Guidewire InsuranceSuite™

REST API Client Guide

Release 10.1.0

GUIDEWIRE
Adapt and succeed™

Product Name: Guidewire InsuranceSuite
Product Release: 10.1.0
Document Name: REST API Client Guide
Document Revision: 14-June-2021

Contents

Support

For assistance, visit the Guidewire Community.

Guidewire customers

https://community.guidewire.com

Guidewire partners

https://partner.guidewire.com

chapter 1

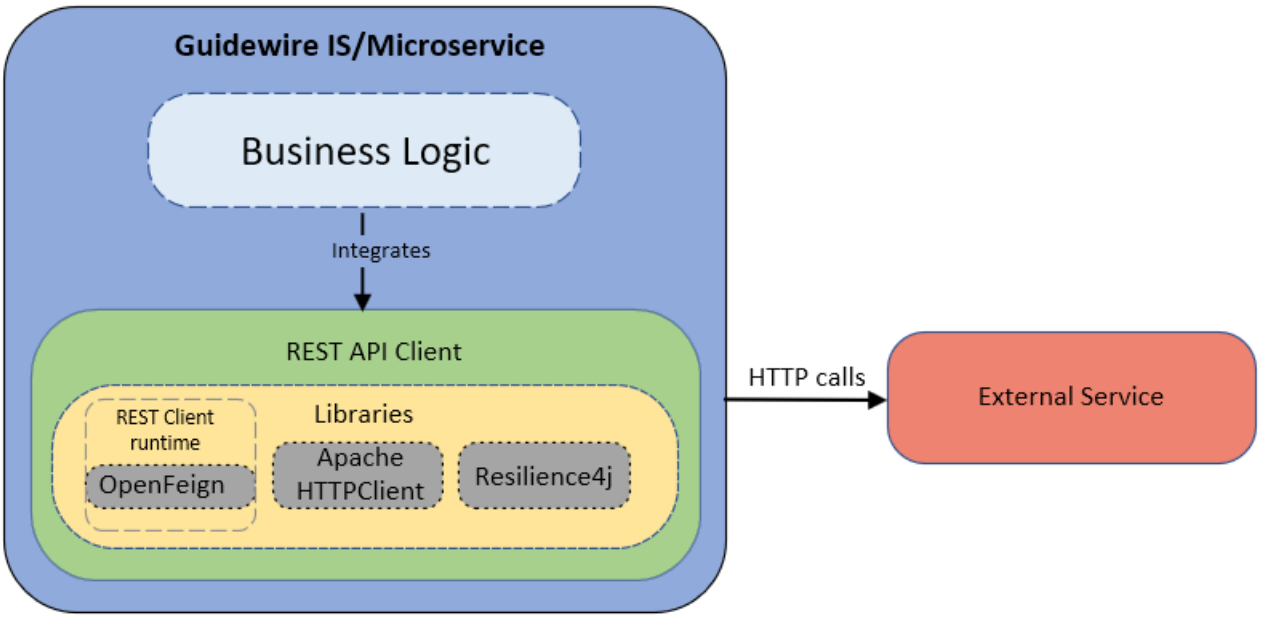
Overview

The InsuranceSuite REST API Client provides functionality that Guidewire InsuranceSuite applications and integrations use to make outbound HTTP calls to *OpenAPI Spec*-compliant REST services. Using the client enables loosely-coupled external code to be invoked by events that are sourced in InsuranceSuite applications.

Note: The REST API Client does not support outbound HTTP calls to InsuranceSuite applications, such as ClaimCenter, PolicyCenter, and BillingCenter.

The Guidewire REST API client consists of two modules, a client library and a client plugin:

- The **client library** provides fault tolerance features that integration developers use to make synchronous REST API calls from InsuranceSuite
- The **client plugin** generates REST API for making outbound HTTP requests to *OpenAPI Spec*-compliant REST services



About the client library

The client library is distributed as a JAR file. The library works with OpenFeign annotated API and Jackson annotated data objects. Integration developers can manually create these objects or provide existing. If their REST service complies with the *OpenAPI Spec* they can use the client plugin to generate that set of classes.

The library provides the following features:

- Code generation in Java and isolation from Gosu for rapid development
- Integrates with `Resilience4j` to provide lightweight, easy-to-use fault tolerance library. Reseliense4j introduces higher-order functions (decorators) to enhance any functional interface, lambda expression, or method reference with a Circuit Breaker, and Retry. You can stack more than one decorator on any functional interface, lambda expression or method reference.
- Integrates with `OpenFeign` to facilitate the creation of web service client implementations, with quality of service features, configurable and customizable clients. Uses *OpenFeign* to implement *timeout*.
- Fault tolerance with *retry*, *circuit breaker*, and *fallback*
- Observability with logging and traceability
- Authentication that supports Basic, API Key, Bearer, OAuth 2.0, and Mutual TLS authentications
- Uses Apache HttpClient for robust client connections
- Uses Apache OAuth implementation for OAuth authentication
- By default uses IS standard headers in logging context
- Uses `Jackson` as JSON parser

About the client plugin

You use client plugin to generate an OpenFeign based REST client against a service that supports the *OpenAPI Spec*. If you have an existing OpenFeign based REST API interfaces you can use the REST API library without the client generation and take advantage of the fault tolerance features.

The client plugin provides the following features:

- Uses `OpenAPI Generator` to generate API client libraries, server stubs, documentation and configuration automatically against an `OpenAPI Spec`-compliant REST API service endpoint.

chapter 2

Before you begin

To start using the REST API client, install the following software:

Software	URL	Description
Java 8, or Java 11 as your default JDK		The REST Client supports both Java 8 and Java 11
InsuranceSuite version 2020.11 (Banff) or later		The REST API client is available and supported on InsuranceSuite version 2020.11 (Banff) or later
Gradle	https://github.com/gradle/gradle	Build automation framework. Using the REST API client requires basic familiarity with Gradle.
IntelliJ	https://www.jetbrains.com/idea/	Guidewire recommends using InteliJ as Interactive Development Environment (IDE)

The REST API client is implemented using the following Open Source libraries. Use the provided links to explore their features:

Library	URL	Description
OpenFeign	https://github.com/OpenFeign/feign	OpenFeign is a microservice invocation framework, which is mainly used in the consumer side, that is, invoking services.
Resilience4J	https://resilience4j.readme.io/docs	Fault tolerance library
Jackson	https://github.com/FasterXML/jackson	JSON library

chapter 3

Getting started

Guidewire distributes the REST API client as a ZIP file that contains a Gradle project. You can download the REST API client `rest-api-client-project-`*`version`*`.zip` file from the Guidewire Community website. The project contains the distribution for both, the REST API client plugin and REST API client library. The following table lists all project folders, the project configuration files, and explains their purpose.

Project Folder	Description
/build	Contains the generated REST API client along with all dependent libraries. You deploy this ZIP to the IS application.
/build-script-libs	The libraries that Gradle uses
/client-libs	The REST API client library `client-`*`version`*`.jar` along with a POM file that can be used to build the project with Maven.
/compile-libs	Use the libraries in this folder for testing.
/endpoint	Contains the OpenApi code generator plugin. This folder appears when you open the project in IntelliJ or run *gradle wrap*.
/gradle	Gradle Wrapper executable JAR and its properties file
/libs	The dependent libraries that REST API client uses
build.gradle	Gradle build script for configuring the current project
gradle.properties	Contains the properties for the build script
gradlew	Gradle Wrapper script for Unix-based systems. This file downloads the proper version of Gradle if it is not present in the environment. This folder appears when you open the project in IntelliJ or run *gradle wrap*.
gradlew.bat	Gradle Wrapper script for Windows. This file downloads the proper version of Gradle if it is not present in the environment. This file appears when you open the project in IntelliJ or run *gradle wrap*.
README.md	The project documentation
settings.gradle	Gradle settings script for configuring the Gradle build. Defines all included sub-modules. During initialization, Gradle reads the *settings.gradle* file to figure out which projects to include in a build. .

