OTRS
Open Technology
Real Services

Documentation

OTRS 3.3 - Developer Manual

Build Date:

2013-08-15

OTRS 3.3 - Developer Manual

Copyright © 2003-2013 OTRS AG

René Bakker, Hauke Böttcher, Stefan Bedorf, Shawn Beasley, Jens Bothe, Udo Bretz, Martin Edenhofer, Carlos Javier García, Martin Gruner, Manuel Hecht, Christopher Kuhn, André Mindermann, Henning Oschwald, Thomas Raith, Carlos Fernando Rodríguez, Stefan Rother, Burchard Steinbild.

This work is copyrighted by OTRS AG.

You may copy it in whole or in part as long as the copies retain this copyright statement.

The source code of this document can be found at github.

UNIX is a registered trademark of X/Open Company Limited. Linux is a registered trademark of Linus Torvalds.

MS-DOS, Windows, Windows 95, Windows 98, Windows NT, Windows 2000, Windows XP, Windows 2003, Windows Vista and Windows 7 are registered trademarks of Microsoft Corporation. Other trademarks and registered trademarks are: SUSE and YaST of SUSE Linux GmbH, Red Hat and Fedora are registered trademarks of Red Hat, Inc. Mandrake is a registered trademark of MandrakeSoft, SA. Debian is a registered trademark of Software in the Public Interest, Inc. MySQL and the MySQL Logo are registered trademarks of Oracle Corporation and/or its affiliates.

All trade names are used without the guarantee for their free use and are possibly registered trade marks.

OTRS AG essentially follows the notations of the manufacturers. Other products mentioned in this manual may be trademarks of the respective manufacturer.

Table of Contents

Chapter 1. Getting Started

OTRS is a multi-platform web application framework which was originally developed for a trouble ticket system. It supports different web servers and databases.

This manual shows how to develop your own OTRS modules and applications based on the OTRS styleguides.

1. Development Environment

To facilitate the writing of OTRS expansion modules, the creation of a development environment is necessary. The source code of OTRS and additional public modules can be found on github.

1.1. Framework checkout

First of all a directory must be created in which the modules can be stored. Then switch to the new directory using the command line and check them out of OTRS 3.1 or the git master by using the following command:

```
# for git master
shell> git clone git@github.com:OTRS/otrs.git -b master
# for a specific branch
shell> git clone git@github.com:OTRS/otrs.git -b rel-3_3
```

Check out the "module-tools" module (from github) too, for your development environment. It contains a number of useful tools.

To enable the new OTRS system it is necessary to configure it on the Apache web server and to create the Config.pm. Then the Installer.pl can be executed. The basic system is ready to run now.

1.2. Linking Expansion Modules

A clear separation between OTRS and the modules is necessary for proper developing. Particularly when using a git, a clear separation is crucial. In order to facilitate the OTRS access to the files, links must be created. This is done by a script in the directory module tools (to get this tools, check out the git module "module-tools"). Example: Linking the Calendar Module:

```
shell> ~/src/module-tools/link.pl ~/src/Calendar/ ~/src/otrs/
```

Whenever new files are added, they must be linked as described above.

To remove links from OTRS enter the following command:

```
shell> ~/src/module-tools/remove_links.pl ~/src/otrs/
```

