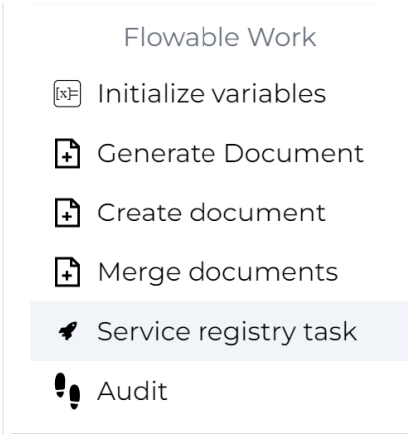
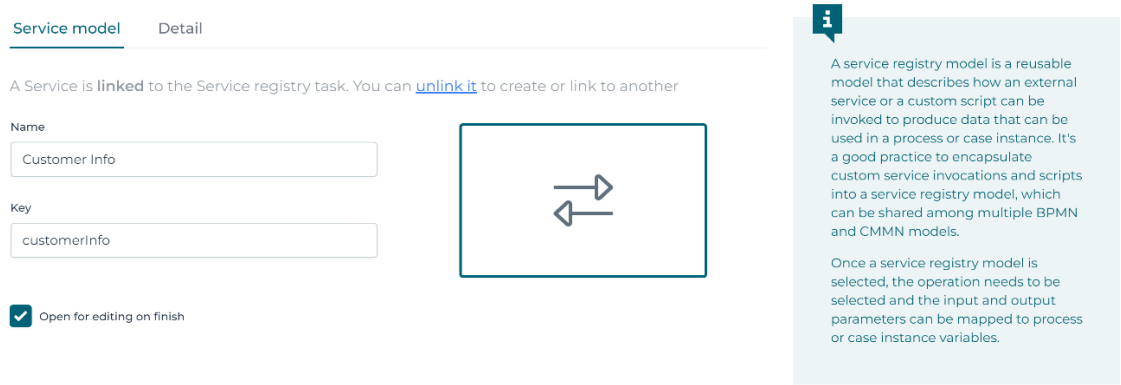
Introduction to the Service Registry

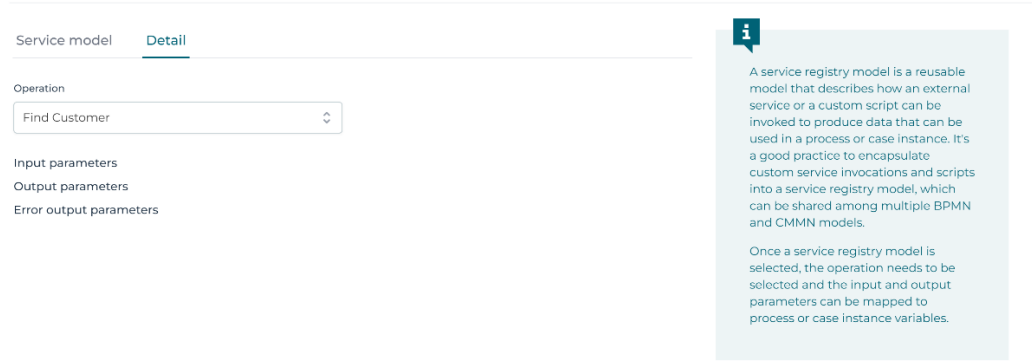
Service Definitions and Service Operations

After a service model is created, it can be used in process or case models. To do so, drag the *Service registry task* to the canvas from the Flowable design palette:



In the Service Registry Task - General attribute panel, the service and one of its operations can be selected (one service can expose multiple operations):





At runtime, the process or case invokes the service configured in the referenced *service definition*.

Scripting

Introduction

Scripting in Flowable allows you to execute logic programmatically using low-code capabilities. A script consists of a set of instructions that are executed during runtime.

Scripts within Case and Process Models

You can use scripts directly in case and process models by using the Script Task to execute scripting logic. This method offers a powerful and efficient way to execute programmatic operations.

Within the script context, the entire application context is available, enabling a wide range of operations. You can leverage various Flowable APIs such as `runtimeService`, `historyService`, `cmmnRuntimeService` and more.

To easily access API methods, you can utilize the Flowable Scripting API (`flw`).

To add a script, configure it directly within the Script Task of your BPMN and CMMN models.

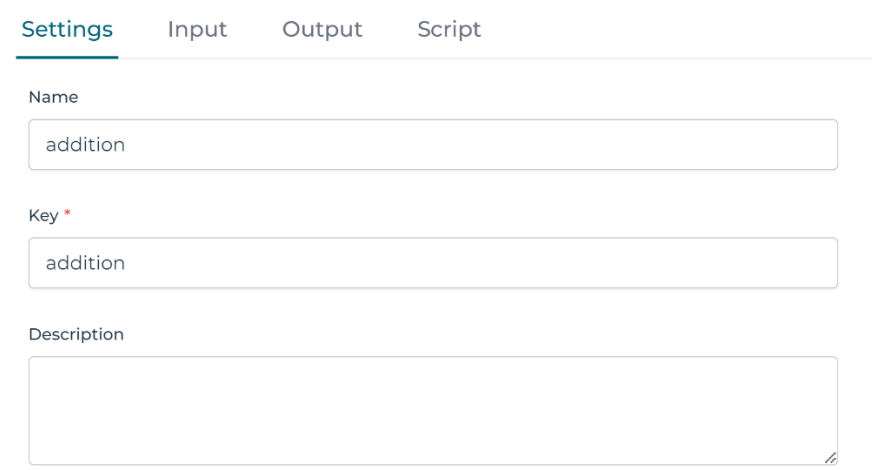
Below is a basic example that demonstrates adding two numbers (case or process variables), performing an addition operation, and storing the result as a JSON object in the process.

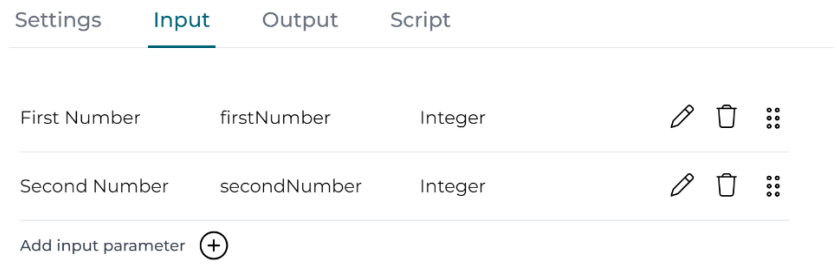
```
// Get variables from Variable Container
var a = flw.getInput("firstNumber");
var b = flw.getInput("secondNumber");

// Perform operation
var c = a + b;

// Creation of the JSON object
var jsonObject = flw.json.createObject();
jsonObject.putInteger('firstNumber', a);
jsonObject.putInteger('secondNumber', b);
jsonObject.putString('operation', 'addition');
jsonObject.putInteger('result', c);

// Set variable to the Variable Container
flw.setOutput("result", jsonObject);
```

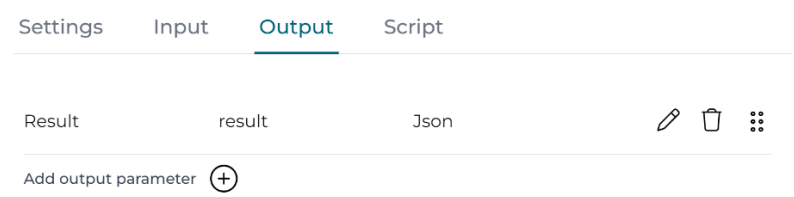
Scripts with the Service Registry

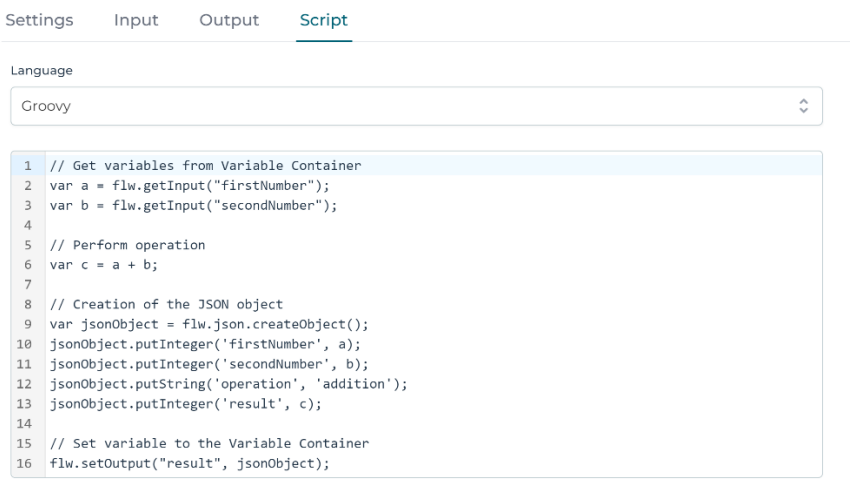
For creating reusable and abstracted scripts, you can utilize the Service Registry engine by configuring a Service Registry Model with a `Service Type` of `Script`.

