AI VOID FIELD GUIDE

Step-by-Step Tutorials

Focused, practical tutorials that teach you how to accomplish specific tasks step by step. Learn by doing with clear instructions and working code examples.

AI VOID

2026-05-11 20:32 IST

Contents

CHAPTER 01

Generate Kotlin Clients with Smithy

What you'll build: A type-safe Kotlin client automatically generated from a custom Smithy service model, demonstrating its usage. **Time needed:** ~60 minutes **Prerequisites:** Basic understanding of Kotlin, Familiarity with Gradle build system, Java Development Kit (JDK) 11 or higher installed, Basic understanding of API concepts **Version used:** Smithy 2.0

Introduction to Smithy Kotlin Client Code Generation

Building robust and maintainable API clients can be a tedious and error-prone task. Manually writing data transfer objects (DTOs), request/response structures, and client methods for every API endpoint leads to boilerplate code, potential inconsistencies, and a higher chance of errors when the API changes. This is where Interface Definition Languages (IDLs) like Smithy come to the rescue.

Smithy is a powerful, protocol-agnostic IDL that allows you to define your API's structure and behavior in a single, language-agnostic model. Once defined, Smithy's code generation capabilities can automatically produce clients, servers, documentation, and more for various programming languages. For Kotlin developers, Smithy offers a robust solution to generate type-safe, idiomatic Kotlin clients, significantly reducing development effort and improving API interaction reliability.

In this tutorial, we'll walk through the process of defining a simple Smithy service model and then use the Smithy Kotlin code generator to produce a fully functional Kotlin client. You'll learn how to integrate Smithy into your Gradle build, generate the client, and finally, use it within a Kotlin application. By the end, you'll have a clear understanding of how to leverage Smithy to streamline your API client development workflow.

Understanding Smithy: The Interface Definition Language

Before we dive into code generation, let's understand what Smithy is and why it's so beneficial.

What is Smithy? Smithy is an **Interface Definition Language (IDL)** developed by Amazon Web Services (AWS). It provides a structured, language-agnostic way to define API services, including their data structures, operations, and protocols. Think of it as a blueprint for your API. Instead of writing separate specifications for REST, gRPC, or GraphQL, you define your API once in Smithy.

Why does it exist? What problem does it solve? The core problem Smithy solves is the fragmentation and inconsistency that often arises when building APIs. Without a unified definition, developers might:

- Manually write API specifications in different formats (OpenAPI, WSDL, etc.).
- Implement clients and servers in various languages, leading to duplicated effort and potential mismatches.
- Struggle to keep documentation, code, and tests synchronized with API changes.

Smithy addresses these issues by:

1. **Single Source of Truth:** Your Smithy model becomes the definitive source for your API.
2. **Language Agnostic:** It decouples API definition from specific programming languages or protocols.
3. **Code Generation:** It enables automatic generation of code (clients, servers, documentation, test stubs) for multiple languages (Java, Kotlin, TypeScript, etc.) from that single model.
4. **Protocol Agnostic:** Smithy models can be bound to various wire protocols (like HTTP/JSON, AWS's custom protocols, or even custom binary protocols) using "traits." This means the same model can be used for different communication styles.

Core Concepts in Smithy

Smithy models are composed of several key elements:

- **Shapes:** These are the fundamental building blocks. Shapes define the data types and operations of your service. Common shape types include:
  - `structure`: A composite type, similar to a class or data class, with named members.
  - `string`, `integer`, `boolean`, `timestamp`: Primitive types.
  - `list`, `map`, `set`: Collection types.
  - `union`: A type that can hold one of several possible member types.
  - `operation`: Defines an API call, including its input, output, and potential errors.
  - `service`: The top-level shape that aggregates operations and resources, representing your entire API.
- **Members:** Fields within a `structure`, `union`, `list`, or `map`.
- **Traits:** These are metadata that can be applied to shapes or members to provide additional information or constraints. Traits influence how code is generated and how the API behaves. Examples include `@required`, `@httpLabel`, `@documentation`, `@error`, or `@deprecated`.

Example: Imagine a simple `UserService` that allows you to get user details. In Smithy, it might look something like this:

```
$version: "2.0"

namespace com.example.userservice

/// Represents a user profile.
structure User {
    @required
    userId: String,

    @required
    username: String,

    email: String,
}

/// Input for the GetUser operation.
structure GetUserInput {
    @required
    @httpLabel
    userId: String,
}

/// Output for the GetUser operation.
structure GetUserOutput {
    @required
```

```
    user: User,
}

/// Defines an error when a user is not found.
@error("client")
structure UserNotFoundException {
    message: String,
}

/// A service for managing users.
service UserService {
    version: "1.0",
    operations: [GetUser]
}

/// Retrieves details for a specific user.
@http(method: "GET", uri: "/users/{userId}")
@readonly
operation GetUser {
    input: GetUserInput,
    output: GetUserOutput,
    errors: [UserNotFoundException]
}
```

This model defines a `User` structure, input/output for a `GetUser` operation, an error type, and the `UserService` itself. The `@http` trait on `GetUser` tells Smithy how this operation maps to an HTTP GET request, including the URI path parameter.

📌 **Key Idea:** Smithy provides a centralized, language-agnostic blueprint for your APIs, enabling consistent and automated code generation across different platforms and protocols.

How Smithy Kotlin Client Code Generation Works Internally

Understanding the internal flow of Smithy Kotlin client code generation helps demystify the process and makes troubleshooting easier. It's not magic; it's a well-defined series of steps.

