Version

3.4

CMDBuild®

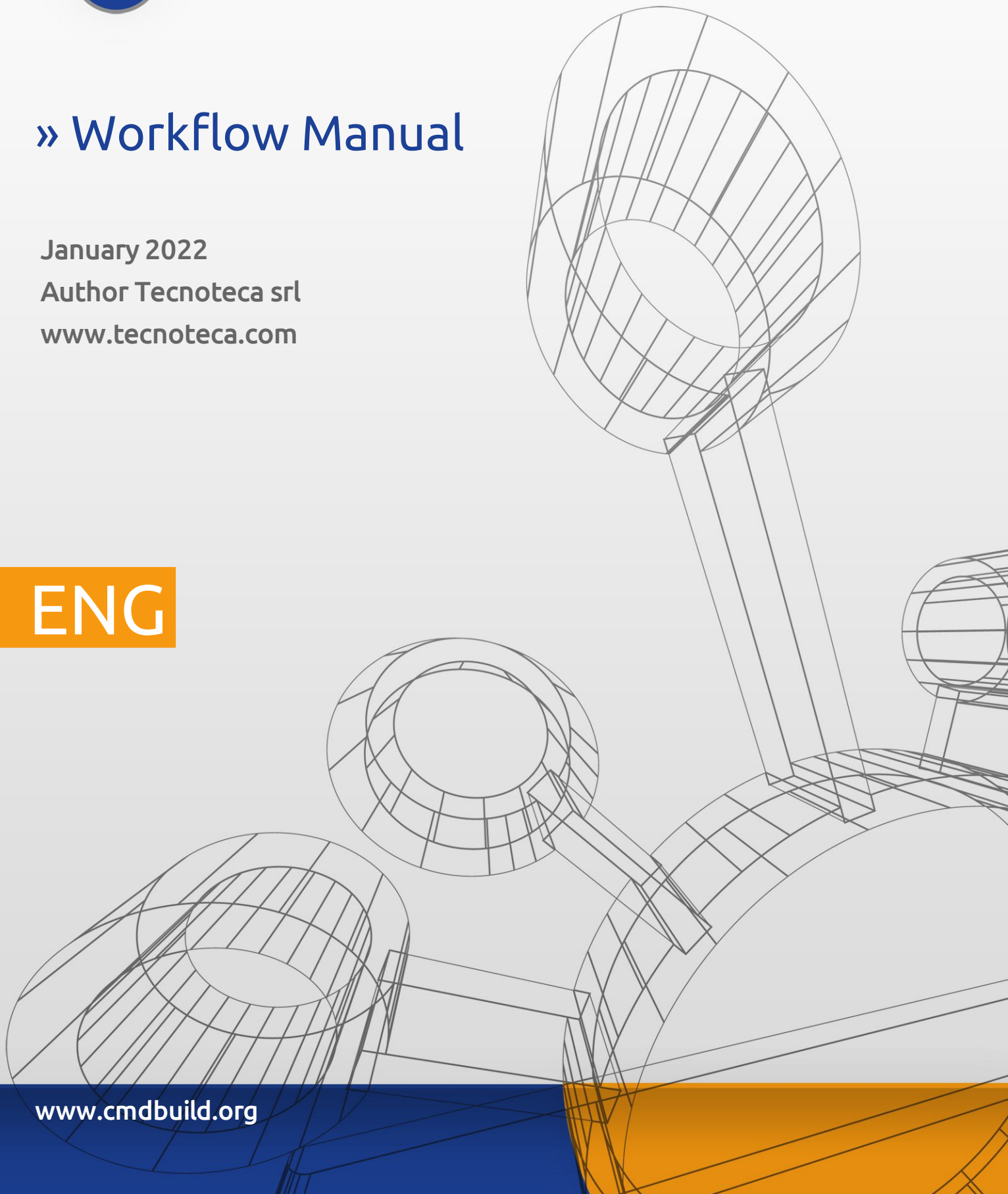
» Workflow Manual

January 2022

Author Tecnoteca srl

www.tecnoteca.com

ENG



CMDBuild ® uses many great technologies from the open source community:
PostgreSQL, Apache, Tomcat, Eclipse, Ext JS, JasperStudio, Enhydra Shark, TWE, OCS Inventory, Liferay, Alfresco, GeoServer, OpenLayers, Quartz, BiMserver, Xeokit.
We are thankful for the great contributions that led to the creation of these products.

CMDBuild ® is a project of Tecnoteca Srl. Tecnoteca is responsible of software design and development, it's the official maintainer and has registered the CMDBuild logo.

The official website is http://www.cmdbuild.org

Contents

1. Introduction

1.1. The application

CMDBuild is an open source web environment for the configuration of custom applications for the Asset Management.

On the one hand, it provides native mechanisms for the administrator, implemented in a "core" code which has been kept separated from the business logic, so that the system can be configured with all its features.

On the other hand, it generates dynamically a web interface for the operators, so that they can keep the asset situation under control and always know their composition, detachment, functional relations and how they update, in order to manage their life-cycle in a comprehensive way.

The system administrator can build and extend his/her own CMDB (hence the name of the project), modeling the CMDB according to the company needs; a proper interface allows you to progressively add new classes of items, new attributes and new relations. You can also define filters, "views" and access permissions limited to rows and columns of every class.

Using external visual editors, the administrator can design workflows, import them into CMDBuild and put them at operators' disposal, so that they can execute them according to the configured automatisms.

In a similar way, using external visual editors, the administrator can design various reports on CMDB data (printouts, graphs, barcode labels, etc.), import them into the system and put them at operators' disposal.

The administrator can also configure some dashboards made up of charts which immediately show the situation of some indicators in the current system (KPI).

A task manager included in the user interface of the Administration Module allows you to schedule various operations (process starts, e-mail receiving and sending, connector executions) and to control CMDB data (synchronous and asynchronous events). Based on their findings, it sends notifications, starts workflows and executes scripts.

The interoperability with other systems is managed through the CMDBuild BUS Service, called WaterWAY.

Thanks to document management systems that support the CMIS standard (Content Management Interoperability Services) - among which there is also the open source solution Alfresco - you will be able to attach documents, pictures, videos and other files.

There is also a Scheduling, which can be supplied both automatically when filling in a data card and manually. This Scheduling will manage single or recurring deadlines related, for example, to certifications, warranties, contracts with customers and suppliers, administrative procedures, etc.

Moreover, you can use GIS features to georeference and display assets on a geographical map (external map services) and / or on vector maps (local GeoServer and spatial database PostGIS) and BIM features to view 3D models (IFC format).

The system also includes a REST webservice, so that CMDBuild users can implement custom interoperability solutions with external systems.

Furthermore, CMDBuild includes two external frameworks:

- the Advanced Connector CMDBuild, which is written in Java and can be configured in

Groovy: it helps the implementation of connectors with external data sources, i.e automatic inventory systems, virtualization or monitoring ones (supplied with non-open source license to the users that subscribe the annual Subscription with Tecnoteca)

- the GUI Framework CMDBuild, which helps the implementation of additional graphical interfaces, i.e. web pages (simplified for non technicians) that have to be published on external portals and that are able to interact with the CMDB through the REST webservice

CMDBuild includes a mobile interface (for smartphone and tablet). It is implemented as multi-platform app (iOS, Android) and is able to interact with the CMDB through the REST webservice (supplied with non-open source license to the users that subscribe the annual Subscription with Tecnoteca).

CMDBuild is an enterprise system: server-side Java, web Ajax GUI, SOA architecture (Service Oriented Architecture), based on webservice and implemented by using the best open source technologies and following the sector standards.

CMDBuild is an ever-evolving system, which has been released for the first time in 2006 and updated several times a year in order to offer more features and to support new technologies.

1.2. Official website

CMDBuild has a dedicated website: http://www.cmdbuild.org

The website gathers a lot of documents on technical and functional features of the project: brochures, slides, manuals (see next paragraph), testimonials, case histories, newsletters, forums.

