d.velop process studio: Administrator

Table of Contents

1. d.velop process studio administration manual

1.1. Basic information about the application

In this chapter, you will find product hints and general information.

1.1.1. About d.velop process studio

d.velop process studio is your central point of entry for process automation. d.velop process studio consists of multiple components and tools that help you to digitize, automate and document your daily work.

d.velop process studio gives you the capabilities to automate business processes. You can allow users to participate in processes in the form of tasks or perform automated service activities. d.velop process studio supports the BPMN standard. This gives you many options when designing processes.

In process monitoring, you have the option to check the status of active processes. If an error occurs, you have the option to apply corrections directly. This is a simple way to ensure the successful execution of your business processes.

The process administration supports you in the performance of actions across several process instances. For example, you can cancel several process instances, migrate them to a new version in case of process changes, retry failed process instances or delete a process version.

If you have modeled your own process in process management, this process is available to you within the d.velop platform.

1.2. Installation and removal

In this topic, you can find information about installing d.velop process studio.

1.2.1. System requirements

Pay attention to the general system requirements for d.velop products.

1.2.2. Installing d.velop process studio

You install the software exclusively using d.velop software manager. If an application is required for different products, the corresponding software packages are also installed automatically.

For further information on installing the software, see the d.velop software manager manual.

1.2.3. Uninstalling d.velop process studio

The software you installed using d.velop software manager can only be uninstalled with d.velop software manager. If the software to be uninstalled has dependencies with other software packages, you must resolve these conflicts accordingly.

For further information on uninstallation, see the d.velop software manager manual.

1.2.4. Installing updates for d.velop process studio

You can only update the software using d.velop software manager.

For further information on updates, see the d.velop software manager manual.

1.2.5. Rolling back an installation of d.velop process studio

You can restore an earlier version of the software that you installed with d.velop software manager. During this process, the software is only reset to a previous version.

For further information on rolling back to an earlier version, see the d.velop software manager manual.

1.2.6. Enabling standard ports for d.velop process studio

- By default, the port for d.velop process is determined dynamically. However, you can also set a port.

1.3. Configuring d.velop process studio

In this topic, you will find information on the configuration and more settings.

1.3.1. Configuring d.velop process

In this topic, you can find information about configuring d.velop process.

Configuring the database

You need a database for the application. It is assumed that you have already created a database for the application to save data in. In addition, two users are required so that the application can log in to the database. For all Unicode characters to be processed correctly, ensure that you set the sorting accordingly when creating the database schema.

The following data is required for the configuration:

- **Database type**: The database management system used, e.g. Microsoft SQL Server.
- **Hostname**: The name or the IP address of the host on which the database can be accessed.
- **Port**: The port at which the database can be accessed on the indicated host.
- **Database name**: The name of the database.
- **Database administrator**: The name of the user possessing the rights to implement schema changes (e.g. creation of tables or indexes).
- **Password of the database administrator**: The password of the administrator
- **Database user**: The name of the user to be used during the runtime for access to the database.
- **Password of the database user**: The password of the user

To configure the database, proceed as follows.

This is how it works

1. Select the feature **Configuration**.
2. Select the entry **Database** in the **Process settings** section.
3. Enter the connection data.
4. Select **Save configuration**.
5. Restart the application.

> **Note**
>
> For Microsoft SQL Server, the JDBC driver is included in the d.velop process delivery.
>
> If you use a different database, you need a JDBC driver that meets the system requirements. By using the proper JDBC driver, you ensure communication with the database management system.

Using a Microsoft SQL Server database

Please note the following for the use and configuration of a Microsoft SQL Server database:

- To establish a connection with the database, please enable the TCP/IP protocol for the SQL server.
- The tables are created in the default schema of the administrative database user.
- The administrative user must be authorized to create tables and indexes in his/her default schema.
- The runtime user must use the same default schema as the administrative user.
- In this schema, the runtime user needs the permissions **SELECT**, **INSERT**, **UPDATE** and **DELETE**.
- The two switches **ALLOW_SNAPSHOT_ISOLATION** and **READ_COMMITTED_SNAPSHOT** must be configured in the database with the value **ON**.

Using an Oracle database

Please note the following for the use and configuration of an Oracle database:

- The administrative user requires the permissions **CREATE TABLE** and **CREATE SESSION**. The tables and indexes are created in the schema and in the default tablespace of this user. The user needs the appropriate quota for the tablespace.
- To establish a connection with the database, a Net Listener with TCP protocol is required. Enter the port of the Listener under **Port**.
- Under **Database name** enter the service name or the SID. This name must be available on the Net Listener.
- The user names are case sensitive.
- The runtime user requires the permission **CREATE SESSION**.
- The runtime user will be assigned the permissions **SELECT**, **INSERT**, **UPDATE** and **DELETE** for the tables in the schema of the administrative user.

Changing the host name and port for d.ecs http gateway registration (optional)

The local computer name and the port configured in the setup are normally used for the d.ecs http gateway registration of the application.

If you want to change this data, you can specify the changes in a configuration file. You can find the configuration file under <installation directory of the application>\conf\process-app.properties. If the file does not exist, you can create a file with this name.

You can enter the properties **port** and **serverName** in the file.

```
serverName=myhost.mydomain
port=8087
```

After applying a change, you need to restart the application for the changes to take effect.

Setting up user roles (optional)

You can use the entry **User roles** in the feature **Configuration** to assign a user role to users. All authenticated users are automatically assigned the role of Process user after the setup of d.velop process.

You can assign the following roles to users:

- **Process administrator**: Has all permissions for working with d.velop process.
- **Process user**: Can start single-use processes and deploy processes.

For example, if you want to assign the role of Process administrator to a user, simply enter the user name in the appropriate field and select him/her.

Setting up HTTPS (optional)

You can encrypt the communication between d.ecs http gateway and the application using HTTPS.

This is how it works

1. Create a P12 file with the name **keystore.p12** in the configuration dossier of the installation directory of the application (\conf\keystore.p12).

2. Optionally, create the file **process-app.properties** in the same directory if it does not exist already.
3. Add the following lines to the file:
   keystorePass=Passwort_des_Keystores
   keyAlias=alias_of_certificate (optional, if only one alias exists in the keystore, this is applied automatically)
4. Restart the application to complete the setup.

Setting up a cluster

To install multiple instances of the application in a cluster, you need to ensure that all instances use the same configuration. To ensure this, a jStore with a cluster configuration must exist on each system on which the application is to be installed. This way, the installations register with the same d.ecs http gateway.

You also need to ensure that the database configuration is the same for all installations.

This is how it works

1. Terminate all instances of the application except for one. This way, you ensure that exactly one instance is configured.
2. Configure the database for the application.
3. Start the configured instance again to apply the changes.
4. Copy the files **conf\process-app-db.properties** and **lib\pa-jdbc-*.jar** in the installation directory of this instance.
5. Paste these files into the installation directory of all other instances of the application.
6. Start all instances.

You can also follow this procedure if you want to apply a change of the connection data for the database.

> ### Warning
> **Procedure for a software update:**
>
> Please note that all instances of the application must be closed before importing a new software version. Afterwards, only restart the instances that have already been updated.

Configuring protocol exports

You would like to configure exports for protocols into a d.3 repository in order to save protocols for long periods.

What you need to know

- You need a configured d.3 repository in which the protocol exports are saved.
- You must be logged on with a user that can administer mappings for the d.3 repository.
- You must create a document type in your repository that has four alphanumeric (or text, depending on your system) properties with a maximum length of 250 characters. Assign these new properties to the process properties next. Enter descriptive names for the properties that match the process properties, e.g. **Process**, **Process Instance**, **Business Key**, and **Attachment**.
- If one of the values is longer than 250 characters (or 255 bytes), this value is truncated as appropriate during the export. Three periods indicate the truncated point.
- If metadata is truncated or if there is an error during the export process, you will see an appropriate note for the process instance in both process monitoring and in the protocol view.
- You need to create an API key for a user who has administration rights. In addition, the user needs read and write access for this type of document.

This is how it works

1. Open the **Assignments** feature.
2. Add a new mapping.
3. Select the source **Processes**.
4. Add a category and link the source **Process log** to the required document type.
5. Link all the properties of the source with document properties of the target. Make sure that all four properties have been mapped.
6. Save the mapping.
7. Open the feature **Configuration** and select **DMS export configuration** in the section **Process settings**.
8. Select the target repository and enter the API key.
9. Save the changes.

After you have configured the protocol export, you can activate the option for exporting the protocol during the definition of a process.

Changing the base client address (optional)

d.velop process generates URLs, e.g. for notifications, which can be opened by a client (e.g. web browser). The URLs start with the base address of the system.

You can specify an alternate base address for the URL in a configuration file if needed. You can find the file in <installation directory of the application>\conf\process-app.properties\.

Enter the property in the **clientBaseUri** file.

```
clientBaseUri=https://myhost.mydomain
```

Then restart the application for the changes to take effect.

1.3.2. Configuring d.velop process modeler

In this topic, you can find information about configuring d.velop process modeler.

Setting up user roles for d.velop process modeler

You do not have to configure separate user roles for d.velop process modeler. The d.velop process configuration is used here. To model processes, you must be entered as an administrator in d.velop process.

See also: Setting up user roles (optional)

Setting up HTTPS for d.velop process modeler (optional)

You can encrypt the communication between d.ecs http gateway and the application using HTTPS.

This is how it works

1. Create a **keystore.p12** file in the configuration folder of the application installation directory (**\config**).
2. Copy the file **application.properties.template** in the directory **\config** and rename the copied file as **application.properties**.
3. Enter the following configuration parameters in the file and enable them:
   a. **quarkus.http.ssl.certificate.key-store-file=path_to_keystore_file**
   b. **quarkus.http.ssl.certificate.key-store-password=password_of_keystore**
4. You also have the option of adding the following configuration parameter in the file and enabling it: **quarkus.http.ssl.certificate.key-store-key-alias=alias_of_certificate** (optional – if only one alias exists in the keystore, this is applied automatically)
5. Restart the application to complete the setup.

1.3.3. Configuring d.velop forms

In this topic, you can find information about configuring d.velop forms.

Setting up user roles for d.velop forms

In the **Configuration** feature, you can assign a user role to users or groups in the area **Forms > User roles**. By default, all users can open, fill out and submit forms even if they are not assigned to a role. User roles are used to assign specific users with additional authorizations for working with forms.

The following roles are available:

- **Administration**:
  The user or group can create, edit and delete forms.
- **Processing:**
  The user or group can create forms.
  The user or group can edit and delete forms, provided the user or group has explicit permissions for these forms.

To assign a role to a user or group, enter the name of the user or group in the relevant field and select the user or group.

1.3.4. Configuring d.velop scripting

In this topic, you can find information about configuring d.velop scripting.

Setting up user roles for d.velop scripting

In the **Configuration** feature, you can assign a user role to users or groups in the area **Scripting > User roles**. By default, all users can run scripts even if they are not assigned to a role. User roles are used to assign specific users with additional authorizations for working with scripts.

The following roles are available:

- **Administration**:
  The user or group can create, edit and delete scripts.
- **Processing:**
  The user or group can create scripts.
  The user or group can edit and delete scripts, provided the user or group has explicit permissions for these scripts.

To assign a role to a user or group, enter the name of the user or group in the relevant field and select the user or group.

Using d.velop scripting in combination with a proxy server

You can use d.velop scripting in combination with a proxy server.

This is how it works

1. Create a system environment variable with the name **DVS_HTTP_PROXY**.
2. Enter the proxy server in the system environment variable. Observe the following:
   - The entry must start with **http://** or **https://**.
   - If the proxy server only offers a self-signed certificate, you have to create an additional system environment variable. Give the system environment variable the name **DVS_HTTP_PROXY_UNSAFE_ALLOW_SELF_SIGNED_CERTIFICATES** and the value **true**.
3. Create the Node.js configuration for the executing d.velop scripting process user (service user of d.3 process manager). To do this, enter the corresponding command in the context of the process user:
   - When using an HTTP proxy server: **npm config set proxy http://proxy_host:port npm config set https-proxy http://proxy.company.com:8080**

- When using an HTTPS proxy server: **npm config set https-proxy https://proxy.company.com:8080 npm config set strict-ssl false**

Operating d.velop scripting in a cluster

If you are running d.velop scripting on several servers in a cluster, the file **top.secret** in the working directory must be identical on all servers. After the first time d.velop scripting is started, copy the file to all nodes so that you can decrypt the encrypted variables of the scripts on all nodes.

Overwriting the FQDN for registration on d.ecs http gateway (optional)

d.velop scripting automatically determines the FQDN (fully qualified domain name) for registration on d.ecs http gateway. In some cases, the app tries to register using an FQDN that cannot be reached by d.ecs http gateway.

You can overwrite the FQDN for the registration using a Windows environment variable.

Create a system environment variable with the name **DVS_FQDN** and enter the FQDN to be used there in the format **test.mydomain.local**.

> **Note**
> Once you have created or changed the environment variable, d.3 process manager must be started. The d.velop scripting change is then applied.

1.3.5. Configuring of eventbridge d.velop actions

In this topic, you can find information about configuring d.velop actions eventbridge.

Setting up a cluster

If you would like to install multiple instances of the application in a cluster, you must ensure that all instances use the same configuration. To do this, on each system with an installation, d.ecs jstore must be available with a cluster configuration. This way, all the installations register on the same d.ecs http gateway instance.

The encrypted data, and optionally the HTTPS configuration, must also be identical on all instances.

This is how it works

1. If available, copy the files **.env**, **eventbridge.crt**, **eventbridge.key** and **top.secret** from the installation directory of the primary instance.
2. Paste the copied files into the installation directory of all the other instances.
3. Restart the instances.

How to proceed in the event of a software update

Exit all instances of the application before importing a new software version. You then have to restart only those instances that were already updated.

Setting up HTTPS (optional)

You can encrypt the communication between d.ecs http gateway and the application using HTTPS.

This is how it works

1. Create a public certificate file with the name **eventbridge.crt** in the installation directory of the application.
2. Create a file with the private key of the certificate with the name **eventbridge.key** in the installation directory of the application.
3. Create the file **.env** in the installation directory (if it is not already available).

4. Add the following lines to the file:

```
https.enabled=true
https.certificate.public.path=./eventbridge.crt
https.certificate.private.path=./eventbridge.key
```

5. Restart the application to complete the setup.

Observe the instructions in the chapter Setting up a cluster.

1.4. Modeling of processes

In this topic, you will find out how to model your own processes. You will get to know the supported BPMN items and find out how to add these to your processes.

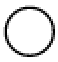
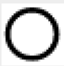
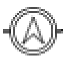
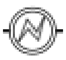
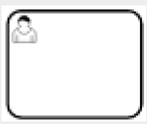
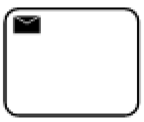
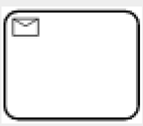
1.4.1. Useful information on process modeling

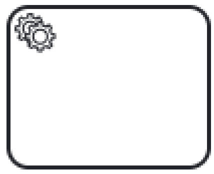
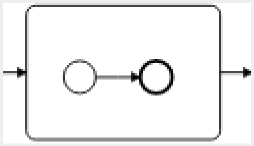
For the creation of process models, the application supports the BPMN standard. This is an XML-based format. The modeling of processes in the d.velop context is aligned with the standard of the Object Management Group (OMG). The value **http://www.omg.org/spec/BPMN/20100524/MODEL** is the default namespace.

For the modeling of process models, we recommend using d.velop process modeler.

