Workflow

workflow

We are using Activiti BPM Platform workflow engine. It is currently very widespread workflow engine, which is used for its ease of use and performance. It's main purpose is control and execution of process written by BPMN 2.0 language in Java.

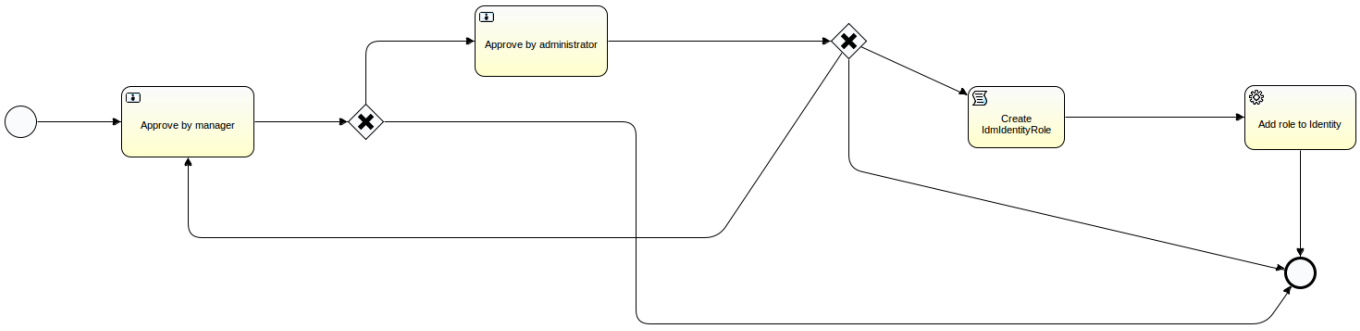
Its advantage lies in integration with Spring framework and it is easy to use in ours devstack. Activiti Platform sets up Rest Api and like that it will be using most of it's functionalities. By running Rest and Activiti Platform on a stand-alone application server, we can communicate even with a non-Java environment. Disadvantage of this solution is lossage of direct Java integration. It is possible to call Spring beans directly from workflow process (by Expression Language), which extends functions workflow and accelerating development.

- **/api/workflow/definitions/** (Workflow definitions)
- **/api/workflow/tasks/** (Instance of user's tasks)
- **/api/workflow/processes/** (Instance of workflow processes)

Design of process

Design of workflow process is realized with XML in format BPM 2.0. It is very helpful to use Activiti Designer. It is plugin in IDE Eclipse. Activiti Designer is user-friendly and allows creating a process without user intervention into XML code. In order to be possible to install the Activiti Designer into Eclipse, add its repository address `https://www.activiti.org/designer/update/` in the Eclipse Install New Software section. A manual could be found here.

Sample of Activiti process in IDE Eclipse plugin:



Example of usage Expression Language

In our example of EL (lower), there is an option to assembly description of user's task. There is element "documentation" in the middle of description in form **${defaultIdmRoleService.get(roleIdentifier).name}**. **${...}** is the definition of the expression,

defaultIdmRoleService is the name of implementation service (Spring bean). Of this service method **get** is called with parameter id of requested role from process variable **roleIdentifier**. Method returns an instance of the requested role and we used attribute name. The result of this expression is a name of role.

Because of this procedure, we need in process attributes identifiers. With identifiers and EL expressions, we can get any necessary data.

```
<userTask id="managerTask1" name="Schválení vedoucím"
    activiti:candidateUsers="admin,user1">
  <documentation>Schválení přiřazení role
    ${defaultIdmRoleService.get(roleIdentifier).name} vedoucím pro uživatele
    ${defaultIdmIdentityService.get(identityIdentifier).getUsername()}.
  </documentation>
</userTask>
```

User task

Workflow activity **UserTask** can assign a task to a specific user by attribute **assignee**. The task can have just one assignee. If we want to assign more users, who can solve the task, we use attribute **candidateUsers**.

In this step there are no differences between assignee and candidateUsers. If a user is assignee or is in a list of **candidateUsers**, the user can solve this task.

Service task

Service task is activity run automatically by the system. Code, which will be executed, can be defined:

- In Java class, which path is defined in attribute **class**.
- In Java class implementing JavaDelegate.
- **By Experssion Language**.

In our example of EL (lower) is definition of service activity, which run expression **#{defaultIdmIdentityService.addRole(newIdmIdentityRole, false)}**.

This expression calls method **addRole** for adding a new role. The new role will be passed to **newIdmIdentityRole** process, which contains new link DTO between role and user. Of service **defaultIdmIdentityService** method **addRole** is called from Spring context.

```
<serviceTask
 id="addRole"
 name="Add role to Identity"
activiti:expression="#{defaultIdmIdentityService.addRole(newIdmIdentityRole,
false)}">
</serviceTask>
```

Script task

Script task activity allows run scripts defined in the workflow. Workflow supports JavaScript and Groovy. We will be using Groovy for it's performance and specific implementation version (in pom.xml). JavaScript engine is part of Java and implementation version will depend on the version on which version whole application will run.