The REST API client project is configured to use the PetStore https://petstore.swagger.io/ REST service to illustrate how to generate the REST API client with this service. Initially, you use this configuration to test the generated client in your IS application. Next, you modify the project settings to configure the OpenAPI-compliant service endpoints and generate your own client. To start using the code generation plugin you need to unzip the distribution zip file and open the Gradle project in your IDE. The instructions in this guide assume that you use IntelliJ.

Step1: Download the REST API client

To download the client, log in to Guidewire Community website and download the `rest-api-client-project-version.zip file.`

Step 2: Unzip the zip file to your environment

1. Unzip the file to your local file system
2. Import the Gradle project to IntelliJ, or the IDE that you use

Step 2: Update the project configuration

Update the following configuration files to configure your client:

1. Open `build.gradle` file to update the following three parameters:
   - `gwRestGen_endpoint_package`

     Defines the generated client package name
   - `gwRestGen_endpoint_source`

     Defines the source for the endpoint. It can be a REST service endpoint OpenAPI spec URL or a local file path to the OpenAPI spec JSON file.
   - `gwRestGen_isInsuranceSuite`

     Defines the client deployment target. When the target is an InsuranceSuite application the build task packages the client along with the necessary libraries.

   ```
   ext {
     gwRestGen_endpoint_package = "petstore"
     gwRestGen_endpoint_source = "https://petstore.swagger.io/v2/swagger.json"
     gwRestGen_isInsuranceSuite = "true"
   }
   ```
2. Update *settings.gradle* to include the client package to the build. You can configure multiple client packages, as shown in the following example. Respectively, you must configure the endpoint parameters for each package in *build.gradle*:

   ```
   include 'claimcenter:v1'
   include 'policycenter:v1'
   ```

Step 3: Build the client

The project provides Gradle task that you use to build the client. For details, see “REST API client plugin tasks” on page 37. In this project the code generation plugin generates a client against the PetStore service. Since the endpoint *source* is defined as an OpenAPI spec URL, you execute the following commands:

1. Run the *restDownload* task to download the OpenAPI spec JSON file. To run a task with the wrapper, use one of the following commands from a Terminal window:
   - On Mac or Linux, run

     ```
     ./gradlew restDownload
     ```
   - On Windows:

     ```
     gradlew restDownload
     ```

   In IntelliJ, you can also run the `restDownload` task from Gradle task window.

   Note: If you define the `gwRestGen_endpoint_source` as a relative path to a JSON file, the `restDownload` does not appear in the available tasks.

2. Generate the client. To run a task with the wrapper, use one of the following commands from a Terminal window:
   - On Mac or Linux, run
     ```
     ./gradlew build
     ```
   - On Windows:
     ```
     gradlew build
     ```

The build generates the client library `<project>/endpoint/build/libs/gwgen.petstore-version.jar`.

Step 4: Publish the generated client

To publish the client to InsuranceSuite application run the *publish* task. To run a task with the wrapper, use one of the following commands from a Terminal window:

- On Mac or Linux, run
  ```
  ./gradlew publish
  ```
- On Windows:
  ```
  gradlew publish
  ```

In IntelliJ, you can also run the `publish` task from Gradle task window.

The task zips up the following libraries in `build/distribution/endpoints-version.zip` file:

- The generated client `<project>/endpoint/build/libs/gwgen.petstore-version.jar`
- The REST API client library `/client/libs/client-version.jar`
- All libraries from */libs* folder

Step 5: Deploy to InsuranceSuite application

1. Stop the InsuranceSuite application
2. Extract `build/distribution/endpoints-version.zip` to `<IS>modules/configuration/plugins/shared/lib` folder in an InsuranceSuite application installation
3. Start the InsuranceSuite application

Step 6: Test the client in InsuranceSuite application

To test the client you create and run a test in Gosu Scratchpad in InsuranceSuite Studio. The test submits REST API requests to the PetStore service to get the store inventory and get information about the sold and available pets.

1. Open InsuranceSuite Studio:
   ```
   gwb studio
   ```
2. To open Gosu scratchpad, click Tools → Gosu Scratchpad
3. In Gosu scratchpad, type the following code:
   ```
   uses gw.restclient.config.Config
   uses gw.restclient.util.StaticMapper
   uses gwgen.petstore.ApiClient
   uses gwgen.petstore.api.StoreApi
   uses gwgen.petstore.api.PetApi
   uses gwgen.petstore.model.Pet
   uses gwgen.petstore.model.Category
   uses gwgen.petstore.model.Pet.StatusEnum

   var mapper = StaticMapper.yaml()
   var config = Config.builder().basePath(ApiClient.BASE_PATH).build()
   var storeApi  = config.buildAPI(StoreApi)
   var inv = storeApi.getInventory()

   inv.forEach( \ key, cnt -> print("${key} = ${cnt}"))
   var petApi = config.buildAPI(PetApi)
   var rtn = petApi.findPetsByStatus({ StatusEnum.AVAILABLE.toString(),
   ```

```
                                        StatusEnum.SOLD.toString() })
print(mapper.writerWithDefaultPrettyPrinter().writeValueAsString(rtn))
```

4. To run the code, click Run on the scratchpad navigation bar.
5. The test completes and generate the similar to the following output. The output below displays only part of the response:

```
"C:\Program Files\Java\jdk1.8.0_152\bin\java.exe"
              -javaagent:C:\ClaimCenter\studio\sdk\lib\idea_rt.jar=50833:C:\ClaimCenter\studio\sdk\bin
              -Dfile.encoding=UTF-8 -classpath C:\Users\user1\AppData\Local\Temp\classpath7313323.jar
              gw.lang.Gosu "C:\Users\user1\AppData\Local\Temp\6dace4b4\Gosu Scratchpad.gsp"
sold = 190
string = 292
pending = 182
availableTINFOIlllllllllllllllllll....LLLLLLLLLL = 1
available = 292
soldTINFOIlllllllllllllllllll....LLLLLLLLLL = 1
---
- id: 7777778888889957853
  category:
    name: "string"
  name: "doggie"
  photoUrls:
  - "string"
  tags:
  - name: "string"
  status: "available"
- id: 7777778888889957854
  category:
    name: "string"
  name: "doggie"
  photoUrls:
  - "string"
  tags:
  - name: "string"
  status: "available"
- id: 7777778888889957865
  category:
    name: "string"
  name: "doggie"
  photoUrls:
  - "string"
  tags:
  - name: "string"
  status: "available"
- id: 213124125
  category:
    id: 12
    name: "cats"
  name: "Garfield"
  photoUrls: []
  status: "available"
- id: 7777778888889957875
  category:
    name: "string"
  name: "doggie"
  photoUrls:
  - "string"
  tags:
  - name: "string"
  status: "available"
- id: 1234
  category:
    name: "string"
  name: "casper"
  photoUrls:
  - "string"
  tags:
  - name: "string"
  status: "available"
....
Process finished with exit code 0
```

chapter 4

REST API client library

This section provides information to help you plan your integration project and provides technical details to successfully configure and use the REST API library. Typically, integration developers use the library in the following two scenarios:

- The integration has a web service client implementations using OpenFeign and only needs to use the fault tolerance features from the library
- The integration does not have an OpenFeign client implementation and the targeted service is not OpenAPI Spec compliant

REST API client library configuration

A configuration describes all the parameters that define a single connection. The REST API client library configuration contains information about associated events, such as *retry*, *circuit breaker*, *fallback*, the settings for *timeout*, traceability, and authentication. The client library `Config` class instance represents the merged view of the configuration settings and provides the means to set the configuration preferences.

