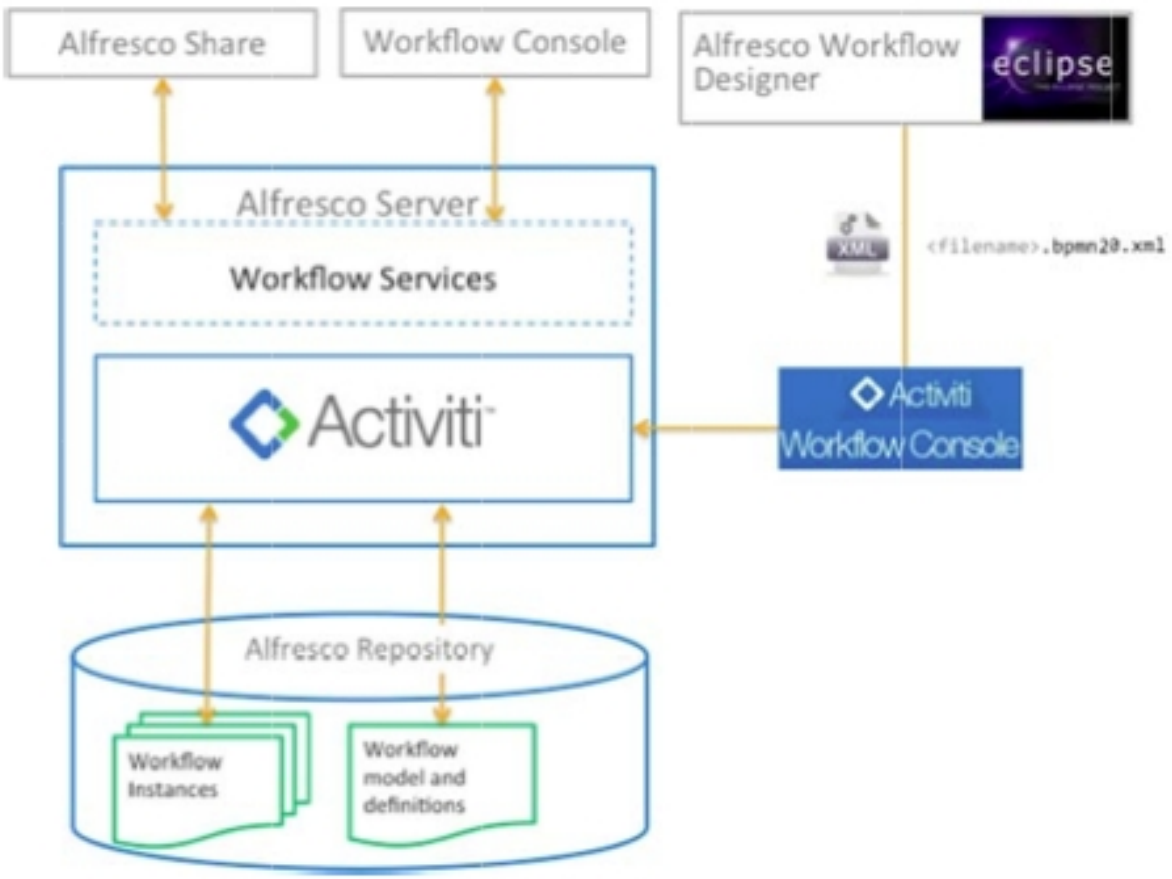
Alfresco Activiti is a BPM platform that can be run on-premise or hosted on a private or public cloud, single or multi-tenant.

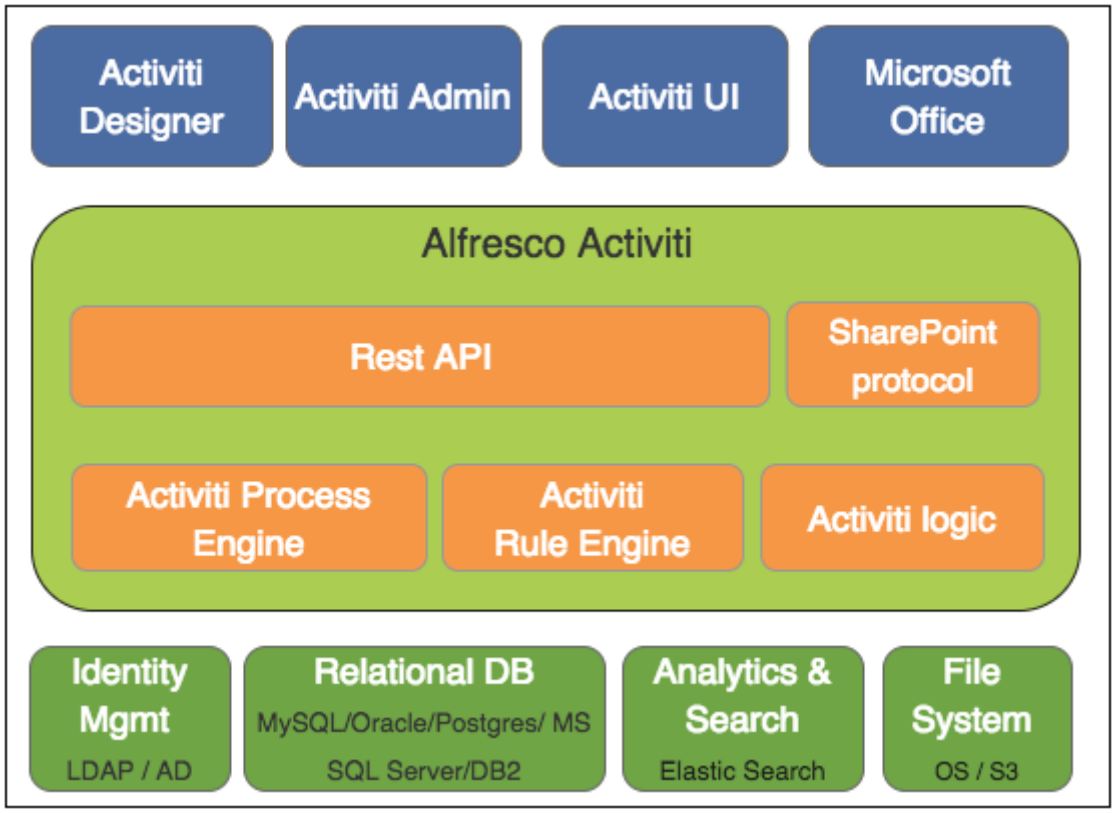
1. Introduction

This guide describes how to develop extensions and customize Alfresco Activiti. We recommend you read the *Activiti Administration Guide* to make sure you have an understanding of how Alfresco Activiti is installed and configured.

2. High Level Architecture

Following diagram gives a high level overview of the technical components in Alfresco Activiti:





Alfresco Activiti is packaged as a regular Java Web application (WAR file) that is deployable in any supported Java web container. The WAR file contains both the Java logic, the REST API resources and the user interface html and javascript files. The application is stateless, which means it does not use any sessions, and requests can be handled by any node in a clustered setup (see the Activiti Administration Guide for more information on multi-node setup).

Following are the technical implementation details:

- The Activiti process engine (enterprise edition) is embedded within Alfresco Activiti and directly used through its Java API.
- The Activiti rule engine is embedded within Alfresco Activiti and directly used through its Java API. The engine executes decisions compliant to the DMN specification (see http://www.omg.org/spec/DMN/Current). Expressions are evaluated using the MVEL expression engine (see https://github.com/mvel/mvel).
- The REST API has two parts:
    - The REST API that exposes operations in the context of the applications that are part of the Alfresco Activiti application. This REST API is used by the Alfresco Activiti user interface and should be used in most cases.
    - The REST API that exposes the core engine Activiti API directly (see the *Activiti User Guide*). Note that this interface is intended for highly custom applications as it exposes the full capabilities and data within the Activiti engine. Therefore, a user with the *tenant admin* or *tenant manager* role is needed to access this part of the REST API for security reasons.

- The application requires Java 7 and is compliant with JEE 6 technologies. The Activiti Engine itself also supports Java 6, however for components such as Elasticsearch, Activiti requires Java 7 or Java 8. See https://www.alfresco.com/services/subscription/supported-platforms for more information on supported platforms.
- The backend logic specific to the Alfresco Activiti logic is implemented using Spring 4 and JPA (Hibernate).
- All user interfaces are written using HTML5 and AngularJS.

Alfresco Activiti uses the following external systems:

- A relational database.
- An elasticsearch installation. Note that the application ships with an embedded elasticsearch by default which requires little configuration.
- A file system (shared file system in multi-node setup) where content is stored.
- An identity management store (LDAP or Active Directory) is optional. By default, a database-backed user and group store is used.

The Activiti process engine used in Alfresco Activiti can be managed using the Activiti Administrator application. This is also provided as a WAR file with Alfresco Activiti distributions.

The Activiti Designer is an Eclipse plugin that can be used by developers to create BPMN 2.0 process definitions within their Eclipse IDE. It is possible to configure the plugin in such a way that it can pull and push process definitions model to the Alfresco Activiti application. For more information on the Designer plugin, see the *Activiti Designer User Guide* (Community edition).

The application can also connect to an Alfresco One installation or to Google Drive (not shown on the diagram).

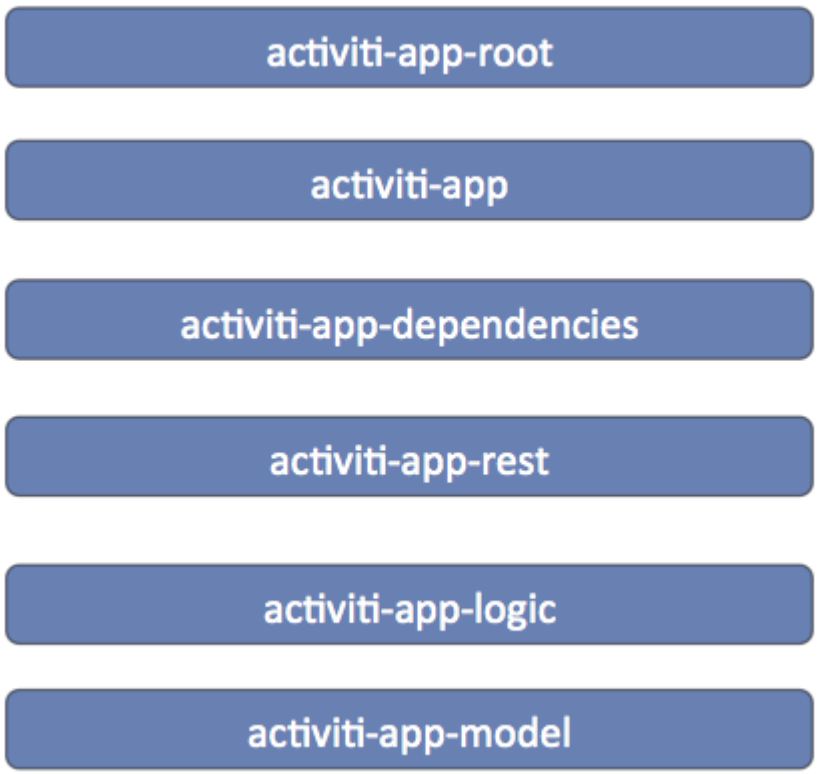
3. Embed the Activiti app in another application

The components of the Activiti app can be included in an existing / other application by referencing the correct Maven dependencies and adding the necessary Spring configuration beans. To simplify, an example application has been created, named activiti-app-embedded-example. If you don't have this example project as part of the Activiti app download, you can ask for a copy with your Alfresco account or sales representative. The Maven pom.xml file in this example project can be used to get an overview of all necessary Maven dependencies. The example project also contains the Spring configuration beans that are needed by the Activiti app components.

The src/main/webapp folder contains all the Javascript sources of the Activiti app in minified format. In addtion, you can have access to the full Javascript source that's provided in a separate bundle. If the context root of the application is changed, make sure to change the URI configuration in the *app-cfg.js* file in the src/main/webapp/scripts folder.

4. Maven modules

When customizing, overriding or creating new logic in Alfresco Activiti, it is useful to be able to develop against the relevant Maven modules. The following Maven modules are the most import one. The diagram is structured in such a way that the lowest module is a dependency of the module one up higher (and so forth).



All Maven modules have **com.activiti** as Maven *groupId*. The version of the artifact is the release version of Alfresco Activiti.

- **activiti-app-model** : Contains the *domain objects*, annotated with JPA annotations for persistency and various Spring repositories for executing the actual database operations. Also has the Java pojo's of the JSON respresentations that are used for example as responses by the REST endpoints.
- **activiti-app-logic** : Contains the services and actual BPM Suite logic.
- **activiti-app-rest** : Contains the REST endpoints that are used by the UI and the public api.

- **activiti-app-dependencies** : Contains all the Alfresco Activiti dependencies. It is also a convenient Maven module (packaging type is *pom*) for development.
- **activiti-app** : Contains configuration classes.
- **activiti-app-root**: Contains the root pom. **Don't use it for development!**

5. Start and task form customization

The start and task forms that are part of a task view can be customized for specific requirements. The following Javascript code example provides an overview of all the form and form field events that can be used to implement custom logic.

By default, a file name *render-form-extensions.js* in the *workflow/extensions* folder is present and loaded in the *index.html* file of the *workflow* folder. It has empty methods by default:

```
var ALFRESCO = ALFRESCO || {};

ALFRESCO.formExtensions = {

    // This method is invoked when the form field have been rendered
    formRendered:function(form, scope) {

    },

    // This method is invoked when input values change (ng-change function
    formFieldValueChanged:function(form, field, scope) {

    },

    // This method is invoked when an input field gets focus (focus event
    formFieldFocus:function(form, field, scope) {

    },

    // This method is invoked when an input field has lost focus (blur eve
    formFieldBlur:function(form, field, scope) {

    },

    // This method is invoked when a person has been selected in the peopl
    formFieldPersonSelected:function(form, field, scope) {

    },

    // This method is invoked when an email has been filled-in in the peop
    formFieldPersonEmailSelected:function(form, field, scope) {

    },
```

```
    // This method is invoked when a person has been removed in the people
    formFieldPersonRemoved:function(form, field, scope) {

    },

    // This method is invoked when a group has been selected in the functi
    formFieldGroupSelected:function(form, field, scope) {

    },

    // This method is invoked when a group has been removed in the functio
    formFieldGroupRemoved:function(form, field, scope) {

    },

    // This method is invoked when content has been uploaded in the upload
    formFieldContentUploaded:function(form, field, scope) {

    },

    // This method is invoked when content has been removed in the upload
    formFieldContentRemoved:function(form, field, scope) {

    },

    // This method is invoked when the REST values or set in a dropdown, r
    formFieldRestValuesSet:function(form, field, scope) {

    },

    // This method is invoked when the complete or an outcome button has b
    formBeforeComplete:function(form, outcome, scope) {

    },

    // This method is invoked when input values change (ng-change function
    formTableFieldValueChanged:function(form, field, columnDefinition, edi

    },

    // This method is invoked when an input field gets focus (focus event
    formTableFieldFocus:function(form, field, columnDefinition, editRow, s

    },

    // This method is invoked when an input field has lost focus (blur eve
    formTableFieldBlur:function(form, field, columnDefinition, editRow, sc

    },

    // This method is invoked when the REST values or set in a dropdown fi
    formTableFieldRestValuesSet:function(form, field, columnDefinition, ed

    },
```

```
        // This method is invoked when the form fields have been rendered
        formTableRendered:function(form, field, columnDefinitions, editRow

        },

        // This method is invoked when the complete button has been clicke
        formTableBeforeComplete:function(form, field, editRow, scope) {

        },

        // This method is invoked when the cancel button has been clicked
        formTableBeforeCancel:function(form, field, editRow, scope) {

    },

        // This method is invoked when input values change (ng-change func
        formValidateFieldValueChanged:function(form, field, scope) {

        },

        // This method is invoked when the complete button has been clicke
        formValidateBeforeSubmit:function(form, outcome, scope) {

        },

        // This method is invoked when input values change (ng-change func
        formTableValidateFieldValueChanged:function(form, field, columnDef

        },

        // This method is invoked when the complete button has been clicke
        // when false (boolean) is returned.
        formTableValidateBeforeComplete:function(form, field, editRow, sco

        },

    // This method is invoked when a task is completed successfully
    taskCompleted:function(taskId, form, scope) {

    },

    // This method is invoked when a task is completed unsuccessfully
    taskCompletedError:function(taskId, errorResponse, form, scope) {

    },

    // This method is invoked when a task is saved successfully
    taskSaved:function(taskId, form, scope) {

    },

    // This method is invoked when a task is saved unsuccessfully
    taskSavedError:function(taskId, errorResponse, form, scope) {
```

```
    }
};
```

This file can be changed to add custom logic. Alternatively, it is also possible to add new javascript files and reference them in the *index.html* file (do take those files in account when upgrading to newer versions of the application) but it is also possible to load additional folders using the activiti resource loader, see *Custom web resources* section below.

