User Guide

Table of Contents

Examples

Configuration

Working example

Working example

We provide demonstrating the usage of the library, depending on the context.

Standalone usage

The example demonstrates the usage of the library accessing a process engine via REST from an arbitrary SpringBoot application. The client executes the following steps:

Table 1. Timing overview of the example

Initial offset	Repeat	Invoked method
8.0 sec	-	Get deployed processes
10.0 sec	5 sec	Start process
12.5 sec	5 sec	Send signal
13.0 sec	5 sec	Correlate message

How does it work

The application uses the library by adding it to the classpath via Apache Maven dependency. That is:

```
<dependency>
  <groupId>org.camunda.bpm.extension.rest</groupId>
  <artifactId>camunda-rest-client-spring-boot-starter</artifactId>
  <version>${project.version}</version>
</dependency>
```

In order to activate the library, the `@EnableCamundaRestClient` has been put on the configuration class of the application. The interesting part is now the `ProcessClient` component. This Spring Component has several methods marked with `@Scheduled` annotation to demonstrate the time-based execution of desired functionality. To do so, the component has two injected resources, both marked with the `@Qualifier("remote")` annotation. This annotation indicates that the remote version of the Camunda API services are used.

In order to configure the library, a block of properties e.g. in `application.yml` is required. The values specify the location of the remote process engine:

```
feign:
  client:
    config:
      remoteRuntimeService:
        url: "http://localhost:8083/engine-rest/"
      remoteRepositoryService:
        url: "http://localhost:8083/engine-rest/"
      remoteExternalTaskService:
        url: "http://localhost:8083/engine-rest/"
```

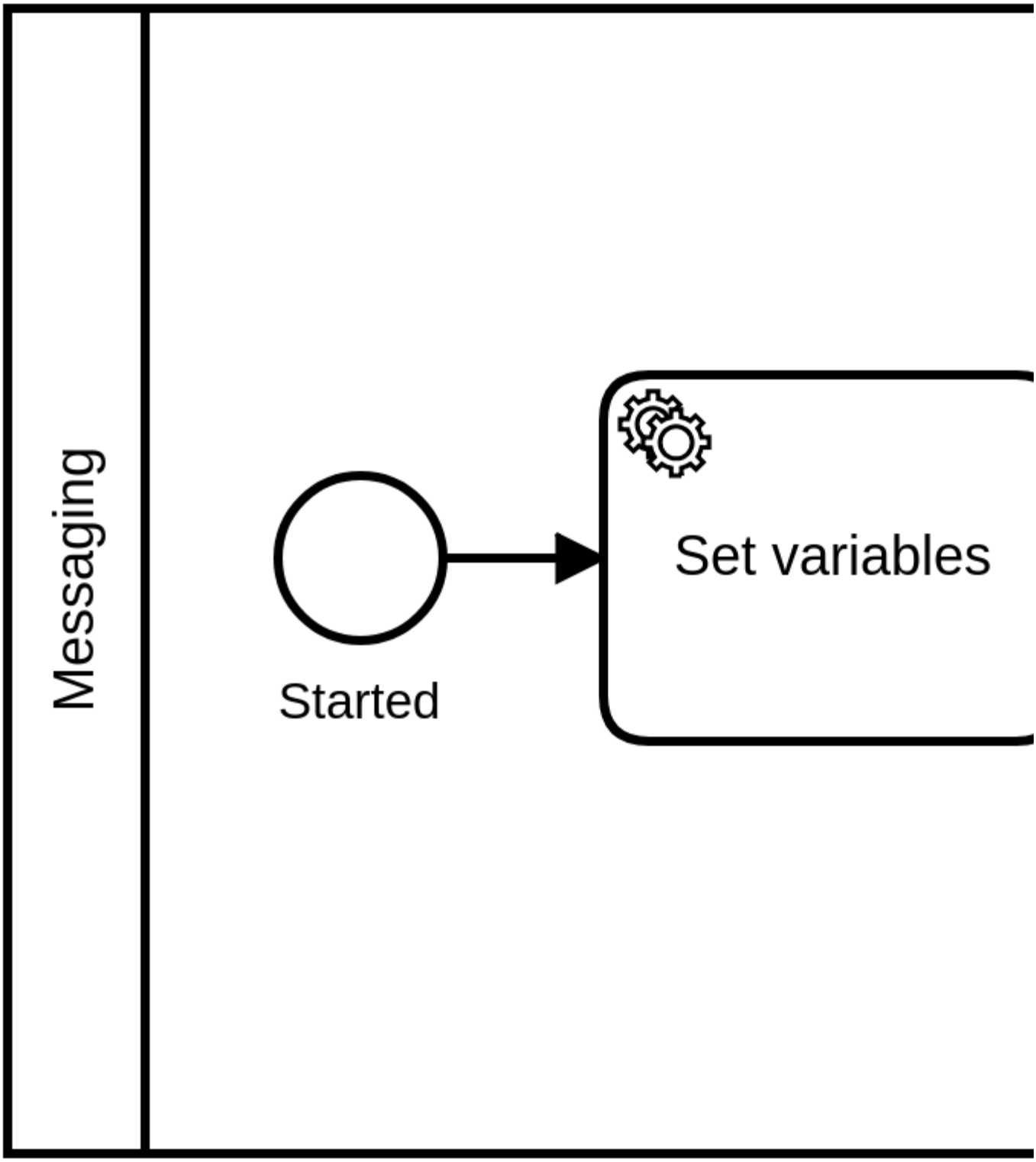
To run this example, you will need the server part from the next example. To activate the server part only, please run from command line:

```
mvn clean install
mvn -f examples/example -Prun
mvn -f examples/example-provided -Prun-server-only
```

Usage inside a process application

The example demonstrates the usage of the library for accessing a process engine via REST from a Camunda process application. The key difference to the previous example is that the required Camunda classes are already present on the classpath and an engine is initialized and is running.