There are two approaches in implementing the library configuration, external and runtime. In the runtime configuration, the configuration parameters can be provided as classes, which instances that are build at runtime. In the external configuration the parameters are non-primitive objects that are externalized in files written in YAML or JSON syntax. The external and runtime configurations differ in the setting of the non JSON type properties. These properties are represented by a class name or the class name of a supplier. When configuring these properties, the external `Config` object expects classes, or the class of a supplier to create the desired object or a subclass of the desired object. The runtime `Config` expects an instance of the class.

The REST API client configuration properties are immutable objects and each object has a `builder()` static method. To clone an object, you must call the `toBuilder()` method. In cases when a supplier requires an additional information, this information is passed as a map to the class. Examples of such suppliers are a credential supplier, a context supplier, and an interceptor supplier.

About the Config class

The `Config()` class has three methods to return the builders:

Method	Description
`builder()`	Returns a naked builder for `Config` objects.

Method	Description
`parseYaml(String input)`	Parses the YAML content and returns a `Config.Builder`
`parseJson(String input)`	Parses the JSON content and returns a `Config.Builder`

The following example illustrates how to use the `Config.builder()` method:

Example

```
Config cfg = Config.parseYaml(STD_CONFIG)
  .basePath(baseUrl)
  .build();

Config cfg = Config.builder()
  .basePath(baseUrl)
  .build();
}
```

The configuration object utilizes builders to construct immutable objects. The `Config` class parses the external configuration to produce a runtime `Config.Builder` object that allows you to modify the configuration. After modifying the configuration, use the builder's `build()` method to construct the new immutable `config` object.

External configuration

The following table lists the external configuration objects:

Parameter	Description
`basePath`	The base path to the REST endpoint URL
`auth`	The authentication method that the REST API client uses. The following methods are supported: • `HttpBasic` • `HttpBearer` • `ApiKey` • `OAuth`
`eventHandlers`	An array of `eventHandlers` that are applied to each request. The type property determines the event handler type: • `retry` • `circuitbreaker` • `fallback`
`loggerName`	Defines a logger for an endpoint. Use this setting to define different logger for each endpoint.
`logLevel`	Defines custom log levels. The **org.apache.log4j.Level** must be at DEBUG level to be able to set the following log levels: • `NONE` No logging • `BASIC` Logs method, URL, response status, and execution time. • `HEADERS` Logs the BASIC information and request/response headers. **Note:** The AUTH headers are also logged and this is a security exposure. Avoid using DEBUG level in production. • `FULL` Logs the HEADERS information, body, and metadata for request/response.

Parameter	Description
	By default the `logLevel` is set to NONE.
`timeout`	The timeout configuration for the request The timeout object is optional. If it is not supplied, a default timeout object with the default settings is provided .
• `connectTimeoutMillis`	Defines the connection timeout in milliseconds. The default is set to 5 ms.
• `readTimeoutMillis`	Defines the timeout for the waiting time on a read request. The timeout is in milliseconds. The default is 5 ms.
`httpClient`	The configuration object for an HTTP client or Mutual Transport Layer Security (TLS) authentication. Providing this configuration is optional.
• `clientSupplierClass`	A functional class that accepts a `Map<String,String>` and returns an Apache `HttpClient`. This setting is ignored if `ClientSupplierClass` is provided.
• `hostnameVerifierSupplierClass`	A functional class that accepts a `Map<String,String>` and returns an object that implements `HostnameVerifier`. This is ignored if `ClientSupplierClass` is provided.
• `contextSupplierClass`	A functional class that accepts a `Map<String,String>` and returns an object that implements `SSLSocketFactory`.
• `properties`	The map of name/values pairs that is passed to the `ClientSupplierClass`, `contextSupplierClass`, and `hostnameVerifierSupplierClass` to get their respective `CredentialSupplier` object. . The JSON presentation looks like: `"properties":{ "key1":"value", "key2":"value"}` .
`additionalInterceptors`	An array of additional interceptors. Interceptors provide a mechanism to intercept and/or mutate outgoing requests or incoming responses. The following interceptors are already added to the REST API client by declaring the corresponding *auth* configuration object: • `ForwardingInterceptor` • `HttpBearerAuth` • `ApiKeyAuth`
• `interceptorSupplierClass` Required	Provides the name of the functional class for the additional supplier. The class accepts a `Map<String,String>` and returns an object that implements `Feign.RequestInterceptor`. Feign provides the `Feign.RequestInterceptor` interface for adding/removing/mutating any part of the request.
• `properties`	The properties are map of name/values that are passed to the `interceptorSupplierClass` class. The JSON presentation follows this format: `"properties":{ "key1":"value", "key2":"value"}`
`traceActionsClass`	A class that implements `TraceabilityActions`. This is required if you use a diagnostic context that is different than *Mapped Diagnostic Context* (MDC) for holding this information. Implementing this class is also required if you use property keys that are different than HTTP header keys that the IS applications use.
`mapperSupplierClass`	A supplier class that returns an object mapper
`credentialSupplierSupplierClass`	A functional class that accepts a property map with the credentials that are used to authenticate the HTTP requests. The class returns an object the implements the `CredentialSupplier` class.

Parameter	Description
• `properties`	The credential property map that is passed to the `credentialSupplierSupplierClass` class. The supplier is initialized with these properties, and is asked for the values of the required credentials depending on the authentication method. For example, • `HttpBasic` uses `username` and `password` • `HttpBearer` uses the schema or the `bearToken` • `ApiKey` uses `apikey` • `OAuth` uses `username`, `password`, `clientId`, `clientSecret`

Runtime configuration

At runtime, the REST library first parses the external configuration parameters and then calls the suppliers to build their instances. The following table lists the runtime configuration parameters:

Parameter	Description
`basePath`	The base path to the REST endpoint URL
`auth`	The authentication method that the REST API client uses. The following methods are supported: • `HttpBasic` • `HttpBearer` • `ApiKey` • `OAuth`
`eventHandlers`	An array of `eventHandlers` that are applied to each request. The type property determines the event handler type: • `retry` • `circuitbreaker` • `fallback`
`loggerName`	Defines a logger for an endpoint. Use this setting to define different logger for each endpoint.
`logLevel`	Defines what information to log: • `NONE` No logging • `BASIC` Logs method, URL, response status, and execution time. • `HEADERS` Logs the BASIC information and request/response headers. • `FULL` Logs the HEADERS information, body, and metadata for request/response. By default the `logLevel` is set to NONE.
`timeout`	The timeout configuration for the request
• `connectTimeoutMillis` Required	Defines the connection timeout in milliseconds. The default is set to 5 ms.
• `readTimeoutMillis` Required	Defines the timeout for the waiting time on a read request. The timeout is in milliseconds. The default is 5 ms.
`httpClient`	The configuration object for an HTTP client or Mutual Transport Layer Security (TLS) authentication
• `client`	An Apache `HttpClient`.