1.4.2. Using BPMN items

When creating your own process models, you can use the following BPMN items of BPMN standard 2.0:

	BPMN item	Restrictions
	**Start event**	Permitted event types: None, Timer Only in event-based subprocesses: Escalation, error
	**End event**	Permitted event types: None, termination, escalation, error
  	**Intermediate boundary event**	Permitted event types: Timer, escalation, error
	**Intermediate catch event** (attached intermediate event with a waiting function)	Permitted event types: timer
	**Intermediate throw event** (attached intermediate event with a triggering function)	Permitted event types: Escalation
	**User Task** (User activity)	
	**Send Task** (Send activity)	Only for connecting services.
	**Receive Task** (Receive activity)	Only for connecting services.

	BPMN item	Restrictions
	**Service Task** (service activity)	Only defined actions allowed (**writeLocalVariables**, **writeGlobalVariable** or **writerMultiInstanceVariables**).
	**Gateway**	Permitted gateway types: Exclusive, parallel, inclusive.
	**Sequence flow**	
	**Subprocess**	Permitted subprocess types: Normal, event-based (see start event)
	**Pool/participant/lane**	A maximum of one pool/participant is permitted per process diagram.

General restrictions

The following restrictions apply to all items:

- Expressions of input and output parameters are subject to the restrictions on expressions.
- Only the following camunda extensions are permitted:
  - **camunda:inputOutput**
  - **camunda:properties**
  - **camunda:property**
  - **camunda:failedJobRetryTimeCycle**
  - **camunda:asyncBefore**
  - **camunda:asyncAfter**
  - **camunda:formKey**

1.4.3. Creating a sample process in a BPMN file

When getting started with the modeling of your own processes, it is useful to create a simple process as an example first. Prerequisite for the modeling of your own processes are a set user roles. In addition, you need a text or BPMN editor with XML view and a tool to call a REST API.

A simple process will look like this:

To model your own process, first create the following basic structure in your BPMN file:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process>
    <startEvent />
    <sequenceFlow />
    <userTask />
    <sequenceFlow />
    <endEvent />
  </process>
</definitions>
```

To this basic structure, add the individual process items.

Adding the process item

The BPMN item **process** encapsulates the entire process. Add the properties **id**, **name** and **isExcecutable** to the BPMN item.

```
<process id="hello_world_process" name="Hello World Process"
isExecutable="true">
```

Explanation of the properties

- **id**: This property is used as a unique identifier for the process.
- **name**: This property is used as a display name in the user interfaces.
- **isExecutable**: This property must contain the value **true** to enable the process to be started.

Adding a start and end event

The BPMN items **startEvent** and **endEvent** are used as markings which indicate where the process begins and where it ends. Add the property **id** to the respective BPMN items.

```
<startEvent id="start" />
<endEvent id="end" />
```

Adding a user activity

The BPMN item **userTask** stands for a user activity in the process. When this activity is entered, one or more users are assigned a task. Add the properties **id**, **name**, and **camunda:candidateUsers** to the BPMN item.

```
<userTask id="task_hello_world" name="Hello World" camunda:candidateUsers="$
{variables.get('dv_initiator')}" />
```

Explanation of the properties

- **id**: This property serves as an identifier of the activity in the BPMN model and is used for the connection by means of sequence flows.
- **name**: This property is added as a subject to the task.
- **camunda:candidateUsers**: This property refers to the person who is to process the task. In the BPMN file, the expression **${variables.get("dv_initiator")}** uses a variable. The variable **dv_initiator** always refers to the person who has started the process. The task is therefore always assigned to the person starting the process.

Adding sequence flows

Connect the individual process items to create a sequence. To connect the process items, use the BPMN item **sequenceFlow**. Add the properties **id**, **sourceRef** and **targetRef** to the BPMN item.

The sample configuration in the BPMN file shows how you connect the start event to the user activity and link the user activity with the end event:

```
<sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
<sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
```

Explanation of the properties

- **id**: This property is a unique identifier for the sequence flow within the BPMN model.
- **sourceRef**: This property refers to the starting point of the sequence flow.
- **targetRef**: This property refers to the target of the sequence flow. In this property you enter the identifiers of the items to be connected.

After you have added the individual process items, the BPMN file will look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process id="hello_world_process" name="Hello World Process"
isExecutable="true">
    <startEvent id="start" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world"
/>
        <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
    <endEvent id="end" />
  </process>
</definitions>
```

Adding graphic information

If you want to view your created process in a diagram, you can add graphic information to the BPMN file. This graphic information defines the size of the diagram and the position of the individual items. You add the graphic information by means of the BPMN item **bmpndi:BPMNDiagram**.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" xmlns:bpmndi="http://
www.omg.org/spec/BPMN/20100524/DI" xmlns:di="http://www.omg.org/spec/DD/
20100524/DI" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
targetNamespace="" exporter="d.velop process modeler">
  <process id="hello_world_process" name="Hello World Process"
isExecutable="true">
```

```
    <startEvent id="start" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
    <endEvent id="end" />
  </process>

  <!-- Diagram information -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane" bpmnElement="hello_world_process">
      <bpmndi:BPMNShape id="start_di" bpmnElement="start">
        <dc:Bounds x="173" y="102" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s1_di" bpmnElement="s1">
        <di:waypoint x="209" y="120" />
        <di:waypoint x="259" y="120" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="task_hello_world_di"
bpmnElement="task_hello_world">
        <dc:Bounds x="259" y="80" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s2_di" bpmnElement="s2">
        <di:waypoint x="359" y="120" />
        <di:waypoint x="409" y="120" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="end_di" bpmnElement="end">
        <dc:Bounds x="409" y="102" width="36" height="36" />
      </bpmndi:BPMNShape>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

1.4.4. Using expressions

You can use variable expressions for the modeling of processes. The expressions use the syntax of the Java Expression Language.

Expressions may only use the following items:

Item	Restriction
Read access to the variables using **variables**. • **variables.get** • **variables.getDisplayValue** • **variables.getObjectValue**	
Generating multi-values • **collections.from**	

Read access to values of the current process instance
 	
• **process.instance.id**
  • ID of the current process instance
• **process.instance.businessKey**
  • Business key of the current process instance	

Operators and characters	Permitted operators and characters: +, -, *, /, %, =, !, <, >, &, |, ?, (,), [,].
Keyword	Permitted keywords: div, mod, eq, ne, lt, gt, le, ge, and, or, not, empty.
Numeric values	

Item	Restriction
Strings	Strings must be put between " " or ' '.
Shortening strings • **strings.shorten**	

Examples

```
${"Hello World"}                                                       //
Hello World
${3}                                                                   // 3
${3 > 4}                                                               //
false
${variables.get("myVariable")}                                         //
Value of myVariable
${variables.get("myVariable") == "Hello"}                              //
true, if myVariable equals Hello, false otherwise
${collections.from("a", "b", "c")}                                     //
creates a collection with "a", "b" and "c"
${strings.shorten("${variables.get('myLongTextVariable')}", 10)}      //
creates a substring of myVariable with the length of 10
```

Hybrid forms are also possible, making it possible to directly combine texts and variable content.

Examples

```
This is an expression with ${variables.get("myVariable")}
3 is ${(3 > 4) ? "greater" : "lesser"} than 4
```

1.4.5. Working with process variables

In this topic, you will find out how to dynamically create your processes using variables.

Useful information on process variables

Process variables are data containing information on a process instance. The following process variables are included in every process instance:

- **dv_initiator**: A process variable of data type **Identity**. The process variable refers to the identity of the user who has started the process instance.
- **dv_start_time**: A process variable of data type **String**. The process variable includes the UTC string of the time at which the process instance was started from the point of view of the application. The time at which the application was instructed to start the process instance applies. As the actual (technical) start of the instance is performed asynchronously, the time included in the variable may vary.

Using other process variables

You can use these variables for special purposes:

- **dv_sender**: A process variable of data type **Identity**. The process variable refers to the identity of the user who is used as the sender of user activities.
- **dv_attachment**: A process variable of data type **DmsObject** or other. The process variable includes a link which is added as an attachment to all user activities.
- **dv_decision:** A process variable of the type **String**. For tasks, this text is highlighted as a **decision** in the process protocol.
- **dv_comment:** A process variable of the type **String**. For tasks, this text is highlighted as a **comment** in the process protocol.

> **Warning**
> The process variable **dv_attachment** is predefined with the data type **DmsObject** . If you want to use a different URI than that for an object of the DMS app, you need to change the value of the data type to **String**.

Defining process variables

In the process definition, you can define variables that are used within a process. Defined variables can be automatically validated and, if necessary, converted at the appropriate interfaces of the application.

You can define the following variables within a process:

- **General process variable**: A variable which is valid in the entire process. You can specify whether it is a start variable.
- **Local process variable**: A variable with a scope that is limited to a single process activity.
- **Service variables**: Input or output variables of a process service.

A variable definition contains the following information:

Property	Possible values	Example
Name	[A-Za-z][A-Za-z0-9_]*	myVariable (**name** property)
Start variable		myStartVariable* (**name** property)
Data type	• String • Number • Boolean • Identity • DmsObject • URL • Object • File	String (**value** property)
Single or multi-value		Single value: String (**value** property) Multi-value: [String] (**value** property)
Mandatory variable		Single value: String! (**value** property) Multi-value: [String] (**value** property)

> **Note**
> Specifics of the **Object** data type:
>
> - You can only define a variable of the type **Object** in a process.
> - A variable of the type **Object** must not contain a multi-value.
> - Variables of the type **Object** are not logged.

> **Note**
> Specifics of the **File** data type:
>
> - A file may not exceed 100 MB in size.
> - The total size of all files in a process instance may not exceed 500 MB.
> - Files will be deleted at the end of the process.
> - Variables of type **File** are not logged.

Warning

Please consider the following important notes when using process variables:

- **Name prefix "dv_"**: You must not use the name prefix **dv_** for variables. These variables are reserved for internal variables. You may use the name prefix **dv_** only when overwriting the variable definition **dv_attachment** .
- **Changing variable definitions when switching the version**: If you change the data type of a variable or switch between a single value and a multi-value between different versions of a process, errors may occur in the execution of the process after the migration. These errors occur if existing process instances already contain values for the corresponding variables in the previous definition. After migrating to the new version, these values lose their validation. Always make sure that no process instance requiring migration is affected when changing the variable definition.

Note

You cannot delete a variable which is defined as a mandatory variable via the interface for changes to variables. If the mandatory variable is also a start variable, a user always needs to transfer a valid value when a process instance is started. If he does not transfer a valid value, the request will be rejected with the error code 400.

For data type **Number** the following restrictions apply:

Minimum	Maximum	Decimal places	Accuracy
-1e16	1e16	5	15 places

The non-primitive data types use the following notation:

Data type	Notation
Identity	**For users:** identity:///identityprovider/scim/users/<someUserId> **For groups**: identity:///identityprovider/scim/groups/<someGroupId>
DmsObject	dmsObject:///dms/r/<repositoryId>/o2/<dmsObjectId>
URL	https://www.d-velop.de **(max. 2,000 characters)**
Object	{ "myKey" : "myValue" } **(JSON-String, max. 50 KB)**
File	Variables of type **File** cannot be set in BPMN. This is only possible via the API.

Variables are defined as a BPMN extension in the form of Camunda properties. In the BPMN model, you define general process and service variables directly in the process node. You define local process variables in the node of the activity in which they are to apply.

General process and service variables

```
 <process id="myProcess" name="Process with variable declaration"
isExecutable="true">
    <extensionElements>
        <camunda:properties>
            <camunda:property name="variable:myVariable" value="[String]!"
/> <!-- Name: "myVariable", Type: "String", Multiple: true, Mandatory:
true, Startvariable: false -->
            <camunda:property name="variable:myStartVariable*"
value="String" /> <!-- Name: "myStartVariable", Type: "String", Multiple:
false, Mandatory: false, Startvariable: true-->
            <camunda:property name="variable:someUserTaskOutput"
value="String" /> <!-- Name: "someUserTaskOutput", Type: "String",
```

```
Multiple: false, Mandatory: false, Startvariable: false -->
            <camunda:property name="service:/myservice:in:input"
value="String!" /> <!-- Name: "input", Type: "String", Multiple: false,
Mandatory: true -->
            <camunda:property name="service:/myservice:out:output"
value="Number!" /> <!-- Name: "output", Type: "Number", Multiple: false,
Mandatory: true -->
        </camunda:properties>
    </extensionElements>
</userTask>
```

Local process variable

```
<userTask id="userTask" name="User Task">
    <extensionElements>
        <camunda:inputOutput>
            <camunda:inputParameter name="someInput">user task input which
is only available within this user task scope</camunda:inputParameter>
            <camunda:outputParameter name="someUserTaskOutput">user task
output which will be written to process scope</camunda:outputParameter>
        </camunda:inputOutput>
        <camunda:properties>
            <camunda:property name="variable:someInput" value="String" />
        </camunda:properties>
    </extensionElements>
</userTask>
```

Using process variables

When modeling your own processes, you can use process variables to create dynamic processes. When getting started with process variables, it is useful to create a simple process with one user activity first. This process will e.g. look like this:

The BPMN of this sample process is structures as follows. To simplify the example, the graphic information on the BPMN diagram is not included in this BPMN sample file.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process id="GettingStarted" name="Beispiel: Einstieg"
isExecutable="true">
    <startEvent id="start"/>
    <userTask id='task_hello_world' name='Hello World'
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
        <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end"
/>
```

```
    <endEvent id="end"/>
  </process>
</definitions>
```

Now add a new variable definition to this process definition.

Adding a variable definition

To be able to use a process variable, you need to add it to the process definition. In the process definition, you add the BPMN items **extensionElements** and **camunda:properties** to the BPMN item **process**. To define the process variable, then add the item **camunda:property** to the BPMN item **camunda:properties**.

This example shows how to add the variable **message** of data type **String**:

```
<process id="GettingStarted" name="Beispiel: Einstieg" isExecutable="true">
        <extensionElements>
      <camunda:properties>
        <camunda:property name="variable:message" value="String" />
      </camunda:properties>
    </extensionElements>
        ...
</process>
```

Explanation of the properties

- **name**: This property defines the name of the process variable. The value is always preceded by the prefix **variable:**.
- **value**: This property defines the data type of the process variable.

Using the process variable

You can now use the variable **message** for the name of the user activity, so that this name is displayed to the user in the task list. To do so, replace the existing test in the property **name** of the user activity with the method for access to the variable **message**:

```
<process id="GettingStarted" name="Beispiel: Einstieg" isExecutable="true">
        ...
        <userTask id="task_hello_world" name="${variables.get('message')}"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
        ...
```

You can now define the value of the variable e.g. for the start of the process.

Determining process variables

Using the object **variables** you can access the process variables. In each case, the call is executed using the notation **variables.<methodName>(<Parameter>)**.

You cannot apply these methods to process variables of type **File**.

get(String variableName)

The method determines the value for the variable with the entered variable name. For variables with a multi-value, the values are returned in a list (collection).

getDisplayValue(String variableName)

The method determines the value for the variable with the entered variable name. An according conversion will take place here depending on the data type.

For variables with a multi-value, the converted values are returned in a string, separated by commas. If the data type of the variables is **Number**, the converted values are returned in a string, separated by semicolons.

Data type	Actual value	Returned value
String	"some value"	"some value"
Number	1.23	"1.23"
Boolean	true	"true"
Identity	identity:///identityprovider/scim/users/userIdOfJohn-Smith	"John Smith" (**displayName-** property of the user, **name-** property for groups in the d.ecs identity provider)
DmsObject	dmsObject:///dms/r/someRepo/o2/T000000001	"/dms/r/someRepo/o2/T000000001"
Object	{"myKey":"myValue"}	"{\"myKey\":\"myValue\"}" (JSON-String)

getObjectValue(String variableName, String objectPath)

This method can only be used for variables of the type **Object**. It allows you to access individual parts of the object.

Example:

Supposed there is a variable of the **Object** type with the name "myObjectVariable" and the following value:

```
{
  "my" : {
    "object" : {
      "path" : 123
    }
  }
}
```

The method **getObjectValue** can be used to determine the following parts of the objects, for example:

Method call	Returned value
getObjectValue("myObjectVariable", "my.object.path")	123
getObjectValue("myObjectVariable", "my.object")	{"path":123}

Generating multi-values

You can use the object **collections** to generate a list (collection) from individual values, e.g. to assign a multi-value to a variable. Generating a list from individual values is useful if you want to register several recipients as a constant in a user activity.

Example

```
${collections.from( "identity:///identityprovider/scim/users/user1",
"identity:///identityprovider/scim/users/user")}
```

from(Object...objects)

The method generates a list (collection) with the transferred individual values.

Useful information on the scope of process variables

Process variables always have a specific scope within a process instance. Depending on the scope, read and write access will be possible.

In every process instance, there is a universal scope which comprises the entire process. All other scopes, such as user or service activities, have read access to the variables in the universal scope.

Additional scopes are generated at the following points:

- Gateways with parallel paths
- Multi-instance constructs
- Subprocesses
- Input/output mapping or timer events during user activities
- Service activities

Read access

Read access to process variables (e.g. the use in an expression) is possible independently of the scope. If no value is found in the current scope, the section above it is searched until a value has been found.

Write access

If you want to change values (e.g. in a user activity with output mapping), the value is always applied to the current scope. Other scopes, e.g. along other branches in the process, are not impacted by this until the end of this scope. The changes are transferred to all other scopes only at the end of the current scope.

End of scopes

At the end of a scope (e.g. when joining parallel branches or at the end of a subprocess), the variables of this scope are discarded. You must configure an output mapping before you can apply variables to a scope located higher in the hierarchy. In the process, the call hierarchy is run through until a scope that already contains this variable is found. This run starts from the current item and works its way towards the top. The new value is then applied to this scope. If the variable is completely new, it is then written to the scope of the entire process.

> **Warning**
> Please note that a timer event generates its own scope during user activities. If you do not define a dedicated input/output mapping for the user activity, all variables of this scope will be applied to the higher scope when the activity is completed.

Impact on process monitoring

The view of a token always shows the variables of the current scope. Changes to the variables are only applied in this scope.

Most tokens have their own scope. The only exception are user activities for which no input/output mapping was configured and which are not executed in parallel to other activities. Theses tokens have the same scope as the token directly above them.

In the view of the token without its own scope, you can find a link to a token from the section above it. When you select this link, you can view the variables of the linked scope.

1.4.6. Using timer events

For process modeling, you can use the BPMN item **intermediate boundary event** (attached intermediate event) of the Timer type. You can attach these intermediate events to user activities or services. When getting started with attached intermediate events, it is useful to create a simple process with one user activity first.

The BPMN model of this sample process is structured as follows:

```xml
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
```

```
  <process id="timer_event_process" name="Timer Event Process"
isExecutable="true">
    <startEvent id="start" />
        <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
  </process>
</definitions>
```

Adding an intermediate event

Add the BPMN item **boundaryEvent** to the process.

```
...
<process id="timer_event_process" name="Timer Event Process"
isExecutable="true">
  ...
  <userTask ... />
  <boundaryEvent id="timer" attachedToRef="task_hello_world">
    <timerEventDefinition>
      <timeDuration>PT1M</timeDuration>
    </timerEventDefinition>
  </boundaryEvent>
  ...
</process>
```

Explanation of the properties

- **id:**The property is used as a unique identifier for the send activity.
- **attachedToRef:** The property defines which BPMN item this intermediate event should be attached to.

Explanation of the event definition:

- **timerEventDefinition:** This BPMN item defines that the intermediate event is a time event
- **timeDuration:** This BPMN item defines that it is a relative time span. In this case, a span of one minute has been defined. If you want to define an absolute point in time, you can use the BPMN item **timeDate**. In both cases, you need to enter the values in the ISO 8601 format. You can also determine the values from variables.

Adding an event path

If an intermediate event occurs in the process, the process follows a separate process path. You can e.g. configure a process in such a way that the process leads from an intermediate event directly to the end event. You insert this configuration into the process as follows:

```
...
<process id="timer_event_process" name="Timer Event Process"
isExecutable="true">
  ...
  <sequenceFlow id="s3" sourceRef="timer" targetRef="end" />
</process>
```

In order for the BPMN diagram to be displayed correctly in the modeling tool as well as the user interface, diagram information has been added. The final BPMN definition looks as follows:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" xmlns:bpmndi="http://
```

```
www.omg.org/spec/BPMN/20100524/DI" xmlns:dc="http://www.omg.org/spec/DD/
20100524/DC" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
targetNamespace="" exporter="d.velop process modeler">
  <process id="timer_event_process" name="Timer Event Process"
isExecutable="true">
    <startEvent id="start" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <boundaryEvent id="timer" attachedToRef="task_hello_world">
      <timerEventDefinition>
        <timeDuration>PT1M</timeDuration>
      </timerEventDefinition>
    </boundaryEvent>
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
    <sequenceFlow id="s3" sourceRef="timer" targetRef="end" />
  </process>

  <!-- Diagram information -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="timer_event_process">
      <bpmndi:BPMNShape id="start_di" bpmnElement="start">
        <dc:Bounds x="179" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="task_hello_world_di"
bpmnElement="task_hello_world">
        <dc:Bounds x="290" y="77" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="timer_di" bpmnElement="timer">
        <dc:Bounds x="372" y="139" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="end_di" bpmnElement="end">
        <dc:Bounds x="462" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s1_di" bpmnElement="s1">
        <di:waypoint x="215" y="117" />
        <di:waypoint x="290" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s2_di" bpmnElement="s2">
        <di:waypoint x="390" y="117" />
        <di:waypoint x="462" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s3_di" bpmnElement="s3">
        <di:waypoint x="408" y="157" />
        <di:waypoint x="480" y="157" />
        <di:waypoint x="480" y="135" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

1.4.7. Using gateways

When modeling your own processes, you can use the BPMN item **Gateway** of the exclusive type (branching point of exclusive type) to create dynamic processes with a decision logic.

When getting started with exclusive gateways, it is useful to create a simple process with one user activity first. This process will e.g. look like this:

The BPMN model of this sample process is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:camunda="http://
camunda.org/schema/1.0/bpmn" targetNamespace="" exporter="d.velop process
modeler">
  <process id="exclusive_gateway_process" name="Exclusive Gateway Process"
isExecutable="true">
    <startEvent id="start" />
        <userTask id="task_hello_world" name="Hello World"
camunda:camundaUsers="${variables.get('dv_initiator')}" />
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
  </process>
</definitions>
```

You now expand this sample process with a gateway. Based on the value of a process variable, this process will either lead to the user activity or will be ended directly.

Adding a gateway

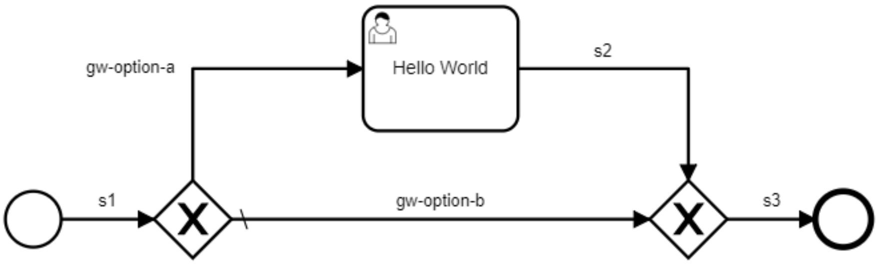
First add the BPMN item **exclusiveGateway** to the process. The item **exclusiveGateway** represents a branching point with only one possible start path. Then replace the existing sequence flows with the following sequence flows:

```
...
<process id="exclusive_gateway_process" name="Exclusive Gateway Process"
isExecutable="true">
  <startEvent id="start" />
  <exclusiveGateway id="gateway" default="gateway-option-b" />
  <userTask id="task_hello_world" name="Hello World" camunda:camundaUsers="$
{variables.get('dv_initiator')}" />
  <exclusiveGateway id="join" />
  <endEvent id="end" />
  <sequenceFlow id="s1" sourceRef="start" targetRef="gateway" />
  <sequenceFlow id="gateway-option-a" sourceRef="gateway"
targetRef="task_hello_world" />
  <sequenceFlow id="gateway-option-b" sourceRef="gateway" targetRef="join"
/>
  <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="join" />
  <sequenceFlow id="s3" sourceRef="join" targetRef="end" />
</process>
```

Explanation of the properties:

- **id:** The property is used as a unique identifier for the gateway.
- **default:** The property defines the ID of the sequence flow to be selected as the default start path of the gateway.

The process will then e.g. look like this:



Adding a condition

All start paths of an exclusive gateway must contain a condition. The application uses this condition to determine which path the process is to take. For the sample process, add the condition to the sequence flow **gw-option-a** that this path is selected depending on the variable **amount**. If this variable corresponds to a value greater than 1000, the sequence flow **gw-option-a** is to be selected.

```
...
<sequenceFlow id="gw-option-a" sourceRef="gateway"
targetRef="task_hello_world" >
  <conditionExpression
xsi:type="tFormalExpression">#{variables.get("amount") > 1000}</
conditionExpression>
<sequenceFlow>
...
```

To be able to use a process variable, you need to add it to the process definition. In the process definition, you add the BPMN items **extensionElements** and **camunda:properties** to the BPMN item **process**. To define the process variable, then add the item **camunda:property** to the BPMN item **camunda:properties**. In this example, you add the variable **amount** of the **Number** type. This is a mandatory variable which must be provided at the start.

```
...
<process id="exclusive_gateway_process" name="Exclusive Gateway Process"
isExecutable="true">
  <extensionElements>
    <camunda:properties>
      <camunda:property name="variable:amount*" value="Number!" />
    </camunda:properties>
  </extensionElements>
  ...
</process>
```

In order for the BPMN diagram to be displayed correctly in the modeling tool as well as the user interface, diagram information has been added. The final BPMN definition looks as follows:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:camunda="http://
camunda.org/schema/1.0/bpmn" xmlns:bpmndi="http://www.omg.org/spec/BPMN/
20100524/DI" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:di="http://www.omg.org/spec/DD/20100524/DI" targetNamespace=""
```

```
exporter="d.velop process modeler">
  <process id="exclusive_gateway_process" name="Exclusive Gateway Process"
isExecutable="true">
    <extensionElements>
      <camunda:properties>
        <camunda:property name="variable:amount*" value="Number!" />
      </camunda:properties>
    </extensionElements>
    <startEvent id="start" />
    <exclusiveGateway id="gateway" default="gateway-option-b" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <exclusiveGateway id="join" />
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="gateway" />
    <sequenceFlow id="gateway-option-a" sourceRef="gateway"
targetRef="task_hello_world">
      <conditionExpression
xsi:type="tFormalExpression">#{variables.get("amount") > 1000}</
conditionExpression>
    </sequenceFlow>
    <sequenceFlow id="gateway-option-b" sourceRef="gateway"
targetRef="join" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="join" />
    <sequenceFlow id="s3" sourceRef="join" targetRef="end" />
  </process>

  <!-- Diagram information -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1"
bpmnElement="exclusive_gateway_process">
      <bpmndi:BPMNShape id="start_di" bpmnElement="start">
        <dc:Bounds x="179" y="199" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="gateway_di" bpmnElement="gateway"
isMarkerVisible="true">
        <dc:Bounds x="275" y="192" width="50" height="50" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="task_hello_world_di"
bpmnElement="task_hello_world">
        <dc:Bounds x="410" y="80" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="join_di" bpmnElement="join"
isMarkerVisible="true">
        <dc:Bounds x="595" y="192" width="50" height="50" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="end_di" bpmnElement="end">
        <dc:Bounds x="702" y="199" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s1_di" bpmnElement="s1">
        <di:waypoint x="215" y="217" />
        <di:waypoint x="275" y="217" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="gateway-option-a_di" bpmnElement="gateway-option-
a">
```

```
        <di:waypoint x="300" y="192" />
        <di:waypoint x="300" y="120" />
        <di:waypoint x="410" y="120" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="gateway-option-b_di" bpmnElement="gateway-option-
b">
        <di:waypoint x="325" y="217" />
        <di:waypoint x="595" y="217" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s2_di" bpmnElement="s2">
        <di:waypoint x="510" y="120" />
        <di:waypoint x="620" y="120" />
        <di:waypoint x="620" y="192" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s3_di" bpmnElement="s3">
        <di:waypoint x="645" y="217" />
        <di:waypoint x="702" y="217" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

You can now define the value of the variable **amount** e.g. for the start of the process and impact the process sequence this way.

1.4.8. Inserting user activities

In this topic, you will find basic information on the use and customization of the BPMN item **User Task** (user activity).

Useful information on user activities

The BPMN standard includes different constructs for assigning recipients within a **User Task** (user activity).

We recommend using the property **camunda:candidateUsers**.

Potential Owner

```
<userTask id='theTask' name='important task' camunda:candidateUsers="$
{variables.get('myPerformer')}" />
```

Using constants for recipients

If you want to state the recipient of an activity as a constant, use the following syntax:

```
${collections.from("identity:///identityprovider/scim/users/someUserId")}
// or
${collections.from("identity:///identityprovider/scim/users/someUserId",
"identity:///identityprovider/scim/users/someOtherUserId")}
```

Using variables for recipients

If the recipients of an activity are to determined from variables, use the following syntax to reference the variables:

```
${variables.get("variableName")}
```

Linking additional information to user activities

To add additional information to user activities for processing, you can use the process variable **dv_attachment** reserved for this purpose.

You can save one URL in this variable. The URL is added during the creation of an activity if you have entered it beforehand.

> **Warning**
> The process variable **dv_attachment** is predefined with the data type **DmsObject** . If you want to use a different URI than that for an object of the DMS app, you need to change the value of the data type to **String**.

Adding metadata to user activities

You can define metadata within the BPMN item **userTask** in order to add this metadata to user activities. Add the item **extensionElements** and, under it, the item **camunda:properties** if the process definition does not contain these items yet. You can now define metadata as follows below the item **camunda:properties**:

```
<userTask>
...
        <extensionElements>
                <camunda:properties>
                        <camunda:property
name="metadata:invoiceNumber:value" value="$
{variables.get("invoiceNumber")}" />
                <camunda:property name="metadata:invoiceNumber:caption"
value="Invoice Number" />
                <camunda:property name="metadata:invoiceNumber:caption:de"
value="Rechnungsnummer" />
                <camunda:property name="metadata:invoiceNumber:type"
value="String" />
        </camunda:properties>
        </extensionElements>
...
</userTask>
```

The property **name** within the item **camunda:property** can accept the following values:

Value	Meaning	
metadata:<key>:value	Defines the value of a metadata item.	The key of the metadata item must correspond to the following expression: [A-Za-z0-9]{1,255}
metadata:<key>:caption	Defines the standard designation of a metadata item (optional). If it does not exist, the key is used as the designation.	
metadata:<key>:caption:<language>	Defines the designation of a metadata item for the language specified (optional). The language code must consist of two letters pursuant to the ISO 639-1 standard. If it does not exist, the standard designation or key is used.	
metadata:<key>:type	(optional) defines the data type of the meta data	Equals the TaskApp data type (**String**, **Number**, **Money**, **Date**). If no type is specified, the type **String** is assumed.

The property **value** can contain either a fixed value or an expression. This value can be a maximum of 255 characters long. When an expression is used, this restriction applies to the result of the expression. In addition, the result of an expression must be an individual value. Values of the type **Collection** or **Array** are not permitted.

