SAP Signavio Process Governance Developer Guide

3.199

Contents

1 SAP Signavio Process Governance Developer Guide

This guide is intended for users who want to write code to customize workflow behavior and integrate with other systems. The SAP Signavio Process Governance User Guide, describes all aspects of using SAP Signavio Process Governance without writing code.

1.1 Starting cases automatically

External systems start new cases using the Trigger API.

1.2 Integrating with external systems

You can also integrate workflows with external systems in three ways:

1. JavaScript actions can connect to external systems via HTTP and WebDAV
2. Send email actions can send email that includes workflow data
3. Custom data connectors populate form field choice options with external data

1.3 Fetching cases and tasks data

SAP Signavio Process Governance does not provide an API for fetching data about tasks and cases. Use the *Analytics* features to manually export reports in CSV format.

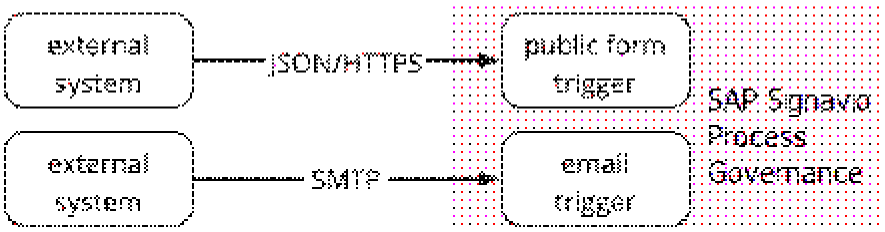
1.4 Interacting with tasks and cases

SAP Signavio Process Governance does not provide an API for interacting with tasks and cases. SAP Signavio Process Governance is intended to only be used via its user interface. This means that you cannot use SAP Signavio

Process Governance as an embedded process engine in your own application.

2 Trigger API

Use the SAP Signavio Process Governance trigger API to start cases automatically. The API supports three trigger types.



Two trigger types: public forms and email messages

External systems start new cases in SAP Signavio Process Governance in three ways:

1. **Public form triggers** start cases from HTTP/JSON API requests
2. **Email triggers** start cases from incoming email

You can use public form triggers and email triggers from external systems, by sending HTTP requests and email.

2.1 API limitation

To ensure our platform remains stable and fair for everyone, we ask developers to use industry standard techniques for limiting requests, caching results, and re-trying requests responsibly.

API rate limits	50 requests/60 seconds
Concurrency limits	Read: 5, Write: 1

2.2 Public forms HTTP API

The API for submitting a SAP Signavio Process Governance public form uses JSON data sent in an HTTP POST request to start a new case. The JSON data holds an 'instance' of the trigger form.

The HTTP request includes the IDs for the process to start (the `sourceWork-flowId`) and each of the form fields. Each form field also has a data type. To discover these field IDs and types, fetch the form definition from the trigger API:

```
GET /api/v1/public/start-form/1ac791d862f8a02e51300000 HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
    Content-Type: application/json;charset=UTF-8

    {
        "form": {
            "fields": [
                {
                    "elementType": "fieldInstance",
                    "id": "p6rsah4otbmkll1j90",
                    "name": "Type of feedback",
                    "required": true,
                    "type": {
                        "name": "choice",
                        "options": [
                            {
                                "id": "0",
                                "name": "Praise"
                            },
                            {
                                "id": "1",
                                "name": "Question"
                            },
                            {
                                "id": "2",
                                "name": "Complaint"
                            }
                        ]
                    },
                    "visible": true
                },
                {
                    "elementType": "fieldInstance",
                    "id": "p6rscwnlqfrzkot2zj",
                    "name": "Feedback",
                    "required": true,
                    "type": {
                        "multiLine": true,
                        "name": "text"
                    },
                    "visible": true
                },
                {
                    "description": "When did you start using the
product?",
                    "elementType": "fieldInstance",
                    "id": "pgoxccp25d71iyj680",
                    "name": "Product start date",
                    "type": {
                        "kind": "date",
                        "name": "date"
                    },
                    "visible": true
                },
                {
                    "elementType": "fieldInstance",
                    "id": "p6rsd7xwzvxg6aimww",
                    "name": "Contact email address",
                    "type": {
                        "name": "emailAddress"
                    },
                    "visible": true
                }
            ]
        },
        "submitAction": "Send Feedback"
    }
```

Most of these fields are unnecessary: the HTTP request you send to trigger the workflow and start a case only needs to include each field’s ID, type and value, as shown in the following literal HTTP request (most HTTP headers omitted for brevity). Note that you can omit the field type for text types.

```
POST /api/v1/public/cases HTTP/1.1
    Content-Type: application/json
    Host: workflow.signavio.com

    {
      "triggerInstance": {
        "data": {
          "formInstance": {
            "value": {
              "fields": [
                {
                  "id": "16rsah4otbmk000000",
                  "value": "0"
                },
                {
                  "id": "16rscwnlqfrz000000",
                  "value": "I’m so happy!"
                },
                {
                  "id": "pgoxccp25d71iyj680",
                  "type": {
                    "name": "date",
                    "kind": "date"
                  },
                  "value": "2018-01-01T00:00:00.000Z"
                },
                {
                  "id": "16rsd7xwzvxg000000",
                  "value": "bob@example.com"
                }
              ]
            }
          }
        },
        "sourceWorkflowId": "1ac791d862f8a02e51000000"
      }
    }
```

Note that the value for a Choice type field, is the choice type option ID, not the option name.

The HTTP response includes the newly-created case's ID and name (most headers omitted):

```
HTTP/1.1 200 OK
    Content-Type: application/json;charset=UTF-8

    {
      "id": "1ac79962d1dfff2e36000000",
      "name": "Feedback #6 - Praise"
    }
```

2.2.1 HTML/JavaScript form example

There's more than one way to build a custom form that uses the HTTP work-flow API. For simplicity, this example uses:

1. An HTML form that the customer will use to enter feedback
2. A JavaScript event handler for the form's `onsubmit` event
3. A JavaScript object that serialises to the correct JSON using `JSON.stringify`
4. A JavaScript **Fetch API request** to submit the form

The following example code aims for simplicity and no external dependencies, but does not support older web browsers and doesn't implement a proper user experience. In practice, you would modify this to take advantage of your favorite JavaScript libraries.

A minimal plain HTML document containing the form could look like this:

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Custom form</title>
  <script type="text/javascript" src="sendForm.js"></script>
</head>
<body>
  <form action="https://workflow.signavio.com/api/v1/public/cases">
    <p>
      <label for="type">Type of feedback</label>
      <select id="type" name="type">
        <option>Praise</option>
        <option>Question</option>
        <option>Complaint</option>
      </select>
    </p>
    <p>
      <label for="feedback">Feedback</label>
      <textarea id="feedback" name="feedback" rows="8"
cols="40"></textarea>
    </p>
    <p>
```

```
      <label for="id">E-mail address</label>
      <input type="email" id="email" name="email">
    </p>
    <p><button type="submit">Send Feedback</button></p>
  </form>
</body>
</html>
```

The script tag refers to the following JavaScript (ES2016), which uses the fetch API to send the HTTP request:

```
// Sends the form data to the workflow’s public form trigger.
const sendForm = (event) => {
  event.preventDefault()
  const triggerData = {
    triggerInstance: {
      sourceWorkflowId: '1ac791d862f8a02e51000000',
      data: {
        formInstance: {
          value: {
            fields: [
              {
                "id": "16rsah4otbmk000000",
                "value": document.querySelector('#type').value
              },
              {
                "id": "16rscwnlqfrz000000",
                "value": document.querySelector('#feedback').value
              },
              {
                "id": "16rsd7xwzvxg000000",
                "value": document.querySelector('#email').value
              }
            ]
          }
        }
      }
    }
  }
  const requestOptions = {
    mode: 'no-cors',
    method: 'POST',
    body: JSON.stringify(triggerData),
    headers: new Headers({
      'Content-Type': 'application/json'
    })
  }
  var currentTime = new Date().getTime();
  fetch(document.querySelector('form').action, requestOptions)
    .then(response => console.log('Form submitted'))
    .catch(error => console.error(error))
}

// When the DOM has loaded, attach the 'sendForm' function to the form.
document.addEventListener('DOMContentLoaded', () => {
  document.querySelector("form").addEventListener('submit', sendForm)
})
```

Use the form’s submit button to start a new case. Check the JavaScript console for the **Form submitted** message, to check that there were no errors.

2.3 Email trigger API

Email triggers start cases from incoming email. When an email trigger starts a case, the email data is available to the workflow. If you generate trigger emails programmatically, you can use trigger emails to asynchronously start cases using structured data.

For example, a customer contact form might include a *Product name* selection, and an *Enquiry* text input. To use this with an email trigger, encode this form data as a JSON trigger email body:

```
{
  "product" : "SAP Signavio Process Governance ",
  "enquiry" : "Is it available in French?"
}
```

In the workflow, add a JavaScript action to parse the trigger email. Add the built-in *Trigger email* variable, and text fields for the *Product* and *Enquiry*. In the script, parse the JSON to extract the values:

```
const enquiryForm = JSON.parse(triggerEmail.bodyText)
product = enquiryForm.product
enquiry = enquiryForm.enquiry
```

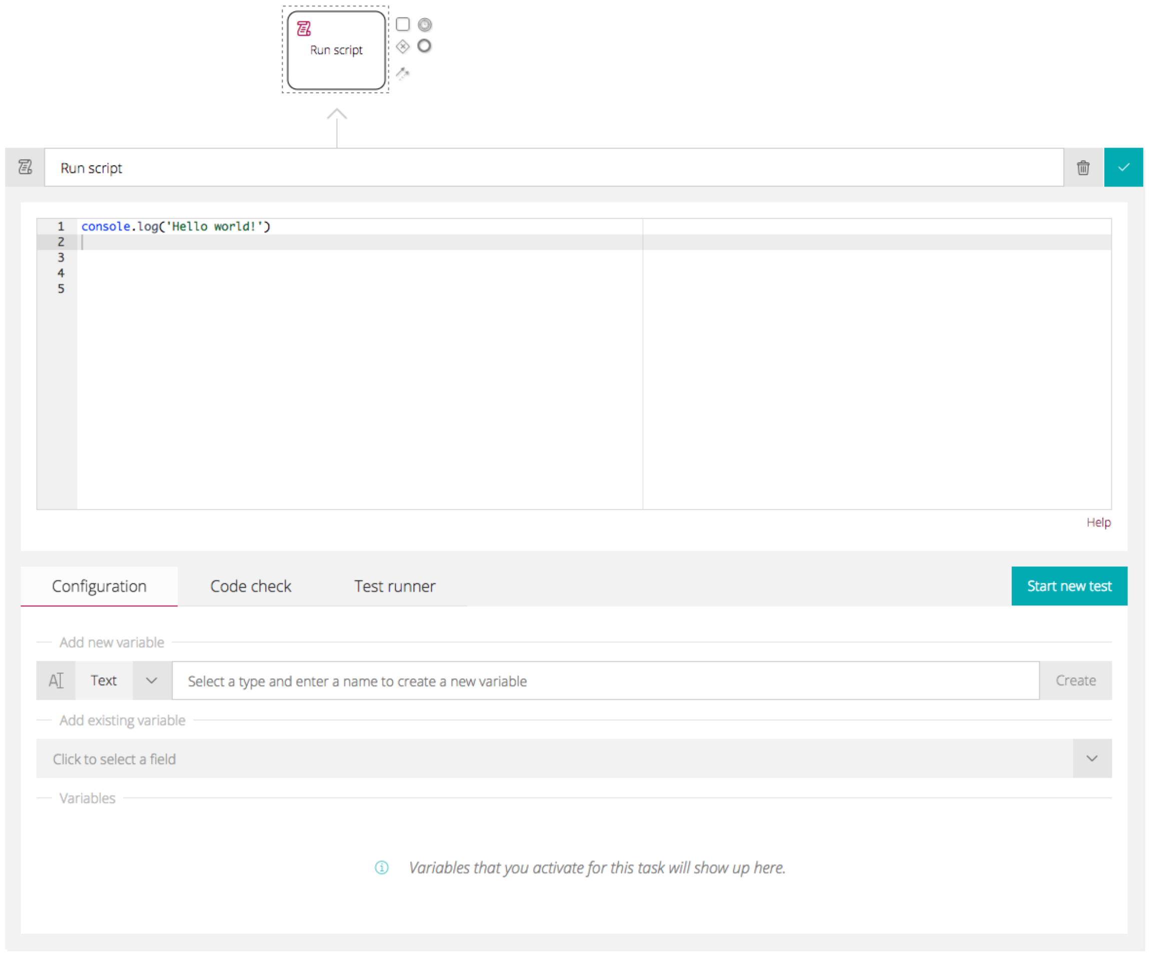
Note that the lack of error handling code means that invalid JSON will cause the script task to fail when executing the case, which will halt the workflow. To continue without setting the product value, wrap the code in try-catch statements.

3 JavaScript actions

Use a JavaScript action to execute JavaScript code during workflow execution.
SAP Signavio Process Governance runs the code on the server, using Node.js. This means that you can use ECMAScript 2015 (ES6) features.
As well as Node.js' JavaScript API, scripts can use additional JavaScript libraries.

3.1 JavaScript action configuration

After creating or selecting a JavaScript action, the configuration panel looks like this:



```
console.log('Hello world!')
```

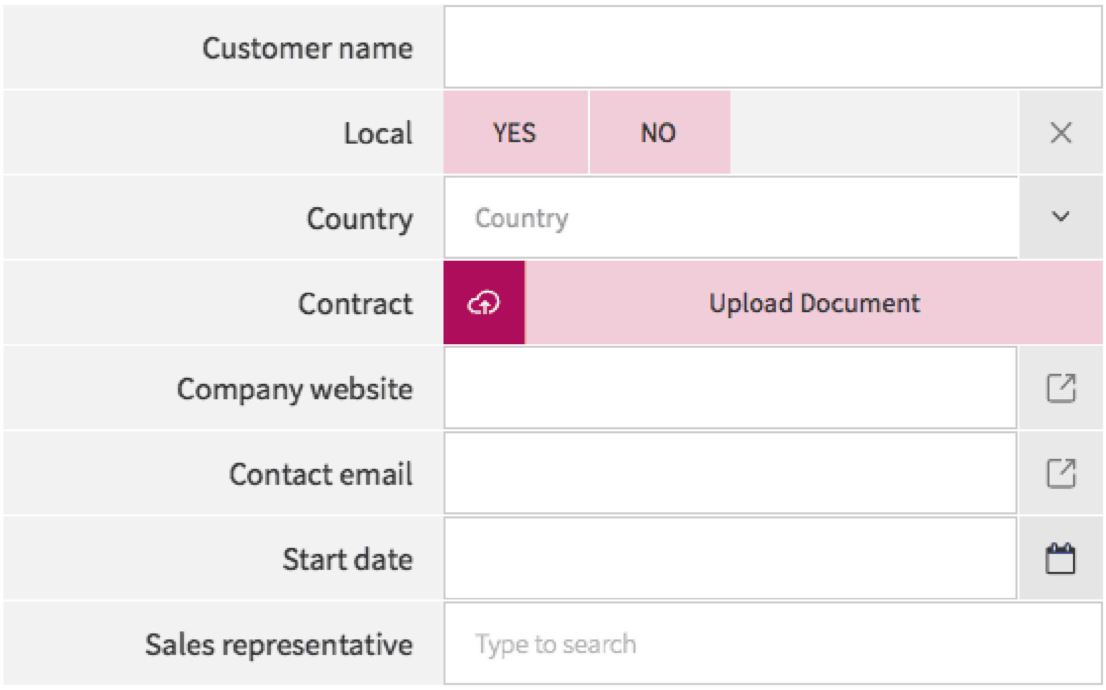
Variables that you activate for this task will show up here.

JavaScript configuration panel

The top section of the panel contains the JavaScript text editor. By default, it already contains `console.log('Hello World!')`. Use the console API for log output when testing scripts.

3.2 Using process variables

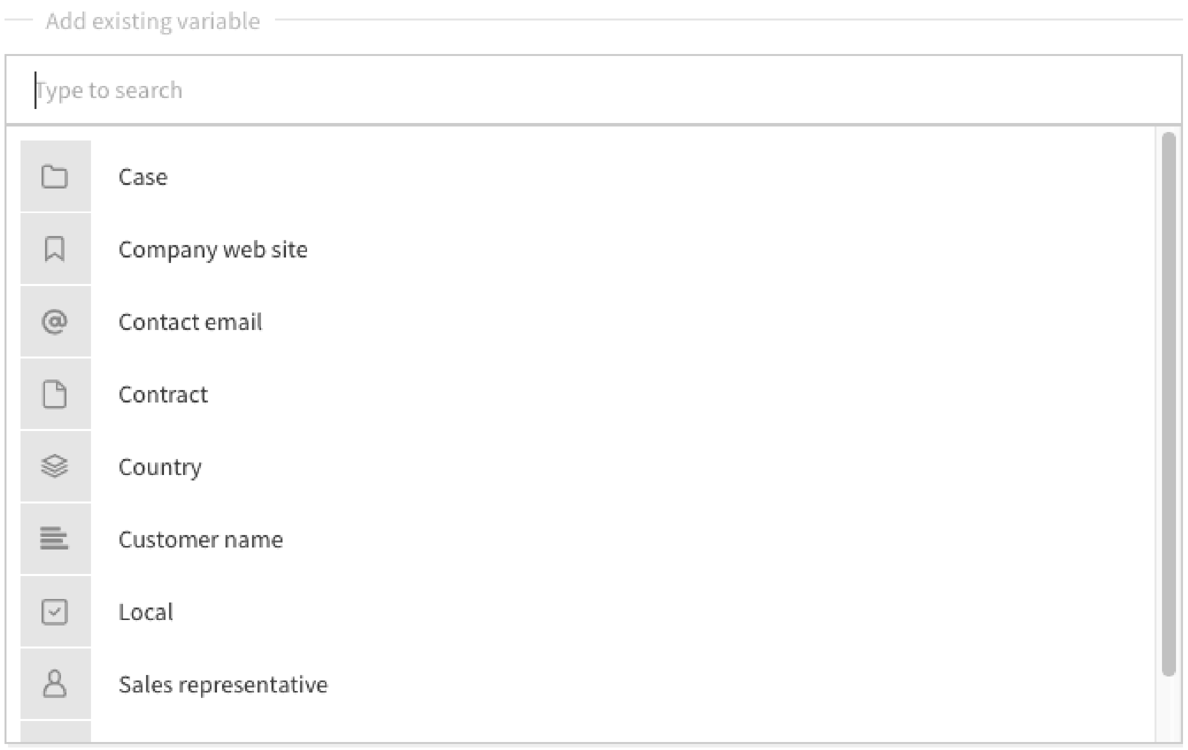
Next, we'll show how to work with data. Suppose that the process includes a form that has each type of field and looks like this:



Customer name	
Local	YES NO
Country	Country
Contract	Upload Document
Company website	
Contact email	
Start date	
Sales representative	Type to search

Form fields that declare process variables

On the JavaScript configuration panel, the **Add existing variable** pick list now shows the form field variables.

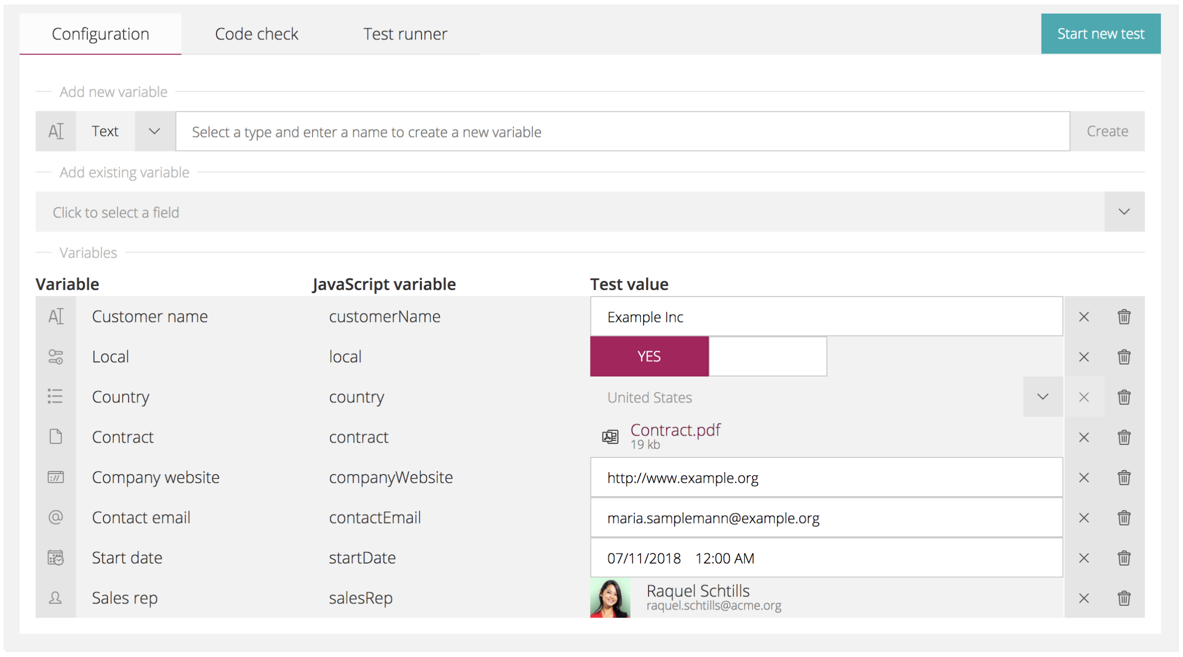


Process variable selection

Select the variables you want to access in the script. The script can access the variables using the **JavaScript variable** name from the **Variables** table. To access object variables' fields, use the field names specified for the corresponding data type: Case, Email, File or User.

You see all available fields after entering the variable name and `.`.

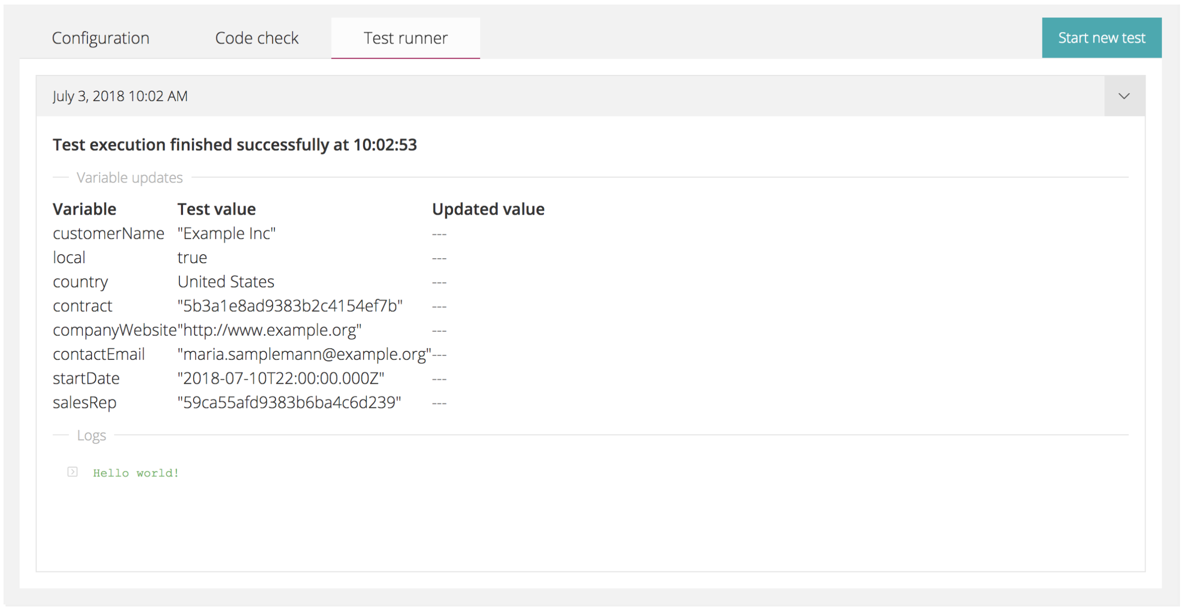
In this example (below), you have selected all variables. For each variable that you select, you get an input field to specify a test value. Here you see all fields with a test value.



Variable	JavaScript variable	Test value
Customer name	customerName	Example Inc
Local	local	YES
Country	country	United States
Contract	contract	Contract.pdf 19 kb
Company website	companyWebsite	http://www.example.org
Contact email	contactEmail	maria.samplemann@example.org
Start date	startDate	07/11/2018 12:00 AM
Sales rep	salesRep	Raquel Schtills raquel.schtills@acme.org

JavaScript test values

Clicking **Start new test** again to see the JSON structure of the variable data for the different variable types.



Test execution finished successfully at 10:02:53

Variable updates

Variable	Test value	Updated value
customerName	"Example Inc"	---
local	true	---
country	United States	---
contract	"5b3a1e8ad9383b2c4154ef7b"	---
companyWebsite	"http://www.example.org"	---
contactEmail	"maria.samplemann@example.org"	---
startDate	"2018-07-10T22:00:00.000Z"	---
salesRep	"59ca55afd9383b6ba4c6d239"	---

Logs