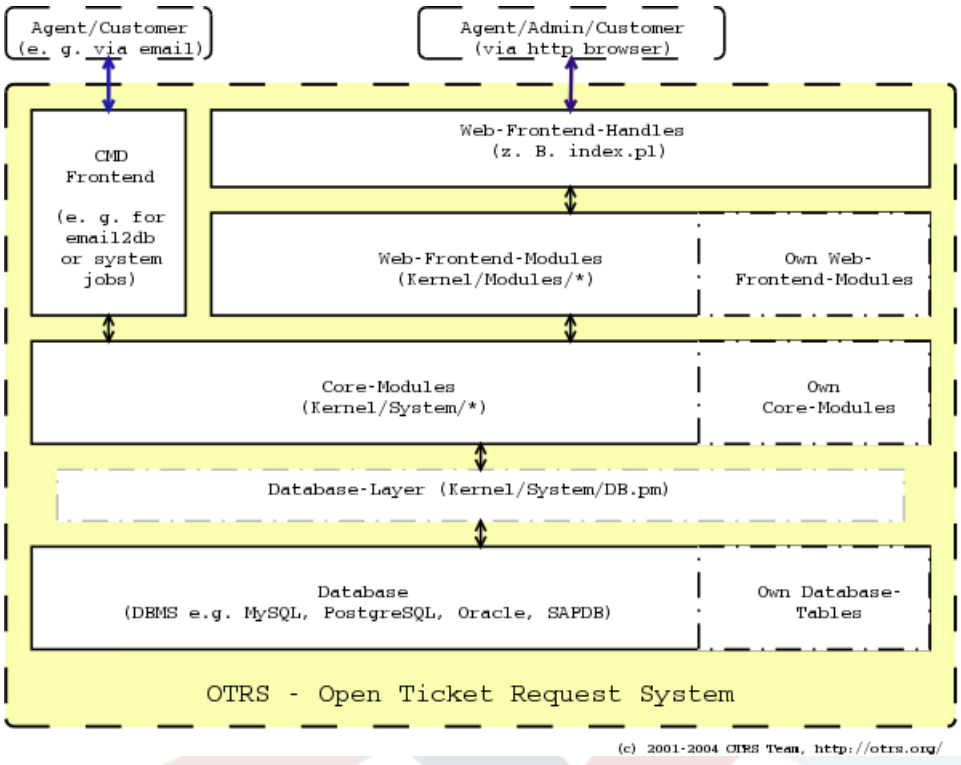
1.3. Necessary Actions after Linking

As soon as the linking is completed, the Sysconfig must be run to register the module in OTRS. Required users, groups and roles must be created manually and access authorizations must be defined. If an additional databank table is required, this must be created manually, too. If an OPM package exists, the SQL commands can be read out to create the tables. Example:

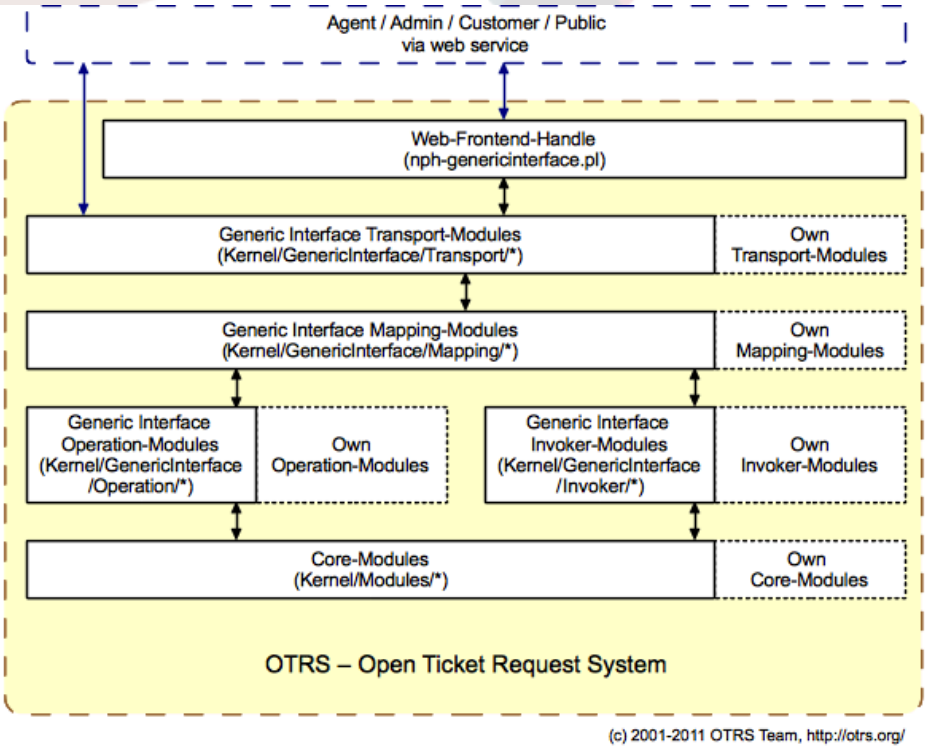
```
shell> cat Calendar.sopm | bin/xml2sql.pl -t mysql
```

2. Architecture Overview

The OTRS framework is modular. The following picture shows the basic layer architecture of OTRS.



Introduced in OTRS 3.1, the OTRS Generic Interface continues OTRS modularity. The next picture shows the basic layer architecture of the Generic Interface.



2.1. Directories

Directory	Description
bin/	commandline tools
bin/cgi-bin/	web handle
bin/fcgi-bin/	fast cgi web handle
Kernel	application codebase
Kernel/Config/	configuration files
Kernel/Config/Files	configuration files
Kernel/GenericInterface/	the Generic Interface API
Kernel/GenericInterface/Invoker/	invoker modules for Generic Interface
Kernel/GenericInterface/Mapping/	mapping modules for Generic Interface, e.g. Simple
Kernel/GenericInterface/Operation/	operation modules for Generic Interface
Kernel/GenericInterface/Transport/	transport modules for Generic Interface, e.g. "HTTP SOAP"
Kernel/Language	language translation files
Kernel/Scheduler/	Scheduler files
Kernel/Scheduler/TaskHandler	handler modules for scheduler tasks, e.g. GenericInterface
Kernel/System/	core modules, e.g. Log, Ticket...
Kernel/Modules/	frontend modules, e.g. QueueView...
Kernel/Output/HTML/	html templates
var/	variable data
var/log	logfiles
var/cron/	cron files
var/httpd/htdocs/	htdocs directory with index.html
var/httpd/htdocs/skins/Agent/	available skins for the Agent interface
var/httpd/htdocs/skins/Customer/	available skins for the Customer interface
var/httpd/htdocs/js/	JavaScript files
scripts/	misc files
scripts/test/	unit test files
scripts/test/sample/	unit test sample data files