The script is used, when EL cannot make expression we want. For example of iteration or creation new instance of Object and setting attributes. In our example (lower) we are creating new instance of link object between role and user and object is filled by identificators. This instance is then put in process variables with key **newIdmIdentityRole** execution.setVariable("newIdmIdentityRole", ir)).

```
<scriptTask
 id="scripttaskCreateIdentityRole"
 name="Create IdmIdentityRole"
 scriptFormat="groovy"
 activiti:autoStoreVariables="false">
      <script>import eu.bcvsolutions.idm.core.model.dto.IdmIdentityRoleDto;
         IdmIdentityRoleDto ir = new IdmIdentityRoleDto();
      ir.setIdentity(identityIdentifier);
      ir.setRole(roleIdentifier);
      ir.setValidFrom(validFrom);
      ir.setValidTill(validTill);
         execution.setVariable("newIdmIdentityRole", ir);
      </script>
 </scriptTask>
```

Email Task

Email task activity sends emails. Our emailer (EmailService) is injected into workflow engine, so it is possible disable sending notification (e.g. for debuging). Email can be send to specific identity by putting in username instead of email address.

```
    <serviceTask id="mailtask1" name="email taskfs" activiti:type="mail">
      <extensionElements>
        <activiti:field name="to">
          <activiti:string><![CDATA[tomiska]]></activiti:string>
        </activiti:field>
        <activiti:field name="subject">
          <activiti:string><![CDATA[Test emai3]]></activiti:string>
        </activiti:field>
        <activiti:field name="html">
          <activiti:string><![CDATA[wf_test]]></activiti:string>
        </activiti:field>
        <activiti:field name="text">
          <activiti:string><![CDATA[wf_test]]></activiti:string>
        </activiti:field>
        <activiti:field name="from">
```

```
          <activiti:string><![CDATA[svanda]]></activiti:string>
        </activiti:field>
      </extensionElements>
    </serviceTask>
```

Workflow localization

localization

We support localization of workflow process. The main principle is, the process definition (XML) does not contain translations. Translations contain frontend part of our application in the standard localization files. Workflow definition only contains information on how to construct the localization key and variables needs for translation.

At this point, it is possible to translate these parts of the workflow process:

- **Workflow process name**
- **Task name**
- **Task description**

How translate the name of a workflow process

For example, we will be using basic process for change the user permission 'approve-identity-change-permissions'. This process contains activity with name 'Generating process name'. This activity generates name of a process with included the name of the applicant.

Without translation looks code this:

```
 Change permissions request for
${identityService.getNiceLabel(identityService.get(applicantIdentifier,
null))}
```

For get the name of the applicant it's calls the identity service and user's full name is returned. We want to use this result in our translation. For this, we have to wrap (uses double braces) name of the applicant.

With wrapped variable looks code this:

```
 Change permissions request for
{{${identityService.getNiceLabel(identityService.get(applicantIdentifier,
null))}}}
```

You can use more variables in one translation. Name of the variables are order number in the source text.

Now we have to create translation in the frontend. In the localization file (usually in the /src/locales/) 'en.json', we have to creates the translation for key '**wf.approve-identity-change-**

permissions.name'.

```
 "wf": {
    "approve-identity-change-permissions" : {
      "name": "Change of permissions for user '{{0}}'"
    }
  }
```

Where 'approve-identity-change-permissions' is the **ID** of the workflow process definition and 'name' is always **name** of process (persisted in the variable 'processInstanceName'). We are using the variable with the name '0', that is our name of the applicant.

Translation with the context

Sometimes there is a need for multiple variants of translations to one key. To do this, you can use a variable defining context. The context defines the end part of the localization key (in our example is context 'secondVariant').

```
 Change permissions request for
{{${identityService.getNiceLabel(identityService.get(applicantIdentifier,
null))}}}{{context_secondVariant}}
```

```
 "wf": {
    "approve-identity-change-permissions" : {
      "name": "Change of permissions for user '{{0}}'",
      "name_secondVariant": "Second variant for change of permissions for
user '{{0}}'"
    }
  }
```

How translate the name and description of a workflow task

For example, we will be using again basic process for change the user permission 'approve-identity-change-permissions'. This process contains activity with the name 'Approve by help desk' and ID 'approveByHelpDesk'. This activity creates the approval task for helpdesk department.

We creates translation in the frontend for this task. In the localization file (usually in the /src/locales/) 'en.json', we have to adds the translation for key '**wf.approve-identity-change-permissions.task.approveByHelpDesk.name**'.

```
 "wf": {
    "approve-identity-change-permissions" : {
      "name": "Change of permissions for user '{{0}}'",
      "task": {
        "approveByHelpDesk": {
          "name": "Approval by 'Helpdesk'"
        }
      }
```

```
    }
  }
```

Where 'approve-identity-change-permissions' is the **ID** of the workflow process definition and under 'task' element is 'approveByHelpDesk', that is **ID** of process task.