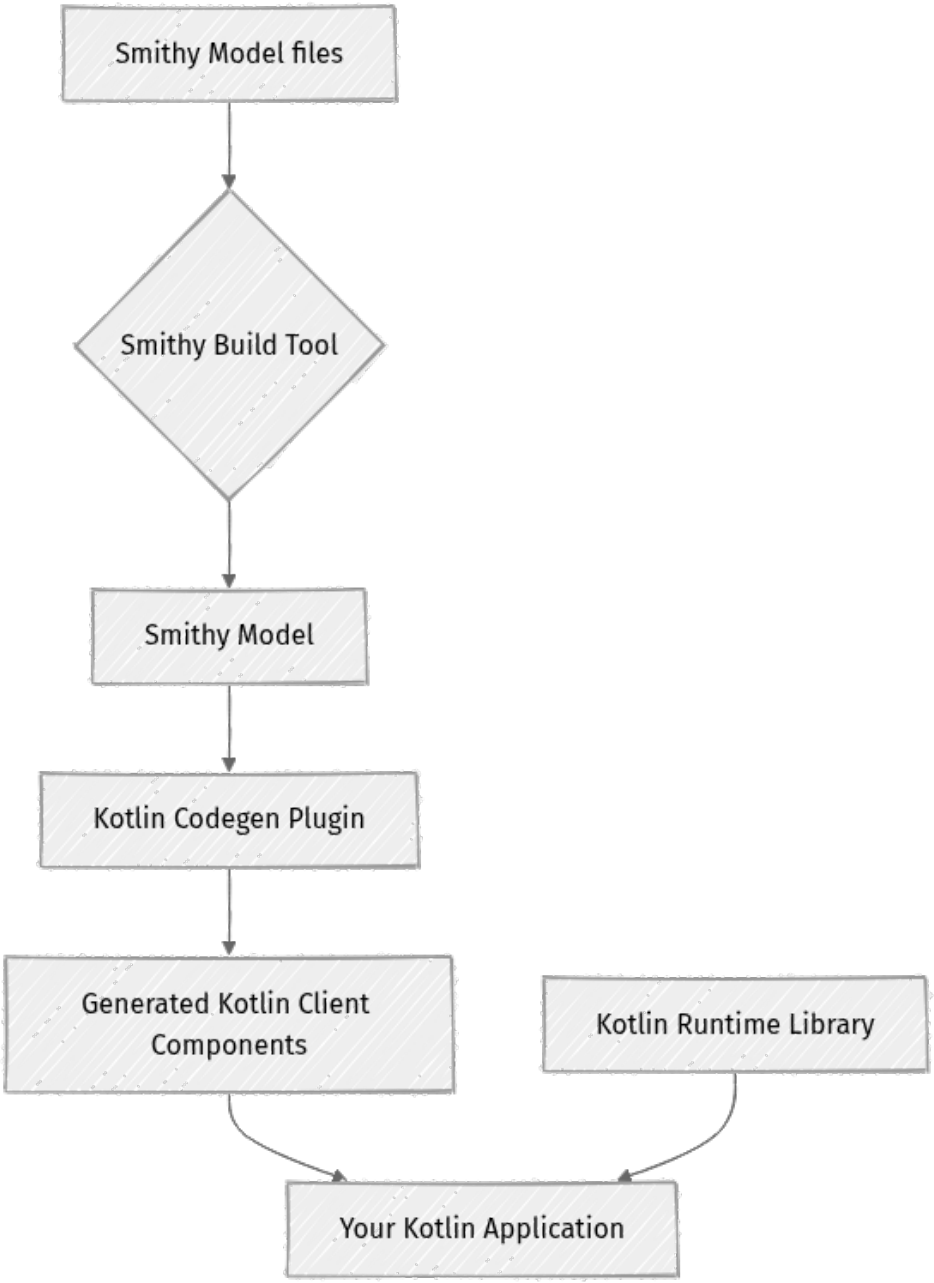
Here's a simplified breakdown of how Smithy transforms your API model into functional Kotlin client code:

1. **Smithy Model Definition:** You start by defining your API using Smithy's IDL, typically in `.smithy` files. This model describes your service, operations, data structures (shapes), and any associated metadata (traits).

2. **Smithy Build Process:** When you invoke the Smithy build tool (usually via a Gradle plugin), it takes your `.smithy` files and compiles them into an **Abstract Syntax Tree (AST)** or an in-memory model representation. This model is then validated for correctness and consistency against Smithy's rules.
3. **Kotlin Code Generation Plugin (`kotlin-codegen`):** The core of the client generation for Kotlin is the `kotlin-codegen` plugin. This plugin is specifically designed to interpret the Smithy model and translate its concepts into idiomatic Kotlin constructs.
4. **Language-Specific Mapping:** The `kotlin-codegen` plugin maps Smithy shapes and traits to corresponding Kotlin types:
    - `service` shapes become Kotlin client interfaces and implementations.
    - `structure` shapes become Kotlin `data class`es.
    - `operation` shapes translate into suspend functions within the client interface, with specific request and response `data class`es.
    - `union` shapes become Kotlin `sealed class`es.
    - `list` and `map` shapes map to Kotlin `List<T>` and `Map<K, V>`.
    - `string`, `integer`, `boolean`, `timestamp` map to their Kotlin primitive equivalents (`String`, `Int`/`Long`, `Boolean`, `Instant`/`LocalDateTime`).
    - **Traits** influence the generated code. For example, `@required` might lead to non-nullable types, `@http` traits guide the generation of HTTP-specific serialization/deserialization logic.
5. **Protocol-Specific Implementation:** The generated client code includes not just the data structures and interfaces, but also the underlying logic for:
    - **Serialization:** Converting Kotlin request objects into the wire format (e.g., JSON, XML) as defined by the protocol binding (e.g., `@http` traits).
    - **Deserialization:** Parsing the wire format response back into Kotlin response objects.
    - **HTTP Request Execution:** Handling the actual network calls, including setting headers, URI construction, and error handling based on the Smithy model's protocol definitions.

6. **Output:** The `kotlin-codegen` plugin writes the generated Kotlin source files to a specified output directory (e.g., `build/smithy/source/kotlin`). These files are then compiled along with your handwritten application code.

Here's a simple flowchart illustrating this process:



> ⚡ **Note:** The "Kotlin Runtime Library" (represented by `smithy-kotlin-runtime`) is a crucial dependency. It provides the foundational components like HTTP client abstractions, serialization utilities, and common types that the generated client code relies on. You don't write this; you just include it as a dependency.

This systematic approach ensures that your client always matches your API's definition, reducing errors and making API evolution much smoother.

Prerequisites and Initial Project Setup

Before we start writing any Smithy models or generating code, let's ensure your development environment is ready and set up a basic Gradle project.