1.3. CMDBuild modules

The CMDBuild application includes two main modules:

- the Administration Module for the initial definition and the next changes of the data model and the base configuration (relation classes and typologies, users and authorization, dashboards, upload report and workflows, options and parameters)
- the Management Module, used to manage cards and relations, add attachments, run workflow processes, visualize dashboards and execute reports

The Administration Module is available only to the users with the "administrator" role; the Management Module is used by all the users who view and edit data.

1.4. Available manuals

This manual is for those who need certain first introductory information on CMDBuild and who are interested in knowing the general philosophy of the project.

You can find all the manuals on the official website (http://www.cmdbuild.org):

- system overview ("Overview Manual")
- system usage for operators (“User Manual”)
- system administration ("Administrator Manual")
- installation and system management ("Technical Manual")
- webservice details and configuration (“Webservice Manual”)

1.5. Applications based on CMDBuild

Tecnoteca has used the CMDBuild environment in order to implement two different pre-configured solutions:

- CMDBuild READY2USE, for the management of assets and IT services, oriented to internal IT infrastructures or services for external clients (www.cmdbuildready2use.org) according to the ITIL best practice (Information Technology Infrastructure Library)
- openMAINT, for the inventory management of assets, properties and related maintenance activities (www.openmaint.org)

Both applications are released with open source license, except for certain external components (data sync connectors, Self-Service portal, mobile APP, etc.), that are reserved to the users that subscribe the annual Subscription with Tecnoteca.

2. Description of the workflow system

2.1. General Information

One important added value of CMDBuild is the possibility of defining processes for operators to execute the management activities.

A process includes an activity sequence, carried out by operators and/or computer applications, every application represents an operation that has to be carried out within the process, related, in this case, to the IT asset management with quality criteria.

Given the amount of processes options, organizational procedures and flexibility pursued by the CMDBuild project, we chose not to implement a series of rigid and predefined processes, but a generic workflow engine to model processes case-by-case.

In the first part of this document you will find general concepts and basic mechanisms implemented in the system.

In the second part, you will find the technical tools available for the configuration of a workflow, for example the widgets definition and the description of API functions which can be used in the scripts for the definition of automatism, performed in the workflow.

2.2. Purposes

The workflow management system provides:

- a standard interface for users
- a secure update of the CMDB
- a tool to monitor provided services
- a repository for activities data, useful to check SLA

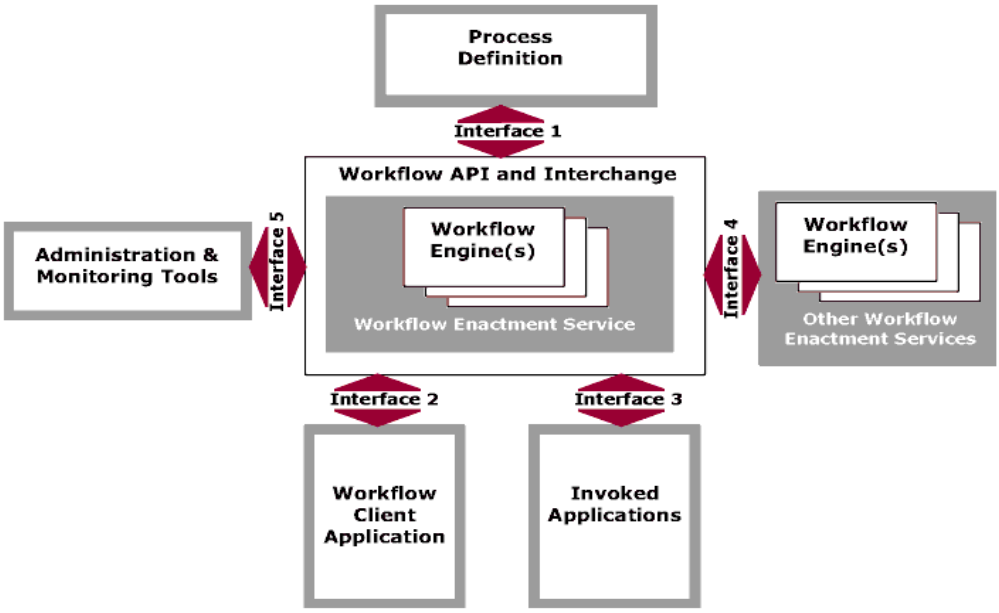
In the IT environment, these basic mechanisms allow the configuration of all processes provided by ITIL "best practices", included Incident Management, Change Management, Request Fulfillment, Service Catalog, etc. In the Facility Management environment, all maintenance processes can be configured.

2.3. Used tools

The application chosen for the workflow management uses the following tools:

- XPDL 2.0 (http://www.wfmc.org/xpdl.html) as definition language (standardized from the WfMC, WorkFlow Management Coalition)
- the Tecnoteca River engine, which provides a standard implementation of the WfMC specifications (http://www.wfmc.org/) for the part of interest of managing workflows in CMDBuild, using XPDL as a native language
- the graphical editor TWE Together Workflow Editor 5.5 or older (http://www.together.at/prod/workflow/twe) for the workflow design and for the definition of integration mechanisms with CMDBuild

The following schema shows the workflow management according to the model standardized with the WfMC.



2.4. Terminology

The workflow "vocabulary" includes the following terms:

- process: sequence of steps that realize an action
- activity: workflow step
- process instance: active process created executing the first step
- activity instance: creation of an activity, accomplished automatically or by an operator

The above terms are arranged into CMDBuild as follows:

- each process is related to a special class defined by the Administration Module under the heading "Processes"; the class includes all attributes of the scheduled activities
- each "process instance" corresponds to a card of the "process" class (at the current activity), combined with the list of its versions (ended activities)
- each activity instance corresponds to a card of the "process" class (current activity) or to a historicized version (ended activity)

Each process has a name, one or more participants, some variables and a sequence of activities and transitions.

The process status can be:

- “active”, i.e. it is still in an intermediate activity
- “complete”, i.e. it ended its activities
- “aborted”, i.e. it has been terminated before reaching its completition
- “suspeded”, i.e. it has been temporarily suspended and can be resumed

Each activity can be distinguished by:

- a name
- a performer, which necessarily corresponds to a "user group" and optionally to an operator

- a type: process start, process ending, activity performed by an operator, activity automatically carried out by the system
- a list of attributes coming from CMDBuild or from inside the workflow, which will be set during its implementation
- a list of widgets (visual controls of some predefined typologies) that will be set during its creation
- a script (in the Beanshell, Groovy or Javascript languages), provided in the automatic activities, through which the operations between an user activity and the following can be carried out

3. Implementation method

3.1. Workflows as special classes

The mechanism for workflow management is implemented in CMDBuild through concepts and procedures similar to the management of calsses and cards.

The workflow management includes:

- “special” Process classes, each corresponding to a type of workflow
- attributes, corresponding to the information presented (for read or write) in the forms which manage the advancement of each single step of the process
- relations with other process instances or standard cards involved in the process
- user groups, that will be able to perform every activity, coinciding with CMDBuild user groups
- special tools for customizing the behavior of the workflow (widgets and scripts written with proper APIs)

Within the same homogeneity criteria between "normal" and "process" classes, we adopted the following technical tactics:

- the new "confidential" superclass called "Activity" contains some attributes shared with specific workflows, whose workflows are underclasses
- the "history" mechanism was used to draw the progress reports of a process
- the "relations" mechanism has been kept to create automatic or manual links between a card and a process instance, or between two process instances

3.2. Building the workflow

The tools usable through the workflow visual editor are of utmost importance in enabling the design of complex processes, and include:

- the choice of those attributes which can be placed on each form corresponding to a user activity
- the choice of widgets (visual controls) which can be placed on each form corresponding to a user activity (viewing, creating or editing cards, viewing or creating relations, single or multiple selection of cards, upload of the attached files, implementation of reports)
- flow-control mechanisms, among them parallel activities and subprocesses
- scripting language (BeanShell, Groovy or Javascript) for the definition of those automatisms which must be carried out between a user activity and the following
- API functions which can be called in the scripts