You can add the same operation mentioned above, "addition", to a Service Registry Model by defining an operation called `addition`:









Note that the script logic remains identical to the script provided earlier. However, you need to explicitly set the input and output parameters as part of the Service Registry operation.

Once configured, you can use the Service Registry Model with the `Service Registry Task` in your case and process models. After selecting the model, the available operations, such as `addition`, are displayed in the Service model's Detail section.

The input and output parameters are pre-populated and can be defined using expressions:

Request Response Handlers

When integrating with REST APIs using the `Service Registry` with `Service Type` set as `REST`, the structure of the requests and responses may not always match the expectation of the process or case.

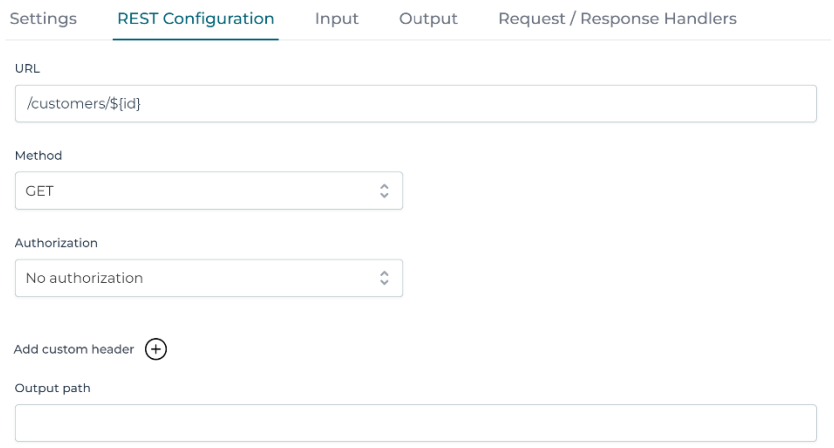
To address this issue, Flowable provides support for `Request / Response Handlers`. These handlers can be configured using scripts or expressions to manipulate data and align it with the required request/response format.

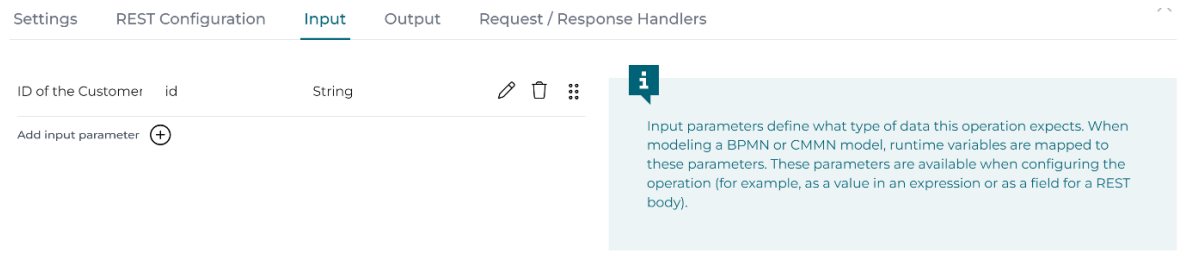
Scripts as Request and Response Handlers

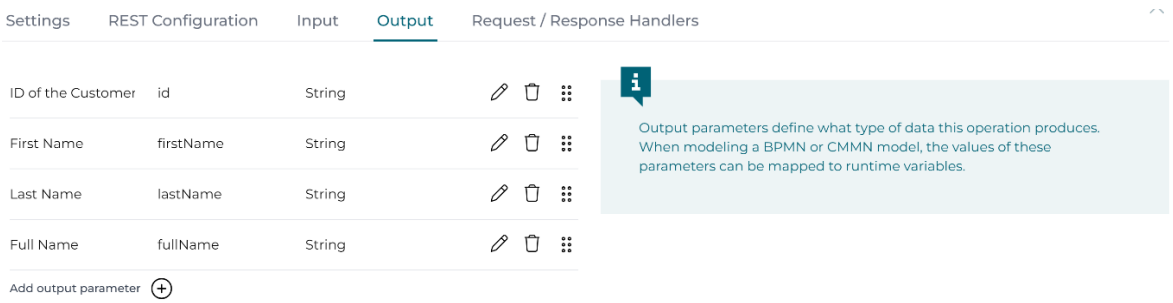
Consider an example where an API returns `Customer` objects in the following format:

```
{
  "id" : "johndoe",
  "firstName": "John",
  "lastName": "Doe"
}
```

To retrieve a customer object using the Service Registry, configure a service operation with a GET API call to the Customer Service:







In the screenshot above, the API expects an `id` as input and returns the `id`, `firstName` and `lastName` of the customer. However, let's assume we also require a new field called `fullName`, which should be calculated based on the `firstName` and `lastName` fields. In this case, you can configure a `Response handler` script:

```
// Retrieve Response Body and transform to JSON object
var response = flw.json.stringToJson(flwHttpResponse.getBody());

// Get attributes from response body
var firstName = response.path("firstName").asString();
var lastName = response.path("lastName").asString();

// Add a new attribute to the response body
var fullName = firstName + " " + lastName;
response.putString("fullName", fullName);

// Set response body
var jsonResponse = flw.json.jsonToString(response);
flwHttpResponse.setBody(jsonResponse);
```

In the above example, the `Response handler` takes the response, transforms it to a JSON object, calculates the `fullName` field based on the `firstName` and `lastName`, adds it to the response, and sets the modified response body using `flwHttpResponse`.

The Service Registry executes the API, applies the `Response handler`, and parses the result for the output parameters. Since the `fullName` field is now part of the response, it will be returned seamlessly as an output parameter.

Expression-based Request and Response Handlers

Instead of using a script, it is also possible to use expressions instead. This allows to execute a custom Java method.

It is not necessary to implement a specific interface, the method is executed as is and doesn't expect a specific return type.

In the expression the variables `flwHttpRequest`, `flwHttpResponse`, `flwServiceOperation` and `flwServiceDefinitionModel` are available.

An example for such an expression is:

```
${myCustomUtil.addFullNameToBody(flwHttpResponse)}
```

While the method needs then to implement the signature: `void addFullNameToBody(HttpResponse httpResponse)`.

NOTE

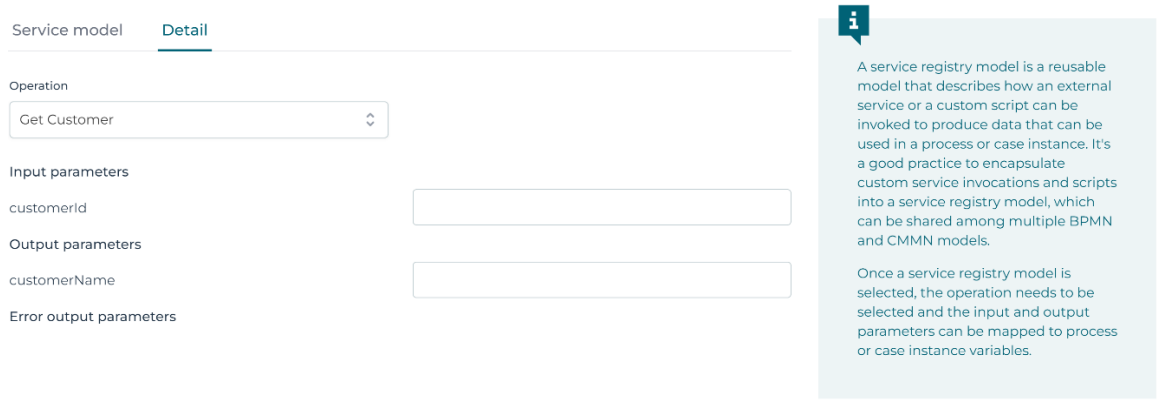
- `flwHttpRequest` is an instance of the class `org.flowable.http.common.api.HttpRequest`
- `flwHttpResponse` is an instance of the class `org.flowable.http.common.api.HttpResponse`
- `flwServiceOperation` is an instance of the class `com.flowable.serviceregistry.api.repository.ServiceOperation`
- `flwServiceDefinitionModel` is an instance of the class `com.flowable.serviceregistry.api.repository.ServiceDefinitionModel`