2.2. Files

.pl = Perl

.pm = Perl Module

.dtl = Dynamic Template Language (html template files

.dist = Default templates of files

.yaml or .yml = YAML files, used for Web Service configuration

2.3. Core Modules

Core modules are located under $OTRS_HOME/Kernel/System/*. This layer is for the logical work. Core modules are used to handle system routines like "lock ticket" and "create ticket". A few main core modules are:

- Kernel::System::Config (to access config options)
- Kernel::System::Log (to log into OTRS log backend)
- Kernel::System::DB (to access the database backend)
- Kernel::System::Auth (to check user authentication)
- Kernel::System::User (to manage users)
- Kernel::System::Group (to manage groups)
- Kernel::System::Email (for sending emails)

For more information, see: http://otrs.github.io/doc/

2.4. Frontend Handle

The interface between the browser, web server and the frontend modules. A frontend module can be used via the http-link.

http://localhost/otrs/index.pl?Action=Module

2.5. Frontend Modules

Frontend modules are located under "$OTRS_HOME/Kernel/Modules/*.pm". There are two public functions in there - "new()" and "run()" - which are accessed from the Frontend Handle (e.g. index.pl).

"new()" is used to create a frontend module object. The Frontend Handle provides the used frontend module with the basic framework objects. These are, for example: ParamObject (to get web form params), DBObject (to use existing database connections), LayoutObject (to use templates and other html layout functions), ConfigObject (to access config settings), LogObject (to use the framework log system), UserObject (to get the user functions from the current user), GroupObject (to get the group functions).

For more information on core modules see: http://otrs.github.io/doc/

2.6. CMD Frontend

The CMD (Command) Frontend is like the Web Frontend Handle and the Web Frontend Module in one (just without the LayoutObject) and uses the core modules for some actions in the system.

2.7. Generic Interface Modules

Generic Interface modules are located under $OTRS_HOME/Kernel/GenericInterface/*. Generic Interface modules are used to handle each part of an web service execution on the system. The main modules for the Generic Interface are:

- Kernel::GenericInterface::Transport (to interact with remote systems)

- Kernel::GenericInterface::Mapping (to transform data into a required format)
- Kernel::GenericInterface::Requester (to use OTRS as a client for the web service)
- Kernel::GenericInterface::Provider (to use OTRS as a server for web service)
- Kernel::GenericInterface::Operation (to execute Provider actions)
- Kernel::GenericInterface::Invoker (to execute Requester actions)
- Kernel::GenericInterface::Debugger (to track web service communication, using log entries)

For more information, see: http://otrs.github.io/doc/

2.7.1. Generic Interface Invoker Modules

Generic Interface Invoker modules are located under $OTRS_HOME/Kernel/GenericInterface/Invoker/*. Each Invoker is contained in a folder called "Controller". This approach helps to define a name space not only for internal classes and methods but for filenames too. for example: $OTRS_HOME/Kernel/GenericInterface/Invoker*/Test/* is the Controller for all Test-type invokers.

Generic Interface Invoker modules are used as a backend to create requests for Remote Systems to execute actions.

For more information, see: http://otrs.github.io/doc/

2.7.2. Generic Interface Mapping Modules

Generic Interface Mapping modules are located under $OTRS_HOME/Kernel/GenericInterface/Mapping/*. These modules are used to transform data (keys and values) from one format to another.

For more information, see: http://otrs.github.io/doc/

2.7.3. Generic Interface Operation Modules

Generic Interface Operation modules are located under $OTRS_HOME/Kernel/GenericInterface/Operation/*. Each Operation is contained in a folder called "Controller". This approach help to define a name space not only for internal classes and methods but for filenames too. for example: $OTRS_HOME/Kernel/GenericInterface/Operation*/Ticket/* is the Controller for all Ticket-type operations.

Generic Interface operation modules are used as a backend to create perform actions requester by a remote system.

For more information, see: http://otrs.github.io/doc/

2.7.4. Generic Interface Transport Modules

Generic Interface Network Transport modules are located under $OTRS_HOME/Kernel/GenericInterface/Operation/*. Each transport module should be placed in a diretory named as the Network Protocol used. for example: The "HTTP SOAP" transport module, located in $OTRS_HOME/Kernel/GenericInterface/Transport */HTTP/SOAP.pm*.