Usually we want translate documentation for process task. In our example contains documentation of the 'approveByHelpDesk' task this code:

```
${processInstanceName}
```

It means, documentation contains the name of a process with wrapped variable contains the name of the applicant. We will uses this variable in translation on the frontend. In the localization file (usually in the /src/locales/) 'en.json', we have to adds the translation for key '**wf.approve-identity-change-permissions.task.approveByHelpDesk.description**'.

```
 "wf": {
    "approve-identity-change-permissions" : {
      "name": "Change of permissions for user '{{0}}'",
      "task": {
        "approveByHelpDesk": {
          "name": "Approval by 'Helpdesk'",
          "description":"Approving a change of permissions for '{{0}}'
(Helpdesk)"
        }
      }
    }
  }
```

Translation of description contains variable with the name of applicant again.

If none transaltion for task in the specific workflow is found, then we try find translation without workflow ID (for example '**wf.task.approveByHelpDesk.description**').

Passage of workflow - solver decision

The main principle of the workflow process is a creation of user task. An assigned user must decide how this task will be solved. The development of process is dependent on assign user's decision. We can explain further in an example of process assign role. A user will make a request to assign himself a specific role. A task is created and will be assigned to superior of the requester. Superior will decide if he agrees to assign a role to the requester or not. This **decision** will be **assessed**. In case of rejection, task will be terminated, otherwise, process will continue with next round of decisions.

It is important to properly define process, so solver will clearly know, of which possibilities he has to choose. And this has to be determined in each user's task. After termination of user's task, evaluation of decision have to be implemented.

Definition of decisions

Definition of decisions in user tasks have to be more complex than text represented of code enumeration "approve/disapprove". Options of decisions is automatically generated as buttons on detail of task. Each button represent one decision.

In workflow we have to know how to define decision as complex type, which contains all attributes for generation these decision buttons.

Decision attributes:

- **id** - Decision's identificator ("aprove").
- **label** - Description of decision for user ("Approve", or "core.content.task.decisions.approve").
- **level** - It sets design of button ("success" button will be green).
- **tooltip** - Description in detail ("By approving you will agree with terms and conditions").
- **showWarning** - Defines if warning message will be shown on end of terminating task (true/false).
- **warningMessage** - Defines contend of showWarning message ("Do you really want to approve request?" or "core.content.role.task.approve.warning").
- **permissions** - Defines required permissions, which user has to have to be able make decision.
- **skipValidation** - If you not used this attribute the behavioral will same as if you set it to false. When you set it to true then the validation of form(userTask) will be ignored.
- **reasonRequired** - This property set to true enforces decision reason to be provided.

From properties

We created our decision system, because Aktiviti does not have decision system. On the other hand, Aktiviti allows define **form properties**. This system allows set for each user's activity, which process attributes will be shown.

In our example (lower) there are two **properties** defined in user task. The first property has id "roleName". It means this task instance gets value of process attribute "roleName". It is also defined if this attribute is readable (**readable** = true), it is not allowed to change (**writable** = false) and it is not required to fill (**required** = false). The second property with id "description" is defined as readable attribute, which can be changed and it is required to be filled.

```
    <userTask id="managerTask1" name="Schválení vedoucím"
activiti:candidateUsers="admin,tomiska">
      <documentation>Schválení přiřazení role
${defaultIdmRoleService.get(roleIdentifier).name} vedoucím pro uživatele
        ${defaultIdmIdentityService.get(identityIdentifier).getUsername()}.
      </documentation>
      <extensionElements>
        <activiti:formProperty
          id="roleName"
          name="Název schvalované role"
          type="string" readable="true"
          writable="false"
          required="false">
        </activiti:formProperty>
```

```
        <activiti:formProperty
          id="description"
          name="Poznámka žadatele"
          type="string"
          readable="true"
          writable="true"
          required="true">
      </extensionElements>
    </userTask>
```

Decision

In previous article we described how **form properties** works. We do not want return complex definition of decison as string, so we need new data typ. Our data type is registered by Spring configuration in Aktiviti engine as **DecisionFormType** with key **decision**.

Decision values are defined by workflow definition as JSON objects and these values are written at creation of task (**approve**, **disapprove**, **backToManager**).

```
 <dataObject id="approve" name="approve" itemSubjectRef="xsd:string">
      <extensionElements>
        <activiti:value>{"label": "Schválit","showWarning":false,
"warningMessage":"Opravdu chcete úkol schválit?",
          "level":"success","tooltip":"Schválit úkol a předat na
administrátora"}
        </activiti:value>
      </extensionElements>
    </dataObject>
    <dataObject id="disapprove" name="disapprove"
itemSubjectRef="xsd:string">
      <extensionElements>
        <activiti:value>{"label": "Zamítnout","showWarning":true,
"warningMessage":"Opravdu chcete žádost zamítnout?",
          "level":"danger","tooltip":"Zamítnout úkolu"}
        </activiti:value>
      </extensionElements>
    </dataObject>
    <dataObject id="backToManager" name="backToManager"
itemSubjectRef="xsd:string">
      <extensionElements>
        <activiti:value>{"label": "Vrátit vedoucímu","showWarning":true,
"warningMessage":"Opravdu chcete žádost vrátit
          vedoucímu?", "level":"warning","tooltip":"Vrátit úkol vedoucímu"}
        </activiti:value>
      </extensionElements>
    </dataObject>
```