Determining the storage duration of a user activity

You can define the storage duration of user activities within the BPMN item **userTask**. Add the item **extensionElements** and, under it, the item **camunda:properties** if the process definition does not contain these items yet. You can now define the storage duration as follows under the item **camunda:properties**.

```
<userTask>
...
        <extensionElements>
                <camunda:properties>
                        <camunda:property name="retentionTime" value="P7D"
/>
                </camunda:properties>
        </extensionElements>
...
</userTask>
```

The property **value** can contain either a fixed value or an expression. Define the storage duration as a period of time in days pursuant to ISO 8601, for example, **P30D** for 30 days.

Determining the usage of actions in the user interface of a user activity

You can define the usage of actions in the user interface of a user activity within the BPMN item **userTask**. Add the item **extensionElements** and, under it, the item **camunda:properties** if the process definition does not contain these items yet. You can now define the storage duration as follows under the item **camunda:properties**.

```
<userTask>
...
        <extensionElements>
                <camunda:properties>
                        <camunda:property name="actionScope:complete"
value="[list, details]" />
                </camunda:properties>
        </extensionElements>
...
</userTask>
```

The property **name** of the element contains the prefix **actionScope** followed by the action which should be configured (in this example **complete**). The property **value** contains the options for the usage. For information on possible actions and usage options, see the API documentation of the task app.

Adding user interfaces to a user activity

When you use the BPMN item **User Task** (user activity) for the creation of a process model, you can add a user interface to the activity. The user will then e.g. be shown a button which he/she can use to start a task.

To add a user interface, use the property **formKey**.

Example

Form Key

```
<userTask id="someTask" camunda:formKey="uri:/myapp/myform">
  ...
</userTask>
```

The key consists of a prefix and a value. These are separated by a colon. The following prefixes can be used:

Prefix	Meaning	Example
uri	The user interface is indicated using a URI.	uri:https://example.com/

Using process variables

The key can also contain process variables. These can be indicated in the form of Expression Language.

Example

```
uri:${variables.get("formUri")}

uri:https://example.com/?queryParam=${variables.get("formParam")}
```

External access to process variables

You can use the expression **${process.task.variablesUri}** to generate a URI in order to permit an external application to access the current variables of this user interface. You can use the HTTP methods **GET** and **PUT** to read and change these variables. In this user interface, an input and output mapping needs to be defined for the variables used.

Example

```
uri:/myApp?variablesUri=${process.task.variablesUri}
```

Using process variables in a user interface

If a user activity contains a form, you have read and write access via HTTP to the process variables used.

When getting started with process variables in a form, it is useful to create a simple process with one user activity first. This process will e.g. look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process id="GettingStarted" name="Example: Getting started"
isExecutable="true">
    <startEvent id="start"/>
    <userTask id='task_hello_world' name='Hello World'
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <endEvent id="end"/>
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
  </process>
</definitions>
```

To simplify the example, the graphic information on the BPMN diagram is not included in this BPMN sample file.

Creating a form

First create an HTML file. This may look as follows. The code examples are based on the functionality in Google Chrome. Other browsers may require different syntax.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Getting started</title>
```

```
</head>
<body/>
</html>
```

Store the HTML file so that it is accessible by an HTTP gateway. Add the URI of the HTML document to the BPMN item **userTask** .

```
...
<process id="GettingStarted" name="Example: Getting started"
isExecutable="true">
        ...
    <userTask id='task_hello_world'
name='Hello World' camunda:formKey="uri:/res/process/process-
form.html?variables=${process.task.variablesUri}" camunda:candidateUsers="$
{variables.get('dv_initiator')}" />
    ...
```

Explanation of the property

- **camunda:formKey**: This property defines the URI which deploys the user interface for the user activity. The prefix **uri:** is mandatory here. The variable **${process.task.variablesUri}** is appended to the URI as a query parameter. The variable is resolved by the application during runtime into a URI which deploys an HTTP endpoint for reading and writing of the variables.

Creating the user interface

First create the basic structure for an HTML table in your HTML document. Over the further course, the variables will then be displayed in this table.

```
...
<body>
    <table id="variables">
                <thead>
                        <tr>
                                <th>Name</th>
                                <th>Value</th>
                        </tr>
                </thead>
                <tbody id="variableValues">
                </tbody>
        </table>
</body>
...
```

Querying the process variables

In the head of the HTML document, you create a function for reading the process variables. The process variables are queried by the application using an HTTP GET request according to the REST API. Loading the variables is initiated by the event **onLoad** of the item **body**.

```
...
<head>
        ...
        <script type="text/javascript">

                var urlParams = new URLSearchParams(window.location.search);
                var variablesUri = urlParams.get('variables');
```

```
                function loadVariables() {
                        fetch(variablesUri)
                                .then(response => response.json());
                }

        </script>
</head>
<body onload="loadVariables()">
...
```

After the variables have been loaded, you can now view them in the prepared table. The following example uses the JavaScript library jQuery. Add the library jQuery in the head of the HTML document and implement suitable methods to view the variables.

```
...
<head>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/
jquery.min.js"></script>
        <script type="text/javascript">

                var urlParams = new URLSearchParams(window.location.search);
                var variablesUri = urlParams.get('variables');

                function loadVariables() {
                        fetch(variablesUri)
                                .then(response => response.json())
                                .then(json =>
createTableFromVariables(json.variables || {}));
                }

                function createTableFromVariables(variables) {
                        Object.keys(variables).forEach(variable => {
                                var value = variables[variable];
                                $('#variableValues').append(
`<tr class="variable">${createKeyColumn(variable)}$
{createValueColumn(value)}</tr>`);
                        });
                }

                function createKeyColumn(variable) {
                        return `<td><span class="key">${variable}</span></
td>`;
                }

                function createValueColumn(value) {
                        return `<td><input class="value" type="text"
value="${value}"></td>`;
                }

        </script>
</head>
<body onload="loadVariables()">
...
```

The variables are now queried and displayed after the HTML document has been loaded.

Writing the process variables

Add methods to the script section which enable writing the variables back into the application. Then add a **button** below the table which calls the method **saveVariables**.

```
...
<head>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/
jquery.min.js"></script>
        <script type="text/javascript">

                var urlParams = new URLSearchParams(window.location.search);
                var variablesUri = urlParams.get('variables');

                ...

                function saveVariables() {
                        var variables = createVariablesFromTable();

                        // These variables are read-only and must not be
sent to server when setting variables
                        delete variables['dv_initiator'];
                        delete variables['dv_start_time'];

                    const headers = new Headers();
                        headers.append('Cache-Control', 'no-cache');
                        headers.append('Content-Type', 'application/json');

                        let promise = fetch(variablesUri, {
                                method: 'PUT',
                                headers: headers,
                                credentials: 'same-origin',
                                body: JSON.stringify({variables})
                        });

                        promise.then(function(response) {
                                let status = response.status;
                                if (status !== 200 ) {
                                        window.alert("Saving variables
failed: " + status);
                                } else {
                                        window.alert("Variables saved");
                                }

                        });
                }

                function createVariablesFromTable() {
                        var variables = {};
                        $("tr.variable").each(function() {
                                $this = $(this);
                                var key = $this.find("span.key").html();
                                var value = $this.find("input.value").val();
                                variables[key] = value;
                        });
                        return variables;
```

```
                }

        </script>
</head>
<body onload="loadVariables()">
        ...
        <br/>
        <button onClick="saveVariables()">Save</button>
</body>
```

You can use the form displayed on execution of the process to view and change the current process variables.

The final HTML document will look as follows. The HTML document contains several complementary CSS definitions for optimized display of the user interface items.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Process Form</title>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/
jquery.min.js"></script>
        <style>
                table {
                  border-collapse: collapse;
                  width: 100%;
                }

                table, th, td {
                  border: 1px solid black;
                }

                td {
                  padding: 3px;
                }

                .key {
                        color: gray;
                }

                th {
                  text-align: left;
                }

                input {
                        width: 100%;
                        border: 0;
                }
        </style>
    <script type="text/javascript">

                var urlParams = new URLSearchParams(window.location.search);
                var variablesUri = urlParams.get('variables');

                function loadVariables() {
```

```
                        fetch(variablesUri)
                                .then(response => response.json())
                                .then(json =>
createTableFromVariables(json.variables || {}));
                }

                function saveVariables() {
                        var variables = createVariablesFromTable();

                        // These variables are read-only and must not be
sent to server when setting variables
                        delete variables['dv_initiator'];
                        delete variables['dv_start_time'];

                const headers = new Headers();
                        headers.append('Cache-Control', 'no-cache');
                        headers.append('Content-Type', 'application/json');

                        let promise = fetch(variablesUri, {
                                method: 'PUT',
                                headers: headers,
                                credentials: 'same-origin',
                                body: JSON.stringify({variables})
                        });

                        promise.then(function(response) {
                                let status = response.status;
                                if (status !== 200 ) {
                                        window.alert("Saving variables
failed: " + status);
                                } else {
                                        window.alert("Variables saved");
                                }

                        });
                }

                function createTableFromVariables(variables) {
                        Object.keys(variables).forEach(variable => {
                                var value = variables[variable];
                                $('#variableValues').append(
`<tr class="variable">${createKeyColumn(variable)}$
{createValueColumn(value)}</tr>`);
                        });
                }

                function createKeyColumn(variable) {
                        return `<td><span class="key">${variable}</span></
td>`;
                }

                function createValueColumn(value) {
                        return `<td><input class="value" type="text"
value="${value}"></td>`;
                }
```

```
                function createVariablesFromTable() {
                        var variables = {};
                        $("tr.variable").each(function() {
                                $this = $(this);
                                var key = $this.find("span.key").html();
                                var value = $this.find("input.value").val();
                                variables[key] = value;
                        });
                        return variables;
                }

        </script>
</head>
<body onload="loadVariables()">
    <table id="variables">
                <thead>
                        <tr>
                                <th>Name</th>
                                <th>Value</th>
                        </tr>
                </thead>
                <tbody id="variableValues">
                </tbody>
        </table>
        <br/>
        <button onClick="saveVariables()">Save</button>
</body>
</html>
```

1.4.9. Using a multi-instance

You can add multiple executions to user activities and subprocesses. This requires that the activity or the process is based on a multi-value variable. The activity is executed once for each value of the variable.

When getting started with multi-instances, it is useful to create a simple process with one user activity first. This process will e.g. look like this:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process id="multi_instance_process" name="Multi Instance Process"
isExecutable="true">
    <startEvent id="start" />
        <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
```

```
  </process>
</definitions>
```

Adding a multi-instance

Add the BPMN item **multiInstanceLoopCharacteristics** to the user activity. Then change the variable which defines the recipient of the user activity to **myPerformer**.

```
...
<process id="multi_instance_process" name="Multi Instance Process"
isExecutable="true">
  ...
  <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('myPerformer')}" >
      <multiInstanceLoopCharacteristics
          isSequential="false"
          camunda:collection="${variables.get('performers')}"
          camunda:elementVariable="myPerformer" />
  </userTask>
  ...
</process>
```

Explanation of the properties

- **isSequential**: Use the value **true** to execute the activities sequentially and **false** to execute them simultaneously.
- **camunda:collection**: Expression returning a multi-variable.
- **camunda:elementVariable**: Variable name used locally for the respective value. A definition must be available for the variable.

In the sample process, an activity is sent for each value of the variable **performers**. The local variable **myPerformer** can be used to indicate that each individual item is a recipient.

In order to be able to use the variables **performers** and **myPerformer**, you need to define them. Add the BPMN items **extensionElements** and **camunda:properties** to the BPMN item **process**. To define the process variables, then add the item **camunda:property** to each of the BPMN items **camunda:properties**.

```
...
<process id="multi_instance_process" name="Multi Instance Process"
isExecutable="true">
  <extensionElements>
    <camunda:properties>
      <camunda:property name="variable:performers*" value="[Identity]!" />
      <camunda:property name="variable:myPerformer" value="Identity"
/>
    </camunda:properties>
  </extensionElements>
  ...
</process>
```

In order for the BPMN diagram to be displayed correctly in the modeling tool as well as the user interface, diagram information has been added. The final BPMN definition looks as follows:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" xmlns:bpmndi="http://
www.omg.org/spec/BPMN/20100524/DI" xmlns:dc="http://www.omg.org/spec/DD/
20100524/DC" xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
targetNamespace="" exporter="d.velop process modeler">
```

```
  <process id="multi_instance_process" name="Multi Instance Process"
isExecutable="true">
    <extensionElements>
      <camunda:properties>
        <camunda:property name="variable:performers*" value="[Identity]!" />
        <camunda:property name="variable:myPerformer" value="Identity"
/>
      </camunda:properties>
    </extensionElements>
    <startEvent id="start" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('myPerformer')}" >
      <multiInstanceLoopCharacteristics
            isSequential="false"
            camunda:collection="${variables.get('performers')}"
            camunda:elementVariable="myPerformer" />
    </userTask>
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
  </process>

  <!-- Diagram information -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="multi_instance_process">
      <bpmndi:BPMNShape id="start_di" bpmnElement="start">
        <dc:Bounds x="179" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="task_hello_world_di"
bpmnElement="task_hello_world">
        <dc:Bounds x="290" y="77" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="end_di" bpmnElement="end">
        <dc:Bounds x="462" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s1_di" bpmnElement="s1">
        <di:waypoint x="215" y="117" />
        <di:waypoint x="290" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s2_di" bpmnElement="s2">
        <di:waypoint x="390" y="117" />
        <di:waypoint x="462" y="117" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

You can now define the value of the variable **performers** e.g. for the start of the process.

1.4.10. Using a start form

You can add a form to a start event, e.g. an application for vacation leave. You can the use the start form as the entry point for this process. To add a form to a start event, use the property **formKey**.

```
<startEvent id="start" name="start" camunda:formKey="uri:/myapp/mystart">
    ...
</startEvent>
```

1.4.11. Using services

In this topic, you will find out how to embed services in your processes. When inserting services, you can decide whether the service will be executed synchronously or asynchronously.

Using a synchronous service

Automated activities are executed in a process by means of services. For the use of synchronous services in BPMN, you can use the BPMN item **Send Task** (send activity). When the process reaches the send activity, the data defined in the process are sent to the HTTP endpoint of the service provider and immediately wait for the result. The process execution is then continued directly. Using a simple service example, this item shows you how you can use a synchronous service. The service example "Hello World" changes any text to "Hello <yourname>" where <yourname> is the text which was sent to the service beforehand.

To execute this example, it is useful to create a simple process with one user activity first. The BPMN model of this sample process is structured as follows:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process id="sync_service_process" name="Sync Service Process"
isExecutable="true">
    <startEvent id="start" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
  </process>
</definitions>
```

Adding a send activity

First add a send activity after the start event.

```
...
<process id="sync_service_process" name="Sync Service Process"
isExecutable="true">
  <startEvent id="start" />
  <sendTask id="call_service"
        name="Call Service 'Hello World'"
        camunda:asyncBefore="true"
            camunda:asyncAfter="true"
        camunda:delegateExpression="${syncService}"
        camunda:exclusive="true">
  </sendTask>
  ...
</process>
```

Explanation of the properties:

- **id**: The property is used as a unique identifier for the send activity.
- **name**: This property is used as a display name in the user interfaces.
- **camunda:***: These properties contain technical information required for the execution.

Note

For **synchronous services**, you always need to enter the values of the example for the **camunda** properties. If you enter different values, the process cannot be deployed.

Now add the BPMN items **extensionElements** and **camunda:inputOutput** to the send activity. You need to define all data which are to be sent to the HTTP endpoint of the service. To define the data, add the BPMN item **camunda:inputParameter** to the process definition for each value. Next you also need to define the expected output values. To do so, add the BPMN item **camunda:outputParameter** to the process definition for each value.

