

Cyclos 4 PRO Documentation

Welcome to the Cyclos 4 PRO Documentation. First, this manual contains the [Installation and Maintenance guide](#). Second, this manual will give a detailed description and some examples of how to connect to Cyclos using the webservices. Subsequently, this manual explains the Cyclos scripts, these scripts can be executed by clicking on a menu link, by a scheduled task or by an extension point on a certain function. These scripts make it possible to add new functions to Cyclos and customize Cyclos exactly to the needs of your payment system. Finally, this manual will give an explanation of how to login to Cyclos from an external website. This can be useful if you have a large CMS as a website and you want to have an integrated login to Cyclos in this website.

There are some important documentation resources that are not part of this manual, these can be found here:

- There are two (end user) Cyclos 4 manuals (make sure you are not logged into communities.cyclos.org):
 - [Administrator manual](#)
 - [User manual](#)
- Next to the manuals some functions are described with much more technical details in our wiki:
 - [Configurations](#)
 - [Groups](#)
 - [Networks](#)
 - [Advertisements](#)
 - [Users records](#)
 - [Transfer authorization](#)
 - [SMS](#)
 - [Imports](#)
- Cyclos instruction videos:
 - [Cyclos 4 communities](#)
 - [Cyclos 4 PRO](#)

Table of Contents

1. Installation & maintenance	1
1.1. Install Cyclos using Tomcat	1
System requirements	1
Install Java	1
Install PostgreSQL (database)	2
Install Tomcat (web server)	3
Install Cyclos	4
Startup Cyclos	5
Upgrading Cyclos	5
Problem solving	6
1.2. Install Cyclos as a Docker image	7
1.3. Adjustments (optional)	7
Adjust Tomcat/Java memory	7
Enable SSL/HTTPS	7
Clustering	8
Use Apache as frontend for Tomcat	9
Enable SSL on apache	10
Configuring Cyclos to work behind a proxy / load balancer	12
Reserved names that cannot be used in proxy paths	14
Enabling Google Maps	15
External content storage	15
Storage types	15
Storage migrator utility class	17
Using Elasticsearch	17
Logging	17
1.4. Maintenance	18
Backup	18
Restore	18
Backup / restore of very large databases	19
Reset admin password directly on database	19
Sending database to third parties	19
Removing all data from a network	20
2. Full text searches	21
2.1. Using Elasticsearch	22
Language processing	23
Reindexing	23
3. Web services	25
3.1. Introduction	25
REST API	25
WEB-RPC	25

Authentication in web services	25
User and password	26
Login with a session	26
Access clients	26
Channels	27
3.2. Java clients	28
Dependencies	28
Using services from a 3rd party Java application	28
Examples	29
Configure Cyclos	29
Search users	30
Search advertisements	30
Register user	31
Edit user profile	33
Login user	34
Get account information	36
Perform payment	37
3.3. PHP clients	39
Dependencies	39
Using services from a 3rd party PHP application	39
Examples	40
Configuration	40
Search users	40
Search advertisements	41
Login user	41
Perform payment from system to user	42
Perform payment from user to user	43
Error handling	43
3.4. Other clients	44
Examples	47
3.5. Server side configuration to enable web services	48
3.6. Available services and API Changes	48
4. Scripting	49
4.1. Scripting engine	49
Variables bound to all scripts	50
Script storage	52
4.2. Debugging scripts	53
4.3. Script types	53
Library	53
Custom field validation	54
Examples	55
Load custom field values	56

Examples	60
Account number generation	62
Examples	62
Account fee calculation	63
Examples	63
Transfer fee calculation	63
Examples	64
Transfer status handling	65
Examples	66
Session handling	66
Examples	68
Password handling	70
Examples	70
Extension points	70
User extension point	72
Operator extension point	72
Address extension point	73
Phone extension point	73
Record extension point	74
Advertisement extension point	74
Transaction extension point	74
Transaction authorization extension point	77
Transfer extension point	77
Voucher extension point	78
Agreement extension point	78
Import extension points	78
Examples	79
Custom operations	82
Actions	89
Examples	90
Possibilities for custom operations that return a result page	100
Running custom operations on bulk actions	101
Custom wizards	103
Examples	107
Custom web services	109
Examples	111
Service interceptors	112
Recovering from errors in crucial services	113
Examples	113
Custom scheduled tasks	115
Examples	115
Custom SMS operations	117

Examples	117
Inbound SMS handling	118
Examples	120
Outbound SMS handling	120
Examples	121
Link generation	123
Examples	125
Export formats	125
Examples	127
Notifications	129
Examples	130
4.4. Solutions using scripts	131
PayPal Integration	131
Check the root URL	131
Enable transaction number in currency	131
Create a system record type to store the client id and secret	132
Create a user record type to store each payment information	132
Create the library script	133
Create the custom operation script	143
Create the custom operation	145
Configure the system account from which payments will be performed to users	145
Configure the payment type which will be used on payments	145
Grant the administrator permissions	146
Setup the PayPal credentials	146
Grant the user permissions / enable the operation	146
Configuring the script parameters	146
Other considerations	147
Loan module	148
Enable transaction number in currency	148
Create the transfer status flow	148
Create the payment custom fields	149
Configure the system account from which payments will be performed to users	150
Create the payment type which will be used to grant the loan	150
Configure the user account which will receive loans	151
Create the payment type which will be used to repay the loan	151
Create the library script	151
Create the custom operation script	157
Create two extension point scripts	157
Create the custom operation	158
Create the extension points	159

Grant the administrator permissions	160
Enable the custom operation for users which will be able to receive loans	160
Integrating with Global USSD	161
Create the USSD channel	161
Enable the USSD channel for users	161
Create a payment type for USSD	161
Grant permissions for users to perform this payment type	162
Create the library script	162
Create the custom web service script	172
Create the custom web service	173
Enable a Global USSD account	174
Start an USSD session	174
Record edition	174
Create the user record type to work with and give permissions	175
Create the library script	175
Create the custom operation script to send the record email	177
Create the custom operation script to search records	177
Create the custom operation script to create records	178
Create the custom operation script to view records	180
Create the custom operation script to update records	183
Create the custom operation script to remove records	185
Create the custom operation to remove records	186
Create the custom operation to update records	186
Create the custom operation to send the record email	187
Create the custom operation to view records details	188
Create the custom operation to create records	188
Create the custom operation to search records	189
Enable the custom operation for users	190
User balances	190
Create the script to load user account types	191
Create the script to load user groups	191
Create the custom operation script to search users balances	192
Create the custom operation to search users balances	195
Enable the custom operation	196
4.5. Running scripts directly	196
Examples	196
Remove all users, transactions and related data	196
5. Custom alerts	200
6. External login	201

1. Installation & maintenance

This is the installation manual for Cyclos 4 PRO. Be aware that Cyclos is server side software. End users (customers) will be able to access Cyclos directly with a webbrowser or mobile phone. If you have any problems when installing Cyclos using this manual, you can ask for help at our [forum](#).

Cyclos can be installed on a tomcat server or inside a docker container. If you want to have a quick preview of Cyclos it is easier to use the docker container (especially on Linux). Chapter ["Install Cyclos using Tomcat"](#) explains how to install Cyclos using a normal tomcat server and chapter ["Install Cyclos as a Docker image"](#) explains how to install Cyclos using docker.

1.1. Install Cyclos using Tomcat

System requirements

- Operation system: Any OS that can run the Java VM like Windows, Linux, FreeBSD or Mac;
- Make sure you have at least 1GB memory available for Cyclos;
- Java Runtime Environment (JRE), Java 11 is required;
- Web server: Apache Tomcat 8.5 or 9.0. Tomcat 10+ implements Jakarta EE 10, which is not compatible;
- Database server: PostgreSQL 9.6 or higher;
- Cyclos installation package cyclos-version.number.zip;

Install Java

You can check if you have Java installed at this site: <http://java.com/en/download/installed.jsp>
If you don't have Java 11 installed proceed with the steps below:

Linux (Ubuntu)

- Install the openjdk-11-jre package.

Windows

- Download and install the last [Java SE 11](#) or greater
- Install the program to <install_dir> (for windows users e.g. C:\Program Files\Java\jre11).
- Make sure your system knows where to find JAVA, in Windows you should make an environmental variable called "JAVA_HOME" which points to the <install_dir>:
 - In Windows: Control Panel > System and Security > System > Advanced system settings > Environmental Variable

- In case you have different java versions installed make sure the PATH, CLASSPATH and JAVA_HOME point to the right directory.
- You can easily test if everything is set right by executing the following commands in command prompt:

```
echo %CLASSPATH%
```

```
echo %PATH%
```

```
echo %JAVA_HOME%
```

Install PostgreSQL (database)

Linux

- If using Ubuntu Linux, [these](#) instructions are followed, type the following commands in a terminal:
- Install PostgreSQL and PostGIS (Ubuntu 18.04 or greater)

```
sudo apt install postgresql postgis
```

- Access the postgresql command line:

```
sudo -u postgres psql
```

- If you see "postgres=#" you are in the PostgreSQL command line and you can follow the instructions below.

Windows

- If using Windows, download the latest version of PostgreSQL and PostGIS:
 - PostgreSQL: <http://www.postgresql.org/download/windows> (for example the graphical installer)
 - PostGIS: http://postgis.net/windows_downloads (PostGIS can also be installed using the Stack Builder, that starts after PostgreSQL is installed. Also in this case use the default options.)
- Install both PostgreSQL and PostGIS by following the installer steps (use the default options).
- Make sure the bin directory is included in the system variables so that you can run psql directly from the command line:
 - Go to: "Start > Control Panel > System and Security > System > Advanced system settings > Environment Variables...".

- Then go to the system variable with the name "Path" add the bin directory of PostgreSQL as a value, don't forget to separate the values with a semicolon, e.g.:
 - Variable name: Path
 - Variable value: Enter here the bin folder in Postgres installation folder, e.g.: C:\Program Files\PostgreSQL\9.4\bin;
- Go to the windows command line and type the command (you will be asked for the password you specified when installing PostgreSQL):

```
psql -U postgres
```

- If you see "postgres=#" you are in the PostgreSQL command line and you can follow the instructions: [Setup cyclos4 database \(common steps for windows and Linux\)](#).

Setup cyclos4 database (common steps for Windows and Linux)

- Create a database user and password. This password and username you will have to enter in the cyclos.properties file in step 5. Type in the PostgreSQL command line:

```
CREATE USER cyclos WITH ENCRYPTED PASSWORD 'cyclos-password';
```

- Create the database cyclos4, type in the PostgreSQL command line:

```
CREATE DATABASE cyclos4 ENCODING 'UTF-8' TEMPLATE template0 OWNER cyclos;
```

- Create the PostGIS and unaccent extensions on the database, type in the PostgreSQL command line:

```
\c cyclos4
create extension cube;
create extension earthdistance;
create extension postgis;
create extension unaccent;
```

- Exit the PostgreSQL command line by entering "\q" (and pressing enter).

Install Tomcat (web server)

- Download Tomcat (8.5 or greater) at <http://tomcat.apache.org/>
- Extract the zipped tomcat file into a folder <tomcat home>.
- Start tomcat: <tomcat home>/bin/startup.bat (Windows) or <tomcat home>/bin/startup.sh (Linux). You might have to give the startup script file execute permissions.
- Open a browser and go to <http://localhost:8080/> and check if tomcat is working.
- The default memory heap size of Tomcat is very low, we recommend increasing it (see [adjustments](#)).

Install Cyclos

Make sure tomcat is working on port 8080 of the local machine (if you don't run Tomcat as root/admin make sure that the user has write access to the webapps directory)

- Download the latest version of Cyclos from the [license server](#). To download Cyclos from the license server you first have to register on the license server. Registering at the license server allows you to use the free version of Cyclos. Please write down the loginname and password you chose when registering for the license server (it will be needed later on).
- Unzip the cyclos-version.zip into a temporary directory.
- Browse to the temporary directory and copy the directory web (including its contents) into the webapps directory (<tomcat_home>/webapps) of the tomcat installation.
- Rename this web directory to cyclos. This name will define how users access Cyclos. For example, if you run the tomcat server on www.domain.com the URL would be http://www.domain.com/cyclos. Of course it is also possible to run Cyclos directly under the domain name. This can be done by extracting Cyclos in the (<tomcat_home>/ROOT) directory (remove all files on it first), or putting an Apache web server in front of the Tomcat server.
- In the folder <tomcat_home>/webapps/cyclos/WEB-INF/classes you'll find the file cyclos-release.properties. The first thing to do is to copy this file and give it the name cyclos.properties. The original name is not shipped, so in future installations you can just override the entire folder, and your customizations won't be overwritten.
- In the cyclos.properties file you can set the database configuration, here you have to specify the username and password, by default we use 'cyclos4' as database name and 'cyclos' as username and password. For production, it is recommended to change the password *.

```
cyclos.datasource.jdbcUrl = jdbc:postgresql://localhost/cyclos4
cyclos.datasource.user = cyclos
cyclos.datasource.password = cyclos
```

* Some systems do not resolve localhost and the default PostgreSQL port directly. In case of database connectivity problems you might try a URL:

cyclos.datasource.jdbcUrl = jdbc:postgresql://local_ip_address:postgresport/cyclos4
example: cyclos.datasource.jdbcUrl = jdbc:postgresql://192.168.1.1:5432/cyclos4

** Windows might not see linebreaks in the property file, if this is the case we advice you to download an more advanced text editor such as [Notepad++](#).

*** In Windows, in case of problems, you can set the cyclos.tempDir variable manual. Point it to the temp directory inside the WEB-INF directory in Cyclos. E.g. "cyclos.tempDir = C:\Program Files\Tomcat8.5\webapps\cyclos\WEB-INF\temp". In some cases even forward slashes need to be used.

Startup Cyclos

- (Re)start tomcat:
 - Stop trough `<tomcat_home>/bin/stop.bat` (Windows) or `<tomcat_home>/bin/stop.sh` (Linux).
 - Start trough `<tomcat_home>/bin/startup.bat` (Windows) or `<tomcat_home>/bin/startup.sh` (Linux).
 - Windows: you can use Tomcat monitor (available after tomcat installaton)
- When tomcat is started and Cyclos initialized browse to the web directory defined in step 5 (for the default this would be <http://localhost:8080/cyclos>). Be aware starting up Cyclos for the first time might take quite some time, because the database need to be initialized. On slow computer this could take up to 3 minutes!
- Upon the first start of Cyclos you will be asked to fill in the license information.
- After submitting the correct information, the initialization process will finish, and you will automatically login as (global) administrator.

Upgrading Cyclos

- To upgrade Cyclos follow these steps:
 - Before updating always study the release notes and changelog they are published on the Cyclos license server.
 - IMPORTANT! Make a backup of the database. New Cyclos versions generally contain new functionality that requires the database to be modified.
 - Download the latest version of Cyclos from the [license server](#).
 - Unzip the `cyclos-<version>.zip` into a temporary directory.
 - Browse to the temporary directory and rename the directory `web` to `cyclos`.
 - Copy your current `cyclos.properties` file (`<tomcat_home>/webapps/cyclos/WEB-INF/classes/cyclos.properties`) to the same place in the temporary directory.
 - Remove the directory `cyclos` from the tomcat webapps directory (`<tomcat_home>/webapps/cyclos/`).
 - Browse to the temporary directory and copy the directory `cyclos` (including its contents) into the webapps directory (`<tomcat_home>/webapps`) of the tomcat installation.
- We would also recommend to do the following:
 - Between major Cyclos versions the Cyclos API can change, please test on a local server (with the database backup) if all scripts and extensions made through the web services still work.

- In general it is a good practice to test everything before upgrading, if you test with your local database please don't forget to remove the email host and sms gateway so that the users don't receive any notifications.
- If locally everything works fine a live update can be done as described above.
- To avoid overwriting the cyclos.properties file without intention this file is named as cyclos-release.properties in the zip file. It might be interesting to study the new file to see if new settings have become available.

Problem solving

- Often problems can be easily detected by looking at the log files, the log files of tomcat can be found in the logs folder inside tomcat. There are two relevant log files:
 - The Catalina log shows all relevant information about the tomcat server itself.
 - The Cyclos log shows all relevant information about the services and tasks that run in Cyclos.
- If the logs can't help you to pin down the problem, you can search the [Cyclos forum \(installation issues\)](#) if somebody encountered a similar problem.
- If this still has no results, you can post the (relevant) part of the logs to the [Cyclos forum \(installation issues\)](#), together with a description of the problem.
- In case you locked yourself out of the system, see paragraph "[Maintenance](#)" for how to reset the admins password.
- An example of an error that sometimes occurs is "WARN RequestContextFilter – Couldn't write on the temp directory". In this case the user that started tomcat doesn't have the write permission. This can be modified in Linux by executing the following commands as root, assuming a default Tomcat 9 installation in Ubuntu, which runs as the user tomcat9 and has /var/lib/tomcat9/webapps as web apps directory:

```
chown -R tomcat9 /var/lib/tomcat9/webapps/ROOT
chmod -R 755 /var/lib/tomcat9/webapps/ROOT
```

- Another example that can occur if the server has crashed (the Tomcat service wasn't gracefully stopped) is the that your Cyclos database will be locked. This happens due to a feature we introduced in Cyclos 4.14.11 to prevent multiple Cyclos instances to connect to the same database (which can corrupt the data). This lock can be manually removed by running a SQL command in the database. For this, login to the Cyclos database (example: psql --user cyclos cyclos4), and run the following command:

```
delete from database_lock;
```

1.2. Install Cyclos as a Docker image

There is a Docker image for Cyclos, and the installation via docker is very easy, and can be accomplished with a few steps. For more details how to install Cyclos via Docker image, please, visit the [Cyclos repository on Docker hub](#).

Especially when you are using Linux, installing Cyclos using Docker will be very easy. For windows users it might be more difficult, because your system needs to supports Hardware Virtualization Technology and needs to run on 64bit. For older computers hardware virtualization might not available or needs to be set in the bios of the computer. More information about this is available [here](#). If you want to use docker for a quick preview in windows we would only recommend using it, when you have hardware virtualization already enabled on a 64bit machine. For Mac docker is available from OS X 10.8 or higher, more information can be found [here](#).

1.3. Adjustments (optional)

Adjust Tomcat/Java memory

The default memory heap size of Tomcat is very low. You can augment this in the following way:

Windows

In the bin directory of Tomcat create (if it doesn't exist) a file called setenv.bat, edit this file and add the following line:

```
set JAVA_OPTS=-Xmx1g
```

Linux

In the bin directory of Tomcat create (if it doesn't exist) a file called setenv.sh, edit this file and add the following line:

```
JAVA_OPTS="-Xmx1g"
```

Enable SSL/HTTPS

Enabling SSL is crucial on live systems, as it protects sensitive information, like passwords, to be sent plain over the Internet, making it readable by eavesdroppers. Generally it is advised to use a proxy server, like Apache or Nginx that handles HTTPS and then redirect the request to Tomcat. See the section called "Enable SSL on apache" for more details.

Otherwise, if the Tomcat server is directly used from the Internet, to enable SSL / HTTPS you first have to enable (un-comment) the https connector in the file <tomcat_home>/conf/server.xml

```
<Connector port="443" maxHttpHeaderSize="8192"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" disableUploadTimeout="true"
    acceptCount="100" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS" />
```

Generate a key with the keytool from Java:

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA -keystore /path/to/my/keystore
```

After executing this command, you will first be prompted for the keystore password. Passwords are **case sensitive**. You will also need to specify the custom password in the server.xml configuration file, as described later. Next, you will be prompted for general information about this Certificate, such as company, contact name, and so on. This information will be displayed to users who attempt to access a secure page in your application, so make sure that the information provided here matches what they will expect. Finally, you will be prompted for the key password, which is the password specifically for this Certificate (as opposed to any other Certificates stored in the same keystore file). You **MUST** use the same password here as was used for the keystore password itself. (Currently, the keytool prompt will tell you that pressing the ENTER key does this for you automatically). If everything was successful, you now have a keystore file with a Certificate that can be used by your server.

Clustering

Clustering is useful both for scaling (serving more requests) and for high availability (if a server crashes, the application continues to run). The main reason for configuring a cluster in Tomcat is to replicate HTTP sessions. Cyclos, however, doesn't use Tomcat sessions, but handles them internally. This way, there is no special Tomcat configuration to support a Cyclos cluster.

The Cyclos application, however, needs some small configurations to enable clustering. Cyclos uses [Hazelcast](#) to synchronize aspects (such as caches) between cluster servers. To enable clustering, find in cyclos.properties the line containing cyclos.clusterHandler, and set it to hazelcast.

Some extra configuration can be performed in the WEB-INF/classes/hazelcast.xml file. Basically, if the local network runs more than a single Cyclos instance, the group needs to be configured. Configure all files belonging to the same group with the same group name and password. It is also possible to change the default multicast to TCP/IP communication. Just comment the <multicast> tag and uncomment the <tcp-ip> tag, setting up the hosts / ports which will be part of the cluster. For a TCP/IP cluster, Hazelcast needs the host name / port of at least one node already in a cluster (it is not necessary to set all other nodes on each node).

Starting with Cyclos 4.14, the [Hazelcast AWS plugin](#) is bundled with Cyclos. So it is very easy to achieve auto-discovery when Cyclos is running in Amazon services. Follow the examples on

the plugin website for the configuration, which is basically disabling multicast and enabling aws.

To setup high-availability at database (PostgreSQL) level, please, refer to [this document](#).

Use Apache as frontend for Tomcat

You can use apache as a front-end / load balancer for the tomcat. This is very usefull when you have several domains configured on the server. There are several documentations and examples available on the internet, in our example we will use the mod_jk library for apache.

```
sudo apt-get install apache2 libapache2-mod-jk
```

The configuration is done on the file /etc/libapache2-mod-jk/workers.properties. By default this is configured to use the AJP port 8009, this is the default ajp port for tomcat, if you are using a different port you need to configure here.

On tomcat we need to enable the ajp connector. Edit the file tomcat/conf/server.xml and uncomment the AJP connector:

```
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

Now on apache we need to configure the virtualhost to use the ajp connector. On the virtualhost of your domain add the following lines:

```
<IfModule mod_jk.c>
    JkMount /* ajp13_worker
    JkMount / ajp13_worker
</IfModule>
```

This example uses the cyclos as ROOT application on tomcat. If you want to use something like <http://www.yourdomain.com/cyclos> we need to deploy cyclos on the webapps/cyclos directory and configure apache like this:

```
<IfModule mod_jk.c>
    JkMount /cyclos/* ajp13_worker
    JkMount /cyclos ajp13_worker
</IfModule>
```

Now restart both apache and tomcat and check if it works.

Enable SSL on apache

Enabling SSL is highly recommended on live systems, as it protects sensitive information, like passwords, to be sent plain over the Internet, making it readable by eavesdroppers. If you are using apache as a front-end for the tomcat first you need to enable the ssl module.

```
sudo a2enmod ssl
```

After enable the module we need to configure the virtualhost to use the ssl. On the virtualhost of your domain add the following lines:

```
NameVirtualhost www.yourdomain.org:443
<VirtualHost www.yourdomain.org:443>
ServerAdmin youremail@yourdomain.org
ServerName www.yourdomain.org
DocumentRoot /var/www/

<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
<Directory /var/www/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order allow,deny
    allow from all
</Directory>

ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
<Directory "/usr/lib/cgi-bin">
    AllowOverride None
    Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
    Order allow,deny
    Allow from all
</Directory>

ErrorLog /var/log/apache2/domain_error.log
LogLevel warn
CustomLog /var/log/apache2/domain_access.log combined

<IfModule mod_ssl.c>
    SSLEngine on
    SSLProtocol ALL -SSLv2 -SSLv3
    SSLHonorCipherOrder On
    SSLCipherSuite ECDHE-RSA-AES128-SHA256:AES128-GCM-SHA256:RC4:HIGH:!MD5:!aNULL:!EDH
    ServerSignature Off
    BrowserMatch ".*MSIE.*" \
        nokeepalive ssl-unclean-shutdown \
        downgrade-1.0 force-response-1.0
    SSLOptions +FakeBasicAuth +ExportCertData +StrictRequire
    SSLCertificateFile /etc/ssl/certs/yourcertificate.crt
    SSLCertificateKeyFile /etc/ssl/private/yourkey.key
</IfModule>
</VirtualHost>
```


Now we need to generate the certificate, in this example we will use a self-signed certificate, normally used to test your new SSL implementation

Generate a Private Key

The utility "openssl" is used to generate the key and CSR. This utility comes with the OpenSSL package and is usually installed under /usr/local/ssl/bin. If the utility was installed elsewhere, these instructions will need to be adjusted accordingly.

Type the following command at the prompt:

```
openssl genrsa -des3 -out yourkey.key 2048
```

Generate a CSR (Certificate Signing Request)

Once the private key is generated a Certificate Signing Request can be generated. The CSR is then used in one of two ways. Ideally, the CSR will be sent to a Certificate Authority, such as Thawte or Verisign who will verify the identity of the requestor and issue a signed certificate. The second option is to self-sign the CSR, which will be demonstrated in the next section.

During the generation of the CSR, you will be prompted for several pieces of information. These are the X.509 attributes of the certificate. One of the prompts will be for "Common Name (e.g., YOUR name)". It is important that this field be filled in with the fully qualified domain name of the server to be protected by SSL. If the website to be protected will be https://public.akadia.com, then enter public.akadia.com at this prompt. The command to generate the CSR is as follows:

```
openssl req -new -key yourkey.key -out yourcertificate.csr

Country Name (2 letter code) [GB]:CH
State or Province Name (full name) [Berkshire]:Bern
Locality Name (eg, city) [Newbury]:Oberdiessbach
Organization Name (eg, company) [My Company Ltd]:Akadia AG
Organizational Unit Name (eg, section) []:Information Technology
Common Name (eg, your name or your server hostname) []:public.akadia.com
Email Address []:martin dot zahn at akadia dot ch
Please enter the following extra attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Remove Passphrase from Key

One unfortunate side-effect of the pass-phrased private key is that Apache will ask for the pass-phrase each time the web server is started. Obviously this is not necessarily convenient as someone will not always be around to type in the pass-phrase, such as after a reboot or crash. mod_ssl includes the ability to use an external program in place of the built-in pass-

phrase dialog, however, this is not necessarily the most secure option either. It is possible to remove the Triple-DES encryption from the key, thereby no longer needing to type in a pass-phrase. If the private key is no longer encrypted, it is critical that this file only be readable by the root user! If your system is ever compromised and a third party obtains your unencrypted private key, the corresponding certificate will need to be revoked. With that being said, use the following command to remove the pass-phrase from the key:

```
cp yourkey.key yourkey.key.org
openssl rsa -in yourkey.key.org -out yourkey.key
```

The newly created yourkey.key file has no more passphrase in it.

Generating a Self-Signed Certificate

At this point you will need to generate a self-signed certificate because you either don't plan on having your certificate signed by a CA, or you wish to test your new SSL implementation while the CA is signing your certificate. This temporary certificate will generate an error in the client browser to the effect that the signing certificate authority is unknown and not trusted.

To generate a temporary certificate which is good for 365 days, issue the following command:

```
openssl x509 -req -days 365 -in yourcertificate.csr -signkey yourkey.key \
-out yourcertificate.crt
```

Installing the Private Key and Certificate

When Apache with mod_ssl is installed, it creates several directories in the Apache config directory. The location of this directory will differ depending on how Apache was compiled.

```
cp yourcertificate.crt /etc/ssl/certs/yourcertificate.crt
cp yourkey.key /etc/ssl/private/yourkey.key
```

Now restart apache and check if it works.

Configuring Cyclos to work behind a proxy / load balancer

The easiest configuration for a load balancer is Apache connecting to Tomcat using the AJP protocol. In this case, the original request is forwarded to Tomcat as is, keeping the original client IP address and URL. However, in most other cases, the load balancer works as a proxy, sending a new HTTP to Tomcat and forwarding the response to the client. Examples of such proxies include Apache with mod_proxy, Nginx, haproxy and Amazon's Elastic Load Balancing. In either way, generally the proxy will have the server certificate and will terminate the SSL

connection with the client. The second request, to Tomcat, will generally use a plain HTTP connection.

That means the client IP address received by Cyclos, as well as the request URL are different than the original request performed by the client. As Cyclos uses the client IP for logging and blocking in case of abuse, this would lead Cyclos to block the proxy, preventing any further request. However, the proxy will add some extra request headers, with information about the original request. Cyclos then needs to be configured to read both the IP address and which was the connection protocol used by the original request (HTTP or HTTPS) from those headers, instead of directly from the incoming HTTP request. As such, the following settings in `cyclos.properties` are needed:

- `cyclos.header.remoteAddress`: Specifies the name of the header which contains the original client's IP address. The name of this header is usually `X-Forwarded-For`.
- `cyclos.header.protocol`: Specifies the protocol name (`http` or `https`) used on the original request. The name of this header is usually `X-Forwarded-Proto`.

The following cases are handled by Cyclos to match a specific network / configuration from the request URL:

- A request to Tomcat using the root URL specified in a parent configuration (normally the network default). For example, if the network default configuration's root URL is `http://cyclos-net.com`, any requests to `http://cyclos-net.com/*` will match that configuration;
- A request to Tomcat using the root URL specified in a parent configuration plus a specific path of an inherited configuration. Following the previous example, if the child configuration has a custom path of `config`, any requests to `http://cyclos-net.com/config/*` will match that configuration;
- If no custom URL is matched, requests having first subpath (after the web application context path) equals the network internal name. For example, if Cyclos is deployed in a Tomcat under the context path `cyclos` and it has a network called `main`, any requests to `http://localhost:8080/cyclos/main/*` will match this network;
- Same as previous, but with a specific configuration path. Following the previous example, if that network has a configuration with path `config`, requests to `http://localhost:8080/cyclos/main/config/*` will match this configuration;
- Also, the name `global` is reserved as a network internal name. For example, requests to `http://localhost:8080/cyclos/global/*` will be considered in global mode;
- Still if no custom URL is matched, if no network internal name is given and there is a default network, the network internal name can be omitted. For example, requests to `http://localhost:8080/cyclos/*` will be considered in the default network.

When Tomcat is behind a proxy, probably it will never receive requests using the original public URL. Hence, only matching by network (and optionally, configuration paths) will be

used. Still, networks should correctly set the root url in their configurations because they are used to generate full URLs in the server side, for example, when sending links in e-mails or resolving image URLs in rich texts.

Following are common cases that could be configured for a proxy, all assuming Cyclos is deployed to a Tomcat accessible by the proxy via `http://tomcat:8080/cyclos`:

- The proxy handles the (sub)domain of a specific network. For example, it handles requests to `https://main.my-cyclos.com` and forwards them to `http://tomcat:8080/cyclos/main`. The network in Cyclos must have the internal name `main` in this case;
- The proxy handles a specific (sub)domain for the global mode. For example, it handles requests to `https://global.my-cyclos.com` and forwards them to `http://tomcat:8080/cyclos/global`;
- The proxy handles a multi-network entry point on a (sub)domain of a specific network. For example, it handles requests to `https://www.my-cyclos.com` and forwards them to `http://tomcat:8080/cyclos`. So, requests to `https://www.my-cyclos.com/main` will match the `main` network, while requests to `https://www.my-cyclos.com/alt` will match the `alt` network.

Reserved names that cannot be used in proxy paths

There are reserved paths in Cyclos that cannot be used as path in proxies. For example, a proxy could handle requests to `https://www.my-project.com/app` and redirect them to `http://tomcat:8080/cyclos`. In this case, the `/app` path part is public, used by clients, but never visible on Tomcat. To handle such cases, the list of reserved paths is used to generate the correct URIs for scripts and stylesheets, and having any of the reserved paths in the proxy would prevent the URI generation from working correctly.

The list of reserved paths is:

```
cyclos
cyclos.gwt
fonts
js
pay
consent
classic
ui
.well-known
robots.txt
sitemap.xml
sitemap-index.xml
sitemap.xstl
network-information-id.html
activate-access-client
external-redirect-callback
identity
run
content
web-rpc
```

```
java-rpc
api
sms
push-notifications
global
redirect
mobile-redirect
```

Enabling Google Maps

Cyclos supports displaying maps using [Google Maps](#). This has to be enabled in the Cyclos configuration. Starting in June 2016, the Google Maps requires API Keys in order to use. For details on the free daily quota for map views and geocode requests, see [this page](#). As of March 2018, the free quota is 25,000 map views and 2,500 geocode requests. Cyclos uses the geocode requests to map the user-informed address fields to a position (latitude/longitude).

There are 2 API keys that can be set in the Cyclos configuration: The server-side API key and the browser API-key. Each one needs to be generated in the [API Manager](#).

- Enabling the APIs: on the "Library" menu, search for the following APIs and enable them: "Maps JavaScript API", "Maps Static API" and "Geocoding API";
- Creating the API keys: on the "Credentials" menu, choose "Create credentials", then choose "API key". Choose "Server key" and specify a name for it. Save and then create a new one, this time as "Browser key".

Once the API keys are ready, they can be copied / pasted into the Cyclos configuration, on the corresponding "Google maps server API" and "Google maps browser API" fields. The [API Manager](#) allows monitoring of requests performed by each API.

On the main Cyclos web application, addresses are geolocated on the client-side, before saving it. In such cases, the browser API key is used. However, sometimes addresses can be saved without being geolocated, either by third-party software or by importing new users into Cyclos. In such cases, a background task will attempt to geolocate them (might take up to 24 hours for this) using the server API key.

External content storage

Storage types

You have the possibility to configure the storage type to be used for images, documents, imported files and custom fields of type image/file. Additionally, in the configuration details page you can define a specific storage directory for individual (i.e user) documents and in the custom field details page do the same for custom fields. The 'Amazon S3', 'Google Cloud Storage' and 'Filesystem' implementations has support for storage directories, please read below to know how to configure it.

Cyclos comes with four implementations out of the box:

- Database: the content is stored in conjunction with all data in the database. This is the default implementation.
- File system: the content is stored outside the database in specific paths.
- Amazon S3: Amazon Simple Storage Service, the content is stored outside the database in specific buckets.
- Google Cloud storage: the content is stored outside the database in specific bucket.

Besides the built-in implementations you can create your own custom implementation. To do that you must create a Java class implementing [org.cyclos.impl.storage.StoredFileContentManager](https://github.com/cyclos-project/cyclos/blob/master/src/main/java/org/cyclos/impl/storage/StoredFileContentManager.java)

The following are the properties you need to configure in the cyclos.properties

Storage type property

- cyclos.storedFileContentManager: specifies the storage type to be used, it could have the following values: db, file, s3, gcs or the fully-qualified name of a custom Java class implementing [org.cyclos.impl.storage.StoredFileContentManager](https://github.com/cyclos-project/cyclos/blob/master/src/main/java/org/cyclos/impl/storage/StoredFileContentManager.java)

Database storage specific properties

There are no additional properties to be configured.

File system storage specific properties

- cyclos.storedFileContentManager.rootDir: the root directory where the contents will be stored.
- cyclos.storedFileContentManager.directories: comma separated list of folder (storage directory) names that will be created as children of the rootDir and where individual documents and image/file custom field values can be stored.
- cyclos.storedFileContentManager.maxSubDirs: the maximum count of directories to be created below the root directory or a specific storage directory where the content will be stored.

Amazon S3 storage specific properties:

- cyclos.storedFileContentManager.bucketName: the name of the default bucket that will be created (if it doesn't exist) and where the content will be stored.
- cyclos.storedFileContentManager.regionName: the name of the default region where the buckets will be created.
- cyclos.storedFileContentManager.directories: comma separated list of bucket (storage directory) names where individual documents and image/file custom field values can be stored.
- cyclos.storedFileContentManager.accessKeyId: the AWS access key.

- `cyclos.storedFileContentManager.secretAccessKey`: the AWS secret access key.

If you need to create a bucket in a different region than the default one then you need to define a property of the form:

`cyclos.storedFileContentManager.regionName.bucket_name=specific_region_name`

Google Cloud storage specific properties

- `cyclos.storedFileContentManager.bucketName`: the name of the bucket created on Google Cloud, and where the content will be stored.
- `cyclos.storedFileContentManager.credentialsFile`: path to JSON key file downloaded when create the service account

Storage migrator utility class

If you already have a running Cyclos instance and want to change the storage type to use then there is an utility class that will allow to migrate the contents from the current storage to a new one. To use it you must have Java configured in your path then go to the `<TOMCAT_DIR>/webapps/<cyclos_dir>` directory and execute:

```
java -cp "WEB-INF/classes:../../lib/*:WEB-INF/lib/*" \
    org.cyclos.impl.storage.utils.StoredFileContentMigrator
```

and just follow the instructions shown in the usage help.

Using Elasticsearch

If the database performance is unacceptable when performing complex searches on users, advertisements or records, using keywords or geo-distance filters, it is advised to use Elasticsearch to handle such cases. For more details, see Section 2.1, “Using Elasticsearch”.

Logging

By default, Cyclos logs access to services, as well as background tasks, to files. But for production systems, starting with Cyclos 4.11, it is recommended to set logging to an external database. The log destination is configured in `cyclos.properties`, through the `cyclos.log` property. The value can be either `file`, `db` or `none`.

Other properties depend on the log provider:

- For `cyclos.log=file`:
 - `cyclos.log.dir`: The directory where to write logs. Supports `%t` as the temporary directory, `%w` as the web application directory and `%n` as the network internal name.
 - `cyclos.log.maxFiles`: The log files are rotating. This setting indicates the maximum number of files per network.
 - `cyclos.log.maxFileSize`: The maximum size per log file. Examples are 2M or 500K.

- For `cyclos.log=db`: All properties under `cyclos.log.datasource` are used to configure the `datasource`, just like the regular `cyclos.datasource`-prefixed properties. It can be configured to point to an external PostgreSQL database or to the same database used by Cyclos. If an external database, just like the regular database, it must be previously created, but the tables are created automatically on the server startup.

Logs are, by default, asynchronous, so the requests are not delayed until the log is written. This is controlled by the `cyclos.log.threads` property, which sets the number of threads used to concurrently write logs. However, if the server crashes, some log entries may be lost. If the number of threads is set to zero, logs will be synchronous, delaying each response, specially under heavy load, but guaranteeing that logs are written. If using a database log, make sure to set the maximum number of connections equal to the number of threads. When using synchronous logging, the maximum connections to the log database will also limit the number of concurrent request, so, an extra care is needed.

A final note on the log database: there are currently 2 tables - `service_logs` and `task_logs`. The `service_logs` store a row each time a client calls any Cyclos service, while the `task_logs` stores a row for each background task execution. The tables have no indexes, and will store parameters and results as the PostgreSQL's JSON type, so that INSERT operations run as fast as possible. However, for searching data, the JSONB type (binary JSON) is more efficient, and supports indexing. When searching the table with too many logs, instead of searching directly on `service_logs`, it is recommended to create a new table, and query it instead. This new table should have the same columns as `service_logs`, but with JSONB columns instead. Also, add indexes according to your query. A final note on log tables is that they tend to quickly grow in size, so you may need to periodically (according to the database data volume) move old data to another database in order to not impact the logging performance. Also, don't forget to vacuum the table after deleting old records.

1.4. Maintenance

Backup

All data in Cyclos is stored in the database. Making a backup of the database can be done using the `pg_dump` command. The only file that you need to back-up (only once) will be the `cyclos.properties` configuration file. The database can be backed up manually as follows:

```
pg_dump --username=cyclos --password -hlocalhost cyclos4 > cyclos4.sql
```

Note: in this example the name of the database is `cyclos4`, the username `cyclos` and the command will prompt for the password of the `cyclos` user.

Restore

If you want to start using cyclos with the data from a backup. You can just import the backed up database. In this example the name of the database is `cyclos4` the username `cyclos` and

command will prompt for the password cyclos the name of the backup is cyclos4.sql make sure to specify the path if your not in the same directory as the file:

```
psql --username=cyclos --password -hlocalhost cyclos4 < cyclos4.sql
```

Note: in this example the name of the database is cyclos4, the username cyclos and command will prompt for the password cyclos, the name of the backup is cyclos4.sql (make sure to specify the path if your not in the same directory as the file).

Backup / restore of very large databases

When the database is very large (specially if it have a lot of images) it is possible to use a custom format for the dump file, which makes the dump file smaller. To use it, backup with the following command:

```
pg_dump --username=cyclos --password -Fc -hlocalhost cyclos4 > cyclos4.sql
```

To restore the dump, another command needs to be used as well:

```
pg_restore --username=cyclos --password -Fc -hlocalhost -d cyclos4 cyclos4.sql
```

Reset admin password directly on database

If you lost the password of your global administrator, it is still possible to update the value on database directly. To reset the password to 1234, run the following sql in the postgresql query tool (psql).