Parameter Mappings

A service operation has zero or more input parameters and zero or more output parameters as part of its definition. These parameters define what kind of data the service expects when invoked and how the result looks like. For REST services there is an additional zero or more error output parameter mappings. This mapping is only applied if the request resource responded with a status code of 3xx, 4xx or 5xx. If an error occurs and an *error output parameter* mapping is defined, the *output parameter* mapping is ignored. However, it does not define how this data gets mapped into the process or case. As service definitions get

reused between different models, this means that the same service can get and produce data in a different way depending on the **parameter mapping**.

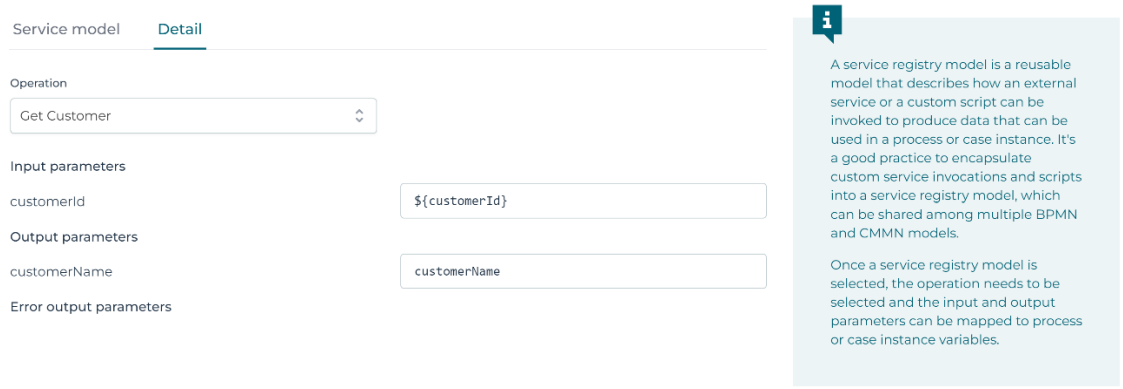
The options for the parameter mapping are defined by the service definition and can be configured through the *input parameters* and *output parameters* attributes. The input and output parameters can be viewed and configured from the Service Registry Task - General panel under the Service model attribute. In the Detail section the parameters defined in the service definition are displayed.



NOTE

In older versions, the technical name is also displayed in this UI and the UI's below. This has been removed in later versions, as it added little value.

Suppose that the service operation has one input and one output parameter, the Detail section of the Service Model property of the Service Registry Task could look like:



When no parameters are defined, no mapping is needed.

Input parameters define what kind of data the service expects when it is invoked. Depending on the type of the parameter, the following values are supported for **mapping** of the **input parameter**:

NOTE

In the list below, when an *expression* is mentioned, this expression has access to all variables of the current process instance or case instance. Also the uses of `${parent.var}` and `${root.var}` are supported. In addtion you can use `${scope.var}` to reference the current instance (case or process). You can use `${scope.definition.attr}` to access any attribute of the current instance definition.

String

- A literal value that is passed as-is.
- An expression. This could be simply to concatenate variables (e.g., `${var1}-${var2}`) or to call a bean (e.g., `myBean.doSomething(someVariable)`).

Integer | Long | Double

- A literal value that is passed as-is (e.g., 123, 12.3, etc.).
- An expression that resolves to a numerical value.

Boolean

- A literal 'true' or 'false'.
- An expression that resolves to a Boolean value.

Date

- A literal ISO8601 formatted text string.
- An expression that resolves to a date (`java.util.Date` or a Joda-time `LocalDate` or `LocalDatetime`).

JSON

- A literal JSON text string (e.g., `{ 'field': 'value' }`)
- An expression that resolves to a JsonNode instance.

Array

- A literal JSON array string or a comma-separated list of text values.
- An expression that resolves to an ArrayNode instance.

In all cases, when a default value is provided as part of the service definition, the default value is used if no data is passed for that input parameter.

IMPORTANT

The input parameter mapping defines how process or case instance data is mapped to the input parameters of the service. In such a mapping, variable values (typically expressions) are referenced as the value for the parameter. The output parameter mapping, on the other hand, defines how the data produced by calling the service maps back into the process or case instance.

For an output mapping:

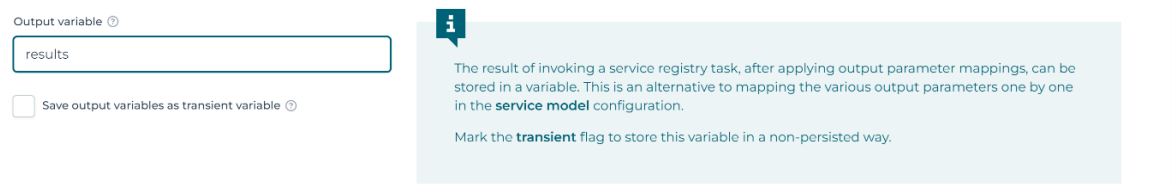
- If the *Target Expression* is a literal value, this is the name of the variable into which the output parameter value is stored.
- If the *Target Expression* is an expression, a writable variable is expected. For example, `${root.var}` is such a variable.

In either case, type coercion (the same as described above for the input mapping) based on the type of the output parameter is applied.

Advanced Attributes

The service registry task has two advanced attributes:

- Save output variables as transient variable.
- Output variable name.



When the *'Save output variables as transient variable'* is checked, all output variables are stored as transient (i.e., non-persisted) variables. This is useful when a service returns data that is not needed to be stored in the process or case instance (e.g., the data is only used in a calculation or as input for another service call).

When the *'Output variable'* is filled, no output parameter mapping is applied. The result of the service call is to be stored as-is under the given variable name. This attribute is important, when dealing with list values (arrays).

REST Parameter

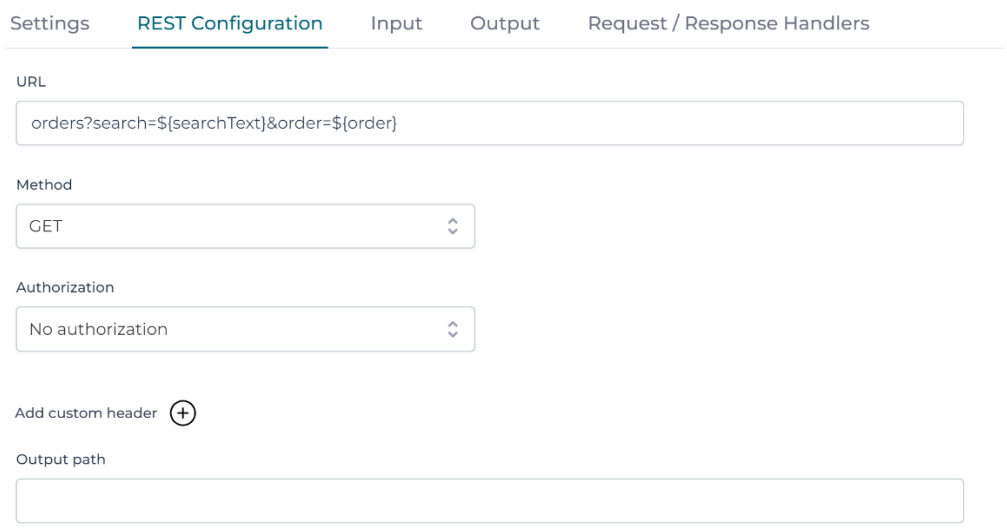
REST Input Parameters

The input parameters of a REST-based service model operation are straightforward, as they define the inputs of the operations. The values of the input parameters are typically used in the URL (using the technical name such as `${myInputParameter}`) or in the body of the request, when doing a POST request.