```
Hello world!
```

JavaScript JSON values

The *contract* and *salesRepresentative* variables have complex types, File and User, so the table only shows an ID. The **Updated value** column shows the result of assigning new values to these variables in the script.

3.3 Update process variables

You can use JavaScript actions to update process variables. Make sure you *re-assign* a new value to the variable instead of mutating the variable itself. Otherwise, the system will ignore the update. For example, the system ignores `contactEmails.push('joan.doe@example.org')`, but correctly processes `contactEmails = [].concat([], 'joan.doe@example.org')`. This restriction doesn't apply to variables you only use in the context of the JavaScript action.

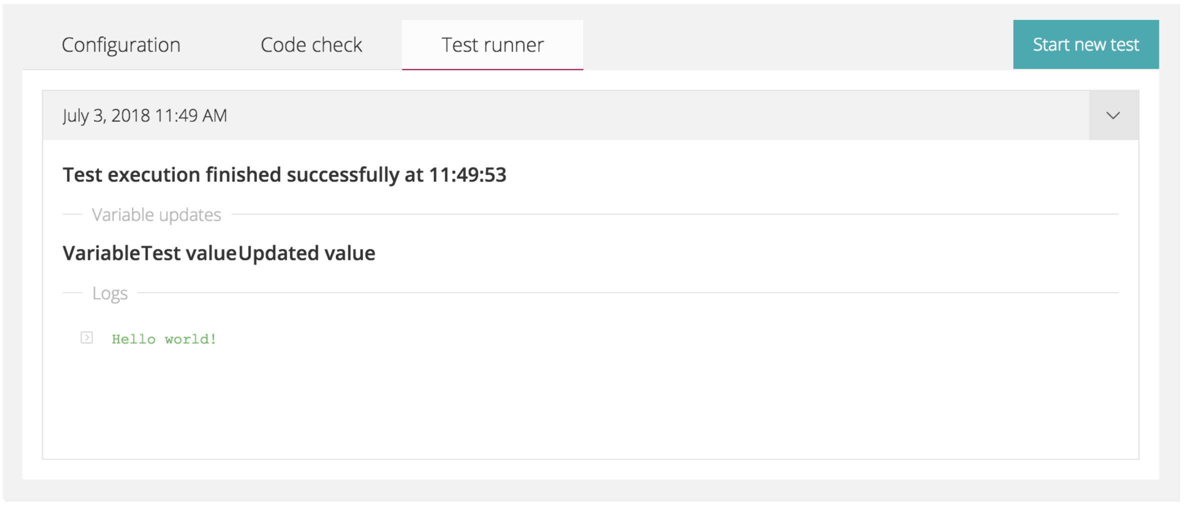
4 Checking your code

In the **Code check** tab, your code is checked automatically against the standard JavaScript code style.

You see a ✓ for a successful check and ⊗ if there are any errors in the code.
If the code contains errors, open the tab to get detailed information.
The detailed code check results indicate which JavaScript rule is violated and where the error occurs. Where possible, you can fix errors by clicking **Autofix**.

5 Testing scripts

Use the **Test Runner** tab to test the script. Click **Start new test** to execute the JavaScript code. The test runner displays the results underneath:



Test execution finished successfully at 11:49:53

Variable updates

VariableTest valueUpdated value

Logs

```
Hello world!
```

JavaScript test output

At the top, you see the test execution date and time. After running multiple tests, you can use this menu to select earlier test runs. The **Variable updates** section shows a table of process variables, with their test values and any updates. The **Logs** section shows console output and any errors.

To check if a variable has no value, use `'variable === null || variable === undefined'`.

6 JavaScript libraries

JavaScript actions support a number of popular JavaScript libraries. To import a package, use the `require` function:

```
const moment = require('moment');
```

You can also choose another name for the imported library:

```
const stringValidator = require('validator');
```

Supported JavaScript libraries

Library	Import	Description
CSV	`csv`	CSV generation, parsing, transformation and serialization
Files	`files`	Built-in API for File variable data
Lodash	`_`	Convenience functions for working with collections and values
moment	`moment`	Parse, validate, manipulate, and display dates; with Twix date range, and moment-business-days support
request	`request`	Simplified HTTP request client
Users	`users`	Built-in API for User variable data
validator	`validator`	String validation and sanitization
WebDAV client	`webdav-client`	Exchange files with a WebDAV endpoint
XML	`xml-js`	XML generation and parsing
SAP Signavio Process Manager client	`spm-client`	Authenticates request to the SAP Signavio Process Manager API automatically

The JavaScript action always imports the `_` (Lodash) and `request` packages, for backwards compatibility.

7 JavaScript action cookbook

This section shows you how to complete common tasks in JavaScript actions.

8 Reading file contents

A JavaScript action may need to read the contents of a file, in order to publish the file to an external web service. To access File content, you need to require the `files` API.

```
const files = require('files')
const fileContent = files.getContent(contract)
```

In this example, `contract` is a file variable that references the file contents that the script reads.

The `getContent` function returns a Node.js File object, whose `buffer` property provides access to the file content bytes. The following example loads a CSV file, converts the content bytes to a UTF-8 string, and parses the string:

```
const files = require('files')
const csv = require('csv')

// Read the reportCsv file variable
const csvFile = files.getContent(reportCsv.id)

csv.parse(csvFile.buffer.toString('utf-8'), {
       auto_parse: true,
       columns: true,
}, (error, data) => {
       console.log(data)
})
```

9 Updating case information

The process variables always include the built-in Case variable, which contains information about the current case. Sometimes, you want to update this case information using data from process variables. You can update some of the this case variable's fields, as follows.

```
// Set the case name using a template.
_case.name = `Case ${_case.caseNumber}`

// Set the case’s due date using a date variable set on a form.
_case.dueDate = releaseDate

// Set the case’s priority, using text values '0' (high) to '3' (low)
// priorities defines constant values high, medium, normal, and low
const priorities = require('priorities')
_case.priority = priorities.low
```

A case name template can only use form trigger fields to set the case name when the process starts. However, when you can set the case name directly in a JavaScript action, you don’t have this restriction.

10 Loading user information

In a JavaScript action, you might need to select a SAP Signavio Process Governance user based on external data, to assign a role. To do this, you can use the built-in `users` API to find a user by their email address.

```
const users = require('users')
reviewer = users.findByEmail(reviewerEmailAddress)
```

This example uses the value of a previously-supplied `reviewerEmailAddress` **Email address** variable to set a `reviewer` **User** variable.

11 Calling an external web service

You can use the request module to call an external web service.

11.1 Fetching data in JSON format

The simplest example of calling an external web service is fetching data in JSON format. For example, the following script fetches JSON data and outputs a date to the script task's log.

```
const requestOptions = {
       url: 'http://www.mocky.io/v2/5bd71b95350000a039fd7f5b',
       auth: { username: 'api', password: 'xxxx-xxxx-xxxx-xxxx' },
       json: true
}

const handleResponse = (error, response, body) => {
       if (error) {
               console.error(error)
       }
       console.info(`HTTP ${response.statusCode} ${response.statusMessage}`)
       if (response.statusCode >= 400) {
               console.error(`Error: ${body}`)
               return
       }
       console.info(body.startDate)
}

request.get(requestOptions, handleResponse)
```

In the `requestOptions` object, setting the `json` property to `true` adds an `Accept:application/json` request header, and automatically parses the JSON response. The `auth` property adds HTTP Basic authentication to the request. If the *request* library cannot send the HTTP request, e.g. because the URL is invalid, the `handleResponse` function's first argument is a client error message. Otherwise, when the server returns an HTTP response, `response.statusCode` is the HTTP status code, which may indicate an HTTP error.

11.2 Sending data in JSON format

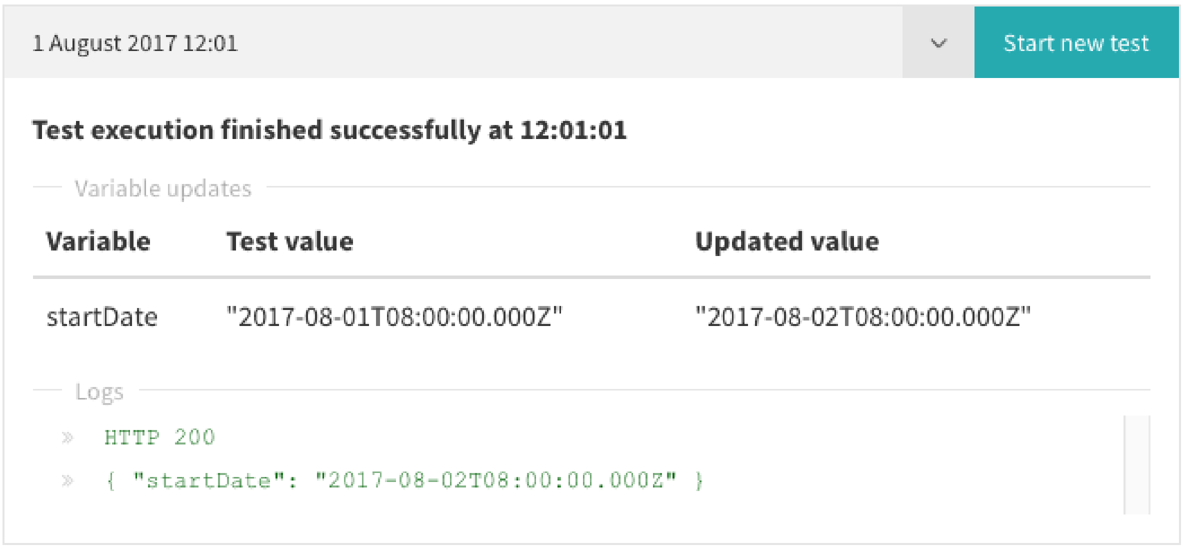
You can use process variables to send process data to an external web service. For example, the following script sends the value of the `startDate` variable in JSON format in an HTTP POST request to an external web service.