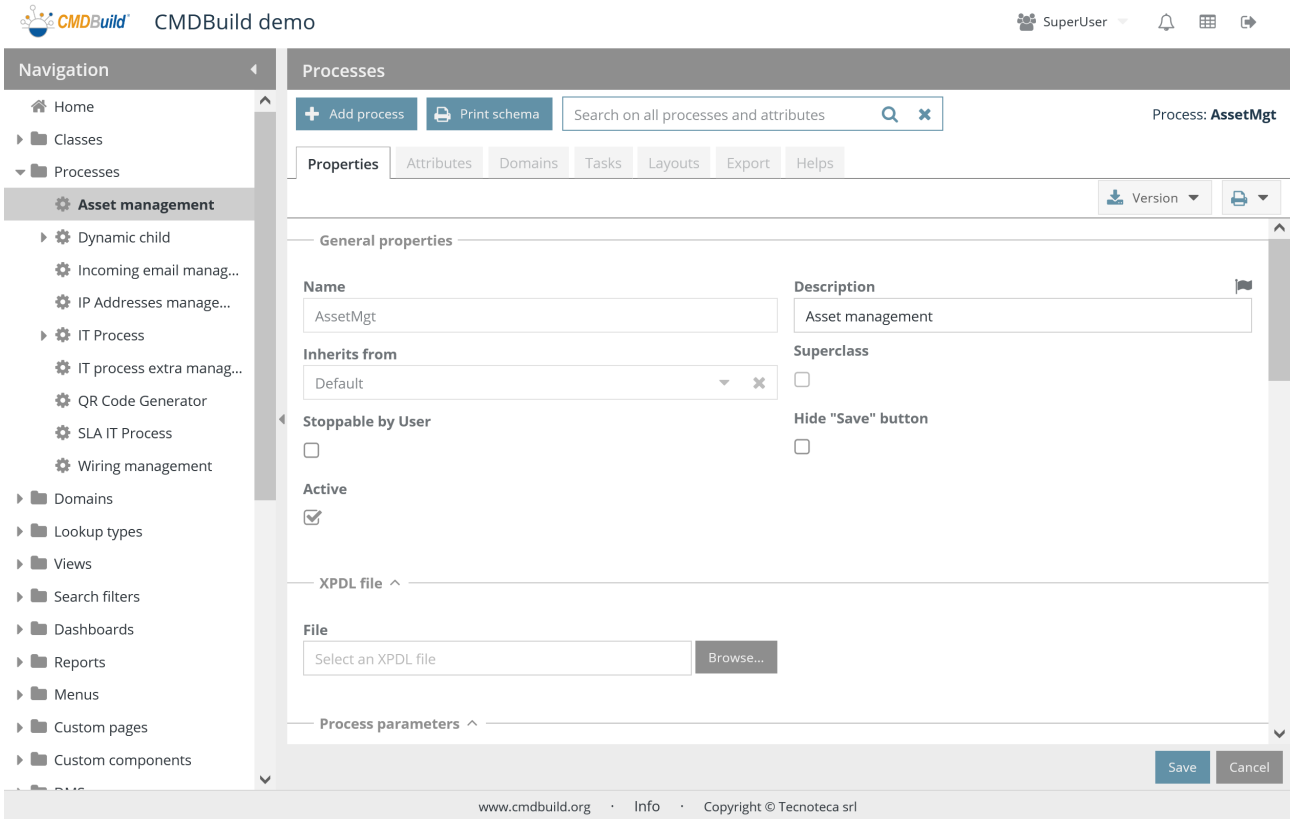
3.3. Defining a new process

To create a new "Process" class, you should follow the following logic sequence of passages:

- analysis of the new process which has to be implemented, in order to single out:
  - a list of the user groups involved in the process
  - a list of user activities, automatic activities, transition conditions, etc
  - a list of descriptive attributes of the process in its user activities, the related typologies (string, integer, etc) and their presentation mode (read-only, reading and writing, possible compulsoriness)
  - a list of values required for the creation of "Lookup" attributes
  - a list of domains required to deal correlations between the new process and other classes or other pre-existing processes (which might also be used to create the "Reference" attributes)
  - a list of widgets to be configure in every user activity
  - a list of widgets to be configure in every automatic process activity
- creation of the new process class, which will be defined in the "Processes" section of the CMDBuild Administration Module, complete of:
  - specific attributes identified in the previous step
  - domains identified in the previous step
- creation of missing user groups, that should be added through the Administration Module
- through the Administration Module (from the “XPDL” TAB available for each “Process” class) export of the new process template, which includes:
  - process name
  - list of process attributes, that will be placed in the various user activities
  - list of "actors" (users) that interact with the process (the "System" role is automatically created to position system activities)
- design of detail flow of the workflow using the TWE external editor, which will help the completion of the template exported by CMDBuild
- save, using the special functions of TWE external editor of the XML file (to be exact XPDL 2.0) corresponding to the designed process
- import of the process schema in CMDBuild, using the upload “XPDL” button, available under the corresponding process under the "Processes" tab in the Administration Module

Once the operations described above are completed, the new process can be used in the Management Module, (Menu “Processes” or headings like “process” in the Navigation Menu), thus the process can be executed using the workflow engine Tecnoteca River.

The above mentioned operations can be carried out when you need to edit an imported process, but the changes must be received only through the new process instances which will be started.



3.4. Initiation and progress of a process

In the Management Module, CMDBuild can perform, through the support of the Tecnoteca River workflow engine, the processes designed with TWE Together Workflow Editor and then imported through the Administration Module.

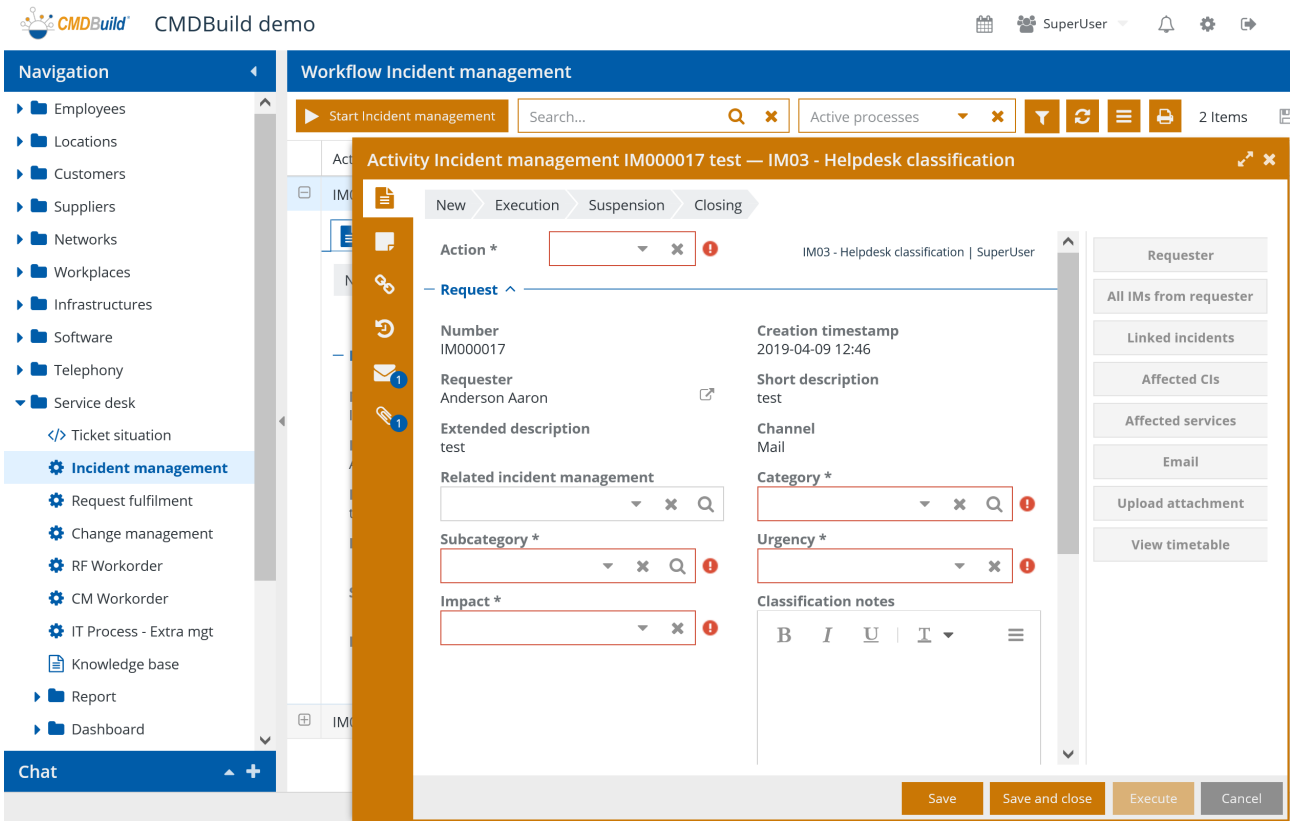
In order to keep the coherence with the CMDBuild functionalities, dedicated to the management of the item cards in the system, the user interface of the Management Module was designed so that it is consistent with the management of the normal data "classes":