```
update passwords
set value=' $2a$10$yM.uw9jC7C1DrRGUhqUc3eSR6FCJH0.HdDt3CJs8YL56iATHcXH7.'
where user_id = (select id from users where username='admin')
and status = 'ACTIVE'
and password_type_id in (select id from password_types where input_method = 'TEXT_BOX' and
password_mode = 'MANUAL');
```

Please make sure to replace the name 'admin' to the username used for the global administrator. Also a common mistake is that people forget to login as global administrator into the global url e.g. <https://www.cyclos-domain/global>.

Sending database to third parties

If Cyclos or a third party asks you to share the database with them it's vital for security that the passwords are removed from the database. The passwords in Cyclos are hashed with one of the strongest algorithms available, but still passwords can be theoretically recovered using brute force (although very unlikely). If the database falls into the wrong hands some users might get compromised. Therefore it is always recommended to follow this procedure before sharing the database with other parties:

- Make a dump of the database (see [Backup](#));

- Restore the database in another (temporarily) database so the data can be changed without risking to change live data (see [Restore](#));
- Run the following command to reset all passwords to '1234':

```
update passwords
set value = '$2a$04$rDPKseEiJhYdjx9RogW2tuzNX4TKG1wce79ooEXiA5.mJF.ooZY/2'
where status <> 'OLD'
and password_type_id in (
  select id
  from password_types
  where input_method = 'TEXT_BOX'
  and password_mode = 'MANUAL'
);
```

- It is also recommended to remove sensitive customer information from the database. For example all email addresses can be changed to a non existing email address as follows:

```
update users
set email = concat(username, '@test.com')
where email is not null;
```

- Then dump the database again. This file can then be sent to the third-party.

Removing all data from a network

A common practice for a first-time configuration of Cyclos, specially with a complex structure for accounts, configurations, products and groups, is to configure all the system, and create some test users and payments. However, after finishing configuration, it might be desirable to remove all users and transfers (payments) from that network, leaving only administrators and configurations. Alternatively, it might be desirable to completely delete an entire network.

Starting with Cyclos 4.11, an interactive utility is included in Cyclos, which can be used for both cases. Please, be advised to perform a full database dump before running the utility, and have Cyclos stopped before running it. To run the utility, go to the <TOMCAT_DIR>/webapps/<cyclos_dir> directory and execute:

```
java -cp "WEB-INF/classes:../../lib/*:WEB-INF/lib/*:/path-to-tomcat/lib" \
  org.cyclos.db.DeleteNetworkData
```

Then follow the instructions presented on the console. When a a lot of data is removed, it might be desirable to run a full vacuum in the database. This operation might take a while. An example on how to run it is:

```
$ vacuumdb --full $DATABASE_NAME
```

2. Full text searches

This chapter covers how [full text searches](#) work in Cyclos, and how to fine-tune them. Full text searches allows retrieving documents using its words, returning documents that match a given textual query (often related as keywords in Cyclos). The full text engine processes words both when indexing (calculating the words on documents) and querying (transforming an input text in a way it matches indexed documents). Some examples of such processing include:

- Removing stopwords - words which are too common in a given language, and likely be contained in multiple documents. In English, 'a', 'the' and 'is' could be example of stopwords.
- Changing words to a common form, or stemming. For example, in English, 'sailing', 'sailed', 'sailor' could all be stored as 'sail'.

Currently the following data types are searched with full text queries when using keywords:

- Users: The profile fields which are set in the user products (or group's permissions in case of administrators) marked to include in user keywords will be searched. Also supports geo-distance searches;
- Advertisements: The advertisement title, description and custom fields, plus the user (owner) profile fields which are set in the user products marked to include in advertisements keywords will be searched. Also supports geo-distance searches;
- Records: The record custom fields, plus the user profile fields which are set in the user products (or group's permissions in case of administrators) marked to include in record keywords will be searched;
- Translation keys: The translation keys are indexed to allow searching for the current, original or English translations. As the keys are normally stored in files, when Cyclos starts, a database table is populated and indexed.

By default, Cyclos uses the native [PostgreSQL's full text indexing](#) capabilities. Also, for geo-spatial distance filters, Cyclos uses [PostGIS](#). But it is also possible to use an external Elasticsearch server to perform these searches. For details on the Elasticsearch integration see the Section 2.1, "Using Elasticsearch" section.

As the PostgreSQL native query syntax can be too much formal for end users, a query preprocessor is included in Cyclos, such that the following variants are supported:

- a b: The value must have words that either start with a or b;
- a +b: The value must have words that start with both a and b;
- a -b: The value must have words that start with a and no words that start with not b;

- "a b": The value must have exactly a followed by b, in this exact order. This is only supported if the database is Postgres 9.6 or later;
- Also, parenthesis can be used to group expressions, such as ((a b) +(c -d)).

Previously, Cyclos used distinct PostgreSQL text search configurations, called dictionaries in Cyclos, that defined the kind of language processing to be applied. This, however, sometimes led to confusion, as using the wrong dictionaries could actually prevent finding existing data. For example, in dutch, the phrase "van der meer" would be totally discarded, as they are all stopwords. Even though, the same phrase can appear in names, requiring the default dictionary to be selected as well.

Starting with Cyclos 4.12, the actual language processing is only performed when using Elasticsearch. In PostgreSQL, Cyclos will always use a text search configuration that doesn't remove any stopword, but removes all diacritics. So, for example 'façade' is stored as 'facade'. This should be more than ok for most systems, specially because we search by prefix by default. So, for example, seaching 'car' will match all of these: 'car', 'cars' and 'cartoon', but ranking 'car' better. For more complex language processing, use Elasticsearch.

2.1. Using Elasticsearch

Searching on PostgreSQL with large databases, using several custom fields and distance filter may present unacceptable performance. For such cases, or for more advanced language analysis, starting with Cyclos 4.12, it is possible to perform such searches using [Elasticsearch](#), a well-known product which is very fast, and designed to scale horizontally. Some cloud providers, such as Amazon, even provides [Elasticsearch as a service](#).

Important: Starting with Cyclos 4.14, the required Elasticsearch version is 7.x. Previous versions are not supported.

The Elasticsearch server / cluster needs to be deployed in a server accessible to Cyclos via its REST API. Then, in cyclos.properties, add the following properties:

- cyclos.searchHandler = elasticsearch: This enables the Elasticsearch integration. Make sure to comment out or remove the line cyclos.searchHandler = db;.
- cyclos.searchHandler.host: One or more (comma-separated) hosts (protocol://hostname:port) of the Elasticsearch server;
- cyclos.searchHandler.pathPrefix: Path within the host on which the Elasticsearch server responds to;
- cyclos.searchHandler.user: Optional user for HTTP basic auth;
- cyclos.searchHandler.password: Optional password for HTTP basic auth.
- cyclos.searchHandler.shards: How many shards the indexes should be split in.
- cyclos.searchHandler.replicas: How many replicas per shard.

Once set-up, when restarted, Cyclos will attempt to find the following Elasticsearch indexes, and, if not found, will create them and index all entities in the database: users, ads and records (translation keys are always searched in Postgresql). The indexing will be executed on the background, and it can take a few minutes, depending on the database size, to index all data.

Textual searches (referred as keywords in Cyclos) are passed to Elasticsearch using its [Simple Query String](#) syntax, which is like the one employed by Cyclos when searching on the database, but more powerful.

Language processing

The actual language processing (such as removing stopwords and stemming text) is performed only on the following fields: advertisement title and description, and custom fields whose setting "Value match" is set to "Language". Note that the "Value match" field will only show up in custom fields when Elasticsearch is used. When handling searches in the database, only the checkbox "Use exact matching on search filters" will show up.

Language-analyzed fields are stored multiple times in the index: one using Elasticsearch Standard Analyzer (to prevent mismatches or when searching as users which uses another language), and another one for each language the owner of the data can have. So, for example, an advertisement owned by a user whose configuration allows English, Portuguese and French will have the field stored in all such analyzers, plus the standard.

When searching for the data, Cyclos uses all languages the logged user has, plus the standard analyzer. So, following the same previous example, if a logged user had only the French language, it would search in both the standard analyzer field, plus the French one. And if another user could see that advertisement, but have only the Spanish language, they would search in both Spanish (in which the no data exist) and in standard, and would ultimately find the advertisement using the standard analyzer.

Reindexing

Being a separated data store, the data on PostgreSQL database and on Elasticsearch might become de-synchronized. The database is always considered correct, and is the definitive trusted store. The data on Elasticsearch is updated automatically, as soon as the corresponding user, advertisement or record is modified on the database. But it might happen that an update request fails, or that the Elasticsearch server to be offline for some time.

Important: If you have modified a profile field (its value or the internal name) then you must reindex users, andvertisements and records.

To handle these cases, Cyclos offers methods that can be executed by scripts, directly through the menu System > Tools > Run script. Here are some examples:

Reindex ALL data on ALL indexes:

```
searchHandler.reindex()
```

Reindex ALL data on users:

```
userSearchHandler.reindex()
```

Reindex ALL data on advertisements:

```
adSearchHandler.reindex()
```

Reindex ALL data on records:

```
recordSearchHandler.reindex()
```

Other examples:

```
//Reindex ALL records of a given record type
import org.cyclos.entities.users.RecordType
def recordType = entityManagerHandler.find(RecordType, "recordTypeInternalName")
recordSearchHandler.reindexByType(recordType)

//Reindex a single user
import org.cyclos.entities.users.User
def user = conversionHandler.convert(User, 'loginName')
userSearchHandler.index(user)

//Reindex a single advertisement by external (masked) id (would be similar to records)
import org.cyclos.entities.marketplace.BasicAd
def ad = entityManagerHandler.find(BasicAd, unmaskId(123456789L))
adSearchHandler.index(ad)
```

3. Web services

Here you will find information on how to call Cyclos services from 3rd party applications.

3.1. Introduction

Cyclos 4 provides two distinct web service interfaces: the REST API (starting with version 4.6) and the WEB-RPC. In both interfaces the security layer is exactly the same (hence, both grant exactly the same permissions), and users are authenticated in the same way, as described below.

REST API

This API is implemented with REST concepts in mind, using JSON data for input and output, making it easier for developers to leverage existing knowledge when using it. It is documented using [Swagger](#), which is the base for [Open API](#). A detailed reference documentation is available online on each Cyclos installation, at <cyclos-root-url>[/network]/api. For example, the Cyclos Demo API is available at <http://demo.cyclos.org/api>. It is possible to disable the API reference documentation page by setting `cyclos.rest.reference = false` in `cyclos.properties`.

The REST API contains a subset of the Cyclos functionality. New functionality will be added on demand, in a cautious manner, as each path, parameter and data model needs to be planned to fit the target architecture. This is the preferred interface for clients to connect to Cyclos, as it should be stable between Cyclos releases, and provides better documentation and tooling support, as Swagger provides tools, for example, to generate clients for distinct languages / frameworks.

WEB-RPC

The WEB-RPC provides access to the entire service layer in Cyclos 4. It is, for example, used by the main Cyclos web application. The available services are [linked here](#). This page also links to the changes between each Cyclos 4 release. As the service layer is exported directly, it tends to contain more changes between releases than the REST interface.

The WEB-RPC also uses JSON objects for input / output, and provides, besides plain HTTP calls, a Java and a PHP client libraries.

Authentication in web services

Regardless the web service interface (REST or WEB-RPC), users are authenticated either as user / password (stateless), logging-in with a session (stateful) or using access clients (stateless). The way authentication data is passed from client to server depends on whether the clients are using the Java API, the PHP API or WEB-RPC calls.

User and password

In this mode, a principal (user identification method), which can be the login name, e-mail, mobile phone, custom field, account number or token value (card number), depending on the channel configuration, is sent on each request together with the password (live systems should be over HTTPS, so should be secure). The drawback is that the username and password need to be stored in the client application, and changing the password on the web (if the same password type is used) will make the application stop working.

Login with a session

In this mode, a first request is made to [LoginService.login\(\)](#) operation, returning a session token. Subsequent requests should pass this session token instead in the subsequent requests. Notice that the first request should be authenticated with user and password. To finish a session, a request to [LoginService.logout\(\)](#) using the session token invalidates the session.

Access clients

Access clients can be configured to prevent the login name and password to be passed on every request by clients, decoupling them from the actual user password, which can then be changed without affecting the client access, and improving security, as each client application has its own authorization token.

To configure access clients, first a new identification method of this type must be created by administrators. Then, in a member product of users which can use this kind of access, permissions over that type should be granted. Finally, the user (or an admin) should create a new access client in Cyclos main access, and get the activation code for it. The activation code is a short (4 digits) code which uniquely identifies an access client pending activation for a given user. To use the access client, on the application side (probably a server-side application or an interactive application), an HTTP POST request should be performed, with the following characteristics:

- URL: <cyclos-root-url>[/network]/activate-access-client
- Standard basic authentication header: Passing the username and password
- Request body: The body content must be the activation code

The result will be a token which should be passed in requests. The activation process should be done only once, and the token will be valid until the access client in Cyclos is blocked or disabled.

Here is an example which can be called by the command-line program curl:

```
curl http[s]://<cyclos-root-url>[/network]/activate-access-client \  
-u "<username>:<password>"
```



```
-d "<4-digit code>"
```

The generated token will be printed on the console, and should be stored on the client application to be used on requests.

Additionally, clients can improve security if they can have some unique identifier which can be relied on, and don't need to be stored. For example, Android devices always provide an unique device identifier. In that case, this identification string can be passed on the moment of activation, and will be stored on the server as a prefix to the generated token. The server will return only the generated token part, and this prefix should be passed on requests together with the generated token. The prefix is passed in the activation command, having the body of the request as: [4-digit activation code]:prefix. So, for example:

```
curl https://www.some-cyclos-instance.com/activate-access-client \
-u "john:johnpassword"
-d "1234:XYZW"
```

Imagining the server returns the fictional token ABCDEFG (the actual token is 64 characters long), the token to be used on requests would be XYZWABCEFG.

Alternatively, it is possible to do a request authenticated by username and password to the [AccessClientService.activate\(\)](#) web service method, passing the activation code and prefix parameters. This can be more convenient for client applications that activate an access client interactively, for example, when the end user types in his username, password and 4-digit activation code.

Channels

Channels can be seen as a set of configurations for an access in Cyclos. There are some built-in channels, and additional ones can be created. The built-in channels are:

- Main web: The main web application. The internal name is main.
- Mobile: The Cyclos (or another 3rd party) mobile application. The internal name is mobile.
- Web services: Is the default channel for clients using any web service client. The internal name is webServices.
- Pay at POS: Special channel not used by external applications, but assumed on the receive payment operation. Is a temporary access where the payment itself takes place as if the payer is logged in on this channel, not allowed to be passed on a client request.
- SMS operation: Channel used by SMS operations, called by SMS gateways. Is not allowed to be passed on a client request.

By default, the channel used on any web service (regardless the interface or user authentication mode) is "Web services". It is possible to specify another channel, for example, with third party web applications (handled as Main web) or third party mobile applications. In such cases, the channel internal name must be passed on each request, but the specific way to pass it depends on whether client is using REST / WEB-RPC, Java API or PHP API.

3.2. Java clients

Cyclos provides native Java access to services, which can be used on 3rd party Java applications. Starting with Cyclos version 4.11, Java 11 is required for clients to use Cyclos.

Dependencies

In order to use the client, you will need some JAR files which are available in the download bundle, on the cyclos-4.x.x/web/WEB-INF/lib directory. Not all jars are required, only the following:

- cyclos-api.jar
- cyclos-api-core.jar
- jackson-core-*.jar
- jackson-databind-*.jar
- jackson-annotations-*.jar
- httpclient-*.jar
- httpcore-*.jar
- commons-collections-*.jar
- commons-logging-*.jar
- commons-codec-*.jar
- guava-*.jar

Those jars, except the cyclos-api.jar and cyclos-api-core.jar, are provided by the following projects:

- [Apache HttpComponents](#), distributed under the [Apache 2.0 license](#).
- [Apache Commons](#), distributed under the [Apache 2.0 license](#).
- [Jackson Project](#), distributed under the [Apache 2.0 license](#).
- [Guava](#), distributed under the [Apache 2.0 license](#).

Using services from a 3rd party Java application

The Java client for Cyclos 4 uses JSON as the data interchange format, Jackson for serializing and deserializing JSON to Java objects and vice-versa and Apache HttpComponents to communicate with the server and invoke the web services. A dynamic proxy for the service interface is obtained and methods can be invoked on it as if it were a local object. The proxy, however, passes the parameters to the server and returns the result back to the client. The Cyclos 4 API library provides the [org.cyclos.server.utils.HttpServiceFactory](#) class, which is used

to obtain the service proxies, and is very easy to use. With it, service proxies can be obtained like this:

```
HttpServiceFactory factory = new HttpServiceFactory();
factory.setRootUrl("https://www.my-cyclos.com/network");
factory.setInvocationData(HttpServiceInvocationData.stateless("username", "password"));
// OR factory.setInvocationData(HttpServiceInvocationData.stateful("session token"));
// OR factory.setInvocationData(HttpServiceInvocationData.accessClient("access client token"));
AccountService accountService = factory.getProxy(AccountService.class);
```

In the above example, the [AccountService](#) can be used to query account information. The permissions are the same as in the main Cyclos application. The user may be either a regular user or an administrator. When an administrator, will allow performing operations over regular users (managed by that administrator). Otherwise, the web services will only affect the own user.

To specify a channel other than Web Services, call `setChannel(name)` on the `HttpServiceInvocationData` before passing it to the factory.

Examples

Configure Cyclos

All following examples use the following class to configure the web services:.

```
import org.cyclos.server.utils.HttpServiceFactory;
import org.cyclos.server.utils.HttpServiceInvocationData;

/**
 * This class will provide the Cyclos server configuration for the web service
 * samples
 */
public class Cyclos {

    private static final String ROOT_URL = "http://localhost:8888/england";

    private static HttpServiceFactory factory;

    static {
        factory = new HttpServiceFactory();
        factory.setRootUrl(ROOT_URL);
        factory.setInvocationData(HttpServiceInvocationData.stateless("admin", "1234"));
    }

    public static HttpServiceFactory getServiceFactory() {
        return factory;
    }

    public static HttpServiceFactory getServiceFactory(
        HttpServiceInvocationData invocationData) {
        HttpServiceFactory factory = new HttpServiceFactory();
        factory.setRootUrl(ROOT_URL);
    }
}
```

```

        factory.setInvocationData(invocationData);
        return factory;
    }
}

```

Search users

```

import org.cyclos.model.users.users.UserQuery;
import org.cyclos.model.users.users.UserVO;
import org.cyclos.model.users.users.UserWithFieldsVO;
import org.cyclos.services.users.UserService;
import org.cyclos.utils.Page;

/**
 * Provides a sample on searching for users
 */
public class SearchUsers {

    public static void main(String[] args) throws Exception {
        UserService userService = Cyclos.getServiceFactory().getProxy(UserService.class);

        // Search for the top 5 users by keywords
        UserQuery query = new UserQuery();
        query.setKeywords("consumer");
        query.setIgnoreProfileFieldsInList(true);
        query.setPageSize(5);
        Page<UserWithFieldsVO> users = userService.search(query);

        System.out.printf("Found a total of %d users\n", users.getTotalCount());
        for (UserVO user : users) {
            System.out.printf("%s\n", user.getDisplay());
        }
    }
}

```

Search advertisements

```

import org.cyclos.model.marketplace.advertisements.BasicAdQuery;
import org.cyclos.model.marketplace.advertisements.BasicAdVO;
import org.cyclos.services.marketplace.AdService;
import org.cyclos.utils.Page;

/**
 * Provides a sample on searching for advertisements
 */
public class SearchAds {

    public static void main(String[] args) throws Exception {
        AdService adService = Cyclos.getServiceFactory().getProxy(AdService.class);
        BasicAdQuery query = new BasicAdQuery();
        query.setKeywords("Gear");
        query.setHasImages(true);
        Page<BasicAdVO> ads = adService.search(query);
        System.out.printf("Found a total of %d advertisements\n", ads.getTotalCount());
    }
}

```

```

        for (BasicAdVO ad : ads) {
            System.out.printf("%s\nBy: %s\n%s\n-----\n",
                ad.getName(), ad.getOwner().getDisplay(),
                ad.getDescription());
        }
    }
}

```

Register user

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import org.cyclos.model.system.fields.CustomFieldDetailedVO;
import org.cyclos.model.system.fields.CustomFieldPossibleValueVO;
import org.cyclos.model.users.addresses.UserAddressDTO;
import org.cyclos.model.users.fields.UserCustomFieldDetailedVO;
import org.cyclos.model.users.fields.UserCustomFieldValueDTO;
import org.cyclos.model.users.groups.BasicGroupVO;
import org.cyclos.model.users.groups.GroupVO;
import org.cyclos.model.users.phones.LandLinePhoneDTO;
import org.cyclos.model.users.phones.MobilePhoneDTO;
import org.cyclos.model.users.users.PasswordRegistrationDTO;
import org.cyclos.model.users.users.PasswordRegistrationData;
import org.cyclos.model.users.users.RegistrationStatus;
import org.cyclos.model.users.users.UserDataParams;
import org.cyclos.model.users.users.UserRegistrationDTO;
import org.cyclos.model.users.users.UserRegistrationData;
import org.cyclos.model.users.users.UserRegistrationResult;
import org.cyclos.model.users.users.UserSearchContext;
import org.cyclos.model.users.users.UserSearchData;
import org.cyclos.services.users.UserService;
import org.cyclos.utils.CustomFieldHelper;

/**
 * Provides a sample on registering a user with all custom fields, addresses
 * and phones
 */
public class RegisterUser {

    public static void main(String[] args) {
        // Get the services
        UserService userService = Cyclos.getServiceFactory().getProxy(UserService.class);

        // The available groups for new users are obtained in the search data
        UserSearchData searchData = userService.getSearchData(UserSearchContext.MENU, null);
        List<BasicGroupVO> possibleGroups = searchData.getInitialGroups();

        // Find the consumers group
        GroupVO group = null;
        for (BasicGroupVO current : possibleGroups) {
            if (current instanceof GroupVO && current.getName().equals("Consumers")) {
                group = (GroupVO) current;
                break;
            }
        }
    }
}

```

```

    }
}

// Get data for a new user
UserDataParams params = new UserDataParams();
params.setGroup(group);
UserRegistrationData data = (UserRegistrationData) userService.getDataForNew(params);

// Basic fields
UserRegistrationDTO user = (UserRegistrationDTO) data.getDto();

user.setPasswords(new ArrayList<PasswordRegistrationDTO>());
List<PasswordRegistrationData> passwords = data.getPasswordsData();

for (PasswordRegistrationData passData : passwords) {
    PasswordRegistrationDTO passDTO = new PasswordRegistrationDTO();
    passDTO.setType(passData.getType());
    passDTO.setValue("1234");
    passDTO.setConfirmationValue("1234");
    passDTO.setAssign(true);
    passDTO.setForceChange(true);
    user.getPasswords().add(passDTO);
}
user.setGroup(group);
user.setName("John Smith");
user.setUsername("johnsmith");
user.setEmail("john.smith@mail.com");
user.setSkipActivationEmail(true);
// Custom fields
List<UserCustomFieldDetailedVO> customFields =
    CustomFieldHelper.getCustomFields(data.getProfileFieldActions());
CustomFieldDetailedVO gender = null;
CustomFieldDetailedVO idNumber = null;
for (CustomFieldDetailedVO customField : customFields) {
    if (customField.getInternalName().equals("gender")) {
        gender = customField;
    }
    if (customField.getInternalName().equals("idNumber")) {
        idNumber = customField;
    }
}
user.setCustomValues(new ArrayList<UserCustomFieldValueDTO>());

// Value for the gender custom field
UserCustomFieldValueDTO genderValue = new UserCustomFieldValueDTO();
genderValue.setField(gender);
for (CustomFieldPossibleValueVO possibleValue : gender.getPossibleValues()) {
    if (possibleValue.getValue().equals("Male")) {
        // Found the value for 'Male'
        genderValue.setEnumeratedValues(Collections.singleton(possibleValue));
        break;
    }
}
user.getCustomValues().add(genderValue);

// Value for id number custom field
UserCustomFieldValueDTO idNumberValue = new UserCustomFieldValueDTO();
idNumberValue.setField(idNumber);

```

```

        idNumberValue.setStringValue("123.456.789-10");
        user.getCustomValues().add(idNumberValue);

        // Address
        UserAddressDTO address = new UserAddressDTO();
        address.setName("Home");
        address.setAddressLine1("John's Street, 500");
        address.setCity("John's City");
        address.setRegion("John's Region");
        address.setCountry("BR"); // Country is given in 2-letter ISO code
        user.setAddresses(Arrays.asList(address));

        // Landline phone
        LandLinePhoneDTO landLinePhone = new LandLinePhoneDTO();
        landLinePhone.setName("Home");
        landLinePhone.setRawNumber("+551133333333");
        user.setLandLinePhones(Arrays.asList(landLinePhone));

        // Mobile phone
        MobilePhoneDTO mobilePhone = new MobilePhoneDTO();
        mobilePhone.setName("Mobile phone 1");
        mobilePhone.setRawNumber("+5511999999999");
        user.setMobilePhones(Arrays.asList(mobilePhone));

        // Effectively register the user
        UserRegistrationResult result = userService.register(user);
        RegistrationStatus status = result.getStatus();
        switch (status) {
            case ACTIVE:
                System.out.println("The user is now active");
                break;
            case INACTIVE:
                System.out.println("The user is in an inactive group, "
                    + "and needs activation by administrators");
                break;
            case EMAIL_VALIDATION:
                System.out
                    .println("The user needs to validate the e-mail "
                        + "address in order to confirm the registration");
                break;
        }
    }
}

```

Edit user profile

```

import java.util.List;

import org.cyclos.model.users.fields.UserCustomFieldValueDTO;
import org.cyclos.model.users.users.EditProfileData;
import org.cyclos.model.users.users.UserDTO;
import org.cyclos.model.users.users.UserLocatorVO;
import org.cyclos.model.users.users.UserVO;
import org.cyclos.services.users.UserService;

public class EditUser {

```

```

public static void main(String[] args) {
    // Get the services
    UserService userService = Cyclos.getServiceFactory().getProxy(UserService.class);

    // Locate the user by username, so we get the id
    UserLocatorVO locate = new UserLocatorVO(UserLocatorVO.USERNAME, "some-user");
    UserVO userVO = userService.locate(locate);

    // Get the profile data
    EditProfileData data = (EditProfileData) userService.getData(userVO.getId());
    UserDTO user = data.getDto();
    user.setName("Some modified name");
    List<UserCustomFieldValueDTO> customValues = user.getCustomValues();
    for (UserCustomFieldValueDTO fieldValue : customValues) {
        if (fieldValue.getField().getInternalName().equals("website")) {
            fieldValue.setStringValue("http://new.url.com");
        }
    }

    // Update the user
    userService.save(user);
    System.out.println("The user was updated.");
}
}

```

Login user

```

import java.util.List;

import org.cyclos.model.access.LoggedOutException;
import org.cyclos.model.access.channels.BuiltInChannel;
import org.cyclos.model.access.login.UserAuthVO;
import org.cyclos.model.banking.accounts.AccountWithStatusVO;
import org.cyclos.model.users.users.UserLocatorVO;
import org.cyclos.model.users.users.UserLoginDTO;
import org.cyclos.model.users.users.UserLoginResult;
import org.cyclos.server.utils.HttpServiceFactory;
import org.cyclos.server.utils.HttpServiceInvocationData;
import org.cyclos.services.access.LoginService;
import org.cyclos.services.banking.AccountService;

/**
 * Cyclos web service example: logs-in a user via web services.
 * This is useful when creating an alternative front-end for Cyclos.
 */
public class LoginUser {

    public static void main(String[] args) throws Exception {
        // This LoginService has the administrator credentials
        LoginService loginService = Cyclos.getServiceFactory().getProxy(LoginService.class);

        // Another option is to use an access client to connect with the
        // server (for the admin)
        // To make it works you must:
        // 1- create an access client
    }
}

```



```

// 2- assign it to the admin (to obtain the activation code)
// 3- activate it making a HTTP POST to the server using this url:
// ROOT_URL/activate-access-client containing only the activation code
// as the body
// 4- put the token returned from the servlet as the parameter of the
// HttpServiceInvocationData.accessClient(...) method
// 5- comment the first line (that using user and password and
// uncomment the following two sentences

// HttpServiceInvocationData adminSessionInvocationData =
// HttpServiceInvocationData
// .accessClient("put_the_token_here");
// LoginService loginService = Cyclos.getServiceFactory(
// adminSessionInvocationData).getProxy(LoginService.class);

String remoteAddress = "192.168.1.200";

// Set the login parameters
UserLoginDTO params = new UserLoginDTO();
params.setUser(new UserLocatorVO(UserLocatorVO.USERNAME, "some-user"));
params.setPassword("1234");
params.setRemoteAddress(remoteAddress);
params.setChannel(BuiltInChannel.MAIN.getInternalName());

// Login the user
UserLoginResult result = loginService.loginUser(params);
UserAuthVO userAuth = result.getUser();
String sessionToken = result.getSessionToken();
System.out.println("Logged-in " + userAuth.getUser().getDisplay()
    + " with session token = " + sessionToken);

// Do something as user. As the session token is only valid per ip
// address, we need to pass-in the client ip address again
HttpServiceInvocationData sessionInvocationData =
    HttpServiceInvocationData.stateful(sessionToken, remoteAddress);
// The services acquired by the following factory will carry on the
// user session data
HttpServiceFactory userFactory = Cyclos.getServiceFactory(sessionInvocationData);
AccountService accountService = userFactory.getProxy(AccountService.class);
List<AccountWithStatusVO> accounts =
    accountService.getAccountsSummary(userAuth.getUser(), null);
for (AccountWithStatusVO account : accounts) {
    System.out.println(account.getType()
        + ", balance: " + account.getStatus().getBalance());
}

// Logout. There are 2 possibilities:

// - Logout as administrator:
loginService.logoutUser(sessionToken);

// - OR logout as own user:
try {
    userFactory.getProxy(LoginService.class).logout();
} catch (LoggedOutException e) {
    // already logged out
}
}

```

```
}
```

Get account information

```
import java.math.BigDecimal;
import java.util.List;

import org.cyclos.model.banking.accounts.AccountHistoryEntryVO;
import org.cyclos.model.banking.accounts.AccountHistoryQuery;
import org.cyclos.model.banking.accounts.AccountStatusVO;
import org.cyclos.model.banking.accounts.AccountVO;
import org.cyclos.model.banking.accounts.AccountWithStatusVO;
import org.cyclos.model.banking.accounttypes.AccountTypeNature;
import org.cyclos.model.banking.accounttypes.AccountTypeVO;
import org.cyclos.model.users.users.UserLocatorVO;
import org.cyclos.model.users.users.UserVO;
import org.cyclos.services.banking.AccountService;
import org.cyclos.utils.Page;

/**
 * Provides a sample on getting the account information for a given user.
 */
public class GetAccountInformation {

    public static void main(String[] args) throws Exception {
        AccountService accountService =
            Cyclos.getServiceFactory().getProxy(AccountService.class);

        // Get the accounts summary
        UserLocatorVO user = new UserLocatorVO(UserLocatorVO.USERNAME, "some-user");
        List<AccountWithStatusVO> accounts = accountService.getAccountsSummary(user, null);

        // For each account, we'll show the balances
        for (AccountWithStatusVO account : accounts) {
            AccountStatusVO status = account.getStatus();
            if (status != null) {
                BigDecimal balance = status.getBalance();
                System.out.printf("%s has balance of %.2f %s\n",
                    account.getType().getName(),
                    balance,
                    account.getCurrency());
            }

            // Also, search for the last 5 payments on each account
            AccountHistoryQuery query = new AccountHistoryQuery();
            query.setAccount(new AccountVO(account.getId()));
            query.setPageSize(5);

            Page<AccountHistoryEntryVO> entries = accountService.searchAccountHistory(query);
            for (AccountHistoryEntryVO entry : entries) {
                AccountVO relatedAccount = entry.getRelatedAccount();
                AccountTypeVO relatedType = relatedAccount.getType();
                AccountTypeNature relatedNature = relatedType.getNature();
                // The from or to...
                String fromOrTo;
                if (relatedNature == AccountTypeNature.SYSTEM) {
```

```

        // ... might be the account type name if a system account
        fromOrTo = relatedType.getName();
    } else {
        // ... or just the user display
        UserVO relatedUser = (UserVO) relatedAccount.getOwner();
        fromOrTo = relatedUser.getDisplay();
    }
    // Display the amount, which can be negative or positive
    BigDecimal amount = entry.getAmount();
    boolean debit = amount.compareTo(BigDecimal.ZERO) < 0;

    System.out.printf("Date: %s\n", entry.getDate());
    System.out.printf("%s: %s\n", debit ? "To" : "From", fromOrTo);
    System.out.printf("Amount: %.2f\n", amount);
    System.out.println();
}
System.out.println("*****");
}
}
}

```

Perform payment

```

import java.math.BigDecimal;
import java.util.List;

import org.cyclos.model.EntityNotFoundException;
import org.cyclos.model.banking.InsufficientBalanceException;
import org.cyclos.model.banking.MaxAmountPerDayExceededException;
import org.cyclos.model.banking.MaxAmountPerMonthExceededException;
import org.cyclos.model.banking.MaxAmountPerWeekExceededException;
import org.cyclos.model.banking.MaxAmountPerYearExceededException;
import org.cyclos.model.banking.MaxPaymentAmountExceededException;
import org.cyclos.model.banking.MaxPaymentsPerDayExceededException;
import org.cyclos.model.banking.MaxPaymentsPerMonthExceededException;
import org.cyclos.model.banking.MaxPaymentsPerWeekExceededException;
import org.cyclos.model.banking.MinTimeBetweenPaymentsException;
import org.cyclos.model.banking.accounts.InternalAccountOwner;
import org.cyclos.model.banking.accounts.SystemAccountOwner;
import org.cyclos.model.banking.transactions.PaymentVO;
import org.cyclos.model.banking.transactions.PerformPaymentDTO;
import org.cyclos.model.banking.transactions.PerformPaymentData;
import org.cyclos.model.banking.transactions.TransactionAuthorizationStatus;
import org.cyclos.model.banking.transfertypes.TransferTypeWithCurrencyVO;
import org.cyclos.model.users.users.UserLocatorVO;
import org.cyclos.server.utils.HttpServiceFactory;
import org.cyclos.services.banking.PaymentService;
import org.cyclos.services.banking.TransactionService;
import org.cyclos.utils.CollectionHelper;

/**
 * Provides a sample on performing a payment between a user and a system
 * account
 */
public class PerformPayment {

```

```

public static void main(String[] args) {
    // Get the services
    HttpServiceFactory factory = Cyclos.getServiceFactory();
    TransactionService transactionService = factory.getProxy(TransactionService.class);
    PaymentService paymentService = factory.getProxy(PaymentService.class);

    // The payer and payee
    InternalAccountOwner payer = new UserLocatorVO(UserLocatorVO.USERNAME, "some-user");
    InternalAccountOwner payee = SystemAccountOwner.instance();

    // Get data regarding the payment
    PerformPaymentData data;
    try {
        data = transactionService.getPaymentData(payer, payee, null);
    } catch (EntityNotFoundException e) {
        System.out.println("Some of the users were not found");
        return;
    }

    // Get the first available payment type
    List<TransferTypeWithCurrencyVO> types = data.getPaymentTypes();
    TransferTypeWithCurrencyVO paymentType = CollectionHelper.first(types);
    if (paymentType == null) {
        System.out.println("There is no possible payment type");
    }

    // The payment amount
    BigDecimal amount = new BigDecimal(10.5);

    // Perform the payment itself
    PerformPaymentDTO payment = new PerformPaymentDTO();
    payment.setType(paymentType);
    payment.setOwner(data.getFrom());
    payment.setSubject(data.getTo());
    payment.setAmount(amount);

    try {
        PaymentVO result = paymentService.perform(payment);
        // Check whether the payment is pending authorization
        TransactionAuthorizationStatus auth = result.getAuthorizationStatus();
        if (auth == TransactionAuthorizationStatus.PENDING_AUTHORIZATION) {
            System.out.println("The payment is pending authorization");
        } else {
            System.out.println("The payment has been processed");
        }
    } catch (InsufficientBalanceException e) {
        System.out.println("Insufficient balance");
    } catch (MaxPaymentsPerDayExceededException e) {
        System.out.println("Maximum daily amount of transfers "
            + e.getMaxPayments() + " has been reached");
    } catch (MaxPaymentsPerWeekExceededException e) {
        System.out.println("Maximum weekly amount of transfers "
            + e.getMaxPayments() + " has been reached");
    } catch (MaxPaymentsPerMonthExceededException e) {
        System.out.println("Maximum monthly amount of transfers "
            + e.getMaxPayments() + " has been reached");
    } catch (MinTimeBetweenPaymentsException e) {
        System.out.println("A minimum period of time should be awaited to make "

```

```

        + "a payment of this type");
    } catch (MaxAmountPerDayExceededException e) {
        System.out.println("Maximum daily amount of "
            + e.getMaxAmount() + " has been reached");
    } catch (MaxAmountPerWeekExceededException e) {
        System.out.println("Maximum weekly amount of "
            + e.getMaxAmount() + " has been reached");
    } catch (MaxAmountPerMonthExceededException e) {
        System.out.println("Maximum monthly amount of "
            + e.getMaxAmount() + " has been reached");
    } catch (MaxAmountPerYearExceededException e) {
        System.out.println("Maximum yearly amount of "
            + e.getMaxAmount() + " has been reached");
    } catch (MaxPaymentAmountExceededException e) {
        System.out.println("Maximum payment amount of "
            + e.getMaxAmount() + " has been reached");
    } catch (Exception e) {
        System.out.println("The payment couldn't be performed");
    }
}
}
}

```

3.3. PHP clients

The recommended way to integrate Cyclos is described in the section called “REST API”. In case that API does not meet your requirements (e.g. some missing operation) a PHP library is provided. The library uses web-rpc calls with JSON objects internally, handling requests and responses, as well as mapping exceptions. As such, the same rules described in the section called “Details on JSON handling” are applied. A PHP class is generated for each Cyclos service interface, and all methods are generated on them. The parameters and result types, however, are not generated, and are either handled as strings, numbers, booleans or generic objects (stdClass).

You can download the PHP client for the corresponding Cyclos version [here](#).

Dependencies

- PHP 5.3 or newer
- PHP CURL extension (package php5-curl in Debian / Ubuntu)
- PHP JSON extension (package php5-json in Debian / Ubuntu)

Using services from a 3rd party PHP application

In order to use the Cyclos classes, we first register an autoload function to load the required classes automatically, like this:

```

function load($c) {
    if (strpos($c, "Cyclos\\") >= 0) {

```

```

        include str_replace("\\", "/", $c) . ".php";
    }
}
spl_autoload_register("load");

```

Then, Cyclos is configured with the server root URL and authentication details:

```

Cyclos\Configuration::setRootUrl("http://192.168.1.27:8888/england");
Cyclos\Configuration::setAuthentication("admin", "1234");
// OR Cyclos\Configuration::setSessionToken("sessionToken");
// OR Cyclos\Configuration::setAccessClientToken("accessClientToken");

```

To specify a channel other than Web Services, call `Cyclos\Configuration::setChannel("channel");`

Afterwards, services can be instantiated using the new operator, and the corresponding methods will be available:

```

$userService = new Cyclos\UserService();
$page = $userService->search(new stdClass());

```

Examples

Configuration

All the following examples include the `configureCyclos.php` file, which contains the following:

```

<?php

function load($c) {
    if (strpos($c, "Cyclos\\") >= 0) {
        include str_replace("\\", "/", $c) . ".php";
    }
}

spl_autoload_register('load');

Cyclos\Configuration::setRootUrl("http://localhost:8888/england");
Cyclos\Configuration::setAuthentication("admin", "1234");

?>

```

Search users

```

<?php

require_once 'configureCyclos.php';

$userService = new Cyclos\UserService();
$query = new stdClass();
$query->keywords = 'Consumer*';
$query->pageSize = 5;
$query->ignoreProfileFieldsInList = true;

```

```

$page = $userService->search($query);

echo("Found a total of $page->totalCount users\n");

if (!empty($page->pageItems)) {
    foreach ($page->pageItems as $user) {
        echo("** $user->display ($user->shortDisplay)\n");
    }
}

?>

```

Search advertisements

```

<?php

require_once 'configureCyclos.php';

$adService = new Cyclos\AdService();
$query = new stdClass();
$query->keywords = 'Computer*';
$query->pageSize = 10;
$query->orderBy = 'PRICE_LOWEST';
$page = $adService->search($query);

echo("Found a total of $page->totalCount advertisements\n");

if (!empty($page->pageItems)) {
    foreach ($page->pageItems as $ad) {
        echo("** $ad->name\n");
    }
}

?>

```

Login user

```

<?php

// Configure Cyclos and obtain an instance of LoginService
require_once 'configureCyclos.php';
$loginService = new Cyclos\LoginService();

// Set the parameters
$params = new stdClass();
$params->user = array("principal" => $_POST['username']);
$params->password = $_POST['password'];
$params->remoteAddress = $_SERVER['REMOTE_ADDR'];

// Perform the login
try {
    $result = $loginService->loginUser($params);
} catch (Cyclos\ConnectionException $e) {
    echo("Cyclos server couldn't be contacted");
    die();
} catch (Cyclos\ServiceException $e) {

```

```

switch ($e->errorCode) {
    case 'VALIDATION':
        echo("Missing username / password");
        break;
    case 'LOGIN':
        echo("Invalid username / password");
        break;
    case 'REMOTE_ADDRESS_BLOCKED':
        echo("Your access is blocked by exceeding invalid login attempts");
        break;
    default:
        echo("Error while performing login: {$e->errorCode}");
        break;
}
die();
}

// Redirect the user to Cyclos with the returned session token
header("Location: "
    . Cyclos\Configuration::getRootUrl()
    . "?sessionToken="
    . $result->sessionToken);

?>

```

Perform payment from system to user

```

<?php
require_once 'configureCyclos.php';

$transactionService = new Cyclos\TransactionService();
$paymentService = new Cyclos\PaymentService();

try {
    $data = $transactionService->getPaymentData('SYSTEM', array('username' => 'c1'), null);

    $parameters = new stdClass();
    $parameters->from = $data->from;
    $parameters->to = $data->to;
    $parameters->type = $data->paymentTypes[0];
    $parameters->amount = 5;
    $parameters->description = "Test from system to user";

    $paymentResult = $paymentService->perform($parameters);
    if (isset($paymentResult->authorizationStatus) && $paymentResult->authorizationStatus
    == 'PENDING_AUTHORIZATION') {
        echo("Not yet authorized\n");
    } else {
        echo("Payment done with id $paymentResult->id\n");
    }
} catch (Cyclos\ServiceException $e) {
    echo("Error while calling $e->service.$e->operation: $e->errorCode");
}

?>

```


Perform payment from user to user

```
<?php
require_once 'configureCyclos.php';

//Perform the payment from user c1 to c2
Cyclos\Configuration::setAuthentication("c1", "1234");

$transactionService = new Cyclos\TransactionService();
$paymentService = new Cyclos\PaymentService();

try {
    $data = $transactionService->getPaymentData(
        array('username' => 'c1'),
        array('username' => 'c2'),
        null);

    $parameters = new stdClass();
    $parameters->from = $data->from;
    $parameters->to = $data->to;
    $parameters->type = $data->paymentTypes[0];
    $parameters->amount = 5;
    $parameters->description = "Test payment to user";

    $paymentResult = $paymentService->perform($parameters);
    if (isset($paymentResult->authorizationStatus) && $paymentResult->authorizationStatus
    == 'PENDING_AUTHORIZATION') {
        echo("Not yet authorized\n");
    } else {
        echo("Payment done with id $paymentResult->id\n");
    }
} catch (Cyclos\ServiceException $e) {
    switch ($e->errorCode) {
        case "VALIDATION":
            echo("Some of the parameters are invalid\n");
            var_dump($e->error);
            break;
        case "INSUFFICIENT_BALANCE":
            echo("Insufficient balance to perform the payment\n");
            break;
        case "MAX_AMOUNT_PER_DAY_EXCEEDED":
            echo("Maximum amount exeeded today\n");
            break;
        default:
            echo("Error with code $e->errorCode while performing the payment\n");
            break;
    }
}

?>
```

Error handling

All errors thrown by the server are translated into PHP by throwing Cyclos\ServiceException. This class has the following properties:

- **service:** The service path which generated the error. For example, `paymentService`, `accountService` and so on.
- **operation:** The name of the operation which generated the error. Is the same name as the method invoked on the service.
- **errorCode:** Is the simple Java exception class name, uppercased, with the word 'Exception' removed. Check the API (as described above) to see which exceptions can be thrown by each service method. Keep in mind that many times the declared exception is a superclass, of many possible concrete exceptions. All methods declare to throw `FrameworkException`, but it is abstract, and is implemented by several concrete exception types, like `PermissionException`. In this example, the `errorCode` will be `PERMISSION`. Another example is the `InsufficientBalanceException` class, which has as `errorCode` the string `INSUFFICIENT_BALANCE`.
- **error:** Contains details about the error. Only some specific exceptions have this field. For example, if the `errorCode` is `VALIDATION`, and the exception variable name `$e`, `$e->error->validation` will provide information on errors by property, custom field or general errors.

3.4. Other clients

For other clients, a "REST level 0", or RPC-like interface is available, using JSON encoded strings for passing parameters and receiving results from services. Each service responds to POST requests to the following URL `http[s]://cyclos.url/[network/]web-rpc/<short-service-name>`, where the short-service-name is the service with the first letter as lowercase. So, for example, `https://my.cyclos.instance.com/network/web-rpc/accountService` is a valid URL, being mapped to [AccountService](#). Other URLs are also supported, as described in the section called "URL mapping".

For authentication, the username and password should be passed as a HTTP header using the standard basic authentication – a header like: "Authentication: Basic <Base64-encoded form of username:password>". Actually, username or other principal type (user identification method) will be chosen according to the configuration. If the configuration allows more than one principal type, it is possible to specify a value in the "Principal-Type" header, which must match the principal type internal name. Alternatively, it is possible to login the user via [LoginService](#) and pass the obtained session token in the "Session-Token" header. A third access option is to use an access client token. In this case, the header "Authorization: Bearer <access client token>" is used to specify the access client token. Alternatively, the header `Access-Client-Token` can be passed in with the token as value.

To specify a channel, pass the header "Channel: <channel internal name>". If no channel is passed, Web Services is assumed.

When the URL is specified up to the service, as stated above, the request body must be a JSON object with the 'operation' and 'params' properties, where operation is the method name, and

params is either an array with parameters, or optionally the parameter if the method has a single parameter (without the array) or even omitted if the method have no parameters. For objects, the parameters are expected to be the same as the Java counterparts (see the [JavaDocs](#) for a reference on the available properties for each object).

As result, if the request was successful (http status code is 200), an object with a single property called result will be returned. The object has the same structure as the object returned by the service method, or is a string, boolean or number for simple types. Requests which resulted in error (status code distinct than 200) will have the following structure:

- **errorCode:** A string generated from the exception java class name. The unqualified class name has the Exception suffix removed, and is transformed to all uppercase, separated by underlines. So, for example, for [org.cyclos.model.ValidationException](#), the error code is VALIDATION; for [org.cyclos.model.banking.InsufficientBalanceException](#), the error code is INSUFFICIENT_BALANCE, and so on.
- Any other properties (public getters) the thrown exception has will also be mapped as a property here, for example, [org.cyclos.model.ValidationException](#) holds a property called validation which contains an object representing a [org.cyclos.utils.ValidationResult](#).

URL mapping

Besides using the URL pointing to the service, and have the POST body as a JSON, selecting the operation and the parameters, it is also possible to choose the operation in the URL itself, as a subpath in the URL. For example, <https://my.cyclos.instance.com/network/web-rpc/userService/search> already maps to the search operation. The POST body, then, is expected to be just the JSON for the parameters, with the same rules as explained above: if is a single parameter, the body can be the JSON value directly, and if no parameters, the POST body can be empty.

Additionally, the service methods that are readonly can be invoked by GET requests. In this case, the parameter can be passed using 2 forms:

- When the parameters are simple (just identifiers or internal names), they can be passed in as URL parts. For example, <https://my.cyclos.instance.com/network/web-rpc/accountService/load/836144284089>
- When there is a single parameter of type object, it can be passed using URL parameters. For example: <https://my.cyclos.instance.com/network/web-rpc/userService/search?keywords=shop&groups=business>

Finally, services are mapped to other 2 URLs besides <name>Service: one without the 'Service' suffix, and another one, pluralized. Also, if an operation doesn't match, it will be attempted by prepending 'get' with the first letter capitalized. This will allow shorter urls on calls, like:

- GET <https://my.cyclos.instance.com/network/web-rpc/users/search?keywords=shop&groups=business> is equivalent to GET <https://my.cyclos.instance.com/network/web-rpc/userService/search?keywords=shop&groups=business>
- GET <https://my.cyclos.instance.com/network/web-rpc/user/data/4534657457> is equivalent to GET <https://my.cyclos.instance.com/network/web-rpc/userService/getData/4534657457>

Details on JSON handling

All output objects, when converted to JSON, will have a property called `class`, which represents the fully-qualified Java class name of the source object. Most clients can just ignore the result. However, when sending requests to classes that expect a polymorphic object, the server needs to know which subclass the passed object represents. In those cases, passing the `class` property, with the fully qualified Java class name is required. An example is the [AdService](#). When saving an advertisement, it could either be a simple advertisement ([AdvertisementDTO](#)) or a webshop advertisement ([AdWebShopDTO](#)). In this case, a `class` property with the fully qualified class name is required. Note, however, that in most cases, the class information is not needed.