1. Verify Java Development Kit (JDK) Installation Smithy and Kotlin development requires a JDK. We'll use JDK 11 or higher.

Open your terminal or command prompt and run:

```
java -version
```

You should see output indicating JDK 11 or a newer version (e.g., 17, 21).

```
openjdk version "17.0.7" 2023-04-18
OpenJDK Runtime Environment (build 17.0.7+7-LTS)
OpenJDK 64-Bit Server VM (build 17.0.7+7-LTS, mixed mode, sharing)
```

If you don't have JDK 11+ installed, please install it using your preferred method (SDKMAN!, Homebrew, official Oracle/OpenJDK downloads).

2. Create a New Gradle Project

We'll use Gradle to manage our project and integrate the Smithy code generation plugin.

First, create a new directory for your project:

```
mkdir smithy-kotlin-client-tutorial
cd smithy-kotlin-client-tutorial
```

Now, initialize a new Kotlin application project using Gradle. We'll use the Kotlin DSL for our build scripts.

```
gradle init --type kotlin-application --dsl kotlin
```

Gradle will ask you a few questions:

- `Select build script DSL:` Choose `2` for Kotlin.
- `Select test framework:` Choose `1` for JUnit Jupiter.

- `Project name:` You can press Enter to accept `smithy-kotlin-client-tutorial`.
- `Source package:` You can press Enter to accept `smithy.kotlin.client.tutorial`.

After initialization, your project structure should look something like this:

```
smithy-kotlin-client-tutorial/
├── gradle/
│   └── wrapper/
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle.kts
├── build.gradle.kts
└── src/
    └── main/
        ├── kotlin/
        │   └── smithy/kotlin/client/tutorial/
        │       └── App.kt
        └── resources/
    └── test/
        └── kotlin/
            └── smithy/kotlin/client/tutorial/
                └── AppTest.kt
```

3. Review Initial `build.gradle.kts`

Open the `build.gradle.kts` file in the root of your project. It should look similar to this:

```
plugins {
    // Apply the application plugin to add support for building a CLI
application in Java.
    application
    // Apply the Kotlin JVM plugin to add support for Kotlin on the JVM.
    kotlin("jvm") version "1.9.22" // Adjust Kotlin version as needed
}

repositories {
    // Use Maven Central for dependencies.
    mavenCentral()
}

dependencies {
    // Align versions of all Kotlin components
    implementation(platform("org.jetbrains.kotlin:kotlin-bom"))

    // Use the Kotlin JDK 8 standard library.
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")

    // Use the JUnit Jupiter API for testing.
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.10.0")
```

```
    // Use the JUnit Jupiter Engine for running tests.
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.10.0")
}

// Apply a specific Java toolchain to ease working on different environments.
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11) // Ensure JDK 11 is used
    }
}

application {
    // Define the main class for the application.
    mainClass.set("smithy.kotlin.client.tutorial.AppKt")
}

tasks.test {
    // Use JUnit Platform for testing.
    useJUnitPlatform()
}
```

> ⚡ **Note:** The `kotlin("jvm") version` might be slightly different depending on when you run `gradle init`. Ensure it's a recent stable version. Also, confirm `languageVersion = JavaLanguageVersion.of(11)` is set under the `java` block to explicitly target JDK 11.

4. Verify Project Setup

Run a quick build to ensure everything is set up correctly:

```
./gradlew build
```

You should see `BUILD SUCCESSFUL` at the end of the output. This confirms your basic Kotlin project is ready.

What you accomplished: You've set up a new Gradle project for a Kotlin application and verified that your JDK is correctly configured, laying the groundwork for integrating Smithy.

Defining Your First Smithy Service Model

Now that our project is set up, let's define a simple API service using Smithy. This will be the blueprint from which our Kotlin client will be generated. We'll create a service that manages simple "widgets."

1. Create the Smithy Model Directory

Smithy models are typically placed in `src/main/smithy`. Create this directory within your project:

```
mkdir -p src/main/smithy
```

2. Define Your Smithy Model File

Inside `src/main/smithy`, create a new file named `widgets.smithy`. This file will contain our service definition.

```
touch src/main/smithy/widgets.smithy
```

Now, open `src/main/smithy/widgets.smithy` and add the following content:

```
$version: "2.0"

namespace com.example.widgets

/// A basic widget managed by the service.
structure Widget {
    @required
    id: String,

    @required
    name: String,

    description: String,

    @required
    price: BigDecimal,
}

/// Input for the CreateWidget operation.
structure CreateWidgetInput {
    @required
    name: String,

    description: String,

    @required
    price: BigDecimal,
}

/// Output for the CreateWidget operation.
structure CreateWidgetOutput {
    @required
    widget: Widget,
}

/// Input for the GetWidget operation.
structure GetWidgetInput {
    @required
    @httpLabel
    id: String,
```

```
}

/// Output for the GetWidget operation.
structure GetWidgetOutput {
    @required
    widget: Widget,
}

/// Defines an error when a widget is not found.
@error("client")
structure WidgetNotFoundException {
    message: String,
}

/// The service for managing widgets.
service WidgetService {
    version: "1.0",
    operations: [CreateWidget, GetWidget]
}

/// Creates a new widget.
@http(method: "POST", uri: "/widgets")
operation CreateWidget {
    input: CreateWidgetInput,
    output: CreateWidgetOutput,
}

/// Retrieves a widget by its ID.
@http(method: "GET", uri: "/widgets/{id}")
@readonly
operation GetWidget {
    input: GetWidgetInput,
    output: GetWidgetOutput,
    errors: [WidgetNotFoundException]
}
```

Let's break down what we've defined:

- `$version: "2.0"`: Specifies the Smithy IDL version.
- `namespace com.example.widgets`: Defines the namespace for our shapes, which will typically map to a Kotlin package name.
- `structure Widget`: A data structure representing a widget, with fields like `id`, `name`, `description`, and `price`.
    - `@required`: A trait indicating that `id`, `name`, and `price` are mandatory fields.
    - `BigDecimal`: A Smithy primitive type for high-precision decimal numbers, which will map to Kotlin's `BigDecimal`.