- there is a special menu item, named "Processes", consistent with the "data sheets" (otherwise "process" elements can be inserted in the Navigation menu with "data sheets" elements or reports and dashboards)
- the process management draws on the standard managements which already exist for the normal cards: “List”, “Card”, "Details" “Notes”, “Relations”, “History”, "Attachments"
- in the “List” TAB of a specific process, the user can see the activities instances, in which they are involved (since they attended that activity or previous activities of that process) with:
  - filters by status (started, completed,...)
  - data area with tabular display of the information (process name, activity name, request description, process status and further attributes defined as "display base" in the Administration Module), which you can click on in order to access to the management card of that activity

- possible evidences of parallel activities for that process instance

- in the “Card” TAB you can visualize or fill in the attributes provided for that process activity instance (write or read-only access can be set up through the TWE editor) otherwise you can carry out further operations through the proper widgets (visual controls) configured with the TWE editor
- in the “Notes” TAB you can visualize or insert notes about the activity instance
- in the “Relations” TAB you can visualize or insert relations between the activity instance and the instances of other classes (“cards”)
- in the “History” TAB you can visualize the previous versions of that activity instance (instances already carried out)
- in the “Email” TAB you can visualize the details of any email related to the instance of that processes
- in the “Attachments” TAB (if the DMS Service is enabled), you can visualize the attachments related to the process instance

The list of activities is displayed high up in the following exemplifying form, while you can carry out an activity filling in the card at the bottom.



4. Widgets prompted to use in the user activities of the workflow

4.1. Widget list

CMDBuild makes some default widgets (visual controls) available, placed in the top-right part of the form, which manage the progress of the process through the provided activity.

Graphically, such controls are designed with buttons with the specified label during the definition step.

About the configuration, widgets are defined as “Extended attributes” (provided in the XPDL standard) using the TWE editor.

In this document, the data types are both standard (integer, string, date, float, boolean) and custom types added in the workflows of CMDBuild (lookup = id + type + description, lookups = lookup array, reference = id + idclass + description, references = reference array).

Visual control	Description	Parameters	Notes
Widget_linkCards	It shows the paginated list of all cards belonging to a class, with possible display on a geographical map	Input: ClassName *string* ButtonLabel *string* SingleSelect *integer* NoSelect *integer* Required *integer* Filter *string* DefaultSelection *string* AllowCardEditing *integer* DisableGridFilterToggler *boolean* Output: CheckArray *references*	The parameter SingleSelect = 1 must be set only if the selection of one single row is allowed (radio-button rather than checkbox) The parameter NoSelect = 1 disables the selection of rows The parameter Required = 1 forces the selection of at least one row The Filter parameter accepts a CQL expression (CMDBuild query language) Example: Filter = “from Person where Id = {client:Customer.Id}” The optional parameter DefaultSelection specifies the CQL query used for the automatic selection when opening the widget The optional parameter AllowCardEditing = 1 adds an icon to edit the card The optional parameter DisableGridFilterToggler = “true” hides the button “Disable filter”
Widget_createModifyCard	It shows the specified card (if ObjId is specified), otherwise it allows the creation of a new card in the specified class	Input: ClassName *string* ButtonLabel *string* ReadOnly integer or Input: Reference *reference* ButtonLabel *string*	Example: ClassName='User' ObjId=client:Requester ButtonLabel = 'Create or modify User' Requester Note: the prefix “client:” is required to access a variable before the workflow is advanced to the following step

		ReadOnly integer or Input: ClassName *string* ObjId *long* ButtonLabel *string* *ReadOnly integer* Output: Reference *reference*	ReadOnly=1 shows the card in read-only mode
Widget_createReport		Input: ReportType *string* ReportCode *string* ButtonLabel *string* ForcePDF *integer* ForceCSV *integer* Parameter-1 Parameter-2 ... Parameter-n Output: ReportURL *string*	ReportType can only take the 'custom' value ReportCode coincides with the report “Code” attribute in the schedule “Report” ForcePDF forces the output in PDF format ForceCSV forces the output in CSV format Parameter-1 ... Parameter-n represent the parameters provided by the report
Widget_manageEmail	It allows to produce and email through template or write new e-mails which will be sent during the advancement of the process.	Input: ButtonLabel *string* Template*1 string* Condition*1 string* NotifyWith*1 string*	Visualizing e-mails, the electronic mailbox will be checked for possible new e-mails Template*1* … Template*n* is the “Name” of the Email template to use to generate the email. Condition*1* … Condition*n* is the cql condition used to enable the email creation with Template1. NotifyWith*1* … NotifyWith*n* is the name of the template to use to send a notification email.
Widget_openNote	It visualizes the page which includes the HTML editor to insert notes	Input: ButtonLabel *string*	It can't be used in the first process activity
Widget_openAttachment	It visualizes the page provided for the uploading of an attachment which has to be enclosed to the current process	Input: ButtonLabel *string*	It can't be used in the first process activity
Widget_calendar	It displays the calendar with the selected dates	Input: ButtonLabel *string* ClassName *string* Filter *string* EventStartDate *date*	From the class ClassName you can collect the dates you want to display in the calendar, with a possible filter. EventStartDate is the attribute to use as event start date.

		EventEndDate *date* EventTitle *string* EventType *string* EventTypeLookup *string* *DefaultDate string*	EventEndDate is the attribute to use as event end date. It is optional. EventTitle indicates the attribute that draws the text and writes it on the calendar for every event. EventType is the attribute to use as event type. EventTypeLookup is the Lookup to use event types. Usually is the LookupType of the attribute EventType. DefaultDate is the attribute to use as opening date in calendar.
Widget_presetFromCard	It populates the current activity with the data recovered from a selected card.	Input: ButtonLabel *string* ClassName *string* Filter *string* AttributeMapping *string*	ClassName, the name of the class, as an alternative to a Filter AttributeMapping is a string with the structure 'a1=c1,a2=c2' that shows how to chart activity attributes with the card ones. The comma separates the assignments.
Widget_startWorkflow (or just Widget_workflow)	It allows to start a new workflow	1) Input: ButtonLabel *string* WorkflowCode *string* or 2) Input: ButtonLabel *string* FilterType *string* Filter *string* Output: processRef *ReferenceType*	WorkflowCode name of the starting process Filter: the cql filter to select a series of card from a CMDBuild table. The result of the filter should be the same as the name list of the processes that should be started from the widget itself.
Widget_customForm	Allows to manage a form or a row grid (by adding, removing and/or modifying the rows)	Input: ButtonLabel *string* ModelType *"[form|class|function]"* Layout *"[grid|form]"* DataType *[raw_json|raw_text|function]* ReadOnly *"[true|false]"* Required *"[true|false]"* AddDisabled *"[true|false]"* DeleteDisabled *"[true|false]"* ImportDisabled *"[true|false]"* ModifyDisabled *"[true|false]"* SerializationType *"[json|text]"* KeyValueSeparator	The structure of the custom form can be defined starting from: form - JSON item array class - attributes of a class function - function input parameters The layout can be a form (as if it is a CMDBuild card) or a row series. The data of the widget can be initialized starting form: raw_json - JSON item array raw_text – well-structured strings of text function – output values of a function Data can be serialized as type of text (see the widget grid) or as type of json.

		string AttributesSeparator *string* RowsSeparator *string* Output: Output *string* variable	

4.1.1. Further information for the use of “string template” in the tool manageEmail

The tool *manageEmail* allows to write e-mails which will be sent during the development of the process. Visualizing e-mails, the electronic mailbox will be checked for possible new e-mails to visualize the grid.