Access to these attributes is realized by form properties (type = decision). In our example (lower) there is user's task, which has form properties **approve** and **disapprove**. In this activity user will

have two decision options (buttons).

```
  <userTask id="managerTask1" name="Schválení vedoucím"
activiti:candidateUsers="admin,tomiska">
      <extensionElements>
        <activiti:formProperty id="disapprove"
type="decision"></activiti:formProperty>
        <activiti:formProperty id="approve"
type="decision"></activiti:formProperty>
      </extensionElements>
    </userTask>
```

Evaluation of decision

We already described how to define decision for user's activity, Now we will evaluate decisions in process.

Before closing user's task, Aktiviti engine is called with form properties values and decision's id (decision which user made). Decision's id will be put as process attribute with **decision** as name.

In next step component **exclusiveGateway** (XOR) is used. This component evaluate conditions on each outgoing path, if evaluation of path is **true**, process will continue on this path. If all outgoing paths are evaluated as **false**, exception is thrown.

In our example (lower) is defined two outgoing paths - **sequenceFlow**. The first path has condition **${decision.equals("disapprove")}**. It means, if **decision** was "approve", then process will continue on this path. The second path is similiar, but has contidion if decision was "disapprove".

```
    <exclusiveGateway id="exclusivegateway2" name="Exclusive
Gateway"></exclusiveGateway>
    <sequenceFlow id="flow7" sourceRef="exclusivegateway2"
targetRef="scripttaskCreateIdentityRole">
      <conditionExpression
xsi:type="tFormalExpression"><![CDATA[${decision.equals("approve")}]]></cond
itionExpression>
    </sequenceFlow>
    <sequenceFlow id="flow8" sourceRef="exclusivegateway2"
targetRef="endevent">
      <conditionExpression
xsi:type="tFormalExpression"><![CDATA[${decision.equals("disapprove")}]]></c
onditionExpression>
    </sequenceFlow>
```

Dynamic detail of task

We described **Form properties** as process attributes, which will be shown in user's task and we defined **decisions** as buttons for controlling the flow (paths) in process. Form properties can be used for dynamic design for each task.

Principle is similiar to **decision**. In process is definition, which components will be used, and on the frontend side is by that definition generated detail by using default frontend components. Aktiviti engine by default provide a few components (**string**, **enum**, **date**), which can be used for task definition.

But these components are insufficient, it cannot define if string will be shown as one-line field or text editor. Because of this we created our special data type, which can be used in user's task:

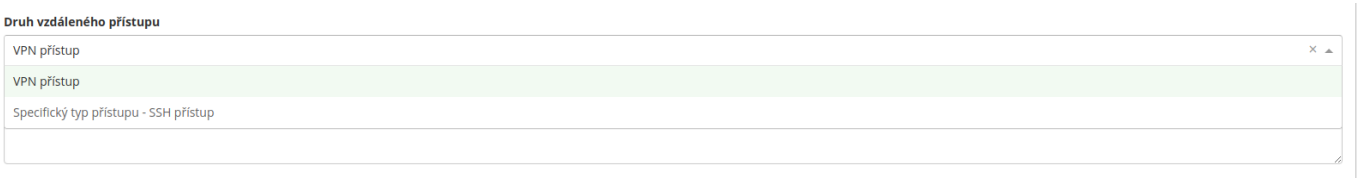
- **textArea** - It will be shown as one-line field (as additional options, key "placeholder" can be used).
- **textField** - It will be shown as multiline editor (as additional options, key "placeholder" can be used).
- **date** - Shows as date component (conversion to "yyyy-MM-dd").
- **checkbox** - Shows as checkbox.
- **selectBox** - Show as selectbox. It is using EnumSelectBox component.
- **taskHistory** - It can show previous tasks from WF in table.

Each component has additional settings **From property**:

- **id** - Unique component identifier in single process. If attribute **variables** is not filled, then id even define process attribute.
- **name** - Name, which will be used as description of component (label). It is possible to use final text or localization key.
- **type** - Type of component. See above (textArea, textField, etc.).
- **variable** - Defines process attribute, which can be used in component (value can be read, but also result can be written).
- **expression** - Expression (EL) can be defined here. Expression will be executed at start of the task and result will be used as value of component.
- **readable** - Defines, if component will be shown (default value is **true**).
- **writable** - Defines, if component could be edited (default value is **true**).
- **required** - Defines, if value will be mandatory (default value is **false**).

Component definitions order in **Form properties** are the same as order components after frontend generating.