- `CreateWidgetInput`, `CreateWidgetOutput`, `GetWidgetInput`, `GetWidgetOutput`: These structures define the specific data payloads for our API operations.
    - `@httpLabel`: On `GetWidgetInput.id`, this trait indicates that the `id` field should be extracted from the URI path (`/widgets/{id}`).
- `WidgetNotFoundException`: An error structure.
    - `@error("client")`: A trait indicating this is a client-side error (e.g., a 4xx HTTP status code).
- `service WidgetService`: Our top-level API service.
    - `version: "1.0"`: The version of our service.
    - `operations: [CreateWidget, GetWidget]`: Lists the operations exposed by this service.
- `operation CreateWidget` and `operation GetWidget`: These define the actual API calls.
    - `@http(...)`: These traits bind the operations to specific HTTP methods and URIs. This is how Smithy knows how to generate HTTP-aware client code.
    - `@readonly`: A trait indicating that the `GetWidget` operation does not modify server state.

> ⚡ **Note:** Smithy models are declarative. You describe what your API looks like, not how it's implemented. The code generator takes care of the "how."

What you accomplished: You've successfully defined your first Smithy service model, including data structures, operations, and HTTP bindings, which will serve as the input for client code generation.

Configuring Gradle for Smithy Kotlin Code Generation

Now that we have our Smithy model, the next crucial step is to configure Gradle to recognize it and use the Smithy Kotlin code generation plugin. This involves adding the necessary plugins and dependencies to your `build.gradle.kts` file.

1. Add Smithy Gradle Plugin and Kotlin Codegen Plugin

Open your `build.gradle.kts` file and locate the `plugins` block. Add the `software.amazon.smithy.gradle.smithy-gradle-plugin` and `software.amazon.smithy.kotlin-codegen` plugins.

```
// build.gradle.kts
plugins {
    application
    kotlin("jvm") version "1.9.22" // Ensure this is a recent Kotlin version

    // Add Smithy Gradle Plugin
    id("software.amazon.smithy.gradle.smithy-gradle-plugin") version "0.6.0" /
/ Use Smithy 2.0 compatible version
    // Add Smithy Kotlin Codegen Plugin
    id("software.amazon.smithy.kotlin-codegen") version "0.33.1-beta" // Use
the latest stable beta or GA version
}
```

> ⚠️ **Common mistake:** Ensure the versions of `smithy-gradle-plugin` and `smithy-kotlin-codegen` are compatible and up-to-date. Using outdated versions can lead to build failures or unexpected generated code. The `0.6.0` for the Gradle plugin is for Smithy 2.0. The `0.33.1-beta` is a recent beta for Kotlin client generation. Always check the official documentation for the latest recommended versions.

2. Add Smithy and Kotlin Runtime Dependencies

Next, we need to add the dependencies that the Smithy build process and the generated Kotlin client code will rely on. These include the Smithy CLI (for parsing the model) and the Smithy Kotlin runtime library.

Add the following to your `dependencies` block in `build.gradle.kts`:

```
// build.gradle.kts
dependencies {
    // ... existing dependencies

    // Dependencies for Smithy build process
    smithyCli("software.amazon.smithy:smithy-cli:1.37.0") // Use Smithy CLI
2.0 compatible version
    smithyBuild("software.amazon.smithy:smithy-model:1.37.0")
    smithyBuild("software.amazon.smithy:smithy-protocol-traits:1.37.0")

    // Dependencies required by the generated Kotlin client
    implementation("aws.sdk.kotlin:http-client-engine-okhttp:0.33.1-beta") //
Example HTTP engine
    implementation("aws.sdk.kotlin:runtime:0.33.1-beta") // Smithy Kotlin
runtime
    implementation("aws.sdk.kotlin:aws-json-protocols:0.33.1-beta") // For
JSON protocol support
}
```

> ⚡ **Note:** The `smithyCli` and `smithyBuild` configurations are specific to the `smithy-gradle-plugin`. They tell Gradle which Smithy core libraries to use during the model compilation phase. The `implementation` dependencies are what your generated client and your application will need at runtime. `http-client-engine-okhttp` is an example of an HTTP engine; you could use Ktor or another one. `aws-json-protocols` is needed because our `@http` traits imply a JSON payload by default for non-GET operations.

3. Configure the Smithy Build Process

Finally, we need to tell the Smithy Gradle plugin how to find our Smithy models and what code to generate. This is done within a `smithy` block in `build.gradle.kts`.

Add the `smithy` block to your `build.gradle.kts`, typically after the `dependencies` block:

```
// build.gradle.kts
// ... after dependencies block

smithy {
    // Define where your Smithy models are located.
    // By default, it looks in src/main/smithy, but explicitly defining is
good practice.
    // sources = setOf("src/main/smithy") // Not strictly needed if using
default, but shown for clarity

    // Define projections for code generation.
    // A projection specifies a subset of the Smithy model and the generators
to apply.
    projections {
        create("client") { // We're creating a projection named "client"
            // The model files to include in this projection.
            // By default, it includes all .smithy files in sources.
            // models = setOf("src/main/smithy/widgets.smithy") // Explicitly
include our model

            // Configure the Kotlin code generator for this projection.
            codegen {
                // The service shape to generate a client for.
                // This MUST match the fully qualified name of your service
shape.
                service("com.example.widgets#WidgetService")
                // The language to generate (Kotlin in this case).
                language("kotlin")
                // The target package for the generated code.
                rootPackage("com.example.widgets.client")
            }
        }
    }
}
```

Let's break down the `smithy` block:

- `projections`: A projection defines a specific view of your Smithy model and the code generators to apply to it. You can have multiple projections (e.g., one for a client, one for a server).
- `create("client")`: We're creating a projection named "client."
- `codegen`: This block configures the code generator for this projection.
    - `service("com.example.widgets#WidgetService")`: This is crucial. It tells the generator which specific service from your Smithy model (`widgets.smithy`) to generate a client for. The format is `namespace#ShapeName`.
    - `language("kotlin")`: Specifies that we want to generate Kotlin code.
    - `rootPackage("com.example.widgets.client")`: This sets the base package for all the generated Kotlin files.