Input parameters	• *string* ButtonLabel • one or more blocks for the e-mails definition ◦ *string template* ToAddresses: recipient’s addresses ◦ *string template* CcAddresses: carbon copy addresses ◦ *string template* Subject: e-mail subject ◦ *string template* Content: e-mail body (HTML) ◦ *string template* Condition: javascript expression whose evaluation defines if the e-mail is generated or not • other optional parameters which include queries or javascript expressions • *flag* ReadOnly: read-only email
output parameters	none

The only-read *flag* is seen as a boolean value; a boolean value (of the process), a positive integer value or a non empty string are considered *true*

In the *template strings* the variables, written in the form {namespace:localname}, are interpreted in a different way depending on the namespace (if omitted, it defaults to "server").

client:name **client**:name.Id **client**:name.Description	Form's *name* variable; for attributes such as LookUp or Reference you have to specify, with the bullet list, whether you want the *Id* or the *Description*
server:name	Process *name* variable in the previous step
xa:name	Variable *name* of the extended attribute definition, extended as template excluding the variables with namespace *js* and *cql*
user:id **user**:name	ID and name of the connected user
group:id **group**:name	ID and name of the connected group
js:name	Variable *name* of the extended attribute definition interpreted as a template and evalued as a javascript code
cql:name.field	Variable *name* of the extended attribute definition interpreted as a template and evalued carrying out a CQL query, whose field is identified by *field*

The definition blocks of the e-mails can be written in two ways:

ToAddresses="**…**"
CcAddresses="..."
Subject="..."
Content="..."

or (if you want to specify more than one e-mail):

ToAddresses1="**…**"
CcAddresses1="..."
Subject1="..."
Content1="..."
ToAddresses2="..."
CcAddresses2="..."
Subject2="..."
Content2="..."
...

4.1.2. Example 1

ToAddresses="foo@example.com"
Subject="**{cql:QueryRequester.Description}** - **{client:Request}**"
QueryRequester="select Description,Email,Office from Employee where Id = **{cql:SillyQuery.Id}**"
SillyQuery="select Id from Employee where Id=**{client:Requester}**"

Address: The recipient's address is statically completed with the string foo@example.com

Body: Message Body Empty

Subject:

- The variable QueryRequester selects an Employee card which includes the fields Description, Email and Office; the extracted values are available using for example the syntax {cql:QueryRequester.Description}, which will be replaced with the field Description extracted from the variable QueryRequester
- Inside QueryRequester, {cql:SillyQuery.Id} will be replaced with the Id field of the card returned from the SillyQuery (indeed nested queries are supported), replaced before with {client:Requester} with the value taken in the form
- {client:Request} of will be completed with the form value

4.1.3. Example 2

...
Content="The requester, **{js:JoinJS}**, belonging to the office **{cql:QueryRequester.Office_value}** requests:
/>
{server:Request}"
JoinJS="**{js:FirstJS}**+"#"+**{js:SecondJS}**"
FirstJS="**{cql:QueryRequester.Description}**.slice(0,**{xa:SplitLength}**)"
SecondJS="**{cql:QueryRequester.Description}**.slice(**{xa:SplitLength}**)"
SplitLength=2
QueryRequester="select Description,Email,Office from Employee where Id = **{Requester}**"

This is an example of higher complexity.

In the body there are three variables which must by replaced:

- {js:JoinJS} values the extended attribute variable like a javascript expression, splitting with # the variables FirstJS and SecondJS, always valued through javascript
- {js:FirstJS} and {js:SecondJS} include both a variable taken from a field of CQL query QueryRequester and a static variable taken from the ones of the extended attribute

- {cql:QueryRequester...} includes a reference to a server side variable called Requester
- {cql:QueryRequester.Office_value} uses the Office reference description instead of its ID (that would be just Office)
- {server:Request} takes a server side variable (as Requester), but it also states the namespace

5. API prompted to use in the automatic activities of the workflow

CMDBuild offers various APIs (Application Programming Interface) which can be used in the automatic activities of the workflow so that it is possible to implement custom behaviors (manipulation of process variables, card creation and relations in CMDB, e-mail sending, report creation, etc)

5.1. Key words

Process
ProcessId: Long Id of the current process
ProcessClass: String Class name of the current process
ProcessCode: String univocal ProcessInstanceId of the current process

Performer
_CurrentUser: ReferenceType reference to the User that performed the last activity of the current process
_CurrentGroup: ReferenceType reference to the Role that performed the last activity of the current process

API
cmdb identifies the native functions in CMDBuild

5.2. Management of CMDBuild items

They concern the CMDBuild specific data; for other data (integer, string, date, float) you can use the manipulation methods offered by the Java language.

ReferenceType

Methods
getId(): Long returns the Reference id
getDescription(): String returns the Reference description

LookupType

Methods
getId(): Long returns theLookup id
getType(): String returns the type of Lookup
getDescription(): String returns the Lookup description
getCode(): String returns the Lookup code

CardDescriptor

Methods
getClassName(): String returns the Class name for a CardDescriptor variable
getId(): Long returns the Id name for a CardDescriptor variable
equals(CardDescriptor cardDescriptor): boolean compares the CardDescriptor variable with the specified one

Card

Methods
getCode(): String returns the Code for a Card variable
getDescription(): String returns the Description for a Card variable
has(String name): boolean controls the presence of the specified attribute in the Card variable
hasAttribute(String name): boolean controls the presence of the specified attribute in the Card variable
get(String name): Object returns the specified attribute value of the Card variable
getAttributeNames(): Set<String> returnsthe attributes list of the Card variable
getAttributes(): Map<String, Object> returns the attributes list and their values of the Card variable. The returned values respect the CMDBuild types (ReferenceType, LookupType, Date, Integer, …)

Attachments

Methods
fetch(): Iterable<AttachmentDescriptor> returns the attachments list of the Card or of the instantiated process
upload(Attachment... attachments):void attaches the documents to the card or to the instantiated process
upload(String name, String description, String category, String url):void creates an attachment with name, description and category specified starting from the file with the specified URL and attaches it to the card or to the instantiated process
selectByName(String... names): SelectedAttachments returns the attachments of the card or of the instantiated process with the specified name
selectAll(): SelectedAttachments returns all attachments of the card or of the instantiated process

AttachmentDescriptor

Methods
getName(): String returns the name of the attachment
getDescription(): String returns the attachment description
getCategory(): String returns the attachment category

Attachment

Methods
getUrl(): String returns the URL of the file

DownloadedReport

Methods
getUrl(): String returns the local URL where the report has been saved
equals(DownloadedReport downloadedReport): boolean compares the DownloadedReport variable with the specified one

5.3. Access methods to CMDBuild

5.3.1. NewCard

Builders
newCard(String className): NewCard creates a new Card created in the specified Class of CMDBuild

Modifiers
withCode(String value): NewCard adds the Code to the new card created in CMDBuild
withDescription(String value): NewCard adds the Description to the new card created in CMDBuild
with(String name, Object value): NewCard adds the value specified for the specified attribute to the new card created in CMDBuild
withAttribute(String name, Object value): NewCard adds the value specified for the specified attribute to the new card created in CMDBuild

Actions
create(): CardDescriptor creates the new card in CMDBuild setting the attributes previously defined

Example:

```
/*
 * Creation of a new card in the "Employee" class having
 * the following attributes:
 *  "Code"       =     "T1000"
 *  "Name"       =     "James"
 *  "Surname"    =     "Hetfield"
 */
cdNewEmployee = cmdb.newCard("Employee")
.withCode("T1000")
.with("Name", "James")
.withAttribute("Surname", "Hetfield")
.create();
```

5.3.2. ExistingCard

Builders
existingCard(String className, Long id): ExistingCard

creates a Card existing in the specified Class having the specified Id to query CMDBuild
existingCard(CardDescriptor cardDescriptor): ExistingCard creates an existing Card indicated by the specified CardDescriptor to query CMDBuild