Example of user's task selectBox:



Data which are displayed in selectBox can be set via Map<String, String> where value is the text which will be displayed in GUI. You can set localization key into value. You can set options directly in WF without service on BE. Just use field Default in Form property configuration {"option1":"label for 1","option2":"label for 2"} or see XML example lower

Example of user's task taskHistory:

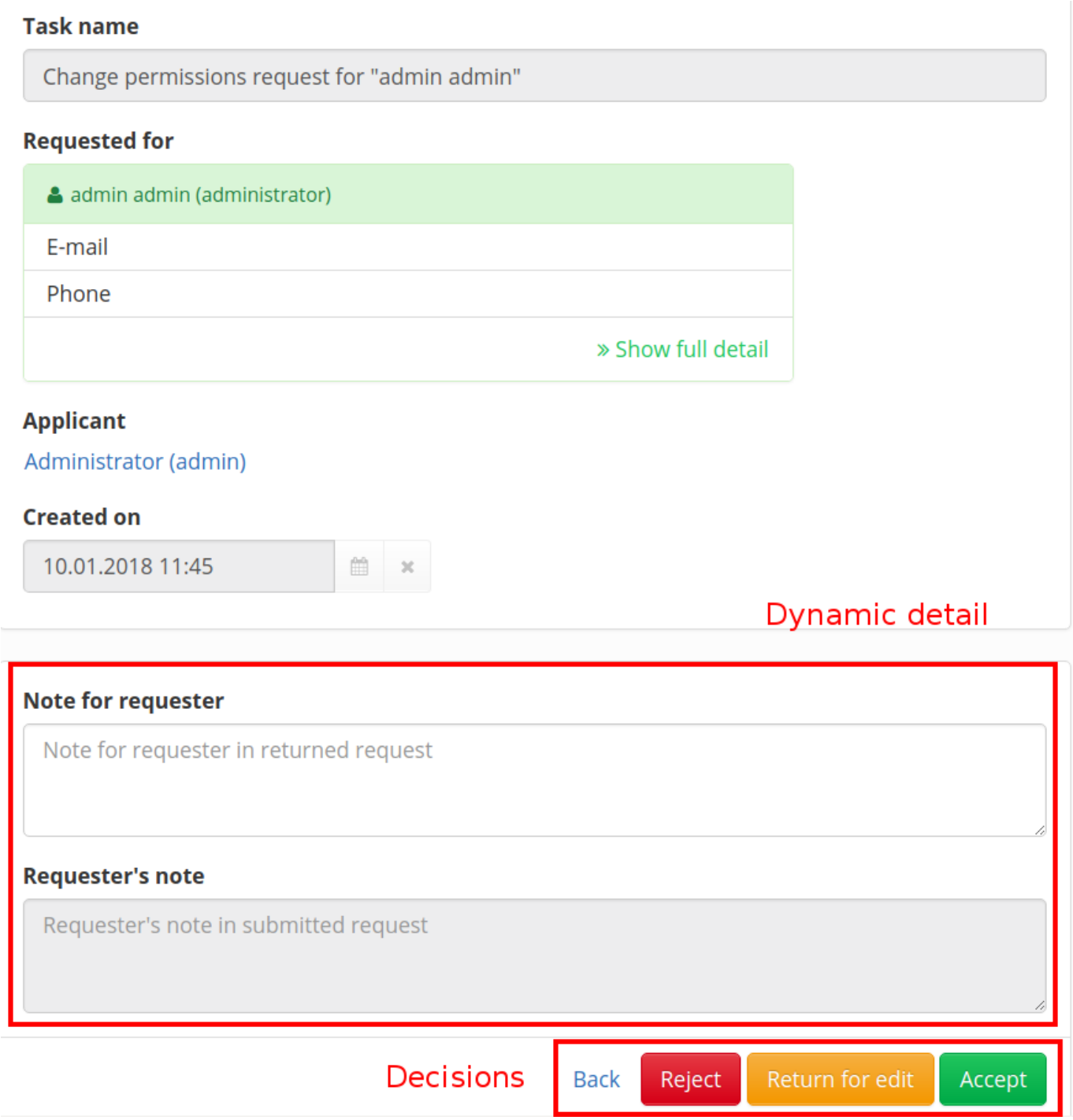
Předchozí úkoly

Název	Datum schválení	Schvalovatel	Poznámka řešitele
Zadost o pristup pro Roman Kucera (admin)	6.11.2018	Roman Kucera (admin) (1)	
Validace správnosti vyplněných údajů - krok 1. z 6: pro uzivatele Roman Kucera (admin)	6.11.2018	[illegible]	udaje jsou spravne

Data which are displayed in table can be set via List<WorkflowHistoricTaskInstanceDto>

In our user's task example (lower) there are generated frontend result and definition in XML.