Whenever a subclass of [EntityVO](#) is needed, numbers or strings are also accepted (besides objects). Numbers always represent the vo identifier (id property). Strings can either be id when they are numeric, or can represent one of the following cases:

- When the type is [BasicUserVO](#) or a subclass, an [UserLocatorVO](#) is created, and the string represents the principal. If the string is 'self' (sans quotes) it will resolve to the logged user;
- When the type is [AccountVO](#), the string represents the account number;
- When the destination VO has an internal name, the string represents it;
- Otherwise, the VO is assumed to be null.

If the value is supposed to be a number handled as user principal (for example, a mobile phone) or account number, it must be prefixed with a single quote. For example, to represent a phone number as string, the following is accepted: '5187653456. If not prefixed, it would be interpreted as user id instead. The single quote prefix is the same as Excel / LibreOffice use to represent a number as string.

Other points to note with JSON handling:

- Whenever a collection is expected, a single value can be passed, resulting in a collection with a single element;
- Java long values (mostly identifiers) are always returned as string, because of the identifier ciphering, the whole 64-bit space is used. In JavaScript, however, integer numbers cannot use 64 bit, resulting in different numbers when reading from JSON.
- Whenever dates are used (represented by the [DateTime](#) class) they are returned / expected to be strings in the ISO 8601 format, without timezone. For example, "2015-01-31T17:29:00"

represent 31 January 2015, at 5:29 pm. Also, for input, the text "now" is accepted (without quotes) to represent the current time.

Examples

Assuming that the authentication header is correctly passed, the following request can be performed to search for users: POST <https://my.cyclos.instance.com/network/web-rpc/userService> with the following body:

```
{
  "operation": "search",
  "params": {
    "keywords": "user",
    "groups": "consumers",
    "pageSize": 5
  }
}
```

The resulting JSON will be something like:

```
{
  "result": {
    "currentPage": "0",
    "pageSize": "5",
    "totalCount": "2",
    "pageItems": [
      {
        "class": "org.cyclos.model.users.users.UserVO",
        "id": "-2717327251475675143",
        "display": "Consumer 1",
        "shortDisplay": "c1"
      },
      {
        "class": "org.cyclos.model.users.users.UserVO",
        "id": "-2717467988964030471",
        "display": "Consumer 3",
        "shortDisplay": "c3"
      }
    ]
  }
}
```

Note the params "groups" property of the input query is a collection of [BasicGroupVO](#). It is being passed the string "consumers", which is matched to the group internal name.

The above request is equivalent to a POST to <https://my.cyclos.instance.com/network/web-rpc/users/search> (using the plural name) with the following body:

```
{
  "keywords": "user",
  "groups": "consumers",
  "pageSize": 5
}
```

```
}
```

Note only the parameters part is passed. If the service method would require multiple parameters, the body should be a JSON array. If a single string, the string should be quoted, just like in JSON.

Also, the above request is equivalent to a GET to <https://my.cyclos.instance.com/network/web-rpc/user/search?keywords=user&groups=consumers&pageSize=5> (singular name). Only methods which take a single parameter object can use query parameters.

3.5. Server side configuration to enable web services

For clients to invoke web services in Cyclos, the following configuration needs to be done on the server (as global or network administrator):

- On the System management > Configurations tab, click a row to go to the configuration details page.
- On the Channels tab, click on the Web services channel row, to go to the channel configuration details page. If using access clients, the channel will be Access client instead.
- Make sure the channel is enabled. Click the edit icon on the right if the channel is not defined on this configuration. Then mark the channel as enabled, choose the way users will be able to access this channel (by default or manually) and the password type used to access the web services channel. You can also set a confirmation password, so sensitive operations, like performing a payment, will require that additional password.
- For the user which will be used for web services, on the view user profile page, under the User management box, click the channels access link.
- On that page, make sure the Web services channel is enabled for that user. Also, only active users may access any channel - on the profile page, on the same User management box, there should be a link with actions like Enable / Block / Disable / Remove. On that page, make sure the user status is Active.
- A side note: If performing payments via Web services, make sure the desired Transfer type is enabled for the Web services channel. To check that, go to System management > Accounts configuration > Account types. Then click the row of the desired account type, select the Transfer types tab and click on the desired payment type (generated types cannot be used for direct payment). There, make sure the Channels field has the Web services channel.

3.6. Available services and API Changes

The available services are documented in the [JavaDocs](#), under each `org.cyclos.services` subpackage.

For the full set of API changes, please, refer to the [online documentation](#).

4. Scripting

4.1. Scripting engine

The Cyclos scripting module (available from version 4.2 onwards) provides an integration layer that allows connecting from Cyclos to third party software, as well executing custom operations and scheduled tasks within Cyclos self. The scripting module offers an easy way to customize and extend Cyclos, without losing compatibility with future Cyclos versions. The scripting engine can access the full Cyclos services layer which makes it a powerful feature. For security reasons only global administrators can add scripts. Network administrators can be given permissions to bound the scripts to elements such as extension points (eg. payment, user profile, advertisement), custom validations (for input fields), custom calculations (account fees, transaction fees), custom operations and scheduled tasks. Any internal entity in Cyclos (e.g. user, address, payment, authorization, reference etc.) can be accessed by the scripts. When developing custom operations it is likely that you want to store and use new values/entities. It is possible to create specific record types and custom fields and make them available to the scripts. The record types can be of the type 'system' or 'user' depending on the requirements.

On this page you will find links with documentation about the available extensions and examples. In the future we will add a repository of useful scripts. If you wrote a script that could serve other projects we will be happy to add it. Please post it on our [Forum](#) or send it to info@cyclos.org.

Global admins can write and store scripts directly within Cyclos. Each script 'type' has its own functions which have to be implemented. A network admin can chose from the available scripts and bind them to Cyclos operations and events, or to new operations. The variables used in the scripts can be managed outside the scripts in the extensions self (by the network admin). This avoids the need for a global admin having to modify a script every time a new or different input value is required. It is also possible to define additional information and confirmation texts that can be displayed to the user when a custom operation is initiated or submitted.

The scripting language currently supported is [Groovy](#). It offers a powerful scripting language that is very similar to Java, with a close to zero learning curve for Java developers. It is possible to write scripts that will be available in a shared script library, so that other scripts within the same context can make use of it. All scripts are compiled to Java bytecode which makes them highly performatic. Currently Cyclos requires Java 8 or above.

Debugging scripts can sometimes be tricky, because the exact context is only available at runtime, and errors can be hidden. A good approach is to set `cyclos.dumpAllErrors` to true in `cyclos.properties`. This way whenever an error is triggered, it is dumped to the application server (i.e., Tomcat) console.

Regarding database transactions, normally scripts run inside a database, and returning without errors means the transaction is committed, while throwing an exception means the transaction is rolled-back. So, be aware that silencing database error in the script (catching them without throwing another exception) may cause a transaction not to be rolled back, and if multiple database operations were performed, the final state can be inconsistent. For example, when performing a payment, a transaction (representing the payment) is created. Then one or more transfers are created (transferring of funds between accounts - there can be multiple if there are fees). Before each transfer the account balance is checked, to make sure it has enough funds. In this case, if some account has no balance and the exception is silenced, the database will have a processed transaction without a corresponding transfer, which is an inconsistent state for Cyclos.

Variables bound to all scripts

When running, scripts have a set of bindings, that is, available top-level variables. At runtime, the bindings will vary according to the script type and context. For example, each extension point type has one or more specific bindings. On all cases, however, the following variables are bound:

- `scriptParameters`: In the script details page, or in every every page where a script is chosen to be used (for example, in the extension point or custom operation details page) there will be a textarea where parameters may be added to the script. They allow scripts to be reused in different contexts, just with different parameters. The text is parsed as [Java Properties](#), and the format is [described here](#). The library parameters are included first (if any), then the own script parameters (if any), then the specific page parameters. This allows overriding parameters at more specific levels.
- `scriptHelper`: An instance of [org.cyclos.impl.system.ScriptHelper](#). Besides having the instance, all methods are automatically exported as closures on the default binding, making it possible to call its methods without using the 'scriptHelper.' prefix. The ScriptHelper contains some useful methods, like:
 - `wrap(object[, customFields])`: wraps the given object in a Map, with some custom characteristics:
 - If the wrapped object contains custom fields, it will allow getting / setting custom field values using the internal name
 - Values will be automatically converted to the expected destination type
 - If a list of custom fields are passed, then they are considered. If not, will attempt to read the current fields for the object, which might not always be available (for example, when creating a new record) or even no longer active (for example, when the product of a user just removed a field, and the value is still there)
 - Example:


```

def bean = scriptHelper.wrap(user)
def gender = bean.gender
// gender will be a org.cyclos.entities.system.CustomFieldPossibleValue
// if gender is an enumerated field
def date = bean.customDate
// date will be a java.util.Date if customDate is a date field
def relatedUser = bean.relatedUser
// relatedUser will be an org.cyclos.entities.users.User
// if relatedUser is linked entity field of type user

```

- `bean(class)`: returns a bean by type. The class reference needs to be passed.
- `addOnCommit(runnable)`, `addOnRollback(runnable)`: Adds callbacks to be executed after the main database transaction ends, either successfully or with failure. Be aware that those callbacks will be invoked outside any transaction scope within Cyclos, so things like 'sessionData.loggedUser' won't work (because it requires retrieving the User object from the database). However, it is more efficient, as no new database access needs to be done. This is mostly useful to notify an external application that some data has been persisted in Cyclos (after we're 100% sure that the data is persistent). Keep in mind that there is a (very) small chance that the main transaction is committed / rolled back but then the server crashes, and the callback weren't yet called. So, when synchronizing with external systems, it is always wise to do some form of timeout / recovery mechanism.
- `addOnCommitTransactional(runnable)`, `addOnRollbackTransactional(runnable)`: Same as the non-transactional counterparts, but they are executed inside a new transaction in Cyclos.
- `maskId(id)`, `unmaskId(id)`: In Cyclos the internal database id's are not visible to the clients, because of security reasons. The id's used in the web application or used in our webservice are therefore always masked/obfuscated. These methods apply or remove the mask to the id.
- `sessionData`: The currently bound [org.cyclos.impl.access.ScriptSessionData](#).
- `entityManager`: The JPA entity manager bound to the current transaction.
- `formatter`: A [org.cyclos.impl.utils.formatting.FormatterImpl](#).
- `objectMapper`: Jackson's [com.fasterxml.jackson.databind.ObjectMapper](#) configured with all JSON rules used by Cyclos.
- `jdbcTemplate`: Spring's [org.springframework.jdbc.core.JdbcTemplate](#) for performing native queries.
- `namedParameterJdbcTemplate`: [org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate](#) for performing native queries. Spring's for

- Service implementations: All *ServiceLocal objects are bound via simple names, starting with lowercase characters, without the 'Local' suffix. For example, [org.cyclos.impl.users.UserServiceLocal](#) is bound as userService.
- Security layer: All *ServiceSecurity objects are bound via simple names, starting with lowercase characters. For example, [org.cyclos.security.users.UserServiceSecurity](#) is bound as userServiceSecurity.
- Internal handlers: All *Handler objects are bound via simple names, starting with lowercase characters. For example, [org.cyclos.impl.access.ConfigurationHandler](#) is bound as configurationHandler.

Script storage

Starting with Cyclos 4.6, a general-purpose storage is available for scripts. It is a key/value storage, implementing the [ObjectParameterStorage](#) interface. It stores the values as JSON in the database. Besides the methods for get/set String, Boolean, Decimal, Integer, Long and Enum, it also supports storing objects. Also, a mechanism is provided for Groovy scripts to access objects directly via the property name, such as `storage.value = value` or `value = storage.value`.

A script storage is obtained using a key (string), and a timeout can (optionally) be set before the storage expires. The storage is accessed via the [ScriptStorageHandler](#). It provides the following methods:

- `get(key) / get(key, timeoutInSeconds)`: Returns a storage by key (string). If a valid storage exists (in the same network), it is returned. Otherwise, a new one is created and returned. Optionally a timeout in seconds can be passed, which sets an expiration for the stored data.
- `exists(key)`: Returns whether a valid storage with a given key exists.
- `remove(key)`: Removes an storage by key.

Some restrictions apply on which kind of objects can be stored or retrieved. Entities can only be stored if they are already persisted (only the id is stored, and the entity is loaded by id from the database when retrieved). Other objects need to have a public empty constructor, plus getters and setters for fields.

Example:

```
// Storage retrieval
def timeout = 60 * 60 * 3 // Expires in 3 hours
def key = "requests_for_${sessionData.loggedUser.id}"
def storage = scriptStorageHandler.get(key, timeout)

// First, store the number of requests for the logged user
storage.requests = (storage.requests ?: 0) + 1

// Then, later on, maybe on another script...
```

```
return "There are ${storage.requests} requests for user ${sessionData.loggedUser.name}"
```

4.2. Debugging scripts

The script editor in Cyclos uses [CodeMirror](#), which provides syntax highlighting. However, as the script complexity grows, better tooling support is needed. We use and support using [Eclipse](#), together with the [Groovy plugin](#).

For this purpose, Cyclos provides, in the details page of scripts, a button names "Get code for debug". It will download a ZIP file with the full code of each script box, with all library code already included (there are comments separating each include). Also the script parameters are returned, including script parameters of included libraries as well. This is the code that is actually executed by Cyclos.

In order to run the scripts in the IDE, a local copy of the Cyclos database is needed, as well as a copy of the JARs bundled with Cyclos. Also, a small Groovy script needs to be written to provide the actual script the context (bound variables). The ZIP file also includes a README.html file, which will explain how to setup the environment and run the script.

4.3. Script types

Library

Libraries are scripts which are included by other scripts, in order to reuse code, and are never used directly by other functionality in Cyclos.

Each script (including other libraries) can have any number of libraries as dependencies. However circular dependencies between libraries (for example, A depends on B, which depends on C, which depends on A) are forbidden (validated when saving a library).

The order in which the code on libraries is included in the final code respects the dependencies, but doesn't guarantee ordering between libraries in the same level. For example, if there are both C and B libraries which depend on A, it is guaranteed that A is included before B and C, but either B or C could be included right after A. So, in the example, your code shouldn't rely that B comes before C. In this case, the library C should depend on B to force the A, B, C order.

Contrary to other script types, libraries don't have bound variables per se: the bindings will be the same as the script including the library.

Also, as libraries are just included in other scripts, no direct examples are provided here. The provided example [scripting solutions](#), however, use libraries.

Custom field validation

These scripts are used to validate a custom field value. The field can be of any type (users, advertisements, user records, transactions and so on). The script code has the following variables bound (besides the [default bindings](#))

- object: The DTO which holds the custom field values. May be an instance of:
 - [org.cyclos.model.users.users.UserDTO](#)
 - [org.cyclos.model.users.contacts.ContactDTO](#)
 - [org.cyclos.model.users.contactinfos.ContactInfoDTO](#)
 - [org.cyclos.model.marketplace.advertisements.BasicAdDTO](#)
 - [org.cyclos.model.users.records.UserRecordDTO](#)
 - [org.cyclos.model.banking.transactions.PerformTransactionDTO](#)
 - [org.cyclos.model.contentmanagement.documents.ProcessDynamicDocumentDTO](#)
 - [org.cyclos.model.system.operations.RunCustomOperationDTO](#)
- field: The [org.cyclos.entities.system.CustomField](#).
- value: The actual custom field value. Depends on the custom field type. May be one of:
 - String (for single line text, multi line text, rich text or url types)
 - Boolean (for boolean type)
 - Integer (for integer type)
 - BigDecimal (for decimal type)
 - [org.cyclos.entities.system.CustomFieldPossibleValue](#) (for single selection type)
 - A collection of [org.cyclos.entities.system.CustomFieldPossibleValue](#) (for multiple selection type)
 - [org.cyclos.model.system.fields.DynamicFieldValueVO](#) (for dynamic selection type)
 - [org.cyclos.entities.users.User](#) (for linked entity of type user)
 - [org.cyclos.entities.banking.Transaction](#) (for linked entity of type transaction)
 - [org.cyclos.entities.banking.Transfer](#) (for linked entity of type transfer)
 - [org.cyclos.entities.users.Record](#) (for linked entity of type record)
 - [org.cyclos.entities.marketplace.BasicAd](#) (for linked entity of type advertisement)
 - [org.cyclos.entities.utils.RawFile](#) (for file type)
 - [org.cyclos.entities.system.Image](#) (for image type)

The script should return one of the following:

- A boolean, indicates that the value is either valid / invalid. When invalid, the general "<Field name> is invalid" error will be displayed;
- A string, means the field is invalid, and the string is the error message. To concatenate the field name directly, use the {0} placeholder, like: "{0} has an unexpected value";
- Any other result will be considered valid.

Examples

E-mail

To have a custom field which is validated as an e-mail, use the following script:

```
import org.apache.commons.validator.routines.EmailValidator

return EmailValidator.getInstance().isValid(value)
```

IBAN account number

To validate an IBAN account number as a custom field, the following script can be used:

```
import org.apache.commons.validator.routines.checkdigit.IBANCheckDigit

return IBANCheckDigit.IBAN_CHECK_DIGIT.isValid(value.replaceAll("\s", ""))
```

CPF Validation

In Brazil, people are identified by a number called CPF (Cadastro de Pessoas Físicas). It has 2 verifying digits, which have a known formula to calculate. Here's the example for validating it in Cyclos:

```
import static java.lang.Integer.parseInt

def boolean validateCPF(String cpf) {
    // Strip non-numeric chars
    cpf = cpf.replaceAll("[^0-9]", "")

    // Obvious checks: needs to be 11 digits, and not all be the same digit
    if (cpf.length() != 11 || cpf.toSet().size() == 1) {
        return false
    }

    int add = 0
    // Check for verifier digit 1
    for (int i = 0; i < 9; i++) add += parseInt(cpf[i]) * (10 - i)
    int rev = 11 - (add % 11)
    if (rev == 10 || rev == 11) rev = 0
    if (rev != parseInt(cpf[9])) return false

    add = 0;
    // Check for verifier digit 2
    for (int i = 0; i < 10; i++) add += parseInt(cpf[i]) * (11 - i)
```

```

    rev = 11 - (add % 11)
    if (rev == 10 || rev == 11) rev = 0
    if (rev != parseInt(cpf[10])) return false

    return true
}

return validateCPF(value)

```

Load custom field values

These scripts are used to load a list of allowed values for a custom field. Custom fields of type dynamic selection are required to have such script. Several other field types can have an optional load values script: string, integer, decimal, date, url, enumerated or linked entity. Enumerated fields naturally have a list of static possible values. The script, however, can be used to show a subset of those options to specific users. Multi-line text, rich text, boolean, image and file types cannot have a load custom field values script.

If a custom field of type string, integer, decimal, date, url or linked entity has a load values script, Cyclos will use a single selection or radio button group widget instead of the regular widget for the custom field. Also, when a load custom field values script is used, the server-side validation will ensure that saved values are valid according to the allowed values list.

The script has a separated code block which loads values for custom fields being used as search filter. The field types supporting load values when filtering are: dynamic selection, linked entity and enumerated. In that case, the bound variables will be different than the ones for the code block that runs over fields used to create or edit some entity (user, advertisement, record, etc).

In all cases, the script will have the following variables bound (besides the [default bindings](#)):

- field: The [org.cyclos.entities.system.CustomField](#)

Also, depending on the custom field nature, there are the following additional bindings, both for the script that runs when creating or modifying an entity and for the script that runs over custom fields used as search filters:

- User (profile) fields:

When the field is used for registering a user or editing a user profile:

- user: The [org.cyclos.entities.users.User](#). Even when registering a user, will always have the 'group' property set with the [org.cyclos.entities.users.Group](#) instance.

When the field is used as search filter or in the built in bulk action "Change custom field value":

- searchContext: The [org.cyclos.impl.users.ProfileFieldSearchContext](#) in which the custom field is being used for search. Note that only the values reflecting filter are used. This enumeration also contains cases for the field in keywords, but it will never be the case when calling this script. In the bulk action mentioned above this parameter is null.

- **overBrokeredUsers:** This flag indicates, in case a broker is logged in, if the search is being done only over users he/she manages (true) or if this is a general search, as member (false).
- Also, as user custom profile fields can be used to search advertisements or records, the same variables bound for those custom fields when used as search filter will also be available for the user profile fields. See below for the extra variables bound for advertisement and record fields when used as search filters.
- **Contact fields (personal contact list):**
When the field is used for creating or modifying a contact:
 - **contact:** The [org.cyclos.entities.users.Contact](#). Even on inserts, is guaranteed to have the 'owner' property set with the [org.cyclos.entities.users.User](#) instance.
 When the field is used for search the contact list:
 - **owner:** The [org.cyclos.entities.users.User](#) instance of the contact owner.
- **Additional contact information (shown in the user profile page):**
When the field is used for creating or modifying an additional contact information:
 - **contactInfo:** The [org.cyclos.entities.users.ContactInfo](#). Even on inserts, is guaranteed to have the 'user' property set with the [org.cyclos.entities.users.User](#) instance.
 There is no additional contact informations search, hence, this script code is never called in this case.
- **Advertisement fields:**
When the field is used for creating or modifying an advertisement:
 - **ad:** The [org.cyclos.entities.marketplace.BasicAd](#). Even on inserts, is guaranteed to have the 'owner' property set with the [org.cyclos.entities.users.User](#) instance.
 When the field is used for searching advertisements:
 - **user:** If searching advertisements of a specific user, contains the [org.cyclos.entities.users.User](#) instance.
 - **adType:** The type of advertisements being searched. Is null when all types are being searched. Otherwise, contains the [org.cyclos.model.marketplace.advertisements.AdType](#) instance.
 - **overBrokeredUsers:** This flag indicates, in case a broker is logged in, if the search is being done only over users he/she manages (true) or if this is a general advertisements search (false).
- **Record fields:**
When the field is used for creating or modifying a record:
 - **record:** The [org.cyclos.entities.users.Record](#). Even on inserts, is guaranteed to have the 'type' property set with the [org.cyclos.entities.users.RecordType](#) instance.

Also, for user records, is guaranteed to have the 'user' property set with the [org.cyclos.entities.users.User](#) instance.

When the field is used for searching records:

- recordType: In most cases, the record type is known, and this contains the [org.cyclos.entities.users.RecordType](#) instance. However, it is also possible to search for records using shared record fields, over multiple record types at the same time. In that case, this variable will be null.
- user: If searching records of a specific user, contains the [org.cyclos.entities.users.User](#) instance. However, in case of general search, or when searching system records, will be null.
- Transaction fields:

When the field is used to perform a payment:

- paymentType: The transaction type, as [org.cyclos.entities.banking.PaymentTransferType](#)
- fromOwner: The [org.cyclos.model.banking.accounts.AccountOwner](#) performing the payment (either [org.cyclos.model.banking.accounts.SystemAccountOwner](#) or [org.cyclos.entities.users.User](#))
- fromOwnerResult: The [org.cyclos.impl.banking.LocateAccountOwnerResult](#) for locating the from account owner
- toOwner: The [org.cyclos.model.banking.accounts.AccountOwner](#) receiving the payment (either [org.cyclos.model.banking.accounts.SystemAccountOwner](#), [org.cyclos.model.banking.accounts.ExternalAccountOwner](#) or [org.cyclos.entities.users.User](#))
- toOwnerResult: The [org.cyclos.impl.banking.LocateAccountOwnerResult](#) for locating the receiver account owner

When the field is used to search an account history:

- account: The account as [org.cyclos.entities.banking.Account](#)
- Custom operation fields:

When the field is used to run a custom operation:

- customOperation: The [org.cyclos.entities.system.CustomOperation](#).
- user: The [org.cyclos.entities.users.User](#). Only present if the custom operation's scope is user.

Custom operation fields are never used on search, so the secondary script code doesn't apply

- Dynamic document fields:

When the document is being printed

- document: The [org.cyclos.entities.contentmanagement.DynamicDocument](#).

Dynamic document fields are never used on search, so the secondary script code doesn't apply

The expected result type should match the custom field type. Must be either one, a collection or an array of:

- Dynamic selection:
 - Strings: In this case, each element will have only values, and the corresponding labels will be the same values.
 - [org.cyclos.model.system.fields.DynamicFieldValueVO](#) (or compatible object / Map): The dynamic field value, containing a value (the internal value) and a label (the display value). The value must be not blank, or an error will be raised. If the label is blank, will show the same text as the value. Also, the first dynamic value with 'defaultValue' set to true will show up by default in the form.
- String: Any returned object will be converted to string
- Integer or decimal: The result may be either numbers or strings
- Date: The script may return instances or either:
 - [java.util.Date](#)
 - [org.cyclos.utils.DateTime](#)
 - Number: in this case will be considered a number of [milliseconds since the epoch date](#)
 - String: a date representation in the [ISO-8601](#) format (yyyy-mm-dd[Thh:mm:ss[+/-offset/timezone-id]]])
- Linked user: The script may return instances or either:
 - [org.cyclos.entities.users.User](#)
 - [org.cyclos.model.users.users.UserVO](#)
 - String: the principal to locate users (login name, account number, e-mail, etc, according to the configuration)
 - Number: user identifier (same as in the database, not the masked id sent to clients)
- Linked transaction: The script may return instances or either:
 - [org.cyclos.entities.banking.Transaction](#)
 - [org.cyclos.model.banking.transactions.TransactionVO](#)
 - String: the transaction number
 - Number: transaction identifier (same as in the database, not the masked id sent to clients)
- Linked transfer: The script may return instances or either:

- [org.cyclos.entities.banking.Transfer](#)
- [org.cyclos.model.banking.transfers.TransferVO](#)
- String: the transfer number
- Number: transfer identifier (same as in the database, not the masked id sent to clients)
- Linked record: The script may return instances or either:
 - [org.cyclos.entities.users.Record](#)
 - [org.cyclos.model.users.records.RecordVO](#)
 - Number: record identifier (same as in the database, not the masked id sent to clients)
- Linked advertisement: The script may return instances or either:
 - [org.cyclos.entities.marketplace.BasicAd](#)
 - [org.cyclos.model.marketplace.advertisements.BasicAdVO](#)
 - Number: advertisement identifier (same as in the database, not the masked id sent to clients)

Examples

Dynamic selection on user profile field: values depending on the user group

This example applies to a custom user profile field, and returns distinct values according to the user group.

```
import org.cyclos.model.system.fields.DynamicFieldValueVO

def values = []
// Common values
values << new DynamicFieldValueVO("common1", "Common value 1")
values << new DynamicFieldValueVO("common2", "Common value 2")
values << new DynamicFieldValueVO("common3", "Common value 3")
if (user.group.internalName == "business") {
  // Values only available for businesses
  values << new DynamicFieldValueVO("business1", "Business value 1")
  values << new DynamicFieldValueVO("business2", "Business value 2")
  values << new DynamicFieldValueVO("business3", "Business value 3")
} else if (user.group.internalName == "consumer") {
  // Values only available for consumers
  values << new DynamicFieldValueVO("consumer1", "Consumer value 1")
  values << new DynamicFieldValueVO("consumer2", "Consumer value 2")
  values << new DynamicFieldValueVO("consumer3", "Consumer value 3")
}
return values
```

And here is the script returning all available values, to be used for search filters:

```
import org.cyclos.model.system.fields.DynamicFieldValueVO

return [
    new DynamicFieldValueVO("common1", "Common value 1"),
    new DynamicFieldValueVO("common2", "Common value 2"),
    new DynamicFieldValueVO("common3", "Common value 3"),
    new DynamicFieldValueVO("business1", "Business value 1"),
    new DynamicFieldValueVO("business2", "Business value 2"),
    new DynamicFieldValueVO("business3", "Business value 3"),
    new DynamicFieldValueVO("consumer1", "Consumer value 1"),
    new DynamicFieldValueVO("consumer2", "Consumer value 2"),
    new DynamicFieldValueVO("consumer3", "Consumer value 3")
]
```

Linked user: list the brokers only

This example applies to a custom field of type linked entity - user. It returns all active brokers in the system, so the user can select one.

```
import org.cyclos.model.access.Role
import org.cyclos.model.users.users.UserQuery
import org.cyclos.model.users.users.UserStatus

def q = new UserQuery()
q.setUnlimited()
q.roles = [Role.BROKER]
q.userStatus = [
    UserStatus.ACTIVE,
    UserStatus.BLOCKED
]
return userService.search(q)
```

Linked transaction on transaction field: list the open loans

This example lists all transactions of a specific payment type (loan grant) to the user performing the payment, filtering by a specific transfer status (open). It could be used on a payment from user to system to repay the loan, which would also need additional processing from a extension point script to mark the loan as repaid (script not included in this example).

```
import org.cyclos.entities.banking.AccountType
import org.cyclos.model.banking.accounts.AccountHistoryQuery
import org.cyclos.model.banking.accounts.AccountVO
import org.cyclos.model.banking.transferstatus.TransferStatusVO
import org.cyclos.model.banking.transfertypes.TransferTypeVO

// Find the account
def accountType = entityManagerHandler.find(AccountType, 'user')
def account = accountService.load(fromOwner, accountType)

// The account history has transfers. We need the transactions.
def q = new AccountHistoryQuery()
q.setUnlimited()
q.account = new AccountVO(account.id)
```

```

q.transferTypes = [
    new TransferTypeVO(internalName: 'debit.loan')
]
q.statuses = [
    new TransferStatusVO(internalName: 'loan.open')
]
def transfers = accountService.searchAccountHistory(q).pageItems

// Return the transaction ids
return transfers.collect {it.transactionId}

```

Account number generation

This kind of script is responsible for generating account numbers, in case more control than the default (random generation) is needed. The script code has the following variables bound (besides the [default bindings](#)):

- type: The [org.cyclos.entities.banking.AccountType](#).
- owner: The [org.cyclos.model.banking.accounts.AccountOwner](#) (either [org.cyclos.model.banking.accounts.SystemAccountOwner](#) or [org.cyclos.entities.users.User](#)).

The script should return a string, which should match the mask set in the configuration (if any). If the script returns null or a blank string, no number is assigned for that account.

The script doesn't need to check if the account number already exists. This is done internally. If the number is already used, the script is called again (up to 10 times, then, an error is raised).

Examples

Controlling the prefix according to the currency and user group

In this example, the mask `##/#####` is expected for the account number. The prefix is composed of 2 digits:

- The first one is 0 if the currency is unit, or 1 otherwise.
- The second one is 0 for system, 1 for business, 2 for consumers or 9 otherwise.

The rest are 7 random digits.

```

import org.cyclos.entities.users.User
import org.cyclos.utils.StringHelper

// Either unit or euro
String prefix = type.currency.internalName == 'internal_units' ? '0' : '1'

if (owner instanceof User) {
    switch (owner.group.internalName) {
        case 'business':
            prefix += '1'
    }
}

```

```

        break
    case 'consumers':
        prefix += '2'
        break
    default:
        prefix += '9'
    }
} else {
    prefix += '0'
}

return prefix + "/" + StringHelper.randomNumeric(7)

```

Account fee calculation

These scripts are used to calculate the amount of an account fee (a fee which is charged periodically or manually over many accounts, according to the 'charged account fees' setting in member products). The script code has the following variables bound (besides the [default bindings](#)):

- fee: The [org.cyclos.entities.banking.AccountFee](#)
- account: The [org.cyclos.entities.banking.UserAccount](#)
- executionDate: The expected fee charge date (of type [java.util.Date](#)). When scheduled, charges usually happen a bit after the exact expected date. For manual account fees, this will be the time the fee has started.

The script should return a number, which will be rounded to the currency's decimal digits. If null or zero is returned, the fee is not charged.

Examples

Charge a different amount according to the user rank

This example allows choosing a distinct account fee amount based on a profile field of the paying user. It is assumed a custom field of type single selection with the internal name rank. It should have 3 possible values, with internal names bronze, silver and gold.

```

// Depending on a user custom field, we'll pick the fee amount
def amounts = [bronze: 10, silver: 7, gold: 5]
def user = scriptHelper.wrap(account.owner)
def rank = user.rank?.internalName ?: "bronze"
return amounts [rank]

```

Transfer fee calculation

These scripts are used to calculate the amount of a transfer fee (a fee triggered by another transfer). The script code has the following variables bound (besides the [default bindings](#)):

- previewParameters: The [org.cyclos.model.banking.transactions.PerformTransactionDTO](#) for the payment being performed. Is only available on the payment preview. On actual processing, is always null. However, this object is useful when the fee amount depends on a some data about the future transaction being performed, such as custom fields.
- fee: The [org.cyclos.entities.banking.TransferFee](#)
- transfer: The [org.cyclos.entities.banking.Transfer](#) which triggered the fee.

The script should return a number, which will be rounded to the currency's decimal digits. If null or zero is returned, the fee is not charged.

Examples

Charging a fee according to a user profile field

This example allows choosing a distinct fee amount based on a profile field of the paying user. It is assumed a custom field of type single selection with the internal name rank. It should have 3 possible values, with internal names bronze, silver and gold. The script then chooses a different percentage according to the user rank.

```
if (transfer.fromSystem) {
    // Only charge users
    return 0
}

// Depending on a user custom field, we'll pick the fee amount
def percentages = [bronze: 0.07, silver: 0.05, gold: 0.02]
def from = scriptHelper.wrap(transfer.fromOwner)
def rank = from.rank?.internalName ?: "bronze"
def percentage = percentages[rank]
return transfer.amount * percentage
```

Charging a fee according to a payment custom field

This example is similar to the above, but based on a transaction custom field in the payment itself. The main difference is the source for custom field values now depend on whether we're calculating the fee during a payment preview (used to show the user the paid fees before the transfer is actually processed) or for the actual transfer processing. That is because the transfer.transaction is not available during preview. However, to allow retrieving the custom fields during preview, there is an extra bound variable, called previewParameters (not available during transfer processing). Similar to the previous example, but this one assumes the single selection field has internal name category, and the possible values have internal names loan, repayment and buying.

```
def percentages = [loan: 0.05, repayment: 0.01, buying: 0.02]
def source = previewParameters ?: transfer.transaction
def bean = scriptHelper.wrap(source)
def category = bean.category?.internalName ?: "buying"
```

```
def percentage = percentages[category]
return transfer.amount * percentage
```

Distributing a fee exactly to different accounts

Some systems charge a fee from users (be it a transfer fee or account fee) which is itself distributed amongst different accounts. For example, a 5% transaction fee is charged from users, and that fee amount is distributed like 12% to account A, 27% to account B and 61% to account C. So, the transaction fee transfer type itself has other 3 fees. The problem in making them all percentage is that each fee charge rounds the charged amount (generally to 2 decimal places, according to the currency), and that may cause the total distributed amount to be different from the total fee amount. A solution for this problem is to make one of the fees calculated by script, so it sums up what each other fee has charged, and charges the remaining. Generally the fee with the largest charge percentage would then use this script, while all other fees will be configured as percentages.

```
import org.cyclos.entities.banking.Transfer
import org.cyclos.model.banking.transferfees.TransferFeeChargeMode
import org.cyclos.utils.BigDecimalHelper

Transfer transfer = binding.transfer
BigDecimal amount = transfer.amount

// Sum what the other fees will charge
int scale = transfer.currency.precision
BigDecimal others = 0
for (def fee in transfer.type.transferFees) {
    if (fee.chargeMode == TransferFeeChargeMode.PERCENTAGE) {
        others += BigDecimalHelper.round(amount * fee.amount, scale)
    }
}

// Charge the rest
return BigDecimalHelper.round(amount - others, scale)
```

Transfer status handling

These scripts are used to determine to which status(es) a transfer may be set after the current status. By default, if no script is used, the possible next statuses (as configured in the transfer status details page) will be available. Using a script, however, allows using finer-grained controls. For example, an specific status could be allowed only by specific administrators, or only under special conditions (for example, checking the account balance or any other condition).

The script code has the following variables bound (besides the [default bindings](#)):

- transfer: The [org.cyclos.entities.banking.Transfer](#)
- flow: The [org.cyclos.entities.banking.TransferStatusFlow](#) of the status being affected.

- status: The [org.cyclos.entities.banking.TransferStatus](#)

The script should return one of the following:

- A single [org.cyclos.entities.banking.TransferStatus](#) (only that status is available as next);
- An array / list / iterator of [org.cyclos.entities.banking.TransferStatus](#) (all are available as next, possibly empty);
- Null – assumes the default behavior: the possible next configured in the status are assumed.

Examples

Restricting a specific status for administrators

In this example, any user can change a transfer status in a given flow. However, only administrators can set a transfer to the status with internal name finished.

```
// Only administrators can set the status to finished
return status.possibleNext.findAll { st ->
    sessionData.admin || st.internalName != "finished"
}
```

Session handling

These scripts can be used to manage user sessions (logins) externally. It can only be set in the network default configuration, as the custom session handling script. There are 4 related operations, each implemented in a code box on this script type:

- Login: Called when a session is created. Should return a session token (string) or an object / Map compatible with [org.cyclos.entities.access.Session](#). If the script returns null, the default login is performed.
- Logout: Called when a user logs out or is disconnected by an administrator. If the script returns null or false, the default logout is performed.
- Resolve: Given the input session token, should return either the logged user (can be either a [org.cyclos.entities.users.BasicUser](#) or a [org.cyclos.impl.users.LocateUserResult](#)) or the [org.cyclos.entities.access.Session](#) directly, which should at least contain the session token, the user attributes and the boolean value indicating if it's a trusted session. If the resolved session is trusted then the correct result must be a [org.cyclos.entities.access.Session](#) to avoid loss the flag when converting the result. If the script returns null, the default session resolution is performed.
- Set properties: Called when properties of a session are being modified.
- Search connected users: This script is called when an administrator searches for connected user, as well as on the administrator home page, as the number of connected users is

shown. Should return either a list or [org.cyclos.utils.Page](#) of results, where each element must be either a [org.cyclos.entities.access.Session](#) or a compatible object, containing at least the sessionToken and user properties filled in. If the script returns null, the default sessions search is performed.

On any of these functions, returning null or having an empty code block will result in the default session management taking place. This way it is possible to implement a custom handling only on special cases. For example, a custom session mechanism might be used only for privileged administrators, whose session tokens comply with an specific format. For reference, Cyclos sessions use 32-character alphanumeric strings, with no punctuation. So, for example, if session tokens generated for those administrators have a different format, say, an [UUID](#), the script can differentiate which sessions tokens correspond to normal sessions (and return null on the Logout and Resolve functions for those token format) and handle only those specific sessions. Also, in such case, the login method could check the user group being logged in, and either perform the login on the underlying system (returning the generated session token) or return null for regular users.

Caution: Errors on any of these functions, specially the first three, may cause users not being able to login or access the system. A good security measure while developing such scripts is to handle a specific (for example, if the login name is 'admin') with the default session resolution, and withdraw this case after the rest of the script is ready. If such situation occurs, a possible workaround is to login in global mode, then disable and lock the custom session handling in the configuration from which the network configuration inherits. Then edit the script and unlock it again in global mode.

The bound variables are:

- user: The [org.cyclos.entities.users.BasicUser](#) performing login. Only available on the login function.
- principal: The [org.cyclos.impl.access.UserPrincipal](#) for the user performing login. Only available on the login function.
- sessionProperties: The [org.cyclos.entities.access.SessionProperties](#) either for the session being created or updated. Only available on the login and setProperties functions.
- trusted: A boolean value indicating if the session will be a trusted one, i.e, the user is performing a login from a trusted device. Trusted sessions doesn't require a confirmation password regardless the channel configuration. Only available on the login function.
- channel: The [org.cyclos.entities.access.Channel](#) representing the channel for which the session should be valid.
- remoteAddress: The remote IP address (string) for which the session should be valid.
- sessionToken: The session token (string) that is either being resolved (on the resolve function) or invalidated (on the logout function).

- sessionTimeout: The [org.cyclos.entities.utils.TimeInterval](#) that should be used as session timeout. Never null, it could be a custom timeout for this specific session or that defined for the corresponding channel configuration.
- query: The [org.cyclos.model.users.users.ConnectedUserQuery](#) for the search function.

Examples

Storing sessions on Cyclos script storages

This example stores user sessions in the Script storage. It is not a realistic example, as Cyclos itself is used to store sessions, but it does demonstrate the usage of a session handling script. Here are the sources for each of the 5 code boxes:

Function to perform the login:

```
import org.apache.commons.lang3.RandomStringUtils
import org.cyclos.entities.access.Channel
import org.cyclos.entities.access.SessionProperties
import org.cyclos.entities.utils.TimeInterval
import org.cyclos.impl.access.UserPrincipal
import org.cyclos.impl.system.ScriptStorageHandler

ScriptStorageHandler scriptStorageHandler = binding.scriptStorageHandler
UserPrincipal principal = binding.principal
TimeInterval sessionTimeout = binding.sessionTimeout;
String remoteAddress = binding.remoteAddress
SessionProperties sessionProperties = binding.sessionProperties
Channel channel = binding.channel

String token = RandomStringUtils.randomAlphanumeric(64)
int timeout = sessionTimeout.milliseconds / 1000
def storage = scriptStorageHandler.get("session_${token}", timeout)
storage.principal = principal
storage.remoteAddress = remoteAddress
storage.timeout = sessionTimeout
storage.sessionProperties = sessionProperties
storage.channel = channel

return token
```

Function to perform the logout:

```
import org.cyclos.impl.system.ScriptStorageHandler

ScriptStorageHandler scriptStorageHandler = binding.scriptStorageHandler
String sessionToken = binding.sessionToken

return scriptStorageHandler.remove("session_${sessionToken}")
```

Function to resolve a session given a token:

```
import org.cyclos.entities.access.Session
```

```

import org.cyclos.impl.system.ScriptStorageHandler

ScriptStorageHandler scriptStorageHandler = binding.scriptStorageHandler
String sessionToken = binding.sessionToken

def storage = scriptStorageHandler.getIfValid("session_${sessionToken}")
Session session = null
if (storage != null) {
    session = new Session()
    session.initFrom(storage.principal)
    session.sessionToken = sessionToken
    session.properties = storage.sessionProperties
    session.channel = storage.channel
    session.remoteAddress = storage.remoteAddress
    session.sessionTimeout = storage.timeout
}
return session

```

Function to set the session properties:

```

import org.apache.commons.lang3.RandomStringUtils
import org.cyclos.entities.access.SessionProperties
import org.cyclos.entities.utils.TimeInterval
import org.cyclos.impl.system.ScriptStorageHandler

import org.cyclos.entities.users.BasicUser
ScriptStorageHandler scriptStorageHandler = binding.scriptStorageHandler
String sessionToken = binding.sessionToken
SessionProperties sessionProperties = binding.sessionProperties

def storage = scriptStorageHandler.getIfValid("session_${sessionToken}")
if (storage != null) {
    storage.sessionProperties = sessionProperties
}

```

Function to search for connected users:

```

import org.cyclos.entities.system.QScriptStorage
import org.cyclos.entities.system.ScriptStorage
import org.cyclos.impl.access.UserPrincipal
import org.cyclos.impl.utils.persistence.EntityManagerHandler
import org.cyclos.model.users.users.ConnectedUserQuery
import org.cyclos.server.utils.JacksonParameterStorage
import org.cyclos.utils.PageImpl
import org.cyclos.utils.StringHelper

import com.fasterxml.jackson.databind.ObjectMapper

ConnectedUserQuery query = binding.query
EntityManagerHandler entityManagerHandler = binding.entityManagerHandler
ObjectMapper objectMapper = binding.objectMapper
QScriptStorage ss = QScriptStorage.scriptStorage

// First we get the persisted script storages which start with 'session_'
PageImpl page = entityManagerHandler
    .from(ss)

```

```

        .where(ss.key.like("session\\_${StringHelper.repeat('_', 64)}", '\\'.charAt(0)),
        ss.expirationDate.after(new Date()))
        .orderBy(ss.creationDate.asc())
        .page(query, ss)

// Each one is parsed as JSON and converted to the expected format
page.pageItems = page.pageItems.collect { ScriptStorage it ->
    def storage = new JacksonParameterStorage(objectMapper, it.content)
    UserPrincipal principal = storage.principal
    return [
        sessionToken: StringHelper.removeStart(it.key, "session_"),
        user: principal.basicUser,
        creationDate: it.creationDate,
        channel: storage.channel,
        remoteAddress: storage.remoteAddress
    ]
}
return page

```

Password handling

These scripts are used to check passwords. In order to use them, the password type's password mode needs to be "Script". The script code has the following variables bound (besides the [default bindings](#)):

- user: The [org.cyclos.entities.users.BasicUser](#) whose password is being checked
- passwordType: The [org.cyclos.entities.access.PasswordType](#) being checked.
- password: The password value being checked (string).