Parameter	Description
• `hostnameVerifier`	A functional class that accepts a `Map<String,String>` and returns an object that implements `HostnameVerifier`. This is ignored if `ClientSupplierClass` is provided.
• `context`	A functional class that accepts a `Map<String,String>` and returns an object that implements `SSLSocketFactory`.
`interceptors`	An array of additional interceptors. Interceptors provide a mechanism to intercept and/or mutate outgoing requests or incoming responses. The following interceptors are already added to the REST API client: • `ForwardingInterceptor` • `HttpBearerAuth` • `ApiKeyAuth`
`traceActionsClass`	A class that implements `TraceabilityActions`. This is required only if you use a diagnostic context that is different than *Mapped Diagnostic Context* (MDC) for holding this information.
`mapper`	An object mapper
`credentialSupplier`	A functional class that accepts a key and a default value with the credentials that are used to authenticate the HTTP requests. The class returns an object the implements the `CredentialSupplier` class.

REST API client library authentication

The client library supports the following authentication methods:

- **Basic authentication**

  This method uses the HTTP header itself to pass the encoded username and password.

- **Bearer authentication**

  Bearer authentication (also called token authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens

- **OAuth 2.0 authentication**

  `OAuth 2.0` is an authorization protocol that gives an API client limited access to user data on a web server

- **API Keys**

  Some APIs use API keys for authorization. An API key is a token that a client provides when making API calls.

See also

- “Using API security” on page 33

HttpBasic authentication

This is the most straightforward method and the easiest. With this method, the sender places a `username:password` into the request header. The username and password are encoded with Base64, which is an encoding technique that converts the username and password into a set of 64 characters to ensure safe transmission.

Parameter	Description
`username`	The user name. This can also be supplied by the credential supplier.
`password`	The password. This can be provided by the credential supplier.
`method`	The method is always `HttpBasic`

See also

- "Using API security" on page 33

HttpBearer authentication

The `HttpBearer` authentication uses a bearer token to allow access to a certain resource or URL. The client must send this token in the Authorization header when making requests to protected resources:

```
Authorization: Bearer <token>
```

Parameter	Description
`method`	The method is always `bearerToken`
`scheme`	The token scheme that by default is set to `bearerToken`
`bearerToken`	The token. The token could be provided by the credential supplier. The token always uses the scheme as the credential name.

See also

- "Using API security" on page 33

ApiKey authentication

An API key is a token that a client provides when making API calls. The key can be sent in the query string:

```
GET /something?api_key=abcdef12345
```

or as a request header:

```
GET /something HTTP/1.1
X-API-Key: abcdef12345
```

or as a cookie:

```
GET /something HTTP/1.1
Cookie: X-API-KEY=abcdef12345
```

API keys are a secret that only the client and server know. Like Basic authentication, API key-based authentication is only considered secure if used together with other security mechanisms such as HTTPS/SSL.

Parameter	Description
`method`	The method is always `ApiKey`
`location`	Defines the location where the name-value pair is written. The location can be `header`, `query`, or `cookie`. The default location is `header`.
`paramName`	The name of the `header`, `query param`, or `cookie`. The default value is `api`.
`apiKey`	The token. The token can also be provided by the credential supplier. The token uses the `paramName` as the credential name.

See also

- "Using API security" on page 33

OAuth authentication

OAuth relies on authentication scenarios called *flows*, which allow the resource owner (user) to share the protected content from the resource server without sharing their credentials. For that purpose, an OAuth 2.0 server issues access tokens that the client applications can use to access protected resources on behalf of the resource owner. For more information about OAuth 2.0, see `oauth.net` and `RFC 6749`.

Parameter	Description
`method`	The method is always `OAuth`
`flow`	Used to implement the work flow. Possible values are `accessCode`, `implicit`, `password`, and `application`.
`authorizationUrl`	The `url` to connect for authorization
`tokenUrl`	The `url` to connect for `accessToken`
`scopes`	The requested scopes
`credentials`	The `OAuth` credentials:
• `clientId`	The client id. This id can also be requested from the credential supplier
• `clientSecret`	The client secret. Can also be requested from the credential supplier.
• `username`	The user name. Can also be requested from the credential supplier.
• `password`	The password .This can also be requested from the credential supplier.

Understanding the credentialSupplier class

The client library invokes the `credentialSupplier` class that you provide to process the individual authentication credential. For example, this class can be used to decrypt the external string in a configuration file or it could request the credential from an active directory.

When executing its build method, the `Config.Builder` calls the `credentialSupplier` class for each *auth* property. The following example illustrates the implementation for decrypting the credentials depending on the *auth* property.

Example

```
class DecryptingCredentialSupplier implements Config.CredentialSupplier {
      @Override
      public String apply(String key, String defaultValue) {
        switch (key) {
          case "username":
          case "clientId":
                return defaultValue;
          case "password":
          case "clientSecret":
          case "BearerToken":
          default: // for ApiKeyAuth.paramName and HttpBearerAuth.scheme
                return defaultValue == null ? defaultValue : cipher.decrypt(Base64.getDecoder().decode(defaultValue));
         }
      }
   }
```

In the following example, the implementation obtains the credentials from Active Directory service.

Example

```
public class JNDICredentialSupplier implements CredentialSupplierSupplier {
  @SneakyThrows
  @Override
  public Config.CredentialSupplier apply(Map<String, String> properties) {
    Properties props = new Properties();
    if (!properties.containsKey(Context.INITIAL_CONTEXT_FACTORY) || !properties.containsKey(Context.PROVIDER_URL)) {
      throw new IllegalArgumentException("Required param missing");
    }
    props.putAll(properties);
    final Context initialContext = new InitialContext(props);
    return (key, value) -> (String) initialContext.lookup(key);
  }
}
```

REST API client library event handlers

The client library implements *retry*, *fallback*, *circuit breaker*, and *timeout* fault tolerance patterns.

The *retry* pattern enables dealing with communication errors that can be corrected by attempting them multiple times. The *fallback* pattern enables your service to continue the execution in case of a failed request to another service. The *timeout* pattern provides an upper bound to latency. See "External configuration" on page 16 and "Runtime configuration" on page 18 for details. The *circuit breaker* addresses the problem of accidental denial of service attacks due to retries and fast fallbacks in case of persisting communication errors.

The *retry*, *fallback*, and *circuit breaker* patterns are implemented using the *Resilience4j* library and configured through the `eventHandlers` configuration parameter. The *timeout* is implemented using *OpenFeign* library as a *timeout* configuration parameter.

The configuration contains information about the associated events in the *retry*, *circuit breaker*, and *fallback* patterns. The `eventHandlers` configuration parameter is an array of `eventHandlers` that are applied to each request.

For implementation examples, see:

- "Using the fault tolerance features" on page 30

The retry event handler

Operations can time out or fail because of broken connections, network glitches, unavailability of upstream services. Client applications deal with these failures by implementing retries. The *retry* is a very simple pattern where failed requests are retried a configurable number of times in case of a failure before the operation is marked as a failure.

You can provide a custom global `RetryConfig`. In order to create a custom global `RetryConfig`, you can use the `RetryConfig` builder.

Parameter	Default value	Description
`type`	`retry`	The configuration property type
`name`	`GW_RETRY`	The name of the retry instance that is used for logging purposes
`maxAttempts`	3	The maximum number of retry attempts
`backoff`		Configures custom `backoff` algorithm for handling retries of failed network calls.