Generic Interface transport modules are used send data to, and recieve data from a Remote System.

For more information, see: http://otrs.github.io/doc/

2.8. Scheduler Task Handler Modules

Scheduler Task Handler modules are located under $OTRS_HOME/Kernel/Scheduler/TaskHandler/*. These modules are used to perform asynchronous tasks. For example, the GenericInterface task handler perform Generic Interface Requests to Remote Systems outside the apache process, this helps the system to be more responsive, preventing possible performance issues.

For more information, see: http://otrs.github.io/doc/

2.9. Database

The database interface supports different databases.

For the OTRS data model please refer to the files in your /doc directory. Alternatively you can look at the data model on github .

Chapter 2. OTRS Internals - How it Works

1. Config Mechanism

1.1. Default Config

There are different default config files. The main one, which comes with the framework, is:

`Kernel/Config/Defaults.pm`

This file should be left untouched as it is automatically updated on framework updates. There is also a sub directory where you can store the default config files for your own modules. These files are used automatically.

The directory is located under:

`$OTRS_HOME/Kernel/Config/Files/*.pm`

And could look as follows:

`Kernel/config/Files/Calendar.pm`

```
# module reg and nav bar
$Self->{'Frontend::Module'}->{'AgentCalendar'} = {
    Description => 'Calendar',
    NavBarName => 'Ticket',
    NavBar => [
        {
            Description => 'Calendar',
            Name => 'Calendar',
            Image => 'calendar.png',
            Link => 'Action=AgentCalendar',
            NavBar => 'Ticket',
            Prio => 5000,
            AccessKey => 'c',
        },
    ],
};

# show online customers
$Self->{'Frontend::NotifyModule'}->{'80-ShowCalendarEvents'} = {
    Module => 'Kernel::Output::HTML::NotificationCalendar',
};
```

1.2. Custom Config

If you want to change a config option, copy it to

`Kernel/Config.pm`

and set the new option. This file will be read out last and so all default config options are overwritten with your settings.

This way it is easy to handle updates - you just need the Kernel/Config.pm.

1.3. Accessing Config Options

You can read and write (for one request) the config options via the core module "Kernel::Config". The config object is a base object and thus available in each Frontend Module.

If you want to access a config option:

```
my $ConfigOption = $Self->{ConfigObject}->Get('Prefix::Option');
```

If you want to change a config option at runtime and just for this one request/process:

```
$Self->{ConfigObject}->Set(
    Key => 'Prefix::Option'
    Value => 'SomeNewValue',
);
```

1.4. XML Config Options

XML config files are located under:

`$OTRS_HOME/Kernel/Config/Files/*.xml`

Each config file has the following layout:

```
<?xml version="1.0" encoding="utf-8" ?>
<otrs_config version="1.0" init="Changes">

    <!--  config items will be here -->

</otrs_config>
```

The "init" attribute describes where the config options should be loaded. There are different levels available and will be loaded/overloaded in the following order: "Framework" (for framework settings e. g. session option), "Application" (for application settings e. g. ticket options), "Config" (for extensions to existing applications e. g. ITSM options) and "Changes" (for custom development e. g. to overwrite framework or ticket options).

If you want to add config options, here is an example:

```
<ConfigItem Name="Ticket::Hook" Required="1" Valid="1" ConfigLevel="300">
    <Description Lang="en">The identifyer for a ticket. The default is Ticket#.</
Description>
    <Description Lang="de">Ticket-Identifikator. Als Standard wird Ticket# verwendet.</
Description>
    <Group>Ticket</Group>
    <SubGroup>Core::Ticket</SubGroup>
    <Setting>
        <String Regex="">Ticket#</String>
    </Setting>
</ConfigItem>
```

If "required" is set to "1", the config variable is included and cannot be disabled.

If "valid" is set to "1", the config variable is active. If it is set to "0", the config variable is inactive.

If the optional attribute "ConfigLevel" is set, the config variable might not be edited by the administrator, depending on his own config level. The config variable "ConfigLevel" sets the level of technical experience of the administrator. It can be 100 (Expert), 200 (Advanced) or 300 (Beginner). As a guideline which config level should be given to an option, it is recommended that all options having to do with the configuration of externel interaction, like Sendmail, LDAP, SOAP, and others should get a config level of at least 200 (Advanced).

The config variable is defined in the "setting" element.

1.4.1. Types of XML Config Variables

The XML config settings support various types of variables.