The script should return a boolean, indicating whether the password is ok or not.

Examples

Matching passwords to the script parameters

This is a very simple example, which checks for passwords according to the script parameters. The parameters can be set either in the script itself or in the password type. This example is very insecure, and shouldn't be used in production. Normally, scripts to check passwords would connect to third party applications, but this is just a very basic example.

```

// Just read the password value from the script parameters
return scriptParameters[user.username] == password

```

Extension points

These scripts are used on extension points (user, user record, transfer, ...), and are attached to specific events (create, update, remove, chargeback, ...). The extension point scripts have 2 functions:

- The data has already been validated, but not saved yet. In this function, we know that the data entered by users is valid, but the main event has not been saved yet.
- The data has been saved, but not committed to database yet. For example, if the script code throws an Exception, the database transaction will be rolled-back, and no data will be persisted.

Here are some example scenarios for performing custom logic, or integrating Cyclos with external systems using extension points:

- Custom credit limit. When a user is performing a payment, an extension point of type transaction could be used, in the function invoked after validation, to check the current balance. If the balance is not enough for the payment and the user has credit limit, a payment from a system account could be done automatically to the user, completing the amount for the payment.
- A [XA transaction](#) could be done with an external system by creating data in the external database in the function which runs after validating, then preparing the commit in the function after the data is saved, and finally registering both a commit and a rollback listener (see the ScriptHelper in [default bindings](#)) to either commit or rollback the prepared transaction.
- It is also possible to 'bind' Cyclos entities with extension points. For example a payment could create a new user record of a specific type and set some values in the record. When a user record value is changed this could trigger another action, for example changing the (bookkeeping) status of a payment.
- A simple notification of performed payments could be implemented by registering a commit listener (see the [ScriptHelper](#) in [default bindings](#)) to implement the notification.
- The profile information of a user needs to be mirrored in an external system. In this case, a user extension point, with the create / update events can be used to send this information. Additional information on addresses and phones can use the same mechanism (they are different extension points). Finally, a change status event for users, to the status [REMOVED](#) indicates that the user has been removed.
- There could be payment custom fields which are not filled-in by users when performing payments, but by extension points of type transaction. Payment custom fields may be configured to not show up in the form, only automatically via extension points.
- An extension point on a new Cyclos advertisement could publish the advertisement as well in an third party system.

These are just some examples. There are many possible uses for the extension points. In the future we will publish usefull extension points at this site.

All extension points have the following additional variables bound to its execution:

- extensionPoint: The [org.cyclos.entities.system.ExtensionPoint](#)

- event: The [org.cyclos.model.system.extensionpoints.ExtensionPointEvent](#). The specific implementation depends on the extension point type.
- context: A `java.util.Map<String, Object>` which can be used to store attributes to be shared between, for example, the script which runs after the data is validated, and the one which runs after the data is saved

The following types of extension points exist:

User extension point

Extension points which monitor events on users, including administrators, brokers and regular users. Additional bindings:

- user: The [org.cyclos.entities.users.User](#)

Events:

- create: a user is being registered. IMPORTANT: When e-mail validation is enabled, the user will be pending until confirming the e-mail. If you have e-mail confirmation enabled, this event might not be what you need, but activate instead.
- activate: a user is being activated for the first time. For example, if e-mail validation is enabled, after the user confirming the e-mail address this event will be triggered. However, the initial status for users (set in group) might be, for example, disabled. In that case, only when the user is first activated this event will be triggered.
- update: a user profile (full name, login name, e-mail or custom fields) is being edited. Additional bindings:
 - currentCopy: A detached copy of the user being edited, as [org.cyclos.entities.users.User](#)
- changeGroup: The user's group is being changed.
 - oldGroup: The current [org.cyclos.entities.users.Group](#)
 - newGroup: The new [org.cyclos.entities.users.Group](#)
 - comments: The comments, as provided by the administrator when changing the group, as string.
- changeStatus: The user's status is being changed. Argument Map:
 - oldStatus: The current [org.cyclos.model.users.users.UserStatus](#)
 - newStatus: The new [org.cyclos.model.users.users.UserStatus](#)
 - comments: The comments, as provided by the administrator when changing the status, as string.

Operator extension point

Extension points which monitor events on operators. Additional bindings:

- operator: The [org.cyclos.entities.users.Operator](#)

Events:

- create: an operator is being registered.
- update: an operator profile (full name, login name, e-mail or custom fields) is being edited.
Additional bindings:
 - currentCopy: A detached copy of the operator being edited, as [org.cyclos.entities.users.Operator](#)
- changeGroup: The operator's group is being changed.
 - oldGroup: The current [org.cyclos.entities.users.OperatorGroup](#). When null, the operator had the full permissions of the user.
 - newGroup: The new [org.cyclos.entities.users.OperatorGroup](#). When null, the operator will have the full permissions of the user.
 - comments: The comments, as provided by the administrator / user when changing the group, as string.
- changeStatus: The operator's status is being changed. Argument Map:
 - oldStatus: The current [org.cyclos.model.users.users.UserStatus](#)
 - newStatus: The new [org.cyclos.model.users.users.UserStatus](#)
 - comments: The comments, as provided by the administrator / user when changing the status, as string.

Address extension point

Extension points which monitor events on addresses. Additional bindings:

- address: The [org.cyclos.entities.users.UserAddress](#)

Events:

- create: An address is being created.
- update: An address is being updated. Additional bindings:
 - currentCopy: A detached copy of the address being edited, as [org.cyclos.entities.users.UserAddress](#)
- delete: An address is being deleted.

Phone extension point

Extension points which monitor events on user phones. Additional bindings:

- phone: The [org.cyclos.entities.users.Phone](#)

Events:

- create: A phone is being created.
- update: A phone is being updated. Additional bindings:
 - currentCopy: A detached copy of the phone being edited, as `org.cyclos.entities.users.Phone`
- delete: A phone is being deleted.

Record extension point

Extension points which monitor events on records, either user or system records. Additional bindings:

- record: The [org.cyclos.entities.users.Record](#)

Events:

- create: A record is being created.
- update: A record is being created. Additional bindings:
 - currentCopy: A detached copy of the record being edited, as [org.cyclos.entities.users.Record](#)
- delete: A record is being created.

Advertisement extension point

Extension points which monitor events on advertisements. Additional bindings:

- ad: The [org.cyclos.entities.marketplace.BasicAd](#)

Events:

- create: An advertisement is being created.
- update: An advertisement is being updated. Additional bindings:
 - currentCopy: An advertisement is being updated. Additional bindings: [org.cyclos.entities.marketplace.BasicAd](#)
- delete: An advertisement is being deleted.

Transaction extension point

Extension points which monitor events on performed transactions.

The following additional bindings are available for both preview and confirm events:

- performTransaction: The [org.cyclos.model.banking.transactions.PerformTransactionDTO](#)
- paymentType: The transaction type, as [org.cyclos.entities.banking.PaymentTransferType](#)
- fromAccount: The source [org.cyclos.model.banking.accounts.AccountOwner](#) performing the payment (either [org.cyclos.entities.banking.Account](#)

- fromOwner: The [org.cyclos.model.banking.accounts.AccountOwner](#) performing the payment (either [org.cyclos.model.banking.accounts.SystemAccountOwner](#) or [org.cyclos.entities.users.User](#))
- fromOwnerResult: The [org.cyclos.impl.banking.LocateAccountOwnerResult](#) for locating the from account owner
- toAccount: The destination [org.cyclos.model.banking.accounts.AccountOwner](#) performing the payment (either [org.cyclos.entities.banking.Account](#)
- toOwner: The [org.cyclos.model.banking.accounts.AccountOwner](#) receiving the payment (either [org.cyclos.model.banking.accounts.ExternalAccountOwner](#), [org.cyclos.model.banking.accounts.SystemAccountOwner](#) or [org.cyclos.entities.users.User](#))
- toOwnerResult: The [org.cyclos.impl.banking.LocateAccountOwnerResult](#) for locating the receiver account owner
- authorizationType: The [org.cyclos.model.banking.transactions.TransactionAuthorizationType](#) of the transaction, if it would be pending authorization, or null if already processed. For the confirm event, will only be available in the script which runs after save.
- authorizationLevel: The [org.cyclos.entities.banking.AuthorizationLevel](#) of the transaction, if it would be pending authorization by level, or null otherwise. For the confirm event, will only be available in the script which runs after save.

Events:

- preview: The user is previewing the transaction. Note that, as there is nothing really being saved, both scripts will run at the same time, i.e., there's no phase 'after validate' and 'after save'. WARNING: This event runs in a readonly transaction, and if the script writes anything to the database, it will fail. The error displayed on the application is a general one, like "There was an error while accessing the database". Make sure to never store anything in the database in a script that runs in this event. Additional bindings:
 - preview: The [org.cyclos.model.banking.transactions.TransactionPreviewVO](#)
- confirm: The transaction has been confirmed, that is, is being performed. Additional bindings:
 - transaction: The [org.cyclos.entities.banking.Transaction](#). Only available for the script which runs after save.
- change status: The transaction status has changed. The kinds of transactions that have status, and the corresponding class for each status are:
 - Scheduled payments: [org.cyclos.model.banking.transactions.ScheduledPaymentStatus](#);
 - Recurring payments: [org.cyclos.model.banking.transactions.RecurringPaymentStatus](#);

- Payment requests: [org.cyclos.model.banking.transactions.PaymentRequestStatus](#);
 - Tickets: [org.cyclos.model.banking.transactions.TicketStatus](#);
 - External payments: [org.cyclos.model.banking.transactions.ExternalPaymentStatus](#).
- Additional bindings:
- transaction: The [org.cyclos.entities.banking.Transaction](#) instance;
 - oldStatus: The previous status;
 - newStatus: The new status.
- change installment status: A scheduled payment installment or recurring payment occurrence status has changed. Additional bindings:
 - installment: The [org.cyclos.entities.banking.Installment](#).
 - oldStatus: The previous status, as [org.cyclos.model.banking.transactions.InstallmentStatus](#).
 - newStatus: The new status, as [org.cyclos.model.banking.transactions.InstallmentStatus](#).
 - send payment request: A payment request is being sent. Additional bindings:
 - performTransaction: The [org.cyclos.model.banking.transactions.SendPaymentRequestDTO](#)
 - paymentType: The transaction type, as [org.cyclos.entities.banking.PaymentTransferType](#)
 - fromOwner: The [org.cyclos.model.banking.accounts.AccountOwner](#) performing the payment (either [org.cyclos.model.banking.accounts.SystemAccountOwner](#) or [org.cyclos.entities.users.User](#))
 - fromOwnerResult: The [org.cyclos.impl.banking.LocateAccountOwnerResult](#) for locating the from account owner
 - toOwner: The [org.cyclos.model.banking.accounts.AccountOwner](#) receiving the payment (either [org.cyclos.model.banking.accounts.ExternalAccountOwner](#), [org.cyclos.model.banking.accounts.SystemAccountOwner](#) or [org.cyclos.entities.users.User](#))
 - toOwnerResult: The [org.cyclos.impl.banking.LocateAccountOwnerResult](#) for locating the receiver account owner
 - create ticket: A ticket is being created. Additional bindings:
 - performTransaction: The [org.cyclos.model.banking.transactions.CreateTicketDTO](#)
 - paymentType: The transaction type, as [org.cyclos.entities.banking.PaymentTransferType](#)
 - fromOwner: The [org.cyclos.model.banking.accounts.AccountOwner](#) performing the payment (either [org.cyclos.model.banking.accounts.SystemAccountOwner](#) or [org.cyclos.entities.users.User](#))

- fromOwnerResult: The [org.cyclos.impl.banking.LocateAccountOwnerResult](#) for locating the from account owner

Transaction authorization extension point

Extension points which monitor transaction authorization actions. Additional bindings:

- transaction: The [org.cyclos.entities.banking.Transaction](#)
- currentLevel: The current [org.cyclos.entities.banking.AuthorizationLevel](#)
- comment: The comment entered by the user performing the action, as string

Events:

- authorize: The transaction is being authorized. Be careful: there might be more authorization levels which need to be authorized before the transaction is finally processed. Additional bindings:
 - nextLevel: The next current [org.cyclos.entities.banking.AuthorizationLevel](#). If the transfer should be processed after the current authorization is saved, this value will be null.
- deny: The transaction is being denied by the authorizer.
- cancel: The transaction is being canceled by the performer.
- expire: The transaction is being expired by the system through a polling task. If the transfer type requires authorization, it is possible to define an expiration period to avoid leaving the payment indefinitely pending state.

Transfer extension point

Argument Map (common for all events):

- transfer: The transfer being affected.

Events:

- create: A transfer is being created.
- chargeback: A transfer is being charged-back. Invoked once for the main transfer, not being invoked for each fee transfers, if any. In the case that the script needs access to the fee transfers, the collection [org.cyclos.entities.banking.Chargeback.getChargebackTransfers\(\)](#) will contain all charged-back transfers (including the original). Additional bindings:
 - chargeback: The [org.cyclos.entities.banking.Chargeback](#). Only available in the script which runs after the data is saved.
- changeStatus: The transfer is being set to a new status. Additional bindings:
 - flow: The [org.cyclos.entities.banking.TransferStatusFlow](#) of the status being changed
 - oldStatus: The current [org.cyclos.entities.banking.TransferStatus](#)

- newStatus: The new [org.cyclos.entities.banking.TransferStatus](#)
- comments: The comments, as provided by the administrator when changing the status, as string.

Voucher extension point

Argument Map (common for all events):

- voucher: The voucher being affected.

Events:

- generate: A voucher is being generated.
- buy: A voucher is being bought by a user.
- redeem: A voucher is being redeemed. Additional bindings:
 - redeemer: The voucher redeemer
- cancel: A generated voucher is being canceled.
- expire: A voucher is being expired.

Agreement extension point

Extension points which triggers when an agreement is accepted or when an optional agreement is no longer accepted by a user.

On public registrations, if the e-mail validation is enabled, the events won't trigger. Only after the user is activated, an event will trigger for each of the accepted agreements.

Additional bindings:

- agreement: The [org.cyclos.entities.access.Agreement](#) being accepted / no longer accepted
- user: The [org.cyclos.entities.users.User](#) performing the action

Events:

- accept: An agreement is being accepted.
- reject: An optional agreement which was previously accepted is no longer accepted.

Import extension points

Extension points which monitor events on imports, such as when importing users, transfers, transactions, etc. Additional bindings:

- importedFile: The [org.cyclos.entities.system.ImportedFile](#) being processed

Events:

- File status changed: The whole imported file status has changed. Additional bindings:

- oldStatus: The previous imported file status, as [org.cyclos.model.system.imports.ImportedFileStatus](#)
- newStatus: The new imported file status, as [org.cyclos.model.system.imports.ImportedFileStatus](#)
- Line read: An imported line was read from the CSV file. Additional bindings:
 - line: The [org.cyclos.entities.system.ImportedLine](#) being read
- Line processed: A line is being processed. On the validated phase the line isn't yet processed. On the saved phase, the line was processed, either with success or error. Additional bindings:
 - line: The [org.cyclos.entities.system.ImportedLine](#) being processed
 - entity: Only on the saved phase when success (null when error). The entity which was created. The actual type depends on the import type. Can be a user, an advertisement, a transfer, a transaction, a record, a voucher, etc.
 - error: Only on the saved phase when error (null when success). The Java error which was thrown when processing the line

Examples

Granting extra credit (on demand) before payments

This example allows, with a custom profile field, to define an extra credit limit the user can use on demand. When performing a payment, if the available balance is not enough, a payment is performed from a system account to the user, up to the limit specified in that profile field. Once the payment is done, the profile field is subtracted. This example expects the system account to have the internal name debitUnits, and it should have a payment transfer type to the user account. That payment transfer type should have the internal name extraCredit. Finally, the custom profile field needs to have the internal name availableCredit, and needs to be of type decimal, and enabled for the user. Then create an extension point of type Transaction, enabled and for the confirm event. This example only works for payments without fees. Use this in the "Script code executed when the data is saved" code block:

```
import org.cyclos.entities.banking.Account
import org.cyclos.entities.banking.PaymentTransferType
import org.cyclos.entities.banking.SystemAccountType
import org.cyclos.model.banking.accounts.SystemAccountOwner
import org.cyclos.model.banking.transactions.PerformPaymentDTO
import org.cyclos.model.banking.transfertypes.TransferTypeVO

// Only process direct payments. Scheduled payments are skipped
if (!(performTransaction instanceof PerformPaymentDTO)) {
    return
}

// Get the available credit as a profile field
```

```

def payer = scriptHelper.wrap(fromOwner)
BigDecimal availableCredit = payer.availableCredit?.abs()
if (availableCredit == null || availableCredit < 0.01) {
    // Nothing to do - no available credit
    return
}

// Get the account and balance
Account account = accountService.load(fromOwner, paymentType.from)
BigDecimal availableBalance = accountService.getAvailableBalance(account, null)
BigDecimal needs = performTransaction.amount - availableBalance
if (needs > 0 && needs <= availableCredit) {
    // Needs some extra credit, and has it available - make a payment from system
    // Find the system account and payment type
    SystemAccountType systemAccountType = entityManagerHandler.find(
        SystemAccountType, "debitUnits")
    PaymentTransferType paymentType = entityManagerHandler.find(
        PaymentTransferType, "extraCredit", systemAccountType)
    PerformPaymentDTO credit = new PerformPaymentDTO()
    credit.from = SystemAccountOwner.instance()
    credit.to = fromOwner
    credit.type = new TransferTypeVO(paymentType.id)
    credit.amount = needs
    paymentService.perform(credit)
    // Now there should be enough credit to perform the payment

    // Update the user available credit
    payer.availableCredit -= needs
}

```

Send an e-mail on every payment

This example allows, for the selected payment types in the extension point details, to send an e-mail to a specific address. Use this in the "Script code executed when the data is saved" code block:

```

import javax.mail.internet.InternetAddress

import org.cyclos.model.ValidationException
import org.cyclos.server.utils.MessageProcessingHelper
import org.springframework.mail.javamail.MimeMessageHelper

// Get the e-mail subject and body
def tx = scriptHelper.wrap(transaction)
def vars = [
    payer: tx.fromOwner.name,
    amount: formatter.format(tx.currencyAmount),
    date: formatter.formatAsDate(new Date()),
    time: formatter.formatAsTime(new Date())
]
def subject = MessageProcessingHelper.processVariables(scriptParameters.subject, vars)
if (subject == null || subject.empty) {
    throw new ValidationException("Missing the 'subject' script parameter")
}
def body = MessageProcessingHelper.processVariables(scriptParameters.message, vars)
if (body == null || body.empty) {

```

```

        throw new ValidationException("Missing the 'message' script parameter")
    }
    def toEmail = tx.email
    def fromEmail = sessionData.configuration.smtpConfiguration.fromAddress
    def sender = mailHandler.mailSender

    // Send the message after commit, so we guarantee the transaction is persisted
    // when the e-mail is sent
    scriptHelper.addOnCommit {
        def message = sender.createMimeMessage()
        def helper = new MimeMessageHelper(message)
        helper.to = new InternetAddress(toEmail)
        helper.from = new InternetAddress(fromEmail)
        helper.subject = subject
        helper.text = body
        // Send the message
        sender.send message
    }

```

Assign / unassign individual products when the user accepts / rejects agreements

Starting with Cyclos 4.13, there are optional agreements. This example assigns / unassigns individual products to the user that accepts / rejects agreements.

The script parameters must be in the form:

```

agreementInternalName1=productInternalName1
agreementInternalName2=productInternalName2
...

```

Make sure the "Run with all permissions" checkbox is selected. Then, use this script in the "Script code executed when the data is saved" code block:

```

import org.cyclos.entities.access.Agreement
import org.cyclos.entities.users.User
import org.cyclos.impl.users.ProductsUserServiceLocal
import org.cyclos.model.system.extensionpoints.AgreementExtensionPointEvent
import org.cyclos.model.users.products.ProductVO
import org.cyclos.model.users.users.UserVO

// Get the variables from context
AgreementExtensionPointEvent event = binding.event
User user = binding.user
Agreement agreement = binding.agreement
ProductsUserServiceLocal productsUserService = binding.productsUserService
Map<String, String> scriptParameters = binding.scriptParameters

// Lookup the product by agreement internal name
def productVO = new ProductVO(internalName: scriptParameters[agreement.internalName])
def userVO = new UserVO(user.id)
def assigned = user.products.find { it.internalName == productVO.internalName } != null

if (event == AgreementExtensionPointEvent.ACCEPT) {
    // Assign the individual product

```

```

    if (!assigned) {
        productsUserService.assign(productVO, userVO)
    }
} else {
    // Unassign the individual product
    if (assigned) {
        productsUserService.unassign(productVO, userVO)
    }
}
}

```

In the extension point itself, select all agreements whose internal names are included in the script parameters, the user groups and both events.

Enforcing the user remains with a minimum balance for a payment type

This example forces the user to remain with a minimum balance for the payment types configured in the extension point. The extension point should be of type transaction, and the event should be confirm. Paste the following on "Script code executed when the data is validated" code block:

```

import org.cyclos.entities.banking.Account
import org.cyclos.entities.utils.CurrencyAmount
import org.cyclos.impl.banking.AccountServiceLocal
import org.cyclos.impl.utils.formatting.FormatterImpl
import org.cyclos.model.ValidationException
import org.cyclos.model.banking.transactions.PerformTransactionDTO

Account account = binding.fromAccount
PerformTransactionDTO performTransaction = binding.performTransaction
AccountServiceLocal accountService = binding.accountService
Map<String, String> scriptParameters = binding.scriptParameters
FormatterImpl formatter = binding.formatter

def minBalance = new BigDecimal(scriptParameters.minBalance)

def balance = accountService.getBalance(account, null)
def newBalance = balance - performTransaction.amount
if (newBalance < minBalance) {
    throw new ValidationException("""This operation cannot be processed,
        as your new account balance would be
        ${formatter.format(new CurrencyAmount(account.currency, newBalance))},
        below the minimum allowed balance of
        ${formatter.format(new CurrencyAmount(account.currency, minBalance))}""")
}

```

Custom operations

These scripts are invoked when a user runs a custom operation. A custom operation is configured to return different data types, and the script must behave accordingly (see [System – Operations](#) for more details).

Custom operations can have different scopes:

- System: Those are executed by administrators (with granted permissions), directly from the main menu;
- User: Custom operations which are related to a user, and can either be executed by the own user (with granted permissions), from the main menu or run by administrator or brokers (also, with granted permissions) when viewing the user profile. In both cases, the custom operation needs to be enabled to users via member products. For example, there might be operations which applies only to businesses, not consumers, and even administrators with permission to run them shouldn't be able to run them over consumers. It is enforced that administrators / brokers will only be able to run custom operations over users they manage;
- Menu: These custom operations are executed by a custom menu entry. This is the only possible custom operation scope that can be run by guests. A classical example of this is a "Contact us" page;
- Internal: An internal custom operation is executed either as an action (see below) or when the user clicks a row returned by another custom operation which returns a table with results;
- Advertisement: Custom operations which are executed over an advertisement;
- Record: Custom operations which are executed over a record;
- Transfer: Custom operations which are executed over a transfer (balance transfer between accounts);
- Contact: Custom operations which are executed over a contact in a user's contact list;
- Additional contact information: Custom operations which are executed over an additional contact information in a user's profile;
- Bulk action: Custom operations executed on bulk actions, over each user individually.

Bound variables:

- customOperation: The [org.cyclos.entities.system.CustomOperation](#)
- user: The [org.cyclos.entities.users.User](#). Only present if the custom operation's scope is either user or bulk action.
- bulkAction: The [org.cyclos.entities.users.CustomOperationBulkAction](#). Only present if the custom operation's scope is bulk action.
- ad: The [org.cyclos.entities.marketplace.BasicAd](#). Only present if the custom operation's scope is advertisement.
- record: The [org.cyclos.entities.users.Record](#). Only present if the custom operation's scope is record.

- transfer: The [org.cyclos.entities.banking.Transfer](#). Only present if the custom operation's scope is transfer.
- contact: The [org.cyclos.entities.users.Contact](#). Only present if the custom operation's scope is contact.
- contactInfo: The [org.cyclos.entities.users.ContactInfo](#). Only present if the custom operation's scope is additional contact information.
- menuItem: The [org.cyclos.entities.contentmanagement.MenuItem](#). Only present if the custom operation's scope is menu.
- inputFile: The [org.cyclos.model.utils.FileInfo](#). Only present if the custom operation is configured to accept a file upload, and if a file was selected.
- formParameters: A `java.util.Map<String, Object>`, keyed by the form field internal name. The value depends on the form field type. Could be a string, a number (Integer or BigDecimal), a boolean, a date (`java.util.Date`), a [org.cyclos.entities.system.CustomFieldPossibleValue](#), a collection of [org.cyclos.entities.system.CustomFieldPossibleValues](#), or a linked entity, such as [org.cyclos.entities.users.User](#), [org.cyclos.entities.users.Record](#), [org.cyclos.entities.banking.Transfer](#) or [org.cyclos.entities.banking.Transaction](#).
- exportFormat: The [org.cyclos.entities.system.ExportFormat](#) indicating if an operation which returns a result page will return just the data (when null) or will be exported to a file.
- currentPage: An integer indicating the current page, when getting paged results. Starts with zero. Only available if the result type is result page.
- pageSize: An integer indicating the requested page size when getting paged results. Only available if the result type is result page.
- skipTotalCount: A boolean indicating whether the total count should be skipped for this search.
- returnUrl: Only if the custom operation return type is external redirect. Contains the url (as string) which Cyclos expects the external site to redirect the user after the operation completes.
- storage / parameterStorage / : Only if the custom operation return type is external redirect. Contains an [ObjectParameterStorage](#) which is shared in both the first script and the callback handling script. This object is enhanced with `propertyMissing` methods, to support "syntactic sugar" on Groovy scripts, like `parameterStorage.name = value`. When this form is used, it is assumed that the input / output are plain strings.
- execution / externalRedirectExecution: Only if the custom operation return type is external redirect. Contains the [ExternalRedirectExecution](#) which stores the context for this execution.

- request: The [org.cyclos.model.utils.RequestInfo](#). Only if the custom operation return type is external redirect. Contains the information about the current request, so the script function which handles the callback can identify the context to complete the process.

Return value:

The required return value depends on the custom operation result type. In all cases, the result type for the `CustomOperationService.run()` method is a [org.cyclos.model.system.operations.RunCustomOperationResult](#). But, depending on the custom operation result type, the value returned by the script is handled differently, as shown below:

- Plain text or Rich text: In these cases, the result has a title and a content. The script must return one of the following:
 - A plain string, which is considered as the result content. The header will be the custom operation name;
 - An object (or map) containing the following properties:
 - content: The result content;
 - title: The result title.
- Notification: In all cases, notifications are assumed to be HTML formatted. The script must return one of the following:
 - A plain string, which is considered as an information notification;
 - A string prefixed with either [INFO], [WARN] or [ERROR]. In this case, those prefixes are removed from the notification and the notification level is set ;
 - An object (or map) containing the following properties:
 - notification: The notification text
 - notificationLevel: A value of [org.cyclos.model.utils.NotificationLevel](#), defaulting to information.
- File download: The script must return an instance of [org.cyclos.model.utils.FileInfo](#), or an object or Map with the same properties. The properties are:
 - content: Required. The file content. May be an InputStream, a File or a String (containing the file content itself).
 - contentType: Required. The MIME type, such as text/plain, text/html, image/jpeg, application/pdf, etc.
 - name: Optional file name, which will be used by browsers to suggest the file name to save.
 - length: Optional file length, which may aid browsers to monitor the progress of file downloads.

- Page results: The script must return an object (or map) with the following properties:
 - columns: Either this or headers must be returned. Contains each column definition. Each column is a [org.cyclos.model.system.operations.PageResultColumn](#) or equivalent object. Each column can define a result property to display (otherwise it is assumed that each result is an array, accessed by index). Additionally, defines the type, header, width, align, vertical align. The type is a [org.cyclos.model.system.operations.PageResultColumnType](#) or equivalent string, such as: 'string', 'boolean', 'number', 'date' or 'currencyAmount'. Returning as currency amount is a special case, where exports can use this information to correctly format the amount. Also, when the type is boolean, number or date, results are sent with a suitable representation in a standard form, rather than formatted.
 - headers: Can be returned instead of columns. A list containing the column headers. Is supported to ease simple cases and to maintain compatibility with scripts written from Cyclos versions before 4.5.
 - rows: Optional. A list of objects, each containing properties. Each column matches the corresponding object property to display each cell. An object can have additional properties, which can be used to pass parameters to the url when clicking a row.
 - results: Optional. Can be returned instead of rows. A list of lists, containing the table cells. The inner lists should have the same size as the columns.
 - totalCount: Optional, used to page results. If a total count is returned, a result page navigator is shown to the user, and records can be returned page-by-page. The script should probably use the currentPage and pageSize bound variables.
 - hasNextPage: Indicates that there are more rows to be returned than this page. Ignored if the totalCount is returned. Another way to return this is to limit results to 'pageSize + 1'. When more results are returned than the page size, the list is truncated, but Cyclos has the information that there's more data.
- URL: The script must return one of the following:
 - A plain string, which is considered as the URL, and the user is redirected to that URL in the same browser window;
 - An object (or map) containing the following properties:
 - url: The destination URL
 - newWindow: A boolean value indicating whether the application will open a new browser window with the destination URL. Most browsers block popups by default, and opening in a new window is probably considered a popup by browsers. Hence, when opening a new window, on the first execution, users might be prompted whether the popup is allowed. Then they might need to run the operation again once the popup is allowed.

- External redirect: This return type has 2 different scripts:
 - The first script should prepare the data in some external system, and then return the URL to which the user should be redirected. An example using this kind of script is the PayPal Integration, in which the first script creates a payment in PayPal, to later one be confirmed by the user. Two noteworthy variables bound to the script context which are necessary for this script are:
 - returnUrl: The Cyclos URL that will call the second script. Normally, external services receive such URL to redirect the user once the operation is finished.
 - parameterStorage: A [parameter storage](#) which can be used to share data between the first and the second script. Any data stored here will be automatically persisted and retrieved for the second script. There is a limit of a few hours (4-5) for the same execution context to be valid between the first and the second scripts, after which it is removed.
 - The second script is triggered after the external site redirects the user back to Cyclos. This script must return an HTML content which is shown to the user. After being redirected back to Cyclos, the previous web application state, such as breadcrumb, current page, etc, will be lost. Just the returned HTML content will be shown.
- Bulk action: The script is executed for each user affected by the bulk action, and must return one of the following:
 - Null: Represents the user was skipped;
 - Boolean: True represents the user was processed, false means the user was skipped;
 - Status: If the result is a [org.cyclos.model.users.bulkactions.BulkActionUserStatus](#), or a string corresponding to one of its item names, a that status is assumed, with no message;
 - String: The string (unless represents the name of a BulkActionUserStatus item) represents the message, assuming the user was successfully processed;
 - Throwable: An error. Normally errors are expressed by throwing exceptions, but it is also possible to return one;
 - Specific result: An instance of [org.cyclos.impl.users.BulkActionUserResult](#).

Additionally, as part of the returned result object, you can specify what to do after a successful operation execution. Although you can specify that information for all result types, the Cyclos web application will process it only for 'Notification', 'URL' (with 'newWindow' in true) and 'External redirect' result types (except for 'autoRunAction'). The following properties can be specified in the result:

- backTo: Contains the [org.cyclos.model.system.operations.CustomOperationVO](#) to go back. If the page containing the given operation is not found in the history then the Cyclos web application will stay in the current page;

- **backToRoot:** A boolean value indicating if the application must go back to the page that originated the custom operation executions. If we already are in a 'root page' then the Cyclos web application will stay in the current page. For example, an operation with scope 'User' containing an action (action 1) and this in turn containing another action (action 1.1) could generate the following pages history: View user profile → Run user custom operation → Run Custom operation action1 → Run Custom operation action1.1, in this case the flag 'backToRoot' in true means go back to the 'View user profile' page;
- **reRun:** A boolean value indicating if the page we went back to or the current one (if 'backTo' was not specified or 'backToRoot' is false) must be executed again before display it;
- **autoRunAction:** Either the id or internal name of the action that should be executed automatically. If it is specified, the Cyclos web application don't show the result and run the action automatically, as it will if the user manually execute it by the corresponding action button.

The custom operation scripts also support 3 other script blocks. They both receive the following bind variables:

- [The custom operation](#);
- Either the user, record, advertisement, contact, contactInfo or transfer (depending on scope). See [Bound variables](#);
- [The form parameter](#): The map will contains only the custom fields passed as parameter to the operation. When running any custom operation through the Cyclos web application the parameters contained in this map will be those mapped in the action definition plus those defined by the script of the action container operation (i.e the operation containing the action). If used through the REST API then you could pass any extra operation custom field as parameter and it will be contained in the Map. In case of the [script used to check for the availability](#) of an operation, if the operation is not an 'Internal' one then this map will be binded (to avoid errors in the script if it references the variable) but empty.

These blocks are:

- **Code executed before the form is show, to fill the initial field values:** This script will be executed before showing the form the user enters the script parameters (custom fields), so it can determine the default form parameters dynamically. It should return a `java.util.Map<String, Object>`, containing, per custom field internal name, the default value that should be presented for the user. Additionally, as part of the returned result object, you can specify an action to run automatically with the following property:
 - **autoRunAction:** Either the id or internal name of the action that should be executed automatically. The Cyclos web application ignore this property if it is specified, it will show the form with the actions or the result if the operation can run directly.
- **Code executed to determine whether the custom operation will be available:** This script can decide to disable the custom operation from being shown as an option to be executed. For

example, for scope Record, some custom operation could only make sense if the record has a particular custom field value. If this script returns false, the operation will not be shown as an option. Any other value will enable the operation. This script will also run for custom operations used as [actions \(see below\)](#) of other custom operations. In this case, the 'formParameters' bound variable will contain only the parameters that would be sent to the action, if it is executed, according to the mapped values in the action configuration. Also, for actions, an additional available variable for the script is containerCustomOperation, which is main custom operation which is currently being executed, and that will contain the action.

- Script code executed when the external site redirects the user back to Cyclos: This script is executed only if the operation result type is 'External redirect'. It runs after the external service redirects the user back to Cyclos. The script will have access to the same 'storage' object that was available to the main script block, so using that object it is possible to pass data between both executions. The callback also runs with the same 'sessionData' as the original script, they will have the same logged user, permissions, etc. Additionally, it is possible to read the original request parameters using the 'request' variables.

Actions

Custom operations can have additional actions. Each action points to another custom operation with scope 'Internal'. The original custom operation can be configured for which actions are available, which parameters are passed to that action and the visibility (when are shown to the user). Each parameter of the action operation may be mapped to a parameter of the original custom operation or left for the original operation script to resolve the parameters that will be set to the action operation. The visibility is applied to know if an action should be shown to the user, it can be before, after or both (before and after) the operation is executed.

Actions can be configured on custom operations of result type 'Plain text', 'Rich text' or 'Result page'. The label defined for the custom operation pointed by an action will be used as the button label associated to that action, as the full name could be too large for buttons.

To control the actions, the corresponding original script has to set a property named 'actions' as part of the returned result. It should be a map keyed by the action operation internal name, and whose values contains the following properties:

- parameters: Contains another map, keyed by parameter (form field of the action operation) internal name, with the value that should be used as that parameter. If there is a static mapping between an action parameter and an owner operation input field, and the script returns a parameter value, the script takes precedence;
- enabled: Whether the action must be enabled or not. Default to true.

Actions showed before the execution of the original custom operation are customized by [script executed before the form is show](#) and those showed after are customized by the [execution script](#).

Here is an example of a script for a custom operation of result type 'Rich text' with two actions:

```
return [
  content: "This is the content displayed after the operation is executed",
  actions: [
    action1: [
      // action1 is the internal name of the custom operation (with scope 'Internal')
      pointed by an action.
      parameters: [
        input1: "Value for input 1",
        input2: "1234" // Here input1 and input2 are internal names of
        // form fields in the action1 custom operation
      ]
      // action1 is enabled by default
    ],
    action2: [
      enabled: false // action2 is disabled for this execution
    ]
  ]
]
```

Here is an example of a script executed before show the form for a custom operation of result type 'Rich text' with two actions (with visibility before or both) :

```
return [
  field1: "Default value" // Here field1 is a form field in the custom operation
  actions: [
    action1: [
      // action1 is the internal name of the custom operation (with scope 'Internal')
      pointed by an action.
      parameters: [
        input1: "Value for input 1",
        input2: "1234" // Here input1 and input2 are internal names of
        // form fields in the action1 custom operation
      ]
      // action1 is enabled by default
    ],
    action2: [
      enabled: false // action2 is disabled before the execution
    ]
  ]
]
```

Examples

Contact us page

This example allows creating a "contact us" page, which sends an e-mail to a specified address. To use it, you will need the following content in the script parameters box:

```
to=admin@project.org
from=noreply@project.org
subject=Contact form
message=The message was sent.\nThank you for your contact.
```



```
mailHeader=An user has sent a contact form with the following data:
mailFrom=From:
mailEmail=E-Mail:
mailSubject=Subject:
mailMessage=Message:

invalidEmail=Invalid e-mail address
```

Then, use the following script code:

```
import javax.mail.internet.InternetAddress

import org.cyclos.impl.utils.validation.validations.EmailValidation
import org.cyclos.model.ValidationException
import org.springframework.mail.javamail.MimeMessageHelper

def sender = mailHandler.mailSender
def message = sender.createMimeMessage()
def helper = new MimeMessageHelper(message)

if (!EmailValidation.isValid(formParameters.email)) {
    throw new ValidationException(scriptParameters.invalidEmail);
}

helper.to = new InternetAddress(scriptParameters.to)
helper.from = new InternetAddress(scriptParameters.from)
helper.subject = scriptParameters.subject
helper.text = """
${scriptParameters.mailHeader}
${scriptParameters.mailFrom} ${formParameters.from}
${scriptParameters.mailEmail} ${formParameters.email}
${scriptParameters.mailSubject} ${formParameters.subject}
${scriptParameters.mailMessage} ${formParameters.message}
"""
sender.send message

return scriptParameters.message
```

The custom operation needs form parameters with the following internal names: "from", "email", "subject" and "message".

Generating an account number for all accounts which doesn't have a number yet

If the account number (a feature new to Cyclos 4.4) is enabled, existing accounts will not have numbers automatically assigned. However, a custom operation can be created and executed a single time, assigning a number to all accounts (even system accounts) which don't have a number yet. To accomplish this, create a custom operation script with the following code:

```
import static org.cyclos.impl.utils.QueryHelper.processBatch

import org.cyclos.entities.banking.QAccount

def a = QAccount.account
def accounts = entityManagerHandler
```

```

        .from(a)
        .where(a.number.isNull())
        .iterate(a)

    int affected = 0
    processBatch(entityManagerHandler, accounts) { account ->
        def number = accountService.generateNumber(account.type, account.owner)
        account.number = number
        affected++
    }

    return "Generated the account number for ${affected} accounts"

```

Returning a string (notification / rich / plain text) and external redirect

Examples of a custom operation which returning a text (a notification in that case) can be found in the [loan solution example](#). An example of an external redirect is the [PayPal integration example](#).