Example of user's task with dynamic detail:



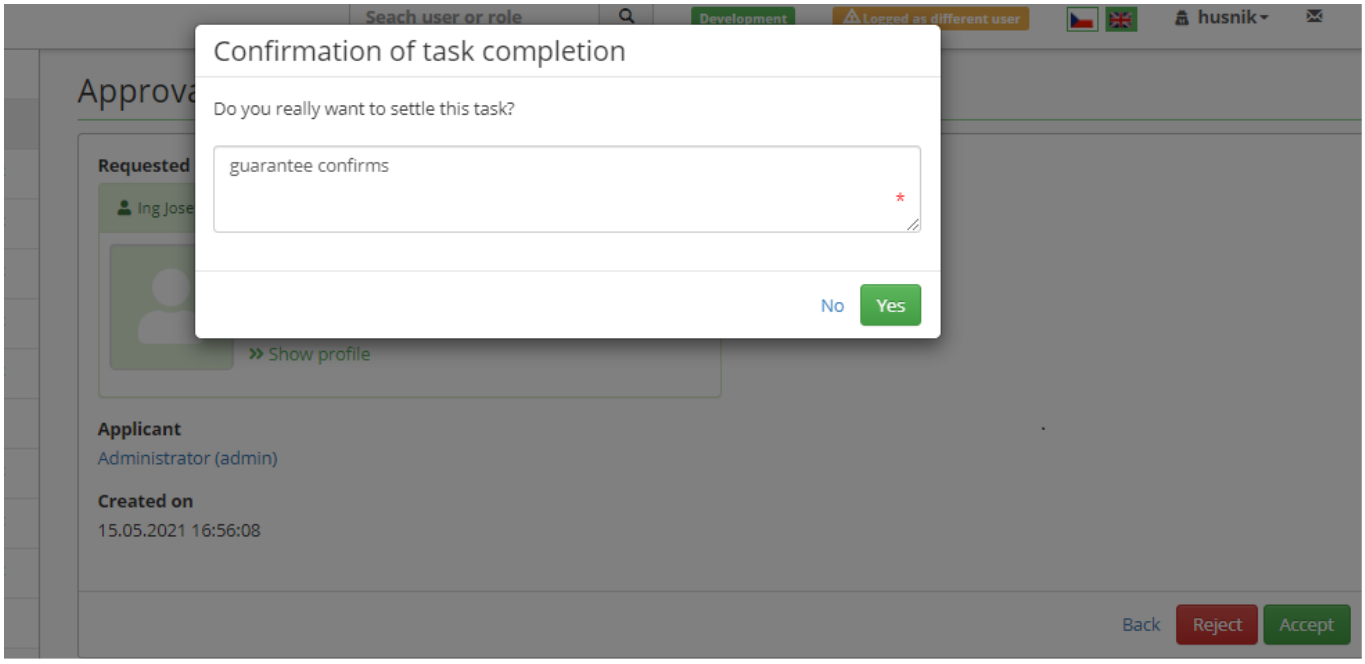
Example of user's task definition with dynamic detail:

```
  <userTask id="approveTask" name="Schválení uživatelem s rolí
&quot;SuperAdminRole&quot;"
activiti:candidateUsers="#{defaultIdmIdentityService.findAllByRole(defaultId
mRoleService.getByName(&quot;superAdminRole&quot;).getId())}">
      <documentation>Schválení přiřazení role
${defaultIdmRoleService.get(roleIdentifier).name} pro uživatele
${applicantUsername}.</documentation>
      <extensionElements>
        <activiti:formProperty id="disapprove"
type="decision"></activiti:formProperty>
        <activiti:formProperty id="approve"
```