In every event method the full form variable is passed as a parameter. This form variable contains the form identifier and name, but also the full set of form fields with type and other configuration information.

In addition the changed field is passed when applicable and the Angular scope of the form renderer is also included. This is a regular Angular directive (i.e. isolated) scope, with all methods available.

For example, to get the current user:

```
formRendered:function(form, scope) {
    var currentUser = scope.$root.account;
    console.log(currentUser);
}
```

6. Custom form fields

Custom form field types can be added through custom *form stencils*. A form stencil is based on the default form stencil and can have default form field types removed, reordered, tweaked (changing the name, icon, etc.) or have new form field types.

Form stencils are defined in the *Stencils* section of the *Kickstart App*. A new form field type consists of the following:

- An html template that is rendered when drag and dropping from the palette on the form canvas is the form builder.
- An html template that is rendered when the form is displayed at runtime.
- An optional custom AngularJS controller in case custom logic needs to be applied to the form field.
- An optional list of third party scripts that are needed when working with the form field at runtime.

6.1. Example 1: Static image

This is a very basic example of a custom form field type that simply displays a static image.

*Create a new form stencil in the *Kickstart App* and click the *Add new item* link.

The *Form runtime template* (the html used when the form is rendered at runtime) and the *Form editor template* (the html used in the form builder) is the same here:

```
<img src="http://activiti.org/images/activiti_logo.png"></img>
```

6.2. Example 2: Dynamic image

Create another new item for the form stencil. This time, we'll create a configurable image. So unlike the static image of the previous example, here the user building the form will be able to select the image that will be displayed.

The *Form runtime template* needs to show the image that the form builder has selected. We'll assume we have set a property *url* (see later on). Note how we're using *ng-src* here (see AngularJs docs on ng-src) to have a dynamic image:

```
<img ng-src="{{field.params.customProperties.url}}"></img>
```

Note the syntax **field.params.customProperties** to get access to the non-default properties of the the form field.

The *Form editor template* simply needs to be a generic depiction of an image or even simpler like here, just a bit of text

```
<i>The custom image here</i>
```

Don't forget to add a property *url* to this stencil item with the name *url* and type *text*.

6.3. Example 3: Dynamic pie chart

This example is more advanced then the previous two: here, we'll have a simple list of number fields with a button at the bottom to add a new line item, while generating a pie chart on the right.

We'll use the 'Epoch' library as an example here. Download the following files from its Github site:

- d3.min.js
- epoch.min.js

Create a new form stencil item and name it "Chart". Scroll down to the *Script library imports* section, and upload the two libraries. At runtime, these third party libraries will be included when the form is rendered.

Note: The order in which the third party libraries are defined is important. Since the Epoch library depends on d3, d3 needs to be first in the table and epoch second (as that is the order in which they are loaded at runtime).

The *Form editor template* is the easy part. We could just use an image of a pie chart here.

```
<img src="url_to_pie_chart_image.png"></img>
```

Let's first define the controller for this form field type. The controller is an AngularJs controller, that will do mainly three things:

- Keep a model of the line items
- Implement a callback for the button that can be clicked
- Store the value of the form field in the proper format of Activiti

```
angular.module('activitiApp')
    .controller('MyController', ['$rootScope', '$scope', function ($rootSc

        console.log('MyController instantiated');

        // Items are empty on initialisation
        $scope.items = [];

        // The variable to store the piechart data (non angular)
        var pieChart;

        // Epoch can't use the Angular model, so we need to clean it
        // (remove hashkey etc, specific to Angular)
        var cleanItems = function(items) {
            var cleanedItems = [];
            items.forEach(function(item) {
               cleanedItems.push( { label: item.label, value: item.value}
            });

            return cleanedItems;
        };

        // Callback for the button
        $scope.addItem = function() {

            // Update the model
            $scope.items.push({ label: 'label ' + ($scope.items.length + 1
```

```
            // Update the values for the pie chart
            // Note: Epoch is not an angular lib so doesn't use the model
            if (pieChart === undefined) {

                pieChart = jQuery('.activiti-chart-' + $scope.field.id).ep
                    type: 'pie',
                    data: cleanItems($scope.items)
                });
                console.log('PieChart created');

            } else {

                $scope.refreshChart();

            }

        };


        // Callback when model value changes
        $scope.refreshChart = function() {
            pieChart.update(cleanItems($scope.items));
            console.log('PieChart updated');
        };


        // Register this controller to listen to the form extensions metho
        $scope.registerCustomFieldListener(this);

        // Deregister on form destroy
        $scope.$on("$destroy", function handleDestroyEvent() {
            console.log("destroy event");
            $scope.removeCustomFieldListener(this);
        });

        // Setting the value before completing the task so it's properly s
        this.formBeforeComplete = function(form, outcome, scope) {
            console.log('Before form complete');
            $scope.field.value = JSON.stringify(cleanItems($scope.items));
        };

        // Needed when the completed form is rendered
        this.formRendered = function(form, scope) {
            console.log(form);
            form.fields.forEach(function(field) {
                if (field.type === 'readonly'
                      && $scope.field.id == field.id
                      && field.value
                      && field.value.length > 0) {

                    $scope.items = JSON.parse(field.value);
                    $scope.isDisabled = true;

                    pieChart = jQuery('.activiti-chart-' + $scope.field.id
                        type: 'pie',
```

```
                        data: cleanItems($scope.items)
                    });

                }
            });
        };

}]);
```

The *Form runtime template* needs to reference this controller, use the model and link the callback for the button:

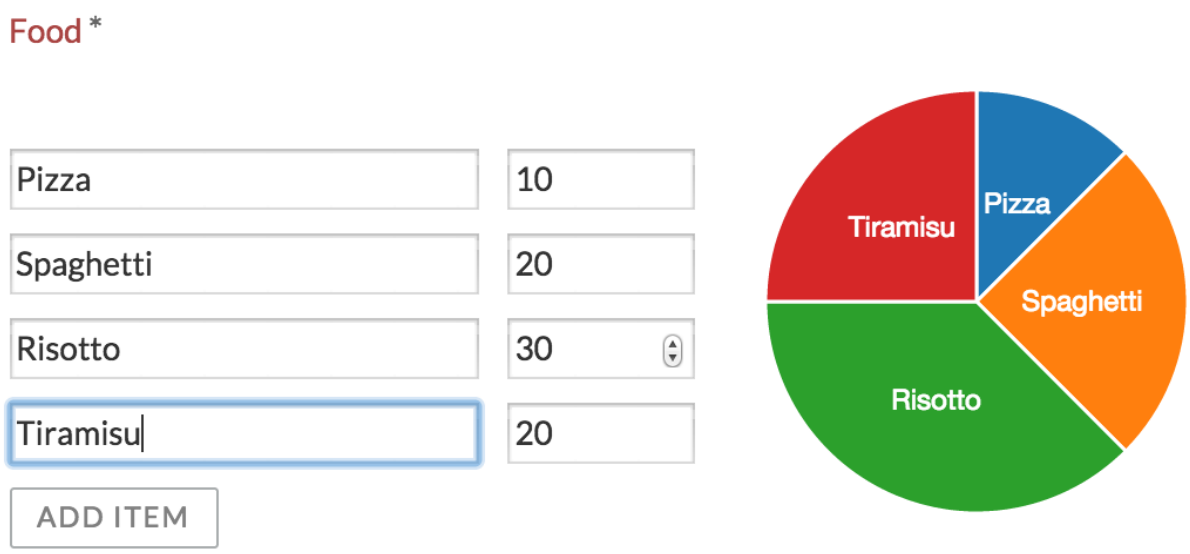
```
<link rel="stylesheet" type="text/css" href="https://cdnjs.cloudflare.com/

<div ng-controller="MyController" style="float:left;margin: 35px 20px 0 0;
    <div ng-repeat="item in items">
          <input type="text" ng-model="item.label" style="width:200px; mar
          <input type="number" ng-model="item.value" style="width: 80px; m
    </div>

    <div>
        <button class="btn btn-default btn-sm" ng-click="addItem()" ng-dis
           Add item
        </button>
    </div>
</div>

<div class="epoch category10" ng-class="'activiti-chart-' + field.id" styl
<div class="clearfix"></div>
```

At runtime, the following will be rendered:



7. Custom web resources

If you want to add additional javascript functionality or override css rules, you can configure lists of additional web resources that are loaded by the browser for each Activiti app. You do this by configuring a new *resource* in the *tomcat/webapps/activiti-app* folder. Following is an example of a new resource section in the *app-cfg.js* file located in the *tomcat/webapps/activiti-app/scripts* folder:

```
ACTIVITI.CONFIG.resources = {
    '*': [
        {
            'tag': 'link',
            'rel': 'stylesheet',
            'href': ACTIVITI.CONFIG.webContextRoot + '/custom/style.css?v=
        }
    ],
    'workflow': [
        {
            'tag': 'script',
            'type': 'text/javascript',
            'src': ACTIVITI.CONFIG.webContextRoot + '/custom/javascript.js
        }
    ]
};
```

The *ACTIVITI.CONFIG.resources* object makes it possible to load different files for each of the Activiti applications using their names as key for a list of additional resources that shall be loaded, the different app names are: *landing*, *analytics*, *editor*, *idm* and *workflow*. The * key means that a default resource list will be used unless there is a specific config key for the app being loaded.

For example, if a user would enter the *editor* app, with the config above deployed, *custom/style.css* would be the only custom resource that would be loaded. If a user would go to the *workflow* app, *custom/javascript.js* would be the only custom resource that would be loaded. So if *workflow* also wants to load the *custom/style.css* that would have to be specified again inside the *workflow* resource list.

Note: Remember to modify the *v*-parameter when you have done changes to your files to avoid the browser from using a cached version of your custom logic.

8. Document Templates

Use the **Generate Document** task to generate a PDF or Microsoft Word document based on a Word document template (.docx). You can insert process variables in the MS Word template that will be replaced with actual values during document transformation.

A document template can be:

- **Tenant wide**: Anyone can use this template in their processes. Useful for company templates.
- **Process model specific**: This template is uploaded while modeling the process model, and is bound to the lifecycle of the process model.

When exporting an App model, process model document templates are included by default and are uploaded again on import. Tenant document templates are not exported, however matched by the document template name as names are unique for tenant document templates.

In the .docx template, you can insert process variables using the following syntax:

```
<<[myVariable]>>
```

Since the above method does not perform null checks, an exception will be thrown at runtime if the variable is null. Therefore, use the following method to prevent such errors:

```
<<[variables.get("myVariable")]>>
```

If this variable is null, a default value will be inserted instead. You can also provide a default value:

```
<<[variables.get("myVariable", "myDefaultValue")]>>
```

Note: Form field types such as Dropdown, Radio button, and Typeahead use *myVariable_ID* for ID and *myVariable_LABEL* for label value. The ID is the actual value used by service tasks and are inserted by default. To display the label value in the generated document, use *myVariable_LABEL*.

The document generation method uses the Aspose library in the backend. For more information about the template syntax and options, see Aspose documentation.

When using the **Generate Document** task, make sure to use the correct syntax for your variables and expressions. Surround your variables with <<[..]>> characters. For example:

<<[variableid]>>

<<[variables.get("variableid")]>>

<<[variables.get("variableid","adefaultifnull")

Some more examples:

- If/else conditional blocks (see Aspose Documentation > Using Conditional Blocks):
  - Text type: *<<if [textfield==day]>> AM, <<else>> PM <</if>>*
  - Amount type: *<<if [annualsalary > $40000]>>, it is generous, <<else>> a standard starting salary <</if>>*
  - Checkbox: *<<if [senstitiveflag=="true"]>>it is Confidential, <<else>> Not Confidential <</if>>*
- Date type: *<<[datefield]>>*
  - Format date type: *<<[datefield]>>:"yyyy.MM.dd">>*
- Number/amount: *<<[amountfield]>>*
- String Boolean: *<<[Genericcheckbox]>>*
- Radio button / Typehead / dropdown: Select *<<[Options_LABEL]>>* with an ID *<<[Options_ID]>>*

The audit log is also generated the same way. For example, the following snippet from the template shows advanced constructs:

Start time: <<[auditInfo.getStartTime()]>>
End time: <<[auditInfo.getEndTime()]>>
Completed by: <<[auditInfo.getAssignee()]>>

<<if [auditInfo.getFormData().size() > 0]>>

Name	Id	Value
<<foreach [in getFormData()]>><<[getFieldName()]>>	<<[getFieldId()]>>	<<[getValue()]>><</foreach>>

<</if>><<if [auditInfo.getSelectedOutcome() != null]>>Selected outcome: **<<[auditInfo.getSelectedOutcome()]>>**<<else>>No outcome selected (default complete)<</if>>

<<if [auditInfo.getComments().size() > 0]>>

Comments

<<foreach [in auditInfo.getComments()]>>
<<[getAuthor()]>>: <<[getMessage()]>>
<</foreach>><</if>>

It is also possible to have custom Spring bean that processes the process variables just before rendering the document, see this section.

9. Custom Logic

Custom logic in a business process is often implemented using a JavaDelegate implementation or a Spring bean. See the Activiti Engine User Guide for more details.

To build against a specific version of Alfresco Activiti, add the following dependency to your Maven *pom.xml* file:

```
<dependencies>
    <dependency>
        <groupId>com.activiti</groupId>
        <artifactId>activiti-app-logic</artifactId>
        <version>${suite.version}</version>
    </dependency>
</dependencies>
```

9.1. Java Delegates

The simplest option is to create a class that implements the *org.activiti.engine.delegate.JavaDelegate* interface, like this:

```
package my.company;

import org.activiti.engine.delegate.DelegateExecution;
import org.activiti.engine.delegate.JavaDelegate;

public class MyJavaDelegate implements JavaDelegate {

    public void execute(DelegateExecution execution) throws Exception {
        System.out.println("Hello from the class delegate");
        execution.setVariable("var1", "Hello from the class delegate");
    }

}
```

Build a jar with this class, and add it to the classpath. In the Service task configuration, set the *'class'* property to using the fully qualified classname (in this case *my.company.MyJavaDelegate*).

9.2. Spring Beans

Another option is to use a Spring bean. It is possible to use a *delegateExpression* on a service task that resolves at runtime to an instance of *org.activiti.engine.delegate.JavaDelegate*. Alternatively, and probably more useful, is to use a general Spring bean. The application automatically scans all beans in the **com.activiti.extension.bean** package. For example:

```
package com.activiti.extension.bean;

import org.activiti.engine.impl.pvm.delegate.ActivityExecution;
import org.springframework.stereotype.Component;

@Component("helloWorldBean")
public class HelloWorldBean {

        public void sayHello(ActivityExecution execution) {
                System.out.println("Hello from " + this);
                execution.setVariable("var3", " from the bean");
        }


}
```

Build a jar with this class, and add it to the classpath. To use this bean in a service task, set the *expression* property to *${helloWorldBean.sayHello(execution)}*.

It is possible to define custom configuration classes (using the Spring Java Config approach) if this is needed (for example when sharing dependencies between delegate beans, complex bean setup, etc.). The application automatically scans for configuration classes in the **package com.activiti.extension.conf;** package. For example:

```
package com.activiti.extension.conf;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CustomConfiguration {

        @Bean
        public SomeBean someBean() {
                return new SomeBean();
        }

}
```

Which can be injected in the bean that will be called in a service task:

```
package com.activiti.extension.bean;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.activiti.extension.conf.SomeBean;
```

```
@Component("helloWorldBeanWithInjection")
public class HelloWorldBeanWithInjection {

        @Autowired
        private SomeBean someBean;

        public void sayHello() {
                System.out.println(someBean.getValue());
        }

}
```

To get the current user, it is possible to use the *com.activiti.common.security.SecurityUtils* helper class.

9.3. Default Spring Beans

The following beans are available out of the box in Alfresco Activiti:

9.3.1. Audit Log Bean ("auditLogBean")

The *auditLogBean* can be used to generate audit logs in .pdf format for a completed process instance or a completed task. The log will be saved as a field value for the process and the task (if a task audit log is generated).

Note: Audit logs can only be used against a completed process instance or a completed task.

The following code can be used in the expression of a service task to generate a process instance audit log named 'My first process instance audit log'. The third argument determines if the current date shall be appended to the file name. The pdf will be associated with the process field 'myFieldName'.

${auditLogBean.generateProcessInstancePdf(execution, 'My first process instance audit log', true, 'myFieldName')}

To create a task audit log named 'My first task audit log' add the following expression to the "complete" event in a task listener. Again the third argument determines if the current date shall be appended to the file name. The pdf will be associated with the field 'myFieldName'.

${auditLogBean.generateTaskPdf(task, 'My first task audit log', true, 'myFieldName')}

You can view the audit logs from the My Tasks app by clicking the "Audit Log" link when viewing the details of a completed process or task. When doing so the following two rest calls are made.