- **Label** : This is the human-readable name for the input parameter.
- **Name** : This is the technical name of the parameter, as used at runtime by the engine logic. It can be referenced in the URL using `${myInputParameter}`.
- **Type** : The type of the input.
- **Description** : The description of the input parameter name for documentation purposes.
- **Required** : Whether the input parameter is required.

- **Default Value** : The default value to use in case the parameter has not been provided. Can be a fixed value or an *expression*. For example, if you want to initialize an empty string as default value for a String, you can use `${''}`.
- **Query Parameter** : Checking the box identifies the parameter as a query parameter.

An example of using input parameters in the URL part of the service operation configuration is shown in the following image:



The parameters can, of course, be referenced anywhere in the URL string
e.g. `clients/${clientId}`

REST Output Parameters

The options for an output parameter are a bit more complex, as the configuration needs to be flexible enough to be able to process a variety of different JSON responses of calling REST API's:

- **Label**: This is the human-readable name for the parameter.
- **Source**: The source of the output parameter data.
  - **Default**: The response body is the source of data and output parameter are processed as usual.
  - **Full Response body**: The body as a whole will be used. The possible types are limited to *JSON*, *Array* or *String*.
  - **All response headers**: All response headers are stored in an array like `[{key:'Content-Type',value: 'application/json'}]`.
  - **Specific response header**: A specific header is extracted from the response headers. The header name is case-insensitive.
  - **Status code**: The response status code.
- **Name**: This is the technical name of the parameter, as used at runtime by the engine logic. If no other options are configured, this value is used to find in the JSON response a field with this specific name.
- **Type** : The type of output parameter.

- **Body Location**: Allows to map the output value to a more complex field with dots, like `customer.name`.
- **Excluded from body**: Only relevant for input parameters.
- **Path**: Uses JSON pointer syntax to allow navigation through nested JSON structures like `/customers/0/name` to indicate where the value can be found.
- **Mapping name**: In certain exceptional situations, it could be that names overlap. For example, assume that a REST API returns `{ buyer: { name: 'a' }, seller: { name: 'b' }}` as response. Both will map to the technical `name` and at runtime, the BPMN/CMMN service task will not be able to know which `name` value should be taken. To solve this, a different `mapping name` value should be given to allow the runtime to distinguish.
- **Map on error**: The flag to mark the *output parameter* as *error output parameter*.

Configuration Examples

To clarify the descriptions above, let us look at some examples:

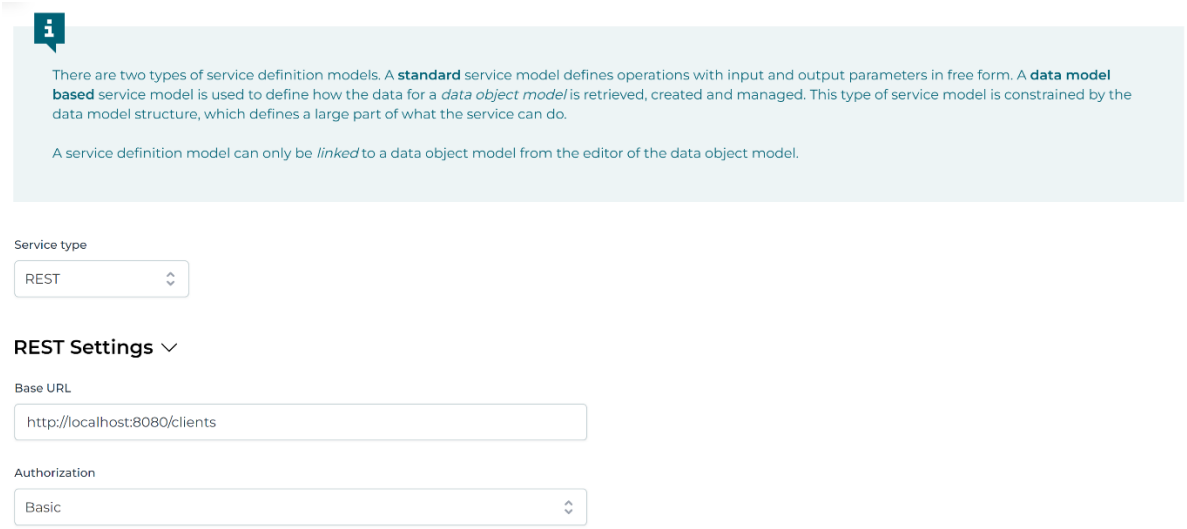
Getting a Simple JSON Value

In this example, we are using a REST service that fetches information that we want to store in a process or case instance.

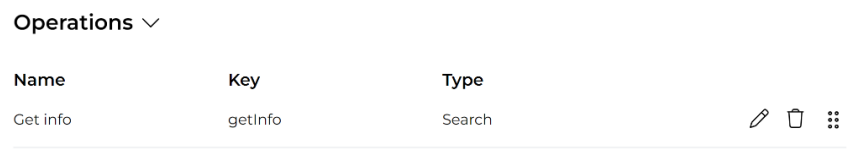
Suppose we have a REST service that fetches client information using the URL `/clients/${id}`. The service returns client information as follows:

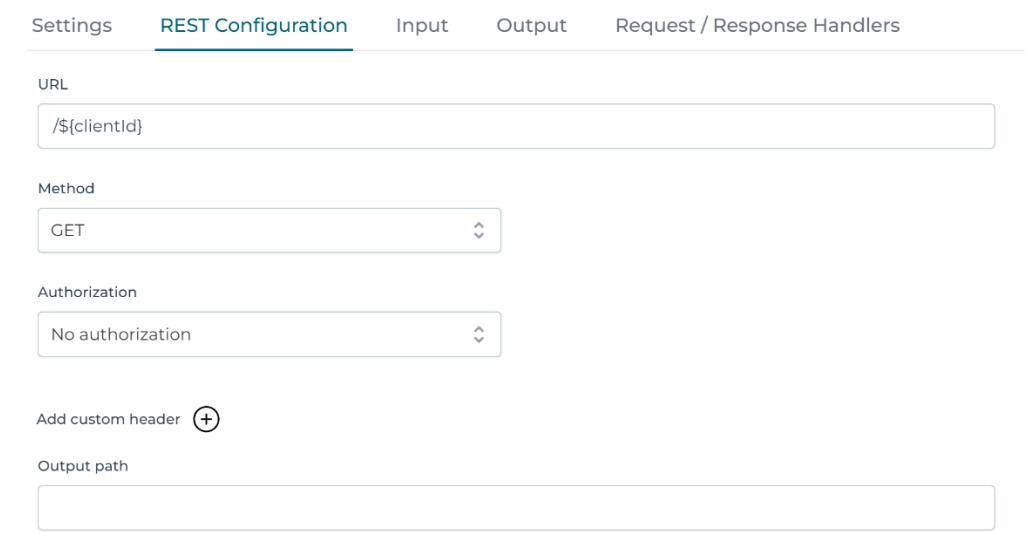
```
{
    "id": 123,
    "firstName": "John",
    "lastName": "Doe",
    "address": "Somelane 3, 1234 City",
    "birthDate": "1970-01-01T01:02:03.456Z",
    "creditScore": 123
}
```

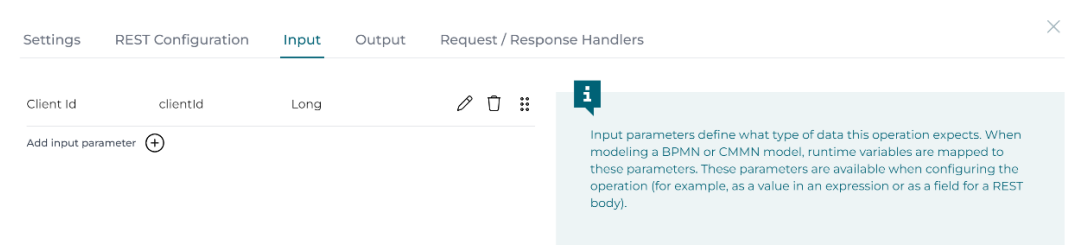
The service definition looks as follows:

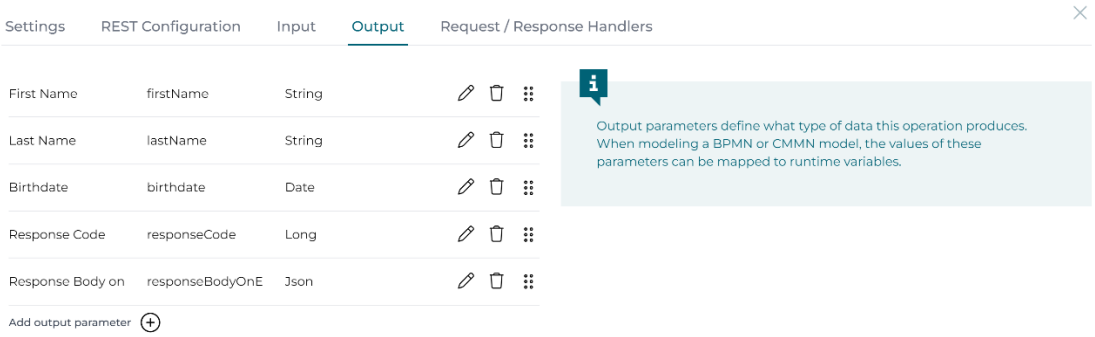


And its one operation:





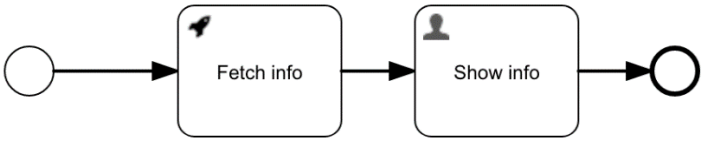




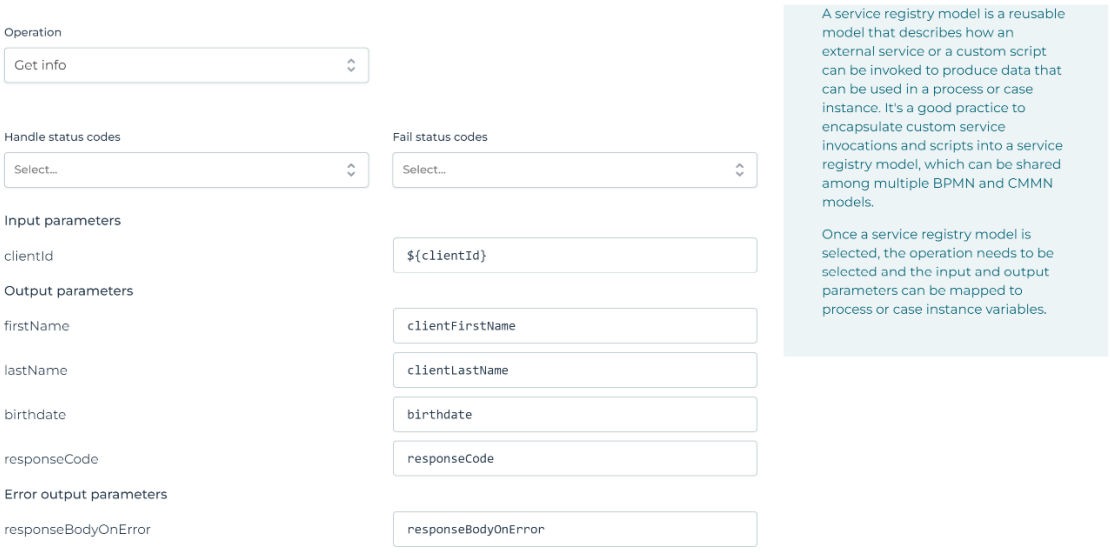
Note how the input parameter `clientId` is used as part of the URL for this operation.

Also, note that we are only returning a handful of the available properties. Not having the other properties (e.g., creditScore) in the output parameter list filters those out.

We can now create a simple process that uses that service:



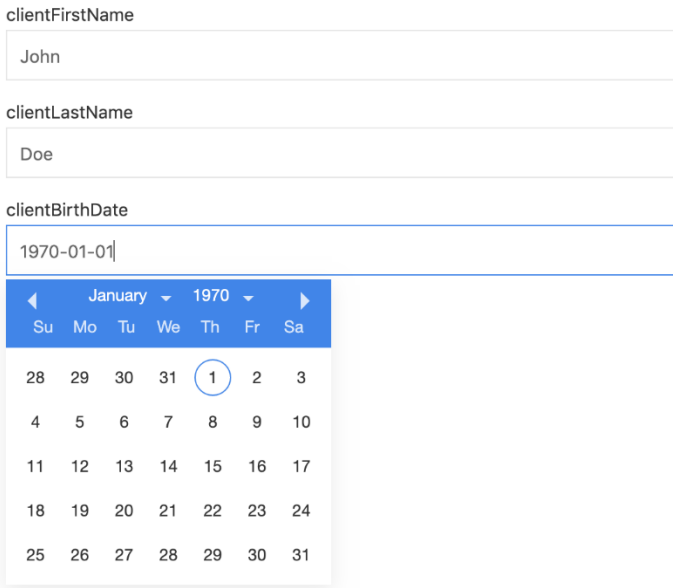
Assuming the process model has a start form with `clientId`, the parameter mapping looks as follows:



Here we are mapping the value of the `clientId` from the start form to the single input parameter and the first name, last name, and birthDate are mapped to specific output

variables. If this process was part of a case, we could push the variables upwards using `${root.clientFirstName}` or `${parent.clientFirstName}`. The response code is stored in the variable `responseCode`. The error output parameter `responseBody` is mapped to the variable `responseBodyOnError`.

If we run this process say in Flowable Work, the user task after starting the process instance now shows something like this, which validates that the service was invoked and variables were passed back and forth correctly.