```
...
<process id="sync_service_process" name="Sync Service Process"
isExecutable="true">
  <startEvent id="start" />
  <sendTask id="call_service" ...>
    <extensionElements>
      <camunda:inputOutput>
        <camunda:inputParameter name="service.uri">/process/services/
helloworld/sync</camunda:inputParameter>
        <camunda:inputParameter name="yourName">$
{variables.getDisplayValue("dv_initiator")}</camunda:inputParameter>
        <camunda:outputParameter name="greeting">$
{variables.get("greeting")}</camunda:outputParameter>
      </camunda:inputOutput>
    </extensionElements>
  </sendTask>
  ...
</process>
```

Explanation of the parameters:

- **service.uri**: This parameter must always be inserted. It defines the URI used to reach the HTTP endpoint of the service. The example shows the URI of the service "Hello World". The value of this variable must be a constant, i.e. it must not contain any expressions.
- **yourName**: This parameter was defined as an input value by the specific service. This example uses the view name of the user who has started the process.
- **greeting**: The service has defined the parameter **greeting** as the only output value. This parameter is automatically written to a variable of the same name in the scope of the send activity when the service responds. The parameter defines that the content of this result value is written into the variable **greeting** in the scope of the entire process when the process is continued.

To enable the communication between the application and the service, you need to add the interface of the service to the process. To add the interface, enter the BPMN items **extensionElements** and **camunda:properties** in the process definition. Then add one BPMN item **camunda:property** each for the service input value **yourName**, the service output value **greeting**, as well as the resulting process variable **greeting** . All values are of data type **String**.

```
...
<process id="sync_service_process" name="Sync Service Process"
isExecutable="true">
  <extensionElements>
    <camunda:properties>
      <camunda:property name="service:/process/services/helloworld/
sync:in:yourName" value="String" />
      <camunda:property name="service:/process/services/helloworld/
```

```
sync:out:greeting" value="String" />
          <camunda:property name="service:/process/services/helloworld/
sync:name" value="Hello World" />
      <camunda:property name="variable:greeting" value="String" />
    </camunda:properties>
  </extensionElements>
  ...
</process>
```

The variable **greeting** now contains the text "Hello <display name of process start user>". Now use this variable as the name of the user activity so that the user receives the friendly greeting as the subject of the task to complete.

```
...
<process id="sync_service_process" name="Sync Service Process"
isExecutable="true">
  ...
  <userTask id="task_hello_world" name="${variables.get('greeting')}"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
  ...
</process>
```

To ensure that the process activities are executed in the right sequence, you still need to adapt the sequence flow items to the new activities.

```
<process id="sync_service_process" name="Sync Service Process"
isExecutable="true">
  ...
  <sequenceFlow id="s1" sourceRef="start" targetRef="call_service" />
  <sequenceFlow id="s2" sourceRef="call_service"
targetRef="task_hello_world" />
  <sequenceFlow id="s3" sourceRef="task_hello_world" targetRef="end" />
</process>
```

In order for the BPMN diagram to be displayed correctly in the modeling tool as well as the user interface, diagram information has been added. The final BPMN definition looks as follows:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" xmlns:bpmndi="http://
www.omg.org/spec/BPMN/20100524/DI" xmlns:di="http://www.omg.org/spec/DD/
20100524/DI" xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
targetNamespace="" exporter="d.velop process modeler">
  <process id="sync_service_process" name="Sync Service Process"
isExecutable="true">
    <extensionElements>
      <camunda:properties>
        <camunda:property name="service:/process/services/helloworld/
sync:in:yourName" value="String" />
        <camunda:property name="service:/process/services/helloworld/
sync:out:greeting" value="String" />
        <camunda:property name="service:/process/services/helloworld/
sync:name" value="Hello World" />
        <camunda:property name="variable:greeting" value="String" />
      </camunda:properties>
    </extensionElements>
    <startEvent id="start" />
    <sendTask id="call_service"
```

```
          name="Call Service 'Hello World'"
          camunda:asyncBefore="true"
          camunda:asyncAfter="true"
          camunda:delegateExpression="${syncService}"
          camunda:exclusive="true">
      <extensionElements>
        <camunda:inputOutput>
          <camunda:inputParameter name="service.uri">/process/services/
helloworld/sync</camunda:inputParameter>
          <camunda:inputParameter name="yourName">$
{variables.getDisplayValue("dv_initiator")}</camunda:inputParameter>
          <camunda:outputParameter name="greeting">$
{variables.get("greeting")}</camunda:outputParameter>
        </camunda:inputOutput>
      </extensionElements>
    </sendTask>
    <userTask id="task_hello_world" name="${variables.get('greeting')}"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="call_service" />
    <sequenceFlow id="s2" sourceRef="call_service"
targetRef="task_hello_world" />
    <sequenceFlow id="s3" sourceRef="task_hello_world" targetRef="end" />
  </process>

  <!-- Diagram information -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane" bpmnElement="sync_service_process">
      <bpmndi:BPMNShape id="start_di" bpmnElement="start">
        <dc:Bounds x="173" y="102" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s1_di" bpmnElement="s1">
        <di:waypoint x="209" y="120" />
        <di:waypoint x="250" y="120" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="call_service_di" bpmnElement="call_service">
        <dc:Bounds x="250" y="80" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s2_di" bpmnElement="s2">
        <di:waypoint x="350" y="120" />
        <di:waypoint x="390" y="120" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="task_hello_world_di"
bpmnElement="task_hello_world">
        <dc:Bounds x="390" y="80" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s3_di" bpmnElement="s3">
        <di:waypoint x="490" y="120" />
        <di:waypoint x="532" y="120" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="end_di" bpmnElement="end">
        <dc:Bounds x="532" y="102" width="36" height="36" />
      </bpmndi:BPMNShape>
    </bpmndi:BPMNPlane>
```

```
  </bpmndi:BPMNDiagram>
</definitions>
```

Using an asynchronous service

Automated activities are executed in a process by means of services. For the use of asynchronous services in BPMN, you can use the BPMN items **Send Task** (send activity) and **Receive Task** (receive activity). When the process reaches the send activity, the data defined in the process are sent to the HTTP endpoint of the service provider. The process execution is then put on hold until the service provider returns the processing result. Using a simple service example, this item shows you how you can use an asynchronous service. The service example "Hello World" changes any text to "Hello <yourname>" where <yourname> is the text which was sent to the service beforehand.

To execute this example, it is useful to create a simple process with one user activity first. The BPMN model of this sample process is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process id="async_service_process" name="Async Service Process"
isExecutable="true">
    <startEvent id="start" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
  </process>
</definitions>
```

Adding a send activity

First add a send activity after the start event.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  <startEvent id="start"/>
  <sendTask id="call_service"
        name="Call Service 'Hello World'"
        camunda:asyncBefore="true"
        camunda:delegateExpression="${asyncService}"
        camunda:exclusive="true">
  </sendTask>
  ...
</process>
```

Explanation of the properties:

- **id**: The property is used as a unique identifier for the send activity.
- **name**: This property is used as a display name in the user interfaces.
- **camunda:***: These properties contain technical information required for the execution.

> **Note**
> For **asynchronous services**, you always need to enter the values of the example for the **camunda** properties. If you enter different values, the process cannot be deployed.

Now add the BPMN items **extensionElements** and **camunda:inputOutput** to the send activity. You need to define all data which are to be sent to the HTTP endpoint of the service. To define the data, add the BPMN item **camunda:inputParameter** to the process definition for each value.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  <startEvent id="start"/>
  <sendTask id="call_service" ...>
    <extensionElements>
      <camunda:inputOutput>
        <camunda:inputParameter name="service.uri">/process/services/
helloworld/async</camunda:inputParameter>
        <camunda:inputParameter name="yourName">$
{variables.getDisplayValue("dv_initiator")}</camunda:inputParameter>
                <camunda:inputParameter name="service.callbackBase">/</
camunda:inputParameter>
          </camunda:inputOutput>
    </extensionElements>
  </sendTask>
  ...
</process>
```

Explanation of the parameters:

- **service.uri**: This parameter is mandatory and therefore must always be inserted. The parameter defines the URI used to reach the HTTP endpoint of the service. The example shows the URI of the service "Hello World". The value of this variable must be a constant, i.e. the value must not contain any expressions.
- **yourName**: This parameter was defined as an input value by the specific service. This example uses the view name of the user who has started the process.

Modifying the response URIs

You can change the origin of the URIs passed to the service for the response. You have the following two options for this.

Specifying the Origin on the SendTask of a concrete service

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  <startEvent id="start"/>
  <sendTask id="call_service" ...>
    <extensionElements>
      <camunda:inputOutput>
        ...
                <camunda:inputParameter name="service.callbackBase">/</
camunda:inputParameter>
                ...
          </camunda:inputOutput>
    </extensionElements>
  </sendTask>
  ...
</process>
```

Add another **camunda:inputParameter** named **service.callbackBase** to the SendTask. This parameter is valid for this specific service call.

Specifying the Origin for the entire process

Enter the BPMN elements **extensionElements** and **camunda:properties** into the process definition.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  <extensionElements>
    <camunda:properties>
          ...
      <camunda:property name="service:/process/services/helloworld/
async:callbackBase" value="/" />
      <!-- alternative <camunda:property name="service:*:callbackBase"
value="/" /> -->
      ...
    </camunda:properties>
  </extensionElements>
  ...
</process>
```

Create a **camunda:property** element named **service:<service>:callbackBase**. The **<service>** placeholder represents the URI of the service, or * for all services.

The specified Origin must match the format **https://domain[:port]** in both cases. Alternatively, the specification of / represents a relative URI.

Adding a receive activity

Now add the receive activity to the process definition. The receive activity is used to relay the processed data from the service to the process.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  ...
  </sendTask>
  <receiveTask id="receive_service_response"
        name="Wait for Service 'Hello World'"
        camunda:asyncAfter="true"
        camunda:exclusive="true">
  </receiveTask>
  ...
</process>
```

Explanation of the properties:

- **id**: The property is used as a unique identifier for the send activity.
- **name**: This property is used as a display name in the user interfaces.
- **camunda:***: These properties contain technical information required for the execution. For asynchronous services, you always need to enter the value **true** for the property **camunda:asyncAfter**.

Add the BPMN items **extensionElements** and **camunda:inputOutput** to the receive activity. You need to define all data which are expected as a response by the HTTP endpoint of the service. To define the data, insert the BPMN item **camunda:outputParameter** for each value.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
```

```
  ...
  </sendTask>
  <receiveTask id="receive_service_response" ...>
    <extensionElements>
      <camunda:inputOutput>
        <camunda:outputParameter name="greeting">$
{variables.get("greeting")}</camunda:outputParameter>
      </camunda:inputOutput>
    </extensionElements>
  </receiveTask>
  ...
</process>
```

Explanation of the parameters:

- **greeting**: The service has defined the parameter **greeting** as the only output value. This parameter is automatically written to a variable of the same name in the scope of the receive activity when the service responds. The parameter defines that the content of this result value is written into the variable **greeting** in the scope of the entire process when the process is continued.

To enable the communication between the application and the service, you need to add the interface of the service to the process. To add the interface, enter the BPMN items **extensionElements** and **camunda:properties** in the process definition, if not already done in previous steps. Then add one BPMN item **camunda:property** each for the service input value **yourName**, the service output value **greeting**, as well as the resulting process variable **greeting** . All values are of data type **String**.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  <extensionElements>
    <camunda:properties>
      <camunda:property name="service:/process/services/helloworld/
async:in:yourName" value="String" />
      <camunda:property name="service:/process/services/helloworld/
async:out:greeting" value="String" />
      <camunda:property name="service:/process/services/helloworld/
async:name" value="Hello World" />
      <camunda:property name="variable:greeting" value="String" />
    </camunda:properties>
  </extensionElements>
  ...
</process>
```

The variable **greeting** now contains the text "Hello <display name of process start user>". Now use this variable as the name of the user activity so that the user receives the friendly greeting as the subject of the task to complete.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  ...
  <userTask id="task_hello_world" name="${variables.get('greeting')}"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
  ...
</process>
```

To ensure that the process activities are executed in the right sequence, you still need to adapt the sequence flow items to the new activities.

```
...
<process id="async_service_process" name="Async Service Process"
isExecutable="true">
  ...
  <endEvent id="end" />
  <sequenceFlow id="s1" sourceRef="start" targetRef="call_service" />
  <sequenceFlow id="s2" sourceRef="call_service"
targetRef="receive_service_response" />
  <sequenceFlow id="s3" sourceRef="receive_service_response"
targetRef="task_hello_world" />
  <sequenceFlow id="s4" sourceRef="task_hello_world" targetRef="end" />
</process>
```

In order for the BPMN diagram to be displayed correctly in the modeling tool as well as the user interface, diagram information has been added. The final BPMN definition looks as follows:

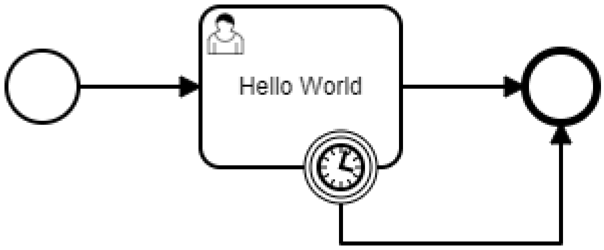
```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" xmlns:bpmndi="http://
www.omg.org/spec/BPMN/20100524/DI" xmlns:di="http://www.omg.org/spec/DD/
20100524/DI"  xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
targetNamespace="" exporter="d.velop process modeler">
  <process id="async_service_process" name="Async Service Process"
isExecutable="true">
    <extensionElements>
      <camunda:properties>
        <camunda:property name="service:/process/services/helloworld/
async:in:yourName" value="String" />
        <camunda:property name="service:/process/services/helloworld/
async:out:greeting" value="String" />
        <camunda:property name="service:/process/services/helloworld/
async:name" value="Hello World" />
        <camunda:property name="variable:greeting" value="String" />
      </camunda:properties>
    </extensionElements>
    <startEvent id="start" />
    <sendTask id="call_service"
          name="Call Service 'Hello World'"
          camunda:asyncBefore="true"
          camunda:delegateExpression="${asyncService}"
          camunda:exclusive="true">
      <extensionElements>
        <camunda:inputOutput>
          <camunda:inputParameter name="service.uri">/process/services/
helloworld/async</camunda:inputParameter>
          <camunda:inputParameter name="yourName">$
{variables.getDisplayValue("dv_initiator")}</camunda:inputParameter>
        </camunda:inputOutput>
      </extensionElements>
    </sendTask>
    <receiveTask id="receive_service_response"
          name="Wait for Service 'Hello World'"
          camunda:asyncAfter="true"
          camunda:exclusive="true">
      <extensionElements>
        <camunda:inputOutput>
```

```
          <camunda:outputParameter name="greeting">$
{variables.get("greeting")}</camunda:outputParameter>
        </camunda:inputOutput>
      </extensionElements>
    </receiveTask>
    <userTask id="task_hello_world" name="${variables.get('greeting')}"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="call_service" />
    <sequenceFlow id="s2" sourceRef="call_service"
targetRef="receive_service_response" />
    <sequenceFlow id="s3" sourceRef="receive_service_response"
targetRef="task_hello_world" />
    <sequenceFlow id="s4" sourceRef="task_hello_world" targetRef="end" />
  </process>

  <!-- Diagram information -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="async_service_process">
      <bpmndi:BPMNEdge id="s4_di" bpmnElement="s4">
        <di:waypoint x="630" y="117" />
        <di:waypoint x="662" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s3_di" bpmnElement="s3">
        <di:waypoint x="490" y="117" />
        <di:waypoint x="530" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s2_di" bpmnElement="s2">
        <di:waypoint x="350" y="117" />
        <di:waypoint x="390" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s1_di" bpmnElement="s1">
        <di:waypoint x="215" y="117" />
        <di:waypoint x="250" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="start_di" bpmnElement="start">
        <dc:Bounds x="179" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="call_service_di" bpmnElement="call_service">
        <dc:Bounds x="250" y="77" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="receive_service_response_di"
bpmnElement="receive_service_response">
        <dc:Bounds x="390" y="77" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="task_hello_world_di"
bpmnElement="task_hello_world">
        <dc:Bounds x="530" y="77" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="end_di" bpmnElement="end">
        <dc:Bounds x="662" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

1.4.12. Using escalations

When modeling your own processes, you can use escalations to influence the process flow. When you start using escalations, it is useful to create a simple process with one user activity and one timer event first. This process will e.g. look like this:



The BPMN model of this sample process is structured as follows:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" targetNamespace=""
exporter="d.velop process modeler">
  <process id="escalation_process" name="Escalation Process"
isExecutable="true">
    <startEvent id="start" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <boundaryEvent id="timer" attachedToRef="task_hello_world">
      <timerEventDefinition>
        <timeDuration>PT1M</timeDuration>
      </timerEventDefinition>
    </boundaryEvent>
    <endEvent id="end" />
    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
    <sequenceFlow id="s3" sourceRef="timer" targetRef="end" />
  </process>
</definitions>
```

We now want to modify this sample process so an escalation is triggered when a timer event occurs. The escalation will then start a subprocess that describes the escalation path. First, you need to create an escalation object for this modification.

Adding an escalation object

First add the BPMN item **escalation**.