```
const requestOptions = {
    url: 'http://www.mocky.io/v2/5bc4b5573000005700758b2d',
    json: { startDate: startDate }
  }

  const handleResponse = (error, response, body) => {
    if (error) {
      console.error(error)
    }
    console.info(`HTTP ${response.statusCode} ${response.statusMessage}`)
    if (response.statusCode >= 400) {
      console.error(`Error: ${body}`)
      return
    }
    startDate = body.startDate
  }

  request.post(requestOptions, handleResponse)
```

This example uses a test endpoint configured using Mocky to return an HTTP response. This has the following result in the SAP Signavio Process Governance test console:



Test execution finished successfully at 12:01:01

Variable updates

Variable	Test value	Updated value
startDate	"2017-08-01T08:00:00.000Z"	"2017-08-02T08:00:00.000Z"

Logs

```
» HTTP 200
» { "startDate": "2017-08-02T08:00:00.000Z" }
```

Updating a variable via an external web service

The two log statements, starting with *HTTP 200*, show the HTTP response from the web service. The response body (as set-up in Mocky) contains JSON data that includes an updated value for the `startDate` variable, changing the date from `2017-08-01` to `2017-08-02`.

The script then parses this JSON response using `JSON.parse` and updates the `startDate` variable in SAP Signavio Process Governance, as shown in the **Updated value** column in the test console's variables table.

11.3 Sending form data

Although JSON is the most common data form for modern APIs, you sometimes need to simulate HTML forms, as follows.

```
const requestOptions = {
        url: 'https://requestinspector.com/inspect/01cv047qnc7q3w55wc1b7m1nmw',
        form: { startDate: startDate }
}

const handleResponse = (error, response, body) => {
        if (error) {
                console.error(error)
        }
        console.info(`HTTP ${response.statusCode} ${response.statusMessage}`)
        if (response.statusCode >= 400) {
                console.error(`Error: ${body}`)
        }
}

request.post(requestOptions, handleResponse)
```

This script sends the request to Request Inspector - a web-based HTTP request debugging tool.

In the script, using the `requestOptions.form` option to set the request data also sets the request content type to `application/x-www-form-urlencoded` that HTML forms use. Add the `json: true` request option if you expect a JSON response, as when Fetching data in JSON format.

11.4 Calling a SOAP 1.1 XML API

Some legacy web services use SOAP. To call a SOAP 1.1 API, set the request and response media types to `text/xml`, and use the *xml-js* JavaScript libraries to parse the XML response. To call a SOAP 1.2 API, use the `applic-ation/soap+xml` media type.

The script sets the `soapRequest` to a literal XML document, using variable interpolation inside backticks to set the `username` and `password` variables. Note that example code requires additional to escape XML special characters (`<`, `>` and `&`) in these variables.

12 Parsing CSV data

Comma Separated Values (CSV) is used as a common format to exchange tabular data between spreadsheets and relational databases. In a JavaScript action, you can parse CSV data using the `csv` library.

12.1 Literal CSV text

To start with a simple example, you can read CSV data from a JavaScript string.

```
const csv = require('csv')

  const csvData = `Employee,Employee email,Supervisor,Supervisor email
  Alice,alice@example.org,,
  Ben,ben@example.org,Alice,alice@example.org
  Charlie,charlie@example.org,Alice,alice@example.org
  Edward,edward@example.org,Ben,ben@example.org
  Fiona,fiona@example.org,Ben,ben@example.org`

  const logSupervisors = (error, rows) => {
    if (error) {
      console.error(error)
      return
    }
    rows.forEach(row => console.log(`Employee ${row['Employee']} has
supervisor ${row['Supervisor']}`))
  }

  const options = { auto_parse: true, columns: true }
  csv.parse(csvData, options, logSupervisors)
```

The `csv.parse` function has parameters for the CSV data, its own configuration options, and a function to pass the results to.

12.2 CSV file

If you upload a CSV file to a workflow, you can read its contents from a script task. To use a CSV file containing configuration data, you can use a core information event to assign a fixed file to a File variable.

```
  const _ = require('_')
  const csv = require('csv')
  const files = require('files')
  const users = require('users')

  const employeeColumn = 'Employee email'
  const supervisorColumn = 'Supervisor email'
  const options = { auto_parse: true, columns: true, relax_column_count:
true }
  const csvContents = files.getContent(employeesCsv.id)

  csv.parse(csvContents.buffer.toString('utf-8'), options, (error,
employeeList) => {
    if (error) {
      console.log(`Error parsing CSV: ${error}`)
      return
    }
    if (employeeList.length === 0) {
      console.log(`Error: CSV contains zero rows`)
      return
    } else {
      console.log(`${employeeList.length} rows read from CSV`)
    }
    if (!employeeList[0][employeeColumn]) {
      console.log(`Error: missing '${employeeColumn}' column`)
      return
    }

    const entry = _.find(employeeList, [employeeColumn,
selectedEmployee.emailAddress])
    if (!entry) {
      console.log(`Error: no entry for employee
'${selectedEmployee.emailAddress}'`)
      return
    }
    console.log(`Found entry: ${JSON.stringify(entry)}`)

    if (!entry[supervisorColumn]) {
      console.log(`Error: missing '${supervisorColumn}' column for
employee`)
      return
    }

    selectedSupervisor = users.findByEmail(entry[supervisorColumn])
    if (!selectedSupervisor) {
      console.log(`Error: no SAP Signavio Process Governance user account
found for '${entry[supervisorColumn]}'`)
      return
    }
    console.log(`${selectedEmployee.firstName} has supervisor
${selectedSupervisor.firstName}`)
  })
```

This example uses two input variables: a File variable called *Employees CSV*, and a *Selected employee* User variable. The file contains CSV data with the same structure as the Literal CSV text example.

The code searches the CSV for an employee with the same email address as the *Selected employee*. When it finds the employee, it uses the `users` library to retrieve the SAP Signavio Process Governance user the corresponds to the supervisor email address. Finally, the script updates the *Selected supervisor* User output variable.

This example shows that parsing CSV files requires several layers of error handling, because the structure may be slightly wrong.

12.3 CSV via HTTP

Instead of including CSV data in a workflow, you can fetch it via a web service request. Google Sheets, for example, has a *Publish to web* feature that makes a spreadsheet available as CSV.

```
const csv = require('csv')

const fetchCsv = () => new Promise((resolve, reject) => {
        const csvUrl = 'https://docs.google.com/spreadsheets/d/e/2PACX-
1vR2qpKHYs3Zp1cjLLYAcxF1kkXqu_1dRN-j8dG4qvDFX0y-
YPoDvD4unzKNHTl8e65U9SMLFuhDMJDU/pub?gid=1608745320&single=true&output=cs
v'
        request.get(csvUrl, (error, response, body) => {
                if (error) {
                        reject(error)
                }
                console.info(`HTTP ${response.statusCode} ${response.statusMessage}`)
                if (response.statusCode >= 400) {
                        reject(new Error(body))
                }
                resolve(body)
        })
})

const parseCsv = (csvText) => new Promise((resolve, reject) => {
        const options = { auto_parse: true, columns: true }
        csv.parse(csvText, options, (error, rows) => {
                if (error) {
                        reject(error)
                }
                const supervisors = _.mapValues(_.keyBy(rows, 'Employee email'),
'Supervisor email')
                resolve(supervisors)
        })
        })

fetchCsv()
        .then(parseCsv)
        .then(supervisors => console.log(JSON.stringify(supervisors)))
        .catch((error) => console.error(error))
```

The code defines two JavaScript Promise objects, to fetch the CSV then parse it. The `parseCsv` promise transforms the result of parsing the CSV into a JavaScript object that maps employee to supervisor email addresses. At the bottom, the script executes both promises and logs the result:

```
{
        "alice@example.org": "",
        "ben@example.org": "alice@example.org",
        "charlie@example.org": "alice@example.org",
        "edward@example.org": "ben@example.org",
        "fiona@example.org": "ben@example.org"
}
```

13 Generating and parsing XML

XML is used as a common exchange format for data. In a JavaScript action you can parse and generate XML using the `xml-js` library.

The first example parses XML content of a local variable into a JavaScript object structure. You can also retrieve XML content from another source, like a process variable or an external web service.

The example uses the compact mode by setting the option `{ compact: true }` when calling the conversion function. The compact mode creates an object structure which resembles XML structure. It contains special properties like `_attributes` and `_text` which allow you to access the text content of XML elements and attributes.

```
const convert = require('xml-js')

const xml = `
<customers>
       <customer status="silver">John Doe</customer>
       <customer status="gold">Alice Allgood</customer>
</customers>
`

// Parse the XML string to a JavaScript object using the compact mode
const obj = convert.xml2js(xml, { compact : true})

const john = obj.customers.customer[0]
console.log(`${john._text} - ${john._attributes.status}`)
```

The second example shows how to generate XML from a JavaScript object. Note that the structure also contains the special properties `_attributes` and `_text` to indicate which parts of the object will be XML attributes or text content.

This example use compact mode to set the respective option.

```
const convert = require('xml-js')

const obj = {
       customers : {
               customer: [
                       {
                               _text: "John Doe",
                               _attributes: {
                                       status: "silver"
                               }
                       },
                       {
                               _text: "Alice Allgood",
                               _attributes: {
                                       status: "gold"
                               }
                       }
```

```
                ]
        }
}

// Generate the XML string from a JavaScript object
const xml = convert.js2xml(obj, { compact: true })

console.log(xml)
```

See the xml-js library documentation for more information about generation and parsing options.

14 Exchanging files with a WebDAV endpoint

In a JavaScript action, you might want to upload a file that has been uploaded to a task form to your own file server or load a file from an external server to use the content during the script execution. If your file server supports the WebDAV protocol, you can use the `webdav-client` library.

In this example, the content of the file variable `myFile` is uploaded to the endpoint `https://webdav.example.com`. When uploading the file content make sure to use the `buffer` property as the library expects either a buffer or a string for the file content.

```
const webdav = require('webdav-client')
const files = require('files')

// Create an authenticated connection to the WebDAV endpoint
const connectionOptions = {
       url:'https://webdav.example.com',
       authenticator: new webdav.BasicAuthenticator(),
       username:'signavio',
       password:'WyvsWGxWUtx3wTw1BkmfOtf4'
}
const connection = new webdav.Connection(connectionOptions)

// Read the content of the myFile file variable
const fileContent = files.getContent(myFile)

// Upload the content to the root directory using the original file name
connection.put(`/${myFile.name}`, fileContent.buffer, (error) => {
       if (error) {
               throw new Error(`File upload did not work: ${error}`)
       }
})
```

As an alternative, it is also possible to generate a string with the file content and upload it.