Process instance audit log:

```
GET app/rest/process-instances/{process-instance-id}/audit
```

Task audit log:

```
GET app/rest/tasks/{task-id}/audit
```

9.3.2. Document Merge Bean ("documentMergeBean")

The *documentMergeBean* can be used to merge the content of multiple documents (files of type .doc or .docx) from a process into a single document which will be become the value of a provided process variable. The filename of the new document will be set to the filename of the first field in the list followed by the string "_merged" and the suffix from the same field.

In the following example, the content of *myFirstField* and *mySecondField* will be merged into a new document with the field ID set to *myFirstField* and the filename set to:

<filename-from-myFirstField>_merged.<filenameSuffix-from-myFirstFields>

The new document will become the value of a process variable named *myProcessVariable*.

${documentMergeBean.mergeDocuments('myFirstField;mySecondField', 'myProcessVariable', execution)}

9.3.3. Email Bean ("emailBean")

The *emailBean* can be used to retrieve the email of the current user or the process initiator.

To get the email of the current user use the following expression where 123 is the userId:

${emailBean.getEmailByUserId(123, execution)}

To get the email of the process initiator use the following expression:

${emailBean.getProcessInitiator(execution)}

9.3.4. User Info Bean ("userInfoBean")

The *userInfoBean* makes it possible to get access to general information about a user or just the email of a user.

To get general information about a user (the data that can be found in com.activiti.domain.idm.User), use the following expression where userId is the database ID of the user and can be supplied either as a Long or a String.

${userInfoBean.getUser(123, execution)}

To get the email of a user use the following expression where 123 is the database id of the user and can be supplied either as a Long or a String.

${userInfoBean.getEmail(123, execution)}

To get the first name of a user use the following expression where 123 is the database id of the user and can be supplied either as a Long or a String.

${userInfoBean.getFirstName(123, execution)}

To get the last name of a user use the following expression where 123 is the database id of the user and can be supplied either as a Long or a String.

${userInfoBean.getLastName(123, execution)}

To get both first name and last name of a user use the following expression where 123 is the database id of the user and can be supplied either as a Long or a String.

${userInfoBean.getFullName(123, execution)}

To get a user object representing the current user use the following expression where the returned value is an instance of LightUserRepresentation containing fields like id, firstName, lastName, email, externalId, pictureId.

${userInfoBean.getCurrentUser()}

To get a user's primary group name use the following expression where 123 is the database id of the user and can be supplied either as a Long or a String.

${userInfoBean.getPrimaryGroupName(123)}

To get a group object representing a user's primary group use the following expression where the return value is an instance of LightGroupRepresentation, containing id, name, externalId and status, and where 123 is the database id of the user and can be supplied either as a Long or a String.

${userInfoBean.getPrimaryGroup(123)}

9.4. Hook points

A *hook point* is a place where custom logic can be added. Typically this is done by implementing a certain interface and putting the class implementing the interface on the classpath where it can be found by the classpath component scanning (package *com.activiti.extension.bean* for example)..

9.4.1. Login/LogoutListener

interface: com.activiti.api.security.LoginListener and com.activiti.api.security.LogoutListener

Maven module: activiti-app-logic

An implementation of this class will get a callback when a user logs in or logs out.

Example:

```
package com.activiti.extension.bean;

@Component
public class MyLoginListener implements LoginListener {

        private static final Logger logger = LoggerFactory.getLogger(GfkLo

        public void onLogin(User user) {

                logger.info("User " + user.getFullName() + " has logged in

        }

}
```

9.4.2. Process engine configuration configurer

interface: com.activiti.api.engine.ProcessEngineConfigurationConfigurer

Maven module: activiti-app-logic

An implementation of this class will get called when the Activiti process engine configuration is initialized, but before the process engine is built. This allows for customization to the process engine configuration.

Example:

```
@Component
public class MyProcessEngineCfgConfigurer implements ProcessEngineConfigur

        public void processEngineConfigurationInitialized( SpringProcessEn
                ... // Tweaking the process engine configuration
        }
```

```
}
```

9.4.3. Rule engine configuration configurer

interface: com.activiti.api.engine.DmnEngineConfigurationConfigurer

Maven module: activiti-app-logic

An implementation of this class will get called when the Activiti rule engine configuration is initialized, but before the process engine is built. This allows for customization to the rule engine configuration.

Example:

```
@Component
public class MyDmnEngineCfgConfigurer implements DmnEngineConfigurationCon

        public void dmnEngineConfigurationInitialized(DmnEngineConfigurati
                ... // Tweaking the rule engine configuration
        }

}
```

9.4.4. Process engine event listeners

It is possible to listen to events fired by the Activiti engine. By default (and if enabled) there is a listener that captures these events, processes them before sending them to Elasticsearch (which is used for analytics). If the event data should be going somewhere else, for example an external BI warehouse, the following interface should be implemented and can be used to execute any logic when the event is fired.

See the *example apps* folder that comes with Alfresco Activiti. It has a *jdbc-event-listener* folder, in which a Maven project can be found that captures these events and stored them relationally in another database.

interface: com.activiti.service.runtime.events.RuntimeEventListener

Maven module: activiti-app-logic

All implementations exposing this interface will be injected into the process engine at run time.

Example:

```
package com.activiti.extension.bean;

import com.activiti.service.runtime.events.RuntimeEventListener;
```

```
import org.activiti.engine.delegate.event.ActivitiEvent;

@Component
public class PostgresEventListener implements RuntimeEventListener {

    @Override
    public boolean isEnabled() {
        return true;
    }

    @Override
    public void onEvent(ActivitiEvent activitiEvent) {
        // TODO: handle event here
    }

    @Override
    public boolean isFailOnException() {
        return false;
    }
}
```

9.4.5. Processing document generation variables

interface: com.activiti.api.docgen.TemplateVariableProcessor

Maven module: activiti-app-logic

This section describes the implementation of the document generation task for generating a document based on a MS Word docx template.

An implementation of this class will get called before the variable is passed to the template processor, making it possible to change the value that will be used as the variable name in the template.

Example:

```
@Component
public class MyTemplateVariableProcessor implements TemplateVariableProces

        public Object process(RuntimeDocumentTemplate runtimeDocumentTempl
                return value.toString() + "___" + "HELLO_WORLD";
        }

}
```

Using the above example, you can add *"HELLO_WORLD"* to all variable usages in the template. However, you can also add sophisticated implementations based on process definition lookup using the process definition ID from the execution and inject the RepositoryService in your bean.

In addition to the process definition, the *runtimeDocumentTemplate* is passed to distinguish for which process and template the variables are being prepared. **Note: Only variables with the format *variables.get("myVariable")* in the .docx template will be passed to the *TemplateVariableProcessor* implementation.**

9.4.6. Business Calendar

Use the business calendar when calculating due dates for tasks.

You can override the default business calendar implementation, for example, to include bank holidays, company holidays, and so on. To override the default implementation, add a Spring bean implementing the *com.activiti.api.calendar.BusinessCalendarService* to the classpath with the *@Primary* notation.

Check the Javadoc on the *BusinessCalendarService* for more information.

```
@Primary
@Service
public class MyBusinessCalendarService implements BusinessCalendarService

  ...

}
```

Below is an example implementation that takes weekend days into account when calculating due dates.

```
@Primary
@Service
public class SkipWeekendsBusinessCalendar implements BusinessCalendarServi

    protected static final int DAYS_IN_WEEK = 7;
    protected List<Integer> weekendDayIndex;

    protected DateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");

    public SkipWeekendsBusinessCalendar() {

        // add Saturday and Sunday as weekend days
        weekendDayIndex.add(6);
        weekendDayIndex.add(7);
    }

    public Date addToDate(Date date, int years, int months, int days, int
        return calculateDate(new DateTime(date), years, months, days, hour
    }

    public Date subtractFromDate(Date date, int years, int months, int day
```

```
        return calculateDate(new DateTime(date), years, months, days, hour
    }

    protected Date calculateDate(DateTime relativeDate, int years, int mon
        // if date is on a weekend skip to a working day
        relativeDate = skipWeekEnds(relativeDate, step);
        Period period = new Period(years, months, 0, days, hours, minutes,

        // add weekends to period
        period = period.plusDays(countWeekEnds(relativeDate, period, step)

        // add/subtract period to get the final date, again if date is on
        return skipWeekEnds(addPeriod(relativeDate, period, step), step).t
    }

    protected DateTime addPeriod(DateTime relativeDate, Period period, int
        if (step < 0) {
            return relativeDate.minus(period);
        }
        return relativeDate.plus(period);
    }

    protected DateTime skipWeekEnds(DateTime relativeDate, int step) {
        while(weekendDayIndex.contains(relativeDate.getDayOfWeek())) {
            relativeDate = relativeDate.plusDays(step);
        }
        return relativeDate;
    }

    protected int countWeekEnds(DateTime relativeDate, Period period, int
        // get number of days between two dates
        int days = Math.abs(Days.daysBetween(relativeDate, addPeriod(relat
        int count = 0;

        for(int weekendDay : weekendDayIndex) {
            count+=countWeekDay(relativeDate, weekendDay, days, step);
        }
        return count;
    }

    protected int countWeekDay(DateTime relativeDate, int weekDay, int day
        int count = 0;
        DateTime dt = relativeDate.toDateTime();

        // if date's day of week is not the target day of week
        // skip to target day of week
        if(weekDay != relativeDate.getDayOfWeek()) {
            int daysToSkip = 0;

            if (step > 0) {
                if (weekDay > relativeDate.getDayOfWeek()) {
                    daysToSkip = weekDay - relativeDate.getDayOfWeek();
                } else {
                    daysToSkip = weekDay - relativeDate.getDayOfWeek() + D
                }
```

```
            } else {
                if (weekDay > relativeDate.getDayOfWeek()) {
                    daysToSkip = Math.abs(weekDay - relativeDate.getDayOfW
                } else {
                    daysToSkip = relativeDate.getDayOfWeek() - weekDay;
                }
            }

            // return if target day of week is beyond range of days
            if (daysToSkip > days) {
                return 0;
            }

            count++;
            dt = dt.plusDays(daysToSkip * step);
            days-=daysToSkip;
        }

        if (days>=DAYS_IN_WEEK) {
            dt = dt.plusDays(days * step);
            count+=(Weeks.weeksBetween(relativeDate, dt).getWeeks() * step
        }

        return count;
    }

    @Override
    public DateFormat getStringVariableDateFormat() {
        return dateFormat;
    }
```

9.5. Custom Rest endpoints

It's possible to add custom REST endpoints to the BPM Suite, both in the regular REST API (used by the BPM Suite html/javascript UI) and the *public* API (using basic authentication instead of cookies).

The REST API in Alfresco Activiti is built using Spring MVC. Please check the Spring MVC documentation on how to create new Java beans to implement REST endpoints.

To build against the REST logic of Alfresco Activiti and its specific dependencies, add following dependency to your Maven *pom.xml* file:

```
<dependencies>
    <dependency>
        <groupId>com.activiti</groupId>
        <artifactId>activiti-app-rest</artifactId>
        <version>${suite.version}</version>
    </dependency>
</dependencies>
```

The bean needs to be in the com.activiti.extension.rest package to be found!

A very simple example is shown below. Here, the Activiti TaskService is injected and a custom response is fabricated. Of course, this logic can be anything.

```
package com.activiti.extension.rest;

import com.activiti.domain.idm.User;
import com.activiti.security.SecurityUtils;
import org.activiti.engine.TaskService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/rest/my-rest-endpoint")
public class MyRestEndpoint {

    @Autowired
    private TaskService taskService;

    @RequestMapping(method = RequestMethod.GET, produces = "application/js
    public MyRestEndpointResponse executeCustonLogic() {

        User currentUser = SecurityUtils.getCurrentUserObject();
        long taskCount = taskService.createTaskQuery().taskAssignee(String

        MyRestEndpointResponse myRestEndpointResponse = new MyRestEndpoint
        myRestEndpointResponse.setFullName(currentUser.getFullName());
        myRestEndpointResponse.setTaskCount(taskCount);
        return myRestEndpointResponse;

    }

    private static final class MyRestEndpointResponse {

        private String fullName;
        private long taskCount;

                // Getters and setters

    }

}
```

Create a jar containing this class, and add it to the Alfresco Activiti classpath.

A class like this in the **com.activiti.extension.rest** package will be added to the rest endpoints for the application (e.g. for use in the UI), which use the cookie approach to determine the user. **The url will be mapped under /app**. So, if logged in into the UI of the BPM Suite, one could go to *http://localhost:8080/activiti-app/app/rest/my-rest-endpoint* and see the result of the custom rest endpoint:

```
{"fullName":" Administrator","taskCount":8}
```

To add a custom REST endpoint to the *public REST API*, protected by basic authentication, a similar class should be placed in the **com.activiti.extension.api package**:

```
package com.activiti.extension.api;

import com.activiti.domain.idm.User;
import com.activiti.security.SecurityUtils;
import org.activiti.engine.TaskService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/enterprise/my-api-endpoint")
public class MyApiEndpoint {

    @Autowired
    private TaskService taskService;

    @RequestMapping(method = RequestMethod.GET, produces = "application/js
    public MyRestEndpointResponse executeCustonLogic() {

        User currentUser = SecurityUtils.getCurrentUserObject();
        long taskCount = taskService.createTaskQuery().taskAssignee(String

        MyRestEndpointResponse myRestEndpointResponse = new MyRestEndpoint
        myRestEndpointResponse.setFullName(currentUser.getFullName());
        myRestEndpointResponse.setTaskCount(taskCount);
        return myRestEndpointResponse;

    }

    private static final class MyRestEndpointResponse {

        private String fullName;
        private long taskCount;

        // Getters and setters

    }
```

```
}
```

Note that the endpoint needs to have */enterprise* as first element in the url, as this is configured in the SecurityConfiguration to be protected with basic authentication (more specific, the *api/enterprise/** is).

Which can be accessed like the regular API:

```
> curl -u admin@app.activiti.com:password http://localhost:8080/activiti-a

> {"fullName":" Administrator","taskCount":8}
```

Note: Due to classloading, it is currently not possible to put jars with these custom rest endpoints in the global or common classpath (for example tomcat/lib for Tomcat). They should be put in the web application classpath (for example WEB-INF/lib).

9.6. Custom rule expression functions

The Activiti rule engine uses MVEL as an expression language. In addition to the build in MVEL expression functions there are some additional custom expression functions provided. These are accessible through the structured expression editor within the decision table editor.

The provided custom methods can be overridden by your own custom expression functions or custom methods can be added. This is possible via a hook point in the Activiti rule engine configuration (see Rule engine configuration configurer).

You can configure the Engine with additional expression functions by implementing *CustomExpressionFunctionRegistry*.

interface:
com.activiti.dmn.engine.impl.mvel.config.CustomExpressionFunctionRegistry

Maven module: activiti-dmn-engine

Example:

```
import com.activiti.dmn.engine.CustomExpressionFunctionRegistry;
import org.springframework.stereotype.Component;

import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

@Component
```

```
public class MyCustomExpressionFunctionsRegistry implements CustomExpressi

    public Map<String, Method> getCustomExpressionMethods() {
        Map<String,Method> myCustomExpressionMethods = new HashMap<>();

        try {
            String expressionToken = "dosomething";
            Method customExpressionMethod = SomeClass.class.getMethod("som
            myCustomExpressionMethods.put(expressionToken, customExpressio
        } catch (NoSuchMethodException e) {
            // handle exception
        }

        return myCustomExpressionMethods;
    }
}
```

This registry must be provided to the rule engine configuration using the hook point (see Rule engine configuration configurer).

This example adds the expression function from the example above to the default custom expression functions.

Example:

```
import com.activiti.dmn.engine.DmnEngineConfiguration;
import org.springframework.beans.factory.annotation.Autowired;

public class MyDmnEngineCfgConfigurer implements DmnEngineConfigurationCon

    @Autowired
    MyCustomExpressionFunctionsRegistry myExpressionFunctionRegistry;

    public void dmnEngineConfigurationInitialized(DmnEngineConfiguration d

        dmnEngineConfiguration.setPostCustomExpressionFunctionRegistry(
            myExpressionFunctionRegistry);
    }
}
```

Overriding the default custom expression functions can be done by;

```
dmnEngineConfiguration.setCustomExpressionFunctionRegistry(
    myExpressionFunctionRegistry);
```

10. Custom Data Models

You can create Custom Data Models that connect to external sources and perform custom data operations when working with entity objects.

10.1. Implementing the Custom Data Model service

Implement `AlfrescoCustomDataModelService` to manage operations such as insert, update, and select data in Custom Data Models.

interface: *com.activiti.api.datamodel.AlfrescoCustomDataModelService*

maven module: activiti-app-logic

To implement the *AlfrescoCustomDataModelService* interface:

1. Create an external class named *AlfrescoCustomDataModelServiceImpl* and add it to the classpath.

   Note that it should be in a scannable package such as *com.activiti.extension.bean*.