Modifiers
withCode(String value): ExistingCard sets the Code for the Card requested to CMDBuild
withDescription(String value): ExistingCard sets the Description for the Card requested to CMDBuild
with(String name, Object value): ExistingCard it sets the specified attribute with the specified value for the Card requested to CMDBuild
withAttribute(String name, Object value): ExistingCard it sets the specified attribute with the specified value for the Card requested to CMDBuild
withAttachment(String url, String name, String category, String description): ExistingCard it attaches a file (pointed out through a server local url) to the selected card by setting the file name, its category and its description
attachments(): ExistingCard it allows you to access the attachments of the selected card
selectAll(): ExistingCard it allows you to select all documents of the selected card
selectByName(String name1, String name2, ...):ExistingCard it allows you to select all documents of the selected card

Actions
update() updates the Card in CMDBuild by setting the attributes previously indicated with the specified values
delete() deletes the Card from CMDBuild If the "attachments" modifier has been used, it will delete only the selected files
fetch(): Card requests the Card to CMDBuild with the attributes previously indicated. If no modifier has been used, it requests the whole Card (with all attributes)
fetch(): Iterable<AttachmentDescriptor> If the "attachments" modifier has been used, the method returns the list of the card attachments
upload(Attachment attachment, Attachment attachment2,..) to be used in the presence of the "attachments" modifier: it attaches one or more files to the card

upload(Attachment attachment, String description, String category, String url) to be used in the presence of the "attachments" modifier: it attaches to the card a single file with specified description and category
download(): Iterable<Attachment> If the "attachments" modifier has been used, the method returns the selected attachments of the card
copyTo() If the "attachments" modifier has been used, the method copies a selected attachment of the card into a specified destination
copyToAndMerge() If the "attachments" modifier has been used, the method copies a selected attachment of the card into a specified destination or skip this action if the attachment already exists
moveTo() If the "attachments" modifier has been used, the method moves a selected card attachment into a specified destination

Examples:

```
/*
 * It modifies the card previously created in the class “Employee”
 * by setting the following attributes:
 *  “Phone”      =     “754-3010”
 *  “Email”      =     “j.hetfield@somemail.com”
 */
cmdb.existingCard(cdNewEmplyee)
.with(“Phone”, “754-3010”)
.withAttribute(“Email”, “j.hetfield@somemail.com”)
.update();

/*
 * (Logic) delete of the card previously created in the class
 * “Emplyoee”
 */
cmdb.existingCard(cdNewEmplyee)
.delete();
/*
 * Delete of the card attachment that was previuosly
 * created in the “Employee” class
 */

Iterable <AttachmentDescriptor> attachments =
cmdb.existingCard(cdNewEmplyee)
```

```
    .attachments()
    .fetch();

    /*
     * Delete of the card attachment that was previuosly
     * created in the “Employee” class
     */
    cmdb.existingCard(cdNewEmplyee)
    .attachments()
    .selectByName(String[]{"attachment-name"})
    .delete();
```

5.3.3. NewProcessInstance

Builders
newProcessInstance(String processClassName): NewProcessInstance creates a new process instance created in CMDBuild for the specified process

Modifiers
withDescription(String value): NewProcessInstance adds the Description to the new card created in CMDBuild
with(String name, Object value): NewProcessInstance adds the value specified for the specified attribute to the new process created in CMDBuild
withAttribute(String name, Object value): NewProcessInstance adds the value specified for the specified attribute to the new process created

Actions
start(): ProcessInstanceDescriptor creates the new process in CMDBuild setting the attributes previously defined, and does not advance
startAndAdvance(): ProcessInstanceDescriptor creates the new process in CMDBuid setting the attributes previously defined, and advances at the following step

Example:

```
    /*
     * Creation of a new card in the “RequestForChange” class
     * having the following attributes
     *  “Requester”  = “James Hetfield”
     *  “RFCExtendedDescription” = “My printer is broken”
     */
```

```
pidNewRequestForChange =
cmdb.newProcessInstance("RequestForChange")
.with("Requester", "James Hetfield")
.withAttribute("RFCExtendedDescription", "My printer is broken")
.startAndAdvance();
```

5.3.4. ExistingProcessInstance

Builders
existingProcessInstance(String processClassName, int processId): ExistingProcessInstance creates a process instance existing in the specified process class with the specified Id

Modifiers
withProcessInstanceId(String value): ExistingProcessInstance sets the process instance Id
with(String name, Object value): ExistingProcessInstance sets the specified attribute with the specified value for the process instance
withAttribute (String name, Object value): ExistingProcessInstance sets the specified attribute with the specified value for the process instance
withDescription(String value): ExistingProcessInstance sets the specified attribute with the specified value for the process instance
attachments(): Attachments allows you to access the attachments of the process instance

Actions
abort(): void aborts the process instance
advance(): void advances a process instance
resume(): void resumes the hanging process instance
suspend(): void suspends the open process instance
update(): void updates the process instance

Example:

```
/*
* Update of the process instance in the class "Request
* for change" with Id = pid by editing the requester and
* advancing the process at the following step
*/

cmdb.existingProcessInstance("RequestForChange",  pid)
.with("Requester", cdNewEmployee.getId())
.advance();
```

5.3.5. NewRelation

Builders
newRelation(String domainName): ExistingProcessInstance creates a new relation added in the specified Domain of CMDBuild

Modifiers
withCard1(String className, int cardId): NewRelation sets the card in the source side of the relation
withCard2(String className, int cardId): NewRelation sets the card in the target side of the relation
withAttribute(String attributeName, Object attributeValue): NewRelation sets the value of a relation attribute

Actions
create() creates the new relation in CMDBuild among the Cards indicated in the specified Domain

Example:

```
/*
 * Creation of a new relation in the “AssetAssignee” domain
 * between a card of the selected “Asset” class,
 * through the "Item" Reference attribute, and
 * the card previously created in the “Employee” class
 */
cmdb.newRelation(“AssetAssignee”)
.withCard1(“Employee”, cdNewEmployee.getId())
.withCard2(“Asset”, Item.getId())
.create();
```

5.3.6. ExistingRelation

Builders
existingRelation(String domainName): ExistingRelation creates an existing relation in the specified Domain of CMDBuild

Modifiers
withCard1(String className, int cardId): ExistingRelation sets IdClass and l'ObjId of the Card from the source side of the relation
withCard2(String className, int cardId): ExistingRelation

sets IdClass and l'ObjId of the Card from the target side of the relation

Actions
delete() deletes the relation existing among the Cards indicated in the specified Domain

Example:

```
/*
 * Delete the relation on the "AssetAssignee" domain
 * among the cards previously indicated
 */
cmdb.existingRelation("AssetAssignee")
.withCard1("Employee", cdNewEmployee.getId())
.withCard2("Asset", Item.getId())
.delete();
```

5.3.7. QueryClass

Builders
queryClass(String className): QueryClass creates a query that queries the class specified in CMDBuild

Modifiers
withCode(String value): QueryClass sets the Card Code for the filter used to query CMDBuild
withDescription(String value): QueryClass sets the Card Description for the filter used to query CMDBuild
with(String name, Object value): QueryClass sets the value for the specified attribute of the Card for the filter used to query CMDBuild
withAttribute(String name, Object value): QueryClass sets the value for the specified attribute of the Card for the filter used to query CMDBuild

Actions
fetch(): List<Card> performs the search query on the specified Class of CMDBuild and returns the list of those Cards that respect the filter previously set

Example:

```
/*
 * List of the cards of the “Employee” class having
 * the “State” attribute set to 'Active'
 */
Employees = cmdb.queryClass(“Employee”)
.with(“State”, “Active”)
.fetch();
```

5.3.8. QueryLookup

Builders
queryLookup(String type): QueryAllLookup creates a query that queries the lookup type specified in CMDBuild

Actions
fetch(): Iterable<Lookup> performs the search query on the specified lookup type of CMDBuild and returns the list of those Lookups

5.3.9. CallFunction

Builders
callFunction(String functionName): CallFunction creates a call to a stored procedure previously defined in PostgreSQL

Modifiers
with(String name, Object value): CallFunction sets the value of the input parameter specified for the stored procedure

Actions
execute(): Map<String, Object> performs the stored procedure and returns the list of the output parameters with the related values

Example:

```
/*
 * Call of the stored PostgreSQL procedure
 * “cmwf_getImpact”(IN “DeliveryDate” date, IN “Cost” integer,
 * OUT “Impact” character varying)
 * that computes the impact level (attribute of
 * “Impact” process) of an activity on a scale of "High",
```

```
 * "Medium" and "Low", given in input the expected delivery
 * date (process attribute "ExpectedDeliveryDate") and
 * the price (attribute "ManHoursCost") expressed in hour/employee
 */
spResultSet = cmdb.callFunction("cmwf_getImpact")
.with("DeliveryDate", ExpectedDeliveryDate.getTime())
.with("Cost", ManHoursCost)
.execute();
Impact = spResultSet.get("Impact")
```

Note: SQL functions - which should be called - must be defined according to CMDBuild standards. For their definitio,n see the Administrator Manual, section Cart TAB, paragraph "Definition of the data source (PostgreSQL function)".