4. Run the Smithy Build Task

Now, with everything configured, you can trigger the Smithy code generation. The `smithy-gradle-plugin` adds several tasks, including `smithyBuild` and `generateSmithy`.

Run the build task:

```
./gradlew smithyBuild
```

Or, if you just want to generate code without a full project build:

```
./gradlew generateSmithy
```

If successful, you should see `BUILD SUCCESSFUL` and output indicating that Smithy has processed your model. More importantly, it will have created a new directory structure for the generated code.

> ⚠️ **Common mistake:** If `smithyBuild` fails, check:

- Typos in your `build.gradle.kts` (plugin IDs, versions, `service` name).

- Typos or syntax errors in your `widgets.smithy` file.

- Missing `smithyBuild` or `smithyCli` dependencies.

- Incorrect `service` name in the `codegen` block (must be `namespace#ShapeName`).

What you accomplished: You've successfully configured your Gradle project to use the Smithy Gradle plugin and Kotlin code generator, and you've initiated the code generation process from your `widgets.smithy` model.

Generating and Exploring the Kotlin Client

After successfully running `./gradlew smithyBuild` or `./gradlew generateSmithy`, the Smithy Kotlin code generator has done its work! Let's explore the fruits of its labor.

1. Locating the Generated Code

The generated Kotlin client code is placed in a specific directory within your project's `build` folder. By default, for a projection named "client", it will be located at:

`build/smithy/source/kotlin/client`

Navigate to this directory in your terminal:

```
ls -R build/smithy/source/kotlin/client
```

You should see a directory structure mirroring your `rootPackage` configuration (`com/example/widgets/client`) and within it, several Kotlin files.

```
build/smithy/source/kotlin/client/com/example/widgets/client/
├── model/
│   ├── CreateWidgetInput.kt
│   ├── CreateWidgetOutput.kt
│   ├── GetWidgetInput.kt
│   ├── GetWidgetOutput.kt
│   ├── Widget.kt
│   └── WidgetNotFoundException.kt
└── WidgetServiceClient.kt
```

2. Exploring the Generated Files

Let's open and examine some of these files to see how Smithy translates your model into Kotlin.

- **`model/Widget.kt`**: This file contains the `Widget` data class, directly derived from your `Widget` structure in `widgets.smithy`.

```kotlin
// build/smithy/source/kotlin/client/com/example/widgets/client/model/
Widget.kt
package com.example.widgets.client.model

import java.math.BigDecimal
import kotlin.jvm.JvmInline
import kotlinx.serialization.Serializable

@Serializable
data class Widget(
    /**
     * The widget's unique identifier.
     */
    val id: String,
    /**
     * The name of the widget.
     */
    val name: String,
    /**
     * A description of the widget.
     */
    val description: String? = null, // Optional because it wasn't
@required
    /**
     * The price of the widget.
     */
    val price: BigDecimal
)
```

  Notice how `@required` fields become non-nullable Kotlin properties, and optional fields (`description`) become nullable. The `BigDecimal` Smithy type correctly maps to `java.math.BigDecimal`. The `Serializable` annotation is added for JSON serialization.

- **`model/CreateWidgetInput.kt`**: This is another data class for the input of our `CreateWidget` operation.

```
// build/smithy/source/kotlin/client/com/example/widgets/client/model/
CreateWidgetInput.kt
package com.example.widgets.client.model

import java.math.BigDecimal
import kotlin.jvm.JvmInline
import kotlinx.serialization.Serializable

@Serializable
data class CreateWidgetInput(
    /**
     * The name of the widget.
     */
    val name: String,
    /**
     * A description of the widget.
     */
    val description: String? = null,
    /**
     * The price of the widget.
     */
    val price: BigDecimal
)
```

  Similar to `Widget`, but only contains the fields defined in `CreateWidgetInput`.

- **`model/WidgetNotFoundException.kt`**: Error shapes become Kotlin exception classes.

```
// build/smithy/source/kotlin/client/com/example/widgets/client/model/
WidgetNotFoundException.kt
package com.example.widgets.client.model

import aws.sdk.kotlin.runtime.ServiceException
import aws.smithy.kotlin.runtime.http.response.HttpResponse
import kotlin.String

/**
 * Defines an error when a widget is not found.
 */
class WidgetNotFoundException private constructor(builder: Builder) :
    ServiceException(builder.message) {
    override val errorDetails: ErrorDetails = builder.errorDetails ?: Err
orDetails()

    // ... constructor and builder boilerplate ...

    companion object {
        operator fun invoke(block: Builder.() -> Unit = {}): WidgetNotFou
ndException =
            Builder().apply(block).build()
    }

    class Builder : ServiceException.Builder() {
        override var message: String? = null
        override var errorDetails: ErrorDetails? = null

        override fun build(): WidgetNotFoundException = WidgetNotFoundExc
eption(this)
    }
}
```

It extends `ServiceException` from the Smithy Kotlin runtime, providing a structured way to handle API-specific errors.