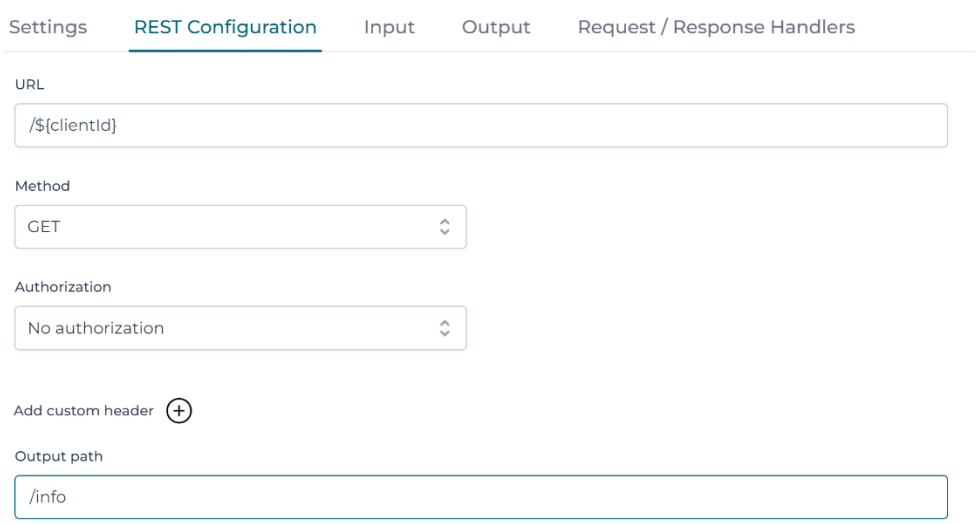


Using Paths to get a Nested Value

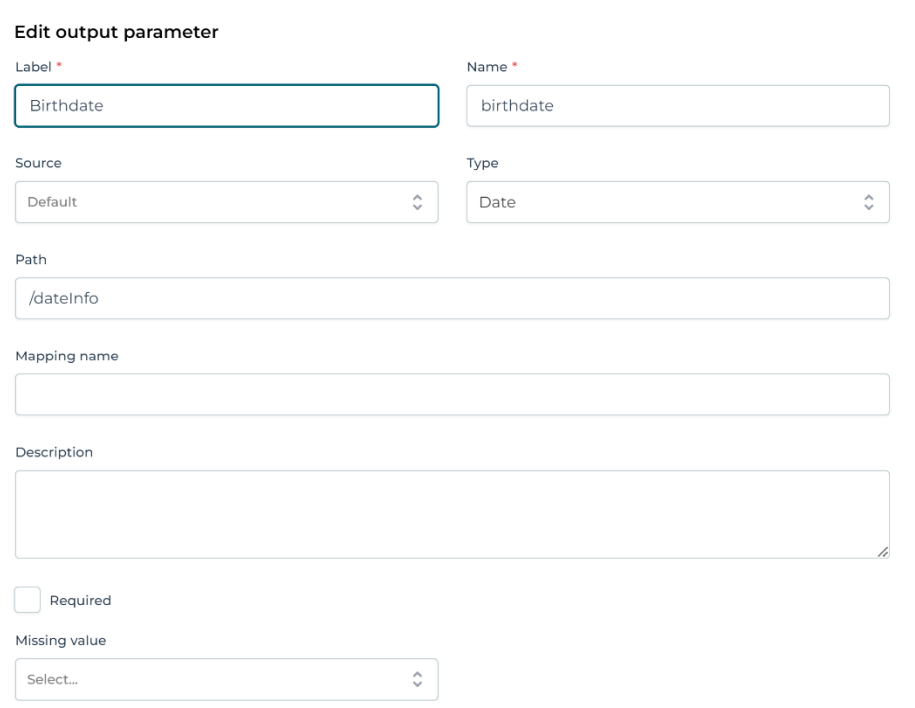
Let us adapt the example in the previous section to include nested fields in the response. For example, the *firstName* and *lastName* are under the *info* field and the *birthDate* is further nested under the *dateInfo* property:

```
{
    "id": 123,
    "info": {
        "firstName": "John",
        "lastName": "Doe",
        "address": "Somelane 3, 1234 City",
        "dateInfo": {
            "birthDate": "1970-01-01T01:02:03.456Z"
        },
        "creditScore": 123
    }
}
```

To make the example in the previous section work, the **output path** is set to `info` (as everything is nested under that property), and the **path** of the birthDate is set to `dateInfo` (the path is relative to the output path of the operation):



And for the `birthDate` parameter:

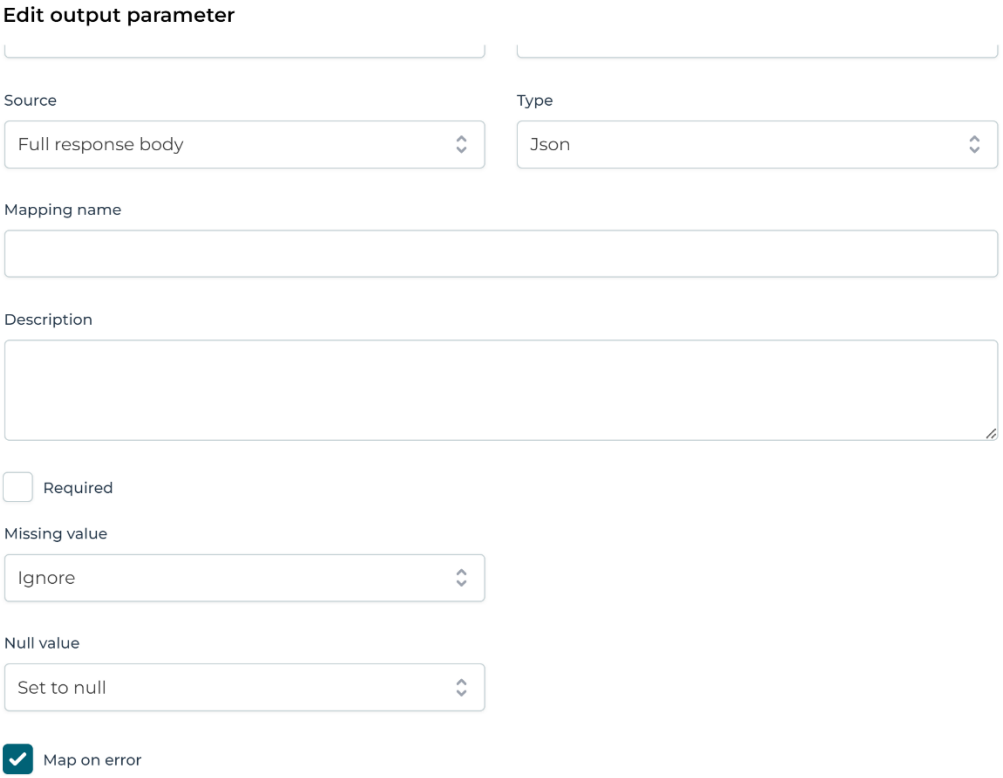


NOTE

The `Path` is used to navigate to the object and the `Name` is used to denote the attribute in the json response body.

Creating error output parameter

To create an error output parameter, the `Map on error` flag must be set to `true`.



Getting a List/Array of Values

In this example, we have a REST endpoint that returns an array of clients:

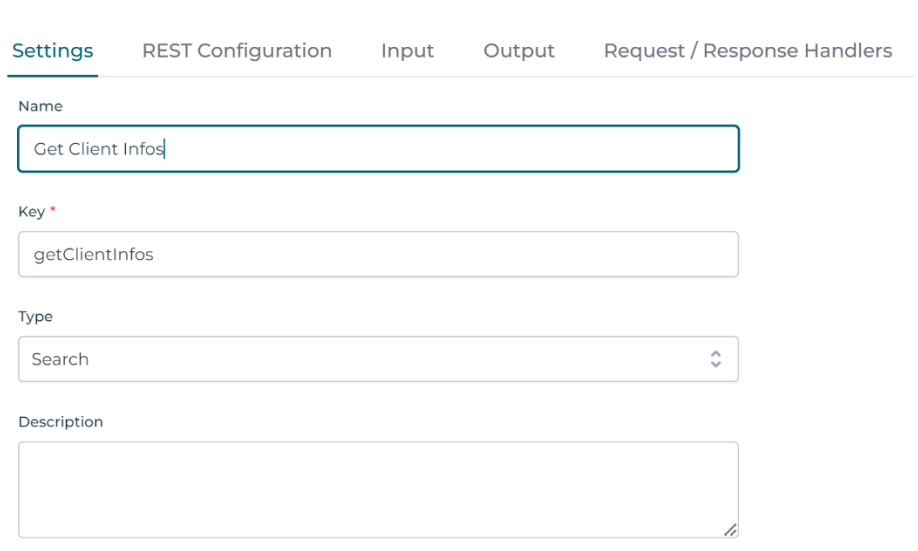
```
[
    {
        "id": 123,
        "info": {
            "name": "John",
            "lastName": "Doe",
            "address": "Somelane 3, 1234 City",
            "dateInfo": {
                "birthDate": "1960-01-01T01:02:03.456Z"
            },
            "creditScore": 123,
           "employer": {
                "name": "Flowable",
                 "address": "Somelane 3, 4432 Bern"
           }
        }
    },
    {
        "id": 456,
        "info": {
            "name": "Jane",
            "lastName": "Doe",
            "address": "Somelane 3, 1234 City",
            "dateInfo": {
                "birthDate": "1963-01-04T01:02:03.456Z"
            },
```

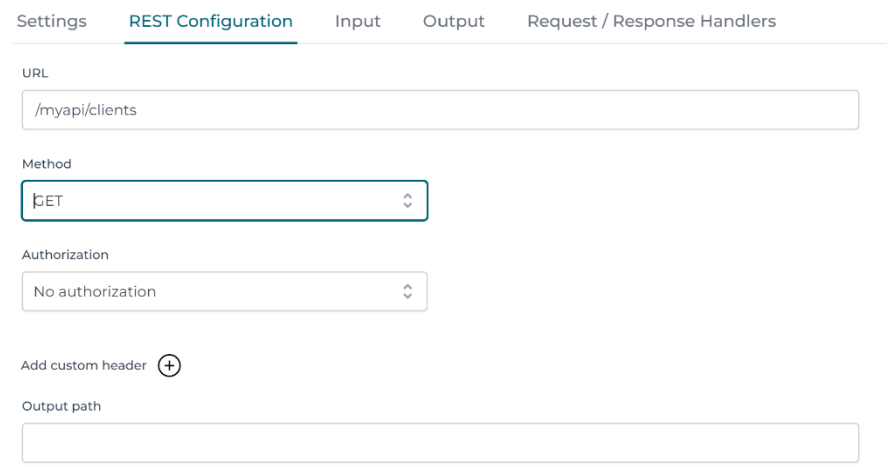
```
                "creditScore": 5,
                "employer": {
                    "name": "Acme",
                     "address": "Somelane 177, 10365 Berlin"
                }
            }
        },
        {
            "id": 789,
            "info": {
                "name": "Jimmy",
                "lastName": "Doe",
                "address": "Somelane 3, 1234 City",
                "dateInfo": {
                    "birthDate": "1990-01-01T01:02:03.456Z"
                },
                "creditScore": 123,
                "employer": {
                    "name": "Megacorp",
                    "address": "Somelane 256, 60604 Chicago"
                }
            }
        }
    ]
```