2. Implement the class as follows:

```
package com.activiti.extension.bean;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.activiti.api.datamodel.AlfrescoCustomDataModelService;
import com.activiti.model.editor.datamodel.DataModelDefinitionRepresentati
import com.activiti.model.editor.datamodel.DataModelEntityRepresentation;
import com.activiti.runtime.activiti.bean.datamodel.AttributeMappingWrappe
import com.activiti.variable.VariableEntityWrapper;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

@Service
public class AlfrescoCustomDataModelServiceImpl implements AlfrescoCustomD

    @Autowired
    protected ObjectMapper objectMapper;

    @Override
    public String storeEntity(List<AttributeMappingWrapper> attributeDefin
            DataModelDefinitionRepresentation dataModel) {
        // save entity data and return entity id
    }

    @Override
    public ObjectNode getMappedValue(DataModelEntityRepresentation entityV
```

```
        // fetch entity data and return as an ObjectNode
    }

    @Override
    public VariableEntityWrapper getVariableEntity(String keyValue, String
        // fetch entity data and return as a VariableEntityWrapper
    }

}
```

The above implementation of *AlfrescoCustomDataModelServiceImpl* class will get called, for example when a select, insert, or update operation on a custom data model is performed.

11. Custom reports

Alfresco Activiti provides a number of out-of-the-box reports in the Analytics app, which can be augmented with your own custom reports.

Custom reports have full access to the ElasticSearch indexes generated by Activiti when this is enabled (see the Admin guide for details on how to configure events to be sent to ElasticSearch), but must be implemented as a custom Spring bean.

The following section assumes that you have a reasonable understanding of what ElasticSearch is and an understanding of indexes, types and type mappings. The ElasticSearch Definitive Guide is a great learning resource if you are new to the engine and there is also a Reference Guide which you should find helpful to refer to as you start using it directly yourself.

11.1. Setting up ElasticSearch for development

See the *Activiti Administration Guide* to configure ElasticSearch in embedded mode within Alfresco Activiti. Alternatively, you can set up an external instance of ElasticSearch 1.7.3 on your machine and configure Alfresco Activiti to connect to it. The following section describes the steps in more detail.

11.1.1. Download and set up ElasticSearch

ElasticSearch provides downloads in multiple formats, however the ZIP download is recommended for general development usage. As mentioned above, you must use ElasticSearch 1.7.3 with the Java client embedded in Alfresco Activiti. The same can be downloaded from the ElasticsSearch releases archive.

Once you have extracted this to a local directory, configure to ensure the ElasticSearch Java client is able to find the cluster in *config/elasticsearch.yml*. In general, you can leave the ElasticSearch default settings as-is.

```
network.host: 127.0.0.1
```

Now you can run ElasticSearch from a terminal, e.g.

```
./bin/elasticsearch
```

Or on Windows

```
bin/elasticsearch.bat
```

The logging output to the console should indicate that ElasticSearch is running successfully on ports 9200 and 9300, e.g.

```
[2016-03-15 16:43:50,117][INFO ][node                     ] [Bird-Brain] v
[2016-03-15 16:43:50,117][INFO ][node                     ] [Bird-Brain] i
[2016-03-15 16:43:50,171][INFO ][plugins                  ] [Bird-Brain] l
[2016-03-15 16:43:50,198][INFO ][env                      ] [Bird-Brain] u
[2016-03-15 16:43:51,831][INFO ][node                     ] [Bird-Brain] i
[2016-03-15 16:43:51,832][INFO ][node                     ] [Bird-Brain] s
[2016-03-15 16:43:51,922][INFO ][transport                ] [Bird-Brain] b
[2016-03-15 16:43:51,936][INFO ][discovery                ] [Bird-Brain] e
[2016-03-15 16:43:55,710][INFO ][cluster.service          ] [Bird-Brain] n
[2016-03-15 16:43:55,729][INFO ][http                     ] [Bird-Brain] b
[2016-03-15 16:43:55,729][INFO ][node                     ] [Bird-Brain] s
[2016-03-15 16:43:55,751][INFO ][gateway                  ] [Bird-Brain] r
[2016-03-15 16:44:41,023][INFO ][cluster.service          ] [Bird-Brain] a
```

11.1.2. Setting up Sense

Sense is a simple Kibana plugin which allows you to send requests to a running ElasticSearch node and see the results displayed in its console. It is especially helpful when you are developing your queries as it allows you to to try these out without any coding, but it also defines a basic cURL-like syntax for requests which is used in all the code examples in the ElasticSearch docs.

You must install the latest version of Kibana in order to be able to use the Sense plugin. This version of Kibana supports ElasticSearch 1.7.x but in order to use it you must add the following lines to the end of your `config/kibana.yml` file in order to turn off the core Kibana functionality which will not work with older versions of ElasticSearch. This does not matter, since the Sense plugin runs standalone and connects to ElasticSearch directly.

```
kibana.enabled: false        # disable the standard kibana discovery, visu
elasticsearch.enabled: false # do not require a running Elasticsearch 2.0
```

After this we can install the Sense plugin and start Kibana

```
./bin/kibana plugin --install elastic/sense
./bin/kibana
```

You should see some information output to the console to indicate that Kibana is running and you can navigate to Sense by pointing your browser to http://localhost:5601/app/sense

To test that everything is working and to start getting a feel for the data mappings defined in the Activiti indexes, enter the following query into the Sense UI and hit the green execute button.

```
GET _mapping
```

11.1.3. Configuring Alfresco Activiti

In order to have some events show up in the Elasticsearch indexes, you must first turn on event generation and processing for your installation and also enable Elasticsearch itself.

To turn on event generation and processing, add the following to your tomcat/lib/activiti-app.properties` file

```
event.generation.enabled=true
event.processing.enabled=true
```

The most straightforward configuration for Elasticsearch is to use the embedded mode, which will start a new local, standalone node within the webapp itself. Don't forget to turn on HTTP access also for development (do not do this in production!) so that you can use cURL or Sense to send queries to Elasticsearch directly.

```
elastic-search.server.type=embedded
elastic-search.enable.http=true
```

Alternatively you can install an external instance of Elasticsearch.

Using unicast

By default, ElasticSearch will use multicast to find other nodes. You configure ElasticSearch to use unicast to reach the local ElasticSearch node that you have started. The following configuration, added to the `activiti-app.properties` file, will configure client mode and unicast discovery in addition to turning on event generation and processing so that these get sent to ElasticSearch.

```
elastic-search.server.type=client
elastic-search.cluster.name=elasticsearch
elastic-search.discovery.type=unicast
elastic-search.discovery.hosts=127.0.0.1[9300-9400]
```

Once you have completed this setup you should restart Alfresco Activiti to ensure that the settings are applied. The console logging output should contain some information indicating that the ElasticSearch client has successfully connected to the cluster.

11.2. Implementing Custom Reports

Assuming that you have started to see some data show up in the ElasticSearch store and therefore in the out-of-the-box reports, and you have used the Sense tool or cURL to develop some custom search queries of your own, you are ready to start implementing the custom Spring bean required in order to plug the report into the Alfresco Activiti UI.

11.2.1. Basic concepts

A custom report is a custom section available in the Analytics app and also within each published app, which shows one or more custom reports.

Each report is implemented by a Spring bean which is responsible for two things

1. Perform an ElasticSearch search query using the Java client API
2. Convert the search results (hits or aggregations) into chart or table data and add this to the response

The UI will automatically display the correct widgets based on the data that your bean sends.

11.2.2. Bean implementation

Your Spring bean will be discovered automatically via annotations but must be placed under the package `com.activiti.service.reporting`. Since this package is used for the out-of-the-box reports it is recommended that custom reports use the sub-package such as `com.activiti.service.reporting.custom`.

The overall structure of the class will be as follows, for the full source please see the web link at the end of this section.

```
package com.activiti.service.reporting.custom;

import com.activiti.domain.reporting.ParametersDefinition;
import com.activiti.domain.reporting.ReportDataRepresentation;
import com.activiti.service.api.UserCache;
import com.activiti.service.reporting.AbstractReportGenerator;
import org.activiti.engine.ProcessEngine;
import org.elasticsearch.client.Client;
import org.springframework.stereotype.Component;

import java.util.Map;

@Component(CustomVariablesReportGenerator.ID)
public class CustomVariablesReportGenerator extends AbstractReportGenerato

    public static final String ID = "report.generator.fruitorders";
    public static final String NAME = "Fruit orders overview";

    @Override
    public String getID() {
        return ID;
    }

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public ParametersDefinition getParameterDefinitions(Map<String, Object
        return new ParametersDefinition();
    }

    @Override
    public ReportDataRepresentation generateReportData(ProcessEngine proce
                                                       Client elasticSearc
                                                       Map<String, Object>

        ReportDataRepresentation reportData = new ReportDataRepresentation

        // Perform queries and add report data here

        return reportData;
    }
```

You must implement the `generateReportData()` method which is declared abstract in the superclass, and you can choose to override the `getParameterDefinitions()` method if you need to collect some user-selected parameters from the UI to use in your query.

Implementing `generateReportData()`

The `generateReportData()` method of your bean is responsible for two things

- Perform one or more ElasticSearch queries to fetch report data
- Populate chart/table data from the query results

A protected helper method `executeSearch()` is provided which provides a consise syntax to execute an ElasticSearch search query given a query and optional aggregation, the implementation of which also provides logging of the query generated by the Java client API before it is sent. This can help with debugging your queries using Sense, or assist you in working out why the Java client is not generating the query you expect.

```
return executeSearch(elasticSearchClient,
                indexName,
                ElasticSearchConstants.TYPE_VARIABLES,
                new FilteredQueryBuilder(
                        new MatchAllQueryBuilder(),
                        FilterBuilders.andFilter(
                                new TermFilterBuilder("processDefinitionKe
                                new TermFilterBuilder("name._exact_name",
                        )
                ),
                AggregationBuilders.terms("customerOrders").field("stringV
        );
```

The log4j configuration required to log queries being sent to ElasticSearch via `executeSearch()` is as follows

```
log4j.logger.com.activiti.service.reporting.AbstractReportGenerator=DEBUG
```

Alternatively you can manually execute any custom query directly via the `Client` instance passed to the `generateReportData()` method, e.g.

```
return elasticSearchClient
                .prepareSearch(indexName)
                .setTypes(ElasticSearchConstants.TYPE_PROCESS_INSTANCES)
                .setQuery(new FilteredQueryBuilder(new MatchAllQueryBuilde
                .addAggregation(
                        new TermsBuilder(AGGREGATION_PROCESS_DEFINITIONS).
                                .subAggregation(new FilterAggregationBuild
                                        .filter(new ExistsFilterBuilder(Ev
                                        .subAggregation(new ExtendedStatsB
```

Generating chart data from queries can be accomplished easily using the converters in the `com.activiti.service.reporting.converters` package. This avoids the need to iterate over returned query results in order to populate chart data items.

Initially two converters `AggsToSimpleChartBasicConverter` and `AggsToMultiSeriesChartConverter` are provided to populate data for pie charts (which take a single series of data) and bar charts (which take multiple series) respectively. These two classes are responsible for iterating over the structure of the ES data, while the member classes of `com.activiti.service.reporting.converters.BucketExtractors` are responsible for extracting an actual value from the buckets returned in the data.

```
ReportDataRepresentation reportData = new ReportDataRepresentation();

PieChartDataRepresentation pieChart = new PieChartDataRepresentation();
pieChart.setTitle("No. of orders by customer");
pieChart.setDescription("This chart shows the total number of orders place

new AggsToSimpleChartBasicConverter(searchResponse, "customerOrders").setC
        pieChart,
        new BucketExtractors.BucketKeyExtractor(),
        new BucketExtractors.BucketDocCountExtractor()
);

reportData.addReportDataElement(pieChart);

SingleBarChartDataRepresentation chart = new SingleBarChartDataRepresentat
chart.setTitle("Total quantities ordered per month");
chart.setDescription("This chart shows the total number of items that were
chart.setyAxisType("count");
chart.setxAxisType("date_month");

new AggsToMultiSeriesChartConverter(searchResponse, "ordersByMonth").setCh
        chart,
        new BucketExtractors.DateHistogramBucketExtractor(),
        new BucketExtractors.BucketAggValueExtractor("totalItems")
);

reportData.addReportDataElement(chart);
```

For more details see the full source on the activiti-custom-reports GitHub project.

12. Cookie configuration

Alfresco Activiti uses an HTTP cookie to store a user session. You can use multiple cookies for different browsers and devices. The application uses a database table to store the cookie values (called *tokens* internally), to allow a shared persistent session store in a multi-node setup.

It’s possible to change the settings regarding cookies:

Property	description	default
security.cookie.max-age	The maximum age of a cookie, expressed in seconds. The max-age determines the period in which the browser will send the cookie with the requests.	2678400 (31 days)
security.cookie.refresh-age	To avoid that a users is suddenly logged out when using the application when reaching the max-age above, tokens are refreshed after this period (expressed in seconds). Refreshing means a new token will be created and a new cookie will be returned which the browser will use for subsequent requests. Setting the refresh-age low, will result in many new database rows when the user is using the application.	86400 (1 day)

By default, cookies will have the *secure* flag set, when the request being made is HTTPS. If you only want to use the remember-me cookie over HTTPS (i.e. make the *secure* flag mandatory), set the following property to true:

Property	default
security.cookie.always-secure	false

To avoid that the persistent token table gets too full, a background job periodically removes obsolete cookie token values. Possible settings:

Property	description	default
security.cookie.database-removal.max-age	The maximum age an entry in the database needs to have to be removed.	Falls back to the security.cookie.max-age setting if not found. This effectively means that cookies which are no longer valid could be removed immediately from the database table.

Property	description	default
security.cookie.database-removal.cronExpression	The cron expression determining when the obsolete database table entries for the cookie values will checked for removal.	0 0 1 * * ? (01:00 at night)

13. Custom Identity Synchronization

Alfresco Activiti needs to have user, group, and membership information in its database. The main reason is performance (for example quick user/group searches) and data consistency (for example models are linked to users through foreign keys). In the Alfresco Activiti logic, this is typically referred to as Identity Management (IDM).

Out of the box, all IDM data is stored directly in the database. So when you create a user or group as a tenant administrator, the data simply ends up in the Alfresco Activiti database tables.

However, typically the users/groups of a company are managed in a centralized data store such as LDAP (or Active Directory). Alfresco Activiti can be configured to connect to such a server and synchronize the IDM data to the Alfresco Activiti database table. See the section "External Identity Management" in the *Admin Guide* for more information on how to set this up. The basic idea behind it is that the LDAP server will periodically be polled and the IDM data in the database tables will be synchronized: created, updated or deleted depending on what the LDAP server returns and what currently is in the database tables.

This section describes what is needed to have a similar synchronization of IDM data coming from another source. The *com.activiti.service.idm.LdapSyncService* responsible for synchronizing IDM data from an LDAP/Active Directory store, uses the same hook points as the ones described below and can thus be seen as an advanced example.

13.1. Example implementation

Let's create a simple example synchronization service that demonstrates clearly the concepts and classes to be used. In this example, we'll use a simple text file to represent our 'external IDM source'. The *users.txt* looks as follows (each line is a user and user data is separated by semi-colons):

```
jlennon;John;Lennon;john@beatles.com;johnpassword;10/10/2015
rstarr;Ringo;Starr;ringo@beatles.com;ringopassword;11/10/2015
gharrison;George;Harrison;george@beatles.com;georgepassword;12/10/2015
pmccartney;Paul;McCartney;paul@beatles.com;paulpassword;13/10/2015
```

The *groups.txt* file is similar (the group name followed by the member ids and a timestamp):

```
beatles:jlennon;rstarr;gharrison;pmccartney:13/10/2015
singers:jlennon;pmccartney:17/10/2015
```

The application expects *one* instance implementing the **com.activiti.api.idm.ExternalIdmSourceSyncService** interface to be configured when synchronizing with an external IDM source. This interface requires a few methods to either synchronous or asynchronous do a full or differential sync. In a full sync, all data is looked at and compared. A differential sync only returns what has changed since a certain date. The latter is of course used for performance reasons. For example, the default settings for LDAP do a full sync every night and a differential sync every four hours.

You can also implement the **com.activiti.api.idm.ExternalIdmSourceSyncService** interface directly, but there is an easier way: all the logic to fetch data from the tables, compare, create, update or delete users, groups or membership is encapsulated in the **com.activiti.api.idm.AbstractExternalIdmSourceSyncService** class. It is advised to extend this class when creating a new external source synchronization service, as in that case the only logic that needs to be written is the actual fetching of the IDM data from the external source.