1.4.2. String

A config element for numbers and single-line strings. Checking the validity with a regex is possible. The check attribute checks elements on the file system. This contains files and directories.

```
<Setting>
    <String Regex="" Check="File"></String>
</Setting>
```

1.4.3. Textarea

A config element for multiline text.

```
<Setting>
    <TextArea Regex=""></TextArea>
</Setting>
```

1.4.4. Options

This config element offers preset values as a pull-down menu.

```
<Setting>
    <Option SelectedID="Key">
        <Item Key=""></Item>
        <Item Key=""></Item>
    </Option>
</Setting>
```

1.4.5. Array

With this config element arrays can be displayed.

```
<Setting>
    <Array>
        <Item></Item>
        <Item></Item>
    </Array>
</Setting>
```

1.4.6. Hash

With this config element hashes can be displayed.

```
<Setting>
    <Hash>
        <Item Key=""></Item>
        <Item Key=""></Item>
```

```
    </Hash>
</Setting>
```

1.4.7. Hash with SubArray, SubHash

A hash can contain content, arrays or hashes.

```
<Setting>
    <Hash>
        <Item Key=""></Item>
        <Item Key="">
            <Hash>
                <Item Key=""></Item>
                <Item Key=""></Item>
            </Hash>
        </Item>
        <Item Key="">
            <Array>
                <Item></Item>
                <Item></Item>
            </Array>
        </Item>
        <Item Key=""></Item>
    </Hash>
</Setting>
```

1.4.8. FrontendModuleReg (NavBar)

Module registration for Agent Interface.

```
<Setting>
    <FrontendModuleReg>
        <Group>group1</Group>
        <Group>group2</Group>
        <Description>Logout</Description>
        <Title></Title>
        <NavBarName></NavBarName>
        <NavBar>
            <Description>Logout</Description>
            <Name>Logout</Name>
            <Image>exit.png</Image>
            <Link>Action=Logout</Link>
            <NavBar></NavBar>
            <Type></Type>
            <Block>ItemPre</Block>
            <AccessKey>l</AccessKey>
            <Prio>100</Prio>
        </NavBar>
    </FrontendModuleReg>
</Setting>
```

1.4.9. FrontendModuleReg (NavBarModule)

Module registration for Admin Interface

```
<Setting>
    <FrontendModuleReg>
        <Group>admin</Group>
        <Group>admin2</Group>
        <Description>Admin</Description>
        <Title>User</Title>
        <NavBarName>Admin</NavBarName>
```

```
        <NavBarModule>
            <Module>Kernel::Output::HTML::NavBarModuleAdmin</Module>
            <Name>Users</Name>
            <Block>Block1</Block>
            <Prio>100</Prio>
        </NavBarModule>
    </FrontendModuleReg>
</Setting>
```

2. Database Mechanism

OTRS comes with a database layer that supports different databases.

2.1. How it works

The database layer (Kernel::System::DB) has two input options: SQL and XML.

2.1.1. SQL

The SQL interface should be used for normal database actions (SELECT, INSERT, UPDATE, ...). It can be used like a normal Perl DBI interface.

2.1.1.1. INSERT/UPDATE/DELETE

```
$Self->{DBObject}->Do(
    SQL=> "INSERT INTO table (name, id) VALUES ('SomeName', 123)",
);

$Self->{DBObject}->Do(
    SQL=> "UPDATE table SET name = 'SomeName', id = 123",
);

$Self->{DBObject}->Do(
    SQL=> "DELETE FROM table WHERE id = 123",
);
```

2.1.1.2. SELECT

```
my $SQL = "SELECT id FROM table WHERE tn = '123'";

$Self->{DBObject}->Prepare(SQL => $SQL, Limit => 15);

while (my @Row = $Self->{DBObject}->FetchrowArray()) {
    $Id = $Row[0];
}
return $Id;
```

Note

Take care to use Limit as param and not in the SQL string because not all databases support LIMIT in SQL strings.

```
my $SQL = "SELECT id FROM table WHERE tn = ? AND group = ?";

$Self->{DBObject}->Prepare(
    SQL   => $SQL,
    Limit => 15,
    Bind  => [ $Tn, $Group ],
);
```

```
while (my @Row = $Self->{DBObject}->FetchrowArray()) {
    $Id = $Row[0];
}
return $Id;
```

Note

Use the Bind attribute where ever you can, especially for long statements. If you use Bind you do not need the function Quote().

2.1.1.3. QUOTE

String:

```
my $QuotedString = $Self->{DBObject}->Quote("It's a problem!");
```

Integer:

```
my $QuotedInteger = $Self->{DBObject}->Quote('123', 'Integer');
```

Number:

```
my $QuotedNumber = $Self->{DBObject}->Quote('21.35', 'Number');
```

Note

Please use the Bind attribute instead of Quote() where ever you can.

2.1.2. XML

The XML interface should be used for INSERT, CREATE TABLE, DROP TABLE and ALTER TABLE. As this syntax is different from database to database, using it makes sure that you write applications that can be used in all of them.

Note

The <Insert> has changed in >=2.2. Values are now used in content area (not longer in attribut Value).