```
...
<definitions ...>
  <process id="escalation_process" name="Escalation Process"
isExecutable="true">
    ...
  </process>
  <escalation id="time_escalation" name="Time Escalation"
escalationCode="123" />
</definitions>
```

Explanation of the properties:

- **id:** The property is used as a unique identifier for the escalation.
- **name**: The display name of the escalation.
- **escalationCode:** The property defines a technical code that can be used on items waiting for escalations in order to identify the triggering escalation.

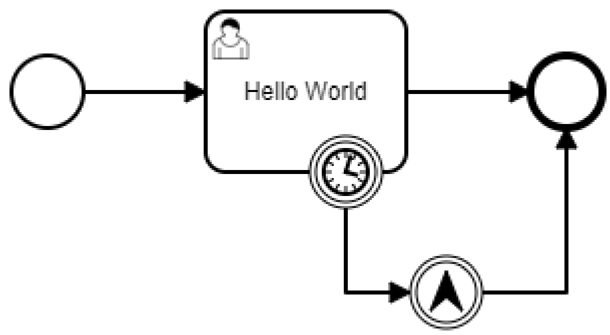
Now the escalation can be used in the BPMN item **IntermediateThrowEvent** or an **EndEvent** in order to trigger a corresponding escalation event.

Adding an escalation event

First add the BPMN item **IntermediateThrowEvent** with an **EscalationEventDefinition** element to the process. Then change the target of the sequence flow **s3** running from **end** to **throw_time_escalation**, and then add a new sequence flow **s4** running from **throw_time_escalation** to **end**.

```
...
<process id="escalation_process" name="Escalation Process"
isExecutable="true">
  ...
  <intermediateThrowEvent id="throw_time_escalation">
    <escalationEventDefinition escalationRef="time_escalation" />
  </intermediateThrowEvent>
  <endEvent id="end" />
  ...
  <sequenceFlow id="s3" sourceRef="timer" targetRef="throw_time_escalation"
/>
  <sequenceFlow id="s4" sourceRef="throw_time_escalation" targetRef="end" />
</process>
<escalation id="time_escalation" name="Time Escalation"
escalationCode="123" />
...
```

This is what the process now looks like:



Adding an escalation path

Now add a subprocess whose BPMN item **StartEvent** contains the item **EscalationEventDefinition**.

```
...
<process id="escalation_process" name="Escalation Process"
isExecutable="true">
  <extensionElements>
    <camunda:properties>
      <camunda:property name="variable:escalationCode" value="String" />
    </camunda:properties>
  </extensionElements>
```

```
  <startEvent id="start" />
  ...
  <sequenceFlow id="s4" sourceRef="throw_time_escalation" targetRef="end" />
  <subProcess id="escalation_subprocess" name="Escalation Process"
triggeredByEvent="true">
    <startEvent id="sub_start" isInterrupting="false">
      <escalationEventDefinition escalationRef="time_escalation"
camunda:escalationCodeVariable="escalationCode" />
    </startEvent>
    <endEvent id="sub_end" />
    <sequenceFlow id="sub_s1" sourceRef="sub_start" targetRef="sub_end" />
  </subProcess>
</process>
<escalation id="time_escalation" name="Time Escalation"
escalationCode="123" />
...
```

Explanation of the properties:

- subProcess
  - **triggeredByEvent**: It must contain the value **true**. The property specifies that the start event of this subprocess is triggered by an event (an escalation event in this case).
  - **isInterrupting**: If it is **true**, the process branch that triggered the escalation is terminated. If it is **false**, the process branch continues to run in parallel.
- escalationEventDefinition
  - **escalationRef:** Contains the ID of the escalation that is to be used to trigger this event.
  - **camunda:escalationCodeVariable:** (optional) name of the variables in which the value of **escalationCode** of the occurring escalation is to be saved (variable must be defined).

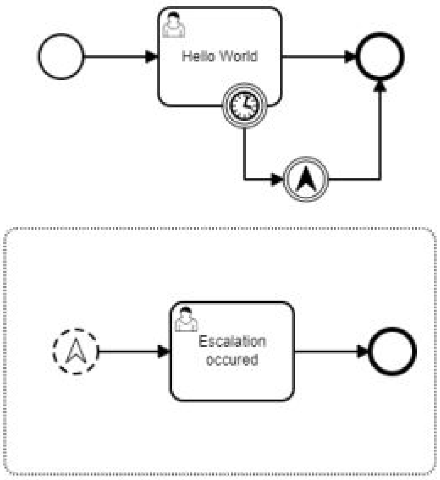
Note that when you use the property **camunda:escalationCodeVariable**, you also need to add a definition of the corresponding process variables in the main process.

Now add an additional user activity that should be run in the event of an escalation.

```
...
<process id="escalation_process" name="Escalation Process"
isExecutable="true">
  ...
  <subProcess id="escalation_subprocess" name="Escalation Process"
triggeredByEvent="true">
    <startEvent id="sub_start" isInterrupting="false">
      <escalationEventDefinition escalationRef="time_escalation"
camunda:escalationCodeVariable="escalationCode" />
    </startEvent>
      <userTask id="sub_task_escalation" name="Escalation occurred"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <endEvent id="sub_end" />
    <sequenceFlow id="sub_s1" sourceRef="sub_start"
targetRef="sub_task_escalation" />
    <sequenceFlow id="sub_s2" sourceRef="sub_task_escalation"
targetRef="sub_end" />
  </subProcess>
</process>
<escalation id="time_escalation" name="Time Escalation"
escalationCode="123" />
...
```

For reasons of simplification, the user that started the process is entered as the recipient of the user activity of the escalation path.

This is what the final process looks like:



In order for the BPMN diagram to be displayed correctly in the modeling tool as well as the user interface, diagram information has been added. The final BPMN definition looks as follows:

```
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:camunda="http://camunda.org/schema/1.0/bpmn" xmlns:bpmndi="http://
www.omg.org/spec/BPMN/20100524/DI" xmlns:di="http://www.omg.org/spec/DD/
20100524/DI"  xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
targetNamespace="" exporter="d.velop process modeler">
  <process id="escalation_process" name="Escalation Process"
isExecutable="true">
    <extensionElements>
      <camunda:properties>
        <camunda:property name="variable:escalationCode" value="String" />
      </camunda:properties>
    </extensionElements>

    <startEvent id="start" />
    <userTask id="task_hello_world" name="Hello World"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
    <boundaryEvent id="timer" attachedToRef="task_hello_world">
      <timerEventDefinition>
        <timeDuration>PT1M</timeDuration>
      </timerEventDefinition>
    </boundaryEvent>
    <intermediateThrowEvent id="throw_time_escalation">
      <escalationEventDefinition escalationRef="time_escalation" />
    </intermediateThrowEvent>
    <endEvent id="end" />

    <sequenceFlow id="s1" sourceRef="start" targetRef="task_hello_world" />
    <sequenceFlow id="s2" sourceRef="task_hello_world" targetRef="end" />
    <sequenceFlow id="s3" sourceRef="timer"
targetRef="throw_time_escalation" />
    <sequenceFlow id="s4" sourceRef="throw_time_escalation" targetRef="end"
/>
```

```
    <subProcess id="escalation_subprocess" name="Escalation Process"
triggeredByEvent="true">
      <startEvent id="sub_start" isInterrupting="false">
        <escalationEventDefinition escalationRef="time_escalation"
camunda:escalationCodeVariable="escalationCode" />
      </startEvent>
      <userTask id="sub_task_escalation" name="Escalation occurred"
camunda:candidateUsers="${variables.get('dv_initiator')}" />
      <endEvent id="sub_end" />
      <sequenceFlow id="sub_s1" sourceRef="sub_start"
targetRef="sub_task_escalation" />
      <sequenceFlow id="sub_s2" sourceRef="sub_task_escalation"
targetRef="sub_end" />
    </subProcess>

  </process>
  <escalation id="time_escalation" name="Time Escalation"
escalationCode="123" />

  <!-- Diagram information -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="escalation_process">
      <bpmndi:BPMNShape id="start_di" bpmnElement="start">
        <dc:Bounds x="179" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="task_hello_world_di"
bpmnElement="task_hello_world">
        <dc:Bounds x="290" y="77" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="timer_di" bpmnElement="timer">
        <dc:Bounds x="372" y="139" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="throw_time_escalation_di"
bpmnElement="throw_time_escalation">
        <dc:Bounds x="432" y="202" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="end_di" bpmnElement="end">
        <dc:Bounds x="492" y="99" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="s1_di" bpmnElement="s1">
        <di:waypoint x="215" y="117" />
        <di:waypoint x="290" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s2_di" bpmnElement="s2">
        <di:waypoint x="390" y="117" />
        <di:waypoint x="492" y="117" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s3_di" bpmnElement="s3">
        <di:waypoint x="390" y="175" />
        <di:waypoint x="390" y="220" />
        <di:waypoint x="432" y="220" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="s4_di" bpmnElement="s4">
        <di:waypoint x="468" y="220" />
```

```
        <di:waypoint x="510" y="220" />
        <di:waypoint x="510" y="135" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNShape id="escalation_subprocess_di"
bpmnElement="escalation_subprocess" isExpanded="true">
        <dc:Bounds x="165" y="290" width="350" height="200" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="sub_start_di" bpmnElement="sub_start">
        <dc:Bounds x="205" y="372" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="sub_task_escalation_di"
bpmnElement="sub_task_escalation">
        <dc:Bounds x="290" y="350" width="100" height="80" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="sub_end_di" bpmnElement="sub_end">
        <dc:Bounds x="442" y="372" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="sub_s1_di" bpmnElement="sub_s1">
        <di:waypoint x="241" y="390" />
        <di:waypoint x="290" y="390" />
      </bpmndi:BPMNEdge>
      <bpmndi:BPMNEdge id="sub_s2_di" bpmnElement="sub_s2">
        <di:waypoint x="390" y="390" />
        <di:waypoint x="442" y="390" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>
```

1.5. Modeling forms

d.velop forms is an application for managing forms. With the appropriate administrator authorization, you can create forms. Users that do not have this authorization can open, fill out and submit the forms. Thanks to their responsive design, forms can be opened on any device.

Each created form has an embedding URL. You can use this URL to make the form available to your users. You can also use the form within a process by entering the URL as a processing dialog within a user step.

Forms are created using a graphical user interface. You can arrange various input fields (for text, numbers, date values or a file, for example) in your form. You can also adapt your form to your requirements using CSS or Javascript.

If users open the form for a process step or document, you have access to the process variables/document properties within the form. For instance, this allows you to map the process variables or document properties to the form fields. When the form is opened, the current values for the process variables or document properties are automatically written to the form fields. Users can edit these values. The changed values are then written back to the process variables or document properties when the form is submitted.

1.5.1. Getting started with working with forms

In this topic, you can learn about the basic functions for working with forms. You can learn how to create, open and manage forms.

Calling a form

To ensure that other users can open, fill out and submit your form, you need the URL of the form. This URL is displayed in the browser address bar when you open the form as a preview. If you click **Preview**, the form opens in a new browser tab. This can be helpful for testing the form.

Alternatively, you can obtain the URL of a form by choosing **Copy URL to clipboard** in the three-dot menu for the relevant form in the form list. From the clipboard, you can then either distribute the URL directly to other users or use the URL to integrate the form into a different application (for example, in a task or process).

> **Note**
>
> The absolute URL is used when you copy to the clipboard. When you integrate the URL in other applications, you should specify the URL as relative.

Creating a form

You can create a new form in form management. To do so, you must have the appropriate authorizations.

This is how it works

1. Click the button for adding a form (plus sign) in the toolbar.
2. Enter a name for the form and press **Create**.
3. Select a component from the component menu and drag it to your form using drag and drop. To start with, it is best to limit yourself to the components in the **Basic** category.
4. Adapt the component label. You can try the rest of the settings later.
5. Click **Save**.
6. Add additional components if necessary.
7. Save your form.

> **Note**
>
> D.velop forms uses the "Form Builder" component from form.io as its form builder. You can find a guide to using this component here.

Importing and exporting forms

You can export and import forms. This can be helpful, for example, if you want to transfer a form from one system to another.

This is how it works

1. Open the form overview in the source system by opening the **Process studio** feature on the start page, for example.
2. Open the three-dot menu next to the relevant form and choose **Download form**. The form definition is now downloaded as a JSON file.
3. Open the form overview in the target system.
4. In the toolbar, click the import button (⬆) and select your exported form.

Managing forms

You can find the **Forms** area under **Tasks and processes** in the configuration. Choose **Form Management** to go to the overview list of all the existing forms. In this overview list, you can create new forms and edit or delete existing ones.

Note
You can only view the **Form Management** entry if you have been assigned the **Administration** user role. Only your administrators can assign it to you.

1.5.2. Forms within user activities

You can complete user tasks with the aid of forms. You can complete a task from the form in three different ways:

1. By using an "Event" button in the form
2. By using a custom action in a script
3. By using the method **window.postMessage** in a script

Completing the task with an "Event" button - This is how it works

1. Add a button to your form.
2. Configure the button as follows:
   - **Action: Event**
   - **Button Event: taskCompletion**

When a user opens the task with the form and clicks the button, the task is completed. Any available process variables and document properties are also saved.

Completing the task with an event with a custom action - This is how it works

1. Add a component to your form that supports the execution of a script (e.g. a button with the action **Custom**).
2. Call the following code in the script:

```
form.emit("customEvent", {"type":"taskCompletion"});
```

When a user opens the task with the form and clicks the button, the task is completed. Any available process variables and document properties are also saved.

Unlike when you complete the task with an "Event" button, this variant also lets you execute additional custom code before completing the task.

Completing the task using the method "parent.postMessage" - This is how it works

1. Add a component to your form that supports the execution of a script (e.g. a button with the action **Custom**).
2. Call the following code in the script:

```
parent.postMessage("TaskApp.completeTask", "*");
```

This completes the task without saving any available process variables and document properties.

1.5.3. Using document properties in the form

For access to the properties of a document in a form, you can call the form URL using the following request parameters:

- **dmsRepoId=<dms repo id>**
- **dmsObjectId=<dms object id>**

The document properties are automatically determined from the DMS app while loading the form and made available in the form as JSON objects with the variable **dmsProperties**.

Example for the JSON object "dmsProperties"