- **`WidgetServiceClient.kt`**: This is the core client interface and its implementation. It defines the operations you can call.

```
// build/smithy/source/kotlin/client/com/example/widgets/client/
WidgetServiceClient.kt
package com.example.widgets.client

import aws.sdk.kotlin.runtime.client.SdkClient
import aws.sdk.kotlin.runtime.client.SdkClientConfig
import aws.sdk.kotlin.runtime.http.operation.SdkHttpOperation
import aws.sdk.kotlin.runtime.http.operation.create
import aws.sdk.kotlin.runtime.http.response.HttpResponse
import aws.smithy.kotlin.runtime.client.SdkDsl
import aws.smithy.kotlin.runtime.http.engine.HttpClientEngine
import aws.smithy.kotlin.runtime.http.engine.HttpClientEngineConfig
import aws.smithy.kotlin.runtime.http.operation.CustomOperationDeserializ
er
import aws.smithy.kotlin.runtime.http.operation.SdkHttpOperation.Companio
n.invoke
import aws.smithy.kotlin.runtime.http.response.set
import aws.smithy.kotlin.runtime.operation.OperationExecution
import aws.smithy.kotlin.runtime.serde.SdkField
import aws.smithy.kotlin.runtime.serde.SdkObjectDescriptor
import aws.smithy.kotlin.runtime.serde.SerialKind
import aws.smithy.kotlin.runtime.serde.Serializable
import aws.smithy.kotlin.runtime.serde.Serializer
import aws.smithy.kotlin.runtime.serde.json.JsonDeserializer
import aws.smithy.kotlin.runtime.serde.json.JsonEncoder
import aws.smithy.kotlin.runtime.serde.json.JsonSerdeProvider
import aws.smithy.kotlin.runtime.serde.json.JsonSerializer
import aws.smithy.kotlin.runtime.serde.json.trait.JsonShapeIdTrait
import aws.smithy.kotlin.runtime.time.Instant
import com.example.widgets.client.model.CreateWidgetInput
import com.example.widgets.client.model.CreateWidgetOutput
import com.example.widgets.client.model.GetWidgetInput
import com.example.widgets.client.model.GetWidgetOutput
import com.example.widgets.client.model.WidgetNotFoundException
import kotlin.Any
import kotlin.Exception
import kotlin.String
import kotlin.Suppress
import kotlin.Unit
import kotlin.collections.List
import kotlin.collections.Map
import kotlin.collections.Set
import kotlin.jvm.JvmInline
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.flowOf

/**
 * The service for managing widgets.
 */
interface WidgetServiceClient : SdkClient {
    /**
     * Creates a new widget.
     */
    suspend fun createWidget(input: CreateWidgetInput): CreateWidgetOutpu
t

    /**
```

```
     * Retrieves a widget by its ID.
     */
    suspend fun getWidget(input: GetWidgetInput): GetWidgetOutput

    companion object : SdkClient.CompanionObject<WidgetService, WidgetSer
viceClientConfig.Builder, WidgetServiceClient> {
        override fun build(block: WidgetServiceClientConfig.Builder.()
-> Unit): WidgetServiceClient =
            WidgetServiceClientImpl(WidgetServiceClientConfig.Builder().a
pply(block).build())
    }
}

// ... actual implementation details will follow below the interface ...
// This implementation will contain the HTTP request/response handling
logic
// based on the @http traits in your Smithy model.
```

Here, you'll find:

- An interface `WidgetServiceClient` with suspend functions for `createWidget` and `getWidget`. These functions take the generated input data classes and return the generated output data classes.
- A `companion object` with a `build` function, which is the idiomatic way to construct the client.
- Further down (not fully shown above), there will be an implementation class (`WidgetServiceClientImpl`) that handles the actual HTTP calls, serialization, deserialization, and error mapping, all based on the Smithy model's protocol definitions.

This exploration confirms that Smithy has successfully translated our declarative API model into concrete, type-safe Kotlin code. This generated code is ready to be used in your application.

What you accomplished: You've located and inspected the automatically generated Kotlin client code, understanding how Smithy shapes and operations are mapped to Kotlin data classes, interfaces, and functions.

Using the Generated Client in Your Application

Now that we have our type-safe Kotlin client generated, let's put it to use in a simple application. We'll modify our `App.kt` file to demonstrate how to instantiate the client and call its operations.

1. Update `App.kt`

Open `src/main/kotlin/smithy/kotlin/client/tutorial/App.kt`. Replace its content with the following:

```
package smithy.kotlin.client.tutorial

import aws.sdk.kotlin.runtime.http.engine.HttpClientEngine
import aws.sdk.kotlin.runtime.http.engine.HttpClientEngineBase
import aws.sdk.kotlin.runtime.http.request.HttpRequest
import aws.sdk.kotlin.runtime.http.response.HttpResponse
import aws.smithy.kotlin.runtime.ClientException
import aws.smithy.kotlin.runtime.content.ByteStream
import aws.smithy.kotlin.runtime.http.HttpStatusCode
import aws.smithy.kotlin.runtime.http.response.complete
import aws.smithy.kotlin.runtime.time.Instant
import com.example.widgets.client.WidgetServiceClient
import com.example.widgets.client.WidgetServiceClientConfig
import com.example.widgets.client.model.CreateWidgetInput
import com.example.widgets.client.model.GetWidgetInput
import com.example.widgets.client.model.Widget
import com.example.widgets.client.model.WidgetNotFoundException
import java.math.BigDecimal
import kotlinx.coroutines.runBlocking

/**
 * A simple HTTP client engine that mocks responses for our tutorial.
 * In a real application, you'd use a real engine like OkHttpEngine or
KtorEngine.
 */
class MockHttpClientEngine : HttpClientEngineBase("MockHttpClientEngine") {
    override suspend fun roundTrip(request: HttpRequest): HttpResponse {
        println("MockHttpClientEngine received request: $
{request.method.name} ${request.url.path}")

        val path = request.url.path
        val method = request.method.name

        if (method == "POST" && path == "/widgets") {
            // Simulate CreateWidget response
            val newWidget = Widget(
                id = "widget-123",
                name = "Test Widget",
                description = "A widget created by the mock client.",
                price = BigDecimal("99.99")
            )
            val body = """
                {"widget": {"id": "${newWidget.id}", "name": "$
{newWidget.name}", "description": "${newWidget.description}", "price": ${newWi
dget.price}}}
            """.trimIndent()
            return HttpResponse(
                status = HttpStatusCode.Created,
                headers = buildMap { put("Content-Type", "application/json") }
,
                body = ByteStream.fromString(body)
            )
        } else if (method == "GET" && path == "/widgets/widget-456") {
            // Simulate GetWidget response for a known ID
            val existingWidget = Widget(
                id = "widget-456",
                name = "Existing Widget",
                description = "This widget already exists.",
                price = BigDecimal("123.45")
            )
```

```
            val body = """
                {"widget": {"id": "${existingWidget.id}", "name": "${existingW
idget.name}", "description": "${existingWidget.description}", "price": ${exist
ingWidget.price}}}
            """.trimIndent()
            return HttpResponse(
                status = HttpStatusCode.OK,
                headers = buildMap { put("Content-Type", "application/json") }
,
                body = ByteStream.fromString(body)
            )
        } else if (method == "GET" && path.startsWith("/widgets/")) {
            // Simulate WidgetNotFoundException for unknown IDs
            val body = """{"message": "Widget not found: ${path.substringAfter
Last("/")}"}"""
            return HttpResponse(
                status = HttpStatusCode.NotFound,
                headers = buildMap { put("Content-Type", "application/json") }
,
                body = ByteStream.fromString(body)
            )
        }

        return HttpResponse(
            status = HttpStatusCode.InternalServerError,
            headers = buildMap { put("Content-Type", "text/plain") },
            body = ByteStream.fromString("MockHttpClientEngine: Unhandled
request")
        ).also { it.complete() } // Ensure the response body is consumed
    }
}

fun main() = runBlocking {
    println("Starting Smithy Kotlin Client Tutorial Application...")

    // 1. Configure and instantiate the generated client
    val client = WidgetServiceClient {
        // In a real application, you'd configure the endpoint and a real HTTP
client engine
        // endpointUrl = "http://localhost:8080"
        httpClientEngine =
MockHttpClientEngine() // Using our mock engine for this tutorial
    }

    try {
        // 2. Call the CreateWidget operation
        println("\n--- Calling CreateWidget ---")
        val createInput = CreateWidgetInput(
            name = "My New Widget",
            description = "A widget created via the generated client.",
            price = BigDecimal("25.50")
        )
        val createdWidgetOutput = client.createWidget(createInput)
        println("Created Widget: ${createdWidgetOutput.widget.id} - ${createdW
idgetOutput.widget.name} (Price: ${createdWidgetOutput.widget.price})")

        // 3. Call the GetWidget operation for an existing widget (mocked)
        println("\n--- Calling GetWidget for known ID ---")
        val getExistingInput = GetWidgetInput(id = "widget-456")
        val existingWidgetOutput = client.getWidget(getExistingInput)
        println("Retrieved Existing Widget: ${existingWidgetOutput.widget.id}
- ${existingWidgetOutput.widget.name} (Price: ${existingWidgetOutput.widget.pr
```

```
ice})")

        // 4. Call the GetWidget operation for a non-existent widget (mocked
to throw error)
        println("\n--- Calling GetWidget for non-existent ID ---")
        try {
            val getNonExistentInput = GetWidgetInput(id = "non-existent-123")
            client.getWidget(getNonExistentInput)
        } catch (e: WidgetNotFoundException) {
            println("Caught expected error: ${e.message}")
        } catch (e: ClientException) {
            println("Caught unexpected client exception: ${e.message}")
        }

    } catch (e: Exception) {
        println("An unexpected error occurred: ${e.message}")
        e.printStackTrace()
    } finally {
        // Always close the client when done to release resources
        client.close()
        println("\nApplication finished.")
    }
}
```