• `type`		The type is `random` or `exponential`.
		• In `random` exponential backoff the clients wait random interval between consecutive retries. Typically the wait interval is calculated using the following formula:
		`wait_interval = base * multiplier^n`
		• With the `exponential` backoff the clients wait progressively longer intervals between consecutive retries. In this algorithm the wait interval is calculated using the following formula:
		`wait_interval = (base * 2^n) +/- (random_interval)`
		• *base* is the initial interval, for example wait for the first retry
		• *n* is the number of failures that have occurred

Parameter	Default value	Description
• `interval`	500 ms	The base `backoff` period that is set in milliseconds.
• `multiplier`	1.5	The `multiplier` is an arbitrary multiplier that can be replaced with any suitable value for `backoff` period. The value is in Double format.
• `randomizationFactor`	0.5	The randomization factor. The value is in Double format.
`retryOnExceptionPredicateClass`	`throwable -> true`	Configures a `Predicate` that evaluates if an exception has to be retried. The `Predicate` must return true, if the exception has to be retried, otherwise it must return false. To invoke retry when specified exceptions occurred
`retryOnResultPredicateClass`	`result -> false`	Configures a `Predicate` that evaluates if a result has to be retried. The `Predicate` must return true, if the result has to be retried, otherwise it must return false. Used to invoke retry when the specified results are received.
`intervalFunctionClass`	`numOfAttempts -> waitDuration`	A class that extends `IntervalFunction`. This functional is used to modify the waiting interval after a failure. By default the wait duration remains constant.

See also:

• "Using the fault tolerance features" on page 30

The circuitBreaker event handler

The circuit breaker is a pattern that helps preventing cascading failures in a system. The circuit breaker pattern allows you to build a fault-tolerant and resilient system that can survive gracefully when key services are either unavailable or have high latency.

It is implemented as a stateful software component that switches between three states:

- *closed* - requests can flow freely
- *open* - requests are rejected without being submitted to the remote resource
- *half-open* - one probe request is allowed to decide whether to close the circuit again

You can provide your own custom global `CircuitBreakerConfig`. In order to create a custom global `CircuitBreakerConfig`, you can use the `CircuitBreakerConfig.builder()` method. Using the builder, you can configure the following properties.

Parameter	Default value	Description
`type`	`circuitBreaker`	The type is always `circuitBreaker`
`automaticTransitionFromOpen ToHalfOpenEnabled`	`false`	If set to `true` it means that the CircuitBreaker automatically transitions from open to half-open state and no call is needed to trigger the transition. A thread is created to monitor all the instances of CircuitBreakers to transition them to HALF_OPEN once `waitDurationInOpenState` passes. Whereas, if set to false the transition to HALF_OPEN

Parameter	Default value	Description
		only happens if a call is made, even after waitDurationInOpenState is passed. The advantage here is no thread monitors the state of all CircuitBreakers.
`failureRateThreshold`	50%	Configures the failure rate threshold in percentage. When the failure rate is equal or greater than the threshold the CircuitBreaker transitions to open and starts short-circuiting calls.
`minimumNumberOfCalls`	100	Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate or slow call rate. For example, if minimumNumberOfCalls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated. If only 9 calls have been recorded the CircuitBreaker does not transition to open even if all 9 calls have failed.
`name`	`GW_CIRCUITBREAKER`	The `circuitBreaker` name. It is used to create or retrieve the circuit breaker instance.
`permittedNumberOfCalls InHalfOpenState`	10	Configures the number of permitted calls when the CircuitBreaker is in `HALF_OPEN` state.
`slidingWindowType`	`COUNT_BASED`	If the sliding window is COUNT_BASED, the last `slidingWindowSize` calls are recorded and aggregated. If the sliding window is TIME_BASED, the calls of the last `slidingWindowSize` seconds recorded and aggregated.To determine the count type, default is `COUNT_BASED`, alternative is `TIME_BASED`
`slowCallDurationThreshold`	60000 ms	Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls.
`slowCallRateThreshold`	50	Configures a threshold in percentage. The CircuitBreaker considers a call as slow when the call duration is greater than `slowCallDurationThreshold`. When the percentage of slow calls is equal or greater the threshold, the `CircuitBreaker` transitions to open and starts short-circuiting calls.
`writableStackTraceEnabled`	`true`	Enables writing the stack trace.
`waitIntervalFunctionIn OpenStateSupplierClass`		A functional supplier class. When passed a map, it returns an object that implements `IntervalFunction`.
`recordExceptionSupplierClass`		A functional supplier class. When passed a map, it returns an object that implements `Predicate<Throwable>`.
`properties`		The map that is passed to the above supplier. The presentation in JSON looks like

Parameter	Default value	Description
		`"properties":{ "key1":"value", "key2":"value"}).`

See also:

- "Using the fault tolerance features" on page 30

The fallback event handler

The fallback pattern enables your service to continue the execution in case of a failed request to another service. Instead of aborting the computation because of a missing response, you fill in a fallback value.

Typically, *Fallbacks* are called when *Exceptions* are thrown. Exceptions can occur when the HTTP request fails, or when one of the `FeignDecorators` activates the `CircuitBreaker`.

Parameters	Description
`type`	always `fallback`
`exceptionTypes`	A collection of Exception classes that the fallback uses
`exceptionHandlerClass`	A class that handles the exception and returns an appropriate object, external
`exceptionHandler`	An object class that implements a `Function` that accepts a `Throwable` and returns an appropriate response at runtime

The following example shows the configuration code for `sampleRESTApi` endpoint connection:

Example

```
Config.builder()
        .basePath(sampleRESTApiRule.baseUrl())
        .eventHandler(FallbackSetup.builder()
          .exceptionType(RetryableException.class)
          .exceptionHandler(t -> Collections.singletonList(SAMPLE_API))
          .build())
        .build()
        .buildAPI(sampleRESTApi.class)
        .findAll()
```

See also:

- "Using the fault tolerance features" on page 30

REST API client configuration schema

The following JSON file shows configuration schema:

```
{
      "type": "object",
      "id": "com.guidewire.restclient.config.ConfigEx",
      "description": "This is an external representation of the config object.\n  This accepts suppliers for what the
objects that the configuration needs.\n  The config object can be initialized via ConfigEx.parse()",
      "properties": {
      "additionalInterceptors": {
      "type": "array",
      "description": "This is a collection of additional interceptors, authentication, tracing are added as
interceptors.",
      "items": {
      "type": "object",
      "id": "com.guidewire.restclient.config.InterceptorSetupEx",
      "additionalProperties": false,
      "properties": {
      "interceptorSupplierClass": {
```

```
      "type": "string",
      "$ref": "java.lang.Class",
      "required": true,
      "description": "This is the supplier class, it is passed the properties to get a RequestInterceptor
implementation"
      },
      "properties": {
      "type": "object",
      "description": "This is a map of name/value that is passed to the supplier",
      "additionalProperties": {
      "type": "string"
      }
      }
      }
      }
      },
      "auth": {
      "type": "object",
      "id": "com.guidewire.restclient.config.AuthMethodEx",
      "description": "This is how to authenticate: 'HttpBasic', 'HttpBearer', 'ApiKey', or 'OAuth' are supported,
each has specific attributes based on their 'method' property.",
      "oneOf": [ {
      "$ref": "com.guidewire.restclient.config.ApiKeyAuthEx"
      }, {
      "$ref": "com.guidewire.restclient.config.HttpBasicAuthEx"
      }, {
      "$ref": "com.guidewire.restclient.config.HttpBearerAuthEx"
      }, {
      "$ref": "com.guidewire.restclient.config.OAuthEx"
      } ]
      },
      "basePath": {
      "type": "string",
      "description": "This is the base path to the endpoint."
      },
      "credentialSupplierSupplierClass": {
      "type": "string",
      "$ref": "java.lang.Class",
      "description": "this is a class that implements Credential supplier.  The supplier can be initialized the with
properties below, and is asked for the value of the various credentials, i.e., ApiKeyAuth is used the if paramName is
defaulted it is 'apikey', so the supplier is asked for 'apikey' value. HttpBasicAuth uses username and password.
HttpBearerAuth uses the schema or default to 'bearToken. OAuth uses username, password, clientId, clientSecret."
      },
      "eventHandlers": {
      "type": "array",
      "description": "This is the event handlers that should be applied to each request, RetrySetupEx,
FallbackSetupEx, CircuitBreakerEx.",
      "items": {
      "type": "object",
      "id": "com.guidewire.restclient.config.EventHandlerSetupEx",
      "oneOf": [ {
      "$ref": "com.guidewire.restclient.config.CircuitBreakerSetupEx"
      }, {
      "$ref": "com.guidewire.restclient.config.FallbackSetupEx"
      }, {
      "$ref": "com.guidewire.restclient.config.RetrySetupEx"
      } ]
      }
      },
      "httpClient": {
      "type": "object",
      "id": "com.guidewire.restclient.config.HttpClientSetupEx",
      "description": "This is setup information for ssl or tls.",
      "additionalProperties": false,
      "properties": {
      "clientSupplierClass": {
      "type": "string",
      "$ref": "java.lang.Class",
      "description": "this is a function class that accepts a Map<String,String> and returns an HttpClient object.
Note that if this is declared then the other suppliers are ignored."
      },
      "contextSupplierClass": {
      "type": "string",
      "$ref": "java.lang.Class",
      "required": true,
      "description": "this is a function class that accepts a Map<String,String> and returns an object that
implements SSLSocketFactory"
      },
      "hostnameVerifierSupplierClass": {
```

```
      "type": "string",
      "$ref": "java.lang.Class",
      "required": true,
      "description": "this is a function class that accepts a Map<String,String> and returns an object that
implements HostnameVerifier"
      },
      "properties": {
      "type": "object",
      "description": "this is a map of name/values which is passed to the httpClientSupplier, contextSupplier, and
hostnameVerifierSupplier to get their respective object. to get a CredentialSupplier.",
      "additionalProperties": {
      "type": "string"
      }
      }
      }
      },
      "logLevel": {
      "type": "string",
      "description": "This what to log, default is NONE:, NONE: No logging; BASIC: method, URL, response status, and
execution time; HEADERS: the BASIC information and request/response headers; FULL: the HEADERS information, body, and
metadata for request/response",
      "enum": [ "NONE", "BASIC", "HEADERS", "FULL" ]
      },
      "loggerName": {
      "type": "string",
      "description": "This to override the logger for this end point; you may want to log different end points to
different loggers.  Note that the logger must be at DEBUG to enable the additional logLevel messages"
      },
      "mapperSupplierClass": {
      "type": "string",
      "$ref": "java.lang.Class",
      "description": "This is an ObjectMapper, but default the standard mapper is used."
      },
      "properties": {
      "type": "object",
      "description": "this is a map of name/values which is passed to the CredentialSupplierSupplier to get a
CredentialSupplier.",
      "additionalProperties": {
      "type": "string"
      }
      },
      "timeout": {
      "type": "object",
      "id": "com.guidewire.restclient.config.TimeoutSetupEx",
      "description": "This is timeout information.",
      "additionalProperties": false,
      "properties": {
      "connectTimeoutMillis": {
      "type": "integer",
      "required": true
      },
      "readTimeoutMillis": {
      "type": "integer",
      "required": true
      }
      }
      },
      "traceActionsClass": {
      "type": "string",
      "$ref": "java.lang.Class",
      "description": "This is whether to deal with trace headers, but default the standards headers are forwarded."
      }
      }
```


Using REST API client library

To start using the REST API Client, follow the steps in the “Getting started” on page 11 guide. Use the information in this section to learn how to use the following library features:

- “Consuming a REST service” on page 28
- “Using externalized configuration files” on page 29
- “Using the fault tolerance features” on page 30
- “Using API security” on page 33

- “Adding interceptors” on page 35

Consuming a REST service

The REST API library follows the functional programming paradigm along with the *Decorator* pattern to construct a set of features around REST calls. The examples in this section use the `StoreApi` from PetStore https://petstore.swagger.io/ REST service to illustrate how you can use the REST API client library with your own *OpenAPI Spec* compliant service.

How to make a REST call with a default set of fault tolerance parameters

In the following example, the configuration builder builds the configuration using the default settings.

Example

```
Map<String, Integer> response = Config.builder().build()
  .buildAPI(StoreApi.class)
  .getInventory();
```

How to make a REST call with custom fault tolerance parameters

In the following example, the `maxAttempt` method overwrites the default retry attempts (3) and sets it to 5.

Example

```
Map<String, Integer> response = Config.builder()
  .eventHandler(RetrySetup.builder()
    .maxAttempt(5)
    .build())
  .build()
  .buildAPI(StoreApi.class)
  .getInventory();
```

How to externalize the Rest API client configuration

You can provide the parameters in external files in JSON or YAML format. To retrieve the parameters use the *Config* object and one of the *parseJson(String cfgString)* and *parseYaml(String cfgString)*. For more information, see “Using externalized configuration files” on page 29.

Example

```
Config config = Config.parseYaml(configString).build();
Map<String, Integer> response = config.buildAPI(StoreApi.class)
  .getInventory();
```

How to order the fault tolerance decorators

The order of the fault tolerance decorators is important. In the following example, the *fallback* is wrapped within a *retry* because the *fallback* is defined first. Even though `maxAttempt` is 5, only one *retry* attempt is made because the *fallback* event handler always successfully returns an empty map for any *Throwable* exception.

Example

```
Config config = Config.builder()
  .eventHandler(FallbackSetup.builder()
    .exceptionType(Throwable.class)
    .exceptionHandler(throwable -> Collections.emptyMap())
    .build())
  .eventHandler(RetrySetup.builder()
    .maxAttempt(5)
    .build())
  .build();
  Map<String, Integer> response = config.buildAPI(StoreApi.class).getInventory();
```

Using externalized configuration files

Instead of programmatically setting up the configuration values, you can store them in external files. Use the `parseYaml()` or `parseJson()` methods to decrypt the externally configured values or fetch them from a more secure location.

The following example illustrates the usage of the `CredentialSupplier` map that the builder accepts during the build to get the individual credentials from the selected provider. These values remain set in this configuration object, and the secrets are kept encrypted.