So let's create a **FileSyncService** class. Note the package, *com.activiti.extension.bean*, which is automatically component scanned. The class is annotated with *@Component* (@Service would also work).

```
package com.activiti.extension.bean;

@Component
public class FileSyncService extends AbstractExternalIdmSourceSyncService

  ...

}
```

The **com.activiti.api.idm.ExternalIdmSourceSyncService** defines the different abstract methods that can be implemented. For example:

The *additionalPostConstruct()* method will be called after the bean is constructed and the dependencies are injected.

```
    protected void additionalPostConstruct() {
                    // Nothing needed now
    }
```

It's the place to add additional post construction logic, like reading properties from the configuration file. Note the *env* variable is available for that, which is a standard *org.springframework.core.env.Environment* instance:

```
    protected void additionalPostConstruct() {
        myCustomConfig = env.getProperty("my.custom.property");
    }
```

The *getIdmType()* method simply returns a String identifying the external source type. It is used in the logging that is produced when the synchronization is happening.

```
    protected String getIdmType() {
      return "FILE";
    }
```

The *isFullSyncEnabled(Long tenantId)* and *isDifferentialSyncEnabled(Long tenantId)* configures whether or not respectively the *full* and/or the *differential* synchronization is enabled.

```
protected boolean isFullSyncEnabled(Long tenantId) {
    return true;
}

protected boolean isDifferentialSyncEnabled(Long tenantId) {
    return false;
}
```

Note that the tenantId is passed here. In a non-multitenant setup, this parameter can simply be ignored. All methods of this superclass have the tenantId parameter. In a multi-tenant setup, one should write logic to loop over all the tenants in the system and call the sync methods for each of the tenants separately.

The following two methods will configure when the synchronizations will be scheduled (and executed asynchronously). The return value of these methods should be a (Spring-compatible) cron expression. Note that this typically will be configured in a configuration properties file rather than hardcoded. When nulll is returned, that particular synchronization won't be scheduled.

```
    protected String getScheduledFullSyncCronExpression() {
        return "0 0 0 * * ?"; // midnight
    }

    protected String getScheduledDifferentialSyncCronExpression() {
        return null;
    }
```

Now we get to the important part of the implementation: the actual fetching of users and groups. This is the method that is used during a **full synchronization**.

```
protected ExternalIdmQueryResult getAllUsersAndGroupsWithResolvedMembers(L
    try {
      List<ExternalIdmUserImpl> users = readUsers();
      List<ExternalIdmGroupImpl> groups = readGroups(users);
      return new ExternalIdmQueryResultImpl(users, groups);
    } catch (Exception e) {
      e.printStackTrace();
    }
    return null;
  }
```

The return result, an instance of **com.activiti.domain.sync.ExternalIdmQueryResult**, which has a list of users in the form of **com.activiti.domain.sync.ExternalIdmUser** instances and a list of groups in the form of **com.activiti.domain.sync.ExternalIdmGroup** instances.

Note that each group has its members and child groups in it. Also note that these are all *interfaces*, so you are free to return any instance that implements these interfaces. By default there are simple POJO implementations of said interfaces: **com.activiti.domain.sync.ExternalIdmQueryResultImpl**, **com.activiti.domain.sync.ExternalIdmUserImpl** and **com.activiti.domain.sync.ExternalIdmGroupImpl**. These POJOs are also used in the example implementation above.

Important note: the *ExternalIdmUser* interface also defines a *getPassword()* method. Only return the actual password here if you want the user to authenticate against the default Activiti tables. The returned password will be securily hashed and stored that way. Return null if the authentication is done against an external system (LDAP is such an example). See further down to learn more about custom authentication.

The *readUsers()* and *readGroups()* methods will read the .txt mentioned above from the classpath and create instances of user and groups classes using the information in those files. For example:

```
protected List<ExternalIdmUserImpl> readUsers() throws IOException, ParseE

                List<ExternalIdmUserImpl> users = new ArrayList<ExternalId

            InputStream inputStream = this.getClass().getClassLoader().get
            BufferedReader bufferedReader = new BufferedReader(new InputSt
            String line = bufferedReader.readLine();
            while (line != null) {

                String[] parsedLine = line.split(";");

                ExternalIdmUserImpl user = new ExternalIdmUserImpl();
                user.setId(parsedLine[0]);
                user.setOriginalSrcId(parsedLine[0]);
                user.setFirstName(parsedLine[1]);
                user.setLastName(parsedLine[2]);
                user.setEmail(parsedLine[3]);
                user.setPassword(parsedLine[4]);
                user.setLastModifiedTimeStamp(dateFormat.parse(parsedLine[

                users.add(user);
                line = bufferedReader.readLine();
            }

            inputStream.close();
            return users;
    }

        protected List<ExternalIdmGroupImpl> readGroups(List<ExternalIdmUs

                List<ExternalIdmGroupImpl> groups = new ArrayList<External

            InputStream inputStream = this.getClass().getClassLoader().get
            BufferedReader bufferedReader = new BufferedReader(new InputSt
            String line = bufferedReader.readLine();
            while (line != null) {

                String[] parsedLine = line.split(":");
                String groupId = parsedLine[0];

                ExternalIdmGroupImpl group = new ExternalIdmGroupImpl();
                group.setOriginalSrcId(groupId);
                group.setName(groupId);

                List<ExternalIdmUserImpl> members = new ArrayList<External
                String[] memberIds = parsedLine[1].split(";");
                for (String memberId : memberIds) {
                        for (ExternalIdmUserImpl user : users) {
                                if (user.getId().equals(memberId)) {
```

```
                                        members.add(user);
                                    }
                                }
                            }
                            group.setUsers(members);

                            group.setLastModifiedTimeStamp(dateFormat.parse(parsedLine

                            groups.add(group);
                            line = bufferedReader.readLine();
                        }

                        inputStream.close();
                        return groups;
                    }
```

For the **differential synchronization** a similar implementation could be made. Note that now a timestamp is passed, which indicates that the method should only return user/groups that are changed since that timestamp.

```
protected List<? extends ExternalIdmUser> getUsersModifiedSince(Date lat

  ...

}

protected List<? extends ExternalIdmGroup> getGroupsModifiedSince(Date l

  ....

}
```

The last two methods we need to implement are to indicate which users should become a tenant admin (or a tenant manager in a multi-tenant setup). This method should return an array of string with the **id used in the external IDM store**. More specifically, the strings in this array will be compared with the value in the **ExternalIdmUser.getOriginalSrcId()** method. Note that in practice these strings often will come from a configuration file rather than being hardcoded.

```
protected String[] getTenantManagerIdentifiers(Long tenantId) {
  return null; // No tenant manager
}

protected String[] getTenantAdminIdentifiers(Long tenantId) {
    return new String[] { "jlennon" };
}
```

That's all there is to it. As shown, no actual synchronization logic needs to be written when extending from the **AbstractExternalIdmSourceSyncService** class. The implementation should only worry about configuration and the actual fetching of the user and group information.

13.2. Synchronization on boot

On a first boot, it's needed to sync all users/groups for the first time, or else nobody would be able to log in. The LDAP synchronization logic does this automatically. When creating a custom synchronization service, a custom *BootstrapConfigurer* can be used to do the same thing:

```
package com.activiti.extension.bean;

@Component
public class MyBootstrapConfigurer implements BootstrapConfigurer {

  @Autowired
  private FileSyncService fileSyncService;

  public void applicationContextInitialized(org.springframework.context.Ap
    fileSyncService.asyncExecuteFullSynchronizationIfNeeded(null);
  }

}
```

So what we're doing here is implementing the **com.activiti.api.boot.BootstrapConfigurer** interface. If there is an instance implementing this interface on the classpath, it will be called when the application is booting up (more precisely: after the Spring application context has been initialized). Here, the class we created in the previous section, *FileSyncService* is injected. Note we add it to the component scanned package again and added the *@component* identifier.

We simply call the *asyncExecuteFullSynchronizationIfNeeded()* method. The null parameter means 'the default tenant' (i.e. this is a non-multitenant setup). This is a method from the *com.activiti.api.idm.ExternalIdmSourceSyncService* interface, which will do a full sync if no initial synchronization was done before.

As a side note, all synchronization logs are stored in a table **IDM_SYNC_LOG** in the database.

13.3. Synchronization log entries

When a synchronization is executed, a log is kept. This log contains all information about the synchronization: users/groups that are created, updates of existing users/groups, membership additions/deletions, etc.

To access the log entries, an HTTP REST call can be done:

```
GET /api/enterprise/idm-sync-log-entries
```

Which returns a result like this (only an initial synchronization happened here):

```
[{"id":1,"type":"initial-ldap-sync","timeStamp":"2015-10-16T22:00:00.000+0
```

This call uses the following url parameters:

- *tenantId*: Defaults to the tenantId of the users
- *start* and *size*: Used for getting paged results back instead of one (potentially large) list.

Note that this call can only be done by a *tenant administrator*, or *tenant manager* in a multi-tenant setup.

We can now get the detailed log for each sync log entry, by taking an id from the previous response:

```
GET /api/enterprise/idm-sync-log-entries/{id}/logfile
```

This returns a .log file that contains for our example implementation

```
created-user: created user John Lennon (email=john.lennon@thebeatles.com)
added-capability: added capability tenant-mgmt to user jlennon
created-user: created user Ringo Starr (email=ringo.starr@thebeatles.com)
created-user: created user George Harrison (email=george.harrison@beatles.
created-user: created user Paul McCartney (email=paul.mccartney@beatles.co
created-group: created group beatles
added-user-to-group: created group membership of user jlennon for group be
added-user-to-group: created group membership of user rstarr for group bea
added-user-to-group: created group membership of user gharrison for group
added-user-to-group: created group membership of user pmccartney for group
created-group: created group singers
added-user-to-group: created group membership of user jlennon for group si
added-user-to-group: created group membership of user pmccartney for group
```

13.4. Custom authentication

When using a custom external IDM source, you may need to authenticate against that source (For example, LDAP). See the section on global security overriding for more information on how to use our users.txt file as shown above as an authentication mechanism.

14. Security configuration overrides

The security is configured by the **com.activiti.conf.SecurityConfiguration** class in Alfresco Activiti. It allows to switch between database and LDAP/Active Directory authentication out of the box. It also configures REST endpoints under "*/app*" to be protected using a cookie-based approach with tokens and REST endpoints under "*/api*" to be protected by Basic Auth.

You can override these defaults, if the out-of-the-box options are not adequate for your environment. The following sections describe the different options.

All the *overrides* described in the following sections follow the same pattern of creating a Java class that implements a certain interface. This class needs to be annotated by *@Component* and must be found in a package that is component-scanned.

14.1. Global security override

This is the most important override. It allows to replace the default authentication mechanism used.

The interface to implement is **com.activiti.api.security.AlfrescoSecurityConfigOverride**. It has one method *configureGlobal* which will be called instead of the default logic (which sets up either database-backed or LDAP-backed authentication) if an instance implementing this interface is found on the classpath.

Building further on the example in the custom IDM synchronization example, let's use the *users.txt* file to, in combination with the *FileSyncService* there, have the application use the user information in the file to also execute authentication.

Spring Security (which is used as underlying framework for security) expects an implementation of the *org.springframework.security.authentication.AuthenticationProvider* to execute the actual authentication logic. What we have to do in the *configureGlobal* method is then instantiate our custom class:

```
package com.activiti.extension.bean;

@Component
```

```
public class MySecurityOverride implements AlfrescoSecurityConfigOverride

  public void configureGlobal(AuthenticationManagerBuilder auth, UserDetai
    MyAuthenticationProvider myAuthenticationProvider = new MyAuthenticati
    myAuthenticationProvider.setUserDetailsService(userDetailsService);
    auth.authenticationProvider(myAuthenticationProvider);
  }

}
```

Note how we pass the default *UserDetailsService* to this authentication provider. This class is responsible for loading the user data (and its capabilities or *authorities* in Spring Security lingo) from the database tables. Since we synchronized the user data using the same source, we can just pass it to our custom class.

So the actual authentication is done in the *MyAuthenticationProvider* class here. In this simple example, we just have to compare the password value in the *users.txt* file for the user. To avoid having to do too much low-level Spring Security plumbing, we let the class extend from the **org.springframework.security.authentication.dao.AbstractUserDetailsAuthenticationProvider** class.

```
public static class MyAuthenticationProvider extends AbstractUserDetailsAu

  protected Map<String, String> userToPasswordMapping = new HashMap<String

  protected UserDetailsService userDetailsService;

  public MyAuthenticationProvider() {

    // Read users.txt, and create a {userId, password} map
    try {
      InputStream inputStream = this.getClass().getClassLoader().getResour
      BufferedReader bufferedReader = new BufferedReader(new InputStreamRe
      String line = bufferedReader.readLine();
      while (line != null) {
        String[] parsedLine = line.split(";");
        userToPasswordMapping.put(parsedLine[0], parsedLine[4]);
        line = bufferedReader.readLine();
      }

      inputStream.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }


  protected void additionalAuthenticationChecks(UserDetails userDetails, U

    // We simply compare the password in the token to the one in the users
```

```
    String presentedPassword = authentication.getCredentials().toString();
    String actualPassword = userToPasswordMapping.get(userDetails.getUsern

    if (!StringUtils.equals(presentedPassword, actualPassword)) {
      throw new BadCredentialsException("Bad credentials");
    }
  }

  protected UserDetails retrieveUser(String username, UsernamePasswordAuth

    // Here we simply defer the loading to the UserDetailsService that was

    UserDetails loadedUser = null;
    try {
      loadedUser = userDetailsService.loadUserByUsername(username);
    } catch (Exception e) {
      throw new AuthenticationServiceException(e.getMessage(), e);
    }
    return loadedUser;
  }

}
```

There's one last bit to configure. By default, the application is configured to log in using the email address. Set the following property to switch that to the *externalId*, meaning the id coming from the external IDM source (*jlennon* in the *users.txt* file for example):

```
security.authentication.use-externalid=true
```

Use the following property to configure case-sensitivity for logins:

```
security.authentication.casesensitive=true
```

Alternatively, you can override the *AuthenticationProvider* that is used (instead of overriding the *configureGlobal*) by implementing the **com.activiti.api.security.AlfrescoAuthenticationProviderOverride** interface.

14.1.1. REST Endpoints security overrides

You can change the default security configuration of the REST API endpoints by simply implementing the **com.activiti.api.security.AlfrescoApiSecurityOverride** interface. By default, the REST API endpoints use the Basic Authentication method.

Similarly, you can override the default cookie+token based security configuration with the regular REST endpoints (those used by the UI) by implementing the **com.activiti.api.security.AlfrescoWebAppSecurityOverride** interface.

14.1.2. UserDetailsService override

If the default **com.activiti.security.UserDetailsService** does not meet the requirement (although it should cover most use cases), you can override the implementation with the **com.activiti.api.security.AlfrescoUserDetailsServiceOverride** interface.

14.2. PasswordEncoder override

By default, Alfresco Activiti uses the **org.springframework.security.crypto.password.StandardPasswordEncoder** for encoding passwords in the database. Note that this is only relevant when using database-backed authentication (so does not hold LDAP/Active Directory). This is an encoder that uses SHA-256 with 1024 iterations and a random salt.

You can override the default setting by implementing the **com.activiti.api.security.AlfrescoPasswordEncoderOverride** interface.

15. REST API

By default, Alfresco Activiti comes with a REST API. It includes an Enterprise equivalent of the Activiti Open Source REST API exposing the generic Activiti Engine operations, and a dedicated set of REST API endpoints for features specific to Alfresco Activiti.

In addtion, there's an internal REST API that is used as REST endpoints by the Javascript UI. Do NOT use this API as the REST API urls might modify the product to use unsupported features. Also, the internal REST API uses a different authentication mechanism tailored towards web browser usage.

15.1. Authentication

The REST API uses Basic Authentication for user authentication. Therefore, you must set all requests with the *Authorization* header.

15.2. Impersonation

The REST API uses authorization rules to determine a user's access control for a process instance or task.

Alternatively, you can impersonate a user with an Admin account to authenticate and set a different user for authorization. To enable this, add the *activiti-user* and *activiti-user-value-type* request headers to the REST API. Where, *activiti-user* should be set to the required user account identifier and *activiti-user-value-type* to the user account identifier type. The header *activiti-user-value-type* can be one of the following values:

- *userIdType*: User's database ID
- *userEmailType*: User's Email address
- *userExternalIdType*: User's ID in an external authentication service such as LDAP or Active Directory

For example, in the *external-form-example* Web application, an Admin account is used for authentication and a different user account to implement authorization.

Note: You must have an Admin role to be able to add the above request headers. In addition, the users should have already been added to Activiti manually, or by synchronization with LDAP or Active Directory.

15.3. Activiti Engine REST API

The Activiti Engine REST API is a supported equivalent of the Activiti Open Source API. This means that all operations described in the Activiti User Guide are available as documented there, except for REST endpoints that are not relevant for the enterprise product (e.g. forms, as they are implemented differently).

This REST API is available on **<your-server-and-context-root>/api/**

For example, fetching process definitions is described in the Activiti User Guide as an HTTP GET on *repository/process-definitions*. This maps to *<your-server-and-context-root>/api/repository/process-definitions*.

Important: Requests on this REST API can only be done using a user that is a *tenant admin* (responsible for one tenant) or a *tenant manager* (responsble for many tenants). This matches the Activiti Engine (Java) API, which is agnostic of user permissions. This means that when calling any of the operations, the tenant identifier must **always be provided in the url**, even if the system does not have multi tenancy (there will always be one tenant in that case):

For example *<your-server-and-context-root>/api/repository/process-definitions?tenantId=1*

15.4. Alfresco Activiti API

This REST API exposes data and operations which are specific to Alfresco Activiti. In contrast to the Activiti Engine REST API it can be called using any user. The following sections describe the various REST API endpoints.