2.1.2.1. INSERT

```
<Insert Table="some_table">
    <Data Key="id">1</Data>
    <Data Key="description" Type="Quote">exploit</Data>
</Insert>
```

2.1.2.2. CREATE TABLE

Possible data types are: BIGINT, SMALLINT, INTEGER, VARCHAR (Size=1-1000000), DATE (Format: yyyy-mm-dd hh:mm:ss) and LONGBLOB.

```
<TableCreate Name="calendar_event">
```

```
    <Column Name="id" Required="true" PrimaryKey="true" AutoIncrement="true" Type="BIGINT"/>
    <Column Name="title" Required="true" Size="250" Type="VARCHAR"/>
    <Column Name="content" Required="false" Size="250" Type="VARCHAR"/>
    <Column Name="start_time" Required="true" Type="DATE"/>
    <Column Name="end_time" Required="true" Type="DATE"/>
    <Column Name="owner_id" Required="true" Type="INTEGER"/>
    <Column Name="event_status" Required="true" Size="50" Type="VARCHAR"/>
    <Index Name="calendar_event_title">
        <IndexColumn Name="title"/>
    </Index>
    <Unique Name="calendar_event_title">
        <UniqueColumn Name="title"/>
    </Unique>
    <ForeignKey ForeignTable="users">
        <Reference Local="owner_id" Foreign="id"/>
    </ForeignKey>
</TableCreate>
```

2.1.2.3. DROP TABLE

```
<TableDrop Name="calendar_event"/>
```

2.1.2.4. ALTER TABLE

The following shows an example of add, change and drop columns.

```
<TableAlter Name="calendar_event">
    <ColumnAdd Name="test_name" Type="varchar" Size="20" Required="true"/>

    <ColumnChange NameOld="test_name" NameNew="test_title" Type="varchar" Size="30"
 Required="true"/>

    <ColumnChange NameOld="test_title" NameNew="test_title" Type="varchar" Size="100"
 Required="false"/>

    <ColumnDrop Name="test_title"/>

    <IndexCreate Name="index_test3">
        <IndexColumn Name="test3"/>
    </IndexCreate>

    <IndexDrop Name="index_test3"/>

    <UniqueCreate Name="uniq_test3">
        <UniqueColumn Name="test3"/>
    </UniqueCreate>

    <UniqueDrop Name="uniq_test3"/>
</TableAlter>
```

The next shows an example how to rename a table.

```
<TableAlter NameOld="calendar_event" NameNew="calendar_event_new"/>
```

2.1.2.5. Code to process XML

```
my @XMLARRAY = @{$Self->ParseXML(String => $XML)};

my @SQL = $Self->{DBObject}->SQLProcessor(
    Database => \@XMLARRAY,
);
push(@SQL, $Self->{DBObject}->SQLProcessorPost());
```

```
for (@SQL) {
    $Self->{DBObject}->Do(SQL => $_);
}
```

2.2. Database Drivers

The database drivers are located under `$OTRS_HOME/Kernel/System/DB/*.pm`.

2.3. Supported Databases

- MySQL
- PostgreSQL
- Oracle
- Microsoft SQL Server (mssql)

3. Log Mechanism

OTRS comes with a log backend that can be used for application logging and debugging.

3.1. Use and Syntax

All module layers have ready-made Log Objects which can be used by

```
$Self->{LogObject}->Log(
    Priority => 'error',
    Message => 'Need something!',
);
```

3.2. Example

The following example shows how to use the log mechanism without a module layer.

```
use Kernel::Config;
use Kernel::System::Encode;
use Kernel::System::Log;

my $ConfigObject = Kernel::Config->new();
my $EncodeObject = Kernel::System::Encode->new(
    ConfigObject => $ConfigObject,
);
my $LogObject    = Kernel::System::Log->new(
    ConfigObject => $ConfigObject,
);

$Self->{LogObject}->Log(
    Priority => 'error',
    Message => 'Need something!',
);
```

4. Skins

Since OTRS 3.0, the visual appearance of OTRS is controlled by "skins".

A skin is a set of CSS and image files, which together control how the GUI is presented to the user. Skins do not change the HTML content that is generated by OTRS (this is what "Themes" do), but they control how it is displayed. With the help of modern CSS standards it is possible to change the display thoroughly (e.g. repositioning parts of dialogs, hiding elements, ...).

4.1. Skin Basics

All skins are in `$OTRS_HOME/var/httpd/htdocs/skins/$SKIN_TYPE/$SKIN_NAME`. There are two types of skins: agent and customer skins.

Each of the agents can select individually, which of the installed agent skins they want to "wear".

For the customer interface, a skin has to be selected globally with the config setting `Loader::Customer::SelectedSkin`. All customers will see this skin.