Returning a file

This is an example where the user selects a document to download. It is assumed that the custom operation has a form field of type single selection with internal name file. Then, each possible value should have the internal name corresponding to a pdf file in a given folder. Once the user chooses the file, it is downloaded.

```

import org.cyclos.model.ValidationException

// Assume there is a pdf file for each possible value of the field
String fileName = formParameters.file.internalName
String dir = scriptParameters.dir ?: "/usr/share/documents"
File file = new File(dir, "${fileName}.pdf")
if (!file.exists()) {
    throw new ValidationException("File not found")
}
return [
    content: file,
    contentType: "application/pdf",
    name: file.name,
    length: file.length(),
    lastModified: file.lastModified()
]

```

View users I've traded with

In this example, a user can see the other users he has traded with (either performed or received payments). The custom operation needs to have user scope and result type result page. Also it needs to have the URL action as Cyclos location, and the location needs to be 'user_profile'. Finally, set as URL parameters the value 'id' (without quotes). For more details, see the next section.

```

import org.cyclos.impl.banking.AccountServiceLocal

```

```

import org.cyclos.model.ValidationException
import org.springframework.jdbc.core.ColumnMapRowMapper
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate

AccountServiceLocal accountService = binding.accountService

List<Long> accountIds = accountService.list(user).collect {acc -> acc.id}
if (accountIds.empty) {
    throw new ValidationException("No accounts")
}

NamedParameterJdbcTemplate jdbc = binding.namedParameterJdbcTemplate

// First count the number of users / currencies that traded with the current user
Integer totalCount = null
if (!skipTotalCount) {
    totalCount = jdbc.queryForObject("""
        select count(*)
        from (
            select distinct user_id, currency_id
            from (
                select user_id, currency_id
                from (
                    select u.id as user_id, at.currency_id, max(t.date) as last_date,
max(t.amount) as max_amount, count(*) as count
                    from transfers t inner join accounts a on t.to_id = a.id
                    inner join users u on a.user_id = u.id
                    inner join account_types at on a.account_type_id = at.id
                    where t.from_id in (:accountIds)
                    group by u.id, at.currency_id
                    union
                    select u.id as user_id, at.currency_id, max(t.date) as last_date,
max(t.amount) as max_amount, count(*) as count
                    from transfers t inner join accounts a on t.from_id = a.id
                    inner join users u on a.user_id = u.id
                    inner join account_types at on a.account_type_id = at.id
                    where t.to_id in (:accountIds)
                    group by u.id, at.currency_id
                ) t1
            group by user_id, currency_id
        ) t2
    ) t3
    """, [accountIds: accountIds], Integer);
}

// Then get the data
int pageSize = binding.pageSize
int currentPage = binding.currentPage
def rows = jdbc.query("""
    select u.id, u.display_for_managers, t.currency_id as "currencyId", t.last_date
as "lastDate", t.max_amount as "maxAmount", t.count
    from (
        select user_id, currency_id, max(last_date) as last_date, max(max_amount) as
max_amount, sum(count) as count
        from (
            select u.id as user_id, at.currency_id, max(t.date) as last_date, max(t.amount) as
max_amount, count(*) as count
            from transfers t inner join accounts a on t.to_id = a.id

```

```

        inner join users u on a.user_id = u.id
        inner join account_types at on a.account_type_id = at.id
        where t.from_id in (:accountIds)
        group by u.id, at.currency_id
        union
        select u.id as user_id, at.currency_id, max(t.date) as last_date, max(t.amount) as
max_amount, count(*) as count
        from transfers t inner join accounts a on t.from_id = a.id
        inner join users u on a.user_id = u.id
        inner join account_types at on a.account_type_id = at.id
        where t.to_id in (:accountIds)
        group by u.id, at.currency_id
    ) t1
    group by user_id, currency_id
) t inner join users u on t.user_id = u.id
order by t.count desc, u.display_for_managers
limit :limit
offset :offset
"""', [accountIds: accountIds, limit: pageSize + 1, offset: pageSize * currentPage], new
ColumnMapRowMapper());

// Build the result
return [
    columns: [
        [header: "User", property: "display_for_managers", width: "40%"],
        [header: "Last date", property: "lastDate", align: "center", type: "date",
width: "20%"],
        [header: "Max amount", property: "maxAmount", currencyProperty: "currencyId",
align: "right", width: "20%"],
        [header: "Transactions", property: "count", align: "right", type: "number",
width: "20%"],
    ],
    rows: rows,
    totalCount: totalCount
]

```

Search currently running background tasks

In this example, an administrator can search for background tasks which are scheduled to run. In regular usage, there shouldn't have many background tasks running at a given time. However, some operations such as indexing entities on Elasticsearch or charging account fees will generate many tasks. The custom operation needs to have system scope and result type result page.

```

import org.apache.commons.lang3.StringUtils
import org.cyclos.entities.utils.QBackgroundTaskExecution

def e = QBackgroundTaskExecution.backgroundTaskExecution
def page = entityManagerHandler.from(e)
    .orderBy(e.id.asc())
    .page(currentPage, pageSize, skipTotalCount, e)
return [
    columns: [
        [header: "Execution started at", property: "submittedAt"],
        [header: "Task", property: "className"],
    ],

```

```

    [header: "Context", property: "context"],
  ],
  rows: page.pageItems.collect {
    [
      submittedAt: it.submittedAt,
      className: StringUtils.substringAfterLast(it.className, '.'),
      context: it.context
    ]
  },
  totalCount: page.totalCount,
  hasNextPage: page.hasNextPage
]

```

Get easy invoice link / QR-Code

This example presents users a link and QR-code which can be shared for other users to pay him / her (easy invoice). The QR-code can be scanned by the Cyclos mobile application from the payer. To use it, you will need the following content in the script parameters box:

```

## The message shown above
message=You can copy and share the following easy invoice link or QR-code, \
which can be scanned by the Cyclos mobile application:

## Currency to be appended to the URL.
## Not needed if users have a single currency.
currency=unit

## Payment type to be appended to the URL.
## Not needed if users have a single payment type.
paymentType=user.tradeTransfer

```

Then, use the following script code:

```

import org.apache.commons.text.StringEscapeUtils
import org.cyclos.utils.StringHelper

def rootUrl = sessionData.configuration.fullUrl

// Get the amount
def amount = formParameters.amount.toPlainString()

// Get the description
def description = formParameters.description

// Get the to user his username
def to = user.username

def parameters = "&amount=${amount}"
if (StringHelper.isNotBlank(description)) {
  description = StringHelper.encodeURIComponent(description)
  description = StringHelper.replace(description, "+", "%20")
  parameters += "&description=${description}"
}
if (StringHelper.isNotBlank(scriptParameters.currency)) {
  parameters += "&currency=${scriptParameters.currency}"
}

```

```

if (StringHelper.isNotBlank(scriptParameters.paymentType)) {
    parameters += "&type=${scriptParameters.paymentType}"
}

def url = "${rootUrl}/pay/?to=${to}${parameters}"
def qrCode = "${rootUrl}/api/tickets/easy-invoice-qr-code/*:${to}?size=medium${parameters}"

// Return the result
return """
<p>${scriptParameters.message}</p>
<p>
    <br>
    <a href="${url}" target="_blank">${StringEscapeUtils.escapeHtml4(url)}</a>
</p>
<p style="text-align:center">
    <br>
    
</p>
"""

```

Then create the custom operation:

- Name: Easy invoice
- Script: Select the Get easy invoice link / QR-Code script
- Scope: user
- Result type: Rich text

Finally, after saving, add the following form parameters:

- Amount
 - Internal name: amount
 - Data type: Decimal
 - Decimal digits: 2 (adjust according to the currency)
 - Required: Yes
- Description
 - Internal name: description
 - Data type: Multi-line text
 - Required: No

Loan request (content page with action)

This example shows some input fields for users to request a loan. Then shows the loan details and an action for the user to send the loan application. When clicked an e-mail is sent with the request data, so the administration can actually handle that loan. As both scripts calculate the loan, another script of type library is used. It contains a parameter for the interest rate. So, first is the code for the "Loan application" library script:

```

import java.math.RoundingMode

import org.cyclos.entities.users.User

import groovy.xml.MarkupBuilder

/**
 * Calculates the installment amount by composite monthly interests
 */
def installmentAmount(double rate, double totalAmount, int installments) {
    rate /= 100.0
    double cf = rate / (1 - (1 / Math.pow(1 + rate, installments)))
    return new BigDecimal(totalAmount * cf).setScale(2, RoundingMode.HALF_UP)
}

/**
 * Returns an HTML with the loan request details
 */
String loanRequestHTML(double rate, double reqAmount, int installments, User user) {
    def instAmount = installmentAmount(rate, reqAmount, installments)
    def totalAmount = instAmount * installments
    def out = new StringWriter()
    MarkupBuilder html = new MarkupBuilder(out)
    html.div {
        table {
            if (user != null) {
                tr {
                    td width:"200px", { b "Requested by user" }
                    td "${user.name} (${user.username})"
                }
            }
            tr {
                td width:"200px", { b "Monthly interest rates" }
                td "${formatter.format(rate as BigDecimal)}% per month"
            }
            tr {
                td { b "Requested amount" }
                td formatter.format(reqAmount, 2)
            }
            tr {
                td { b "Total amount to be repaid" }
                td formatter.format(totalAmount as BigDecimal, 2)
            }
            tr {
                td colspan: 2, { b "Installments" }
            }
            tr {
                td style:"text-align:center", { b "Due date" }
                td style:"text-align:right", { b "Due amount" }
            }
            def cal = Calendar.getInstance()
            for (int i = 0; i < installments; i++) {
                cal.add(Calendar.MONTH, 1)
                tr {
                    td style:"text-align:center", { b formatter.formatAsDate(cal.time) }
                    td style:"text-align:right", formatter.format(instAmount, 2)
                }
            }
        }
    }
}

```

```

    }
  }
}
return out.toString()
}

```

This script needs a parameter which is the interest rate. So, paste this in the script parameters field:

```

monthlyInterests=0.75
email=admin@admin-email.com

```

Here is the code for the custom operation that requests the loan. Don't forget to include the loan application library in the script.

```

def rate = scriptParameters.monthlyInterests as double
def reqAmount = formParameters.amount as BigDecimal
def instCount = formParameters.installments as int

return [
  title: "Loan request details",
  content: loanRequestHTML(rate, reqAmount, instCount, null),
  actions: [
    submitLoanApplication: [
      parameters: [
        user: user.id
      ]
    ]
  ]
]

```

And here is the code for the custom operation that submits the loan request. The script should also include the loan application library.

```

import javax.mail.internet.InternetAddress

import org.springframework.mail.javamail.MimeMessageHelper

def rate = scriptParameters.monthlyInterests as double
def reqAmount = formParameters.amount as BigDecimal
def instCount = formParameters.installments
def user = formParameters.user
def body = loanRequestHTML(rate, reqAmount, instCount, user)

def sender = mailHandler.mailSender
def message = sender.createMimeMessage()
def helper = new MimeMessageHelper(message, true, "UTF-8")

helper.to = new InternetAddress(scriptParameters.email)
helper.from = new InternetAddress(user.email, user.name)
helper.subject = "Loan request"
helper.setText(body, true)
sender.send message

return "The loan request was sent to the administration"

```


Before creating the custom operation for the loan application itself, create the one for the action, with the following fields:

- Name: Send loan application
- Internal name: submitLoanApplication
- Label: Send
- Script: Select the one with the code to send the application
- Main menu: Banking
- Scope: internal
- Result type: notification

Then, after saving, add the following form parameters:

- Total amount
 - Internal name: amount
 - Data type: Decimal
 - Decimal digits: 2
 - Required: Yes
- Number of installments
 - Internal name: installments
 - Data type: Integer
 - Required: Yes
- User
 - Internal name: user
 - Data type: Linked entity
 - Linked entity type: User
 - Required: Yes

Then create the custom operation with the loan application form:

- Name: Loan application
- Internal name: loanApplication
- Script: Select the loan application request script
- Scope: user
- Result type: Rich text

Then, after saving, add the following form parameters:

- Total amount
 - Internal name: amount
 - Data type: Decimal
 - Decimal digits: 2
 - Required: Yes
- Number of installments
 - Internal name: installments
 - Data type: Integer
 - Required: Yes

And add another action in the actions tab:

- Total amount: Map to this operation's Total amount
- Number of installments: Map to this operation's Number of installments
- User: Set it for the script to define the value

After granting permission to the Loan application custom operation, it should appear in the menu

Possibilities for custom operations that return a result page

Custom operations that return a page of results are very versatile. For example, they can be printed as PDF or exported to CSV, or page results (if the script returns the total count).

Also, on the custom operation form it is possible to define an action to be executed when a row is clicked by the user. The possible actions are:

- Navigate to an external URL: When clicking a row, the user is redirected to an external URL.
- Navigate to a location in Cyclos: A list of common locations in Cyclos are presented.
- Run an internal custom operation: Allows running a custom operation which has the scope = 'Internal'. This new operation will probably present some content to the user.

In all cases an action is set to a row, parameters can be passed to the next page. This is very important, as will provide context on which data was selected. For an internal custom operation to receive a parameter, first on the result page custom operation the field 'URL parameters' must be set, having a comma-separated value of object properties to be passed to the internal custom operation. This will pass all such properties from the clicked row to the internal custom operation. Then, the internal custom operation needs to have form fields defined with the matching internal name. The following is an example script for a custom operation which lists fictional external records. It needs to have as URL action the custom operation presented ahead to show an external record details, and pass the URL parameter 'recordId' (without quotes):

```

return [
  columns: [
    [header:"Name", property:"name"]
  ],
  rows: [
    [name: "Record 1", recordId: 1],
    [name: "Record 2", recordId: 2],
    [name: "Record 3", recordId: 3],
    [name: "Record 4", recordId: 4],
    [name: "Record 5", recordId: 5],
    [name: "Invalid Record", recordId: 99999],
  ]
]

```

Then another custom operation, which should be defined as internal, and have a form field which internal name 'recordId' (without quotes):

```

import org.cyclos.model.EntityNotFoundException

// Validate the id
def recordId = formParameters.recordId
def validIds = 1..50
if (!(recordId in validIds)) {
  throw new EntityNotFoundException([
    entityType: "External record",
    key: recordId as String])
}

return [
  title: "Details for record ${recordId}",
  content: "This is the description for record ${recordId}"
]

```

Running custom operations on bulk actions

Custom operations of scope bulk action are executed once per user affected by the bulk action. If an operation has to be executed over a set of users, it can be convenient to create a bulk action for that.

In order to run a bulk action that runs a custom operation, first the script needs to be created. Then the bulk action, with its (optional) form parameters. Finally, the administrator needs permission to manage bulk actions and also permission to run that custom operation over users.

Here there are two examples:

Perform payment

This is an example of a custom operation that performs a payment from a system account to each user. The script checks if the user has the account that receives the payment. If not, it is marked as skipped for that bulk action. If the user has the account, the payment is performed.

The script needs as parameters, the internal name of the system account and payment type, like this (make sure to check that the internal names are correct):

```
systemAccount=debit
paymentType=toUser
```

Then the custom operation script block should be as follow:

```
import org.cyclos.entities.banking.TransferType
import org.cyclos.model.EntityNotFoundException
import org.cyclos.model.banking.accounts.SystemAccountOwner
import org.cyclos.model.banking.transactions.PerformPaymentDTO
import org.cyclos.model.banking.transfertypes.TransferTypeVO
import org.cyclos.model.users.bulkactions.BulkActionUserStatus

def tt = entityManagerHandler.find(TransferType,
    "${scriptParameters.systemAccount}.${scriptParameters.paymentType}")

// Check if the user has the destination account type
try {
    accountService.load(user, tt.to)
} catch (EntityNotFoundException e) {
    return BulkActionUserStatus.SKIPPED
}

// Perform the payment
def dto = new PerformPaymentDTO()
dto.from = SystemAccountOwner.instance()
dto.to = user
dto.type = new TransferTypeVO(tt.id)
dto.amount = formParameters.amount
paymentService.perform(dto)
return BulkActionUserStatus.SUCCESS
```

Remove canceled tokens

In this example a bulk action can be used to remove canceled tokens optionally filtering by type. If a type is selected, only the canceled tokens of that type will be removed, otherwise all canceled tokens will be removed.

First create the script to load token principal types:

- Script name: Load token types (Can be changed)
- Run with all permissions: Yes
- Parameters: (Leave empty)
- Script code:

```
import org.cyclos.model.system.fields.DynamicFieldValueVO

def values = []
principalTypeService.listUserTokenPermissions(null).each{
    values.add(new DynamicFieldValueVO(it.getType().getInternalName(), it.getType().getName()))
}
```

```
}  
return values
```

And the script for the custom operation:

- Script name: Remove canceled tokens (Can be changed)
- Run with all permissions: Yes
- Parameters: (Leave empty)
- Script code:

```
import org.cyclos.model.access.principaltypes.TokenPrincipalTypeVO  
import org.cyclos.model.access.tokens.TokenQuery  
import org.cyclos.model.access.tokens.TokenStatus  
import org.cyclos.model.users.users.UserVO  
import org.cyclos.model.utils.ModelHelper  
import org.cyclos.utils.CollectionHelper  
  
def query = new TokenQuery()  
query.setUnlimited()  
query.setUser(conversionHandler.convert(UserVO.class, user))  
query.setStatuses(CollectionHelper.asSet(TokenStatus.CANCELED))  
if (formParameters.tokenType) {  
    query.setType(ModelHelper.voFromString(TokenPrincipalTypeVO.class,  
        formParameters.tokenType.value))  
}  
  
tokenService.search(query).getPageItems().each{ tokenService.remove(it.getId()) }  
  
return "Tokens removed successfully"
```

Then create the custom operation:

- Name: Remove canceled tokens (Can be changed)
- Scope: Bulk action
- Script: Remove canceled tokens (The script created above)
- Show form: Always

Finally, after saving, add the following form parameter:

- Display name: Token type
- Internal name: tokenType
- Data type: Dynamic selection
- Load values script: Load token types (The first script created)
- Field type: Dropdown
- Required: No (Set it in "Yes" if you don't want to let the user who runs the bulk action remove tokens of all types at once)

Custom wizards

These scripts are invoked when a user runs a custom wizard. A custom wizard can be of the following types:

- **Registration:** Replaces the registration form. Gives the opportunity to present custom fields defined in the wizard itself, allowing more data to be collected and processed by script. Besides creating the script and the wizard, in the Configuration menu, the wizards should be set for registration on large screens (desktops), medium screens (tablets) and small screens (phones). When all 3 are set, the regular registration form / API is disabled;
- **User:** The wizard is executed by users via a menu item. Can also be set for administrators and brokers to run over other users via the profile;
- **System:** The wizard is executed by administrators via a menu item;
- **Guest:** The wizard is executed by guests. Can be shown in a menu via the menu entries in content management.

A wizard is comprised of several steps, which are manually ordered. When the wizard starts, by default the first step is shown. On each transition, by default the next step is executed, until the last step. After finishing the wizard, a result is shown. The script can control which is the first step, and which are the possible transitions between steps. The transitions are determined before the step is shown, because each possible transition is displayed as a different button to users. When the script doesn't return any transitions, the default is to use a single transition to the next step in the defined order.

Bound variables:

- **wizard:** The [org.cyclos.entities.system.CustomWizard](#) being executed.
- **execution:** The [org.cyclos.entities.system.CustomWizardExecution](#).
- **step:** The current [org.cyclos.entities.system.CustomWizardStep](#). On transition, is the next step.
- **previousStep:** The previous [org.cyclos.entities.system.CustomWizardStep](#). Only present on transitions.
- **transition:** The transition id the user has selected. Only present on transitions.
- **user:** The [org.cyclos.entities.users.User](#). The meaning depends on the wizard type. For registration wizards is only present in the finish function, and is the newly registered user. For user wizards is the user over which it is being executed.
- **steps:** The list of all steps, together with the possible transitions executed so far (including the current step. Is a list of [org.cyclos.entities.system.CustomWizardStep](#).
- **storage:** The [org.cyclos.impl.system.CustomWizardExecutionStorage](#). Can be used by scripts to store / retrieve data at any moment of the execution.

- registration: The [org.cyclos.model.users.users.PublicRegistrationDTO](#) which is filled-in on each step. Only present for registration wizards. To persist any modification, assign it back to the storage (storage.registration = registration).
- customValues: A map keyed by custom field internal name, whose values are the values filled-in during the execution. Do not confuse it with custom profile fields, which are stored in the registration object. To persist any modification, assign it back to the storage (storage.customValues = customValues).
- returnUrl: The URL to pass to the external system on external redirects. Is the URL to which the external system should redirect the user when returning to Cyclos.
- request: The [org.cyclos.model.utils.RequestInfo](#). Only present on the script that runs the callback after an external redirect. Contains the information about the current request, so the script function which handles the callback can identify the context to complete the process.

Code blocks:

There are several code blocks on custom wizard scripts. The expected return type depends on the code block:

- Script code executed when the wizard finishes: This is the only required code block. The script is executed after finishing the last step. For registration wizards, the script is executed after the user has been registered, so additional actions can be performed on that user, and the result is ignored. For other wizards, this is the action executed on finish, and the script can return one of the following:
 - A plain string. The result content, which is handled either as plain text or HTML depending on the wizard configuration;
 - An object containing the properties title and result.
- Script code executed when a new execution starts: This code is executed whenever a new execution starts. The result is handled as the initial step, and can be one of the following:
 - A plain string, interpreted as the internal name of the first step. The only transition will be the next step in the defined order;
 - A [org.cyclos.entities.system.CustomWizardStep](#). The only transition will be the next step in the defined order;
 - An object (or map) containing the following properties:
 - step: The step, interpreted as described above;
 - transitions: A collection or single value interpreted as transitions. A transition can either be a string or step, interpreted as step, or an object with the following properties:
 - id: The transition id. Will be the one clients need to pass in to the transition operation.

- **label:** The transition label. Is the text shown to users in the execution page for this transition.
- **step:** A string or step to which the transition leads to.

When no step is returned, the first step is assumed. When no transitions are returned, it is assumed a transition with id 'next' which goes to the following step after the first one.

- **Script code executed on transitions between steps:** This code is executed whenever an execution is transitioned between steps. As the previous and next steps are known (in variables 'previousStep' and 'step', respectively), the script determines which are the possible transition from the next step to the subsequent steps. The result can be one or a collection of transitions, which are either strings (interpreted as step internal name) or objects / maps containing the following properties:

- **id:** The transition id. Will be the one clients need to pass in to the transition operation.
- **label:** The transition label. Is the text shown to users in the execution page for this transition.
- **step:** A string or step to which the transition leads to.

When nothing is returned, a single transition 'next' will be used, pointing to the next step.

- **Script code executed before the user is redirected to an external site:** This code is executed when the user confirms a step which is configured as external redirect. It is used to interact with an external system during the wizard. For example, a top-up could be required during the registration process. The result is a URL to which the user will be redirected. Please, note the returnUrl bound variable, which is the URL to pass to the external system, indicating where the external system should redirect the user back to Cyclos.
- **Script code executed when the external site redirects the user back to Cyclos:** This code is executed after the external redirect completes. The result can be one of the following:
 - **null or true:** The execution will automatically transition to the next step in the defined order;
 - **false:** Is interpreted as canceling the external redirect action. The execution will stay in the current step.
 - **string:** Is handled as the identifier of the transition for the next step.

Tips:

- Use the 'storage' object to store and retrieve custom data on any step of the current execution. The storage also provides access to specialized data within the execution. See [the class JavaDoc](#) for more details.
- The script can send notifications, which are displayed in the current step. For that, use either 'storage.info', 'storage.warn' or 'storage.error' with the message. Once a transition happens, any pending notifications are cleared.

- If the wizard has a step that redirect to an external site, that step cannot be the last one. That is because after the redirect back to Cyclos, the wizard will transition automatically to the next step. The wizard execution page will have to query the current status. If the execution would have ended, all the context would have been lost for that execution.

Examples

Registration with a required top-up

This example requires a top-up via PayPal for the public user registration. It uses the same PayPal library from PayPal Integration. Make sure you have the library code updated.

The example uses 3 script blocks for the wizard script, plus script parameters. So, set the following in the script:

- Parameters:

```
# Settings for the access token record type
auth.recordType = paypalAuth
auth.clientId = clientId
auth.clientSecret = clientSecret
auth.token = token
auth.tokenExpiration = tokenExpiration

# Settings for PayPal
mode = sandbox
currency = EUR
paymentDescription = Initial top-up

# Settings for the Cyclos payment
amountMultiplier = 1
accountType = debitUnits
paymentType = paypalCredits

# Messages
error.invalidRequest = Invalid request
error.transactionNotFound = Transaction not found
error.transactionAlreadyApproved = The transaction was already approved
error.payment = There was an error while processing the top-up. Please, try again.
error.notApproved = The top-up was not approved
message.canceled = The top-up was canceled
message.done = The top-up was approved
```

- Script code executed when the wizard finishes:

```
import org.cyclos.entities.users.User
import org.cyclos.impl.system.CustomWizardExecutionStorage
import org.cyclos.impl.system.ScriptHelper
import org.cyclos.model.ValidationException

import groovy.transform.TypeChecked

@TypeChecked
def performPayments() {
```

```

def variables = binding.variables as Map<String, Object>
def scriptParameters = variables.scriptParameters as Map<String, String>
def service = new PayPalService(variables)
def storage = variables.storage as CustomWizardExecutionStorage
def scriptHelper = variables.scriptHelper as ScriptHelper
def user = variables.user as User
def orderId = storage.getString('paypalOrderId')

// If no order id, return an error
if (orderId == null) {
    throw new ValidationException('No PayPal payment data')
}

def order = service.getOrderFromPayPal(orderId)
if (order.status == "APPROVED") {
    // Add a commit listener to perform the payments,
    // it will be executed after a successful registration
    scriptHelper.addOnCommitTransactional({
        // Execute the PayPal payment
        def capturedOrder = service.captureOrder(orderId)
        try {
            // Try to perform the payment in Cyclos,
            // if fails, refund the payment in PayPal
            service.perform(capturedOrder, user)
        } catch (Exception ex) {
            service.refundCapturedOrder(capturedOrder, null, user)
        }
    })
} else {
    throw new ValidationException(scriptParameters.'error.notApproved'
        ?: "The payment was not approved")
}
}

performPayments()

```

- Script code executed before the user is redirected to an external site:

```

import org.cyclos.impl.system.CustomWizardExecutionStorage

import groovy.transform.TypeChecked

@TypeChecked
def createOrder(){
    def variables = binding.variables
    def service = new PayPalService(variables)
    def storage = variables.storage as CustomWizardExecutionStorage

    def customValues = variables.customValues as Map<String, Object>
    def amount = customValues.amount as Number
    def returnUrl = variables.returnUrl as String

    def order = service.createOrder(amount, returnUrl)
    def link = (order.links as Map<String, Object>[])
        .find {it.rel == "approve"}
    if (link) {
        // Store the returned order id
    }
}

```

```

        storage.setString('paypalOrderId', order.id as String)
        return link.href
    } else {
        throw new IllegalStateException("No approval url returned from PayPal")
    }
}

createOrder()

```

- Script code executed when the external site redirects the user back to Cyclos:

```

import org.cyclos.impl.system.CustomWizardExecutionStorage
import org.cyclos.model.ValidationException
import org.cyclos.model.utils.RequestInfo

import groovy.transform.TypeChecked

@TypeChecked
def paypalCallback() {
    def variables = binding.variables as Map<String, Object>
    def scriptParameters = variables.scriptParameters as Map<String, String>
    def storage = variables.storage as CustomWizardExecutionStorage
    def request = variables.request as RequestInfo

    if (request.getParameter('cancel')) {
        // The operation has been canceled. Don't transition
        storage.warn(scriptParameters.'message.canceled'
            ?: 'The top-up was canceled')
        return false
    }
    // If no order id, return an error
    if (storage.getString('paypalOrderId') == null) {
        throw new ValidationException('Invalid request')
    }
}

paypalCallback()

```

Then, in your registration wizard, create a custom field with internal name 'amount', of type 'Decimal', and required. Assign that field to a wizard step, which cannot be the last one.

Custom web services

These scripts are invoked when a request is received in some path under <cyclos-root-url>[/network]/run/**. To actually run them, it is needed to create a custom web service definition in the "System - Tools - Custom web services" menu.

The custom web services have the following important properties:

- The accepted HTTP methods: GET, POST or Both;
- Whether the script will be executed as guest (optionally using a fixed HTTP username / password, with basic authorization) or as an authenticated user, like with other web services, using the same headers described in ???;

- An IP address whitelist, to control which hosts can call the custom web service;
- The URL mappings, which is a list of paths (one per line) to be matched after the `<cyclos-root-url>[/network]/run` root path. It is possible to specify the following types of paths:
 - Simple paths. For example, 'users', matches '`<cyclos-root-url>[/network]/run/users`'
 - Nested paths. For example, 'users/list', matches '`<cyclos-root-url>[/network]/run/users/list`'
 - Wildcards. For example, 'users/*', matches '`<cyclos-root-url>[/network]/run/users/a`', but not '`<cyclos-root-url>[/network]/run/users/a/b`'
 - Nested wildcards. For example, 'users/**', matches '`<cyclos-root-url>[/network]/run/users/a/b/c`'
 - Path variables. For example, 'users/{groupId}/{userId}', matches '`<cyclos-root-url>[/network]/run/users/123/78`', and a map with {groupId:123,userId:78} is available to the script

Bound variables:

- customWebService: The [org.cyclos.entities.system.CustomWebService](#).
- request: The [org.cyclos.model.utils.RequestInfo](#) representing the incoming request.
- path: A string containing the path part after the `<cyclos-root-url>[/network]` prefix. Is neither initiated or terminated with / (slash)
- pathVariables: A [org.cyclos.utils.ParameterStorage](#) representing the path variables. Will be filled if the URL mapping contains {var} definitions, and contains the actually matched values

Return value: The script may return one of the following data:

- A [org.cyclos.model.utils.ResponseInfo](#), allowing to totally customize the response
- Null. In this case, the response will have status code 200 and no body.
- A string. In this case, the response will have status code 200, Content-type: text/plain, and the returned string as body
- An arbitrary object / collection. In this case, the response will have status code 200, Content-type: application/json, and the body will contain a JSON representation of the returned object

If the script captures an error and wants to customize the response, instead of silencing the exception in a catch clause and returning a [org.cyclos.model.utils.ResponseInfo](#), which will cause the current transaction to commit, possibly leaving the database in an inconsistent state, the script should throw a [org.cyclos.model.utils.ResponseException](#), which contains a ResponseInfo internally. This way the main transaction is rolled back. Other exceptions than

ResponseExceptions are returned as HTTP status codes other than 200, and the details are returned as JSON, in the same way as ???.

Sometimes it is useful to extend the Cyclos API to clients, like doing specific payments, or running a series of operations in a single request. However, it is important to use the same permissions as the user would normally have, to prevent security breaches. To do so, 3 steps are needed:

- Make sure the script uses the security layer: Whenever using a service, use the security layer instead of the direct service implementation. For example, to use the UserService, instead of using the userService bound variable, use userServiceSecurity instead.
- Ensure the custom web service has user authentication: On the custom web service details page, ensure it runs as user, not as guest.
- On the script, make sure it runs with the user permissions: On the details page of the script used by the custom web service, make sure the checkbox called Run with all permissions is unchecked. This guarantees the script will run with the exact permissions as the user

Examples

Perform a payment

This example allows a caller to quickly perform a payment between 2 users. It is assumed that the URL mapping is something like payment/{from}/{to}/{amount} and there is a single possible payment type between the 2 users.

```
import org.cyclos.model.banking.transactions.PerformPaymentDTO
import org.cyclos.model.users.users.UserLocatorVO

def pmt = new PerformPaymentDTO()
pmt.from = new UserLocatorVO(principal: pathVariables.from)
pmt.to = new UserLocatorVO(principal: pathVariables.to)
pmt.amount = pathVariables.getDecimal('amount')

// Perform the payment and return the complete PaymentVO
return paymentService.perform(pmt)
```

Single-sign-on (login users without their passwords)

With this example it is possible to login a user (create a session) without their password. This is useful when Cyclos works as a single-sign-on, with the user authenticated by some other system. Just be extra careful with the external security which will be employed, such as creating an IP address whitelist, a guest user / password, etc on the custom web service, otherwise, anyone could impersonate any user.

The script receives 2 query parameters: 'user', which is the login name (or some other identification, such as e-mail) of the user to be logged in, and 'remoteAddress', which is the remote IP address of the client accessing the third party software.

The script code is the following:

```
import org.cyclos.impl.access.SessionDataFactory
import org.cyclos.impl.access.SessionHandler
import org.cyclos.impl.access.SessionHandler.CreateSessionParameters
import org.cyclos.model.access.RequestData
import org.cyclos.model.access.channels.BuiltInChannel
import org.cyclos.model.users.users.UserLocatorVO
import org.cyclos.utils.StringHelper

def principal = request.parameters.user
def remoteAddress = request.parameters.remoteAddress
def user = userLocatorHandler.locate(new UserLocatorVO(principal: principal))
def requestData = new RequestData()
requestData.copyPropertiesTo(requestData)
if (StringHelper.isNotBlank(remoteAddress)) {
    requestData.remoteAddress = remoteAddress
}
def runAs = SessionDataFactory.direct(user)
    .requestData(requestData)
    .channel(BuiltInChannel.MAIN)
    .build()
def session = sessionHandler.create(new CreateSessionParameters(runAs))
return session.sessionToken
```

Then create a custom web service, select that script and set the URL mapping to something like login. When performing a request to <cyclos-root>/run/login?user=consumer1&remoteAddress=183.165.12.7, a session will be created for that user, and the session token will be returned. It is then possible to redirect the client to <cyclos-root>/?Session-Token=<returned-session-token> and the user will be logged-in to Cyclos.

Service interceptors

These scripts are invoked before and / or after specific service operations. The services are those that extend [org.cyclos.services.Service](#), not the REST api. The REST services use the internal services, so, ultimately, they can be intercepted too.

In order to apply these kind of scripts, a service interceptor needs to be created, and among its properties, the following can be highlighted:

- Which service(s) are intercepted;
- Which operation(s) are intercepted;
- Which script is executed;
- Whether the interceptor is enabled or not.

Multiple service interceptors may apply over the same operation. Hence, the order is important. For this reason interceptors are manually ordered.

Interceptors run in the same database transaction as the regular service operation. Each operation defines whether the transaction is read-write or read-only. Operations that just

read data run in a read-only transaction. In that case, attempting to write data in the database will fail. Also, even if the transaction is read-write, in the script that runs after the operation, it might happen that an error was thrown, marking the transaction as rollback. As such, service interceptor scripts should be very careful when writing to the database. If this is needed, it is recommended to do it in another database transaction, running after the original transaction ends. The [ScriptHelper](#) class (which is bound to the script context on the scriptHelper variable) provides the `addOnCommitTransactional` and `addOnRollbackTransactional` methods which allow running a closure after the main transaction ends either as commit or rollback). Those methods run the code block itself inside another transaction, in which it is safe to write to database.

There is a shared context for all interceptors, of type [org.cyclos.impl.system.ServiceInterceptorContext](#). This context can be used to replace parameters before the original operation invocation, or even to skip the invocation altogether and return a value determined by the script. Also, the context can be used to store attributes which will be shared amongst interceptors or between the code that runs before and after the service invocation itself. The `propertyMissing` mechanism from Groovy is supported by the context implementation. For example, `context.myVariable = 'x'` will set the attribute `myVariable`.

Bound variables:

- interceptor: The [org.cyclos.entities.system.ServiceInterceptor](#);
- service: The java class representing the service interface being intercepted;
- operation: The java method representing the service operation being intercepted;
- context: The [org.cyclos.impl.system.ServiceInterceptorContext](#).

The return value from the script, in both codes that run before or after, is ignored.

Recovering from errors in crucial services

If there is an error in the service interceptor script, and it is applied to crucial services, such as login or the application configuration, it may render the network unusable. In order to recover from it, it is possible to go to the global mode (`<cyclos_root_url>/global`), go to the network details and click on "Disable service interceptors". It will disable all service interceptors for that network, allowing the regular usage again. After fixing the scripts, any interceptors need to be manually enabled again.

Examples

Modifying the general transfers overview default filters

This example will set the default filters on transfer overview to not include chargebacks, neither transfers that were charged back. A service interceptor needs to be applied on the

AccountService.getAccountHistoriesOverviewData operation. The script should have this on the code that runs after the service is executed (the code for before may be left empty):

```
import org.cyclos.model.banking.accounts.AccountHistoriesOverviewQuery
import org.cyclos.model.banking.transfers.TransferNature

if (context.success) {
    AccountHistoriesOverviewQuery query = context.result.query
    // Include all transfer natures except chargeback
    query.natures = EnumSet.complementOf(EnumSet.of(TransferNature.CHARGEBACK))
    // Also don't include transfers that were themselves charged-back
    query.chargedBack = false
}
```

Processing variables in the content of menu entries

This example processes the content of a menu entry to replace variables. The example variables are [profile fields](#) of the logged user. A service interceptor needs to be applied on the MenuEntryService.getItemsDetails operation. The script should have this on the code that runs after the service is executed (the code for before may be left empty):

```
import org.cyclos.model.contentmanagement.contentitems.MenuItemDetailedVO
import org.cyclos.utils.StringHelper

if (context.success) {
    MenuItemDetailedVO item = context.result
    if (item.content != null && sessionData.loggedIn) {
        def profileFieldVariables =
        profileFieldHandler.getProfileFieldVariables(sessionData.loggedBasicUser);
        item.content = StringHelper.replaceVariables(item.content, profileFieldVariables);
    }
}
```

Making mobile phone enabled for SMS by default on registrations by administrators or brokers

This example sets mobile phones to be enabled for SMS by default when registering a user by administrator or broker. To achieve this, create a service interceptor that captures the UserService.getDataForNew operation. The script should have this on the code that runs after the service is executed (the code for before may be left empty):

```
if (context.success) {
    def phoneData = context.result?.mobilePhoneData
    if (phoneData?.canManuallyVerify) {
        phoneData.verified = true
    }
}
```


Custom scheduled tasks

These scripts are called periodically by custom scheduled tasks. See [System – Scheduled tasks](#) for more details.

The bound variables are:

- `scheduledTask`: The [org.cyclos.entities.system.CustomScheduledTask](#) being executed
- `log`: The [org.cyclos.entities.system.CustomScheduledTaskLog](#) for this execution

Return value:

- The script should return a string, which is logged as message, and can be viewed on the application

Examples

Periodically importing a file

This example imports a file with users, which is expected to be located at a given directory in the file system. For other import types, it is just a matter of using distinct [org.cyclos.model.system.imports.ImportedFileDTO](#) subclasses (some require setting some parameter, like in the example, the group for users). The scheduled task just triggers the import. From that point, the import is processed on the background, and the status can be monitored on System - Tools - Imports menu.

To use it, you will need the following content in the script parameters box (either in script itself or in the custom scheduled task's script parameters):

```
filename=/tmp/imports/users.csv
group=consumers
```

Then use the following code in the script box:

```
import org.cyclos.model.system.imports.UserImportedFileDTO
import org.cyclos.model.users.groups.GroupVO
import org.cyclos.model.utils.FileSizeUnit
import org.cyclos.server.utils.SerializableInputStream

// Resolve the users filename and the group
String filename = scriptParameters['filename']
String groupInternalName = scriptParameters['group']

// Download the file to a local temp file
File file = new File(filename)
if (!file.exists()) {
    return "The expected file, ${filename}, doesn't exist"
}
if (file.length() == 0) {
    return "The file ${filename} is empty"
}
```

```

// Caution! the SerializableInputStream automatically deletes the file
// when closed, except when calling, except when calling .file()
def stream = new SerializableInputStream(file)
stream.file()

// Import
UserImportedFileDTO dto = new UserImportedFileDTO()
dto.fileName = filename
// It is important to mark the file as automatic import,
// otherwise manual interaction would be required for processing
dto.processAutomatically = true
dto.group = new GroupVO([internalName: groupInternalName])
importService.upload(dto, stream)

// Build a result string
def fileSize = FileSizeUnit.nearestFileSize(file.length())
return "Started import of ${filename}. File size is ${fileSize}"

```

Periodically update a static HTML page

In this example, every time the scheduled task runs, a static HTML file is updated. In the file, it is written the total number of users and the balances of each system account.

```

import org.cyclos.entities.users.QUser
import org.cyclos.model.banking.accounts.AccountWithStatusVO
import org.cyclos.model.banking.accounts.SystemAccountOwner
import org.cyclos.model.users.groups.BasicGroupNature
import org.cyclos.model.users.users.UserStatus

import groovy.xml.MarkupBuilder

def now = new Date()

QUser u = QUser.user
int users = entityManagerHandler
    .from(u)
    .where(u.status.notIn(UserStatus.REMOVED, UserStatus.PURGED),
        u.group.nature.eq(BasicGroupNature.USER_GROUP))
    .count()
List<AccountWithStatusVO> accounts = accountService.
    getAccountsSummary(SystemAccountOwner.instance(), null)

File out = new File("/var/www/html/summary.html")

def sessionData = binding.sessionData
def formatter = binding.formatter
MarkupBuilder builder = new MarkupBuilder(new FileWriter(out))
builder.html {
    head {
        title "${sessionData.configuration.applicationName} summary"
        meta charset: "UTF-8"
    }
    body {
        p {
            b "Total users"

```

```

        span ": ${users}"
      }
      accounts.each { a ->
        p {
          b a.type.name
          span " balance: ${formatter.format(a.status.balance)}"
        }
      }
      br()
      br()
      br()
      p style: "font-size: small", "Last updated: ${formatter.format(now)}"
    }
  }
  return "File ${out.absolutePath} updated"
}

```

Custom SMS operations

These scripts are invoked when a user executes a custom sms operation, as configured in the sms channel in the configuration. The function should implement the logic for that operation.

Bound variables:

- configuration: The [org.cyclos.entities.system.CustomSmsOperationConfiguration](#). With it, it is possible to navigate up to the [org.cyclos.entities.system.SmsChannelConfiguration](#).
- phone: The [org.cyclos.entities.users.MobilePhone](#)
- sms: The [org.cyclos.impl.utils.sms.InboundSmsData](#), containing the operation alias and the operation parameters
- parameterProcessor: The [org.cyclos.impl.utils.sms.SmsParameterProcessor](#), which is a helper class to obtain operation parameters as specific data types

There are no expected return values for this script.

Examples

Pay taxi with an SMS message

In this example SMS operation, users can pay taxi drivers via SMS. Make sure all the following are configured:

- In the script details, the checkbox "Run with all permissions" is disabled;
- There should be a single single transfer type enabled for the SMS operations channel, and the user performing the operation needs to have permission to perform that payment;
- A custom profile field with internal name taxild of type single line text, and marked as unique needs to be enabled for the product of taxi owners;
- a user identification method of type custom field, called "Taxi id" with the taxild field needs to be created. Make sure its internal name is also taxild;

- In the configuration details, in the channels tab, enable SMS operations. Then, in that channel, make sure "Taxi id" is allowed as user identification method to perform payments
- Still in the same channel configuration page, create a new SMS operation of type Custom, selecting the alias "taxi" and the selected script.

Then, customers can perform the payment by sending an sms in the format: taxi <taxi id> <amount>. Below is the script that should be used:

```
import org.cyclos.model.ValidationException
import org.cyclos.model.banking.TransferException
import org.cyclos.model.banking.transactions.PerformPaymentDTO
import org.cyclos.model.banking.transactions.PerformPaymentData
import org.cyclos.model.messaging.sms.OutboundSmsType
import org.cyclos.model.users.users.UserLocatorVO

// Read the parameters
String taxiId = parameterProcessor.nextString("taxiId")
BigDecimal amount = parameterProcessor.nextDecimal("amount")

// Find the user by taxi id
def locator = new UserLocatorVO(
    principalType: "taxiId",
    principal: taxiId)

// Find the payment type
PerformPaymentData data = transactionService.getPaymentData(
    phone.user, locator)
if (data.paymentTypes?.empty) {
    throw new ValidationException("No possible payment types")
}

// Perform the payment
def pmt = new PerformPaymentDTO()
pmt.amount = amount
pmt.from = data.from
pmt.to = data.to
pmt.type = data.paymentTypes[0]
try {
    vo = paymentServiceSecurity.perform(pmt)
    outboundSmsHandler.send(phone,
        "The payment was successful",
        OutboundSmsType.SMS_OPERATION_RESPONSE)
    // Also notify the taxi, for example, by connecting to the
    // taxi company system, which notifies the taxi driver...
} catch (TransferException e) {
    outboundSmsHandler.send(phone,
        "The payment couldn't be performed",
        OutboundSmsType.SMS_OPERATION_RESPONSE)
}
```

Inbound SMS handling

These scripts are invoked when a gateway sends SMS messages to Cyclos. There are two functions in this script: one to generate the gateway response and another one to resolve

basic SMS data from an inbound HTTP request. Both functions are optional, defaulting to the normal behavior (when not using a script).

The common bound variables are:

- configuration: The [org.cyclos.impl.system.ConfigurationAccessor](#) for the inbound SMS
- channelConfiguration: The [org.cyclos.entities.system.SmsChannelConfiguration](#)

The functions are:

- Resolve basic SMS data: Function used to read an inbound sms request and return an object containing the phone number, the SMS message and the splitted SMS message into parts. Only the phone number and SMS message are required. If the message parts are empty, it will be assumed the message will be split by spaces.
 - Bound variables:
 - request: The [org.cyclos.model.utils.RequestInfo](#)
 - Return value:
 - An [org.cyclos.impl.utils.sms.InboundSmsBasicData](#) instance, or a compatible Object or Map
 - If null is returned, falls back to the default processing
- Generate gateway response: Function used to determine the HTTP status code, headers and body to be returned to the SMS gateway. It can be called either when the bare minimum parameters – mobile phone number and sms message – were not sent by the gateway or when the gateway has sent a valid SMS. Keep in mind that if an operation has resulted in error, from a gateway perspective, the SMS was still delivered correctly, and the response should be a successful one. Maybe when the bare minimum parameters weren't send, the script could choose to return a different message. When no code is given, the default processing will be done, returning the HTTP status code 200 with "OK" in the body.
 - Bound variables:
 - request: The [org.cyclos.model.utils.RequestInfo](#) Only present if the inbound SMS was valid (there was a phone number and sms message)
 - inboundSmsData: The [org.cyclos.impl.utils.sms.InboundSmsData](#), which contains the operation alias and parameters
 - inboundSms: The [org.cyclos.entities.messaging.InboundSms](#), which is a log of the incoming message
 - inboundSmsResponseType: The [org.cyclos.impl.utils.sms.InboundSmsResponseType](#), which is the type of response according to the operation execution
 - inboundSmsException: The exception that cause the operation to fail

- Return value:
 - An [org.cyclos.model.utils.ResponseInfo](#) instance, or a compatible Object or Map
 - If null is returned, falls back to the default processing

Examples

Receiving a SMS in JSON format

This example assumes the request body is a JSON object:

```
import java.nio.charset.StandardCharsets

import org.cyclos.impl.utils.sms.InboundSmsBasicData

def body = new InputStreamReader(request.body, StandardCharsets.UTF_8)
def json = objectMapper.readTree(body)

InboundSmsBasicData result = new InboundSmsBasicData()
result.phoneNumber = json.get("phoneNumber")?.asText()
result.message = json.get("message")?.asText()
return result
```

Receiving a SMS with a custom format

This example reads the phone number from a request header, and the message from the request body:

```
import org.apache.commons.io.IOUtils
import org.cyclos.impl.utils.sms.InboundSmsBasicData

// Read the phone from a header, and the message from the body
InboundSmsBasicData result = new InboundSmsBasicData()
result.phoneNumber = request.headers."phone-number"
result.message = IOUtils.toString(request.body, "UTF-8")
return result
```

Outbound SMS handling

These scripts are invoked to send SMS messages. By default, Cyclos connects to gateways via HTTP POST / GET, which can be set in the configuration. However, the sending can be customized (or totally replaced) via a script. As in most cases the custom sending just wants to customize some aspects of the sending, not all, it is possible that the script just creates a subclass of [org.cyclos.impl.utils.sms.GatewaySmsSender](#), customizing some aspects of it (for example, by overriding the buildRequest method and adding some headers, or the resolveVariables method to have some additional variables which can be sent in the POST body).

Bound variables:

- configuration: The [org.cyclos.impl.system.ConfigurationAccessor](#)
- phone: The [org.cyclos.entities.users.MobilePhone](#). May be null, if is a reply to an unregistered user.
- phoneNumber: The international phone number, in the [E.164](#) standard string. Never null.
- message: The SMS message to send

Return value:

- An [org.cyclos.model.messaging.sms.OutboundSmsStatus](#) enum value
- A string which represents the exact name of an [OutboundSmsStatus](#) enum value
- If null is returned, it is assumed a sending success

Examples

Sending SMS requests as JSON

This example posts the SMS message as JSON to the gateway, and awaits the response before returning the status:

```
import static groovyx.net.http.ContentType.*
import static groovyx.net.http.Method.*

import java.util.concurrent.CountDownLatch

import org.cyclos.model.messaging.sms.OutboundSmsStatus

import groovyx.net.http.HTTPBuilder

// Read some gateway data from the configuration
def smsConfig = configuration.outboundSmsConfiguration
def url = smsConfig.gatewayUrl
def user = smsConfig.username
def pwd = smsConfig.password

// Send the POST request
def http = new HTTPBuilder(url)
if (user) {
    // Maybe send the username and password
    def auth = user;
    if (pwd) {
        auth += ":{pwd}"
    }
    http.headers["Authorization"] = "Basic ${auth.bytes.encodeBase64()}"
}
http.headers["Content-Type"] = "application/json; charset=UTF-8"
CountDownLatch latch = new CountDownLatch(1)
def error = false
http.request(POST, JSON) {
    body = [
        to: phoneNumber,
        text: message
    ]
}
```

```

    }

    response.success = { resp, result ->
        latch.countDown()
    }

    response.failure = { resp ->
        error = true
        latch.countDown()
    }
}

//Await for the response
latch.await()
return error ? OutboundSmsStatus.SUCCESS : OutboundSmsStatus.UNKNOWN_ERROR

```

Sending SMS requests as XML

This example posts the SMS message as XML to the gateway, and awaits the response before returning the status:

```

import static groovyx.net.http.ContentType.*
import static groovyx.net.http.Method.*

import java.util.concurrent.CountDownLatch

import org.cyclos.model.messaging.sms.OutboundSmsStatus

import groovyx.net.http.HTTPBuilder

// Read some gateway data from the configuration
def smsConfig = configuration.outboundSmsConfiguration
def url = smsConfig.gatewayUrl
def user = smsConfig.username
def pwd = smsConfig.password

// Send the POST request
def http = new HTTPBuilder(url)
if (user) {
    // Maybe send the username and password
    def auth = user
    if (pwd) {
        auth += ":{pwd}"
    }
    http.headers["Authorization"] = "Basic ${auth.bytes.encodeBase64()}"
}
http.headers["Content-Type"] = "application/xml; charset=UTF-8"
CountDownLatch latch = new CountDownLatch(1)
def error = false
http.request(POST, XML) {
    // Pass the body as a closure - parsed as XML
    body = {
        "sms-message" {
            "destination-phone" phoneNumber
            text message
        }
    }
}

```



```

    }

    response.success = { resp, xml ->
        latch.countDown()
    }

    response.failure = { resp ->
        error = true
        latch.countDown()
    }
}

//Await for the response
latch.await()
return error ? OutboundSmsStatus.UNKNOWN_ERROR : OutboundSmsStatus.SUCCESS

```

Link generation

These scripts are used to generate links (URLs) which are used to point users to specific functionality. Some systems have a custom front-end for users, which means that when they receive e-mails with links, instead of pointing the links to the default Cyclos page, it should point to the custom front-end page.