15.4.1. Server Information

To retrieve information about the Alfresco Activiti version:

```
GET api/enterprise/app-version
```

Response:

```
{
   "edition": "Alfresco Activiti Enterprise BPM Suite",
   "majorVersion": "1",
   "revisionVersion": "0",
   "minorVersion": "2",
   "type": "bpmSuite",
}
```

15.4.2. Profile

This operation returns account information for the current user. This is useful to get the name, email, the groups that the user is part of, the user picture, etc.

```
GET api/enterprise/profile
```

Response:

```
{
     "tenantId": 1,
     "firstName": "John",
     "password": null,
     "type": "enterprise",
     "company": null,
     "externalId": null,
     "capabilities": null,
     "tenantPictureId": null,
     "created": "2015-01-08T13:22:36.198+0000",
     "pictureId": null,
     "latestSyncTimeStamp": null,
     "tenantName": "test",
     "lastName": "Doe",
     "id": 1000,
     "lastUpdate": "2015-01-08T13:34:22.273+0000",
     "email": "johndoe@alfresco.com",
     "status": "active",
```

```
    "fullname": "John Doe",
    "groups": [
        {
            "capabilities": null,
            "name": "analytics-users",
            "tenantId": 1,
            "users": null,
            "id": 1,
            "groups": null,
            "externalId": null,
            "status": "active",
            "lastSyncTimeStamp": null,
            "type": 0,
            "parentGroupId": null
        },
        {
            "capabilities": null,
            "name": "Engineering",
            "tenantId": 1,
            "users": null,
            "id": 2000,
            "groups": null,
            "externalId": null,
            "status": "active",
            "lastSyncTimeStamp": null,
            "type": 1,
            "parentGroupId": null
        },
        {
            "capabilities": null,
            "name": "Marketing",
            "tenantId": 1,
            "users": null,
            "id": 2001,
            "groups": null,
            "externalId": null,
            "status": "active",
            "lastSyncTimeStamp": null,
            "type": 1,
            "parentGroupId": null
        }
    ]
}
```


To update user information (first name, last name or email):

```
POST api/enterprise/profile
```

The body of the request should resemble the following text:

```
{
    "firstName" : "John",
    "lastName" : "Doe",
    "email" : "john@alfresco.com",
    "company" : "Alfresco"
}
```

To get the user picture, use following REST call:

```
GET api/enterprise/profile-picture
```

To change this picture, do an HTTP POST to the same url, with the picture as multipart file in the body.

Finally, to change the password:

```
POST api/enterprise/profile-password
```

with a json body that looks like

```
{
    "oldPassword" : "12345",
    "newPassword" : "6789"
}
```

15.4.3. Runtime Apps

When a user logs into Alfresco Activiti, the landing page is displayed containing all the apps that the user is allowed to see and use.

The corresponding REST API request to get this information is

```
GET api/enterprise/runtime-app-definitions
```

Response:

```
{
     "size": 3,
     "total": 3,
     "data": [
          {
               "deploymentId": "26",
               "name": "HR processes",
               "icon": "glyphicon-cloud",
               "description": null,
```

```
                "theme": "theme-6",
                "modelId": 4,
                "id": 1
            },
            {
                "deploymentId": "2501",
                "name": "Sales onboarding",
                "icon": "glyphicon-asterisk",
                "description": "",
                "theme": "theme-1",
                "modelId": 1002,
                "id": 1000
            },
            {
                "deploymentId": "5001",
                "name": "Engineering app",
                "icon": "glyphicon-asterisk",
                "description": "",
                "theme": "theme-1",
                "modelId": 2001,
                "id": 2000
            }
        ],
        "start": 0
}
```


The *id* and *modelId* property of the apps are important here, as they are used in various operations described below.

15.4.4. App Definitions List

To retrieve all App definitions including ones that were not deployed at runtime:

```
GET api/enterprise/models?filter=myApps&modelType=3&sort=modifiedDesc
```

The request parameters

- *filter* : Possible values: *myApps*, *sharedWithMe*, *sharedWithOthers*, *favorite*.
- *modelType* : Must be *3* for App definition models.
- *sort* : *modifiedDesc*, *modifiedAsc*, *nameAsc*, *nameDesc* (default *modifiedDesc*).

15.4.5. App Import And Export

It is possible to export app definitions and import them again. From the REST API point of view, this is useful to bootstrap an environment (for users or continous integration).

To export an app definition, you need the *modelId* from a runtime app or the *id* of an app definition model, and call

```
GET api/enterprise/app-definitions/{modelId}/export
```

This will return a zip file containing the app definition model and all related models (process definitions and forms).

To import an app again, post the zip file as multipart file to

```
POST api/enterprise/app-definitions/import
```

To import an app to an existing app definition to create a new version instead of importing a new app definition, post the zip file as multipart file to

```
POST api/enterprise/app-definitions/{modelId}/import
```

15.4.6. App Publish and Deploy

Before an app model can be used, it needs to be published. This can be done through following call:

```
POST api/enterprise/app-definitions/{modelId}/publish
```

A JSON body is required for the call. You can either use an empty one or the following example:

```
{
    "comment": "",
    "force": false
}
```

To add it to your landing page, *deploy* the published app:

```
POST api/enterprise/runtime-app-definitions
```

Where, *appDefinitions* is an array of IDs, for example:

```
{
    "appDefinitions" : [{"id" : 1}, {"id" : 2}]
}
```

15.4.7. Process Definition Models List

To retrieve a list of process definition models:

```
GET api/enterprise/models?filter=myprocesses&modelType=0&sort=modifiedDesc
```

The request parameters

- *filter* : Possible values: *myprocesses*, *sharedWithMe*, *sharedWithOthers*, *favorite*.
- *modelType* : Must be *0* for process definition models.
- *sort* : Possible values: *modifiedDesc*, *modifiedAsc*, *nameAsc*, *nameDesc* (default *modifiedDesc*).

15.4.8. Model Details and History

Both app definition and process definition models are versioned.

To retrieve details about a particular model (process, form, decision rule or app):

```
GET api/enterprise/models/{modelId}
```

Example response:


```
{
     "createdBy": 1,
     "lastUpdatedBy": 1,
     "lastUpdatedByFullName": " Administrator",
     "name": "aad",
     "id": 2002,
     "referenceId": null,
     "favorite": false,
     "modelType": 0,
     "comment": "",
     "version": 3,
     "lastUpdated": "2015-01-10T16:24:27.893+0000",
     "stencilSet": 0,
     "description": "",
     "createdByFullName": " Administrator",
     "permission": "write",
     "latestVersion": true
}
```


The response shows the current version of the model.

To retrieve a thumbnail of the model:

```
GET api/enterprise/models/{modelId}/thumbnail
```

To get the version information for a model:

```
GET api/enterprise/models/{modelId}/history
```

Example response:


```
{
     "size": 2,
     "total": 2,
     "data": [
          {
               "createdBy": 1,
               "lastUpdatedBy": 1,
               "lastUpdatedByFullName": " Administrator",
               "name": "aad",
               "id": 3000,
               "referenceId": null,
               "favorite": null,
               "modelType": 0,
               "comment": "",
               "version": 2,
               "lastUpdated": "2015-01-10T16:15:50.579+0000",
               "stencilSet": 0,
               "description": "",
               "createdByFullName": " Administrator",
               "permission": null,
               "latestVersion": false
          },
          {
               "createdBy": 1,
               "lastUpdatedBy": 1,
               "lastUpdatedByFullName": " Administrator",
               "name": "aad",
               "id": 2000,
               "referenceId": null,
               "favorite": null,
               "modelType": 0,
               "comment": null,
               "version": 1,
               "lastUpdated": "2015-01-10T16:07:41.831+0000",
               "stencilSet": 0,
               "description": "",
               "createdByFullName": " Administrator",
               "permission": null,
               "latestVersion": false
          }
     ],
     "start": 0
}
```

To get a particular older version:

```
GET api/enterprise/models/{modelId}/history/{modelHistoryId}
```

To create a new model:

```
POST api/enterprise/models/
```

with a json body that looks like:

```
{
    "modelType": 0,
    "name": "My process",
    "description": "This is my favourite process!"
}
```

The modelType property defines the kind of model that is created:

- 0 is a BPMN 2.0 process model
- 1 is a step process model
- 2 is a form model
- 3 is an app model
- 4 is a decision table model

Following properties are optional:

- *stencilSet* : the identifier of the stencilset in case a non-default stencilset needs to be used.

To update the details of a model:

```
PUT api/enterprise/models/{modelId}
```

with a json body that looks like:

```
{
    "name": "New name",
    "description": "New description"
}
```

To favorite a model:

```
PUT api/enterprise/models/{modelId}
```

with as json body:

```
{
    "favorite": true
}
```

To delete a model:

```
DELETE api/enterprise/models/{modelId}
```

To duplicate a model:

```
POST api/enterprise/models/{modelId}/clone
```

with as json body:

```
{
    "name": "Cloned model"
}
```

To convert a step process to a BPMN 2.0 process, add *"modelType" : 0* to the body.

15.4.9. BPMN 2.0 Import and Export

To export a process definition model to a BPMN 2.0 xml file:

```
GET api/enterprise/models/{processModelId}/bpmn20
```

For a previous version of the model:

```
GET api/enterprise/models/{processModelId}/history/{processModelHistoryId}
```

To import a BPMN 2.0 xml file:

```
POST api/enterprise/process-models/import
```

With the BPMN 2.0 xml file in the body as a multipart file and the file as value for the *file* property.

15.4.10. Process Definitions

Get a list of process definitions (visible within the tenant of the user):

```
GET api/enterprise/process-definitions
```

Example response:

```
{
     "size": 5,
     "total": 5,
     "data": [
          {
                              "id": "demoprocess:1:7504",
                              "name": "Demo process",
                              "description": null,
                              "key": "demoprocess",
                              "category": "http://www.activiti.org/test"
                              "version": 1,
                              "deploymentId": "7501",
                              "tenantId": "tenant_1",
                              "hasStartForm": true
                },
          ...
     ],
     "start": 0
}
```

Following parameters are avilable:

- *latest*: A boolean value, indicating that only the latest versions of process definitions must be returned.
- *appDefinitionId*: Returns process definitions that belong to a certain app.

To get the candidate starters associated to a process definition:

```
GET api/enterprise/process-definitions/{processDefinitionId}/identitylinks
```

Where:

- *processDefinitionId*: The ID of the process definition to get the identity links for.
- *family*: Indicates groups or users, depending on the type of identity link.

- *identityId*: The ID of the identity.

To add a candidate starter to a process definition:

```
POST api/enterprise/process-definitions/{processDefinitionId}/identitylink
```

Request body (user):

```
{
    "user" : "1"
}
```

Request body (group):

```
{
    "group" : "1001"
}
```

To delete a candidate starter from a process definition:

```
DELETE api/enterprise/process-definitions/{processDefinitionId}/identityli
```

15.4.11. Start Form

When process definition has a start form (*hasStartForm* is *true* as in the call above), the start form can be retrieved as follows:

```
GET api/enterprise/process-definitions/{process-definition-id}/start-form
```

Example response:


```
{
  "processDefinitionId": "p1:2:2504",
  "processDefinitionName": "p1",
  "processDefinitionKey": "p1",
  "fields": [
    {
      "fieldType": "ContainerRepresentation",
      "id": "container1",
      "name": null,
      "type": "container",
      "value": null,
      "required": false,
      "readOnly": false,
      "overrideId": false,
```

```
      "placeholder": null,
      "optionType": null,
      "hasEmptyValue": null,
      "options": null,
      "restUrl": null,
      "restIdProperty": null,
      "restLabelProperty": null,
      "layout": null,
      "sizeX": 0,
      "sizeY": 0,
      "row": 0,
      "col": 0,
      "visibilityCondition": null,
      "fields": {
        "1": [
          {
            "fieldType": "FormFieldRepresentation",
            "id": "label1",
            "name": "Label1",
            "type": "text",
            "value": null,
            "required": false,
            "readOnly": false,
            "overrideId": false,
            "placeholder": null,
            "optionType": null,
            "hasEmptyValue": null,
            "options": null,
            "restUrl": null,
            "restIdProperty": null,
            "restLabelProperty": null,
            "layout": {
              "row": 0,
              "column": 0,
              "colspan": 1
            },
            "sizeX": 1,
            "sizeY": 1,
            "row": 0,
            "col": 0,
            "visibilityCondition": null
          }
        ],
        "2": [ ]
      }
    },
    {
      "fieldType": "DynamicTableRepresentation",
      "id": "label21",
      "name": "Label 21",
      "type": "dynamic-table",
      "value": null,
      "required": false,
      "readOnly": false,
      "overrideId": false,
```

```
      "placeholder": null,
      "optionType": null,
      "hasEmptyValue": null,
      "options": null,
      "restUrl": null,
      "restIdProperty": null,
      "restLabelProperty": null,
      "layout": {
        "row": 10,
        "column": 0,
        "colspan": 2
      },
      "sizeX": 2,
      "sizeY": 2,
      "row": 10,
      "col": 0,
      "visibilityCondition": null,
      "columnDefinitions": [
        {
          "id": "p2",
          "name": "c2",
          "type": "String",
          "value": null,
          "optionType": null,
          "options": null,
          "restUrl": null,
          "restIdProperty": null,
          "restLabelProperty": null,
          "required": true,
          "editable": true,
          "sortable": true,
          "visible": true
        }
      ]
    }
  ],
  "outcomes": [ ]
}
```


Note: To retrieve field values such as the typeahead field, use the following REST endpoint:

```
GET api/enterprise/process-definitions/{processDefinitionId}/start-form-va
```

This returns a list of form values.

15.4.12. Start Process Instance

```
POST api/enterprise/process-instances
```

With a json body that contains following properties:

- *processDefinitionId* : The process definition identifier. Do not use it with processDefinitionKey.
- *processDefinitionKey* : The process definition key. Do not use it with processDefinitionId.
- *name*: The name to give to the created process instance.
- *values*: A JSON object with the the form field Id and form field values. The Id of the form field is retrieved from the start form call (see above).
- *outcome*: If the start form has outcomes, this is one of those values.
- *variables*: Contains a JSON array of variables. Values and outcomes can't be used with variables.

The response will contain the process instance details including the ID.

Once started, the completed form (if defined) can be fetched using:

```
GET /enterprise/process-instances/{processInstanceId}/start-form
```

15.4.13. Process Instance List

To get the list of process instances:

```
POST api/enterprise/process-instances/query
```

with a json body containing the query parameters. Following parameters are possible:

- processDefinitionId
- appDefinitionId
- state (possible values are *running*, *completed* and *all*)
- sort (possible values are *created-desc*, *created-asc*, *ended-desc*, *ended-asc*)
- start (for paging, default 0)
- size (for paging, default 25)

Example response:

```
{
        "size": 6,
```

```
        "total": 6,
        "start": 0,
        "data":[
                {"id": "2511", "name": "Test step - January 8th 2015", "bu
                ...
        ]
}
```

To get a process instance:

```
GET api/enterprise/process-instances/{processInstanceId}
```

To get diagram for a process instance:

```
GET api/enterprise/process-instances/{processInstanceId}/diagram
```

To delete a Process Instance:

```
DELETE api/enterprise/process-instances/{processInstanceId}
```

To suspend a process instance:

```
PUT api/enterprise/process-instances/{processInstanceId}/suspend
```

To activate a process instance:

```
PUT api/enterprise/process-instances/{processInstanceId}/activate
```

Where, *processinstanceId* is the Id of the process instance.