Example

```
Config config = Config.parseYaml(configString)
  .credentialSupplier(directoryCredSupplier)
  .build()
```

Defining the configuration parameters in a YAML file

The following example shows an external YAML file that stores the parameters of the `OAuth` authentication object. The `credentialSupplierSupplierClass` property provides the name of a custom supplier class `example.restclient.PBECipherSupplier`:

Example

```
---
basePath: "http://localhost:49947"
auth:
  method: "OAuth"
  flow: "password"
  tokenUrl: "http://localhost:49947/api/v1/token"
  scopes: "openid"
  credentials:
    clientId: "0oarlaum9pzLAAzFK0h7"
    username: "username"
    password: "6jp4DQ5+2AnasaX10aWykg=="
credentialSupplierSupplierClass: "example.restclient.PBECipherSupplier"
properties:
  salt: "0oarlaum9pzLAAzFK0h7"
eventHandlers:
- type: "retry"
  name: "IO"
  maxAttempts: 4
  backoff:
    type: "random"
  retryOnExceptionPredicateClass: "example.restclient.faulttolerance.IOExceptionRetryPredicate"
- type: "retry"
  name: "Auth"
  maxAttempts: 1
  retryOnExceptionPredicateClass: "example.restclient.faulttolerance.AuthExceptionRetryPredicate"
  intervalFunctionClass: "example.restclient.faulttolerance.ImmediateInterval"
- type: "retry"
  name: "Default"
  backoff:
    type: "exponential"
  retryOnExceptionPredicateClass: "example.restclient.faulttolerance.DefaultRetryPredicate"
- type: "fallback"
  exceptionTypes:
  - "java.io.IOException"
  - "feign.FeignException"
  exceptionHandlerClass: "example.restclient.faulttolerance.SuspendFallback"
- type: "circuitBeaker"
```

Implementing the credential supplier class

The following example shows the implementation of `example.restclient.PBECipherSupplier` class:

Example

```
package example.restclient;
```

```
import gw.restclient.config.Config;
import gw.restclient.config.CredentialSupplierSupplier;
import gw.restclient.util.PBECipher;

import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.Map;

public class PBECipherSupplier implements CredentialSupplierSupplier {
public static final String PASS_PHRASE = "some string of characters used as a passPhrase";
public static final String SALT = "salt";

@Override
  public Config.CredentialSupplier apply(Map<String, String> properties) {
     String passPhrase = System.getProperty("restClient.PBE.passPhrase", PASS_PHRASE);
     PBECipher cipher = new PBECipher(passPhrase,
                                      properties.get(SALT).getBytes(StandardCharsets.ISO_8859_1));
       return (key, value) -> {
           if (value == null) {
              return null;
           }
           switch(key) {
            case "username":
            case "clientId":
              return value;
            case "password":
            case "clientSecret":
            case "BearerToken":
            default: // for ApiKeyAuth.paramName and HttpBearerAuth.scheme
              return cipher.decrypt(Base64.getDecoder().decode(value));
          }
     };
  }
}
```

Using the fault tolerance features

The REST client includes a set of fault tolerance features that can be applied to a REST call.

How to use the retry

This is used to retry executing a piece of code multiple times if a particular type of exception is thrown. Configure the `retry` properties using the following settings:

Table 4.8-1 Configurations

Setup property	Default value	Description
`maxAttempts`	3	The maximum number of attempts before failing
`exponentialBackoff`	None	An exponential backoff function to modify the waiting interval after each failure; If not specified, the `intervalFunction` is used.
`intervalFunction`	Always returns 500ms	A function to modify the waiting interval after a failure
`retryOnResultPredicate`	Always false for any result value	A predicate determining if a result has to be retried; It must return true, if the result has to be retried, otherwise it must return false.
`retryOnExceptionPredicate`	Always true for any *Throwable*	A predicate determining if an exception has to be retried; It must return true, if the exception has to be retried, otherwise it must return false.

The following example shows how to create the configuration:

Example

```
Config config = Config.builder()
  .eventHandler(RetrySetup.builder()
    .retryOnResultPredicate(Objects::nonNull)
    .retryOnExceptionPredicate(e -> e instanceof IOException)
    .maxAttempts(7)
    .build();
```

To specify multiple `retry` attempts, wrap the *retry* within another *retry*, as shown in the following example. This results in executing the throw statement 10 times:

Example

```
Config config = Config.builder()
  .eventHandler(RetrySetup.builder().maxAttempts(5).build())
  .eventHandler(RetrySetup.builder().maxAttempts(2).build());
  .build();
```

How to use CircuitBreaker

You invoke circuit breaker when API calls failed or appeared to be slow. The circuit breaker starts short-circuiting API calls when a failure rate or slow call rate threshold is reached. A circuit breaker instance is used by multiple REST calls to determine the state of the circuit breaker. An instance is uniquely identifiable by its name within a JVM.

Table 4.8-2 States

State	Description
CLOSED	The starting state for circuit breaker, failure rate or slow call rate threshold is calculated.
OPEN	Circuit breaker transition to this state once the failure rate or the slow call rate threshold is reached. No API calls are executed. Transition to HALF-OPEN state once wait interval has elapsed.
HALF-OPEN	Permits a configurable number of calls to see if the backend is still unavailable or has become available again. Once wait interval has elapsed, it transitions to CLOSED state when failure rate or slow call rate is below threshold. Otherwise, it transitions to OPEN state.

Table 4.8-3 Configuration

Setup property	Default value	Description
`failureRateThreshold`	50	Configures the failure rate threshold in percentage.
`slowCallRateThreshold`	100	Configures the slow call rate threshold in percentage.
`slowCallDurationThreshold`	60,000 ms	Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls.
`permittedNumberOfCallsInHalfOpenState`	10	Configures the number of permitted calls when the CircuitBreaker is half open.
`slidingWindowType`	`COUNT_BASED`	Configures the type of the sliding window which is used to record the

Setup property	Default value	Description
		outcome of calls when the CircuitBreaker is closed.
`slidingWindowSize`	100	Configures the size of the sliding window which is used to record the outcome of calls.
`minimumNumberOfCalls`	100	Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate or slow call rate.
`automaticTransitionFromOpenToHalfOpenEnabled`	`false`	If set to true it means that the CircuitBreaker automatically transitions from open to half-open state and no call is needed to trigger the transition.
`waitIntervalFunctionInOpenState`	Always returns 500ms	A function to modify the waiting interval in open state (when circuit breaker short-circuiting the calls).
`writableStackTraceEnabled`	`true`	Enables writable stack traces. Set to false to reduce log spam when the circuit breaker is open as the cause of the exceptions is already known.
`recordException`	Always `false` for any exceptions	A predicate determining if an exception has to be recorded; It must return true, if the exception has to be recorded, otherwise it must return false.

Define circuit breaker with default name and configurations

Example

```
Config config = Config.builder()
  .eventHandler(CircuitBreakerSetup.builder().build());  // default name is GW_CIRCUITBREAKER
  .build();
```

Define circuit breaker with custom name and configurations

Example

```
Config config = Config.builder()
  .eventHandler(CircuitBreakerSetup.builder()
    .name("customeCircuitBreakerName")
    .failureRateThreshold(50)
    .slowCallRateThreshold(100)
    .slowCallDurationThreshold(Duration.ofMillis(60000))
    .permittedNumberOfCallsInHalfOpenState(10)
    .slidingWindowType(CircuitBreakerSetup.SlidingWindowType.COUNT_BASED)
    .slidingWindowSize(100)
    .minimumNumberOfCalls(100)
    .waitIntervalFunctionInOpenState(x -> 500L)
    .automaticTransitionFromOpenToHalfOpenEnabled(false)
    .recordException(e -> e instanceof IOException)
    .writableStackTraceEnabled(true)
    .build())
  .build();
```

How to use fallback

The following example demonstrates how to use fallback to recover from an exception by returning a specific value:

Example

```
Config config = Config.builder()
  .eventHandler(FallbackSetup.builder()
    .exceptionType(IOException.class)
    .exceptionHandler(e -> "fallback") // This results in returning the String "fallback".
    .build();
```

How to use timeout

The following example demonstrates how to set a timeout for a specified piece of code to execute:

Example

```
Config config = Config.builder()
  .timeout(TimeoutSetup.builder()
    .connectTimeoutMillis(1000)
    .readTimeoutMillis(500)
    .build())
.build();
```

Using API security

The REST client includes several API authentication methods that can be easily configured for your API calls. The supported authentication methods are *Basic*, *Bearer*, *ApiKey*, and *OAuth 2.0*.