Imagine the process engine has the following process deployed:



The client (running technically in the same JVM, but accessing the engine via REST) again executes the following steps:

Table 2. Timing overview of the example

Initial offset	Repeat	Invoked method
8.0 sec	-	Get deployed processes
10.0 sec	5 sec	Start process

12.5 sec	5 sec	Send signal
13.0 sec	5 sec	Correlate message

How does it work

The application uses the library by adding it to the classpath via Apache Maven dependency. That is:

```
<dependency>
  <groupId>org.camunda.bpm.extension.rest</groupId>
  <artifactId>camunda-rest-client-spring-boot-starter-provided</artifactId>
  <version>${project.version}</version>
</dependency>
```

Note Please note that we use a different starter. The suffix `provided` in the artifact name indicates that the engine is already a part of the application and doesn't need to be put on classpath.

In order to activate the library, the `@EnableCamundaRestClient` has been put on the configuration class of the application. The interesting part is now the `ProcessClient` component. This Spring Component has several methods marked with `@Scheduled` annotation to demonstrate the time-based execution of desired functionality. To do so, the component has two injected resources, both marked with the `@Qualifier("remote")` annotation. This annotation indicates that the remote version of the Camunda API services are used.

In order to configure the library, a block of properties e.g. in `application.yml` is required:

```
feign:
  client:
    config:
      remoteRuntimeService:
        url: "http://localhost:8083/engine-rest/"
      remoteRepositoryService:
        url: "http://localhost:8083/engine-rest/"
      remoteExternalTaskService:
        url: "http://localhost:8083/engine-rest/"
```

Support Matrix

Support Matrix

Here are currently implemented methods.

Runtime Service @ 0.0.4

- `# startProcessInstanceByKey`
- `# startProcessInstanceById`
- `# correlateMessage`
- `# createMessageCorrelation`
- `# signal`
- `# signalEventReceived`
- `# createSignalEvent`
- `# getVariable`
- `# getVariables`
- `# setVariable`
- `# setVariables`
- `# removeVariable`
- `# removeVariables`
- `# getVariableTyped`
- `# getVariablesTyped`
- `# setVariableTyped`
- `# setVariablesTyped`
- `# getVariableLocal`
- `# getVariablesLocal`
- `# setVariableLocal`
- `# setVariablesLocal`
- `# removeVariableLocal`
- `# removeVariablesLocal`

- `# getVariableTypedLocal`
- `# getVariablesTypedLocal`
- `# setVariableTypedLocal`
- `# setVariablesTypedLocal`

RepositoryService @ 0.0.4

- `# createProcessDefinitionQuery`

ExternalTaskService @ 0.0.4

- `# complete`

Logging

Logging

OpenFeign library used in the `camunda-rest-client-spring-boot` has a high-configurable logging facility.

In order to configure it, a block of properties e.g. in `application.yml` is required:

```
logging:
  level:
    org.camunda.bpm.extension.rest.client.RuntimeServiceClient: DEBUG
    org.camunda.bpm.extension.rest.client.RepositoryServiceClient: DEBUG
    org.camunda.bpm.extension.rest.client.ExternalTaskServiceClient: DEBUG
```

In order to enable Request/Response logging, you need to configure additional Feign logging by providing a factory bean:

```
import feign.Logger;

@Configuration
public class MyConfiguration {
    /**
     * Full debug of feign client, including request/response
     */
    @Bean
    public Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

Error Decoding

Error Decoding

The `camunda-rest-client-spring-boot` uses HTTP to access a remote Camunda REST API. If any errors occur on this access, Camunda REST API will send with corresponding HTTP error code and embed information about the error into http response. `camunda-rest-client-spring-boot` tries to parse this response and throw an exception on the client side similar to the original exception thrown on the remote Camunda Engine.

By default, the library tries to decode HTTP codes and will throw a `RemoteProcessEngineException`. If the response decoding was successful, the *cause* of the thrown `RemoteProcessEngineException` will be the instance of the exception class thrown on remote Camunda engine and the *reason* of the latter exception will be the original reason from the server.

If anything goes wrong on HTTP error decoding, the `RemoteProcessEngineException` will contain a generic message extracted from the REST call. If the error decoding is deactivated, `FeignException` is wrapping any exception occurring during the remote access.

Configuration

By default, the HTTP error decoding is switched on and the libary reacts on HTTP codes 400 and 500. Those defaults can be changed by setting the following properties.

In order to configure it, a block of properties e.g. in `application.yml` is required. Here are the defaults:

```
camunda:
  rest:
    client:
      error-decoding:
        enabled: true
        http-codes: 400, 500
```

Tip: If you are using the remote version of the `ExternalTaskService` this will report HTTP 404 if you try to complete a non-existing task. By changing the `camunda.rest.client.error-decoding.http-codes` property you can cover this response too.