```
{
  "myProperty" : "Hello!",
  "otherProperty" : 4711
}
```

> **Note**
>
> If the key of a document property contains items, this results in sub-objects.
>
> **Example:**
>
> The document property with the key **this.is.a.key** and the value **My value** results in the following JSON object:
>
> ```
> {
>   "this" : {
>     "is" : {
>       "a" : {
>         "key" : "My value"
>       }
>     }
>   }
> }
> ```

You can use document properties in two ways:

1. In a script
2. In a form item

Using a document property in a script

In a script, you have both read and write access to the JSON object at all times. When the form is submitted at the end with the event **taskCompletion** or **saveDmsProperties**, the current values are determined from the JSON object and written to the document properties.

Binding a form item to a document property

To bind a form item to a document property, open the **API** tab in the form item settings and specify the document property in the format **dmsProperties.<´dms property key>** under **Property Name**. This form item is now prefilled with the value from the document property when the form is loaded. The user that opened the form can then adjust the value if necessary. When the form is submitted at the end with the event **taskCompletion**, the current value of the form item is written to the document properties again.

> **Warning**
>
> The document properties are loaded and saved in the name of the user that is logged in and that opened the form. Ensure that the user has permissions for the document.

1.5.4. Using process variables in the form

You can use a form as a processing view for a user task in a BPMN process performed with d.velop process. To do so, enter the URL of the form (as a relative URL if possible) as the **form key** for the user task in the format "uri:<form url>".

To ensure that the form can access process variables, add the request parameter **processVariable-sUri=${process.task.variablesUri}** to the URL.

The process variables are determined automatically based on the process while loading the form and made available in the form as JSON objects with the variable **processVariables**.

Example for the JSON object “processVariables"

```
{
  "myVariable" : "Hello!",
  "otherVariable" : 4711
}
```

The process variables can be used in two ways:

1. In a script
2. In a form item

Using a process variable in a script

In a script, you have both read and write access to the JSON object at all times. When the form is submitted at the end with the event **taskCompletion**, the current values for the process variables are determined from the JSON object and sent to the process.

Binding a form item to a process variable

To bind a form item to a process variable, open the **API** tab in the form item settings and specify the process variable in the format **processVariables.<process variable key>** under **Property Name**. This form item is now prefilled with the value from the process variables when the form is loaded. The user that opened the form can then adjust the value if necessary. When the form is submitted at the end with the event **taskCompletion** or **saveProcessVariables**, the current value of the form item is transferred to the process again as a process variable.

> **Note**
> The process variables are loaded and saved in the name of the user that is logged in. To ensure that this user has the right to do so, the user must currently be responsible for processing the related task. This is because the form is displayed in the task.

1.5.5. Using a form in a task

To use a form as a processing view in a task, you can enter the form’s URL as a **form** link when creating the task with the task app API. We recommend entering the URL as a relative URL.

In some cases, the context actions **complete**, **forward** and **claim** are not required for tasks if a form is used as a processing view. You can hide the context actions in the task. To hide the form URL, add the appropriate request parameter below:

- **disableTaskAction=complete**
- **disableTaskAction=forward**
- **disableTaskAction=claim**

Completing a task from a form

You can complete user tasks with the aid of forms. You can complete a task from the form in three different ways:

1. By using an "Event" button in the form
2. By using a custom action in a script
3. By using the method **window.postMessage** in a script

Completing the task with an "Event" button - This is how it works

1. Add a button to your form.
2. Configure the button as follows:

- **Action: Event**
- **Button Event: taskCompletion**

When a user opens the task with the form and clicks the button, the task is completed. Any available process variables and document properties are also saved.

Completing the task with an event with a custom action - This is how it works

1. Add a component to your form that supports the execution of a script (e.g. a button with the action **Custom**).
2. Call the following code in the script:

```
form.emit("customEvent", {"type":"taskCompletion"});
```

When a user opens the task with the form and clicks the button, the task is completed. Any available process variables and document properties are also saved.

Unlike when you complete the task with an "Event" button, this variant also lets you execute additional custom code before completing the task.

Completing the task using the method "parent.postMessage" - This is how it works

1. Add a component to your form that supports the execution of a script (e.g. a button with the action **Custom**).
2. Call the following code in the script:

```
parent.postMessage("TaskApp.completeTask", "*");
```

This completes the task without saving any available process variables and document properties.

Saving document properties and process variables using a form

You can save document properties and process variables without completing a task. You can save without completing the task in two different ways:

1. By using an "Event" button
2. By using a custom action

Saving document properties and process variables with an "Event" button - This is how it works

This is how it works

1. Add a button to your form.
2. Configure the button as follows:
   - **Action: Event**
   - **Button Event: save**

When you click the button, any available process variables and document properties are saved. If no process variables or document properties are available, saving is skipped.

Saving document properties and process variables with an event with a custom action - This is how it works

1. Add a component to your form that supports the execution of a script (e.g. a button with the action **Custom**).
2. Call the following code in the script:

```
form.emit("customEvent", {"type":"save"});
```

If a user opens the task with the form and clicks the button, any available process variables and document properties are saved. If no process variables or document properties are available, saving is skipped.

Unlike when you save using an "Event" button, this variant also lets you execute additional custom code before saving.

You can use the type **saveDmsProperties** to save document properties without simultaneously saving process variables:

```
form.emit("customEvent", {"type":"saveDmsProperties"});
```

You can use the type **saveProcessVariables** to save process variables without simultaneously saving DMS object properties:

```
form.emit("customEvent", {"type":"saveProcessVariables"});
```

You can react to the result of the different saving events as follows:

```
form.on("saveSuccess", () => { console.log("Saved successfully") });
form.on("saveFailed", () => { console.error("Failed to save") });
```

1.6. Using scripts

You can use d.velop scripting to create and manage scripts. You can use scripts to enhance your environment with custom logic: For instance, you can integrate your created scripts into processes or webhooks (d.velop documents).

With the relevant authorization, you can create and edit scripts. Users that do not have this authorization can execute scripts.

Each created script has an embedding URL. You can use this URL to execute the respective scripts using a **GET**, **POST**, **PATCH**, **PUT** or **DELETE** request.

1.6.1. Getting started with working with scripts

In this topic, you can learn about the basic functions for working with scripts. You can learn how to create, open and manage scripts.

Calling a script

To execute a script, you need the URL of the script. You can obtain the script URL by choosing **Copy URL to clipboard** in the three-dot menu for the relevant script in the script list. You can now use this URL to execute the script.

You can call the script using the following HTTP methods:

- **GET**
- **POST**
- **PUT**
- **PATCH**
- **DELETE**

With the methods **GET** and **DELETE**, it is not possible to pass data in the request body to the script.

> **Note**
> A script can only be executed by a system user.

Creating a script

You can create a new script in script management. To do so, you must have the appropriate authorizations.

This is how it works

1. Click the button for adding a script (plus sign) in the toolbar.
2. Enter a name for the script and click **Create.** The editor area displays the functional framework of an executable "Hello, World" script that you can modify or expand.
3. Save your script.

Importing and exporting scripts

You can export and import scripts. This can be helpful, for example, if you want to transfer a script from one system to another.

This is how it works

1. Open the script overview in the source system by opening the **Process studio** feature on the start page, for example.
2. Open the three-dot menu next to the relevant script and choose **Download Script**. The script definition is now downloaded as a JSON file.
3. Open the script overview in the target system.
4. In the toolbar, click the import button ( ) and select your exported script.

> **Note**
> Encrypted variables are exported with an empty value. After the import, you may have to enter these variables. You can decide whether variables are also to be imported or not. If the variables are also to be imported, all already existing variables are overwritten by the import when a script is overwritten.

Managing scripts

You can find the **Scripting** area under **Tasks and processes** in the configuration. Choose **Script management** to go to the overview list of all the existing scripts. In the overview list, you can create new scripts and edit or delete existing ones.

> **Note**
> You can only view the **Script management** entry if you have been assigned with the **Administration** user role. Only your administrators can assign it to you.

1.6.2. Using external libraries in a script

Within a script, you can use external libraries to address third-party systems, for example.

This is how it works

1. To use an external library, go to the **External libraries** tab.
2. Click the plus sign to add a new external library. You can use packages provided by npmjs.com.
3. Enter a valid library name and click **Apply**. The entered library is validated. If you do not define an explicit version, the last version number is applied automatically.
4. Import the library to the script using the **require** function. The library can now be used in the script code.

> **Note**
> When you add the external library, it is downloaded from npmjs.com. For you to use this function, d.velop scripting must be able to reach the address npmjs.com on the internet.

1.6.3. Using variables

You can create variables and use them within a script. Variables can be useful, for example, if you do not want to enter an API key or passwords in the code.

This is how it works

1. Open the desired script.
2. Navigate to **Variables** and click **Add Variable**.
3. Enter a name for the variable under **Name**. You can use the name to reference the variable within the script.
4. Enter a value for the variable under **Value** . If you wish to save the value in encrypted form, activate **encrypted**.
5. Use the **Request** object to reference the variable you created.

1.6.4. Testing a created script

You can test your created scripts at any time to ensure that they are executable.

This is how it works

1. Select a script from the script list.
2. Click **Test** to display the test area.
3. Click **Execute** to call the relevant script. The script response and its associated status code are displayed.

1.6.5. Working with the "Request" object

If you create a script, you can use a **Request** object within the script. Using the **Request** object, you can reference variables you have previously created. To this end, use the **var(name)** and **variables()** functions.

Depending on the format, you can access the request body by means of **data()**, **json()** or **text()**.

Use the **param(name)** function to access request parameters.

If you want to scrape a specific header value, you can use the **get(name)** function for this.

Reference:

```
class Request {
    /**
     * @type {Object.<string, string>}
     */
    #headers;
    /**
     * @type {Buffer}
     */
    #body;
    /**
     * @type {Object.<string, string>}
     */
    #variables;

    constructor(method, path, params, headers, body, variables) {
```

```
        this.method = method;
        this.path = path;
        this.params = params;
        this.#headers = headers;
        this.#body = body;
        this.#variables = variables;
    }

    get(name) {
        if (!name) {
            return this.#headers;
        }
        return this.#headers[name.toLowerCase()];
    }

    param(name) {
        if (!name) {
            return this.params;
        }
        return this.params[name.toLowerCase()];
    }

    var(name) {
        return this.#variables[name]
    }

    variables() {
        return this.#variables;
    }

    data() {
        return this.#body;
    }

    end() {
        console.warn("req.end() is deprecated and will be removed in a
future version!")
    }

    json() {
        return JSON.parse(this.#body.toString('utf-8'));
    }

    text() {
        return this.#body.toString('utf-8');
    }

}
```

1.6.6. Working with the "Response" object

If you create a script, you can use a **Response** object within the script.

The **Response** object is the outcome of the script. You can submit the status code, the response body and any desired header values.

You can use the **status(code)** method to submit the HTTP status code.

If you want to issue your response in JSON format, use the **json(obj)** method. For other formats, you can use the **send(body)** method and insert the corresponding **content type** header by means of **set(key, value)**.

The **set(key,value)** method also makes it possible to submit any other desired header values. The **key** parameter is the header name, and **value** is the corresponding value.

The **Response** object looks as follows:

```
class Response {
    #header = {};
    #status = 0;
    #body = Buffer.alloc(0);

    status(code) {
        this.#status = code;
        return this;
    }

    json(obj) {
        if (Buffer.isBuffer(obj)) {
            this.set("Content-Type", "application/
json").send(JSON.stringify(obj)); // If a buffer gets here explicitly
            return;
        }
        this.set("Content-Type", "application/json").send(obj);
    }

    set(key, value) {
        if (value === undefined) {
            if (typeof key === 'object') {
                this.#header = key;
                return this;
            }
            throw new Error('when value is left out then key needs to be a
map of strings');
        }
        if (this.#header === undefined || this.#header === null) {
            this.#header = {};
        }
        this.#header[key] = value;
        return this;
    }

    send(body) {
        if (typeof body === 'string') {
            this.#body = Buffer.from(body, 'utf-8');
        } else if (Buffer.isBuffer(body)) {
            this.#body = body;
        } else if (typeof body === 'object') {
            this.#body = Buffer.from(JSON.stringify(body), 'utf-8');
        } else
            console.error("Body is not acceptable!", typeof body)
    }
```

```
    body() {
        return this.#body
    }

    header() {
        return this.#header
    }

    //Returns HTTP Status Code
    getStatus() {
        return this.#status
    }

    //Deprecated
    end() {
        console.warn("res.end() is deprecated and will be removed in a
future version!")
    }

}
```

Activating the offline mode of d.velop scripting for systems without Internet access

As of version 1.3.0 or 2023.Q4, d.velop scripting automatically detects if there is no Internet access on the server. If you use d.velop scripting in on-premises deployment and there is no internet access on the server, you can add an environment variable. The environment variable skips automatic detection of Internet access and switches d.velop scripting permanently offline.

This is how it works

1. Open the Windows **Run** dialog by pressing **Windows key + R**.
2. Enter **sysdm.cpl** and confirm with **OK**.
3. Navigate to **Advanced > Environment Variables**.
4. Under **System Variables**, create the variable **node.offline** with the value **true**.
5. Restart the d.3 process manager.

You have activated the offline mode of d.velop scripting. Note the following information regarding offline mode:

- When creating scripts, **npm install** is no longer executed.
- Libraries that were added via d.velop scripting are no longer taken into consideration. Manually copy any libraries that you require into the directory C:\d3\nodejs18\node_modules.
- d.velop scripting first tries to link the **node_modules** folder. If permissions for the link are missing, the folder is copied for execution. If the folder cannot be linked or copied, add the following code at the start of the script:

```
process.env.NODE_PATH = "C:\\d3\\nodejs18\\node_modules"
require("module").Module._initPaths();
```

1.7. Use of events related to documents

In d.velop process studio, you can deploy events that are related to documents. You can consume these events in order to start processes, for example. Further information is available in the user documentation for d.velop process studio

The following events are available:

- After the creation of a document or a dossier
- After the creation of a new document version
- After an update to document properties
- After the changing of a document status
- After the release of a document
- After the deletion of a document or a dossier
- After the creation of a generated document

The deployment of these results has to be configured by a member of the technical administration. A synchronization with d.velop documents is initially necessary so that the available data from the repository (such as categories) are available. If multiple repositories are configured, either the standard repository or (if no standard repository was configured) the first repository is used as the basis.

This is how it works

1. Select the feature **Configuration**.
2. In the **Events** section, select the **Configure repository** option.
3. Use the introductory description to make sure the correct repository was identified.
4. Click **Synchronize**. The required data from the repository is determined using the permissions of the logged in user. If the relevant permissions or data (such as categories or properties) have changed, repeat this process.
5. If you want the properties of the documents to be communicated to the consumers of the events, enable **Allow access to advanced properties**.
6. Under **Enter service user from d.velop documents**, enter (if applicable) the ID of the user who is represented by the API key which is entered in d.velop documents and in whose name webhooks are sent.
7. Enable all events from d.velop documents that should be offered for consumption.

1.8. Administration of processes

In this topic, you will find out how to administer your processes. Read about how to get an overview of your processes and how to correct broken processes.

1.8.1. Useful information on the "Process administration" feature

You can use the feature **Process administration** to start actions affecting entire process versions. Generally, the actions include several process instances. Process administration allows you to perform actions for different processes and versions in sequence without having to change the view.

You can perform the following actions for entire process versions:

- **Retry process versions**: If several errors occur in one process, you can retry the process version for troubleshooting. To retry a process version, use the action **Retry**.
- **Migrate process instances to the next process version**: A new version of the process is automatically created when processes are updated. You can use the action **Migrate** to migrate all instances of a process version to the newest version.
- **Delete process versions**: If there are no more running process instances for a version, you can delete the process version. To delete a process version, use the action **Delete process version**.
- **Cancel process instances**: You can cancel all open instances of a process version. By canceling, all open activities for this process version are removed and the process instance cannot be retried. To cancel a process instance, use the action **Cancel instances**.

The following actions are also available:

- **Deploy new process**: You can import a new process (or a new version of an existing process) from a **.bpmn** file.

1.8.2. Viewing all process instances

The feature **Process monitoring** can be used to obtain an overview of all process instances. You can use the business key to perform targeted searches of individual processes or limit the list of results using the filters **Processes**, **Versions**, **Activities** and **Status**. A process has one of the following four statuses:

The instance was canceled.

The instance is in an error state.

The instance was ended successfully.

The instance is currently still running.

If there is an additional exclamation mark for the status, this indicates an error or a warning during the process's protocol export.

Select the process to see more detailed information on a process instance. There are context actions available in the detail view which may differ depending on the process type. You can use the context actions to cancel the process instance, migrate it to the next version, view it in an instance diagram or show it in the source. You can also display the process protocol if you want to obtain an overview of the process sequence.

1.8.3. Viewing an instance diagram

In the detail view of an instance, you can view the process instance in a diagram with the context action **Instance diagram**.

To maintain an overview of very complex instance diagrams, you can change the diagram view. Use the mouse wheel to change the size of the diagram. When you have enlarged the view using the mouse wheel, you can move the diagram by holding the left mouse button. Use **Reset view** to return to the original view.

1.8.4. Changing the variables of a token

When you navigate to the detail view of a token in the feature **Process monitoring**, you can change the variables of the process instance. You can change variables to remove the potential error cause of a process, for example.

Select **Edit variables** to enable the edit mode and change the values in the input fields. There are two options for exiting the edit mode:

- Select **Save changes** to write the new values to the database and leave the edit mode. All changes have a direct impact on the entire process.
- Select **Discard changes** to restore the initial values and leave the edit mode. This action has no impact on the process.

1.8.5. Retrying a failed token

When you see a process with the status **Error** in the feature **Process monitoring**, you can retry the failed token for troubleshooting. Observe the error cause in the detail view of a token, as you may need to change the variables of the process instance for troubleshooting.

To retry a token, navigate to the detail view of the token. Select **Retry** and confirm the notice.

1.8.6. Re-executing an asynchronous service

When you use an asynchronous service, errors may occur during communication with the external service when you execute the service activity. The error can occur if the service endpoint accepts the service request but does not respond to it owing to technical problems. You can call the service again in process monitoring to rectify this error.

This is how it works

1. Open the process instance concerned and the service token for which there was no response in the feature **Process monitoring**.
2. Select **Move token**.
3. Select the target activity for the token and confirm the dialog.

1.8.7. Useful information on protocolling processes

If you want to be able to trace the runs of individual process instances, you can activate protocolling for the associated process version.

There are two options for displaying the protocol: In process monitoring, you can use the context action **Process protocol** to display the protocol. For tasks that are part of a protocolled process, you can display the protocolling in the perspective **Process** in the task overview.

Different views

There are two different views for the process protocols:

- The **professional view** focuses on user activities. In the professional view, the protocol contains information about the process start and the process end, user activities, persons involved, and data entered.
- All the activities of the process instance that have been run through are processed in the **technical view**. That includes all the items that exist in the BPMN definition, service calls, error situations that have occurred, and many other technical details. Technical views of protocols are available only in process monitoring.

Protocol storage

Protocols are available during the entire runtime of a process instance. The protocols can be stored for up to one year after the process instance is completed, depending on the configuration. The protocol data is deleted after the storage period has expired.

Exporting protocols to a d.3 repository

If you would like to store protocol data beyond the storage period, you can save the protocols in your d.3 repository. To save protocols in the repository, the first thing you need to do is configure the protocol export. After a process instance has been completed, the protocol is then automatically saved in your d.3 repository.

You can still display the protocols in process monitoring and in the task view after the export.

Special process variables

For user tasks, you can add additional information to the protocol in the form of a decision and a comment. To do this, you can use an editing dialog to transfer the process variables **dv_decision** and **dv_comment**.

The content is highlighted in a special manner in the protocol.

1.8.8. Enabling the logging of processes

When you deploy a process definition, you can activate protocolling. By default, protocols are stored for 30 days starting at the end of the process.

You must configure protocolling separately for each process.

This is how it works

1. Open the feature **Process administration**.
2. Select the action **Deploy new process**.
3. Select the BPMN file you want to deploy.
4. Click **Upload**.
5. Click **Protocol** to open the settings for the protocolling.
6. Activate the protocolling.
7. You also have the option of activating protocol exports to your d.3 repository, which allows you to use audit-compliant archiving.
8. Change the storage time of the protocols if necessary.
9. Confirm the dialog with **Deploy**.

If you deploy a new version of a process definition, the most recently used settings for protocolling the process version are applied. If you would like to change the settings later on, you need to deploy the process definition again.

1.8.9. Creating and configuring of process events

You can use process events to start processes through a JSON endpoint (e.g. via webhook). To do so, you first create a process event.

This is how it works

1. Open the feature **Process administration**.
2. Choose **Process events**.
3. Click the plus sign to create a new process event.
4. Enter a name for the process event to be created and click **Add**.
   The process event is created. You must now configure it.
5. In the **Process** selection field, select the process you want to start.
6. You can also define a **business key** or **correlation key** for the start of the process. For this purpose, you specify an expression in the appropriate field.
7. In the **Manage mappings** area, you can assign data from the JSON endpoint to process variables.
8. Click the plus sign to define a new mapping.
9. Enter a name for the mapping.
10. In the **Endpoint path** field, enter the expression whose result is written to the process variable at the start.
11. Select the **process variable**.
12. Click **Add** to carry out the mapping.
13. Repeat steps 8 to 12 for all the mappings you want to define. Note that you must assign all the mandatory variables for the process.
14. Save the process event.

In the overview of process events, you can determine the URL for triggering a process event in the three dots menu by clicking **Copy endpoint URL to clipboard**.

You can find additional information about calling the endpoint in the API documentation.

1.8.10. Useful information on process events

When creating process events, you can use variable expressions to map values from the JSON of the endpoint call. The expressions use the syntax of the Java Expression Language.

Expressions may only use the following items:

Item	Restriction
Read access to the properties from the JSON transmitted during the call: **input.getValue**	The string transferred to the method must correspond to a valid JSON path (see examples).
Generating multi-values: **collections.from**	
Operators and characters	Permitted operators and characters: +, -, *, /, %, =, !, <, >, &, |, ?, (,), [,].
Keyword	Permitted keywords: div, mod, eq, ne, lt, gt, le, ge, and, or, not, empty.
Numeric values	
Strings	Strings must be put between " " or ' '.

Examples

JSON transmitted during the endpoint call


```
{
   "doc":{
      "caption":"Process description - 2021-11-17",
      "categoryId":"dv.fol.basis.Correspondence",
      "dateCreated":"2021-11-17T07:39:36.502+00:00",
      "dateLastAccess":"2021-11-17T07:39:36.502+00:00",
      "dateUpdated":"2021-11-17T07:39:36.502+00:00",
      "dateUpdatedFile":"2021-11-17T07:39:36.502+00:00",
      "fileExtension":"PDF",
      "fileName":"correspondence",
      "id":"KR00000227",
      "number":"KR00000227",
      "size":51500,
      "status":"F",
      "versions":[
         {
            "id":0,
            "status":"F",
            "dateCreated":"2021-11-17T07:39:37.000+00:00",
            "fileName":"KR00000227.1",
            "fileExtension":"PDF",
            "size":51500
         }
      ],
      "properties":[
         {
            "id":"dv.fol.basis.Subject",
            "name":"Subject",
            "index":2,
            "dataType":"STRING",
            "isMultiValue":false,
            "value":"Process description"
         },
         {
            "id":"dv.fol.basis.Date",
            "name":"Date",
            "index":50,
            "dataType":"DATE",
            "isMultiValue":false,
            "value":"2021-11-17"
         }
      ]
   },
```

```
   "fileDestination":"somwhere",
   "importOk":"1",
   "user":{
      "id":"A64C2BD6-8EF5-4C6D-B1BA-7F632A2AC3ED",
      "name":"Jane Doe",
      "d3Id":"jdoe"
   },
   "docType":{
      "id":"dv.fol.basis.Correspondence",
      "name":"Correspondence",
      "type":"DOCUMENT_TYPE"
   }
}
```

Examples

```
// Hello World
${"Hello World"}

// 3
${3}

// false
${3 > 4}

// Value of doc.status in JSON
${input.getValue("$.doc.status")}

// Value of a defined element in the doc properties in JSON.
// If an element is found, an array with one element is returned. Can NOT
be used in combination with string concatenation.
${input.getValue("$.doc.properties[?(@.id ==
\'dv.fol.basis.Subject\')].value")}

// Value of a defined element in the doc properties in JSON.
// If an element is found, its value is returned and may be used within
string concatenation. With this expression you have to ensure, that an
element exists.
${input.getValue("$.doc.properties[?(@.id ==
\'dv.fol.basis.Subject\')].value")[0]}

// creates a collection with "a", "b" and "c"
${collections.from("a", "b", "c")}
```

Hybrid forms are also possible, making it possible to directly combine texts and variable content.

```
dmsObject:///dms/r/yourRepositoryId/o2/${input.getValue("$.doc.id")}
```

1.9. Tip of the Day

Find out more about how to make your work easier and get useful tips regarding various functions.

1.9.1. Backing up the key for encrypted variables in scripts

You encrypt and decrypt all variables in scripts with a single key. The key appears after the first start of d.velop scripting in the working directory in the **top.secret** file. Without the **top.secret** file, you can no longer decrypt encrypted variables in existing scripts. Please note the following recommendations:

- Back up the **top.secret** file separately from the existing backup functions.
- Only give the executing user (user who starts d.velop process manager) read permissions for the **top.secret** file.

1.9.2. Calling up an interface with your own certification authority in a script

Many on-premises environments have their own certification authority (CA) for signing server certificates. Often, these CAs are not signed by a trustworthy party, with the result that the NodeJS execution environment does not trust these certificates. This leads to no REST queries being able to be sent towards these systems from the scripting app.

For the execution of REST queries, you must import the public part of the CA certificate in PEM format. Since the certificates are often already saved by Windows in the certificate store, you can export the public part of the certificate from the certificate store.

> **Note**
>
> The certificate must be in PEM format. Export the certificate with the format “Base-64 encoded X509” or convert the certificate by openSSL. You can then use the file or the content of the file as a string in the script.

You have two options to include self-signed certificates:

Option 1

You can globally integrate your own certificates in the NodeJS execution environment. Use the system environment variable **NODE_EXTRA_CA_CERTS=<path to PEM file>** and restart the d.velop process manager.

Option 2

Add the dependency **undici** to your script. You can then load the certificate from the hard disk. Alternatively, you can enter the certificate string in the script and add it to the **Agent** object as the CA:

```
//Load undici agent and get dispatch function setter
const { Agent, setGlobalDispatcher } = require('undici')
module.exports = async (req, res) => {
        //Read cert file from filesystem
    const fs = require('fs');
    const certString = fs.readFileSync("C:\\certs\\test.pem", 'utf8')
        //Create a new agent using the loaded cert string as ca
    const agent = new Agent({
    connect: {
        ca:  certString
     }
    })
        //Set global dispatcher function
    setGlobalDispatcher(agent)
        //All regular fetch calls will now use the new agent
    const response = await fetch("https://my.secured.host/scripting", {
        // Adding method type
        method: "GET",
        // Adding headers to the request
        headers: {
            "Accept": "application/json"
        },
    })
    if(!response.ok) {
```

```
        console.log("Response was not ok")
    }
    const result = await response.json();
    console.log("Response from fetch", result)
    res.status(200).set("Content-Type", "text/plain").send(result);
}
```

1.9.3. Useful information on creating scripts

When developing JavaScript scripts, it is important to use tried-and-tested methods in order to make the code efficient and faultless.

In this article, you will find information on the use of the **async/await** method for certain processes. Additionally, we explain why you should use the **fetch** method instead of **axios** for requests.

Simplified handling of processes with the "async/await" method

When coding with JavaScript, many tasks can be performed concurrently. Although there are parallel processes, the overall process is not hindered. Use of the **async/await** method is very helpful in controlling parallel processes.

Tips for using the "async/await" method:

- **Using "async" functions:** Use the **async** keyword to flag functions that execute asynchronous actions. The code becomes more legible if you use the **await** method in these functions.
- **Handle errors with "try/catch" functions**: For an ideal error handling, you should insert**await** instructions into **try** and **catch** functions. In this way, you can identify errors and react to the accordingly.

Example

```
module.exports = async (req, res) => {

    try {
        let resp = await fetch("/identityprovider/scim/user");
        if (resp.ok) {
            console.log(await resp.json());
        }
    } catch (e) {
        console.error(e)
    }

}
```

Use of the "fetch" method for requests instead of "axios"

If you want to retrieve data from the Internet, you require an option for sending requests. The **axios** method is widespread but the **fetch** method reduces the number of dependencies required. The **fetch** method is already included in the application.

Tips for using the "fetch" method:

- **Similar use as in the browser:** The use of the **fetch** method is similar to how it is done in the browser. You can use the method to access data from the Internet.
- **Reacting to responses**: If you use the **fetch** method, you always receive a response. You can combine the methods **await** and **fetch** in order to access the response.

1.10. Frequently asked questions

In this topic, you will find answers to frequently asked questions.

1.10.1. Why does an unknown server error occur when a variable is inserted?

If you want to insert a process variable which already exists with different capitalization, an error may occur. The error occurs if the sorting of the database does not take into account capitalization and therefore cannot distinguish between the two variables.

To prevent the error, please use the capitalization already used when inserting a variable.

1.10.2. Why is the loading indicator displayed for so long when the database configuration is saved?

When the database configuration is saved, it may occur that the loading indicator is displayed for several minutes. If this is the case, you maybe have not entered the data correctly during the configuration. You need to delete the configuration data for the database connection manually and reenter it.

This is how it works:

1. Exit the application via the Windows service manager.
2. Delete the file **process-app-db.properties** in the **conf** directory under the default installation directory.
3. Start the application via the Windows service manager.
4. Configure the database connection in the feature **Configuration**.
5. Restart the application.

1.10.3. How do I update the JDBC driver of the database?

When an update of the database used is available, you may need to update the JDBC driver.

This is how it works

1. End the service of the application.
2. Open the **lib** folder in the default installation directory of the application.
3. Delete the file **pa-jdbc-***.jar** (the asterisks are placeholders for the extension specific to the database management system)
4. Restart the service of the application.
5. In the feature **Configuration**, open the entry **Database** under **Process settings**.
6. Click **Upload driver** and select the JAR file.
7. Restart the application.

1.10.4. Where can I find a summary of all the administrative actions?

When you switch to the perspective **Action summary** in the **Process administration** feature, you get an overview of all ongoing, started, completed and failed actions. You can use the trash icon to remove completed and failed actions from the log manually.

1.10.5. Where do I configure the maximum memory size for a process instance?

You can use the file **process-custom.vmoptions** to configure the maximum memory size. The file **process-custom.vmoptions** is referenced from the file **process.vmoptions** referenziert. Unlike the file **process.vmoptions**, the new file **process-custom.vmoptions** remains unchanged during updates and your changes are retained.

If you specify an identical parameter in both files, the content of the file **process-custom.vmoptions** is given priority because this file is loaded later.

1.11. Additional information sources and imprint

If you want to deepen your knowledge of d.velop software, visit the d.velop academy digital learning platform at https://dvelopacademy.keelearning.de/.

Our E-learning modules let you develop a more in-depth knowledge and specialist expertise at your own speed. A huge number of E-learning modules are free for you to access without registering beforehand.

Visit our Knowledge Base on the d.velop service portal. In the Knowledge Base, you can find all our latest solutions, answers to frequently asked questions and how-to topics for specific tasks. You can find the Knowledge Base at the following address: https://kb.d-velop.de/

Find the central imprint at https://www.d-velop.com/imprint.