How to configure Basic authentication

Example

```
Config config = Config.builder()
    .auth(HttpBasicAuth.builder()
      .username("user")
      .password("password")
      .build())
    .build();
```

How to configure Bearer authentication

Example

```
Config config = Config.builder()
  .auth(HttpBearerAuth.builder()
    .bearerToken("tokenValue")
    .scheme("Bearer")
    .build())
  .build();
```

How to configure ApiKey authentication

API keys are supplied by client users and applications calling REST APIs to track and control how the APIs are used. For example, to meter access and prevent abuse or malicious attack. API keys include a key ID that identifies the client responsible for the API service request. This key ID is not a secret, and must be included in each request. API keys can also include a confidential secret key used for authentication, that only the client and the API service know.

Example

```
Config config = Config.builder()
  .auth(ApiKeyAuth.builder()
    .apiKey("key")
    .location(ApiKeyAuth.Location.header)
    .paramName("ApiKey")
```

```
    .build())
  .build();
```

How to configure OAuth 2.0 authentication

Configuring the *OAuth 2.0* client credentials flow

The OAuth 2.0 client credentials grant flow permits a web service (confidential client) to use its own credentials, instead of impersonating a user, to authenticate when calling another web service. In this scenario, the client is typically a middle-tier web service, a daemon service, or a web site. The client credentials flow is the simplest OAuth 2 grant, with a server-to-server exchange of your application's *clientId*, *clientSecret* for an OAuth application access token.

Example

```
Config config = Config.builder()
  .auth(OAuth.builder()
    .tokenUrl("tokenUrl")
    .redirectUrl("redirectUrl")
    .scopes("scope1 scope2")
    .flow(OAuth.OAuthFlow.application)
    .tokenStore(new OAuthTokenStore.Default())
    .credentials(OAuthCredentials.builder()
      .clientId("clientId")
      .clientSecret("clientSecret")
      .build())
    .build())
  .build();
```

Configuring the *OAuth 2.0* password grant

The password grant type is a way to exchange a user's credentials for an access token. The password grant involves only one step: the application presents a traditional username and password login to collect the user’s credentials and makes a POST request to the server to exchange the password for an access token.

Example

```
Config config = Config.builder()
  .auth(OAuth.builder()
    .tokenUrl("tokenUrl")
    .redirectUrl("redirectUrl")
    .scopes("openid")
    .flow(OAuth.OAuthFlow.password)
    .tokenStore(new OAuthTokenStore.Default())
    .credentials(OAuthCredentials.builder()
      .clientId("clientId")
      .username("username")
      .password("password")
      .build())
    .build())
  .build();
```

How to configure Mutual TLS

With mutual authentication, a connection can occur only when the client trusts the server's digital certificate and the server trusts the client's certificate. The exchange of certificates is carried out by means of the Transport Layer Security (TLS) protocol. The REST client supports two-way authentication which can be configured by using the *config* object

Example

```
Config config = Config.builder()
  .basePath(basePath)
  .ssl(SSLSetup.builder()
    .socketFactory()
    .hostnameVerifier()
  .build())
.build();
```

Adding interceptors

This example applies to IS internal services that use Zipkin. This interceptor forwards tracing values from MDC to headers if they have not been set previously:

Example

```
Config config = Config.builder()
  .requestInterceptor(new ForwardingInterceptor(Action implementation))
  .build();
```

chapter 5

REST API client plugin

The client plugin generates REST API for making outbound HTTP requests to external REST services. To use the plugin, users import the plugin into their projects. Developers use IntelliJ or other development environments that are integrated with Gradle. The plugin provides access to code generation tasks that the developers use within their Gradle build scripts. Guidewire recommends using IntelliJ, because IntelliJ with its Gradle integration recognizes the tasks and makes using them easier.

REST API client plugin tasks

The client plugin provides Gradle Codegen plugin tasks that you use to generate REST API:

Using *gwUrlDownload*

Downloads the OpenAPI for a given RESTful service endpoint and saves it as a local JSON file. This local file can be used as a source by the *gwRestCodegen* task. The *gwUrlDownload* task accepts the following parameters:

Table 5.1-1 Parameters

Params	Description
`sourceUrl`	The REST service endpoint OpenAPI Spec URL
`target`	The file path to save the OpenAPI Spec content

Example

```
gradle gwUrlDownload -DsourceUrl="https://petstore3.swagger.io/api/v3/openapi.json" -Dtarget="petshop.json"
```

Using *gwConfigCodegen*

Generates a configuration file that can be updated by the developers to customize the REST client code generation. The file is stored in the project root folder.

Table 5.1-2 Parameters

Params	Description
`packageName`	The name of the Java package that the generated code has to be associated with

Params	Description
`projectName`	The name of the Gradle subproject and folder that the generated client has to be placed in

Example

```
gradle gwConfigCodegen -DpackageName="com.example" -DprojectName="petshop"
```

Using *gwRestCodegen*

Generates a REST HTTP client project, Java source code, tests and support files.

Table 5.1-3 Parameters

Params	Description
`packageName`	The REST client Java package name to use
`source`	The source can be defined as one of the following: • a REST service endpoint OpenAPI spec URL • a local file path to the OpenAPI spec JSON file
`target`	The output folder where the generated client has to be saved
`configFile` (optional)	The file path to the YAML OpenAPI Codegen configuration settings file
`isInsuranceSuite` (optional)	A flag indicating that the task runs within an InsuranceSuite project

Common usage:

Example

```
gradle gwRestCodegen -Dtarget="~/petshop" -Dsource="https://petstore3.swagger.io/api/v3/openapi.json"
```

Generation that uses a configuration file

Use the `gwConfigCodegen` task to generate the configuration file:

Example

```
gradle gwRestCodegen -Dtarget="~/petshop" -Dsource="https://petstore3.swagger.io/api/v3/openapi.json" -
DconfigFile="petshop/codegen.config.yaml"
```

Using subproject auto-generated tasks

Tasks	Description
`restDownload`	For URL-based endpoint sources, this all-in-one task downloads the REST API definition for that endpoint, generates client code and compiles a JAR library. This task is not present if the endpoint source is file-based.
`restConfig`	This task auto-generates `codegen.config.yaml` file with default settings inside your endpoint sub-project folder. See “Using *gwConfigCodegen*” on page 37 for more information. If a configuration of the code generation is necessary, then first generate the configuration file and then the code.

Tasks	Description
`restCodegen`	This task generates code that is based on the given endpoint definition and configuration settings. See “Using *gwRestCodegen*” on page 38 task. This task automatically compiles and generates a JAR library for the endpoint in the build directory of the subproject.

Using REST API client plugin

The following topics provide examples of how to generate REST API code using the client plugin tasks.

How to generate code using URL-based source with default configuration

Run the code generation task:

```
gradle restDownload
```

How to generate code using URL-based source with customized configuration

1. Generate a configuration file

   ```
   gradle restConfig
   ```

2. Update the configuration `codegen.config.yaml` file
3. Run the code generation task

   ```
   gradle restDownload
   ```

How to generate code using file-based source with default configuration

Run the code generation task:

```
gradle restCodegen
```

How to generate code using file-based source with customized configuration

1. Generate a configuration file

   ```
   gradle restConfig
   ```

2. Update the configuration `codegen.config.yaml` file
3. Run the code generation task

   ```
   gradle restCodegen
   ```