Whenever the script returns null, the default link to Cyclos is generated, so the script may handle specific users / groups, and fallback to the default for other users by returning null.

The script code has the following variables bound (besides the [default bindings](#)):

- type: The [org.cyclos.impl.utils.LinkType](#) instance, which is an enumeration defining which link type is being generated;
- user: The [org.cyclos.entities.users.User](#) which will receive the link. May be null depending on the link type;
- urlFilePart: The URL part which is used by the default link in Cyclos. Kept mostly for backwards compatibility, because if the default is desired, the script should return null.

For links used on notifications (type is NOTIFICATION), the following additional bound variables are used:

- location: The notification location, as [org.cyclos.model.utils.Location](#);
- entityId: The identifier of the entity related to the notification;
- entityIdParam: The parameter name to pass the entity identifier.

For links used to verify the e-mail, which are: registration validation (type is REGISTRATION_VALIDATION), e-mail change validation (type is EMAIL_CHANGE) and forgot password request (type is FORGOT_PASSWORD), the following additional bound variables are used:

- validationKey: The key which is sent by e-mail to validate the action.

For generating the URL which is used as callback for custom operations of type external redirect (type is EXTERNAL_REDIRECT), the following additional bound variables are used:

- execution, externalRedirectExecution: The context for the external redirect, as [org.cyclos.entities.system.ExternalRedirectExecution](#). This class contains both the 'id' and 'verificationToken' which are used to resume the custom operation after the external redirect is performed, and hence, should be appended to the generated URL.

For generating the URL which is used as callback for custom wizards that perform an external redirect (type is WIZARD_EXTERNAL_REDIRECT), the following additional bound variables are used:

- execution: The context for the wizard execution, as [org.cyclos.entities.system.CustomWizardExecution](#). This class contains the 'key' to resume the wizard execution after the external redirect is performed, and hence, should be appended to the generated URL.
- storage: The storage for wizard execution, as [org.cyclos.impl.system.CustomWizardExecutionStorage](#). It can be used to lookup the custom fields which were filled so far, as well as the registration parameters, if the wizard is of type registration.

For generating the URL to pay a specific ticket (type is TICKET), the following additional bound variables are used:

- ticket: The ticket to be paid, as [org.cyclos.entities.banking.Ticket](#). This class contains the 'ticketNumber' which is used to pay the ticket, and hence, should be appended to the generated URL.

For generating the URL to pay an easy invoice (type is EASY_INVOICE), the following additional bound variables are used, besides the already mentioned 'user' which is the easy invoice destination (all additional bound variables are optional):

- paymentTo: The easy invoice payee, as [org.cyclos.impl.users.LocateUserResult](#). Generally its 'principal' property should be used to generate the parameter value;
- paymentAmount: The easy invoice amount, as BigDecimal;
- paymentType: The easy invoice payment type, as [org.cyclos.entities.banking.PaymentTransferType](#);
- paymentDescription: The easy invoice description, as string;
- paymentCustomValues: Values for custom fields on the easy invoice. Is a map of strings, keyed by [org.cyclos.entities.banking.TransactionCustomField](#).

For generating a link to a redirect to a mobile application page (type is MOBILE_REDIRECT), or a URL with a custom schema (cyclos://) for the mobile application to open directly (type is MOBILE), the following additional bound variables are used:

- `mobileUrlFilePart`: The URL part of the mobile page to open.

Examples

Link generation depending on the location

This examples returns distinct values according to location.

```
import org.cyclos.impl.utils.LinkType
import org.cyclos.model.utils.Location

def root = 'https://mydomain.com'

// This example only handles notification for a few locations
if (type != LinkType.NOTIFICATION) {
  return null
}

switch (location) {
  case Location.EXTERNAL_PAYMENT:
    return root + '/external-payment/' + entityId
  case Location.TRANSFER:
    return root + '/transfer/' + entityId
  default:
    return null
}
```

Export formats

These scripts are invoked when exporting data to a file in a custom format.

There are several contexts which can be exported:

- Account history
- Transfers overview
- Transfer details
- Transactions search (such as scheduled payments search);
- Transactions overview (such as payment requests overview);
- Transaction details (such as payment, scheduled payment or payment request);
- Users search
- User balances overview
- Account limits overview
- Records search (for system or specific user, of a given type)
- Records overview (as administrator or broker, of a given type)
- Shared fields records search

- Tokens search (such as cards)
- Vouchers search
- Voucher details
- Custom operation results (when returning a result page)

The following variables are bound on the script execution:

- format: The [org.cyclos.entities.system.ExportFormat](#)
- context: The [org.cyclos.model.system.exportformats.ExportFormatContext](#) representing the context being exported.
- query: When exporting a search results, is the [org.cyclos.model.QueryParameters](#) with the search filters.
- data: Depending on the context, is either an iterator or the object being exported:
 - ACCOUNT_HISTORY: An iterator of [org.cyclos.impl.banking.AccountHistoryEntry](#);
 - TRANSFERS_OVERVIEW: An iterator of [org.cyclos.entities.banking.Transfer](#);
 - TRANSFER_DETAILS: A single [org.cyclos.entities.banking.Transfer](#);
 - TRANSACTIONS_SEARCH and TRANSACTIONS_OVERVIEW: An iterator of [org.cyclos.entities.banking.Transaction](#);
 - TRANSACTION_DETAILS: A single [org.cyclos.entities.banking.Transaction](#);
 - USERS_SEARCH: An iterator of [org.cyclos.entities.users.User](#);
 - USERS_WITH_BALANCES: An iterator of [org.cyclos.model.banking.accounts.UserWithBalanceVO](#);
 - BALANCE_LIMITS_OVERVIEW: An iterator of [org.cyclos.entities.banking.UserAccount](#);
 - RECORDS_SEARCH, RECORDS_OVERVIEW and SHARED_FIELDS_RECORDS_SEARCH: An iterator of [org.cyclos.entities.users.Record](#);
 - TOKENS_SEARCH: An iterator of [org.cyclos.entities.access.Token](#);
 - VOUCHERS_SEARCH: An iterator of [org.cyclos.entities.banking.Voucher](#);
 - VOUCHER_DETAILS: A single [org.cyclos.entities.banking.Voucher](#);
 - CUSTOM_OPERATION: An iterator of arrays of strings;
- additionalData: Is a map whose values depends on the context.

The script must return one of these types, containing the file content:

- java.io.InputStream;
- byte[];
- java.io.Reader;

- java.io.File;
- If not any of the above, will be handled as string (calling toString()).

Examples

Exporting the account history as Swift MT940 format

This script allows exporting the account history entries in the [MT940](#), which is used by some accounting software for importing / exporting transactions. To accomplish this, create a script of type Export format with the following code:

```
import java.text.SimpleDateFormat

import org.cyclos.entities.banking.Account
import org.cyclos.entities.users.User
import org.cyclos.impl.banking.AccountHistoryEntry
import org.cyclos.model.banking.accounts.AccountHistoryQuery
import org.cyclos.utils.StringHelper

def timeZone = sessionData.configuration.timeZone
def dateFormat = new SimpleDateFormat("yyMMdd")
dateFormat.timeZone = timeZone
def entryDateFormat = new SimpleDateFormat("yyMMdd")
entryDateFormat.timeZone = timeZone

def formatAmount(BigDecimal amount) {
    return amount.abs().toString().replace('.', ',')
}

def formatSignal(BigDecimal amount) {
    return amount.compareTo(BigDecimal.ZERO) > 0 ? 'C' : 'D'
}

def formatOwner(Account account) {
    String text
    if (account.owner instanceof User) {
        text = account.owner.username
    } else {
        text = account.type.internalName ?: account.type.name
    }
    return formatText(text)
}

def formatDescription(AccountHistoryEntry entry) {
    def description = entry.transaction?.description ?: entry.type.valueForEmptyDescription
    return formatText(description)
}

def formatText(text) {
    // First replace line breaks or multiple spaces by a single space, trimming to 60 chars
    text = (text ?: '').replaceAll("[\n\r]+", " ")
    text = text.replaceAll("\\s+", " ")
    text = StringHelper.trim(StringHelper.truncate(text, 60))
    // Second make sure that no special characters are used
    text = StringHelper.asciiOnly(StringHelper.unaccent(text))
}
```

```

    // Finally make sure that no colon character is used, this might mess up the mt940 file
    return text.replaceAll('\\:', ' ')
}

// Get the account
AccountHistoryQuery query = binding.query
Account account = conversionHandler.convert(Account, query.account)

// Get the begin date
Date begin = conversionHandler.toDate(query.period?.begin) ?: account.creationDate

// Get the end date
Date now = new Date()
Date end = conversionHandler.toDate(query.period?.end) ?: now
if (end.after(now)) {
    end = now
}

// Get the balance at begin / end
def balanceBegin = accountService.getBalance(account, begin)
def balanceEnd = accountService.getBalance(account, end)
def currency = scriptParameters.currencyCode

// Write the header
StringBuilder out = new StringBuilder("":20:CN${dateFormat.format(end)}
:25:${scriptParameters.iban}
:28:000
:60F:
${formatSignal(balanceBegin)}${dateFormat.format(begin)}${currency}${formatAmount(balanceBegin)}
""")

// Process each entry
scriptHelper.processBatch(data) { AccountHistoryEntry entry ->
    def date = entryDateFormat.format(entry.date)
    def amount = formatAmount(entry.amount)
    def signal = formatSignal(entry.amount)
    def fromTo = formatOwner(entry.relatedAccount)
    def description = formatDescription(entry)
    out << ":61:${date}${signal}${amount}NOV NONREF\n"
    out << ":86:${fromTo} > ${description}\n"
}

// Write the footer
out << ":62F:
${formatSignal(balanceEnd)}${dateFormat.format(end)}${currency}${formatAmount(balanceEnd)}"

// Return the output content
return out

```

Also set the following in the script parameters box:

```

# The currency code that will be exported on the file
currencyCode = EUR
# The IBAN account number that will be exported in the file
iban = NL70TRIO0123456789

```

Then, create a new export format in System > System configuration > Export formats, with the following fields:

- Name: MT940 (change as desired)
- Internal name: mt940
- Content type: application/octet-stream
- Binary: No
- Character encoding: UTF-8
- File extension: mt940
- Contexts: Account history
- Script: Select the previously created script.

Notifications

These scripts are invoked before generating the notification to be stored in the database. Later, the notifications will be sent through a polling task.

The following variables are bound on the script execution:

- type: The notification type (never null). It could be an admin notification or an user notification. To know the list of all available types please check the following links:
 - For Administrators: [org.cyclos.model.messaging.notifications.AdminNotificationType](#)
 - For members:
 - [org.cyclos.model.messaging.notifications.PersonalNotificationType](#)
 - [org.cyclos.model.messaging.notifications.AccountNotificationType](#)
- entity: The entity associated with the notification (never null).
This parameter should be used to retrieve all values used to create the notification content (body, title, etc). For the type of entity associated to each notification type please check the links to the notification types above;
- user: The user to be notified (never null). In case of member / administrator it will be an instance of [org.cyclos.entities.users.User](#), otherwise, it will be an [org.cyclos.entities.users.Operator](#).

The script can return null or an empty map if no customizations must be applied for the given input parameters, otherwise the returned map can contain any of the following:

- title: (String) the title of the notification
- body: (String) the body of the notification
- sms: (String) the sms message to be send

- fcm: (Map) the customizations for the push notifications (sent through Firebase Cloud Messaging) with any of the following:
 - title: (String) the title of the push notification. If not given then the title above will be used (if any)
 - body: (String) the body of the push notification. If not given then the body above will be used (if any)
 - imageUrl: (String) the url of the image associated to the push notification. If not given, the user's profile image is used (only for users, not operators)
 - iosBadge: (Boolean) flag indicating if a badge must be shown for iOS notifications. Default is true;
 - androidIconColor: the color in #rrggbb format (e.g: #23AB34) used to colorize the small notification icon shown in Android devices. By default no color is sent, it depends on the Android version;
 - data: (Map), any value in this map will be set to the message as a data field (key-value pair). The key or value may not be null.
Those fields only have sense for the mobile application, i.e they are useful only if there is a compatible application that can process it (added here only for completeness).

Examples

Include the balance for a "Payment received" notification

This script will add the user balance to the body of the notification generated for the type: [PAYMENT_RECEIVED](#)

To accomplish this, create a script of type Notification with the following code:

```
import org.cyclos.model.messaging.notifications.AccountNotificationType
import org.cyclos.utils.StringHelper

def received = type == AccountNotificationType.PAYMENT_RECEIVED
def performed = type == AccountNotificationType.ALL_NON_SMS_PERFORMED_PAYMENTS
if (received || performed) {
  def account = received ? entity.to : entity.from
  def balance = formatter.format(entity.currency,
    accountService.getBalance(account, null))
  def amount = formatter.format(entity.currencyAmount)
  def owner = formatter.format(received ? entity.fromOwner : entity.toOwner)
  def ownerShort = StringHelper.truncate(owner, 30)
  if (received) {
    return [
      body: "You have received a payment of $amount from $owner."
        + " Your new balance is: $balance.",
      sms: "Payment of $amount received from $ownerShort."
        + " New balance: $balance."
    ]
  } else {
```



```

    return [
        body: "You have performed a payment of $amount to $owner."
        + " Your new balance is: $balance.",
        sms: "Payment of $amount performed to $ownerShort."
        + " New balance: $balance."
    ]
}

// default values for the rest of the notification types
return null

```

Finally, you must set the script in the corresponding configuration (Notifications section).

4.4. Solutions using scripts

Examples of solutions that require a single script can be found directly in the specific script description page (links directly above). Solutions that need several scripts and configurations can be found in this section.

PayPal Integration

It is possible to integrate Cyclos with [PayPal](#), allowing users to buy units with their PayPal account. This is done with a custom operation which allows users to confirm the payment in PayPal and then, once the payment is confirmed, a payment from a system account is performed to the corresponding user account, automating the process of buying units. However, keep in mind the rates charged by PayPal, which vary according to some conditions.

To do so, first you'll need a PayPal premium or business account (for testing – using PayPal sandbox – any account is enough). You'll need to go to the [PayPal Developer page](#) to create an application on >REST API apps>, and get the client id and secret.

Then several configurations are required in Cyclos. Scripts can only be created as global administrators switched to a network, so it is advised to use a global admin to perform the configuration. Carefully follow each of the following steps:

Check the root URL

Make sure that the configuration for users use a correct root url. In System > System configuration > Configurations, select the configuration set for users and make sure the Main URL field points to the correct external URL. It will be used to generate the links which will be sent to PayPal redirect users back to Cyclos after confirming / canceling the operation.

Enable transaction number in currency

This can be checked under System > Currencies select the currency used for this operation, mark the Enable transfer number option and fill in the required parameters.

Create a system record type to store the client id and secret

Under System > System configuration > Record types, create a new system record type, with the following characteristics:

- Name: PayPal Authentication
- Internal name: paypalAuth
- Display style: Single form
- Main menu: System

For this record type, create the following fields:

- Client ID
 - Internal name: clientId
 - Data type: Single line text
 - Required: yes
- Client Secret
 - Internal name: clientSecret
 - Data type: Single line text
 - Required: yes
- Token
 - Internal name: token
 - Data type: Single line text
 - Required: no
- Token expiration
 - Internal name: tokenExpiration
 - Data type: Date
 - Required: no

Create a user record type to store each payment information

Under System > System configuration > Record types, create a new user record type, with the following characteristics:

- Name: PayPal payment
- Internal name: paypalPayment
- Display style: List
- Main menu: Banking

- User management section: Banking

For this record type, create the following fields:

- Payment ID
 - Internal name: paymentId
 - Data type: Single line text
 - Required: no
- Amount
 - Internal name: amount
 - Data type: Decimal
 - Required: no
- Transaction
 - Internal name: transaction
 - Data type: Linked entity
 - Linked entity type: Transaction
 - Required: no

Create the library script

Under System > Tools > Scripts, create a new library script, with the following characteristics:

- Name: PayPal
- Type: Library
- Included libraries: none
- Parameters:

```
# Settings for the access token record type
auth.recordType = paypalAuth
auth.clientId = clientId
auth.clientSecret = clientSecret
auth.token = token
auth.tokenExpiration = tokenExpiration

# Settings for the payment record type
payment.recordType = paypalPayment
payment.paymentId = paymentId
payment.amount = amount
payment.transaction = transaction

# Settings for PayPal
mode = sandbox
currency = EUR
paymentDescription = Buy Cyclos units
```

```

# Settings for the Cyclos payment
amountMultiplier = 1
accountType = debitUnits
paymentType = paypalCredits

# Messages
error.invalidRequest = Invalid request
error.transactionNotFound = Transaction not found
error.transactionAlreadyApproved = The transaction was already approved
error.payment = There was an error while processing the payment. Please, try again.
error.notApproved = The payment was not approved
message.canceled = You have cancelled the operation.\nFeel free to start again if needed.
message.done = You have successfully completed the payment. Thank you.

```

- Script code:

```

import static groovyx.net.http.ContentType.*
import static groovyx.net.http.Method.*

import java.util.concurrent.CountDownLatch

import org.apache.commons.codec.binary.Base64
import org.cyclos.entities.banking.PaymentTransferType
import org.cyclos.entities.banking.SystemAccountType
import org.cyclos.entities.users.RecordCustomField
import org.cyclos.entities.users.SystemRecord
import org.cyclos.entities.users.SystemRecordType
import org.cyclos.entities.users.User
import org.cyclos.entities.users.UserRecord
import org.cyclos.entities.users.UserRecordType
import org.cyclos.impl.banking.PaymentServiceLocal
import org.cyclos.impl.messaging.AlertServiceLocal
import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.users.RecordServiceLocal
import org.cyclos.impl.utils.persistence.EntityManagerHandler
import org.cyclos.model.EntityNotFoundException
import org.cyclos.model.banking.accounts.SystemAccountOwner
import org.cyclos.model.banking.transactions.PaymentVO
import org.cyclos.model.banking.transactions.PerformPaymentDTO
import org.cyclos.model.banking.transfertypes.TransferTypeVO
import org.cyclos.model.messaging.alerts.SystemAlertType
import org.cyclos.model.users.records.RecordDataParams
import org.cyclos.model.users.records.UserRecordDTO
import org.cyclos.model.users.recordtypes.RecordTypeVO
import org.cyclos.model.users.users.UserLocatorVO
import org.cyclos.utils.ParameterStorage

import groovy.transform.TypeChecked
import groovy.transform.TypeCheckingMode
import groovyx.net.http.HTTPBuilder
import groovyx.net.http.Method

/**
 * Class used to store / retrieve the authentication information for PayPal
 * A system record type is used, with the following fields: client id (string),
 * client secret (string), access token (string) and token expiration (date)

```

```

*/
@TypeChecked
class PayPalAuth {
    String recordTypeName
    String clientIdName
    String clientSecretName
    String tokenName
    String tokenExpirationName

    SystemRecordType recordType
    SystemRecord record
    Map<String, Object> wrapped

    public PayPalAuth(Map<String, Object> variables) {
        def params = variables.scriptParameters as Map<String, Object>
        recordTypeName = params.'auth.recordType' ?: 'paypalAuth'
        clientIdName = params.'auth.clientId' ?: 'clientId'
        clientSecretName = params.'auth.clientSecret' ?: 'clientSecret'
        tokenName = params.'auth.token' ?: 'token'
        tokenExpirationName = params.'auth.tokenExpiration' ?: 'tokenExpiration'

        // Read the record type and the parameters for field internal names
        recordType = (variables.entityManagerHandler as EntityManagerHandler)
            .find(SystemRecordType, recordTypeName)

        // Should return the existing instance, of a single form type.
        // Otherwise it would be an error
        def dataParams =
            new RecordDataParams(recordType: new RecordTypeVO(id: recordType.id))
        record = (variables.recordService as RecordServiceLocal)
            .newEntity(dataParams) as SystemRecord
        if (!record.persistent) throw new IllegalStateException(
            "No instance of system record ${recordType.name} was found")

        wrapped = (variables.scriptHelper as ScriptHelper).wrap(record, recordType.fields)
    }

    public String getClientId() {
        wrapped[clientIdName]
    }
    public String getClientSecret() {
        wrapped[clientSecretName]
    }
    public String getToken() {
        wrapped[tokenName]
    }
    public Date getTokenExpiration() {
        wrapped[tokenExpirationName] as Date
    }
    public void setClientId(String clientId) {
        wrapped[clientIdName] = clientId
    }
    public void setClientSecret(String clientSecret) {
        wrapped[clientSecretName] = clientSecret
    }
    public void setToken(String token) {
        wrapped[tokenName] = token
    }
}

```

```

    public void setTokenExpiration(Date tokenExpiration) {
        wrapped[tokenExpirationName] = tokenExpiration
    }
}

/**
 * Class used to store / retrieve PayPal payments as user records in Cyclos
 */
@TypeChecked
class PayPalRecord {
    String recordTypeName
    String paymentIdName
    String amountName
    String transactionName

    UserRecordType recordType
    Map<String, RecordCustomField> fields

    private EntityManagerHandler entityManagerHandler
    private RecordServiceLocal recordService
    private ScriptHelper scriptHelper

    public PayPalRecord(Map<String, Object> variables) {

        def params = variables.scriptParameters as Map<String, Object>
        recordTypeName = params.'payment.recordType' ?: 'paypalPayment'
        paymentIdName = params.'payment.paymentId' ?: 'paymentId'
        amountName = params.'payment.amount' ?: 'amount'
        transactionName = params.'payment.transaction' ?: 'transaction'

        entityManagerHandler = variables.entityManagerHandler as EntityManagerHandler
        recordService = variables.recordService as RecordServiceLocal
        scriptHelper = variables.scriptHelper as ScriptHelper
        recordType = entityManagerHandler.find(UserRecordType, recordTypeName)
        fields = [:]
        recordType.fields.each {f -> fields[f.internalName] = f}
    }

    /**
     * Creates a payment record, for the given user and JSON,
     * as returned from PayPal's create payment REST method
     */
    public UserRecord create(User user, Number amount) {
        RecordDataParams newParams = new RecordDataParams(
            [user: new UserLocatorVO(id: user.id),
             recordType: new RecordTypeVO(id: recordType.id)])
        def dto = recordService.getDataForNew(newParams).getDto() as UserRecordDTO
        def wrapped = scriptHelper.wrap(dto, recordType.fields)
        wrapped[amountName] = amount

        // Save the record DTO and return the entity
        Long id = recordService.save(dto)
        return entityManagerHandler.find(UserRecord, id)
    }

    /**
     * Finds the record by id
     */

```

```

public UserRecord find(Long id) {
    try {
        UserRecord userRecord = entityManagerHandler.find(UserRecord, id)
        if (userRecord.type != recordType) {
            return null
        }
        return userRecord
    } catch (EntityNotFoundException e) {
        return null
    }
}

/**
 * Removes the given record, but only if it is of the
 * expected type and hasn't been confirmed
 */
public void remove(UserRecord userRecord) {
    if (userRecord.type != recordType) {
        return
    }
    Map<String, Object> wrapped = scriptHelper
        .wrap(userRecord, recordType.fields)
    if (wrapped[transactionName] != null) return
    entityManagerHandler.remove(userRecord)
}

/**
 * Class used to interact with PayPal services
 */
@TypeChecked
class PayPalService {
    String mode
    String baseUrl
    String currency
    String paymentDescription

    String accountTypeName
    String paymentTypeName
    double multiplier

    SystemAccountType accountType
    PaymentTransferType paymentType
    PayPalAuth auth
    PayPalRecord record

    private ScriptHelper scriptHelper
    private PaymentServiceLocal paymentService
    private AlertServiceLocal alertService
    private ParameterStorage storage
    private Map<String, Object> params

    PayPalService(Map<String, Object> variables) {
        this.auth = new PayPalAuth(variables)
        this.record = new PayPalRecord(variables)
        scriptHelper = variables.scriptHelper as ScriptHelper
        paymentService = variables.paymentService as PaymentServiceLocal
        alertService = variables.alertService as AlertServiceLocal
    }
}

```

```

storage = variables.parameterStorage as ParameterStorage
params = variables.scriptParameters as Map<String, Object>

mode = params.mode ?: 'sandbox'
if (mode != 'sandbox' && mode != 'live') {
    throw new IllegalArgumentException("Invalid PayPal parameter " +
        "'mode': ${mode}. Should be either sandbox or live")
}
baseUrl = mode == 'sandbox'
    ? 'https://api.sandbox.paypal.com' : 'https://api.paypal.com'

currency = params.currency
if (currency == null || currency.empty) {
    throw new IllegalArgumentException("Missing PayPal parameter 'currency'")
}

def emh = variables.entityManagerHandler as EntityManagerHandler
accountTypeName = params.accountType
if (accountTypeName == null || accountTypeName.empty)
    throw new IllegalArgumentException("Missing PayPal parameter 'accountType'")
paymentTypeName = params.paymentType
if (paymentTypeName == null || paymentTypeName.empty)
    throw new IllegalArgumentException("Missing PayPal parameter 'paymentType'")
accountType = emh.find(SystemAccountType, accountTypeName)
if (!accountType.currency.transactionNumber?.used) {
    throw new IllegalStateException("Currency " + accountType.currency
        + " doesn't have transaction number enabled")
}
paymentType = emh.find(PaymentTransferType, paymentTypeName, accountType)
multiplier = Double.parseDouble((params.amountMultiplier as String) ?: "1")
paymentDescription = params.paymentDescription ?: ""
}

/**
 * Creates an order in PayPal and the corresponding user record
 */
Map<String, Object> createOrder(User user, Number amount, String callbackUrl) {
    // Create the UserRecord for this payment
    UserRecord userRecord = record.create(user, amount)
    //store the record's id to retrieve it after the payment was confirmed in PayPal
    storage['recordId'] = userRecord.id

    // Create the payment in PayPal
    def json = createOrder(amount, callbackUrl)
    //store the PayPal order id to retrieve it after the payment was confirmed in
    PayPal
    storage['orderId'] = json.id

    return json
}

/**
 * Creates an order in PayPal with a given amount, without updating any record
 */
Map<String, Object> createOrder(Number amount, String callbackUrl) {
    callbackUrl += callbackUrl.contains("?") ? "&" : "?"
    String returnUrl = "${callbackUrl}success=true"
    String cancelUrl = "${callbackUrl}cancel=true"

```



```

def jsonBody = [
  intent: "CAPTURE",
  application_context: [
    return_url: returnUrl,
    cancel_url: cancelUrl,
    user_action: "PAY_NOW"
  ],
  purchase_units: [
    [
      description: paymentDescription,
      amount: [
        value: amount,
        currency_code: currency
      ]
    ]
  ]
]

// Create the payment in PayPal
return performRequest("${baseUrl}/v2/checkout/orders", jsonBody, POST)
}

/**
 * Capture the order (execute the payment in PayPal)
 */
Map<String, Object> captureOrder(String orderId) {
  return performRequest("${baseUrl}/v2/checkout/orders/${orderId}/capture", null,
POST)
}

/**
 * Get the order information from PayPal
 */
Map<String, Object> getOrderFromPayPal(String orderId) {
  return performRequest("${baseUrl}/v2/checkout/orders/${orderId}", null, GET)
}

/**
 * Executes a PayPal payment, and creates the payment in Cyclos
 */
Map<String, Object> execute(UserRecord userRecord) {
  def wrapped = scriptHelper.wrap(userRecord)
  def orderId = storage['orderId'] as String
  // Execute the payment in PayPal
  def capturedOrder = captureOrder(orderId) as Map<String, Object>
  // Update the payment id
  wrapped[record.paymentIdName] = getPaymentIdFromCapturedOrder(capturedOrder)
  def vo
  try {
    // Try to perform the payment in Cyclos, if it fails, refund the payment in
PayPal
    vo = perform(capturedOrder, userRecord.user)
  } catch (Exception ex) {
    refundCapturedOrder(capturedOrder, userRecord, userRecord.user)
    throw ex
  }
  if (vo != null) {
    // Update the record, setting the linked transaction

```

```

        wrapped[record.transactionName] = vo
        userRecord.lastModifiedDate = new Date()

    }
    return capturedOrder
}
/**
 * Refund the completed order using the refund link returned when the
 * order was captured and remove the user record if given
 */
void refundCapturedOrder(Map<String, Object> capturedOrder, UserRecord userRecord, User
user) {
    def refundLink = getRefundLinkFromCapturedOrder(capturedOrder)
    if (refundLink) {
        def refundedOrder
        try {
            // Make the refund
            refundedOrder = performRequest(refundLink, null, POST)
        } catch (Exception ex) {
            //Do nothing because an alert is going to be created
        }

        if (!refundedOrder || refundedOrder.status != "COMPLETED") {
            createRefundFailAlert(capturedOrder, user)
        } else if (userRecord) {
            record.remove(userRecord)
        }
    }
}
/**
 * Create the system alert for a failed refund
 */
private void createRefundFailAlert(Map<String, Object> capturedOrder, User user) {
    def errorMessage = ""User: ${user.username}. The PayPal payment
    (${getPaymentIdFromCapturedOrder(capturedOrder)}) was completed,
    but there was an error in Cyclos and the attempt to refund in PayPal failed.""
    alertService.create(SystemAlertType.CUSTOM, errorMessage)
}
/**
 * Performs the payment in Cyclos
 */
PaymentVO perform(Map<String, Object> capturedOrder, User subject) {
    if (getPaymentStatusFromCapturedOrder(capturedOrder) == 'COMPLETED') {
        def amount = new BigDecimal(getAmountFromCapturedOrder(capturedOrder) as
String)

        BigDecimal finalAmount = amount * multiplier
        // Perform the payment in Cyclos
        PerformPaymentDTO dto = new PerformPaymentDTO()
        dto.owner = SystemAccountOwner.instance()
        dto.subject = subject
        dto.amount = finalAmount
        dto.type = new TransferTypeVO(paymentType.id)
        return paymentService.perform(dto)
    } else {
        return null
    }
}
}

```

```

/**
 * Get the payment id of a captured order result
 */
@TypeChecked(TypeCheckingMode.SKIP)
String getPaymentIdFromCapturedOrder(capturedOrder) {
    return capturedOrder.purchase_units[0].payments.captures[0].id
}

/**
 * Get the payment status of a captured order result.
 * We must get the status from the captured payment
 * because it will be NOT completed even when the order status is completed
 */
@TypeChecked(TypeCheckingMode.SKIP)
String getPaymentStatusFromCapturedOrder(capturedOrder) {
    return capturedOrder.purchase_units[0].payments.captures[0].status
}

/**
 * Get the amount of a captured order result
 */
@TypeChecked(TypeCheckingMode.SKIP)
String getAmountFromCapturedOrder(capturedOrder) {
    return capturedOrder.purchase_units[0].payments.captures[0].amount.value
}

/**
 * Get the refund link of a captured order result
 */
@TypeChecked(TypeCheckingMode.SKIP)
String getRefundLinkFromCapturedOrder(capturedOrder) {
    return capturedOrder.purchase_units[0].payments.captures[0].links.find {it.rel
== "refund"}?.href
}

/**
 * Performs a synchronous request, posting and accepting JSON
 */
@TypeChecked(TypeCheckingMode.SKIP)
Map<String, Object> performRequest(url, jsonBody, Method method) {
    def http = new HTTPBuilder(url)
    CountDownLatch latch = new CountDownLatch(1)
    Map<String, Object> responseJson = null
    def responseError = []

    // Check if we need a new token
    if (auth.token == null || auth.tokenExpiration < new Date()) {
        refreshToken()
    }

    // Perform the request
    http.request(method, JSON) {
        headers.'Authorization' = "Bearer ${auth.token}"
        headers.'Content-Type' = "application/json"

        body = jsonBody

```

```

        response.success = { resp, json ->
            responseJson = json as Map<String, Object>
            latch.countDown()
        }

        response.failure = { resp ->
            responseError << resp.statusLine.statusCode
            responseError << resp.statusLine.reasonPhrase
            latch.countDown()
        }
    }

    latch.await()
    if (!responseError.empty) {
        throw new RuntimeException("Error making PayPal request to ${url}"
            + ", got error code ${responseError[0]}: ${responseError[1]}")
    }
    return responseJson
}

/**
 * Refreshes the access token
 */
@TypeChecked(TypeCheckingMode.SKIP)
private void refreshToken() {
    def http = new HTTPBuilder("${baseUrl}/v1/oauth2/token")

    CountDownLatch latch = new CountDownLatch(1)
    Map<String, Object> responseJson = null
    def responseError = []

    http.request(POST, JSON) {
        String auth = Base64.encodeBase64String((auth.clientId + ":"
            + auth.clientSecret).getBytes("UTF-8"))
        headers.'Accept-Language' = 'en_US'
        headers.'Authorization' = "Basic ${auth}"

        send URLENC, [
            grant_type: "client_credentials"
        ]

        response.success = { resp, json ->
            responseJson = json as Map<String, Object>
            latch.countDown()
        }

        response.failure = { resp ->
            responseError << resp.statusLine.statusCode
            responseError << resp.statusLine.reasonPhrase
            latch.countDown()
        }
    }

    latch.await()
    if (!responseError.empty) {
        throw new RuntimeException("Error getting PayPal token, " +
            "got error code ${responseError[0]}: ${responseError[1]}")
    }
}

```

```

        // Update the authentication data
        auth.token = responseJson.access_token
        auth.tokenExpiration = new Date(System.currentTimeMillis() +
            ((responseJson.expires_in as Integer) - 30) * 1000))
    }
}

```

Create the custom operation script

Under System > Tools > Scripts, create a new custom operation script, with the following characteristics:

- Name: Buy units with PayPal
- Type: Custom operation
- Run with all permissions: yes
- Included libraries: PayPal
- Parameters: leave empty
- Script code executed when the custom operation is executed:

```

import org.cyclos.entities.users.User

import groovy.transform.TypeChecked

@TypeChecked
def createPayment(){
    def variables = binding.variables
    def formParameters = variables.formParameters as Map<String, Object>
    def service = new PayPalService(variables)

    def user = variables.user as User
    def amount = formParameters.amount as Number
    def returnUrl = variables.returnUrl as String
    def result = service.createOrder(user, amount, returnUrl)

    def links = result.links as Map<String, Object>[]
    def link = links.find {it.rel == "approve"}
    if (link) {
        return link.href
    } else {
        throw new IllegalStateException("No approval url returned from PayPal")
    }
}

createPayment()

```

- Script code executed when the external site redirects the user back to Cyclos:

```

import org.cyclos.entities.users.UserRecord
import org.cyclos.impl.system.ScriptHelper
import org.cyclos.model.utils.RequestInfo

```

```

import org.cyclos.server.utils.ObjectParameterStorage

import groovy.transform.TypeChecked

@TypeChecked
def paypalCallback() {
    def variables = binding.variables as Map<String, Object>
    def scriptParameters = variables.scriptParameters as Map<String, Object>
    def service = new PayPalService(variables)
    def storage = variables.storage as ObjectParameterStorage
    def recordId = storage['recordId'] as Long
    def request = variables.request as RequestInfo
    def record = service.record

    // No record?
    if (recordId == null) {
        return "[ERROR] " +
            (scriptParameters.'error.invalidRequest' ? : "Invalid request")
    }

    // Find the corresponding record
    UserRecord userRecord = record.find(recordId)
    if (userRecord == null) {
        return "[ERROR] " +
            (scriptParameters.'error.transactionNotFound'
            ? : "Transaction not found")
    }
    def wrapped = (variables.scriptHelper as ScriptHelper).wrap(userRecord)

    if (request.getParameter("cancel")) {
        // The operation has been canceled.
        // Remove the record and send a message.
        record.remove(userRecord)
        return "[WARN]" + scriptParameters.'message.canceled'
            ? : "You have cancelled the operation.\nFeel free to start again if needed."
    } else {
        // Execute the payment
        try {
            def order = service.execute(userRecord)
            if (service.getPaymentStatusFromCapturedOrder(order) == 'COMPLETED') {
                return scriptParameters.'message.done'
                    ? : "You have successfully completed the payment. Thank you."
            } else {
                return "[ERROR] " + scriptParameters.'error.notApproved'
                    ? : "The payment was not approved"
            }
        } catch (Exception e) {
            return "[ERROR] " + scriptParameters.'error.payment'
                ? : "There was an error while processing the payment. Please, try
again."
        }
    }
}

paypalCallback()

```

Create the custom operation

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: Buy units with PayPal (can be changed – will be the label displayed on the menu)
- Enabled: yes
- Scope: user
- Script: Buy units with PayPal
- Script parameters: leave empty
- Result type: External redirect
- Has file upload: no
- Main menu: Banking
- User management section: Banking
- Information text: you can add here some text explaining the process – it will be displayed in the operation page
- Confirmation text: leave empty (can be used to show a dialog asking the user to confirm before submitting, but in this case is not needed)

For this custom operation create the following form field:

- Name: Amount
- Internal name: amount
- Data type: Decimal
- Required: yes

Configure the system account from which payments will be performed to users

Under System > Accounts configuration > Account types, choose the (normally unlimited) account from which payments will be performed to users. Then set its internal name to some meaningful name. The example configuration uses debitUnits as internal name, but it can be changed. Save the form.

Configure the payment type which will be used on payments

Still in the details page for the account type, on the Transfer types tab, create a new Payment transfer type with the following characteristics:

- Name: Units bought with PayPal (can be changed as desired)
- Internal name: paypalCredits (can be changed as desired, but this name is used in the example configuration)

- To: select the user account which will receive the payment
- Enabled: yes

Grant the administrator permissions

Under System > User configuration > Groups, select the Network administrators group. Then, in the Permissions tab:

- In System > System records, set the permissions view, create and edit for the Paypal authentication record
- In User data > User records, make the Paypal payment visible only (make sure the create, edit and remove are unchecked, as this record is not meant to be manually edited)
- Save the permissions

Setup the PayPal credentials

Click System > System records > Paypal authentication. If this menu entry is not showing up, refresh the browser page (by pressing F5) and try again. Update the Client ID and Client Secret fields exactly with the ones you got in the application you registered in the [PayPal Developer page](#). Remember that PayPal has a sandbox, which can be used to test the application, and a live environment. For now, use the sandbox credentials. The other 2 fields can be left blank. Save the record.

Once the record is properly set, if you want to remove it from the menu, you can just remove the permission to view this system record in the administrator group page.

Grant the user permissions / enable the operation

In System > User configuration > Products (permissions), select the member product for users which will run the operation.

- In the Custom operations field, make the Buy units with PayPal both enabled and allowed to run.
- In Records, enable the PayPal payment record. It can be made visible to the users themselves. If not, only admins will be able to see the records.
- Save the product. From this moment, the operation will show up for users in the banking menu.

Configuring the script parameters

In the PayPal library script, in parameters, there are several configurations which can be done. All those settings can be overridden in the custom operation's script parameters, allowing using distinct configurations for distinct operations. For example, it is possible to have distinct operations to perform payments in distinct currencies. In that case, the script parameters for each operation would define the currency again.

Here are some elements which can be configured:

- Internal names for the records used to store the credentials and payments.
- Paypal mode: the 'mode' settings can be either sandbox or live, indicating that operations are performed either in a test or in the real environment. To go live, you'll need a premium or business account in PayPal, and you need to use the live credentials (client ID and client secret) in Cyclos.
- Payment currency: the 'currency' defines the 3-letter, [ISO 4217](#) code for the currency in PayPal. Sometimes, according to country-specific laws, the currency used for payments may be limited. For example, Brazilians can only pay other Brazilians in Reais. Make sure the PayPal destination account can receive payments for the specified currency, otherwise payments or refunds will fail.
- Description for payments in PayPal: using the 'paymentDescription' setting.
- Amount multiplier: Sometimes it may be desired that the payment performed in Cyclos isn't of the exact amount of the payment in PayPal. This can normally be resolved using transfer fees, but it could also be handy to use this multiplier. If left in 1, the payment in Cyclos will have the same amount as the one in PayPal. If greater / less than 1, the payment in Cyclos will be greater / less than the one in PayPal. For example, if the multiplier is 1.05, and the PayPal payment was 100 USD, the payment in Cyclos will have the amount 105. Or, if the multiplier is 0.95 and the PayPal payment was 200 EUR, the payment in Cyclos will be of 190.
- System account from which the payment will be performed to users: the 'accountType' setting is the internal name of the system account type from which payments will be performed, as explained previously. Make sure it is exactly the same as set in the account type.
- Payment type: the 'paymentType' setting is the internal name of the payment transfer type used. Make sure it is exactly the same internal name set in the payment type that was created in previous steps.
- Messages: several messages (displayed to the user) can be set / translated here.

Other considerations

Make sure the payment type is from an unlimited account, so payments in Cyclos won't fail because of funds. The way the example script is done, first the payment is executed in PayPal and, if authorized, a payment is made in Cyclos. If this payment fails, to avoid inconsistency between the Cyclos account and the PayPal payment, a refund payment is performed in PayPal. If that refund fails, it is created a Custom system alert so it is advisable to have admins receiving that type of alert.

Loan module

Loan features in Cyclos 4 can be implemented using scripting. As loans tend to be very specific for each project, having it implemented with scripts brings the possibility to tailor the behavior to each project.

The example provided works as follows:

- An administrator has a custom operation to grant the loan, setting the amount, number of installments and first installment date.
- The loan is a payment from a system account to a user. It has a status, which can be either open or closed.
- The same custom operation also performs a scheduled payment from the user to system, with each installment amount and due date corresponding to the loan installments. This scheduled payment has (with a custom field) a link to the original loan. Also, the loan payment has a link to the scheduled payment, making it easy to navigate between them. However, if the loan is pending authorization, the scheduled payment won't be created.
- Each installment will be processed at the respective due date, allowing users to repay the loan with internal units. The administrator can, however, mark individual installments as settled, which means the installment won't be repaid internally, but with some other way (for example, with money or using other Cyclos payments).
- Once the scheduled payment is closed, an extension point updates the status of the original payment to closed.
- If the original loan was submitted to authorization, an extension point is triggered when it is authorized, and then creates the scheduled payment. **IMPORTANT:** If the administrator performing a payment also has the permission to authorize it, the payment will be immediately processed. So, be careful when testing with a single administrator group when authorization is desired, as in that case the loan would never get to authorization.

In order to configure the loan script, follow carefully each of the following steps:

Enable transaction number in currency

This can be checked under System > Currencies select the currency used for this operation, mark the Enable transfer number option and fill in the required parameters.

Create the transfer status flow

Under System > Accounts configuration > Transfer status flows, create a new one, with the following characteristics:

- Name: Loan status (can be changed as desired)
- Internal name: loan (this name is used in the example configuration)

After saving, create the following statuses:

- Closed (can be changed as desired)
 - Internal name: closed
 - Possible next statuses: <None>
- Open (can be changed as desired)
 - Internal name: open
 - Possible next statuses: Closed

Create the payment custom fields

Under System > Accounts configuration > Payment fields, create a new one, with the following fields:

- Installments count
 - Name: Installments count (can be changed as desired)
 - Internal name: numberOfInstallments (can be changed as desired, but this name is used in the example configuration)
 - Data type: integer
 - Required: yes
- First due date
 - Name: First due date (can be changed as desired)
 - Internal name: firstDueDate (can be changed as desired, but this name is used in the example configuration)
 - Data type: date
 - Required: yes
- Loan
 - Name: Loan (can be changed as desired)
 - Internal name: loan (can be changed as desired, but this name is used in the example configuration)
 - Data type: Linked entity
 - Linked entity type: Transaction
 - Required: no
- Repayment
 - Name: Repayment (can be changed as desired)

- Internal name: repayment (can be changed as desired, but this name is used in the example configuration)
- Data type: Linked entity
- Linked entity type: Transaction
- Required: no

Configure the system account from which payments will be performed to users

Under System > Accounts configuration > Account types, choose the (normally unlimited) account from which payments will be performed to users. Then set its internal name to some meaningful name. The example configuration uses debitUnits as internal name, but it can be changed later. Save the form.