15.4.14. Get Process Instance Details

```
GET api/enterprise/process-instances/{processInstanceId}
```

15.4.15. Delete a Process Instance

```
DELETE api/enterprise/process-instances/{processInstanceId}
```

15.4.16. Process Instance Audit Log As JSON

If you need the audit log information as a JSON you can use the next URL:

```
GET api/enterprise/process-instances/{process-instance-id}/audit-log
```

Response

200 Ok

Returns a JSON string representing the full audit log for the requested process instance. For example:


```
{
  "processInstanceId": "5",
  "processInstanceName": "myProcessInstance",
  "processDefinitionName": "TEST decision process",
  "processDefinitionVersion": "1",
  "processInstanceStartTime": "Wed Jan 20 16:18:46 EET 2016",
  "processInstanceEndTime": null,
  "processInstanceInitiator": "Mr Activiti",
  "entries": [
    {
      "index": 1,
      "type": "startForm",
      "timestamp": "Wed Jan 20 16:18:46 EET 2016",
      "selectedOutcome": null,
      "formData": [
        {
          "fieldName": "Text1",
          "fieldId": "text1",
          "value": "TEST"
        }
      ],
      "taskName": null,
      "taskAssignee": null,
      "activityId": null,
      "activityName": null,
      "activityType": null
    },
    {
      "index": 2,
      "type": "activityExecuted",
      "timestamp": "Wed Jan 20 16:18:46 EET 2016",
      "selectedOutcome": null,
      "formData": [],
      "taskName": null,
      "taskAssignee": null,
      "activityId": "startEvent1",
      "activityName": "",
      "activityType": "startEvent"
    },
    {
      "index": 3,
      "type": "activityExecuted",
      "timestamp": "Wed Jan 20 16:18:47 EET 2016",
```

```
      "selectedOutcome": null,
      "formData": [],
      "taskName": null,
      "taskAssignee": null,
      "activityId": "sid-15E18ED8-252F-4A24-9E93-68F53FE28535",
      "activityName": "",
      "activityType": "serviceTask"
    },
    {
      "index": 4,
      "type": "activityExecuted",
      "timestamp": "Wed Jan 20 16:18:48 EET 2016",
      "selectedOutcome": null,
      "formData": [],
      "taskName": null,
      "taskAssignee": null,
      "activityId": "sid-001FD811-C171-40E3-9C62-602621672022",
      "activityName": "",
      "activityType": "userTask"
    },
    {
      "index": 5,
      "type": "taskCreated",
      "timestamp": "Wed Jan 20 16:18:48 EET 2016",
      "selectedOutcome": null,
      "formData": [],
      "taskName": null,
      "taskAssignee": "Mr Activiti",
      "activityId": null,
      "activityName": null,
      "activityType": null
    }
  ],
  "decisionInfo": {
    "calculatedValues": [
      {
        "name": "outputVariable1",
        "value": "1.0"
      }
    ],
    "appliedRules": [
      {
        "title": "Rule 1 (TEST Decision Table 1)",
        "expressions": [
          {
            "type": "CONDITION",
            "variable": "text1",
            "value": "== 'TEST'"
          },
          {
            "type": "OUTCOME",
            "variable": "outputVariable1",
            "value": "1"
          }
        ]
```

```
      }
    ]
  }
}
```

15.4.17. Process instance variables

A process instance can have several variables.

To get process instance variables:

```
GET api/enterprise/process-instances/{processInstanceId}/variables
```

Where, *processInstanceId* is the Id of the process instance.

To create process instance variables:

```
POST api/enterprise/process-instances/{processInstanceId}/variables
```

To update existing variables in a process instance:

```
PUT api/enterprise/process-instances/{processInstanceId}/variables
```

Example response:

```
{
     "name":"myVariable",
     "type":"string",
     "value":"myValue"
}
```

Where:

- **name** - Name of the variable
- **type** - Type of variable, such as string
- **value** - Value of the variable

To update a single variable in a process instance:

```
PUT api/enterprise/process-instances/{processInstanceId}/variables/{variab
```

To get a single variable in a process instance:

```
GET api/enterprise/process-instances/{processInstanceId}/variables/{variab
```

To get all process instance variables:

```
GET api/enterprise/process-instances/{processInstanceId}/variables
```

To get a specific process instance variable:

```
GET api/enterprise/process-instances/{processInstanceId}/variables/{variab
```

To delete a specific process instance variable:

```
DELETE api/enterprise/process-instances/{processInstanceId}/variables/{var
```

15.4.18. Process Instance Identity links

To create an identity link of a process instance:

```
POST api/enterprise/process-instances/{processInstanceId}/identitylinks
```

Example request:

```
{
     "user": "1",
     "type": "customType"
}
```

Get identity links of a process instance:

```
GET api/enterprise/process-instances/{processInstanceId}/identitylinks
```

Get identity links by family type of a process instance:

```
GET api/enterprise/process-instances/{processInstanceId}/identitylinks/{fa
```

Where, *Family* should contain users or groups, depending on the identity you want to link.

To get involved people in a process instance:

```
GET api/enterprise/process-instances/{processInstanceId}/identitylinks
```

You can get identity links for either user or groups. For example:

```
GET api/enterprise/process-instances/{processInstanceId}/identitylinks/use

GET api/enterprise/process-instances/{processInstanceId}/identitylinks/gro
```

15.4.19. Task List

```
POST api/enterprise/tasks/query
```

includes a JSON body containing the query parameters. Following parameters are available:

- appDefinitionId
- processInstanceId
- processDefinitionId
- text (the task name will be filtered with this, using *like* semantics : %text%)
- assignment
    - assignee : where the current user is the assignee
    - candidate: where the current user is a task candidate
    - group_x: where the task is assigned to a group where the current user is a member of. The groups can be fetched through the profile REST endpoint
    - no value: where the current user is involved
- state (*completed* or *active*)
- sort (possible values are *created-desc*, *created-asc*, *due-desc*, *due-asc*)
- start (for paging, default 0)
- size (for paging, default 25)

Example response:

```
{
        "size": 6,
        "total": 6,
```

```
        "start": 0,
        "data":[
                {
                        "id": "2524",
                        "name": "Task",
                        "description": null,
                        "category": null,
                        "assignee":{"id": 1, "firstName": null, "lastName"
                        "created": "2015-01-08T10:58:37.193+0000",
                        "dueDate": null,
                        "endDate": null,
                        "duration": null,
                        "priority": 50,
                        "processInstanceId": "2511",
                        "processDefinitionId": "teststep:3:29",
                        "processDefinitionName": "Test step",
                        "processDefinitionDescription": null,
                        "processDefinitionKey": "teststep",
                        "processDefinitionCategory": "http://www.activiti.
                        "processDefinitionVersion": 3,
                        "processDefinitionDeploymentId": "26",
                        "formKey": "5"
                }
                ...
        ]
}
```


15.4.20. Task Details

```
GET api/enterprise/tasks/{taskId}
```

Response is similar to the list response.

15.4.21. Task Form

```
GET api/enterprise/task-forms/{taskId}
```

The response is similar to the response from the Start Form.

To retrieve Form field values that are populated through a REST backend:

```
GET api/enterprise/task-forms/{taskId}/form-values/{field}
```

Which returns a list of form field values

To complete a Task form:

```
POST api/enterprise/task-forms/{taskId}
```

with a json body that contains:

- *values*: A json object with the the form field ID - form field values. The Id of the form field is retrieved from the start form call (see above).
- *outcome*: Retrieves outcome values if defined in the Start form.

To save a Task form:

```
POST api/enterprise/task-forms/{taskid}/save-form
```

Example response:

```
{

"values": {"formtextfield":"snicker doodle"},
"numberfield":"6",
"radiobutton":"red"

}
```

Where the json body contains:

- *values* : A json object with the the form field ID - form field values. The Id of the form field is retrieved from the Start Form call (see above).

15.4.22. Create a Standalone Task

To create a task (for the user in the authentication credentials) that is not associated with a process instance:

```
POST api/enterprise/tasks
```

with a json body that contains the following properties:

- name
- description

15.4.23. Task Actions

To update the details of a task:

```
PUT api/enterprise/tasks/{taskId}
```

with a json body that can contain *name*, *description* and *dueDate* (ISO 8601 string)

For example:

Example request:

```
{
  "name" : "IchangedTaskName",
  "description" : "description-updated",
  "dueDate" : "2015-01-11T22:59:59.000Z",
  "priority":10,
  "formKey": "100"
}
```

To delegate a task:

```
PUT api/enterprise/tasks/{taskId}/action/delegate
```

Example request:

```
{
     "userId": "1000"
}
```

To resolve a task:

```
PUT api/enterprise/tasks/{taskId}/action/resolve
```

To complete a task (standalone or without a task form) (**Note**: No json body needed!):

```
PUT api/enterprise/tasks/{taskId}/action/complete
```

To claim a task (in case the task is assigned to a group):

```
PUT api/enterprise/tasks/{taskId}/action/claim
```

No json body needed. The task will be claimed by the user in the authentication credentials.

To assign a task to a user:

```
PUT api/enterprise/tasks/{taskId}/action/assign
```

with a json body that contains the *assignee* property set to the *ID* of a user.

To involve a user with a task:

```
PUT api/enterprise/tasks/{taskId}/action/involve
```

with a json body that contains the *userId* property set to the *ID* of a user.

To remove an involved user from a task:

```
PUT api/enterprise/tasks/{taskId}/action/remove-involved
```

with a json body that contains the *userId* property set to the *ID* of a user.

To attach a form to a task:

```
PUT api/enterprise/tasks/{taskId}/action/attach-form
```

with a json body that contains the *formId* property set to the the *ID* of a form.

To attach a form to a task:

```
DELETE api/enterprise/tasks/{taskId}/action/remove-form
```

15.4.24. Task Variables

To create new task variables:

```
POST api/enterprise/tasks/{taskId}/variables
```

To get all task variables:

```
GET api/enterprise/tasks/{taskId}/variables
```

To get a task variable by name:

```
GET api/enterprise/tasks/{taskId}/variables/{variableName}
```

To update an existing task variable:

```
PUT api/enterprise/tasks/{taskId}/variables/{variableName}
```

Example response:

```
{
     "name":"myVariable",
     "scope":"local",
     "type":"string",
     "value":"myValue"
}
```

Where:

- **name** - Name of the variable.
- **scope** - Global or local. If global is provided, then the variable will be a process-instance variable.
- **type** - Type of variable, such as string.
- **value** - Value of the variable.

To delete a task variable:

```
DELETE api/enterprise/tasks/{taskId}/variables/{variableName}
```

To delete all task variables:

```
DELETE api/enterprise/tasks/{taskId}/variables
```

Where, *taskId* is the ID of the task.

15.4.25. Task Identity links

To get all identity links for a task:

```
GET api/enterprise/tasks/{taskId}/identitylinks
```

To create an identity link on a task:

```
POST api/enterprise/tasks/{taskId}/identitylinks
```

Example response:

```
{
     "user": "1",
     "type": "customType"
}
```

To get a single identity link on a task:

```
GET api/enterprise/tasks/{taskId}/identitylinks/{family}/{identityId}/{typ
```

To delete an identity link on a task:

```
DELETE api/enterprise/tasks/{taskId}/identitylinks/{family}/{identityId}/{
```

Where:

- *taskId*: The ID of the task.
- *family*: Indicates either groups or users, depending on the type of identity.
- *identityId*: The ID of the identity.
- *type*: The type of identity link.

15.4.26. User Task Filters

Custom task queries can be saved as a user task filter. To get the list of task filters for the authenticated user:

```
GET api/enterprise/filters/tasks
```

with an option request parameter *appId* to limit the results to a specific app.

To get a specific user task filter:

```
GET api/enterprise/filters/tasks/{userFilterId}
```

To create a new user task filter:

```
POST api/enterprise/filters/tasks
```

with a json body that contains following properties:

- *name* : Name of the filter.

- *appId* : App ID where the filter can be used.
- *icon* : Path of the icon image.
- *filter*
    - *sort* : Possible values: created-desc, created-asc, due-desc, due-asc.
    - *state* : Open, completed.
    - *assignment* : Involved, assignee, or candidate.

To update a user task filter:

```
PUT api/enterprise/filters/tasks/{userFilterId}
```

with a json body that contains following properties:

- *name* : Name of the filter
- *appId* : App ID where the filter can be used.
- *icon* : Path of the icon image.
- *filter*
    - *sort* : Created-desc, created-asc, due-desc, due-asc.
    - *state* : Open, completed.
    - *assignment* : Involved, assignee, or candidate

To delete a user task filter:

```
DELETE api/enterprise/filters/tasks/{userFilterId}
```

To order the list of user task filters:

```
PUT api/enterprise/filters/tasks
```

with a json body that contains following properties:

- *order* : Array of user task filter IDs.
- *appId* : App ID.

To get a list of user process instance filters

```
GET api/enterprise/filters/processes
```

with an option request parameter *appId* to limit the results to a specific app.

To get a specific user process instance task filter:

```
GET api/enterprise/filters/processes/{userFilterId}
```

To create a user process instance task filter:

```
PUT  api/enterprise/filters/processes
```

with a json body that contains following properties:

- *name* : Name of the filter.
- *appId* : App ID where the filter can be used.
- *icon* : Path of the icon image.
- *filter*
    - *sort* : Created-desc, created-asc.
    - *state* : Running, completed, or all.

To update a user process instance task filter:

```
PUT  api/enterprise/filters/processes/{userFilterId}
```

with a json body that contains following properties:

- *name* : Name of the filter.
- *appId* : App ID, where the filter can be used.
- *icon* : Path of the icon image.
- *filter*
    - *sort* : Possible values: created-desc, created-asc.
    - *state* : Running, completed, or all.

To delete a user process instance task filter

```
DELETE  api/enterprise/filters/processes/{userFilterId}
```

15.4.27. Comments

Comments can be added to a process instance or a task.

To get the list of comments:

```
GET api/enterprise/process-instances/{processInstanceId}/comments
```

```
GET api/enterprise/tasks/{taskId}/comments
```

To create a comments:

```
POST api/enterprise/process-instances/{processInstanceId}/comments
```

```
POST api/enterprise/tasks/{taskId}/comments
```

with in the json body one property called *message*, with a value that is the comment text.

15.4.28. Checklists

You can add checklists to a task for tracking purposes.

To get a checklist:

```
GET api/enterprise/tasks/{taskId}/checklist
```

To create a checklist:

```
POST api/enterprise/tasks/{taskId}/checklist
```

Example request body:

```
{
    "assignee": {"id": 1001},
    "name": "mySubtask",
    "parentTaskId": "20086"
}
```

To change the order of the items on a checklist:

```
PUT api/enterprise/tasks/{taskId}/checklist
```

with a json body that contains an ordered list of checklist items ids:

- *order* : Array of checklist item ids

15.4.29. Task Audit Info (as JSON)

To obtain the audit information for a specific task in JSON format, use the following URL:

```
GET api/enterprise/tasks/{taskId}/audit
```

Response

200 Ok

If everything works as expected and the task is accessible to the current user, then the response will be as follows:

```
{
  "taskId": "18",
  "taskName": null,
  "processInstanceId": "5",
  "processDefinitionName": "TEST decision process",
  "processDefinitionVersion": 1,
  "assignee": "Mr Activiti",
  "startTime": "Wed Jan 20 22:03:05 EET 2016",
  "endTime": "Wed Jan 20 22:03:09 EET 2016",
  "formData": [],
  "selectedOutcome": null,
  "comments": []
}
```

15.5. History

This section covers the examples for querying historic process instances and task instances in the Activiti API. You can query for historic process instances and tasks to get information about ongoing and past process instances, or tasks.

15.5.1. Historic process instance queries

To run a historic process instance query:

```
POST api/enterprise/historic-process-instances/query
```

To run a historic task instance query:

```
POST api/enterprise/historic-tasks/query
```

15.5.2. Get historic process instances

The following table lists the request parameters to be used in the JSON body POST. For example, to filter historic process instances that completed before the given date (*startedBefore*):

```
POST api/enterprise/historic-process-instances/query
```

With a JSON body request:

```
{
"startedBefore":"2016-06-16",
}
```

Example response:

```
{
"size": 25,
"total": 200,
"start": 0,
  "data": [
    {
      "id": "2596",
      "name": "Date format example - June 7th 2016",
      "businessKey": null,
      "processDefinitionId": "dateformatexample:1:2588",
      "tenantId": "tenant_1",
      "started": "2016-06-07T14:18:34.433+0000",
      "ended": null,
      "startedBy": {
        "id": 1,
        "firstName": null,
        "lastName": "Administrator",
        "email": "admin@app.activiti.com"
      },
{
"id": "2596",
. . .
```

Where, *size is the size of the page or number of items per page. By default, the value is 25. * *start* is the page to start on. Pages are counted from 0-N. By default, the value is 0, which means 0 will be the first page.

processInstanceId	An ID of the historic process instance.
processDefinitionKey	The process definition key of the historic process instance.
processDefinitionId	The process definition id of the historic process instance.
businessKey	The business key of the historic process instance.
involvedUser	An involved user of the historic process instance. Where, *InvolvedUser* is the ID of the user.
finished	Indicates if the historic process instance is complete. Where, the value may only be *True*, as the default values are *True* or *False*.
superProcessInstanceId	An optional parent process id of the historic process instance.
excludeSubprocesses	Returns only historic process instances which aren't sub-processes.
finishedAfter	Returns historic process instances that finished after the given date. The date is displayed in *yyyy-MM-ddTHH:MM:SS* format.
finishedBefore	Returns historic process instances that finished before the given date. The date is displayed in *yyyy-MM-ddTHH:MM:SS* format.
startedAfter	Returns historic process instances that were started after the given date. The date is displayed in *yyyy-MM-ddTHH:MM:SS* format.
startedBefore	Returns historic process instances that were started before the given date. The date is displayed in *yyyy-*

	MM-ddTHH:MM:SS format.
startedBy	Returns only historic process instances that were started by the selected user.
includeProcessVariables	Indicates if the historic process instance variables should be returned.
tenantId	Returns instances with the given *tenantId*.
tenantIdLike	Returns instances with a *tenantId* like the given value.
withoutTenantId	If true, only returns instances without a *tenantId* set. If false, the *withoutTenantId* parameter is ignored.