```
const webdav = require('webdav-client')

// Create a connection to your WebDAV endpoint
const connection = new webdav.Connection('https://webdav.example.com')
// Create the file content
const myNotes = 'The answer is 42!'

// Upload the content to the file myNotes.txt in directory /path/to
connection.put('/path/to/myNotes.txt', myNotes, (error) => {
       if (error) {
               throw new Error(`File upload did not work: ${error}`)
       }
})
```

Reading a file via WebDAV is as simple as uploading one. Instead of `put` use `get` and specify the desired file. The callback offers the second parameter `body` which contains the file content.

```
const webdav = require('webdav-client')

// Create a connection to your WebDAV endpoint
const connection = new webdav.Connection('https://webdav.example.com')

connection.get(`/foobar.txt`, (error, body) => {
        if (error) {
                throw new Error(`File upload did not work: ${error}`)
        }
        console.log(body)
})
```

See the WebDAV library documentation for more methods to modify your files and additional request options. The documentation also explains how you can authenticate to your WebDAV endpoint.

15 SAP Signavio Process Manager API client library

For the integration of SAP Signavio Process Governance and SAP Signavio Process Manager, read more in section Dictionary Integration.

The SPM API client library simplifies HTTP requests to the SAP Signavio Process Manager (SPM) API in JavaScript tasks. It offers JavaScript methods to send HTTP GET, POST, PUT and DELETE requests. The library automatically authenticates requests using the account configured for the SAP Signavio Process Manager or Dictionary integration. All retrievable information is limited to what this account is allowed to access.
Import the library `spm-client` within a JavaScript task to use the SPM client.

```
const spmClient = require('spm-client')
```

The client offers four methods:

- `get('/path/to/endpoint', { key: 'value' })`
- `post('/path/to/endpoint', { key: 'value' })`
- `put('/path/to/endpoint', { key: 'value' })`
- `delete('/path/to/endpoint', { key: 'value' })`

Two parameters can be passed with the endpoint:

- The mandatory API path that you want to call.
- An optional object with request parameters. See the request library documentation for more information about all possible request parameters.

The response is a Promise providing the result of the request. In case of a successful request the response body is provided to the `then` function. In case of an error an error object to the `catch` function is provided.
The error object consists of the following three properties:

- `message` an error message
- `response` (optional) the response object
- `body` (optional) the response body

Example:

```
const spmClient = require('spm-client')
spmClient.get('/p/config', {
  json: true
})
  .then(body => {
    console.log(body)
  })
  .catch(error => console.log(error.message))
```

The example above sets the request parameter json: true. This setting sets the request content type to JSON, and will parse the text response body automatically as JSON.

15.1 Include URL-encoded form content in a request

The SAP Signavio Process Manager API often requires you to send a request body as a URL encoded form. You can use the request parameter form to provide the body for the request.

Example:

```
const spmClient = require('spm-client')
spmClient.post('/path/to/endpoint', {
  form: {
    key: 'value'
  }
})
  .then(body => {
    console.log(body)
  })
  .catch(error => {
    console.log(error.message)
  })
```

15.2 Extract a diagram attribute from a model

The following example extracts the diagram attribute processowner from a model in SAP Signavio Process Manager. The property modelId is provided via a variable from the workflow.

Example:

```
const spmClient = require('spm-client')
spmClient
  .get(`/p/model/${modelId}/json`, { json: true })
  .then(model => {
    // You can also log the diagram properties if you are looking for a
specific property
```

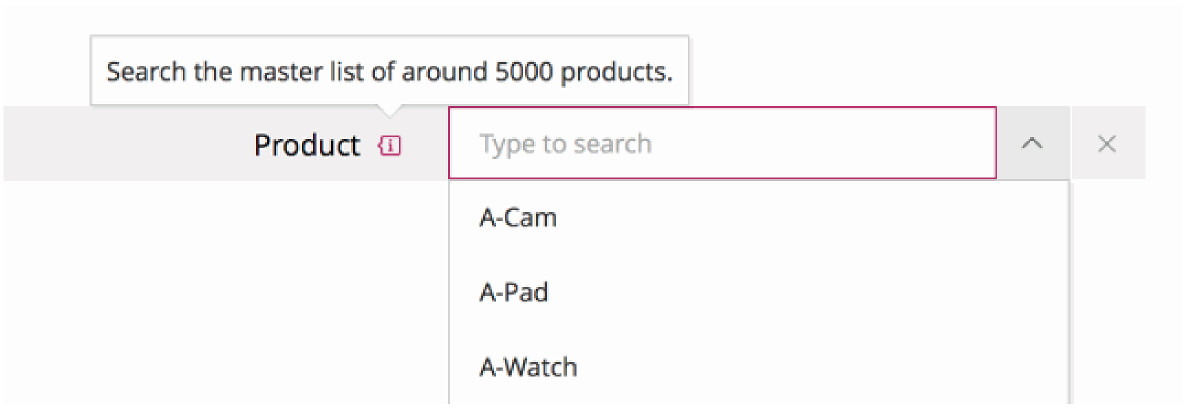
```
    // be aware all custom attribute names start with 'meta-'
console.log(model.properties)

    // notation to extract default attribute named "processowner"
    processOwner = model.properties.processowner

    // notation to extract custom attribute named "additionalInformation"
    customAttribute = model.properties['meta-additionalInformation']
  })
  .catch(error => {
    console.log(error.message)
  })
```

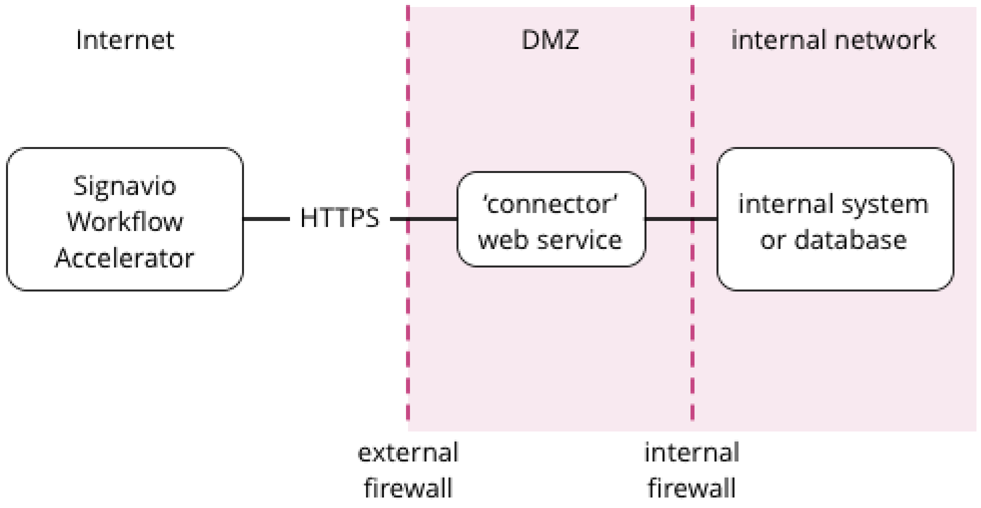
16 Custom data connectors

When you define a process in SAP Signavio Process Governance, you often include your own data in the process definition, such as the list of options for a form field. This works well for small lists that don't change often or that belong to the process, such as a list of document statuses in a document approval process. However, fixed lists in the process definition become difficult to maintain when the data changes frequently or includes a large number of items, such as a list of products or customers.



Use a custom data connector to populate choice options from an external database

With SAP Signavio Process Governance, you can also integrate dynamic structured data from other IT systems into your workflows. The workflow system fetches data from a third-party system using a *connector*, which you implement and host yourself.



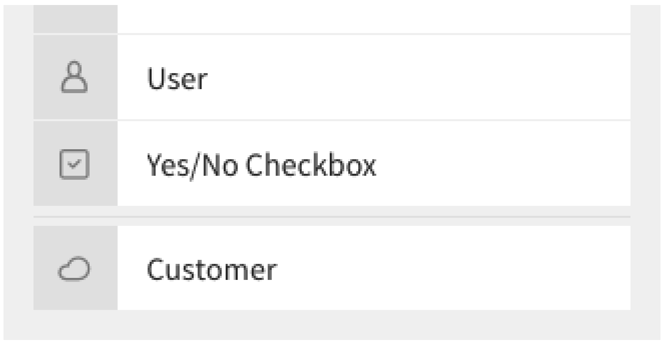
Connector deployment scenario for connecting to a corporate internal system

This diagram shows a typical scenario, with a connector that sends data from an internal database to SAP Signavio Process Governance (SaaS), via the Internet. The connector web service runs in your corporate network's **demilitarized zone** (DMZ), which your IT department configures to provide more security than allowing SAP Signavio Process Governance to connect directly to your internal network.

A connector provides a web service that translates between the external system and SAP Signavio Process Governance. The connector implements a defined interface, which SAP Signavio Process Governance uses to access data in a format it can use. SAP Signavio Process Governance and the connector communicate over *HTTP* or *HTTPS*, which makes it possible to implement connectors in any programming language.

17 Using a connector

A connector can provide data to user task form fields. For example, you can create a connector that provides a list of customers, which adds a **Customer** type in the form builder:



*A **Customer** connector type in the form builder, at the bottom of the list of field types*

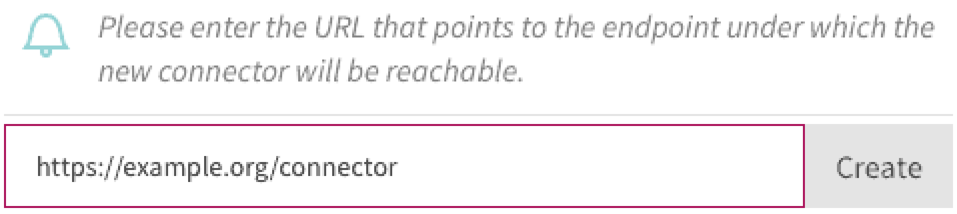
A connector reference field:

- makes it possible to select from a dynamic list of records
- supports auto-complete so you can work with a large number of records
- can include structured data for each record.

18 Configuring a connector

To configure connectors, on the top-right menu, select **Services & Connectors**, then select the **Connectors** tab. When you have published your connector, you can add it here.

Select **Add new connector** and enter the connector's endpoint URL.

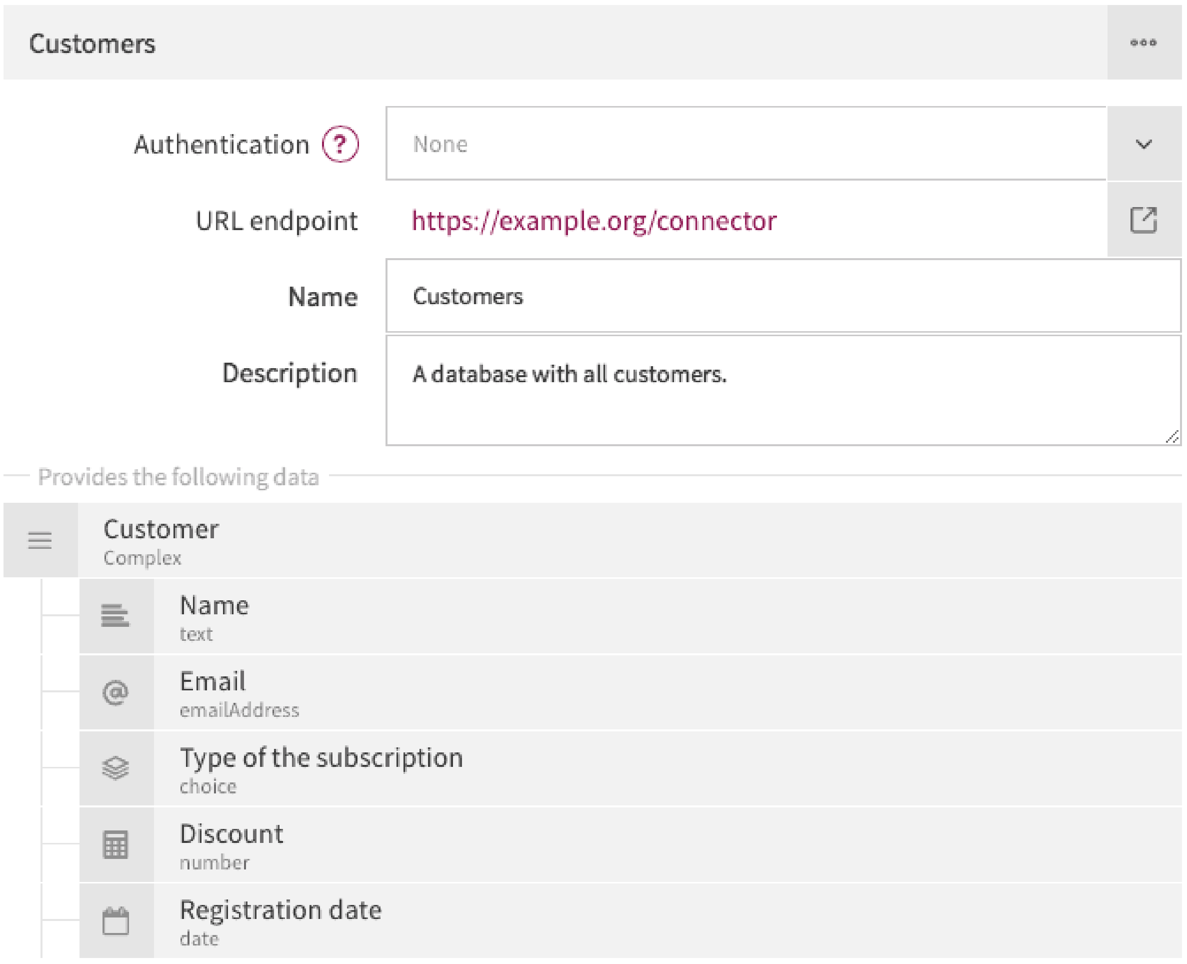


Please enter the URL that points to the endpoint under which the new connector will be reachable.

https://example.org/connector

Adding a new connector with the endpoint URL `https://example.org/connector`

When you add a connector, SAP Signavio Process Governance fetches the connector descriptor and shows a summary:



Customers

Authentication	None
URL endpoint	https://example.org/connector
Name	Customers
Description	A database with all customers.

Provides the following data

- Customer — Complex
  - Name — text
  - Email — emailAddress
  - Type of the subscription — choice
  - Discount — number
  - Registration date — date

Connector summary, including an overview of record and field types

If you make changes to your connector, such as adding or renaming a field, you need to reload the configuration. On the connector’s top-right menu, select **Reload connector** to fetch the latest version of the descriptor.

18.1 Deleting a connector

You can delete a connector if you no longer wish to use it. On the connector’s top-right menu, select **Delete connector** to remove its configuration from SAP Signavio Process Governance. If you delete a connector by mistake, select **Add new connector** and enter the endpoint URL again.

19 Authentication

Publishing a connector makes it publicly accessible, as well as any data that the connector provides. To prevent unauthorized access, the connector can implement authentication, so that only SAP Signavio Process Governance can access the data. Connectors may use one of two authentication mechanisms.

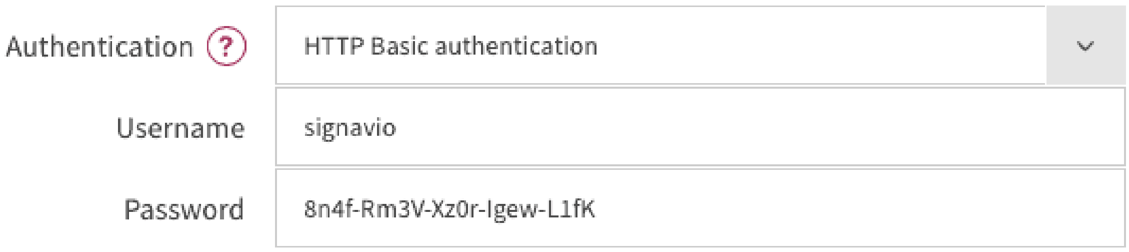
19.1 HTTP Basic authentication

Connectors can use **HTTP basic authentication** to restrict access using a user name and password that you specify when configuring the connector. To implement HTTP Basic authentication, your connector endpoints must:

1. send an HTTP *401 Unauthorized* response, with an empty response body, for any request that does not include valid credentials
2. check the credentials in the `Authorization` HTTP header field, when provided, by decoding the Base64-encoded user name and password and verifying their values.

HTTP Basic authentication sends an unencrypted password over the network, so you should only allow access to private connectors via HTTPS.

To use basic authentication, use the *Authentication* field to select *HTTP Basic authentication*, and enter a user name and password:



Authentication	HTTP Basic authentication
Username	signavio
Password	8n4f-Rm3V-Xz0r-Igew-L1fK

Configuring basic authentication

When you configure a connector to use Basic authentication, SAP Signavio Process Governance will pre-emptively include an `Authorization` header when sending requests to the connector endpoints. In Basic authentication, the header value consists of the authentication scheme name `Basic` followed by a space and the Base64-encoded user name and password, separated by a colon (`signavio:8n4f-Rm3V-Xz0r-Igew-L1fK`). This results in a request header that looks like:

```
Authorization: Basic c2lnbmF2aW86OG40Zi1SbTNWLVh6MHItSWdldy1MMWZL
```

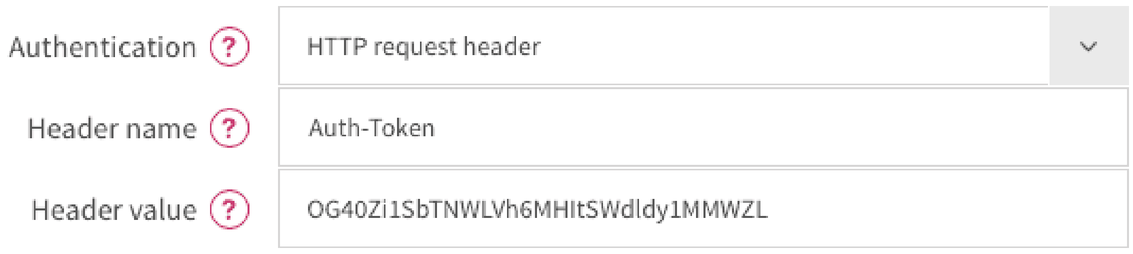
Sending this header with every request avoids an additional *401 Unauthorized* response and a new request for the authentication challenge.

19.2 Token authentication

Similar to an API key, you can choose a password (token) that SAP Signavio Process Governance will include in a request header field or URL query string, for every request it sends to the connector endpoints. In the connector configuration, you can choose between a request header field or a URL query string parameter, and specify the header or parameter name.

> Token authentication sends an unencrypted password over the network, so you should only allow access to private connectors via HTTPS.

The connector endpoints can then authenticate requests by checking the respective header field or query string parameter value.
To use a token in the request header, use the *Authentication* field to select *HTTP request header*, and enter a header name and header value.



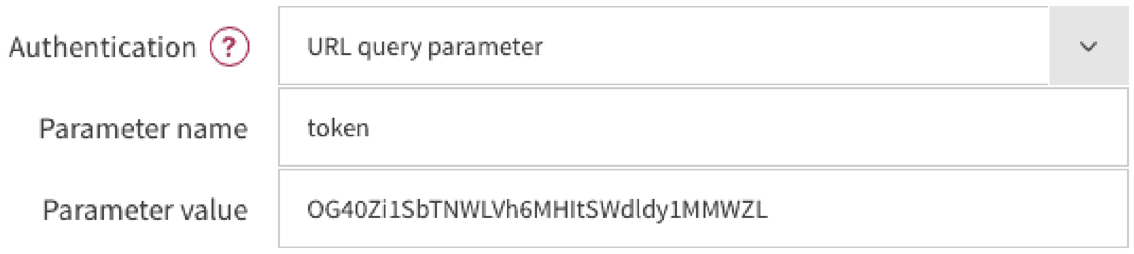
Authentication	HTTP request header
Header name	Auth-Token
Header value	OG40Zi1SbTNWLVh6MHItSWdldy1MMWZL

Configuring request header authentication

HTTP headers only allow a restricted subset of ASCII characters in header names, which typically only use letters and dashes, such as Auth-Token. Header values only support ‘visible ASCII characters’, so to allow arbitrary authentication tokens, use a Base64-encoded value. Configuring token authentication results in a request header like:

```
Auth-Token: OG40Zi1SbTNWLVh6MHItSWdldy1MMWZL
```

For testing, developers may find it more convenient to retrieve the authentication from the URL query string. To use this option, select *URL query parameter* and enter a parameter name and value:



Authentication	URL query parameter
Parameter name	token
Parameter value	OG40Zi1SbTNWLVh6MHItSWdldy1MMWZL

Configuring URL query string parameter authentication

This results in HTTP requests with a URL query string, like this:

```
GET /?token=OG40Zi1SbTNWLVh6MHItSWdldy1MMWZL HTTP/1.1

Host: example.org
```

HTTP does not encrypt query string parameters, which typically appear in log files, so only use query string token authentication for testing a connector on a trusted network with the on-premise edition of SAP Signavio Process Governance, and switch to a header field token for production use.

20 Record storage

SAP Signavio Process Governance supports two different ways of storing selected records: by reference or by value. The following table compares the two options.

Storing records by *value*	Storing records by *reference*
Recommended option for most use cases	Default option, for backwards compatibility with older connector implementations
Stores the entire record in the case	Stores the record ID in the case
Reads the whole record once, when selected	Read the record each time it is displayed
Cases, tasks and reports load faster	Cases, tasks and reports load slower
The case shows the record as it was when selected	The case shows the record's latest value
You can use connector types in reports	You cannot use record fields in filters, grouping or aggregation

The connector specifies each record type's storage option in its record type descriptor's `recordType` property. If the connector offers multiple types, they can use a mixture of *value* and *reference* options.

When you store records by value, the connector must always include a `name` field in the record data. SAP Signavio Process Governance uses this name to display record values.

21 Implementing a connector

To implement a connector, you make data available via four URLs, such as:

```
https://example.org/connector

https://example.org/connector/customer/options?filter=ACME

https://example.org/connector/customer/options/42

https://example.org/connector/customer/42
```

These four URLs correspond to four different kinds of resource.

1. Connector descriptor - defines one or more record types, each of which defines a list of fields.
2. Record type options - a list of records for each record type the connector defines.
3. Record type option (single option) - a single record from the `connector-type-options` list.
4. Record details (optional) - all fields for one record from the list of records.

SAP Signavio Process Governance accesses the connector on the web, via the public Internet, or via a private intranet for an on-premise installation. SAP Signavio Process Governance calls the connector's URL the *endpoint URL*.

For example, consider a connector that accesses a fictional customer database, that you publish at the endpoint URL `https://example.org/connector`. In this example, each customer record has the following fields.

Example - customer record fields

Property	Description
`id`	Unique identifier
`name`	Full name
`email`	Email address
`subscriptionType`	Type of subscription - bronze, silver or gold
`discount`	Default customer discount
`since`	Registration date

A complete example customer record, formatted as `JSON`, would then look like this:

```
{
         "id" : "7g8h9i",
         "name" : "Charlie Chester",
         "email" : "charlie@example.org",
         "subscriptionType" : "silver",
         "discount" : 15,
         "since" : "2012-02-14T09:20:00.000Z"
       }
```

This example now includes enough information to implement a complete connector.

21.1 Connector descriptor

A connector needs a descriptor to provide basic information, such as its name and description, as well as detailed information about the structure of the data the connector provides. When you implement a connector, you must make the descriptor available as the following HTTP resource.

URL	`/` - the connector's *endpoint URL*
Request methods	GET - fetches the connector descriptor
Response content type	`application/json`
Response body	A JSON object with the following fields.

Connector descriptor properties

Property	Description
`key`	Unique alphanumeric key (characters a-z, A-Z, 0-9) that identifies the connector
`name`	The connector name shown in the user interface
`description`	Detailed connector description
`typeDescriptors`	List of one or more descriptors for record types
`version`	The connector version, which should increase if the provided data structure changes
`protocolVersion`	The connector protocol version, currently `1`.

For example, the JSON response body for a connector descriptor without any type descriptors would look like this:

```
{
         "key" : "customers",
         "name" : "Customers",
         "description" : "A database with all customers.",
         "typeDescriptors" : [ ],
         "version" : 1,
         "protocolVersion" : 1
       }
```

In our example, you would retrieve the connector descriptor by sending the HTTP request `GET https://example.org/connector/`.

21.1.1 Record type descriptor

A **record type descriptor** describes the format of the data the connector provides, such as the format of a customer record. In the JSON response, the `typeDescriptors` property's value contains an array of record type descriptor JSON objects.

Record type descriptor properties

Property	Description
`key`	Unique alphanumeric key (characters a-z, A-Z, 0-9) that identifies the record type within the connector descriptor, used in Record type options and Record details URLs
`name`	The type name shown in the form builder user interface
`recordType`	Either `reference` or `value` - specifies how SAP Signavio Process Governance stores selected values, see Record storage
`fields`	An array of record type descriptors
`optionsAvailable`	Boolean value - `true` indicates that the connector provides a list of record options, used to provide a list in the user interface for user selection
`fetchOneAvailable`	Boolean value - `true` indicates that SAP Signavio Process Governance can fetch single records by the ID used in the options list

For example, the JSON object for a customer record type descriptor, with only the default ID field, would look like this:

```
{
    "key" : "customer",
    "name" : "Customer",
    "recordType" : "value",
    "fields" : [
      {
        "key" : "id",
        "name" : "ID",
        "type" : {
          "name" : "text"
        }
      }
    ],
    "optionsAvailable" : true,
    "fetchOneAvailable" : true
  }
```

A **record field descriptor** specifies one field of a record type. A record type has a complex structure that includes one or more fields, such as a customer’s full name. Each field has a key, a name and a data type.

Record field descriptor properties

Property	Description
`key`	Unique alphanumeric key (characters a-z, A-Z, 0-9) that identifies the field type within the record type
`name`	The field name shown in the user interface
`type`	A JSON object that describes field's data type - see Data types and formats

Every record type automatically includes `name` (when storing records by value) and `id` fields with type `text`, so you don't have to define them explicitly. However, you must define at least one field, so define the `id` field explicitly if you have no other fields.

An example for the `email` of our customer record type looks like this :

```
        {
         "key" : "email",
         "name" : "Email",
         "type" : {
           "name" : "emailAddress"
         }
       }
```

A complete example of our connector descriptor would look like this:

```
{
         "key" : "customers",
         "name" : "Customers",
         "description" : "A database with all customers.",
         "typeDescriptors" : [ {
           "key" : "customer",
           "name" : "Customer",
           "recordType" : "value",
           "fields" : [ {
             "key" : "email",
             "name" : "Email",
             "type" : {
               "name" : "emailAddress"
             }
           }, {
             "key" : "subscriptionType",
             "name" : "Type of the subscription",
             "type" : {
               "name" : "choice",
               "options" : [
                       {
                               "id" : "bronze",
                               "name" : "Bronze"
```

```
                }, {
                        "id" : "silver",
                        "name" : "Silver"
                }, {
                        "id" : "gold",
                        "name" : "Gold"
                }
        ]
      }
    }, {
      "key" : "discount",
      "name" : "Discount",
      "type" : {
        "name" : "number"
      }
    }, {
      "key" : "since",
      "name" : "Registration date",
      "type" : {
        "name" : "date",
        "kind" : "datetime"
      }
    } ],
    "optionsAvailable" : true,
    "fetchOneAvailable" : true
  } ],
  "version" : 1,
  "protocolVersion" : 1
}
```

21.2 Record type options

When you use a record type on a form, you will see a form field where you can enter a search query and select one of the options shown. Each result represents a record provided by the connector. In order to show a selection of different records to the user, a connector can provide a list of options for a record type.

To make a list of options available to forms, in the Connector descriptor, set the `optionsAvailable` flag to `true`. The connector must also make the options available as the following HTTP resource.

URL (relative to the endpoint URL)	`/:type/options` - with path parameter `:type` (a record type key)
Query string (optional)	`filter=:query` - added when the user enters a search; `:query` encodes the search string
Request methods	GET - fetches the list of record type options
Response content type	`application/json`
Response body	An array of JSON objects, which should have a limited maximum length. Each object in the array must have the following fields.

Record type options object properties

Property	Description
`id`	Unique string record ID
`name`	The text label shown in the user interface, which could aggregate multiple record fields like `name (email)`

For example, a list of customer options, with URL `https://example.org/connector/customer/options`, would look like this:

```
[ {
        "id" : "1a2b3c",
        "name" : "Alice Allgood"
      }, {
        "id" : "4d5e6f",
        "name" : "Ben Brown"
      }, {
        "id" : "7g8h9i",
        "name" : "Charlie Chester"
      } ]
```

21.3 Record type option (single option)

After someone selects an option, the case user interface may later display the selected option in other contexts. Connectors that set the `optionsAvailable` flag to `true` must also make it possible to look up a single option by its ID, in order to display the option name.

URL (relative to the endpoint URL)	`/:type/options/:id` - with path parameters `:type` (a record type key) and `:id` (the option ID)
Request methods	GET - fetches a single record type option
Response content type	`application/json`
Response body	A single JSON object, with the same fields as the objects in the Record type options response.

For example, a single customer option, with URL `https://example.org/connector/customer/options/1a2b3c`, would look like this:

```
{
        "id" : "1a2b3c",
        "name" : "Alice Allgood"
      }
```

21.4 Record details

When you use a connector form field to select a record, you can use the record's data in the workflow. Depending on Record details, SAP Signavio Process Governance stores the whole record or only the record's ID as a reference, and fetches the entire record either directly or when needed, when accessing the nested data.
To make a record's fields available, in the Connector descriptor, set the `fetchOneAvailable` flag to `true`. The connector must also make the records available as the following HTTP resource.

URL (relative to the endpoint URL)	`/:type/:id` - with path parameters `:type` - a record type key, and `:id` - a record ID
Request methods	GET - fetches details for a single record
Response content type	`application/json`
Response body	A JSON object containing all fields of the record with the requested ID.

For example, a customer record, with URL `https://example.org/connector/customer/7g8h9i`, would look like this:

```
{
          "id" : "7g8h9i",
          "name" : "Charlie Chester",
          "email" : "charlie@example.org",
          "subscriptionType" : "silver",
          "discount" : 15,
          "since" : "2012-02-14T09:20:00.000Z"
        }
```

Selecting this customer record from the customer options list would give the workflow access to all of this customer’s fields. As well as the custom fields, there are three pre-defined properties, shown in the following table.

Record details properties

Property	Description
`id`	The text label shown in the user interface (required)
`name`	The text label shown in the user interface (required when storing records by value)
`version`	A string indicating the version of the record (optional)

22 Adding connector parameters

Connectors can send large amounts of data and options. The list of options may depend on other field values (defined previously or used in the same form).
When parameters are added to a connector, the returned options are limited or filtered, based on parameter values. Parameters are sent when the option fetch request is sent to the connector.

22.1 Using connector parameters

With connector type parameters, any field value available in the same process can be passed from the form to the connector when requesting options. Once available for the connector type, parameters can be specified in the form field configuration. Based on the passed parameter value, the list of available options in the connector field drop-down list changes.
Parameters are applied to one specific field in a form. If you use several fields of the same connector type in one task form, you can specify different sets of parameters for each field.

Parameter values should not be used to implement security features (e.g. restricting view access to connector options).

22.2 Connector descriptor with parameters

A type descriptor (see section Connector descriptor) defines one or multiple parameters.
The following parameter definition is necessary for SAP Signavio Process Governance:

Record field descriptor properties

Property	
`key`	Unique alphanumeric key (characters a-z, A-Z, 0-9) that identifies the parameter within the type descriptor
`name`	Parameter name shown in the form builder user interface
`type`	Simple data type available in SAP Signavio Process Governance (for example text, number, boolean)

To use parameters for connector types, you have to extend the connector descriptor provided by the connector web service.

22.3 Example

For a production plant, the complete list of storage areas is returned from an options request to the connector. The options request looks like https://-some.example/connector/storageAreas/options.
One department only needs a list of storage facilities with ventilation. These areas are defined by a field provided by the case currently active in SAP Sig-navio Process Governance. In our example this field is called *storageFeature*. You set the field to the parameter value *ventilated*. This value is used to retrieve only storage areas matching the parameter value *ventilated*. The changed options request looks like this: https://-some.example/connector/storageAreas/options?storageFeature=ventilated
It is possible to specify multiple parameters of different types with one con-nector type. The following JSON example shows a connector descriptor which has the type `areas` with the parameter `storageFeature` of type `text`:

```
{
    "key" : "storageAreas",
    "name" : "Example Areas",
    "description" : "Example to demonstrate the implementation for
parameters",
    "typeDescriptors" : [ {
      "key" : "areas",
      "name" : "Areas",
      "optionsAvailable" : true,
      "fetchOneAvailable" : true,
      "recordType" : "value",
      "parameters" : [ {
        "key" : "storageFeature",
        "name" : "Storage feature",
        "type" : {
          "name" : "text"
        }
      } ],
      "fields" : [ ]
    } ]
  }
```

22.4 Binding values to connector fields

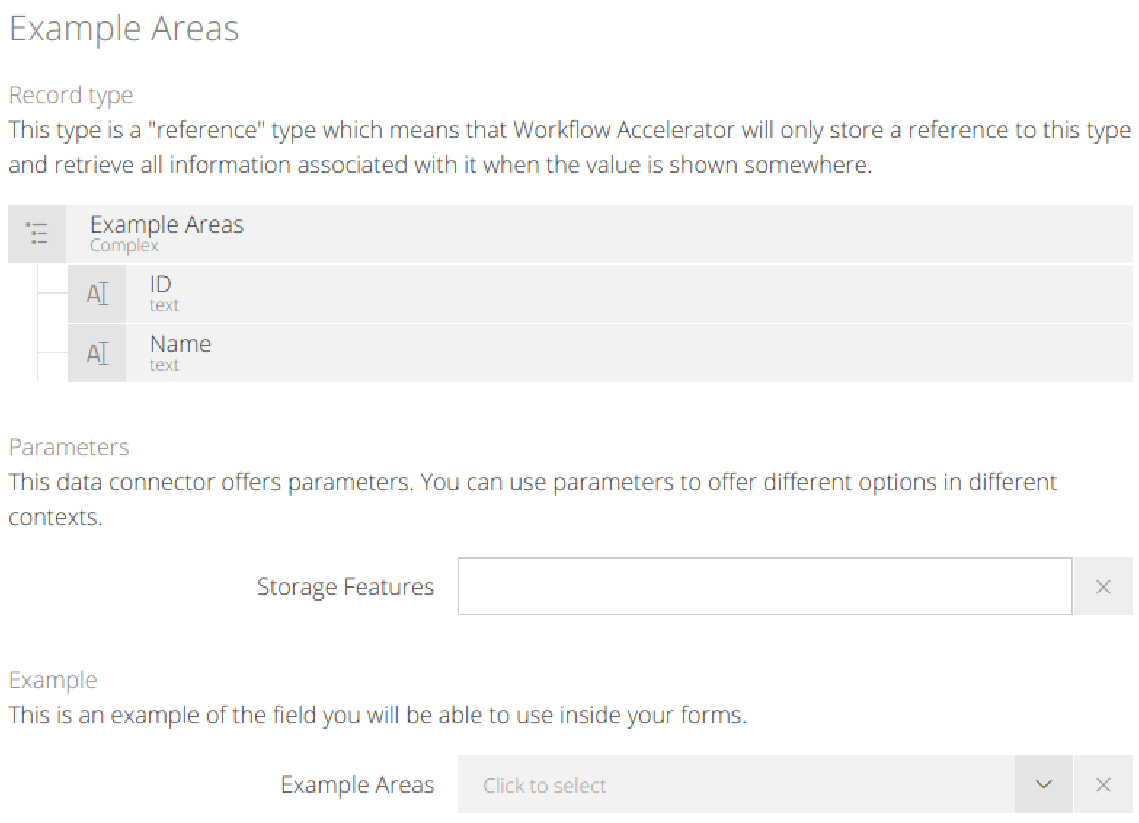
After adding a connector, the data types sent by this connector are available in SAP Signavio Process Governance. When adding one of the parameterized fields to the form builder, the configuration dialog shows the available parameters. Into the field, you can enter static values or reference other workflow variables available in the current workflow into the field. The variables need to match the type specified in the type descriptor.

22.5 Fetching option parameters

When opening the options drop-down list of the parameterized field (*Areas* in our example), the changed options request result is listed. In our example, the options request URL is https://some.example/connector/storageAreas/options?storageFeature=ventilated.

22.6 Testing parameter values

You can test the responses for certain parameter values on the *Services & Connectors* page. For our example the `areas` type is listed. For the given parameter (in this case `storageFeature`) values can be selected from a drop-down list to show the values returned from the connector when fetching the options.



Example Areas

Record type

This type is a "reference" type which means that Workflow Accelerator will only store a reference to this type and retrieve all information associated with it when the value is shown somewhere.

Example Areas
Complex

- ID
  text
- Name
  text

Parameters

This data connector offers parameters. You can use parameters to offer different options in different contexts.

Storage Features:

Example

This is an example of the field you will be able to use inside your forms.

Example Areas: Click to select

23 Connector examples

To help you develop your own connectors, SAP Signavio has published example connectors that show you what a connector implementation looks like. These examples show different programming languages, including Java, Python, JavaScript, Scala and Go:

https://github.com/signavio/connector-examples

These examples have an open-source Apache License.

24 Data types and formats

A data type defines which kind of value and format a field in a record can have. In some JavaScript actions, a type may make additional calculated properties available.

A type descriptor represents a data type as a JSON object, whose `name` property contains the data type name. You use these type descriptors to specify connector field types in a Connector descriptor. Type descriptors may use additional properties for type-specific configuration. Furthermore, the expected format of a record value depends on the data type.

24.1 Case

A case value represents a SAP Signavio Process Governance case's metadata:

```
  {
    "id" : "5b0fd957d1dfff6f53e08b2c",
    "name" : "Approve Q1 report",
    "caseNumber" : 1  ,
    "creatorId" : "59a954b862f8a0632524ba24",
    "createTime" : "2018-05-31T11:15:35.669Z",
    "dueDate" : "2018-07-24T21:59:59.999Z",
    "priority" : "0",
    "link" :
"https://workflow.signavio.com/55207c30e4b08cd0c2906280/cases/case/5b5043
20d1d0ff20f4a0b360",
    "milestone" : "New"
  }
```

This type provides the following properties.

Property	Type	Values
`id`	ID	Unique identifier
`name`	Text	Entered or generated editable name
`caseNumber`	Number	Sequential case number
`creatorId`	User	User who created the case
`createTime`	Date	Date and time the *Creator* started the case
`dueDate`	Date	Optional editable due date
`priority`	Text	Case priority - values `'0'` (high) to `'3'` (low)
`link`	Text	URL of the case page in SAP Signavio Process Governance
`milestone`	Text	The last milestone the case has passed

24.2 Choice

A choice type represents a value from a fixed list of configured options. The value stores the string value of the selected option's `id` property:

```
"value" : "g"
```

A type descriptor for a choice type with three options looks like this:

```
        "type" : {
         "name" : "choice",
         "options" : [
           {
             "id" : "b",
             "name" : "Bronze"
           },
           {
             "id" : "s",
             "name" : "Silver"
           },
           {
             "id" : "g",
             "name" : "Gold"
           }
         ]
       }
```

The type descriptor's `options` property value stores a JSON array of choice option objects. Each option has `id` and `name` properties.

A choice option's `id` property stores a unique alphanumeric key (characters a-z, A-Z, 0-9) that identifies the option within the choice type; no two options may have the same `id`. The user interface shows the `name` property's value to the user.

24.3 Date

A date value represents either a date and time (such as *2012-02-14 09:20*), just a date (*2012-02-14*), or just a time (*09:20*). Date values must always use the `YYYY-MM-DDThh:mm:ss.SSSZ` ISO 8601 date format and the UTC time zone:

```
"value" : "2012-02-14T09:20:00.000Z"
```

The type descriptor has a `kind` property with the value `date`, `time`, or `datetime` that specifies whether the value describes a date, a time of day or both (required):

```
"type" : {
         "name" : "date",
         "kind" : "datetime"
       }
```

For `date` and `time` values, execution only uses the date or time part of the value, respectively.

24.4 Duration

A duration value stores the length of a period of time, such as *2 weeks*, as a number of seconds:

```
"value" : 1209600
```

The type descriptor has no configuration properties:

```
"type" : {
         "name" : "duration"
       }
```

24.5 Email

An email value stores an email, such as the email that triggers a process with an email trigger.

The value includes the following properties.

Property	Type	Description
`id`	ID	The unique identifier for this email
`from`	Email address	The sender email address
`fromName`	Text	The sender's display name (optional)
`to`	List of Email address	The email addresses of the recipients
`replyTo`	Email address	The email address to send replies to (optional)
`cc`	List of Email address	Email addresses that receive a copy of the message (optional)
`subject`	Text	The subject of the email (optional)
`bodyText`	Text	The plain text message (optional)
`bodyHtml`	Text	The HTML code for an HTML email (optional)
`attachmentIds`	List of File	The files to attach to the email (optional)

24.6 Email address

An email address value stores a plain string:

```
"value" : "alice@example.org"
```

The type descriptor has no configuration properties:

```
        "type" : {
         "name" : "emailAddress"
      }
```

24.7 File

A file value stores a reference to a file stored in the SAP Signavio Process Governance database:

```
"value" : "5b5749a7d1dfff20f4adbb79"
```

This type provides the following properties.

Property	Type	Description
`id`	ID	The unique identifier for this email
`contentType`	Text	The file's **media type**
`name`	Text	The file's name
`ownerId`	User	The user who uploaded the file

JavaScript actions can use an API for **Reading file contents**.

The type descriptor has no configuration properties:

```
"type" : {
          "name" : "fileId"
        }
```

24.8 ID

An ID stores a string reference that identifies an object, but has no other meaning:

```
"value" : "5b5749a7d1dfff20f4adbb79"
```

24.9 Link

A link value stores represents an Internet address (a URL), such as a web site address, as a string value:

```
"value" : "http://www.example.org/"
```

The type descriptor has no configuration properties:

```
        "type" : {
         "name" : "link"
       }
```

24.10 List

A list value stores an (ordered) array of values of some other type, such as Email address values:

```
"value" : [
         "alice@example.org",
         "bob@example.org",
         "charlie@example.org"
       ]
```

The type descriptor's `elementType` property specified the list items' type:

```
"type" : {
         "name" : "list",
         "elementType" : {
           "name" : "emailAddress"
         }
       }
```

24.11 Money

A money value stores a JSON object with `amount` and `currency` properties. The `amount` property stores a number. The `currency` property stores an ISO 4217 currency code:

```
"value" : {
         "amount" : 12.40
         "currency" : "EUR"
       }
```

The type descriptor has no configuration properties:

```
"type" : {
         "name" : "money"
       }
```

24.12 Number

A number value stores a plain number, using a single `.` as decimal separator:

```
{
  "value" : 42
}
```

or:

```
{
  "value" : 42.42
}
```

The type descriptor has no configuration properties:

```
"type" : {
  "name" : "number"
}
```

24.13 Text

A text value stores a plain string:

```
"value" : "Example"
```

The type descriptor has an optional `multiLine` that indicates that the SAP Signavio Process Governance user interface should display multiple lines, when set to `true`:

```
"type" : {
  "name" : "text"
}
```

```
"type" : {
  "name" : "text",
  "multiLine" : true
}
```

24.14 User

A user value stores the ID of a user in your organization.

The type descriptor has no configuration properties:

```
"type" : {
  "name" : "userId"
}
```

This type provides the following properties.

Property	Type	Description
`id`	ID	The unique identifier for this user
`emailAddress`	Email address	The user’s email address
`firstName`	Text	The user’s first name
`lastName`	Text	The user’s last name

24.15 Yes/No checkbox

A yes/no checkbox value stores a Boolean value - `true` or `false`:

```
"value" : true
```

The type descriptor has no configuration properties:

```
"type" : {
  "name" : "boolean"
}
```

25 Integration tutorials

Signavio's Applied BPM blog publishes occasional integration tutorials, including the following.

Integrating a workflow with external web services	Using a script task in an approval workflow to fetch data from another system
Supervisor look-up automation with CSV data	Using a list of employees and their supervisors to automate supervisor look-up so you can automatically assign employee request workflow tasks
Starting new cases with the workflow API	Starting workflows using SAP Signavio Process Governance's public form's workflow API, using HTTP and JavaScript
Using an external web service for EU VAT number validation	Using the European Commission's VAT number validation SOAP web service
Automatically starting scheduled workflows	Automatically starting new cases of an employee review process on a fixed schedule
Web service integration: updating an external database from a workflow	Using web services to integrate workflows with external systems by writing a database connector that you can use from script tasks
DocuSign Workflow Integration for Document Signing	Using the DocuSign API to request electronic document signatures as part of an automated document approval DocuSign workflow
How to build a business days calculation workflow	Building a business days calculation workflow with JavaScript code in a SAP Signavio Process Governance script task (to calculate the number of business days between two dates)
Integrating a spreadsheet with a workflow	Automatically reading and writing spreadsheet data in a customer support workflow
How to automatically trigger a workflow with form data via email	Using a form on a public website to trigger a workflow via email