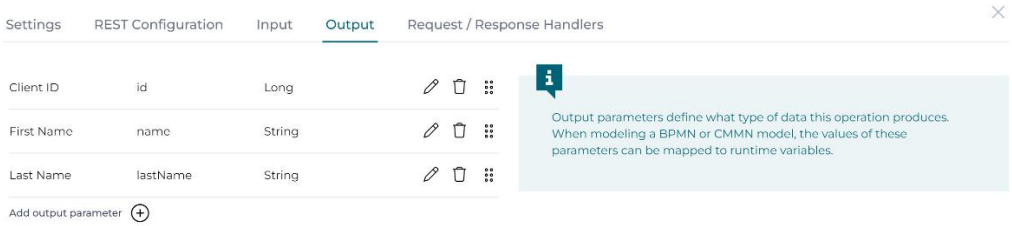

Again reusing the service definition from the previous example, we now add a new operation.

We map the required data from the returned array objects into service output parameters. In this example we are only interested in the `id`, `name` and `lastName` of the person:

The operation configurations looks like follows:







The individual output parameter configuration are listed in the tables below. Only required or non-default values are listed.

Client ID

Parameter	Value
Label	Person ID
Name	id
Source	Default
Type	Long

First Name

Parameter	Value
Label	First Name
Name	name
Source	Default
Type	String
Path	/info

Last Name

Parameter	Value
Label	Last Name
Name	lastName
Source	Default
Type	String
Path	/info

Note that this example uses `Path` and `Name` to point to the `name` and `lastName` attributes. The `Path` is used to navigate to the object and the `Name` is used to denote the attribute in the JSON response body.

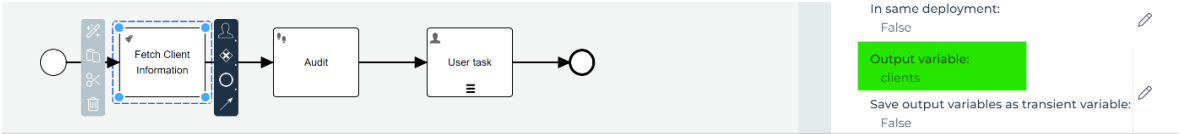
NOTE

It is important to understand that in the case of arrays, the configuration of the output parameters is applied to *each* element of the array.

The result looks like this:

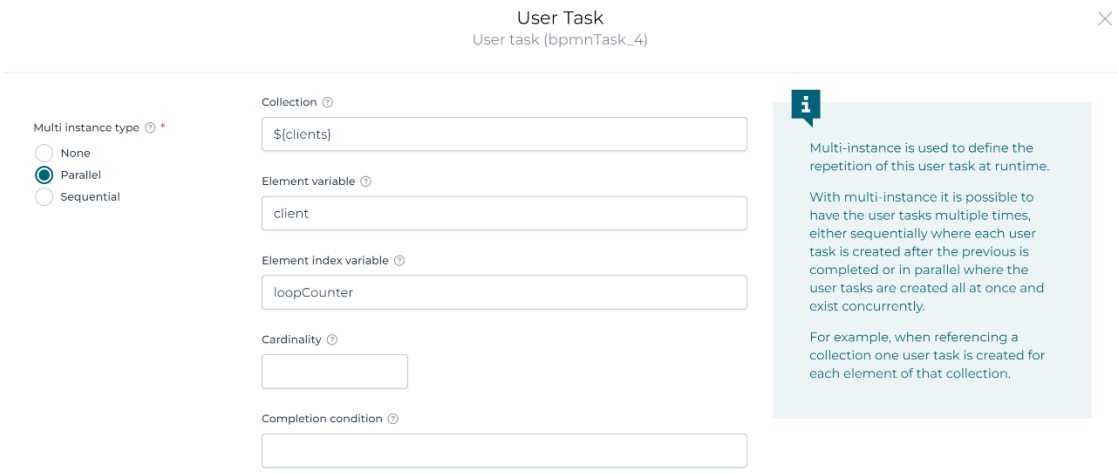
```
[
  {
    "name": "John",
    "lastName": "Doe",
    "employerName": "Flowable",
    "id": 123
  },
  {
    "name": "Jane",
    "lastName": "Doe",
    "id": 456
  },
  {
    "name": "Jimmy",
    "lastName": "Doe",
    "id": 789
  }
]
```

This service can now be used in a process or case model:



Note that only the *Output variable* is set. For array return types, the *Service Output Parameters* mapping can be ignored.

Internally, an ArrayNode is returned. This can be used to configure multi-instance activities, such as user tasks. The configuration of a multi-instance activity could look like this:



Using Mapping name to resolve name overlaps

Based on the JSON structure from the previous example, we now want to extend the service to additionally return the employer name and employer address line together with the client address line.

The problem is, the attribute names `name` and `address` overlap for the client info and the employer in this example. This can be fixed using the **Mapping name** configuration field.

The service output parameter configuration looks like this:

Client Address

Parameter	Value
Label	Client Address
Name	address
Source	Default
Type	String
Path	/info

Employer Name

Parameter	Value
Label	Employer Name
Name	name

Parameter	Value
Source	Default
Type	String
Path	/info/employer
Mapping name	employerName

Employer Address

Parameter	Value
Label	Employer Address
Name	address
Source	Default
Type	String
Path	/info/employer
Mapping name	employerAddress

NOTE

Note the usage of `Mapping name`, `Path` and `Name` in the output parameter configurations.

The target attribute in the service response JSON body is expressed by the `Path` to navigate to the object and the `Name` to denote the attribute. `Mapping name` is then used to resolve the overlapping names.

The result looks like this:


```
[
  {
    "name": "John",
    "lastName": "Doe",
    "address": "Somelane 3, 1234 City",
    "employerAddress": "Somelane 3, 4432 Bern",
    "employerName": "Flowable",
    "id": 123
  },
  {
    "name": "Jane",
    "lastName": "Doe",
    "address": "Somelane 3, 1234 City",
    "employerAddress": "Somelane 177, 10365 Berlin",
    "employerName": "Acme",
    "id": 456
  },
  {
    "name": "Jimmy",
    "lastName": "Doe",
    "address": "Somelane 3, 1234 City",
```

```
    "employerAddress": "Somelane 256, 60604 Chicago",
    "employerName": "Megacorp",
    "id": 789
  }
]
```

'Source': Returning the response payload and response headers

The `Source` configuration of the service output parameter configuration allows you to configure to return the response payload, response headers, etc. The different options are showcased in this section.

It is possible to return the response payload as-is as a service output parameter. To do this, the `Source` setting can be set to `Response body`. The entire response body will be assigned to the output parameter.

Given this JSON payload:

```
{
    "id": 117,
    "info": {
      "name": "Jimmy",
      "lastName": "Doe",
      "address": "Somelane 3, 1234 City",
      "dateInfo": {
        "birthDate": "1990-01-01T01:02:03.456Z"
      },
      "creditScore": 123
}
```

The following output parameter configurations showcase all possible values for `Source`, when applied to the response above.