Create the payment type which will be used to grant the loan

Still in the system account type details page for the account type, on the Transfer types tab, create a new Payment transfer type with the following characteristics:

- Name: Loan (can be changed as desired)
- Internal name: loanGrant (can be changed as desired, but this name is used in the example configuration)
- Default description: Loan grant (can be changed as desired, is the description for payments, visible in the account history)
- To: select the user account which will receive the payment
- Transfer status flows: Loan status
- Initial status for Loan status: Open
- Enabled: yes

After saving, on the "Payment fields" tab, add the following custom fields:

- Installments count
- First due date
- Repayment

If the loan can go through authorization, then create an authorization role in System > Account configuration > Authorization roles. Then, in the payment type details check the "Requires authorization" field. After saving, in the "Authorization levels" tab, add a new authorization level with that role. Afterwards, grant some administrator group the permission to manage that authorization role.

Configure the user account which will receive loans

Under System > Accounts configuration > Account types, choose the user account which will receive payments. Then set its internal name to some meaningful name. The example configuration uses userUnits as internal name, but it can be changed later. Save the form.

Create the payment type which will be used to repay the loan

Still in the user account type details page, on the Transfer types tab, create a new Payment transfer type with the following characteristics:

- Name: Loan repayment (can be changed as desired)
- Internal name: loanRepayment (can be changed as desired, but this name is used in the example configuration)
- Default description: Loan repayment (can be changed as desired, is the description for payments, visible in the account history)
- To: select the system account which granted the loan
- Enabled: yes
- Allows scheduled payment: yes
- Max installments on scheduled payments: 36 (any value greater than zero is fine)
- Show scheduled payments to receiver: yes
- Reserve total amount on scheduled payments: no

After saving, on the Payment fields tab, add the custom field named "Loan".

Create the library script

Under System > Tools > Scripts, create a new library script, with the following characteristics:

- Name: Loan
- Type: Library
- Included libraries: none
- Parameters:

```
# Loan configuration
loan.account = debitUnits
loan.type = loanGrant
#loan.description =

# Repayment configuration
repayment.account = userUnits
repayment.type = loanRepayment
#repayment.description =

# Payment custom fields
```

```

field.loan = loan
field.repayment = repayment

# Monthly compound interest rate (zero for none)
monthlyInterestRate = 0

# Transfer status configuration
status.flow = loan
status.open = open
status.closed = closed

# Custom operation configuration
operation.amount = amount
operation.installments = numberOfInstallments
operation.firstDueDate = firstDueDate

# Messages
message.invalidInstallments = The number of installments is invalid
message.invalidLoanAmount = Invalid loan amount
message.invalidFirstDueDate = The first due date cannot be lower than tomorrow
message.loanGranted = The loan was successfully granted
message.loanGranted.pending = The loan was granted and is now pending authorization
message.authorization.expired = The loan cannot be authorized as the first due date is over

```

- Script code:

```

import org.cyclos.entities.banking.Payment
import org.cyclos.entities.banking.PaymentTransferType
import org.cyclos.entities.banking.ScheduledPayment
import org.cyclos.entities.banking.SystemAccountType
import org.cyclos.entities.banking.TransactionCustomField
import org.cyclos.entities.banking.Transfer
import org.cyclos.entities.banking.TransferStatus
import org.cyclos.entities.banking.TransferStatusFlow
import org.cyclos.entities.banking.UserAccountType
import org.cyclos.entities.users.User
import org.cyclos.impl.access.SessionData
import org.cyclos.impl.banking.PaymentServiceLocal
import org.cyclos.impl.banking.ScheduledPaymentServiceLocal
import org.cyclos.impl.banking.TransferStatusServiceLocal
import org.cyclos.impl.system.ConfigurationAccessor
import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.utils.persistence.EntityManagerHandler
import org.cyclos.model.ValidationException
import org.cyclos.model.banking.accounts.SystemAccountOwner
import org.cyclos.model.banking.transactions.InstallmentDTO
import org.cyclos.model.banking.transactions.PaymentVO
import org.cyclos.model.banking.transactions.PerformPaymentDTO
import org.cyclos.model.banking.transactions.PerformScheduledPaymentDTO
import org.cyclos.model.banking.transactions.ScheduledPaymentVO
import org.cyclos.model.banking.transfers.TransferVO
import org.cyclos.model.banking.transferstatus.ChangeTransferStatusDTO
import org.cyclos.model.banking.transferstatus.TransferStatusVO
import org.cyclos.model.banking.transferstypes.TransferTypeVO
import org.cyclos.model.utils.TimeField
import org.cyclos.server.utils.DateHelper
import org.cyclos.utils.BigDecimalHelper

```

```

import groovy.transform.TypeChecked

@TypeChecked
class Loan {
    Map<String, Object> config
    EntityManagerHandler emh
    PaymentServiceLocal paymentService
    ScheduledPaymentServiceLocal scheduledPaymentService
    TransferStatusServiceLocal transferStatusService
    ScriptHelper scriptHelper
    ConfigurationAccessor configuration

    double monthlyInterestRate
    SystemAccountType systemAccount
    UserAccountType userAccount
    PaymentTransferType loanType
    PaymentTransferType repaymentType
    TransactionCustomField loanField
    TransactionCustomField repaymentField
    TransferStatusFlow flow
    TransferStatus open
    TransferStatus closed

    Loan(Binding binding) {
        def variables = binding.variables
        config = [:]
        def params = variables.scriptParameters as Map<String, Object>
        [
            'loan.account': 'systemAccount',
            'loan.type': 'loanGrant',
            'loan.description': null,
            'repayment.account': 'userUnits',
            'repayment.type': 'loanRepayment',
            'repayment.description': null,
            'field.loan': 'loan',
            'field.repayment': 'repayment',
            'monthlyInterestRate' : null,
            'status.flow': 'loan',
            'status.open': 'open',
            'status.closed': 'closed',
            'operation.amount': 'amount',
            'operation.installments': 'installments',
            'operation.firstDueDate': 'firstDueDate',
            'message.invalidInstallments':
                'The number of installments is invalid',
            'message.invalidLoanAmount': 'Invalid loan amount',
            'message.invalidFirstDueDate':
                'The first due date cannot be lower than tomorrow',
            'message.loanGranted':
                'The loan was successfully granted to the user',
            'message.loanGranted.pending':
                'The loan was granted and is now pending authorization',
            'message.authorization.expired':
                'The loan cannot be authorized as the first due date is over'
        ].each { k, v ->

```

```

        config[k] = params[k] ?: v
    }
    emh = variables.entityManagerHandler as EntityManagerHandler
    paymentService = variables.paymentService as PaymentServiceLocal
    scriptHelper = variables.scriptHelper as ScriptHelper
    scheduledPaymentService =
        variables.scheduledPaymentService as ScheduledPaymentServiceLocal
    transferStatusService =
        variables.transferStatusService as TransferStatusServiceLocal
    configuration =
        (variables.sessionData as SessionData).configuration as
ConfigurationAccessor

    systemAccount = emh.find(SystemAccountType, config.'loan.account' as String)
    if (systemAccount.currency.transactionNumber == null
    || !systemAccount.currency.transactionNumber.used) {
        throw new IllegalStateException("The currency ${systemAccount.currency.name}"
        + " doesn't have transaction number enabled")
    }
    userAccount = emh.find(UserAccountType, config.'repayment.account' as String)
    loanType =
        emh.find(PaymentTransferType, config.'loan.type' as String, systemAccount)
    repaymentType =
        emh.find(PaymentTransferType, config.'repayment.type' as String,
userAccount)
    if (!repaymentType.allowsScheduledPayments) {
        throw new IllegalStateException(
            "The repayment type ${repaymentType.name} doesn't allows scheduled payment")
    }

    loanField = emh.find(TransactionCustomField, config.'field.loan' as String)
    repaymentField =
        emh.find(TransactionCustomField, config.'field.repayment' as String)
    if (!loanType.customFields.contains(repaymentField)) {
        throw new IllegalStateException("The loan type ${loanType.name}"
        + " doesn't contain the custom field ${repaymentField.name}")
    }
    if (!repaymentType.customFields.contains(loanField)) {
        throw new IllegalStateException("The repayment type ${repaymentType.name}"
        + " doesn't contain the custom field ${loanField.name}")
    }
    flow = emh.find(TransferStatusFlow, config.'status.flow' as String)
    open = emh.find(TransferStatus, config.'status.open' as String, flow)
    closed = emh.find(TransferStatus, config.'status.closed' as String, flow)
    monthlyInterestRate = (config.'monthlyInterestRate' as String)?.toDouble() ?: 0
}

BigDecimal calculateInstallmentAmount(BigDecimal amount, int installments,
    Date grantDate, Date firstInstallmentDate) {

    // Calculate the delay
    Date shouldBeFirstExpiration = DateHelper.add(grantDate, TimeField.DAYS, 30)
    int delay = (int) DateHelper.daysBetween(firstInstallmentDate,
shouldBeFirstExpiration)
    if (delay < 0) {
        delay = 0
    }
    double interest = monthlyInterestRate / 100.0

```



```

        double numerator = ((1 + interest) ** (installments + delay / 30.0)) * interest
        double denominator = ((1 + interest) ** installments) - 1
        BigDecimal result = amount * numerator / denominator
        return BigDecimalHelper.round(result, systemAccount.currency.precision)
    }

    void close(ScheduledPayment scheduledPayment) {
        def map = scriptHelper.wrap(scheduledPayment)
        Payment loan = map.get(loanField.internalName) as Payment
        Transfer loanTransfer = loan.transfer
        TransferStatus status = loanTransfer.getStatus(flow)
        if (status != closed) {
            // The loan was not closed: close it
            transferStatusService.changeStatus(new ChangeTransferStatusDTO([
                transfer: new TransferVO(loanTransfer.id),
                newStatus: new TransferStatusVO(closed.id)
            ]))
        }
    }

    Payment grant(User user, Map<String, Object> formParameters) {
        BigDecimal loanAmount = formParameters[config.'operation.amount'] as BigDecimal
        int installments = formParameters[config.'operation.installments'] as int
        Date firstDueDate = formParameters[config.'operation.firstDueDate'] as Date
        Date minDate = DateHelper.shiftToNextDay(new Date(), configuration.timeZone)
        if (installments < 1 || installments > repaymentType.maxInstallments)
            throw new ValidationException(config.'message.invalidInstallments' as String)
        if (loanAmount < 1)
            throw new ValidationException(config.'message.invalidLoanAmount' as String)
        if (firstDueDate < minDate)
            throw new ValidationException(config.'message.invalidFirstDueDate' as String)

        // Grant the loan, copying the installments count and first due date
        PerformPaymentDTO perform = new PerformPaymentDTO([
            from: SystemAccountOwner.instance(),
            to: user,
            type: new TransferTypeVO(loanType.id),
            amount: loanAmount,
            description: config.'loan.description' as String
        ])
        def performBean = scriptHelper.wrap(perform)
        performBean[config.'operation.installments' as String] = installments
        performBean[config.'operation.firstDueDate' as String] = firstDueDate
        PaymentVO loanVO = paymentService.perform(perform)
        Payment loan = emh.find(Payment, loanVO.id)
        if (loan.transfer != null) {
            // The loan is processed. Create the repayment
            createRepayment(loan)
        }
        return loan
    }

    ScheduledPayment createRepayment(Payment payment) {
        Transfer loanTransfer = payment.transfer
        if (loanTransfer == null) {
            return null
        }
        TransferStatus currentStatus = loanTransfer.getStatus(flow)
    }

```

```

if (currentStatus != open) {
    throw new ValidationException(
        "The initial status for flow ${flow.name} in ${loanType.name} "
        + "is not the expected one: ${open.name}, but ${currentStatus?.name} instead")
}

// Read the scheduling information from the loan
def loanBean = scriptHelper.wrap(payment)
def existingRepayment = loanBean[repaymentField.internalName]
if (existingRepayment != null) {
    return existingRepayment as ScheduledPayment
}
BigDecimal loanAmount = payment.amount
Integer installments = loanBean[config.'operation.installments'] as Integer
Date firstDueDate = loanBean[config.'operation.firstDueDate'] as Date

// Make sure the first due date is not expired
Date now = new Date()
if (firstDueDate.before(now)) {
    throw new ValidationException(config.'message.authorization.expired' as String)
}

// Perform the repayment scheduled payment
PerformScheduledPaymentDTO dto = new PerformScheduledPaymentDTO([
    from: payment.toOwner,
    to: payment.fromOwner,
    type: new TransferTypeVO(repaymentType.id),
    amount: payment.amount,
    description: config.'repayment.description' as String
])
def dtoBean = scriptHelper.wrap(dto)
dtoBean.installmentsCount = installments
dtoBean.firstInstallmentDate = firstDueDate
dtoBean[loanField.internalName] = payment

// Interest
if (monthlyInterestRate > 0.00001) {
    BigDecimal installmentAmount = calculateInstallmentAmount(
        loanAmount, installments, new Date(), firstDueDate)

    dto.installments = []
    Date dueDate = firstDueDate
    for (int i = 0; i < installments; i++) {
        def installment = new InstallmentDTO()
        def instBean = scriptHelper.wrap(installment)
        instBean.dueDate = dueDate
        instBean.amount = installmentAmount
        dto.installments << installment
        dueDate = DateHelper.add(dueDate, TimeField.DAYS, 30)
    }
    dtoBean.amount = installmentAmount * installments
}

ScheduledPaymentVO repaymentVO = scheduledPaymentService.perform(dto)
ScheduledPayment repayment = emh.find(ScheduledPayment, repaymentVO.id)

// Update the loan with the repayment link
loanBean[repaymentField.internalName] = repayment

```

```

        return repayment
    }
}

binding.variables.loan = new Loan(binding)

```

Create the custom operation script

Create a new script for the custom operation, with the following characteristics:

- Name: Grant loan
- Type: Custom operation
- Included libraries: Loan
- Run with all permissions: No
- Parameters: leave empty
- Script code executed when the custom operation is executed:

```

import org.cyclos.entities.banking.Payment
import org.cyclos.entities.users.User

import groovy.transform.TypeChecked

@TypeChecked
def grantLoan() {
    def variables = binding.variables
    Loan loan = variables.loan as Loan
    Payment payment = loan.grant(variables.user as User,
        variables.formParameters as Map<String, Object>)
    if (payment.transfer == null) {
        return loan.config['message.loanGranted.pending']
    } else {
        return loan.config['message.loanGranted']
    }
}

grantLoan()

```

Create two extension point scripts

Create a new script for the transaction extension point, which will close the loan when all installments are processed:

- Name: Loan closing
- Type: Extension point
- Included libraries: Loan
- Parameters: leave empty
- Script code executed when the data is saved:

```

import org.cyclos.entities.banking.ScheduledPayment
import org.cyclos.model.ValidationException
import org.cyclos.model.banking.transactions.ScheduledPaymentStatus

import groovy.transform.TypeChecked

@TypeChecked
def closeLoan() {
    ScheduledPayment transaction = binding.variables.transaction as ScheduledPayment
    if (transaction.status == ScheduledPaymentStatus.CANCELED) {
        // Should never cancel a loan scheduled payment
        throw new ValidationException("Cannot cancel a loan")
    } else if (transaction.status == ScheduledPaymentStatus.CLOSED) {
        // Close the loan
        (binding.variables.loan as Loan).close(transaction)
    }
}

closeLoan()

```

Also, create another script for the authorization extension point, which will create the repayment scheduled payment once the loan is authorized:

- Name: Loan authorization
- Type: Extension point
- Included libraries: Loan
- Parameters: leave empty
- Script code executed when the data is saved:

```

import org.cyclos.entities.banking.Payment

import groovy.transform.TypeChecked

@TypeChecked
def createRepayment() {
    Payment transaction = binding.variables.transaction as Payment
    if (transaction.getTransfer() != null) {
        // The transaction was authorized, create the repayment
        (binding.variables.loan as Loan).createRepayment(transaction)
    }
}

createRepayment()

```

Create the custom operation

Under System > Tools > Custom operations, create a new one, with the following characteristics:

- Name: Grant loan (can be changed, is the label displayed to users)

- Enabled: yes
- Scope: User
- Script: Grant loan
- Script parameters: leave empty
- Result type: Notification
- Has file upload: no
- Main menu: Banking
- User management section: Banking
- Information text: you can add here some text explaining the process – it will be displayed in the operation page
- Confirmation text: add here some text which will be displayed in a confirmation dialog before granting the loan

After saving, create the following fields:

- Amount
 - Internal name: amount
 - Data type: Decimal
 - Required: yes
- Installment count
 - Internal name: numberOfInstallments
 - Data type: Integer
 - Required: yes
- First due date
 - Internal name: firstDueDate
 - Data type: Date
 - Required: yes

Create the extension points

Under System > Tools > Extension points, create a two new extension points, each with the following characteristics:

- A transaction extension point (will close the loan when all installments are processed):
 - Name: Close loan
 - Type: Transaction

- Enabled: yes
- Transfer types: Units account – Loan repayment (choose the loan repayment type)
- Events: Change status
- Script: Loan closing
- Script parameters: leave empty
- An authorization extension point (will create the repayment once the loan is authorized):
 - Name: Loan authorization
 - Type: Authorization
 - Enabled: yes
 - Transfer types: Debit account – Loan grant (choose the loan grant type)
 - Events: Authorize
 - Script: Loan authorization
 - Script parameters: leave empty

Grant the administrator permissions

Under System > User configuration > Groups, select the Network administrators group (or the ones that will grant loans). Then, in the Permissions tab:

- Under User management > Run custom operations over users, check the Grant loan operation
- Under Accounts > Transfer status flows, make Loan visible, but not editable.
- Under Accounts > Visible transaction fields, select all related custom fields.
- Under User accounts > Scheduled payments, select View (and maybe process installment and settle too).

Enable the custom operation for users which will be able to receive loans

In System > User configuration > Products (permissions), select the member product for users which will be able to receive loans. In the Custom operations field, make the Grant loan operation enabled. Leave the run checkbox unchecked (or users would be able to grant loans to themselves!).

You can permit users to repay loan installments anticipated in Units. For this you have to check in the member product 'process installment' and the user need to have permissions to make a payment of the transaction type used for the loan repayments.

Integrating with Global USSD

This example allows enabling operations to be performed via USSD. As each USSD gateway has a different protocol, a generic solution is not available. This script assumes the USSD integration is provided by [Global USSD](#).

The provided examples allows getting the account information and performing direct payments. The mobile phone number used in the USSD interaction must exist as a mobile phone in Cyclos for an active user.

It is recommended that a channel named USSD is created in Cyclos, so its settings won't affect other web service clients. For example, the script assumes there is no confirmation password. Also, having a separated channel allows a finer control for users if they want to enable or disable the channel. The steps below assume a specific channel is used.

Create the USSD channel

On the System > System configuration > Channels menu, create a new channel, with the following fields:

- Name: USSD (can be changed as desired)
- Internal name: ussd

Enable the USSD channel for users

On the System > System configuration > Configurations menu, select either the default or a specific configuration. On the channels tab, click USSD. Then fill in the fields as following:

- Enabled: checked
- User access: If 'Enabled by default' is set, all users will be able to use the USSD operations initially. This can be changed as desired.
- User identification method: Mobile phone
- Default user identification method: No default.
- Access password: Login password (can be changed to a PIN if desired).
- Confirmation password: None (this is important, as the script will only ask for the access password).
- Session timeout: Leave blank (no sessions will be used from the Cyclos point-of-view).
- Perform payments - user identification methods: Login name (this is how the payee will be interpreted when performing a payment).

Create a payment type for USSD

Under System > Account configuration > Account types, select the user account. Then on the 'Transfer types' tab, create a new payment type with the following fields:

- Name: USSD payment (can be changed as desired).
- To: (select a user account type which will be the destination, normally the same as From).
- Enabled: checked.
- Channels: USSD.
- User identification methods: Mobile phone (or leave empty, meaning All).

Grant permissions for users to perform this payment type

In the System > User configurations > Products (permissions), select a product which contains the account (or create a new one), adding the USSD payment type in 'User payments'.

Create the library script

Under System > Tools > Scripts, create a new library script, with the following characteristics:

- Name: USSD library
- Type: Library
- Included libraries: none
- Parameters:

```
### Settings

# The session timeout, in seconds
sessionTimeout=60

# The channel internal name which will be used for the operations
channel=ussd

### Translations
mainMenu.title=Main menu
mainMenu.accountInfo=Account information
mainMenu.payment=Perform payment
accountInfo.type=Account
accountInfo.balance=Balance: {0}
accountInfo.reservedAmount=Reserved: {0}
accountInfo.creditLimit=Negative limit: {0}
accountInfo.availableBalance=Available: {0}
accountInfo.noAccount=You don't have any account
accountInfo.error.type=The account {0} is invalid
payment.payee=Pay to user
payment.error.payee=The user {0} is invalid
payment.noPaymentType=No possible payment type to pay to {0} using this channel
payment.type=Payment type
payment.error.type=The payment type {0} is invalid
payment.amount=Amount
payment.error.amount=The amount is invalid: {0}
payment.confirmation=Are you sure to pay {0} to {1}, with type {2}?
payment.performed=You have successfully paid {0} to {1}, with type {2}
payment.error.general=There was an unknown error when performing the payment
payment.error.balance=There is no available balance to perform this payment
```



```

payment.error.maxAmount=The maximum amount has been exceeded for this period
payment.error.maxPayments=The maximum number of payments has been exceeded for this period
payment.error.minTime=The minimum time between the last payment has not yet passed
password.error.invalid={0} is invalid
password.error.blocked={0} has been blocked
general.submit=Submit
general.unregisteredPhone=Your phone number, {0}, is not registered in Cyclos
general.sessionExpired=Your session has expired. Please, restart the operation.
general.returnToMainMenu=(Input 0 to return to Main Menu)"
general.actionAborted=The action {0} was aborted

```

- Script code:

```

import java.text.MessageFormat

import org.cyclos.entities.banking.PaymentTransferType
import org.cyclos.entities.users.MobilePhone
import org.cyclos.entities.utils.CurrencyAmount
import org.cyclos.impl.access.SessionData
import org.cyclos.model.EntityNotFoundException
import org.cyclos.model.ValidationException
import org.cyclos.model.access.IndefinitelyBlockedPasswordException
import org.cyclos.model.access.TemporarilyBlockedPasswordException
import org.cyclos.model.banking.InsufficientBalanceException
import org.cyclos.model.banking.MaxAmountExceededException
import org.cyclos.model.banking.MaxPaymentsExceededException
import org.cyclos.model.banking.MinTimeBetweenPaymentsException
import org.cyclos.model.banking.accounts.AccountOwner
import org.cyclos.model.banking.accounts.AccountVO
import org.cyclos.model.banking.accounts.AccountWithStatusVO
import org.cyclos.model.banking.transactions.PaymentVO
import org.cyclos.model.banking.transactions.PerformPaymentDTO
import org.cyclos.model.banking.transfertypes.TransferTypeVO
import org.cyclos.model.users.users.UserLocatorVO
import org.cyclos.model.utils.ModelHelper
import org.cyclos.server.utils.ObjectParameterStorage
import org.cyclos.utils.BigDecimalHelper
import org.cyclos.utils.StringHelper

class Pages {
    static String MAIN_MENU = "mainMenu"
    static String ACBALANCE_ASKACCOUNT = "acBalanceAskAccount"
    static String ACBALANCE_ASKPASSWORD = "acBalanceAskPassword"
    static String ACBALANCE_DISPLAY = "acBalanceDisplay"
    static String PAYMENT_ASKPAYEE = "payAskPayee"
    static String PAYMENT_ASKAMOUNT = "payAskAmount"
    static String PAYMENT_ASKPAYMENTTYPE = "payAskPaymentType"
    static String PAYMENT_ASKPASSWORD = "payAskPassword"
    static String PAYMENT_PERFORM = "payPerform"
}

class UssdHandler {
    MobilePhone phone
    SessionData userSessionData
    ObjectParameterStorage session
    boolean newSession
    def binding

```

```

static void newXmlMessage(def xml, String message) {
    if (StringHelper.isBlank(message)) {
        return
    }
    xml.div(message)
    xml.div("")
}

UssdHandler(MobilePhone phone, SessionData userSessionData, Object binding) {
    this.phone = phone
    this.userSessionData = userSessionData
    this.binding = binding
    def sessionKey = "ussd_" + phone.normalizedNumber
    newSession = !binding.scriptStorageHandler.exists(sessionKey)
    session = binding.scriptStorageHandler.get(sessionKey,
        binding.scriptParameters.sessionTimeout as int)
}

boolean isNewSession() {
    newSession
}

Object propertyMissing(String name) {
    binding[name]
}

Object methodMissing(String name, args) {
    throw new EntityNotFoundException(entityType: "UssdOperation", key: name)
}

/** Ask for the confirmation password */
private void askPassword(def xml, String pageToSend,
    String title, String message) {
    newXmlMessage(xml, message)
    xml.div() {
        xml.input(
            navigationId: "form",
            title: title,
            name: "PASSWORD",
            type: "number")
    }
    xml.div(scriptParameters["general.returnToMainMenu"])
    xml.navigation(id: "form"){
        xml.link(
            pageId : pageToSend,
            scriptParameters["general.submit"])
    }
}

/** Ask for the payment receiver */
private void askPayee(def xml, String message) {
    newXmlMessage(xml, message)

    xml.div() {
        xml.input(
            navigationId: "form",
            title: scriptParameters["payment.payee"],

```

```

        name: "PAYEE",
        type: "Text")
    }
    xml.div(scriptParameters["general.returnToMainMenu"])
    xml.navigation(id: "form") {
        xml.link(
            pageId : Pages.PAYMENT_ASKPAYMENTTYPE,
            scriptParameters["general.submit"])
    }
}

/** Ask for the payment type */
private void askPaymentType(def xml, String message) {

    def paymentTypes = (session.paymentTypes ?: [:]).collectEntries({ k, v ->
        [
            k,
            entityManagerHandler.find(PaymentTransferType, v)
        ]
    })

    if (paymentTypes.size() == 1) {
        // There is a single payment type - store it and ask the amount
        request.parameters.PAYMENT_TYPE = "1"
        payAskAmount(xml, "")
        return
    }

    newXmlMessage(xml, message)
    // Generate the option list
    paymentTypes.each {
        xml.div("${it.key}: ${it.value.name}")
    }
    // Generate the form to allow user choose
    xml.div() {
        xml.input(navigationId: "form",
            title: scriptParameters["payment.type"],
            name: "PAYMENT_TYPE",
            type: "number")
    }
    xml.navigation(id: "form"){
        xml.link(pageId : Pages.PAYMENT_ASKAMOUNT,
            scriptParameters["general.submit"])
    }
}

/** Ask for the payment amount */
private void askAmount(def xml, String message) {
    newXmlMessage(xml, message)

    xml.div() {
        xml.input(navigationId: "form",
            title: scriptParameters["payment.amount"],
            name: "AMOUNT",
            type: "number")
    }
    xml.div(scriptParameters["general.returnToMainMenu"])
    xml.navigation(id: "form") {

```

```

        xml.link(pageId : Pages.PAYMENT_ASKPASSWORD,
            scriptParameters["general.submit"])
    }
}

/** Check for a password, either returning true and don't touching
 * the XML or returning false and sending an error in the XML */
private boolean checkPassword(def xml, String password, String nextPage) {
    def accessPassword = userSessionData.channelAccessAccessor.accessPassword
    try {
        passwordHandler.checkPassword(false,
            accessPassword,
            userSessionData.loggedUser,
            password)

        return true
    } catch (TemporarilyBlockedPasswordException |
        IndefinitelyBlockedPasswordException e) {
        askPassword(xml, nextPage,
            accessPassword.name,
            MessageFormat.format(scriptParameters["password.error.blocked"],
                accessPassword.name))
        return false
    } catch (Exception e) {
        askPassword(xml, nextPage,
            accessPassword.name,
            MessageFormat.format(scriptParameters["password.error.invalid"],
                accessPassword.name))
        return false
    }
}

/** Performs the payment, returning the result if succeed or sending the XML
 * error if not */
private PaymentVO performPayment(def xml, PerformPaymentDTO dto) {
    try {
        return paymentService.perform(dto)
    } catch (ValidationException e) {
        mainMenu(xml, e.validation?.firstError)
    } catch (InsufficientBalanceException e) {
        mainMenu(xml, scriptParameters["payment.error.balance"])
    } catch (MaxAmountExceededException e) {
        mainMenu(xml, scriptParameters["payment.error.maxAmount"])
    } catch (MaxPaymentsExceededException e) {
        mainMenu(xml, scriptParameters["payment.error.maxPayments"])
    } catch (MinTimeBetweenPaymentsException e) {
        mainMenu(xml, scriptParameters["payment.error.minTime"])
    } catch (Exception e) {
        mainMenu(xml, scriptParameters["payment.error.general"])
    }
    // there was some error
    return null
}

/** Removes all payment-related attributes from the session */
private void clearSessionPayment() {
    [
        "payee",
        "paymentTypes",

```

```

        "paymentType",
        "amount"
    ].forEach(session.&remove)
}

/** Handler for Pages.MAIN_MENU */
String mainMenu(def xml, String message) {
    newXmlMessage(xml, message)
    xml.navigation() {
        xml.link(
            accesskey: "1",
            pageId: Pages.ACBALANCE_ASKACCOUNT,
            scriptParameters["mainMenu.accountInfo"])
        xml.link(
            accesskey: "2",
            pageId: Pages.PAYMENT_ASKPAYEE,
            scriptParameters["mainMenu.payment"])
    }

    // Clear the session attributes for specific actions
    clearSessionPayment()
}

/** Handler for Pages.ACBALANCE_ASKACCOUNT */
void acBalanceAskAccount(def xml, String message) {
    List<AccountWithStatusVO> accountSummaries =
        binding.accountService.getAccountsSummary(userSessionData.loggedUser,
            null)

    if (accountSummaries.isEmpty()) {
        mainMenu(xml, scriptParameters["accountInfo.noAccount"])
        return
    }

    // create a map with visible accounts and add this to context
    def accounts = [:]
    def option = 1
    accountSummaries.each { a ->
        accounts."${option}" = a
        option++
    }
    session.accounts = accounts

    askAccount(xml, null)
}

/** Handler for Pages.ASKACCOUNT */
void askAccount(def xml, String message) {
    def accounts = session.accounts
    if (accounts.size() == 1) {
        request.parameters.ACCOUNT = "1"
        acBalanceAskPassword(xml, null)
        return
    }

    newXmlMessage(xml, message)
}

```

```

// Generate the option list
def key = 1
accounts.each {
    xml.div("${it.key}: ${it.value.type.name}")
}

// Generate the form to allow user choose
xml.div() {
    xml.input(navigationId: "form",
        title: scriptParameters["accountInfo.type"],
        name: "ACCOUNT",
        type: "number")
}
xml.navigation(id: "form"){
    xml.link(pageId : Pages.ACBALANCE_ASKPASSWORD,
        scriptParameters["general.submit"])
}
}

/** Handler for Pages.ACBALANCE_ASKPASSWORD */
void acBalanceAskPassword(def xml, String message) {
    def acc = request.parameters.ACCOUNT
    // Check whether to return to the main menu
    if (acc == "0") {
        mainMenu(xml, MessageFormat.format(scriptParameters["general.actionAborted"],
            scriptParameters["mainMenu.accountInfo"]))
        return
    }
    // Validate the Account
    def accounts = session.accounts
    if (!accounts.containsKey(acc)) {
        askAccount(xml,
            MessageFormat.format(scriptParameters["accountInfo.error.type"], acc))
        return
    }
    // Store the account summary type in the session
    session.accountId = accounts[acc].id

    askPassword(xml, Pages.ACBALANCE_DISPLAY,
        userSessionData.channelAccessAccessor.accessPassword.name,
        null)
}

/** Handler for Pages.ACBALANCE_DISPLAY */
void acBalanceDisplay(def xml, String message) {
    def password = request.parameters.PASSWORD
    // Check whether to return to the main menu
    if (password == "0") {
        mainMenu(xml, MessageFormat.format(scriptParameters["general.actionAborted"],
            scriptParameters["mainMenu.accountInfo"]))
        return
    }
    // Check PASSWORD
    if (!checkPassword(xml, password, Pages.ACBALANCE_DISPLAY)) {
        return
    }

    // Generate the Balance result string

```

```

def account = accountService.getAccountWithStatus(
    new AccountVO(session.accountId as long), null)

xml.div(account.type.name)
def status = account.status

def balance = formatter.format(
    ModelHelper.currencyAmount(account.currency,
    status.balance))
xml.div(MessageFormat.format(scriptParameters["accountInfo.balance"],
    balance))
if (BigDecimalHelper.isPositive(status.reservedAmount)) {
    def reservedAmount = formatter.format(
        ModelHelper.currencyAmount(account.currency,
        status.reservedAmount))
    xml.div(MessageFormat.format(
        scriptParameters["accountInfo.reservedAmount"],
        reservedAmount))
}
if (BigDecimalHelper.isPositive(status.creditLimit)) {
    def creditLimit = formatter.format(
        ModelHelper.currencyAmount(
        account.currency,
        status.creditLimit))
    xml.div(MessageFormat.format(
        scriptParameters["accountInfo.creditLimit"],
        creditLimit))
}
if (!BigDecimalHelper.areEquals(status.balance,
status.availableBalance)) {
    def availableBalance = formatter.format(
        ModelHelper.currencyAmount(account.currency,
        status.availableBalance))
    xml.div(MessageFormat.format(
        scriptParameters["accountInfo.availableBalance"],
        availableBalance))
}

xml.navigation() {
    xml.link(
        accesskey : "0",
        pageId : Pages.MAIN_MENU,
        scriptParameters["mainMenu.title"])
}

/** Handler for Pages.PAYMENT_ASKPAYEE */
void payAskPayee(def xml, String message) {
    askPayee(xml, null)
}

/** Handler for Pages.PAY_ASK_PAYMENT_TYPE */
void payAskPaymentType(def xml, String message) {
    def payee = request.parameters.PAYEE
    // Check whether to return to the main menu
    if (payee == "0") {
        mainMenu(xml, MessageFormat.format(scriptParameters["general.actionAborted"],
            scriptParameters["mainMenu.payment"]))
    }
}

```

```

        return
    }

    // Validate the payee
    AccountOwner accOwnerPayee
    def locator = new UserLocatorVO(principal: payee)
    try {
        accOwnerPayee = transactionService.locateForPayment(locator).accountOwner
    } catch (Exception e) {
        askPayee(xml,
            MessageFormat.format(scriptParameters["payment.error.payee"], payee))
        return
    }

    // Add the payee to session context
    session.payee = accOwnerPayee

    // Get the allowed TT between payer and payee and generate options
    def payer = userSessionData.loggedUser
    def paymentData = transactionService.getPaymentToOwnerData(payer,
        accOwnerPayee, null)
    if (paymentData.paymentTypes.size == 0) {
        askPayee(xml,
            MessageFormat.format(scriptParameters["payment.noPaymentType"],
                payee))
        return
    } else {
        // create a map with allowed paymentTypes and add this to context
        def paymentTypes = [:]
        def option = 1
        paymentData.paymentTypes.each { tt ->
            paymentTypes."<strong>${option}</strong>" = conversionHandler.convert(PaymentTransferType,
                tt)
            option++
        }
        session.paymentTypes = paymentTypes

        // Ask the payment type
        askPaymentType(xml, null)
    }
}

/** Handler for Pages.PAY_ASKAMOUNT */
void payAskAmount(def xml, String message) {
    def tt = request.parameters.PAYMENT_TYPE
    // Check whether to return to the main menu
    if (tt == "0") {
        mainMenu(xml, MessageFormat.format(scriptParameters["general.actionAborted"],
            scriptParameters["mainMenu.payment"]))
        return
    }
    // Validate the TT
    def paymentTypes = session.paymentTypes
    if (!paymentTypes.containsKey(tt)) {
        askPaymentType(xml,
            MessageFormat.format(scriptParameters["payment.error.type"], tt))
        return
    }
}

```



```

    // Store the payment type in the session
    session.paymentType = entityManagerHandler.find(PaymentTransferType,
        paymentTypes[tt])

    // Ask the amount
    askAmount(xml, null)
}

/** Handler for Pages.PAYMENT_ASKPASSWORD */
void payAskPassword(def xml, String message) {
    def amt = request.parameters.AMOUNT

    // Check whether to return to the main menu
    if (amt == "0") {
        mainMenu(xml, MessageFormat.format(scriptParameters["general.actionAborted"],
            scriptParameters["mainMenu.payment"]))
        return
    }

    // Validate the AMOUNT
    BigDecimal amount
    try {
        amount = new BigDecimal(amt)
    } catch (Exception e) {
        askAmount(xml, MessageFormat.format(scriptParameters["payment.error.amount"],
            amt))
        return
    }

    // Add the amount to the session
    session.amount = amount

    // Now ask the password
    askPassword(xml, Pages.PAYMENT_PERFORM,
        userSessionData.channelAccessAccessor.accessPassword.name,
        getPaymentMessage(scriptParameters["payment.confirmation"]))
}

/** Handler for Pages.PAYMENT_PERFORM */
void payPerform(def xml, String message) {
    def password = request.parameters.PASSWORD

    // Check whether to return to the main menu
    if (password == "0") {
        mainMenu(xml, MessageFormat.format(scriptParameters["general.actionAborted"],
            scriptParameters["mainMenu.payment"]))
        return
    }

    // first validate PASSWORD
    checkPassword(xml, password, Pages.PAYMENT_PERFORM)

    // Build the PerformPaymentDTO
    def dto = new PerformPaymentDTO()
    dto.from = userSessionData.loggedUser
    dto.to = session.payee
    dto.amount = session.amount
    dto.type = new TransferTypeVO(session.paymentType.id)
}

```

```

        // perform the payment
        def result = performPayment(xml, dto)
        if (result) {
            // Only handle the success, because on failure the XML is already sent
            mainMenu(xml, getPaymentMessage(scriptParameters["payment.performed"]))
        }
    }

    String getPaymentMessage(String template) {
        def paymentType = session.paymentType
        if (paymentType == null || session.amount == null) return null
        def amount = new CurrencyAmount(paymentType.currency, session.amount)
        return MessageFormat.format(template,
            formatter.format(amount),
            formatter.format(session.payee),
            formatter.format(paymentType))
    }
}

```

Create the custom web service script

Create a new script for the custom web service, with the following characteristics:

- Name: USSD web service
- Type: Custom web service
- Included libraries: USSD library
- Parameters: leave empty
- Script code executed when the custom operation is executed:

```

import java.text.MessageFormat

import org.apache.commons.lang3.StringUtils
import org.cyclos.entities.users.MobilePhone
import org.cyclos.entities.users.QMobilePhone
import org.cyclos.impl.access.SessionDataFactory
import org.cyclos.model.users.users.UserStatus
import org.cyclos.model.utils.ResponseInfo

import groovy.xml.MarkupBuilder

// The XML builder will write to a StringWriter
def stringWriter = new StringWriter()
def xml = new MarkupBuilder(stringWriter)
xml.doubleQuotes = true
xml.omitNullAttributes = true
xml.mkp.xmlDeclaration(version:"1.0", encoding: "UTF-8")

// Resolve the normalized international phone number via the subscriber parameter
String phoneNumber = StringUtils.trimToNull(request.parameters.subscriber)
MobilePhone mobilePhone = null;
if (phoneNumber == null) {
    return new ResponseInfo(422, "The subscriber parameter is missing")
}

```

```

} else {
    // Find the mobile phone in Cyclos
    phoneNumber = "+" + StringUtils.removeStart(phoneNumber, "+");
    def mp = QMobilePhone.mobilePhone;
    mobilePhone = entityManagerHandler.from(mp)
        .where(mp.normalizedNumber.eq(phoneNumber),
            mp.user().status.eq(UserStatus.ACTIVE))
        .singleResult(mp)
}
if (mobilePhone == null) {
    // The mobile phone is not found in Cyclos
    xml.page(version: "2.0") {
        div(MessageFormat.format(scriptParameters['general.unregisteredPhone'],
            phoneNumber))
    }
} else {
    // Get session or create a new one
    def runAs = SessionDataFactory.direct(mobilePhone)
        .channel(scriptParameters.channel)
        .requestData(sessionData.requestData)
        .build()
    def ussdHandler = new UssdHandler(mobilePhone, runAs, binding)

    def page = binding.pathVariables.path ?: Pages.MAIN_MENU
    def message = ""
    if (page != Pages.MAIN_MENU && ussdHandler.newSession) {
        // When there is a new session in a page that is not the main menu,
        // assume the session has expired
        message = scriptParameters['general.sessionExpired']
        page = Pages.MAIN_MENU
    }

    // Invoke the UssdHandler method
    invokerHandler.runAs(runAs) {
        xml.page(version: "2.0") {
            ussdHandler."${page}"(delegate, message)
        }
    }
}

// Now the output stringWriter should contain the XML output. Build the response.
def response = new ResponseInfo(status: 200, stringBody: stringWriter.toString())
response.setHeader("Content-Type", "application/xml; charset=UTF-8")
return response

```

Create the custom web service

Under System > Tools > Custom web services, create a new one, with the following characteristics:

- Name: USSD
- Http method: GET
- Run as: Guest
- Script: USSD web service

- Script parameters: leave empty
- Url mappings: ussd/{path}

In order to provide security in production environment, you need to set a IP Whilelist checking the IP address whitelist box.

Enable a Global USSD account

You need an account at Global USSD with credits to be able to process USSD requests. To do so:

- Go to <https://account.globalussd.com/#register> and create an account.
- Login with your Global USSD account, go to the "Balance" option and add money to your wallet. Only accounts with credits will be able to operate via USSD.

Then you will need to configure a service, which can be found in the "Services" option. Create one with the following fields:

- Name: (fill in a name)
- Service URI: http(s) <cyclos_network_url>/run/ussd/mainMenu
- Content request HTTP-method: GET
- Take note of the "Push URL" value. It will be used to push a new USSD session (it would be of type `http://prod.globalussd.mobi/push?service=<BotID>&subscriber=<MSISDN>`).

Start an USSD session

Finally, assuming there is a user with a given mobile phone number, you can use the "push URL" shown in the Global USSD bot service page to start a session. Just perform a request to that URL, replacing the MSISDN text by the international mobile phone number, and BotID text with Bot service ID. Assuming the mobile phone's provider is supported by Global USSD, the user should see the USSD menu in his mobile phone.

Record edition

This solution lets you edit records using custom operations so you can do it via Web services or from the Mobile app where this functionality is not supported. However, it could be used as a guide for other entities.

The logged user is able to search its own records, update or delete them and create new ones (i.e CRUD operations).

It is also possible to search records of other users (you can not do it directly in Cyclos) and run actions over them, in this example we created an action that simulates send an email to the record's owner (you can implement anything you like).

To configure this, follow carefully each of the following steps:

Create the user record type to work with and give permissions

Under System > System configuration > Record types, create a new User record type, with the following characteristics:

- Name: Daily note
- Internal name: dailyNote (must be exactly like this)
- Display style: List
- Show in Menu: yes

For this record type, create the following fields:

Title

- Internal name: title (must be exactly like this)
- Data type: Single line text
- Required: no
- Show in results: yes

Description

- Internal name: description (must be exactly like this)
- Data type: Multiple line text
- Required: no
- Show in results: yes

Now give permissions:

to admin group (Groups > "Your admin group" > Permissions): in Records, check Enable, View, Create, Edit and Remove over "Daily note".

to user group (Groups > "The corresponding product"): in Records, check Enable, View, Create, Edit and Remove over "Daily note".