5.3.10. QueryRelations

Builders
queryRelations(CardDescriptor cardDescriptor): ActiveQueryRelations creates a query to ask CMDBuild the Cards related to the specified one
queryRelations(String className, long id): ActiveQueryRelations creates a query to ask CMDBuild the Cards related to that specified by className and id

Modifiers
withDomain(String domainName): ActiveQueryRelations sets the Domain to perform the query

Actions
fetch(): List<CardDescriptor> performs the query on CMDBuild using the parameters previously defined, it returns the list of the linked Cards

Example:

```
/*
 * List of "Assets" linked to the "Employee" card indicated
 * by the CardDescriptor cdNewEmployee previously created,
 * through the relation on the domain "AssetAssignee"
 */
assets = cmdb.queryRelation(cdNewEmployee)
.withDomain("AssetAssignee")
.fetch();
```

5.3.11. CreateReport

Builders
createReport(String title, String format): CreateReport creates the Report in the specified format (pdf, csv) with the specified Title

Modifiers
with(String name, Object value): CreateReport sets the input parameter value specified for the Report

Actions
download(): DownloadedReport generates the indicated Report using the parameters previously defined

Example:

```
/*
 * It generated the Report "DismissedAssets" which shows the list
 * of the abandoned Assets
 */
newReport = cmdb.createReport("Assigned assets to")
.download();
```

5.3.12. NewMail

Builders
newMail(): NewMail creates a new e-mail to send

Modifiers
withFrom(String from): NewMail sets the sender of the e-mail to send
withTo(String to): NewMail sets the recipient of the e-mail to send
withTo(String... tos): NewMail sets the recipients of the e-mail to send
withTo(Iterable<String> tos): NewMail sets the recipients of the e-mail to send
withCc(String cc): NewMail sets the carbon copy recipient of the e-mail to send

withCc(String... ccs): NewMail sets the carbon copy recipients of the e-mail to send
withCc(Iterable<String> ccs): NewMail sets the carbon copy recipients of the e-mail to send
withBcc(String bcc): NewMail sets the blind carbon copy recipient of the e-mail to send
withBcc(String... bccs): NewMail sets the blind carbon copy recipients of the e-mail to send
withBcc(Iterable<String> bccs): NewMail sets the blind carbon copy recipients of the e-mail to send
withSubject(String subject): NewMail sets the subject of the e-mail to send
withContent(String content): NewMail sets the text of the e-mail to send
withContentType(String contentType): NewMail sets the content MimeType of the e-mail to send, the allowed values are “text/html” or “text/plain”. If not otherwise specified, the default value is “text/plain”
withAttachment(URL url): NewMail sets the url of a document to enclose to the e-mail
withAttachment(String url): NewMail sets the url (as a string) of a document to enclose to the e-mail
withAttachment(URL url, String name): NewMail sets the url of a document with the specified name to enclose to the e-mail
withAttachment(String url, String name): NewMail sets the url (as a string) of a document to enclose to the e-mail with a specified name
withAttachment(DataHandler dataHandler): NewMail sets the dataHandler as an attachment of the e-mail
withAttachment(DataHandler dataHandler, String name): NewMail sets the dataHandler as an attachment with the specified name
withCard(@Nullable String className, @Nullable Long cardId): NewMail sets the card related to the email
withCard(@Nullable CardDescriptor card): NewMail sets the card related to the email
withCard(@Nullable ReferenceType card): NewMail sets the card related to the email
withAsynchronousSend(bool boolean): NewMail sends the e-mail asynchronously in spite of the script; in this way any timeout problem will be avoided, but you will not be able to intervene in case of error by sending the e-mail

Actions
send() performs the e-mail sending using the previously defined statements

Example:

```
/*
 * Send a new email
 */
cmdb.newMail()
.withFrom("fromaddress@somemail.com")
.withTo("toaddress@somemail.com")
.withCc("ccaddress@somemail.com")
.withSubject("Mail subject")
.withContent("Mail content")
.send();
```

5.4. Methods for types conversion

5.4.1. ReferenceType

Methods
referenceTypeFrom(Card card): ReferenceType returns the ReferenceType item related to the specified Card
referenceTypeFrom(CardDescriptor cardDescriptor): ReferenceType retuns the ReferenceType item related to the specified CardDescriptor
referenceTypeFrom(long id): ReferenceType returns the ReferenceType item related to the card with the specified Id

Example:

```
/*
 * Set the "Requester" process attribute Reference
 * type, given the "cdNewEmployee" CardDescriptor
 * previously created
 */
Requester = cmdb.referenceTypeFrom(cdNewEmployee);
```

5.4.2. LookupType

Methods
selectLookupById(long id): LookupType returns the LookupType item with the specified Id
selectLookupByCode(String type, String code): LookupType returns the LookupType item with specified Type and Code

selectLookupByDescription(String type, String description): LookupType returns the LookupType item with specified Type and Description

Example:

```
/* Set the "State" process attribute Lookup type having:
 *  "Type" =     "Employee state"
 *  "Code" =     "ACTIVE"
 */
State = cmdb.selectLookupByCode("Employee state", "ACTIVE");
```

5.4.3. CardDescriptor

Methods
cardDescriptorFrom(ReferenceType reference): CardDescriptor returns the CardDescriptor of the specified card through the specified ReferenceType item

Example:

```
/*
 * Get the CardDescriptor related to the "Requester"
 * process attribute Reference type
*/
cdSelectedEmployee = cmdb.cardDescriptorFrom(Requester);
```

5.4.4. Card

Methods
cardFrom(ReferenceType reference): Card returns the Card item of the specified card through the specified ReferenceType item

Example:

```
/*
 * Get the complete Card related to the "Requester"
 * process attribute Reference type
 */
selectedEmployee = cmdb.cardFrom(Requester);
```

6. Appendix: Glossary

ATTACHMENT

An attachment is a file associated to a card.

In order to manage the attachments, CMDBuild uses in embedded mode any document system which is compatible with the standard protocol CMIS.

The management of the attachments supports the versioning of those files that have been uploaded a few times, with automatic numbering.

See also: Card

ACTIVITY

Activity: workflow step.

An activity can be an interaction with the operator (interactive) or a script that processes operations via API (automatic).

A process instance is a single process that has been activated automatically by the application or manually by an operator.

See also: Process

ATTRIBUTE

The term refers to an attribute of a CMDBuild class (for example in "supplier" class the attributes can be: name, address, phone number, etc.).

CMDBuild allows you to create new attributes (in classes and domains) or edit existing ones.

In the database, every attribute is related to a column in the table which implements the associated class and corresponds, in the Data Management Module, to a data entry field of the specific card for the class management.

See also: Class, Domain, Report, Superclass, Attribute Type

BIM

Method with the aim to support the whole life cycle of a building: from its construction, use and maintenance, to its demolition, if any.

The BIM method (Building Information Modeling) is supported by several IT programs that can interact through an open format for data exchange, called IFC (Industry Foundation Classes).

CMDBuild includes a connector to sync some CI information (technical or maintenance records) and an interactive viewer for the 3D model of the building represented by the IFC file.