Let's break down the changes:

- **`MockHttpClientEngine`**: Since we don't have a real backend server running, we've created a `MockHttpClientEngine`. This class extends `HttpClientEngineBase` from the Smithy Kotlin runtime and overrides the `roundTrip` method to simulate HTTP responses based on the request path and method. It returns predefined JSON bodies for our `CreateWidget` and `GetWidget` operations, and a `WidgetNotFoundException` for an unknown ID.

- **`main` function**:
  - We use `runBlocking` because the generated client operations are `suspend` functions.
  - **Client Instantiation**: We create an instance of `WidgetServiceClient` using its generated builder. Crucially, we pass our `MockHttpClientEngine` to the `httpClientEngine` configuration. In a real scenario, you'd configure `endpointUrl` to point to your actual backend and use a real `OkHttpEngine()` or similar.
  - **`createWidget` call**: We create an instance of the generated `CreateWidgetInput` data class, populate it, and pass it to the `client.createWidget()` suspend function. The result is a `CreateWidgetOutput` object, from which we can access the `widget`.
  - **`getWidget` call**: We demonstrate retrieving a widget, first a known one, then a non-existent one to showcase error handling.
  - **Error Handling**: We wrap the call to `getWidget` for the non-existent ID in a `try-catch` block to catch the generated `WidgetNotFoundException`. This demonstrates how Smithy maps Smithy `@error` shapes to Kotlin exceptions.
  - **`client.close()`**: It's important to close the client to release any underlying resources (like HTTP client pools).

2. Run Your Application

Now, compile and run your application using Gradle:

```
./gradlew run
```

You should see output similar to this:

```
> Task :run
Starting Smithy Kotlin Client Tutorial Application...

--- Calling CreateWidget ---
MockHttpClientEngine received request: POST /widgets
Created Widget: widget-123 - Test Widget (Price: 99.99)

--- Calling GetWidget for known ID ---
MockHttpClientEngine received request: GET /widgets/widget-456
Retrieved Existing Widget: widget-456 - Existing Widget (Price: 123.45)

--- Calling GetWidget for non-existent ID ---
MockHttpClientEngine received request: GET /widgets/non-existent-123
Caught expected error: Widget not found: non-existent-123

Application finished.
```

```
BUILD SUCCESSFUL in Xs
```

This output confirms that your generated Kotlin client is correctly instantiated, making calls to the (mocked) API, and handling both successful responses and specific API errors as defined in your Smithy model.

What you accomplished: You've successfully used the automatically generated Kotlin client within a sample application, demonstrating how to instantiate it, call its operations, and handle API-specific errors.

Common Pitfalls and Troubleshooting

While Smithy Kotlin client generation is powerful, developers can encounter a few common issues. Being aware of these can save you significant debugging time.

> ⚠️ **Common mistake:** Incorrect Smithy Gradle Plugin and Kotlin Codegen Plugin Versions - **Problem:** Mismatched or outdated versions of `smithy-gradle-plugin` and `smithy-kotlin-codegen` can lead to build errors, incompatible APIs, or unexpected generated code. Smithy 2.0 requires specific plugin versions. - **Solution:** Always refer to the official Smithy Kotlin client generation documentation for the latest recommended versions. Ensure your `smithyCli` and `smithyBuild` dependencies also align with the core Smithy version.