Create the library script

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: Record search library
- Type: Library
- Script code executed when the custom operation is executed:

```
import org.cyclos.model.system.fields.CustomFieldType
import org.cyclos.model.system.fields.ICustomFieldValue
import org.cyclos.model.system.scripts.CustomScriptException
import org.cyclos.model.users.records.UserRecordVO
```

```

import groovy.transform.TypeChecked

@TypeChecked
class Styles {
    static String infoBox = "white-space: normal; text-overflow: ellipsis;"
    static String fieldContainerLabel = "white-space: normal; text-overflow: ellipsis;
font-weight: 400; font-size: 15px; color: #1865a3;"
    static String fieldContainerValue = "white-space: normal; text-overflow: ellipsis;
margin: 2px 0 9px 0; font-size: 16px; line-height: 18px;"
    static String infoBoxInline = "white-space: normal; text-overflow: ellipsis;"
    static String fieldContainerLabelInline = "white-space: normal; text-overflow:
ellipsis; font-weight: 400; font-size: 15px; color: #1865a3;"
    static String fieldContainerValueInline = "white-space: normal; text-overflow:
ellipsis; margin: 2px 0 9px 0; font-size: 16px; line-height: 18px;"
}

@TypeChecked
class Helper {

    static Object getValue(ICustomFieldValue fieldValue) {
        if (!fieldValue) {
            return null;
        }
        switch (fieldValue.getField().getType()) {
            case CustomFieldType.BOOLEAN:
                return fieldValue.getBooleanValue()
            case CustomFieldType.DATE:
                return fieldValue.getDateValue()
            case CustomFieldType.DECIMAL:
                return fieldValue.getDecimalValue()
            case CustomFieldType.INTEGER:
                return fieldValue.getIntegerValue()
            case CustomFieldType.RICH_TEXT:
                return fieldValue.getRichTextValue()
            case CustomFieldType.STRING:
                return fieldValue.getStringValue()
            case CustomFieldType.TEXT:
                return fieldValue.getTextValue()
            case CustomFieldType.URL:
                return fieldValue.getStringValue()
            case CustomFieldType.IMAGE:
            case CustomFieldType.FILE:
            case CustomFieldType.LINKED_ENTITY:
            case CustomFieldType.MULTI_SELECTION:
            case CustomFieldType.SINGLE_SELECTION:
            case CustomFieldType.DYNAMIC_SELECTION:
                throw new CustomScriptException("""There was an error searching records.
Please, contact the administration""")
            default:
                throw new CustomScriptException("""Error searching records:
Unimplemented custom field type. Please, contact the administration""")
        }
    }

    static Object getCustomValueByInternalName(UserRecordVO record, String internalName){
        def value = record.getCustomValues().stream()
            .filter{ it.getField().getInternalName().equals(internalName) }
    }
}

```

```

        .findAny()
    }
    return getValue(value.orElse(null))
}

```

Create the custom operation script to send the record email

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: Send record email
- Type: Custom operation
- Script code executed when the custom operation is executed:

```
"The email was sent"
```

Create the custom operation script to search records

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: Search records
- Type: Custom operation
- Run with all permissions: yes
- Included libraries: Record search library
- Parameters:

```

recordType=dailyNote
titleInternalName=title
descInternalName=description

```

- Script code executed when the custom operation is executed:

```

import org.cyclos.entities.users.User
import org.cyclos.impl.users.RecordServiceLocal
import org.cyclos.impl.utils.formatting.FormatterImpl
import org.cyclos.model.users.records.UserRecordQuery
import org.cyclos.model.users.records.UserRecordVO
import org.cyclos.model.users.recordtypes.RecordTypeVO
import org.cyclos.model.users.users.UserVO

import groovy.transform.TypeChecked

@TypeChecked
class RecordBean {
    Long recordId
    String title
    String description
    String user

    RecordBean(Long id, String title, String description, String user) {
        this.recordId = id
        this.title = title;
    }
}

```

```

        this.description = description
        this.user = user
    }
}

@TypeChecked
RecordBean toRecordBean(UserRecordVO recordVO, String title, String description) {
    def titleRow = Helper.getCustomValueByInternalName(recordVO, title) as String
    def descRow = Helper.getCustomValueByInternalName(recordVO, description) as String
    def userRow = (binding.variables.formatter as FormatterImpl).format(recordVO.getUser())
    return new RecordBean(recordVO.id, titleRow, descRow, userRow)
}

@TypeChecked
def searchRecords() {
    def variables = binding.variables
    def scriptParameters = variables.scriptParameters as Map<String, Object>
    def formParameters = variables.formParameters as Map<String, Object>
    User user = variables.user as User
    RecordServiceLocal recordService = variables.recordService as RecordServiceLocal
    def type = scriptParameters.recordType as String
    def title = scriptParameters.titleInternalName as String
    def description = scriptParameters.descInternalName as String
    def query = new UserRecordQuery()
    if (formParameters.searchOnlyInMyRecords) {
        query.user = new UserVO(user.id)
    }
    query.currentPage = variables.currentPage as Integer
    query.pageSize = variables.pageSize as Integer
    query.skipTotalCount = true == variables.skipTotalCount
    query.type = new RecordTypeVO(RecordTypeVO.INTERNAL_NAME, type)
    query.keywords = formParameters.keywords as String

    def page = recordService.search(query)
    def rows = page.pageItems.stream().collect {
        toRecordBean(it as UserRecordVO, title, description)
    }

    return [
        columns: [
            [header: "Owner", property: "user", width: "15%"],
            [header: "Title", property: "title", width: "35%"],
            [header: "Description", property: "description", width: "50%"]
        ],
        rows: rows,
        totalCount: page.totalCount,
        hasNextPage: page.hasNextPage
    ]
}

return searchRecords()

```

Create the custom operation script to create records

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: Create record

- Type: Custom operation
- Run with all permissions: yes
- Parameters:

```
recordType=dailyNote
titleInternalName=title
descInternalName=description
```

- Script code executed when the custom operation is executed:

```
import org.cyclos.impl.users.RecordServiceLocal
import org.cyclos.model.system.fields.CustomFieldValueDTO
import org.cyclos.model.users.records.RecordDataParams
import org.cyclos.model.users.recordtypes.RecordTypeVO
import org.cyclos.model.utils.NotificationLevel

import groovy.transform.TypeChecked

@TypeChecked
def createRecord() {
    def msg = "The record was created successfully."
    def error = false

    try {
        def variables = binding.variables
        def scriptParameters = variables.scriptParameters as Map<String, Object>
        RecordServiceLocal recordService = variables.recordService as RecordServiceLocal
        def formParameters = variables.formParameters as Map<String, Object>
        def titleField = scriptParameters.titleInternalName
        def descField = scriptParameters.descInternalName
        def type = scriptParameters.recordType
        def dataParams = new RecordDataParams()
        dataParams.setRecordType(new RecordTypeVO(RecordTypeVO.INTERNAL_NAME, type))
        def data = recordService.getDataForNew(dataParams)
        def newRecord = data.getDto()
        def customFieldValues = new ArrayList<CustomFieldValueDTO>()

        data.getFields().stream()
            .filter {
                data.getEditableFieldIds().contains(it.id) &&
                (it.getInternalName().equals(titleField) ||
                 it.getInternalName().equals(descField))
            }
            .forEach {
                def valueDTO = new CustomFieldValueDTO();
                valueDTO.setField(it)
                if (it.getInternalName().equals(titleField)) {
                    valueDTO.setStringValue(formParameters.title as String)
                } else {
                    valueDTO.setTextValue(formParameters.description as String)
                }
                customFieldValues.add(valueDTO)
            }

        newRecord.setCustomValues(customFieldValues)
        recordService.save(newRecord)
    }
}
```

```

    } catch (Exception ex) {
        error = true
        msg = """There was an error trying to create the record.
            Please, contact the administration."""
    }

    return [
        notification: msg,
        notificationLevel: error ? NotificationLevel.ERROR :
NotificationLevel.INFORMATION,
        backTo: error ? null : "searchRecords",
        reRun: !error
    ]
}

return createRecord()

```

Create the custom operation script to view records

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: View record
- Type: Custom operation
- Run with all permissions: yes
- Included libraries: Record search library
- Script code executed when the custom operation is executed:

```

import org.cyclos.entities.users.RecordCustomFieldValue
import org.cyclos.entities.users.User
import org.cyclos.entities.users.UserRecord
import org.cyclos.entities.users.UserRecordType
import org.cyclos.impl.access.SessionData
import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.users.RecordFieldHandler
import org.cyclos.impl.users.RecordServiceLocal
import org.cyclos.impl.utils.formatting.FormatterImpl

import groovy.transform.Field
import groovy.transform.TypeChecked
import groovy.xml.MarkupBuilder

@Field Map<String, Object> variables = binding.variables

@TypeChecked
class RecordView{
    UserRecordType type
    User user
    String formattedCreationDate
    String formattedLastModifiedDate
    User modifiedBy
    User createdBy

```

```

Map<String, String> customValues

RecordView(UserRecordType type, User user, String formattedCreationDate,
String formattedLastModifiedDate, User modifiedBy, User createdBy,
Map<String, String> customValues) {
    this.type = type
    this.user = user
    this.formattedCreationDate = formattedCreationDate
    this.formattedLastModifiedDate = formattedLastModifiedDate
    this.createdBy = createdBy
    this.modifiedBy = modifiedBy
    this.customValues = customValues
}
}

@TypeChecked
def RecordView toRecordView(UserRecord record) {
    FormatterImpl formatter = variables.formatter as FormatterImpl
    Map<String, String> customValues = [:]
    def recordFieldValues = new ArrayList<RecordCustomFieldValue>(record.customValues)
    def recordFieldHandler = variables.recordFieldHandler as RecordFieldHandler
    recordFieldHandler.sortFieldValues(record.type, recordFieldValues)
    recordFieldValues.each {
        def value = Helper.getValue(it)
        if (value != null) {
            customValues[it.field.name] = formatter.format(value)
        }
    }
    String formattedCreationDate = formatter.format(record.creationDate)
    String formattedLastModifiedDate = formatter.format(record.lastModifiedDate)
    return new RecordView(record.type as UserRecordType, record.user,
        formattedCreationDate, formattedLastModifiedDate, record.createdBy,
        record.modifiedBy, customValues)
}

def createContent(StringWriter out, RecordView record) {
    def html = new MarkupBuilder(out)
    html.div(style:"${Styles.infoBox}") {
        div {
            div(style:"${Styles.fieldContainerLabel}") { mkp.yield "Owner:" }
            div(style:"${Styles.fieldContainerValue}") {
                mkp.yield formatter.format(record.user)
            }
        }
        if (record.type.isShowUpdateToUsers()) {
            div {
                div(style:"${Styles.fieldContainerLabel}") { mkp.yield "Created by:" }
                div(style:"${Styles.fieldContainerValue}") {
                    mkp.yield record.createdBy?.display
                }
            }
            div {
                div(style:"${Styles.fieldContainerLabel}") { mkp.yield "Creation date:" }
                div(style:"${Styles.fieldContainerValue}") {
                    mkp.yield record.formattedCreationDate
                }
            }
        }
        if (!record.formattedLastModifiedDate.isEmpty()) {

```

```

        div {
            div(style:"${Styles.fieldContainerLabel}") {
                mkp.yield "Last modification date:"
            }
            div(style:"${Styles.fieldContainerValue}") {
                mkp.yield record.formattedLastModifiedDate
            }
        }
        div {
            div(style:"${Styles.fieldContainerLabel}") {
                mkp.yield "Last modification by:"
            }
            div(style:"${Styles.fieldContainerValue}") {
                mkp.yield record.modifiedBy?.display
            }
        }
    }
}
}
record.customValues.each{ name, val ->
    div {
        div(style:"${Styles.fieldContainerLabel}") { mkp.yield name }
        div(style:"${Styles.fieldContainerValue}") { mkp.yield val }
    }
}
}
}

@TypeChecked
def viewRecord() {
    def sessionData = variables.sessionData as SessionData
    def formParameters = variables.formParameters as Map<String, Object>
    RecordServiceLocal recordService = variables.recordService as RecordServiceLocal
    ScriptHelper scriptHelper = variables.scriptHelper as ScriptHelper
    def id = scriptHelper.unmaskId(formParameters.recordId)
    def record = recordService.find(id) as UserRecord
    def out = new StringWriter()
    createContent(out, toRecordView(record))

    if (record.user == sessionData.loggedBasicUser) {
        return [
            content: out.toString(),
            actions: [
                sendRecordEmail: [
                    enabled: false
                ]
            ]
        ]
    } else {
        return [
            content: out.toString(),
            actions: [
                removeRecord: [
                    enabled: false
                ],
                updateRecord: [
                    enabled: false
                ]
            ]
        ]
    }
}

```

```

    }
}

return viewRecord()

```

Create the custom operation script to update records

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: Update record
- Type: Custom operation
- Run with all permissions: yes
- Parameters:

```

titleInternalName=title
descInternalName=description

```

- Script code executed when the custom operation is executed:

```

import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.users.RecordServiceLocal
import org.cyclos.model.system.fields.CustomFieldValueDTO
import org.cyclos.model.users.records.RecordData
import org.cyclos.model.users.recordtypes.RecordCustomFieldDetailedVO
import org.cyclos.model.utils.NotificationLevel

import groovy.transform.TypeChecked

@TypeChecked
def updateRecord() {

    def variables = binding.variables
    def scriptParameters = variables.scriptParameters as Map<String, Object>
    def formParameters = variables.formParameters as Map<String, Object>
    RecordServiceLocal recordService = variables.recordService as RecordServiceLocal
    ScriptHelper scriptHelper = variables.scriptHelper as ScriptHelper
    def msg = "The record was updated successfully."
    def error = false

    try{
        def titleField = scriptParameters.titleInternalName as String
        def descField = scriptParameters.descInternalName as String
        def newDesc = formParameters.newDescription as String
        def newTitle = formParameters.newTitle as String
        def id = scriptHelper.unmaskId(formParameters.recordId)
        def data = recordService.getData(id) as RecordData
        def newRecord = data.dto
        def fieldValues = [
            titleField: null,
            descField: null
        ] as Map<String, CustomFieldValueDTO>

        newRecord.getCustomValues().each{

```

```

        if (it.field.internalName == titleField) {
            fieldValues.put(titleField, it)
        } else if (it.field.internalName == descField) {
            fieldValues.put(descField, it)
        }
    }

    fieldValues.each { k, v ->
        def isTitle = titleField == k
        def value = isTitle ? newTitle : newDesc
        if (v) {
            if (isTitle) {
                v.stringValue = value
            } else {
                v.textValue = value
            }
        } else {
            data.fields.findAll { RecordCustomFieldDetailedVO it ->
                it.internalName == k &&
                data.editableFieldIds.contains(it.id)
            }.each { RecordCustomFieldDetailedVO it ->
                def valueDTO = new CustomFieldValueDTO();
                valueDTO.field = it
                if (isTitle) {
                    valueDTO.stringValue = value
                } else {
                    valueDTO.textValue = value
                }
                newRecord.customValues << valueDTO
            }
        }
    }

    recordService.save(newRecord)
} catch (Exception ex) {
    error = true
    msg = """There was an error trying to update the record.
        Please, contact the administration."""
}

return [
    notification: msg,
    notificationLevel: error ? NotificationLevel.ERROR :
NotificationLevel.INFORMATION,
    backTo: error ? null : "recordDetails",
    reRun: !error
]
}

return updateRecord()

```

- Script code executed before the form is show, to fill the initial field values:

```

import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.users.RecordServiceLocal
import org.cyclos.utils.StringHelper

import groovy.transform.TypeChecked

```

```

@TypeChecked
def loadRecordFields() {
    def variables = binding.variables
    def scriptParameters = variables.scriptParameters as Map<String, Object>
    def formParameters = variables.formParameters as Map<String, Object>
    RecordServiceLocal recordService = variables.recordService as RecordServiceLocal
    ScriptHelper scriptHelper = variables.scriptHelper as ScriptHelper
    def titleField = scriptParameters.titleInternalName as String
    def descField = scriptParameters.descInternalName as String
    def id = scriptHelper.unmaskId(formParameters.recordId)
    def newRecord = recordService.getData(id).dto

    def titleFieldValue = newRecord.customValues.find {
        it.field.internalName.equals(titleField)
    }
    def descFieldValue = newRecord.customValues.find {
        it.field.internalName.equals(descField)
    }

    return [
        newTitle: StringHelper.emptyIfNull(titleFieldValue?.stringValue),
        newDescription: StringHelper.emptyIfNull(descFieldValue?.textValue)
    ]
}

return loadRecordFields()

```

Create the custom operation script to remove records

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: Remove record
- Type: Custom operation
- Run with all permissions: yes
- Script code executed when the custom operation is executed:

```

import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.users.RecordServiceLocal
import org.cyclos.model.utils.NotificationLevel

import groovy.transform.TypeChecked

@TypeChecked
def removeRecord() {
    def msg = "The record was removed successfully."
    def error = false

    def variables = binding.variables
    RecordServiceLocal recordService = variables.recordService as RecordServiceLocal
    ScriptHelper scriptHelper = variables.scriptHelper as ScriptHelper
    def formParameters = variables.formParameters as Map<String, Object>
    try {
        recordService.remove(scriptHelper.unmaskId(formParameters.recordId))
    }
}

```

```

    } catch(Exception ex) {
        error = true
        msg = """There was an error trying to remove the record.
            Please, contact the administration."""
    }

    return [
        notification: msg,
        notificationLevel: error
        ? NotificationLevel.ERROR : NotificationLevel.INFORMATION,
        backTo: error ? null : "searchRecords",
        reRun: !error
    ]
}

return removeRecord()

```

Create the custom operation to remove records

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: Remove record
- Internal name: removeRecord (must be exactly like this)
- Enable for channels: Main, Web services, Mobile app
- Custom submit label: Remove (can be changed - will be the label displayed in the action button)
- Scope: Internal
- Script: Remove record
- Result type: Notification
- Custom script execute message: This record will be removed. Do you want to continue? (can be edited - it is necessary to alert the user that the record is going to be removed)

Once saved, on the Form fields tab, create a new field, with the following characteristics:

Record id:

- Display name: Record id
- Internal name: recordId (must be exactly like this)
- Data type: Single line text

Create the custom operation to update records

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: Update record

- Internal name: updateRecord (must be exactly like this)
- Enable for channels: Main, Web services, Mobile app
- Custom submit label: Update (can be changed - will be the label displayed in the action button)
- Scope: Internal
- Script: Update record
- Result type: Notification
- Show form: Always

Once saved, on the Form fields tab, create three new fields, with the following characteristics:

Record id:

- Display name: Record id
- Internal name: recordId (must be exactly like this)
- Data type: Single line text

Title:

- Display name: Title
- Internal name: newTitle (must be exactly like this)
- Data type: Single line text

Description:

- Display name: Description
- Internal name: newDescription (must be exactly like this)
- Data type: Multiple line text

Create the custom operation to send the record email

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: Send record email
- Internal name: sendRecordEmail (must be exactly like this)
- Enable for channels: Main, Web services, Mobile app
- Custom submit label: Send email (can be changed - will be the label displayed in the action button)
- Scope: Internal
- Script: Send record email

- Result type: Notification

Create the custom operation to view records details

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: Record details
- Internal name: recordDetails (must be exactly like this)
- Enable for channels: Main, Web services, Mobile app
- Scope: Internal
- Script: View record
- Result type: Rich text

Once saved, on the Form fields tab, create a new field, with the following characteristics:

Record id:

- Display name: Record id
- Internal name: recordId (must be exactly like this)
- Data type: Single line text

Once saved, on the Actions tab, add three new actions, with the following characteristics:

- Operation: Remove record
- Parameters:
 - Record id: "Record id"
- Operation: Update record
- Parameters:
 - Record id: "Record id"
 - Title: Not used
 - Description: Not used
- Operation: Send record email

Create the custom operation to create records

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: Create record
- Internal name: createRecord (must be exactly like this)

- Enable for channels: Main, Web services, Mobile app
- Label: New (can be changed - will be the label displayed in the action button)
- Scope: Internal
- Script: Create record
- Result type: Notification
- Show form: Always

Once saved, on the Form fields tab, create two new fields, with the following characteristics:

Title:

- Display name: Title
- Internal name: title (must be exactly like this)
- Data type: Single line text

Description:

- Display name: Description
- Internal name: description (must be exactly like this)
- Data type: Multiple line text

Create the custom operation to search records

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: Daily notes (can be changed - will be the label displayed on the menu)
- Internal name: searchRecords (must be exactly like this)
- Enable for channels: Main, Web services, Mobile app
- Scope: User
- Script: Search records
- Result type: Result page
- Allow printing results: yes
- Allow exporting results to CSV: yes
- Action when clicking a row: Run an internal custom operation
- Custom operation: Record details
- Parameters to be passed (comma-separated names): recordId
- Main menu: Personal (can be changed - will be the menu where the label is going to be displayed)

- User management section: User management (can be changed - will be the section where the label is going to be displayed)
- Enable for active users: yes

Once saved, on the Form fields tab, create two new fields, with the following characteristics:

Keywords:

- Display name: Keywords
- Internal name: keywords (must be exactly like this)
- Data type: Single line text

Search only in my records:

- Display name: Search only in my records
- Internal name: searchOnlyInMyRecords (must be exactly like this)
- Data type: Boolean

Once saved, on the Actions tab, add a new action, with the following characteristics:

- Operation: Create record
- Visibility: Before and after run the custom operation
- Parameters:
 - Title: Not used
 - Description: Not used

Enable the custom operation for users

In System > User configuration > Products (permissions), select the member product for users which will be able to work with this operation. In the Custom operations field, make sure the Daily notes is both enabled and allowed to run over self.

User balances

This solution provides a custom operator for administrators to search the users' balances with 2 advantages over the regular user balances overview in Cyclos:

- It is possible to select a date, so all presented balances will be for that date;
- The available balances are also shown, but only if no date is set (as credit limit could have changed over time).

These options are not available in the regular balances search because they need to be calculated per user, whereas the current balance is stored in the database. This makes the script inviable when there are too many users. Still, some systems require this functionality.

However, as a drawback, the filters for users are also more limited in this script: it is only possible to filter by a specific group. So, if only the current balance is desired, it is advised to use the built-in functionality instead.

To configure this functionality, follow carefully each of the following steps:

Create the script to load user account types

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: User account types loader
- Type: Load custom field values
- Script code that returns the possible values when either creating or editing an entity:

```
import java.util.stream.Collectors

import org.cyclos.impl.banking.AccountTypeServiceLocal
import org.cyclos.model.banking.accounttypes.AccountTypeNature
import org.cyclos.model.system.fields.DynamicFieldValueVO

import groovy.transform.TypeChecked

@TypeChecked
def loadUserAccountTypes() {
    def variables = binding.variables
    def accountTypeService = variables.accountTypeService as AccountTypeServiceLocal
    return accountTypeService
        .listAllAccessible()
        .stream()
        .filter { it.getNature() == AccountTypeNature.USER }
        .map { new DynamicFieldValueVO(String.valueOf(it.id), it.name) }
        .collect(Collectors.toList())
}

loadUserAccountTypes()
```

Create the script to load user groups

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: User groups loader
- Type: Load custom field values
- Script code that returns the possible values when either creating or editing an entity:

```
import java.util.stream.Collectors

import org.cyclos.impl.access.SessionData
import org.cyclos.impl.users.GroupsHandler
import org.cyclos.model.system.fields.DynamicFieldValueVO

import groovy.transform.TypeChecked
```

```

@TypeChecked
def loadUserGroups() {
    def variables = binding.variables
    def groupsHandler = variables.groupsHandler as GroupsHandler
    def sessionData = variables.sessionData as SessionData

    return groupsHandler
        .getAccessibleUserGroups(sessionData.getLoggedUser())
        .stream()
        .map { new DynamicFieldValueVO(String.valueOf(it.id), it.name) }
        .collect(Collectors.toList())
}

loadUserGroups()

```

Create the custom operation script to search users balances

Under System > Tools > Scripts, create the next script, with the following characteristics:

- Name: User balances
- Type: Custom operation
- Run with all permissions: yes
- Script code executed when the custom operation is executed:

```

import java.util.stream.Collectors

import org.cyclos.entities.banking.Account
import org.cyclos.entities.banking.QAccount
import org.cyclos.entities.system.ExportFormat
import org.cyclos.entities.users.QGroup
import org.cyclos.entities.users.QUser
import org.cyclos.entities.users.UserCustomField
import org.cyclos.impl.InvocationContext
import org.cyclos.impl.access.SessionData
import org.cyclos.impl.banking.AccountServiceLocal
import org.cyclos.impl.contentmanagement.TranslationHandler
import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.users.UserCustomFieldServiceLocal
import org.cyclos.impl.utils.formatting.FormatterImpl
import org.cyclos.impl.utils.persistence.EntityManagerHandler
import org.cyclos.model.banking.BankingKeys
import org.cyclos.model.system.fields.DynamicFieldValueVO
import org.cyclos.model.users.UsersKeys
import org.cyclos.server.utils.DateHelper

import com.querydsl.core.types.Expression
import com.querydsl.core.types.dsl.Expressions

import groovy.transform.TypeChecked

@TypeChecked
def searchBalances(){
    def variables = binding.variables

```

```

def entityManagerHandler = variables.entityManagerHandler as EntityManagerHandler
def translationHandler = variables.translationHandler as TranslationHandler
def accountService = variables.accountService as AccountServiceLocal
def formatter = variables.formatter as FormatterImpl
def exportFormat = variables.exportFormat as ExportFormat
def userCustomFieldService = variables.userCustomFieldService as
UserCustomFieldServiceLocal
def scriptHelper = variables.scriptHelper as ScriptHelper

def formParameters = variables.formParameters as Map<String, Object>
def sessionData = variables.sessionData as SessionData

def a = QAccount.account
def u = QUser.user
def g = QGroup.group

def query = entityManagerHandler
    .from(a)
    .innerJoin(u).on(a.user.eq(u))
    .innerJoin(g).on(u.group.eq(g))

def group = (formParameters.group as DynamicFieldValueVO)?.value
if (group) {
    query.where(g.id.eq(Long.valueOf(group)))
}

def accountType = (formParameters.accountType as DynamicFieldValueVO)?.value
if (accountType) {
    query.where(a.type().id.eq(Long.valueOf(accountType)))
}

def currentPage = variables.currentPage as int
def pageSize = variables.pageSize as int
query
    .limit(pageSize)
    .offset(pageSize * currentPage)
    .orderBy(u.name.asc(), a.type().name.asc())

def totalCount = variables.skipTotalCount ? null : query.fetchCount()

def balanceExpression
def expressions = [
    a.id,
    a.user().displayForManagers,
    a.type().name,
    a.type().currencyId
] as List<Expression>
def date = formParameters.date as Date
if (date) {
    query.where(a.creationDate.before(date))
    def timeZone = sessionData.getConfiguration().getTimeZone()
    date = DateHelper.shiftToEnd(date, timeZone)
    balanceExpression = a.balance(Expressions.constant(date))
    expressions << balanceExpression
}

List<UserCustomField> customFields = []
if (exportFormat && exportFormat.internalName != 'pdf') {

```

```

    // When exporting tabular data, include the profile fields
    customFields = userCustomFieldService.listAll().findAll() { it.includeInExport }
}

def cacheFlusher = InvocationContext.newCacheFlusher()
def rows = query.stream(expressions as Expression[]).map {
    def result = [
        id: it.get(a.id),
        display: it.get(a.user()).displayForManagers,
        accountType: it.get(a.type().name),
        currency: it.get(a.type().currencyId)
    ] as Map<String, Object>
    if (date == null) {
        // We need to fetch the available balance
        def account = entityManagerHandler.find(Account, it.get(a.id))
        def status = accountService.getAccountStatus(account, null, null)
        result.balance = status.balance
        result.availableBalance = status.availableBalance
    } else {
        result.balance = it.get(balanceExpression)
    }
    if (!customFields.empty) {
        def account = entityManagerHandler.find(Account, it.get(a.id))
        def fields = scriptHelper.wrap(account.owner, customFields)
        customFields.each {
            def value = fields[it.internalName]
            result[it.internalName] = value instanceof Date ?
                formatter.formatAsDate(value) : formatter.format(value)
        }
    }
    cacheFlusher.flush()
    return result
}.collect(Collectors.toList())

def columns = [
    [
        header: translationHandler.message(UsersKeys.Users.USER),
        property: "display"
    ],
    [
        header: translationHandler.message(BankingKeys.Accounts.TYPE),
        property: "accountType"
    ],
    [
        header: translationHandler.message(BankingKeys.Accounts.BALANCE),
        property: "balance",
        currencyProperty: "currency",
        align: "right"
    ]
]
if (date == null) {
    columns << [
        header: translationHandler.message(BankingKeys.Accounts.AVAILABLE_BALANCE),
        property: "availableBalance",
        currencyProperty: "currency",
        align: "right"
    ]
}
}

```



```

    customFields.each {
        columns << [ header: it.name, property: it.internalName ]
    }
    return [
        columns: columns,
        rows: rows,
        totalCount: totalCount,
        currentPage: currentPage
    ]
}

searchBalances()

```

Create the custom operation to search users balances

Under System > Tools > Custom operations, create a new one with the following characteristics:

- Name: User balances (can be changed - will be the label displayed on the menu)
- Internal name: userBalances
- Enable for channels: Main, Web services, Mobile app
- Scope: System
- Script: User balances
- Result type: Result page
- Search automatically on page load: no
- Allow printing results: yes
- Allow exporting results to CSV: yes
- Action when clicking a row: Navigate to a Cyclos location
- Location: user_profile
- Parameters to be passed (comma-separated names): id
- Main menu: Banking (can be changed - will be the menu where the label is going to be displayed)

Once saved, on the Form fields tab, create three new fields, with the following characteristics:

Account type:

- Display name: Account type (can be changed as desired)
- Internal name: accountType (must be exactly like this)
- Data type: Dynamic selection
- All selected label: All (can be changed as desired)
- Load values script: User account types loader

Group:

- Display name: Group (can be changed as desired)
- Internal name: group (must be exactly like this)
- Data type: Dynamic selection
- All selected label: All (can be changed as desired)
- Load values script: User groups loader

Date:

- Display name: Date (can be changed as desired)
- Internal name: date (must be exactly like this)
- Data type: Date

Enable the custom operation

Enable this operation in the administrators group, in the Permissions tab, under 'Run system custom operations'.

4.5. Running scripts directly

In many occasions it is handy for administrators to run scripts directly. So, instead of having to create a custom operation script, then a custom operation, then granting permissions, refreshing the browser and running, there is a menu called Run script, which presents a text box where the script may be typed in or pasted, which can be executed directly. Of course, only the [basic bindings](#) are available.

The result of the script can be either a string, which is then displayed as plain text, or an object / map compatible with [org.cyclos.model.system.scripts.ScriptResult](#). So, for example, to return an HTML text with a title, the script can return [title:"The result title", richText:"Formatted text"]. To show a notification, the the script can return [notification:"Notification text"]. The same prefixes available on notifications for [custom operations](#) are available on notifications: [INFO], [WARN] and [ERROR].

Examples

Remove all users, transactions and related data

Here is an example of a script to remove all regular users (not administrators) and related data, as well as all system to system transactions. When running in a network, will remove data on that network. When running in global mode, will affect the entire system. Be advised that there will be no confirmation, and all users and all related data will be removed.

The script works by first recreating all database constraints with the option ON DELETE CASCADE. Then, all users are removed, which will cascade the removal to accounts, transfers, advertisements, records, messages, notifications, references and so on. Be careful when running in systems where specific users are used in the configuration, such as fees that are paid by a specific user, or payment types which are restricted to specific users. In such configurations, all such related data will be removed as well. Also, note that it may take a while to run, so, please, wait before the script completes.

AGAIN: be very careful when using this script! Only run it on test instances and always have a database backup before running it.

```
import java.sql.Connection

import org.cyclos.impl.InvokerHandler
import org.cyclos.impl.access.SessionData
import org.cyclos.impl.search.SearchHandler
import org.cyclos.impl.system.ScriptHelper
import org.cyclos.impl.utils.cache.CacheHandler
import org.cyclos.impl.utils.cache.CacheType
import org.springframework.jdbc.core.ConnectionCallback
import org.springframework.jdbc.core.JdbcTemplate

import groovy.sql.GroovyRowResult
import groovy.sql.Sql
import groovy.transform.Field
import groovy.transform.TypeChecked

@Field SessionData sessionData = binding.sessionData
@Field ScriptHelper scriptHelper = binding.scriptHelper
@Field CacheHandler cacheHandler = binding.cacheHandler
@Field SearchHandler searchHandler = binding.searchHandler
@Field InvokerHandler invokerHandler = binding.invokerHandler
@Field JdbcTemplate jdbc = binding.jdbcTemplate
@Field Connection connection = jdbc.execute({ it } as ConnectionCallback)
@Field Sql sql = new Sql(connection)
@Field Long networkId = sessionData.network?.id

@Field List<GroovyRowResult> constraints = sql.rows("""
SELECT tc.table_name, kcu.column_name, tc.constraint_name,
       ccu.table_name AS foreign_table_name,
       ccu.column_name AS foreign_column_name
FROM
  information_schema.table_constraints AS tc
  JOIN information_schema.key_column_usage AS kcu
    ON tc.constraint_name = kcu.constraint_name
  JOIN information_schema.constraint_column_usage AS ccu
    ON ccu.constraint_name = tc.constraint_name
WHERE tc.constraint_type = 'FOREIGN KEY'
      AND tc.constraint_schema = 'public'
""").toString())

def recreateConstraints(boolean cascade) {
  constraints.forEach { r ->
```

```

        String drop = """alter table ${r.table_name}
            drop constraint ${r.constraint_name}"""
        String create = """alter table ${r.table_name}
            add constraint ${r.constraint_name}
            foreign key (${r.column_name})
            references ${r.foreign_table_name} (${r.foreign_column_name})"""
        if (cascade) {
            create += " on delete cascade"
        }
        sql.execute(drop)
        sql.execute(create)
    }
}

@Field String accountsSql = """select a.id
from accounts a
inner join account_types at on a.account_type_id = at.id
inner join currencies c on at.currency_id = c.id
"""
if (networkId != null) {
    accountsSql += " where c.network_id = ${networkId}"
}

@TypeChecked
def execute() {
    // Recreate constraints with on delete cascade
    recreateConstraints(true)

    // First break the cycle on accounts -> account_rates,
    // otherwise system accounts with rates are deleted as well
    sql.executeUpdate("update accounts set account_rates_id = null"
        + " where id in (" + accountsSql + ")")

    // Remove banking data. This will include system accounts as well
    def tablesAndColumns = [
        transfers: ['from_id', 'to_id'],
        transactions: ['from_id', 'to_id'],
        amount_reservations: ['account_id'],
        closed_account_balances: ['account_id'],
        account_balances: ['account_id'],
        dirty_account_balances: ['account_id'],
        account_rates: ['account_id']]
    tablesAndColumns.forEach { table, cols ->
        cols.forEach { col ->
            sql.executeUpdate("delete from " + table +
                " where " + col + " in (" + accountsSql + ")")
        }
    }

    // Remove users
    def removedUsers = sql.executeUpdate("""
        delete from users
        where (subclass = 'OPERATOR'
            or user_group_id in (
                select id from groups where subclass in ('MEMBER_GROUP', 'BROKER_GROUP')))
        """) + ((networkId == null) ? "" : " and network_id = ${networkId}")

    // Remove some other tables which would be just placeholders with empty data

```

```

sql.executeUpdate("delete from bulk_actions"
    + (networkId == null ? '' : " where network_id = ${networkId}")
sql.executeUpdate("delete from mailing_lists"
    + (networkId == null ? '' : " where network_id = ${networkId}")
sql.executeUpdate("delete from account_fee_logs"
    + (networkId == null ? '' : ""
        where account_fee_id in (
            select af.id
            from account_fees af
            inner join account_types at on af.account_type_id = at.id
            inner join currencies c on at.currency_id = c.id
            where c.network_id = ${networkId})""))

// Recreate constraints as initial
recreateConstraints(false)

return "Removed ${removedUsers} users"
}

def output = execute()

// Clear all caches
CacheType.all().each { cacheHandler.scheduleClear(it) }

// Reindex all data
searchHandler.reindex()

// Execute the script
return output

```

5. Custom alerts

From 4.12 onwards custom alerts can be defined for both system and user. This alerts will be created using [scripting](#).

It can be done in any kind of script just adding a few lines as the next examples show:

- System alert

```
import org.cyclos.impl.messaging.AlertServiceLocal
import org.cyclos.model.messaging.alerts.SystemAlertType

AlertServiceLocal alertService = binding.alertService
alertService.create(SystemAlertType.CUSTOM, /*Alert text*/ "This is a system alert example")
```

- User alert

```
import org.cyclos.impl.messaging.AlertServiceLocal
import org.cyclos.model.messaging.alerts.UserAlertType

AlertServiceLocal alertService = binding.alertService;
// Create the alert for the corresponding user, in this case we use the one logged.
alertService.create(sessionData.getLoggedUser(), UserAlertType.CUSTOM, /*Alert text*/ "This is a user alert example")
```

This type will appear as a new option in the administrators notification settings page under "User alerts" and "System alerts" preferences.

6. External login

Starting with Cyclos 4.2, using [web services](#) together with the right configuration, it is possible to add a Cyclos login form to an external website. The user types in his/hers Cyclos username and password in that form and, after a successful login, is redirected to Cyclos, where the session will be already valid, and the user can perform the operations as usual. After the user clicks logout, or his/hers session expires, the user is redirected back to the external website.

The following aspects should be considered:

- It is needed to have an administrator whose group is granted the permission "Login users via web services". This is needed because the website will relay logins from users their clients to Cyclos.
- The website needs to have that administrator's username and password configured in order to make the web services call. Otherwise you can configure and access client which will allow using a separated key instead of the username / password.
- It is a good practice to create a separated configuration for that administrator. That configuration should have an IP address whitelist for the web services channel. Doing that, no other server, even if the administrator username / password is known by someone else, will be able to perform such operations.
- The Cyclos configuration for users needs the following settings:
 - Redirect login to URL: This is the URL of the external website which contains the login form. This is used to redirect the user when his session expires and a new login is needed, or when the user navigates directly to some URL in Cyclos (as guest), in that case the external web site could receive a parameter named "returnTo" that must be sent back to Cyclos without any modification after a successful login;
 - URL to redirect after logout: This is the URL where the user will be redirected after clicking "Logout" in Cyclos. It might be the same URL as the one for redirect login, but not necessarily.
- Finally, the web service code needs to be created, and deployed to the website. Here is an example, which receives the username and password parameters, calls the web service to create a session for the user (passing his remote address), redirecting the user to Cyclos.

```
<?php

// Configure Cyclos and obtain an instance of LoginService
require_once 'configureCyclos.php';
$loginService = new Cyclos\LoginService();

// Set the parameters
$params = new stdClass();
$params->user = array("principal" => $_POST['username']);
$params->password = $_POST['password'];
```

```

$params->remoteAddress = $_SERVER['REMOTE_ADDR'];

// Perform the login
try {
    $result = $loginService->loginUser($params);
} catch (Cyclos\ConnectionException $e) {
    echo("Cyclos server couldn't be contacted");
    die();
} catch (Cyclos\ServiceException $e) {
    switch ($e->errorCode) {
        case 'VALIDATION':
            echo("Missing username / password");
            break;
        case 'LOGIN':
            echo("Invalid username / password");
            break;
        case 'REMOTE_ADDRESS_BLOCKED':
            echo("Your access is blocked by exceeding invalid login attempts");
            break;
        default:
            echo("Error while performing login: {$e->errorCode}");
            break;
    }
    die();
}

// Redirect the user to Cyclos with the returned session token
header("Location: "
    . Cyclos\Configuration::getRootUrl()
    . "?sessionToken="
    . $result->sessionToken);

?>

```

Important notes

- In case there is a wrong configuration for the "Redirect login to URL" setting, it won't be possible anymore to login to Cyclos. In that case, if the configuration problem is within a network, it is possible to use a global administrator to login in global mode (using the <server-root>/global/login URL), then switch to the network and fix the configuration. If the configuration error is in global mode, you can use a special URL to prevent redirect: <server-root>/global/login!noRedirect=true . However, this flag only works in global mode, to prevent end-users from using it to bypass the redirect.
- Users should never have username / password requested in a plain HTTP connection. Always use a secure (HTTPS) connection. Also, just having an iframe with the form on a secure page, where the iframe itself is displayed in a plain page would encrypt the traffic, but browsers won't show the page as secure. Users won't notice that page as secure, could refuse to provide credentials in such situation.

Creating an alternate frontend to Cyclos

It is possible to not only place a login form in an external website, but to create an entire frontend for users to interact with Cyclos. At first glimpse, this can be great, but consider the following:

- It is a very big effort to create a frontend, as there are several Cyclos services involved, and it might not be clear without a deep analysis on the [API](#) which service / method / parameters should be used on each case.
- The API will change. Even if we try not to break compatibility, it is possible that changes between 4.x to 4.y will contain (sometimes incompatible) [changes to the API](#).
- You will always have a limited subset of the functionality Cyclos offers. You may think that only the very basic features are needed, there will inevitably be the need for more features, and the custom frontend will need to grow. By using Cyclos standard web, all this comes automatically.

Nevertheless, some (large) organizations might find it is better to provide their users with a single, integrated interface. In that case the application server of that interface will be the only one interacting with Cyclos (i.e, users won't directly browse the Cyclos interface). The application will relay web service calls to Cyclos in behalf of users.

To accomplish that, it is needed to first login users in the same way as explained in the previous section. However, after the login is complete, instead of redirecting users to Cyclos, the application needs to store the session token, and probably the user id (as some operations requires passing the logged user id) – both data received after logging in – in a session (in the interface application server). Then, the next web service requests should be sent using that session token and client remote address, instead of the administrator credentials. The way of passing that data depends on the web service access type being used:

- Java clients: Create another [HttpServiceFactory](#), using a stateful [HttpServiceInvocationData](#). Here is an example:

```
import java.util.List;

import org.cyclos.model.access.LoggedOutException;
import org.cyclos.model.access.channels.BuiltInChannel;
import org.cyclos.model.access.login.UserAuthVO;
import org.cyclos.model.banking.accounts.AccountWithStatusVO;
import org.cyclos.model.users.users.UserLocatorVO;
import org.cyclos.model.users.users.UserLoginDTO;
import org.cyclos.model.users.users.UserLoginResult;
import org.cyclos.server.utils.HttpServiceFactory;
import org.cyclos.server.utils.HttpServiceInvocationData;
import org.cyclos.services.access.LoginService;
import org.cyclos.services.banking.AccountService;

/**
 * Cyclos web service example: logs-in a user via web services.
 * This is useful when creating an alternative front-end for Cyclos.
 */
public class LoginUser {
```

```

    public static void main(String[] args) throws Exception {
        // This LoginService has the administrator credentials
        LoginService loginService =
Cyclos.getServiceFactory().getProxy(LoginService.class);

        // Another option is to use an access client to connect with the
        // server (for the admin)
        // To make it works you must:
        // 1- create an access client
        // 2- assign it to the admin (to obtain the activation code)
        // 3- activate it making a HTTP POST to the server using this url:
        // ROOT_URL/activate-access-client containing only the activation code
        // as the body
        // 4- put the token returned from the servlet as the parameter of the
        // HttpServiceInvocationData.accessClient(...) method
        // 5- comment the first line (that using user and password and
        // uncomment the following two sentences

        // HttpServiceInvocationData adminSessionInvocationData =
        // HttpServiceInvocationData
        // .accessClient("put_the_token_here");
        // LoginService loginService = Cyclos.getServiceFactory(
        // adminSessionInvocationData).getProxy(LoginService.class);

        String remoteAddress = "192.168.1.200";

        // Set the login parameters
        UserLoginDTO params = new UserLoginDTO();
        params.setUser(new UserLocatorVO(UserLocatorVO.USERNAME, "some-user"));
        params.setPassword("1234");
        params.setRemoteAddress(remoteAddress);
        params.setChannel(BuiltInChannel.MAIN.getInternalName());

        // Login the user
        UserLoginResult result = loginService.loginUser(params);
        UserAuthVO userAuth = result.getUser();
        String sessionToken = result.getSessionToken();
        System.out.println("Logged-in '" + userAuth.getUser().getDisplay()
            + "' with session token = " + sessionToken);

        // Do something as user. As the session token is only valid per ip
        // address, we need to pass-in the client ip address again
        HttpServiceInvocationData sessionInvocationData =
            HttpServiceInvocationData.stateful(sessionToken, remoteAddress);
        // The services acquired by the following factory will carry on the
        // user session data
        HttpServiceFactory userFactory = Cyclos.getServiceFactory(sessionInvocationData);
        AccountService accountService = userFactory.getProxy(AccountService.class);
        List<AccountWithStatusVO> accounts =
            accountService.getAccountsSummary(userAuth.getUser(), null);
        for (AccountWithStatusVO account : accounts) {
            System.out.println(account.getType()
                + ", balance: " + account.getStatus().getBalance());
        }

        // Logout. There are 2 possibilities:

```

```

        // - Logout as administrator:
        loginService.logoutUser(sessionToken);

        // - OR logout as own user:
        try {
            userFactory.getProxy(LoginService.class).logout();
        } catch (LoggedOutException e) {
            // already logged out
        }
    }
}

```

- PHP clients: In the configuration file, instead of calling `Cyclos \Configuration::setAuthentication($username, $password)`, call the following: `Cyclos \Configuration::setSessionToken($sessionToken)` and `Cyclos \Configuration::setForwardRemoteAddress(true)`, which will automatically send the `$_SERVER['REMOTE_ADDR']` value on requests.
- WEB-RPC: If sending JSON requests directly, instead of passing the Authentication header with the username / password, pass the following headers: Session-Token and Remote-Address.