Server2.url()` is the localhost service, defined in the OpenAPI document.

Step 4

Finally, we can use the client to make a request and print the response.

Step 5

If you have the example server running locally, you can now compile and run this executable and see the response in the console.

Section 5

Pattern matching on the response

Often we'd want to do a bit more than just print the high-level response value. For example, we might like to branch our code based on the response we received from the server.

Step 1

Add a `switch` statement to handle the different possible responses from the server.

Something's missing here, and if you recompile your package you'll see that the compiler helpfully tells you that your `switch` statement didn't cover all scenarios.

main.swift

No Pre

```swift
import OpenAPIRuntime
import OpenAPIURLSession

let client = Client(
    serverURL: try Servers.Server2.url(),
    transport: URLSessionTransport()
)

let response = try await client.getGreeting(query: .init(name: "CLI"))
switch response {
case .ok(let okResponse):
    print(okResponse)
}
```

Step 2

In the event the server provides a response that doesn't conform to the API specification, you still have an opportunity as a client to handle it gracefully. We'll do so by printing a helpful message, indicating that our client doesn't know what to do with this because it hasn't been updated to handle this kind of response.

Everything should now compile again.

Step 3

Finally, let's extract and print the content from the response body.

The `switch` statement over the body allows you to handle the different content types that are specified for the API operation.

Step 4

You can now compile and run the executable again and see the response in the console.

Section 6

Unwrapping the response using the shorthand API

If you don’t need to handle all the response codes and content types, you can also use the shorthand API on the response and body enums, providing you with throwing getters for unwrapping each value.

These conveniences will throw an error if the received response or content type doesn’t match the one you’re requesting.

Step 1

Remove the switch statement and replace it with chained access to the `ok` response, to its `body`, to the `json` content type of the body, and finally to the `message` property of the received greeting.

main.swift

No Pre

```swift
import OpenAPIRuntime
import OpenAPIURLSession

let client = Client(
    serverURL: try Servers.Server2.url(),
    transport: URLSessionTransport()
)

let response = try await client.getGreeting(query: .init(name: "CLI"))
print(try response.ok.body.json.message)
```

Step 2

You can now compile and run the executable again and see the response in the console.

Next

Generating a client in an Xcode project

This tutorial guides you through building *GreetingServiceClient*—an API client for a fictitious service that returns a personalized greeting.

Get started