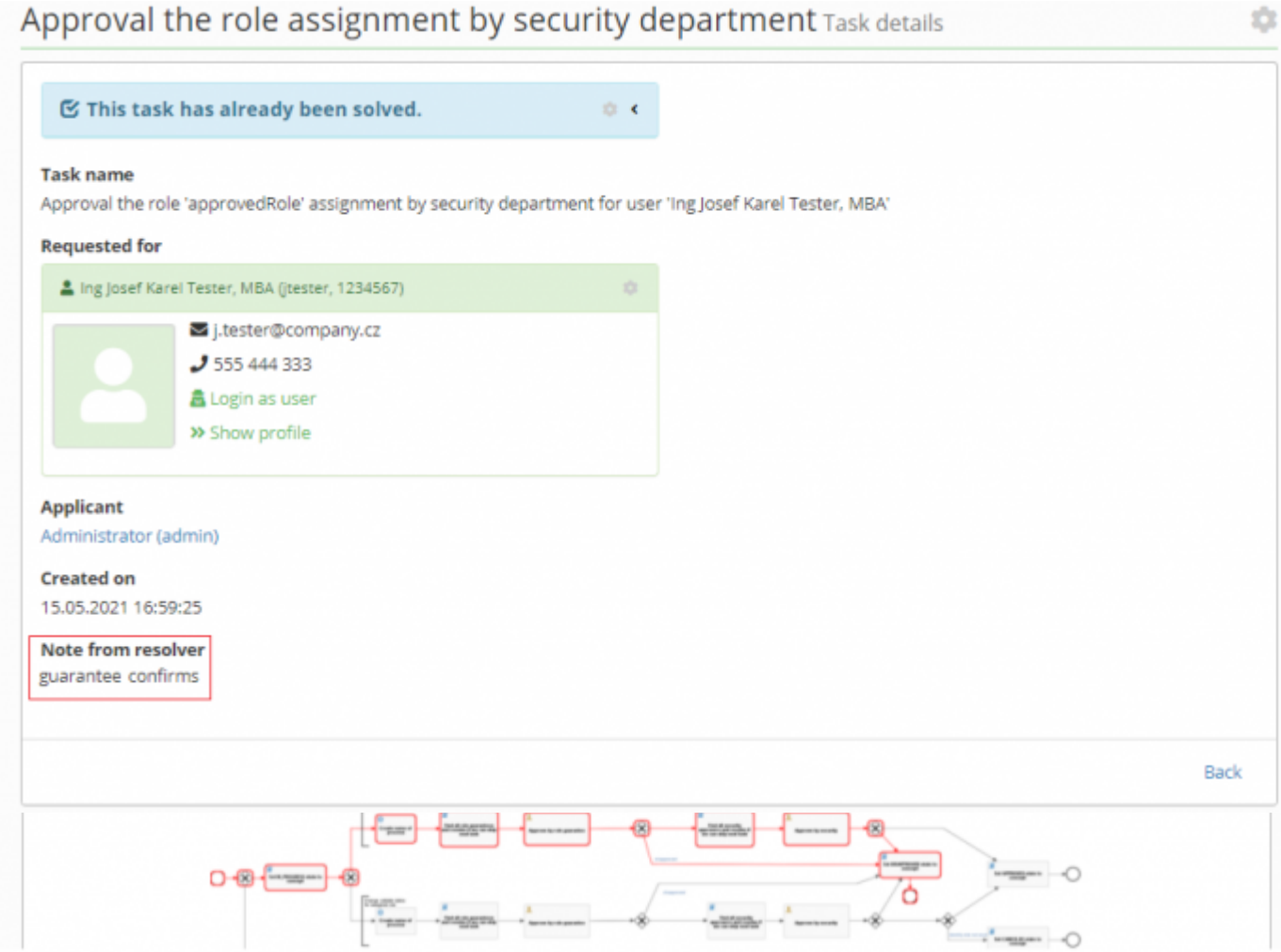
```
type="decision"></activiti:formProperty>
        <activiti:formProperty id="roleName" name="Název schvalované role"
type="textField"
expression="${defaultIdmRoleService.get(roleIdentifier).name}"
writable="false"></activiti:formProperty>
        <activiti:formProperty id="description" name="Poznámka"
type="textArea" writable="false">
          <activiti:value id="placeholder" name="Poznámka"></activiti:value>
        </activiti:formProperty>
        <activiti:formProperty id="validFrom" name="Platnost role od"
type="date"></activiti:formProperty>
        <activiti:formProperty id="validTill" name="Platnost role do"
type="date">
          <activiti:value id="tooltip" name="Platnost přiřazení
role"></activiti:value>
        </activiti:formProperty>
        <activiti:formProperty id="options" type="selectBox"
default="{&quot;option1&quot;:&quot;label for
1&quot;,&quot;option2&quot;:&quot;label for
2&quot;}"></activiti:formProperty>
        <activiti:formProperty id="history"
type="taskHistory"></activiti:formProperty>
      </extensionElements>
    </userTask>
```

Setting solver note message - decision reason

The task solver is able to set a reason of the decision or any other note to the solved task. The text area serving to this purpose is part of the confirmation modal window which appears, when user clicks the decision button. Display of this modal window (see in the picture below) has to be enabled by `decision` property `showWarning` or `reasonRequired` in particular workflow. When `reasonRequired` is used the solver note/decision reason is **mandatory**. More about setting of this feature is in the following configuration paragraph.



When the task is solved the solver's reason can be seen in two places. The first one is in the table of tasks which are part of given workflow. And the second one in the detail of the particular task.



Configuration

There exist several ways how to configure solver notes. The general approach is the setting from workflow. Workflow bpmn20 file contains the following section which is used for configuration of decision buttons in userTask. The section looks like this:

```
<dataObject id="approve" name="approve" itemSubjectRef="xsd:string">
  <extensionElements>
<activiti:value>{"showWarning":false,"level":"success","reasonRequired":true
}</activiti:value>
  </extensionElements>
</dataObject>
```

To avoid the need of workflow modification and an easy configuration of the most common decisions i.e. `approve` and `disapprove` there are shortcuts in IdM properties. Both accepts **true** or **false** values.

`idm.sec.core.wf.approval.reason.required` - sets decision reason required for APPROVAL decision `idm.sec.core.wf.disapproval.reason.required` - sets decision reason required for DISAPPROVAL decision

If decision button `apporve` is set like shown above, `showWarning` attribute ensures displaying of the

modal window, where a solver note can be inserted. The second parameter `reasonRequired` specifies that the solver note is mandatory and the action cannot be finished without it. If only `showWarning` is set the note can be set but its filling is not required. It's important to note that when our workflow calls other sub-workflows it is important and also sufficient to change this setting in the parent workflow only. The setting is propagate from the parent WF to sub-WF and will override the setting of the sub-WF. In most cases the userTask uses two decision types: `id=approve` and `id=disapporve`. One can avoid the need of workflow changing for those two buttons thanks to the IdM properties `idm.sec.core.wf.approval.reason.required` and `idm.sec.core.wf.disapproval.reason.required`. If they are set to true solver note/reason is required for the corresponding type of decision button choice. The setting works globally for all workflows in the IdM. If those IdM properties are set and there is also set `reasonRequired` parameter in a workflow, the setting from the workflow is preferred over that IdM properties for userTasks from that workflow.