See also: CI, GIS

CI

We define CI (Configuration Item) each item that provides a service to a user and has a sufficient detail level for its technical management.

In CMDBuild, the term is applied to a generic context of Asset Management extending the concept usually used in the management of IT infrastructure.

CI examples include: server, workstation, software, plant, electric panel, fire extinguisher, furniture, etc.

See also: Configuration, ITIL

CLASS

A Class is a complex data type having a set of attributes that describe that kind of data.

A Class models an object that has to be managed in the CMDB, such as a company, a building, an asset, a service, etc.

CMDBuild allows the administrator - with the Schema Module - to define new classes or delete / edit existing ones.

A class is represented in the database with a table automatically generated when defining the class and corresponds - in the Data Management Module - to a card for the consultation and update of the cards expected in the model.

See also: Card, Attribute

CMDB

ITIL best practice (Information Technology Infrastructure Library), which has become a "standard de facto" and a non-proprietary system for services management, has introduced the term CMDB referred to the Configuration Item database.

CMDBuild extends the concept of CMDB applying it to a generic Asset Management context.

See also: Database, ITIL

CONFIGURATION

The configuration management process is designed to keep updated and available to other processes the items (Configuration Item) information, their relations and their history.

Even if it known as one of the main processes within the ITIL Best Practice, the same concept is used in CMDBuild for generic contexts of Asset Management.

See also: CI, ITIL

DASHBOARD

In CMDBuild, a dashboard corresponds to an application page including one or more different graphic representations, in this way you can immediately hold in evidence some key parameters (KPI) related to management aspects of the Asset Management service.

See also: Report

DATABASE

The term refers to a structured collection of information, hosted on a server, as well as utility software that handle this information for tasks such as initialization, allocation, optimization, backup, etc..

CMDBuild relies on PostgreSQL, the most powerful, reliable, professional and Open Source database, and uses its advanced features and object-oriented structure.

The Asset Management database, implemented on the basis of the CMDBuild logics and philosophy, is also indicated as CMDB (Configuration Management Data Base).

DOMAIN

A domain is a relation between two classes.

A domain has a name, two descriptions (direct and inverse), classes codes, cardinality and attributes.

The system administrator, using the Administration Module, is able to define new domains or delete / edit existing ones.

It is possible to define custom attributes for each domain.

See also: Class, Relation

DATA FILTER

A data filter is a restriction of the list of those elements contained in a class, obtained by specifying boolean conditions (equal, not equal, contains, begins with, etc.) on those possible values that can be accepted by every class attribute.

Data filters can be defined and used exceptionally, otherwise they can be stored by the operator and then recalled, or configured by the Administrator and made available by operators.

See also: Class, View

GIS

A GIS is a system able to produce, manage and analyze spatial data by associating geographic elements to one or more alphanumeric descriptions.

GIS functionalities in CMDBuild allow you to create geometric attributes (in addition to standard attributes) that represent, on plans / maps, markers position (assets), polylines (transmission lines) and polygons (floors, rooms, etc.).

See also: BIM

GUI FRAMEWORK

It is a framework provided by CMDBuild to simplify the implementation of external custom user interfaces and to grant a simplified access to non-technical users. They can be issued onto any webportals and can be used with CMDBuild through the standard REST webservice.

The CMDBuild GUI Framework is based on javascript JQuery libraries.

See also: Mobile, Webservice

ITIL

It is a "best practices" system that established a "standard de facto"; it is a non-proprietary system for the management of IT services, following a process-oriented schema (Information Technology Infrastructure Library).

ITIL processes include: Service Support, Change Management and the Configuration Management.

For each process, ITIL handles description, basic components, criteria and tools for quality management, roles and responsibilities of the resources involved, integration points with other processes (to avoid duplications and inefficiencies).

CMDBuild uses some ITIL concepts and applies them to a generic context of Asset Management.

See also: Configuration

LOOKUP

The term "Lookup" refers to a pair of values (Code, Description) set by the administrator in the Administration Module.

These values are used to bind the user's choice (at the form filling time) to one of the preset values (also called multiple choice or picklist).

With the Administration Module it is possible to define new "LookUp" tables according to

organization needs.

See also: Attribute type

MOBILE

It is a user interface for mobile tools (smartphones and tablets).

It is implemented as multi-platform app (iOS, Android) and can be used with the CMDB through the REST webservice.

See also: Webservice

PROCESS

The term process (or workflow) refers to a sequence of steps that realize an action.

For each process (type of process) a new process instance will be started when users have to carry out an action on assets according to a procedure implemented as workflow.

A process is activated by starting a new process (filling related form) and ends when the last workflow step is executed.

The workflows managed in CMDBuild are described in the standard markup language XPDL (XML Process Definition Language), ruled by the WFMC (WorkFlow Management Coalition).

See also: Workflow step

RELATION

A relation is a link between two CMDBuild cards or, in other words, an instance of a given domain.

A relation is defined by a pair of unique card identifiers, a domain and attributes (if any).

CMDBuild allows users, through the Management Module, to define new relations among the cards stored in the CMDB.

See also: Class, Domain

REPORT

The term refers to a document (PDF or CSV) containing information extracted from one or more classes and related domains.

The reports can be configured in the Administration Module importing in XML format the description of the layout designed with the visual editor JasperReports. They can be provided to operators in the application menu.

CMDBuild users can print reports using the Management Module, which will result as printouts, charts, documents, labels, etc.

See also: Class, Domain, Database

CARDS

The term "card" refers to an element stored in a class (corresponding to the record of a table in the database).

A card is defined by a set of values, i.e. the attributes defined for its class.

CMDBuild users, through the Management Module, are able to store new cards and update / delete existing ones.

Card information is stored in the database and, more exactly, in the table/columns created for that class (Administration Module).

See also: Class, Attribute

SUPERCLASS

A superclass is an abstract class used as template to define attributes shared between subclasses. From the abstract class, or from abstract class hierarchies, you can derive real classes that contain data and include both shared attributes (specified in the superclass) and specific subclass attributes, besides the relations on the superclass domains and on specific domains.

For example, you can define the superclass "Company" with some basic attributes (VAT number, Business name, Phone number, etc.) and the derived subclasses "Customers" and "Suppliers", each one with both generic attributes of the superclass and its own attributes and relations.

See also: Class, Attribute

TENANT

A "tenant", in CMDBuild, is a part of the CMDB reserved to users belonging to a suborganization of the CMDBuild instance (a Group Society, a Seat, a Division, etc.).

Working in "multitenant" mode, every user works only on data of his/her suborganization and, in case, on common data: "tenants".

The list of usable Tenants can be defined from an applicable class of CMDBuild (seats, companies, customers, etc.) or from a database custom function, where you can implement complex visibility rules.

ATTRIBUTE TYPE

Each attribute has a data type that represents attribute information and management.

The type of attribute and its management modes are defined in the Administration Module.

CMDBuild manages the following attribute types: “Boolean”, “Date”, “Decimal”, “Double”, “Inet” (IP address), “Integer”, “LookUp” (lists set in "Settings" / "LookUp"), “Reference” (foreign key), “String”, “Text”, “TimeStamp”.

See also: Attribute

VIEW

A view includes cards defined with logic criteria of filters applied to one or more CMDB classes.

In particular, a view can be defined in CMDBuild by applying a filter to a class (so it will contain a reduced set of the same rows) or specifying an SQL function which extracts attributes from one or more related classes.

The first view type maintains all functionalities available for a class, the second one allows the sole display and search with fast filter.

See also: Class, Filter

WEBSERVICE

A webservice is an interface that describes a collection of methods, available over a network and working using XML messages.

With webservices, an application allows other information and applications to interact with its methods.

CMDBuild includes a SOAP and a REST webservice, which are provided to external applications to read or write data on CMDB or process operations.

WIDGET

A widget is a component of a GUI that improves user interaction with the application.

CMDBuild uses widgets (presented as "buttons") that can be placed on cards or processes. The buttons open popup windows that allow you to consult or insert data or process other operations.

CMDBuild includes some standards widgets to process the most common operations, but it also supplies the specifications to implement other custom widgets.