15.5.3. Get historic task instances

The following table lists the request parameters that can be used in the JSON body POST. For example, in case of *taskCompletedAfter*:

```
----
POST api/enterprise/historic-tasks/query
----
```

```
With a json body request:
{
"taskCompletedAfter":"2016-06-16",
"size":50,
"start":0
}
```

Example response:

```
{
  "size": 4,
  "total": 4,
  "start": 0,
  "data": [
    {
      "id": "7507",
      "name": "my task",
      "assignee": {
        "id": 1000,
        "firstName": "Homer",
```

```
        "lastName": "Simpson",
        "email": "homer.simpson@gmail.com"
      },
      "created": "2016-06-17T15:14:26.938+0000",
      "dueDate": null,
      "endDate": "2016-06-17T16:09:39.197+0000",
      "duration": 3312259,
      "priority": 50,
. . .
```

taskId	An ID of the historic task instance.
processInstanceId	The process instance id of the historic task instance.
processDefinitionKey	The process definition key of the historic task instance.
processDefinitionKeyLike	The process definition key of the historic task instance, which matches the given value.
processDefinitionId	The process definition id of the historic task instance.
processDefinitionName	The process definition name of the historic task instance.
processDefinitionNameLike	The process definition name of the historic task instance, which matches the given value.
processBusinessKey	The process instance business key of the historic task instance.
processBusinessKeyLike	The process instance business key of the historic task instance that matches the given value.
executionId	The execution id of the historic task instance.
taskDefinitionKey	The task definition key for tasks part of a process
taskName	The task name of the historic task instance.

taskNameLike	The task name with like operator for the historic task instance.
taskDescription	The task description of the historic task instance
taskDescriptionLike	The task description with like operator for the historic task instance.
taskDefinitionKey	The task identifier from the process definition for the historic task instance.
taskDeleteReason	The task delete reason of the historic task instance.
taskDeleteReasonLike	The task delete reason with like operator for the historic task instance.
taskAssignee	The assignee of the historic task instance.
taskAssigneeLike	The assignee with like operator for the historic task instance.
taskOwner	The owner of the historic task instance.
taskOwnerLike	The owner with like operator for the historic task instance.
taskInvolvedUser	An involved user of the historic task instance. Where, *InvolvedUser* is the User ID.
taskPriority	The priority of the historic task instance.
finished	Indicates if the historic task instance is complete.
processFinished	Indicates if the process instance of the historic task instance is finished.
parentTaskId	An optional parent task ID of the historic task instance.

dueDate	Returns only historic task instances that have a due date equal to this date.
dueDateAfter	Returns only historic task instances that have a due date after this date.
dueDateBefore	Returns only historic task instances that have a due date before this date.
withoutDueDate	Returns only historic task instances that have no due-date. When false value is provided, this parameter is ignored.
taskCompletedOn	Returns only historic task instances that have been completed on this date.
taskCompletedAfter	Returns only historic task instances that have been completed after this date.
taskCompletedBefore	Return only historic task instances that have been completed before this date.
taskCreatedOn	Returns only historic task instances that were created on this date.
taskCreatedBefore	Returns only historic task instances that were created before this date.
taskCreatedAfter	Returns only historic task instances that were created after this date.
includeTaskLocalVariables	Indicates if the historic task instance local variables should be returned.
includeProcessVariables	Indicates if the historic task instance global variables should be returned.
tenantId	Returns historic task instances with the given tenantId.

tenantIdLike	Returns historic task instances with a tenantId like the given value.
withoutTenantId	If true, only returns historic task instances without a tenantId set. If false, *withoutTenantId* is ignored.

15.5.4. User and Group lists

A common use case is when a user wants to select another user or group, for example, when assigning a task.

To retrieve users:

```
GET api/enterprise/users
```

Use the following parameters:

- *filter*: Filters by the user's first and last name.
- *email*: Retrieves users by email
- *externalId*: Retrieves users by their external ID.
- *externalIdCaseInsensitive*: Retrieves users by external ID, ignoring case.
- *externalId*: Retrieves users by their external ID (set by the LDAP sync, if used)
- *excludeTaskId*: Excludes users that are already part of this task.
- *excludeProcessId*: Excludes users that are already part of this process instance.

Example response:

```
{
        "size": 2,
        "total": 2,
        "start": 0,
        "data": [
                {
                        "id": 1,
                        "firstName": null,
                        "lastName": "Administrator",
                        "email": "admin@app.activiti.com"
                },
                {
                        "id": 1000,
                        "firstName": "John",
                        "lastName": "Doe",
```

```
                        "email": "johndoe@alfresco.com"
                }
        ]
}
```

To retrieve a picture of a user:

```
GET api/enterprise/users/{userId}/picture
```

To retrieve groups:

```
GET api/enterprise/groups
```

with optional parameter *filter* that filters by group name.

Additional options:

- *externalId*: Retrieves a group by their external ID.
- *externalIdCaseInsensitive*: Retrieves a group by their external ID, ignoring case.

Example response:

```
{
     "size": 2,
     "total": 2,
     "data": [
          {
               "externalId": null,
               "name": "Engineering",
               "id": 2000
          },
          {
               "externalId": null,
               "name": "Marketing",
               "id": 2001
          }
     ],
     "start": 0
}
```

Get the users for a given group:

```
GET api/enterprise/groups/{groupId}/users
```

Example response:

```
{
     "size": 3,
     "total": 3,
     "data": [
          {
               "email": "john@alfresco.com",
               "lastName": "Test",
               "firstName": "John",
               "id": 10
          },
          {
               "email": "mary@alfresco.com",
               "lastName": "Test",
               "firstName": "Mary",
               "id": 8
          },
          {
               "email": "patrick@alfresco.com",
               "lastName": "Test",
               "firstName": "Patrick",
               "id": 9
          }
     ],
     "start": 0
}
```


With a json body that contains:

- *order* : An array of user task filter IDs

15.5.5. Content

Content such as documents and other files can be attached to process instances and tasks.

To retrieve the content attached to a process instance:

```
GET api/enterprise/process-instances/{processInstanceId}/content
```

By default, this will return all content: The related content (for example content uploaded via the UI in the "related content" section of the task detail page) and the field content (content uploaded as part of a form).

To only return the related content, add *?isRelatedContent=true* to the url. Similarly, add *?isRelatedContent=false* when the return response should include only field content.

Similarly, for a task:

```
GET api/enterprise/tasks/{taskId}/content
```

By default, this will return all content: The related content (for example content uploaded via the UI in the "related content" section of the task detail page) and the field content (content uploaded as part of a form).

To only return the 'related content', add *?isRelatedContent=true'* to the url. Similarly, add *?isRelatedContent=false* when the return response should include only field content.

Example response:


```
{
  "size": 5,
  "total": 5,
  "start": 0,
  "data": [
    {
      "id": 4000,
      "name": "tasks.PNG",
      "created": "2015-01-01T01:01:01.000+0000",
      "createdBy": {
        "id": 1,
        "firstName": "null",
        "lastName": "Admin",
        "email": "admin@app.activiti.com",
        "pictureId": 5
      },
      "relatedContent": true,
      "contentAvailable": true,
      "link": false,
      "mimeType": "image/png",
      "simpleType": "image",
      "previewStatus": "queued",
      "thumbnailStatus": "queued"
    }
        ]
}
```


To get content metadata:

```
GET api/enterprise/content/{contentId}
```

To delete content:

```
DELETE api/enterprise/content/{contentId}
```

To get the actual bytes for content:

```
GET api/enterprise/content/{contentId}/raw
```

To upload content to a process instance:

```
POST api/enterprise/process-instances/{processInstanceId}/raw-content
```

where the body contains a *multipart file*. Add the *isRelatedContent* parameter to the url to set whether the content is 'related' or not. For a process instance, this currently won't have any influence on what is visible in the UI. Note that the default value for this parameter is *false*.

To upload content to a task:

```
POST api/enterprise/process-instances/{taskId}/raw-content
```

where the body contains a *multipart file*. Add the *isRelatedContent* parameter to the url to set whether the content is 'related' or not. If *true*, the content will show up in the "related content" section of the task details. Note that the default value for this parameter is *false*.

To relate content (eg from Alfresco) to a process instance:

```
POST api/enterprise/process-instances/{processInstanceId}/content
```

where the json body contains following properties:

- name
- link (boolean)
- source
- sourceId
- mimeType
- linkUrl

Add the *isRelatedContent* parameter to the url to set whether the content is related or not. If *true*, the content will show up in the "related content" section of the task details. Note that the default value for this parameter is true (different from the call above with regular content!).

Example body (from Alfresco OnPremise):

```
{
   "name":"Image.png",
   "link":true,
   "source":"alfresco-1",
   "sourceId":"30358280-88de-436e-9d4d-8baa9dc44f17@swsdp",
   "mimeType":"image/png"
}
```

To upload content for a task:

```
POST api/enterprise/process-instances/{taskId}/content
```

Where the json body contains following properties:

- name
- link (boolean)
- source
- sourceId
- mimeType
- linkUrl

In case of a start form with content fields, there is no task or process instance to relate to.

Following REST endpoints can be used:

```
POST api/enterprise/content/raw
```

15.5.6. Thumbnails

To retrieve the thumbnail of a certain piece of content:

```
GET api/enterprise/content/{contentId}/rendition/thumbnail
```

15.5.7. Identity Management

These are operations to manage tenants, groups and users. This is useful for example to bootstrap environments with the correct identity data.

Tenants

Following REST endpoints are **only available for users that are either a tenand admin or a tenant manager**. The tenant capability also depends for some operations on the type of license (multi-tenant license or not).

Get all tenants (tenant manager only):

```
GET api/enterprise/admin/tenants
```

Create a new tenant (tenant manager only):

```
POST api/enterprise/admin/tenants
```

the json body of this post contains two properties: *name* and *active* (boolean).

Update a tenant:

```
PUT api/enterprise/admin/tenants/{tenantId}
```

the json body of this post contains two properties: *name* and *active* (boolean).

Get tenant details:

```
GET api/enterprise/admin/tenants/{tenantId}
```

Delete a tenant

```
DELETE api/enterprise/admin/tenants/{tenantId}
```

Get tenant events:

```
GET api/enterprise/admin/tenants/{tenantId}/events
```

Get tenant logo:

```
GET api/enterprise/admin/tenants/{tenantId}/logo
```

Change tenant logo:

```
POST api/enterprise/admin/tenants/{tenantId}/logo
```

where the body is a multi part file.

Users

Following REST endpoints are **only available for users that are either a tenand admin or a tenant manager**.

Get a list of users:

```
GET api/enterprise/admin/users
```

with parameters

- *filter* : Filters by user name.
- *status* : Possible values are *pending*, *inactive*, *active*, *deleted*.
- *sort* : Possible values are *createdAsc*, *createdDesc*, *emailAsc* or *emailDesc* (default *createdAsc*).
- *start* : Used for paging.
- *size* : Use for paging.

To create a new user:

```
POST api/enterprise/admin/users
```

with a json body that **must** have following properties:

- email
- firstName
- lastName
- password
- status (possible values are *pending*, *inactive*, *active*, *deleted*)
- type (enterprise or trial. Best to set this to enterprise)
- tenantId

Update user details:

```
PUT api/enterprise/admin/users/{userId}
```

with a json body containing *email*, *firstName* and *lastName*

Update user password:

```
PUT api/enterprise/admin/users
```

with a json body like

```
{
        "users" : [1098, 2045, 3049]
        "password" : "123"
}
```

Note that the *users* property is an array of user ids. This allows for bulk changes.

Update user status:

```
PUT api/enterprise/admin/users
```

with a json body like

```
{
        "users" : [1098, 2045, 3049]
        "status" : "inactive"
}
```

Note that the *users* property is an array of user ids. This allows for bulk changes.

Update user tenant id (only possible for _tenant manager):

```
PUT api/enterprise/admin/users
```

with a json body like

```
{
        "users" : [1098, 2045, 3049]
        "tenantId" : 1073
}
```

Note that the *users* property is an array of user ids. This allows for bulk changes.

Groups

Following REST endpoints are **only available for users that are either a tenand admin or a tenant manager**.

Internally, there are two types of groups:

- Functional groups: Map to organizational units.
- System groups: Provide users capabilities. When you assign a capability to a group, every member of that group is assigned with the capability.

Get all groups:

```
GET api/enterprise/admin/groups
```

Optional parameters:

- *tenantId* : Useful to a Tenant Manager user
- *functional* (boolean): Only return functional groups if true

Get group details:

```
GET api/enterprise/admin/groups/{groupId}
```

Example response:


```
{
     "capabilities": [{
          "name": "access-reports",
          "id": 1
     }],
     "name": "analytics-users",
     "tenantId": 1,
     "users": [
          {
               "tenantId": 1,
               "firstName": null,
               "password": null,
               "type": "enterprise",
               "company": null,
               "externalId": null,
               "capabilities": null,
               "tenantPictureId": null,
               "created": "2015-01-08T08:30:25.164+0000",
               "pictureId": null,
               "latestSyncTimeStamp": null,
               "tenantName": null,
               "lastName": "Administrator",
               "id": 1,
```

```
                "lastUpdate": "2015-01-08T08:30:25.164+0000",
                "email": "admin@app.activiti.com",
                "fullname": " Administrator",
                "groups": null
            },
            {
                "tenantId": 1,
                "firstName": "John",
                "password": null,
                "type": "enterprise",
                "company": null,
                "externalId": null,
                "capabilities": null,
                "tenantPictureId": null,
                "created": "2015-01-08T13:22:36.198+0000",
                "pictureId": null,
                "latestSyncTimeStamp": null,
                "tenantName": null,
                "lastName": "Doe",
                "id": 1000,
                "lastUpdate": "2015-01-08T13:34:22.273+0000",
                "email": "johndoe@alfresco.com",
                "fullname": "John Doe",
                "groups": null
            }
        ],
        "id": 1,
        "groups": [],
        "externalId": null,
        "status": "active",
        "lastSyncTimeStamp": null,
        "type": 0,
        "parentGroupId": null
}
```


Use the optional request parameter *includeAllUsers* (boolean value, by default true) to avoid getting all the users at once (not ideal if there are many users).

Use the following call:

```
GET api/enterprise/admin/groups/{groupId}/users?page=2&pageSize=20
```

Create new group:

```
POST api/enterprise/admin/groups
```

Where the json body contains following properties:

- name

- tenantId
- type (0 for system group, 1 for functional group)
- parentGroupId (only possible for functional groups. System groups can't be nested)

Update a group:

```
PUT api/enterprise/admin/groups/{groupId}
```

Only the *name* property can be in the json body.

Delete a group:

```
DELETE api/enterprise/admin/groups/{groupId}
```

Add a user to a group:

```
POST api/enterprise/admin/groups/{groupId}/members/{userId}
```

Delete a user from a group:

```
DELETE api/enterprise/admin/groups/{groupId}/members/{userId}
```

Get the list of possible capabilities for a system group:

```
GET api/enterprise/admin/groups/{groupId}/potential-capabilities
```

Add a capability from previous list to the group:

```
POST api/enterprise/admin/groups/{groupId}/capabilities
```

where the json body contains one property *capabilities* that is an array of strings.

Remove a capability from a group:

```
DELETE api/enterprise/admin/groups/{groupId}/capabilities/{groupCapability
```

Alfresco repositories

A tenant administrator can configure one or more Alfresco repositories to use when working with content. To retrieve the Alfresco repositories configured for the tenant of the user used to do the request:

```
GET api/enterprise/profile/accounts/alfresco
```

which returns something like:


```
{
     "size": 2,
     "total": 2,
     "data": [
          {
               "name": "TS",
               "tenantId": 1,
               "id": 1,
               "accountUsername": "jbarrez",
               "created": "2015-03-26T14:24:35.506+0000",
               "shareUrl": "http://ts.alfresco.com/share",
               "lastUpdated": "2015-03-26T15:37:21.174+0000",
               "repositoryUrl": "http://ts.alfresco.com/alfresco",
               "alfrescoTenantId": ""
          },
          {
               "name": "TsTest",
               "tenantId": 1,
               "id": 1000,
               "accountUsername": "jbarrez",
               "created": "2015-03-26T15:37:36.448+0000",
               "shareUrl": "http://tstest.alfresco.com/share",
               "lastUpdated": "2015-03-26T15:37:36.448+0000",
               "repositoryUrl": "http://tstest.alfresco.com/alfresco",
               "alfrescoTenantId": ""
          }
     ],
     "start": 0
}
```


16. Disclaimer

While Alfresco has used commercially reasonable efforts to ensure the accuracy of this documentation, Alfresco assumes no responsibility for the accuracy, completeness, or usefulness of any information or for damages resulting from the procedures provided.

Furthermore, this documentation is supplied "as is" without guarantee or warranty, expressed or implied, including without limitation, any warranty of fitness for a specific purpose.