Source: Status Code

Parameter	Value
Label	Response Status Code
Name	responseStatusCode
Source	Status Code
Type	Long

Source: Full response body

Parameter	Value
Label	Full response body
Name	responsePayload
Source	Full response body
Type	JSON

Source: All response headers

Parameter	Value
Label	All response headers
Name	responseHeaders
Source	All response headers
Type	Array

Source: Specific response header

Parameter	Value
Label	Specific response header
Name	responseHeaderSpecific
Source	Specific response header
Type	String

Given the configuration above, the result looks like this:


```
{
  "responseStatusCode": 200,
  "responseHeaders": [
    {
      "key": "Content-Type",
      "value": "application/json; charset=utf-8"
    },
    {
      "key": "Content-Length",
      "value": "309"
    }
  ],
  "responsePayload": {
    "id": 117,
    "info": {
      "name": "Jimmy",
      "lastName": "Doe",
      "address": "Somelane 3, 1234 City",
      "dateInfo": {
```

```
        "birthDate": "1990-01-01T01:02:03.456Z"
      },
      "creditScore": 123
    }
  },
  "responseHeaderSpecific": "application/json; charset=utf-8"
}
```

CAUTION

Caution with the `Full response body` and `All response headers` setting

In general, it is strongly recommended to define the output parameters to return only the data that is needed in the case or process model. In this way, the amount of data potentially stored as variables is kept to the required minimum.

List/Array Behavior

The behavior for arrays is a bit different in terms to the `Source`. As the output parameter configuration is applied to *every element in the array*, the service output in the used configuration above would result in the following service response when using this array response:

REST array response body

```
[
  {
    "id": 17,
    "info": {
      "name": "Jimmy",
      "lastName": "Doe",
      "creditScore": 5
    }
  },
  {
    "id": 456,
    "info": {
      "name": "Jane",
      "lastName": "Doe",
      "creditScore": 5
    }
  }
]
```

Service response after applying the output parameter mappings above

```
[
  {
    "responseStatusCode": 200,
    "responseHeaders": [
      {
        "key": "Content-Type",
        "value": "application/json; charset=utf-8"
      }
    ],
    "responsePayload":
      {
```

```
        "id": 17,
        "info": {
          "name": "Jimmy",
          "lastName": "Doe",
          "creditScore": 5
        }
      },
    "responseHeaderSpecific": "application/json; charset=utf-8"
  },
  {
    "responseStatusCode": 200,
    "responseHeaders": [
      {
        "key": "Content-Type",
        "value": "application/json; charset=utf-8"
      }
    ],
    "responsePayload":
      {
        "id": 456,
        "info": {
          "name": "Jane",
          "lastName": "Doe",
          "creditScore": 5,
        }
      },
    "responseHeaderSpecific": "application/json; charset=utf-8"
  }
]
```

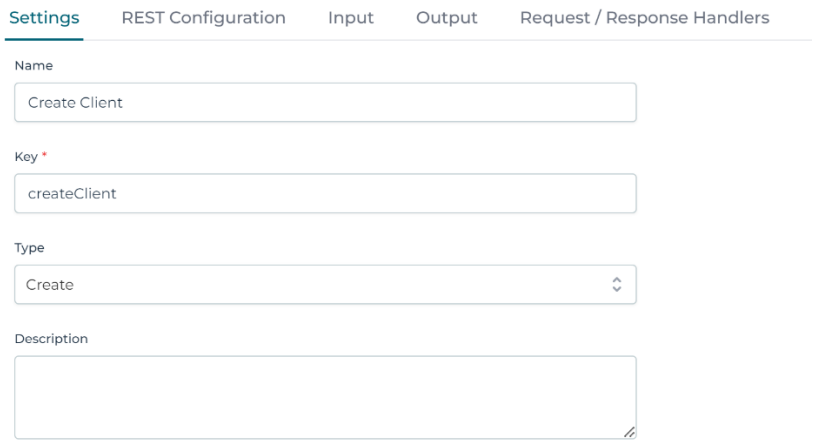

Because the output parameter mappings are applied to each element individually, the resulting array contains objects with the configured service output parameters. This means that, for example, `responseHeaders` are repeated for each array element.

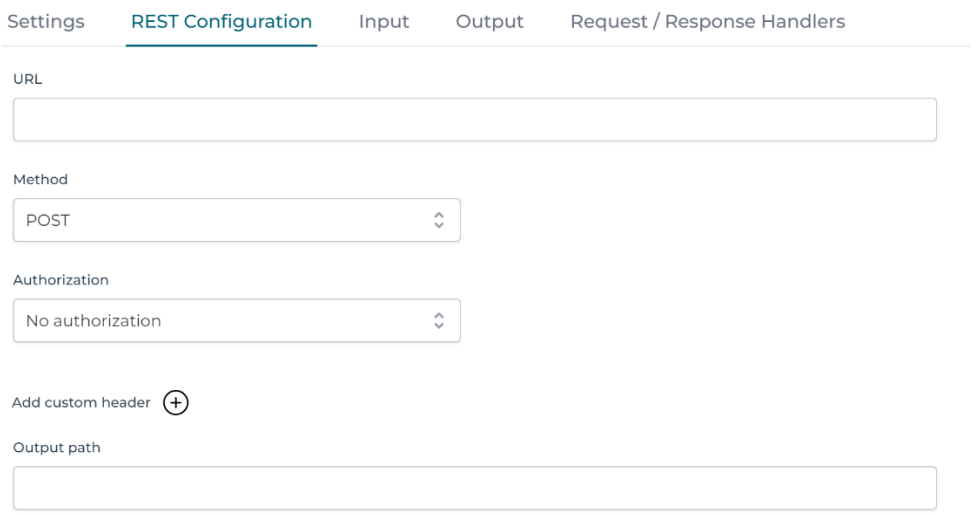
Posting to a REST Service

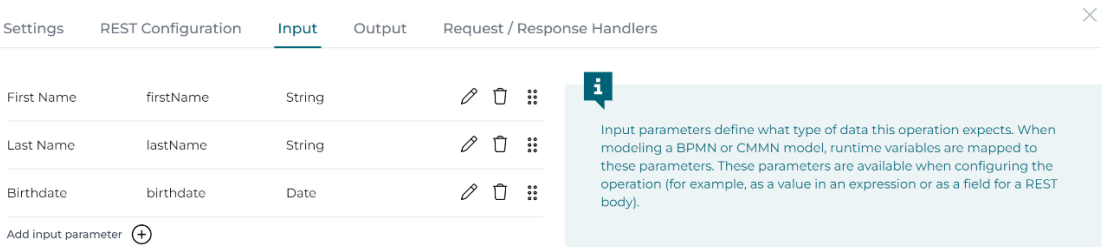
Suppose there exists another REST endpoint that allows creating a client object. Adding such an operation is similar to the steps above. First, add a new 'create' operation to the example from above and now:

- Configure a set of input parameters that populate the JSON body when doing the POST.
- For more complex use cases, a custom Freemarker template can be used.

If the REST endpoint accepts a simple flat JSON structure for creating a client, the operation configuration could, for example, look like:







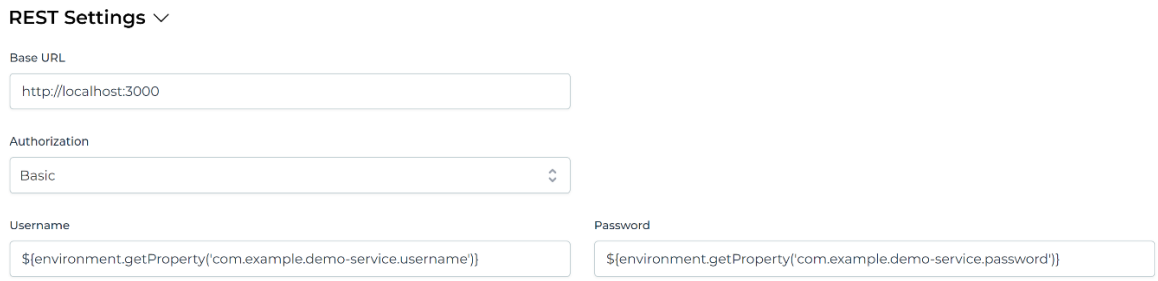
Authorization for a REST Service

Depending on how the REST-backed service is protected the service needs to be configured differently. In most cases the *Authorization* header needs to be set. Flowable offers an easy way to configure the Authorization header for HTTP Basic and HTTP Bearer authorizations by configuring the `authorization` for a model or an operation.

NOTE

The images below are showing the configuration on the definition level (applicable for all operations). However, each operation can be configured separately (or overwrite the configuration from the model) if needed.

Basic Authorization



Bearer Authorization