> ⚠️ **Common mistake:** Smithy Model Syntax Errors - **Problem:** Even a small typo or incorrect syntax in your `.smithy` file (e.g., missing a comma, misspelled trait, incorrect shape reference) can cause the `smithyBuild` task to fail. - **Solution:** The Smithy CLI usually provides detailed error messages, including line numbers. Pay close attention to these. Use an IDE with Smithy language support (if available) for syntax highlighting and basic validation. A common mistake is forgetting the `namespace` or `version` in the model file.

> ⚠️ **Common mistake:** Smithy Model Not Found or Incorrect Service Reference - **Problem:** The `smithyBuild` task might not find your `.smithy` files, or the `service` configuration in your `build.gradle.kts` `smithy` block might be incorrect. - **Solution:** - Verify your `src/main/smithy` directory exists and contains your `.smithy` files. - Double-check the

`service("com.example.widgets#WidgetService")` string in your `build.gradle.kts`. It must exactly match your Smithy service's fully qualified ID (namespace + `#` + service name). Case sensitivity matters!

⚠️ **Common mistake:** Missing or Incorrect Runtime Dependencies - **Problem:** The generated Kotlin client code relies on the `aws.sdk.kotlin:runtime` library and typically an HTTP client engine (like `http-client-engine-okhttp`). If these are missing or their versions are incompatible, your application won't compile or run. You might see `NoClassDefFoundError` or `Unresolved reference` errors. - **Solution:** Ensure all necessary `implementation` dependencies for the Smithy Kotlin runtime, protocol libraries (e.g., `aws-json-protocols`), and an `HttpClientEngine` are correctly added to your `build.gradle.kts` with compatible versions.

⚠️ **Common mistake:** JDK Version Mismatch - **Problem:** Smithy Kotlin code generation, like most modern Kotlin development, requires JDK 11 or higher. If your project is configured for an older JDK, or your environment's `JAVA_HOME` points to an incompatible version, you might encounter compilation issues. - **Solution:** Verify your `java { toolchain { languageVersion = JavaLanguageVersion.of(11) } }` in `build.gradle.kts` and ensure your system's `java -version` output matches. Use `SDKMAN!` or similar tools to manage JDK versions easily.

⚠️ **What can go wrong:** Generated Code Not Recompiled - **Problem:** Sometimes, if you modify your Smithy model but Gradle's incremental build doesn't detect the change, the generated code might not be updated. - **Solution:** Run `./gradlew clean smithyBuild` or `./gradlew clean generateSmithy` to force a clean build and regeneration. This ensures that the latest Smithy model is processed.

Debugging Generated Code: The generated code is just regular Kotlin. If you encounter issues, don't be afraid to inspect the generated files in `build/smithy/source/kotlin/client`. You can set breakpoints, step through the code, and understand how your Smithy model translates into runtime behavior. This is particularly useful for understanding serialization/deserialization logic or how HTTP requests are constructed.

By keeping these common pitfalls in mind, you can navigate the Smithy Kotlin client generation process much more smoothly.

Next Steps and Advanced Topics

Congratulations! You've successfully defined a Smithy service model, generated a type-safe Kotlin client, and used it in a sample application. This is just the beginning of what you can achieve with Smithy. Here are some next steps and advanced topics to explore:

What to Build Next

To solidify your understanding and explore further capabilities, consider extending your current project with these ideas:

1. **Implement a Simple HTTP Server to Back Your Client:** Replace the `MockHttpClientEngine` with a real backend. You could use Ktor, Spring Boot, or a simple embedded HTTP server in Kotlin to implement the `CreateWidget` and `GetWidget` operations based on the Smithy model's HTTP binding traits. This will allow your generated client to interact with a live service, providing a more realistic end-to-end experience.
    - **Challenge:** Implement the server-side logic to handle `POST /widgets` and `GET /widgets/{id}` requests, including returning the `WidgetNotFoundException` for non-existent IDs.
2. **Add More Complex Smithy Shapes and Traits:** Expand your `widgets.smithy` model to include more advanced Smithy features.
    - **Unions:** Define a shape that can hold one of several types (e.g., `WidgetDetail` could be either `SimpleWidget` or `ComplexWidget`).
    - **Enums:** Define a fixed set of string values for a field (e.g., `WidgetStatus: Enum<"ACTIVE", "INACTIVE">`).
    - **Maps and Lists:** Add properties that are collections of other shapes (e.g., a `tags: Map<String, String>` or `relatedWidgetIds: List<String>`).
    - **Custom Traits:** Explore defining your own custom traits to add specific metadata that might influence future code generation or documentation.
    - **Challenge:** Observe how the Kotlin client code generation adapts to these new Smithy constructs, especially how unions map to Kotlin `sealed class`es.

3. **Explore Different Protocols and HTTP Bindings:** While our tutorial used default HTTP/JSON bindings, Smithy is protocol-agnostic.
    - **Custom HTTP Bindings:** Experiment with different `@http` traits (e.g., `method: "PUT"`, `uri` with more complex path segments, `@httpHeader` to send data in headers).
    - **Protocol-Specific Traits:** Investigate how Smithy can be used with other protocols if you were to generate for AWS services, for instance.
    - **Challenge:** Create a new operation in your `WidgetService` that takes an input parameter via an HTTP header using the `@httpHeader` trait, and verify the generated client's usage.

Advanced Topics for Further Learning

- **Smithy for Server-Side Code Generation:** Just as Smithy can generate clients, it can also generate server-side stubs and interfaces, ensuring your client and server implementations are always in sync.
- **Smithy for Documentation Generation:** Smithy models can be used to automatically generate API documentation in various formats, keeping your docs up-to-date with your API definition.
- **Smithy CLI:** Explore using the Smithy CLI directly for building and validating models, which can be useful outside of a Gradle context.
- **Custom Code Generators:** For highly specialized needs, you can even write your own Smithy code generators.
- **Integration with CI/CD:** Incorporate Smithy code generation into your continuous integration/continuous deployment pipelines to automate client updates whenever your API model changes.

By diving deeper into these areas, you'll uncover the full potential of Smithy as a foundational tool for API development, moving beyond just client generation to a truly model-driven approach.