Custom task detail

Dynamic definition of components provide us with quick and easy way to add another item to task. But even this system has a few disadvantages:

* Components cannot interact with each other. * Cannot modified placement of each component on task detail (except for order). * Cannot show complex components like tables with specific properties (show detail, specific filtr, etc.).

But if some of these feature is necessary on task detail, yet it can be made by specially created detail, which can be modified as you wish. it means, in frontend will be created new component (page), which will be used in specific task.

Usage of this specially created detail is setted in user's task detail with parameter **Form key**. If this parameter is filled, **DynamicTaskDetail** will not be used, but component with same name as value of Form key parameter will be used.

```
    <userTask id="managerTask" name="Schválení žádosti vedoucím"
activiti:formKey="dynamicRoleTaskDetail">
```

Definition of link between name of component and it's real reprezentation (import) is determined in **component-descriptor.js**:

```
module.exports = {
  'id': 'core',
  'name': 'Core',
  'description': 'Components for Core module',
  'components': [
    {
      'id': 'dynamicRoleTaskDetail',
      'component':
require('./content/task/identityRole/DynamicTaskRoleDetail')
    }
```

```
  ]
};
```

Localization

There are three ways to name buttons in decision:

- **Custom text** - If we do not want to create localization we can put in custom text. We could put "Solve" as **label**, and because for that value in localization will not be found result, value "Solve" will be written.
- **Localization key** - If we want use specific localization translation, localization key have to be used. In **label** will be put something like "decision.approve.label". In this case value from localization for this key will be used.
- **Automatic assembly of localization key** - If we do not fill value **label**, this value will be filled automatically in form **decision.approve.label**, where approve is id of item. For this key system will search in localization and if key will not be found, key will be shown.

Localization is supported for these decision's items **label**, **tooltip**, **warningMessage**.

In similar way works localization for items in dynamic generated detail task. Automatic key uses prefix **formData**. Supported items are **name**, **tooltip**, **placeholder**

Localization example

```
 "decision" : {
    "approve": {
      "label" : "Schválit",
      "tooltip": "Provede schválení tohoto úkolu.",
      "warning": "Opravdu chcete provést schválení?"
    },
    "disapprove": {
      "label" : "Zamítnout",
      "tooltip": "Zamítnout úkol.",
      "warning": "Opravdu chcete zamítnout úkol?"
    },
    "backToApplicant": {
      "label" : "Vrátit žadateli",
      "tooltip": "Vrátí žádost žadateli.",
      "warning": "Opravdu chcete vrátit žádost žadateli?"
    },
    "createRequest": {
      "label" : "Vytvořit žádost",
      "tooltip": "Vytvoří žádost.",
      "warning": "Opravdu chcete vytvořit žádost?"
    },
    "cancelRequest": {
```

```
      "label" : "Smazat žádost",
      "tooltip": "Zruší tuto žádost.",
      "warning": "Opravdu chcete zrušit tuto žádost?"
    }
  },
  "formData" : {
    "applicantDescription": {
      "name" : "Poznámka žadatele",
      "tooltip": "Poznámka zadaná žadatelem při založení žádosti",
      "placeholder": "Poznámka zadaná žadatelem při založení žádosti"
    },
    "managerDescriptionForApplicant": {
      "name" : "Poznámka pro žadatele",
      "tooltip": "Poznámka, která bude žadateli zobrazena při várcení
žádosti.",
      "placeholder": "Poznámka, která bude žadateli zobrazena při várcení
žádosti."
    },
    "managerDescription": {
      "name" : "Poznámka vedoucího",
      "tooltip": "Poznámka od vedoucího.",
      "placeholder": "Poznámka od vedoucího."
    }
  }
```

Sending notifications

There is global configuration in application properties, which can turned on or off sending notifications from UserTask.

```
# Global property that allow disable or enable sending notification from WF
idm.sec.core.wf.notification.send=false
```

Sending notification can be turnd off in each UserTask by setting Form property **sendNotification**. Example of this setting:

```
<activiti:formProperty
    id="sendNotification"
    type="configuration"
    expression="false"
    writable="false">
</activiti:formProperty>
```

Type of variable have to be **configuration**. It is FormType, which is not propagated to frontend. ID or name have to be **sendNotification**. It does not matter even in case of **idm.sec.core.wf.notification.send** is setted to false, because form property **sendNotification** has **higher priority**.

If Form property **sendNotification** is not explicitly filled, notification will be sent.

New listener **TaskSendNotificationEventListener** was created for purposes of sending notifications. This listener reacts on these events: TASK_ASSIGNED, TASK_COMPLETED (comes later) and TASK_CREATED. Each event sends notification with its own topic. Recipient is picked from TaskEntity from assignee attribute. In case of assignee is not filled, notification is send to all candidates. Before notification is sent, existence of identity is verified.