4.2. How skins are loaded

It is important to note that the "default" skin will *always* be loaded *first*. If the agent selected another skin than the "default" one, then the other one will be loaded only *after* the default skin. By "loading" here we mean that OTRS will put tags into the HTML content which cause the CSS files to be loaded by the browser. Let's see an example of this:

```
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css-cache/
CommonCSS_179376764084443c181048401ae0e2ad.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/ivory/css-cache/
CommonCSS_e0783e0c2445ad9cc59c35d6e4629684.css" />
```

Here it can clearly be seen that the default skin is loaded first, and then the custom skin specified by the agent. In this example, we see the result of the activated loader (`Loader::Enabled` set to 1), which gathers all CSS files, concatenates and minifies them and serves them as one chunk to the browser. This saves bandwith and also reduces the number of HTTP requests. Let's see the same example with the Loader turned off:

```
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Reset.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Default.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Header.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/
Core.OverviewControl.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.OverviewSmall.css" /
>
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/
Core.OverviewMedium.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.OverviewLarge.css" /
>
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Footer.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Grid.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Form.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Table.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Widget.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.WidgetMenu.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.TicketDetail.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Tooltip.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Dialog.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/Core.Print.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/
Core.Agent.CustomerUser.GoogleMaps.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/default/css/
Core.Agent.CustomerUser.OpenTicket.css" />
    <link rel="stylesheet" href="/otrs-web/skins/Agent/ivory/css/Core.Dialog.css" />
```

Here we can better see the individual files that come from the skins.

There are different types of CSS files: common files which must always be loaded, and "module-specific" files which are only loaded for special modules within the OTRS framework.

In addition, it is possible to specify CSS files which only must be loaded on IE7 or IE8 (in the case of the customer interface, also IE6). This is unfortunate, but it was not possible to develop a modern GUI on these browsers without having special CSS for them.

For details regarding the CSS file types, please see the section on the Loader.

For each HTML page generation, the loader will first take all configured CSS files from the default skin, and then for each file look if it is also available in a custom skin (if a custom skin is selected) and load them after the default files.

That means a) that CSS files in custom skins need to have the same names as in the default skins, and b) that a custom skin does not need to have all files of the default skin. That is the big advantage of loading the default skin first: a custom skin has all default CSS rules present and only needs to change those which should result in a different display. That can often be done in a single file, like in the example above.

Another effect of this is that you need to be careful to overwrite all default CSS rules in your custom skins that you want to change. Let's see an example:

```
.Header h1 {
    font-weight: bold;
    color: #000;
}
```

This defines special headings inside of the .Header element as bold, black text. Now if you want to change that in your skin to another color and normal text, it is not enough to write

```
.Header h1 {
    color: #F00;
}
```

Because the original rule for `font-weight` still applies. You need to override it explicitly:

```
.Header h1 {
    font-weight: normal;
    color: #F00;
}
```

4.3. Creating a New Skin

In this section, we will be creating a new agent skin which replaces the default OTRS background color (white) with a custom company color (light grey) and the default logo by a custome one. Also we will configure that skin to be the one which all agents will see by default.

There are only two simple steps we need to take to achieve this goal:

- create the skin files
- configure the new logo and
- make the skin known to the OTRS system.

Let's start by creating the files needed for our new skin. First of all, we need to create a new folder for this skin (we'll call it "custom"). This folder will be `$OTRS_HOME/var/httpd/htdocs/skins/Agent/custom`.

In there, we need to place the new CSS file in a new directory `css` which defines the new skin's appearance. We'll call it `Core.Default.css` (remember that it must have the same name as one of the files in the "default" skin). This is the code needed for the CSS file:

```
body {
    background-color: #c0c0c0; /* not very beautiful but it meets our purpose */
}
```

Now follows the second step, adding a new logo and making the new skin known to the OTRS system. For this, we first need to place our custom logo (e.g. `logo.png`) in a new directory (e.g. `img`) in our skin directory. Then we need to create a new config file `$OTRS_HOME/Kernel/Config/Files/CustomSkin.xml`, which will contain the needed settings as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<otrs_config version="1.0" init="Framework">
    <ConfigItem Name="AgentLogo" Required="0" Valid="1">
        <Description Translatable="1">The logo shown in the header of the agent interface.
 The URL to the image must be a relative URL to the skin image directory.</Description>
        <Group>Framework</Group>
        <SubGroup>Frontend::Agent</SubGroup>
        <Setting>
            <Hash>
                <Item Key="URL">skins/Agent/custom/img/logo.png</Item>
                <Item Key="StyleTop">13px</Item>
                <Item Key="StyleRight">75px</Item>
                <Item Key="StyleHeight">67px</Item>
                <Item Key="StyleWidth">244px</Item>
            </Hash>
        </Setting>
    </ConfigItem>
    <ConfigItem Name="Loader::Agent::Skin###100-custom" Required="0" Valid="1">
        <Description Translatable="1">Custom skin for the development manual.</Description>
        <Group>Framework</Group>
        <SubGroup>Frontend::Agent</SubGroup>
        <Setting>
            <Hash>
                <Item Key="InternalName">custom</Item>
                <Item Key="VisibleName">Custom</Item>
                <Item Key="Description">Custom skin for the development manual.</Item>
                <Item Key="HomePage">www.yourcompany.com</Item>
            </Hash>
        </Setting>
    </ConfigItem>
</otrs_config>
```

To make this configuration active, we need to navigate to the SysConfig module in the admin area of OTRS (alternatively, you can run the script `$OTRS_HOME/bin/otrs.RebuildConfig.pl`). This will regenerate the Perl cache of the XML configuration files, so that our new skin is now known and can be selected in the system. To make it the default skin that new agents see before they made their own skin selection, edit the SysConfig setting "Loader::Agent::DefaultSelectedSkin" and set it to "custom".

In conclusion: to create a new skin in OTRS, we had to place the new logo file, and create one CSS and one XML file, resulting in three new files:

```
$OTRS_HOME/Kernel/Config/Files/CustomSkin.xml
$OTRS_HOME/var/httpd/htdocs/skins/Agent/custom/img/custom-logo.png
$OTRS_HOME/var/httpd/htdocs/skins/Agent/custom/css/Core.Header.css
```

5. The CSS and JavaScript "Loader"

Starting with OTRS 3.0, the CSS and JavaScript code in OTRS grew to a large amount. To be able to satisfy both development concerns (good maintainability by a large number of separate files) and performance issues (making few HTTP requests and serving minified content without unneccessary whitespace and documentation) had to be addressed. To achieve these goals, the Loader was invented.

5.1. How it works

To put it simple, the Loader

- determines for each request precisely which CSS and JavaScript files are needed at the client side by the current application module
- collects all the relevant data
- minifies the data, removing unneccessary whitespace and documentation
- serves it to the client in only a few HTTP reqests instead of many individual ones, allowing the client to cache these snippets in the browser cache
- perform these tasks in a highly performing way, utilizing the caching mechanisms of OTRS.

Of course, there is a little bit more detailed involved, but this should suffice as a first overview.

5.2. Basic Operation

With the configuration settings `Loader::Enabled::CSS` and `Loader::Enabled::JavaScript`, the loader can be turned on and off for JavaScript and CSS, respectively (it is on by default).

Warning

Because of rendering problems in Internet Explorer, the Loader cannot be turned off for CSS files for this client browser (config setting will be overridden). Up to version 8, Internet Explorer cannot handle more than 32 CSS files on a page.

To learn about how the Loader works, please turn it off in your OTRS installation with the aforementioned configuration settings. Now look at the source code of the application module that you are currently using in this OTRS system (after a reload, of course). You will see that there are many CSS files loaded in the <head> section of the page, and many JavaScript files at the bottom of the page, just before the closing </body> element.

Having the content like this in many individual files with a readable formatting makes the development much easier, and even possible at all. However, this has the disadvantage of a large number of HTTP requests (network latency has a big effect) and unneccessary content (whitespace and documentation) which needs to be transferred to the client.

The Loader solves this problem by performing the steps outlined in the short description above. Please turn on the Loader again and reload your page now. Now you can see that there are only very few CSS and JavaScript tags in the HTML code, like this:

```
<script type="text/javascript" src="/otrs30-dev-web/js/js-cache/
CommonJS_d16010491cbd4faaaeb740136a8ecbfd.js"></script>
```

```
<script type="text/javascript" src="/otrs30-dev-web/js/js-cache/
ModuleJS_b54ba9c085577ac48745f6849978907c.js"></script>
```

What just happened? During the original request generating the HTML code for this page, the Loader generated these two files (or took them from the cache) and put the shown `<script>` tags on the page which link to these files, instead of linking to all relevant JavaScript files separately (as you saw it without the loader being active).

The CSS section looks a little more complicated:

```
    <link rel="stylesheet" type="text/css" href="/otrs30-dev-web/skins/Agent/default/css-
cache/CommonCSS_00753c78c9be7a634c70e914486bfbad.css" />

<!--[if IE 7]>
    <link rel="stylesheet" type="text/css" href="/otrs30-dev-web/skins/Agent/default/css-
cache/CommonCSS_IE7_59394a0516ce2e7359c255a06835d31f.css" />
<![endif]-->

<!--[if IE 8]>
    <link rel="stylesheet" type="text/css" href="/otrs30-dev-web/skins/Agent/default/css-
cache/CommonCSS_IE8_ff58bd010ef0169703062b6001b13ca9.css" />
<![endif]-->
```

The reason is that Internet Explorer 7 and 8 need special treatment in addition to the default CSS because of their lacking support of web standard technologies. So we have some normal CSS that is loaded in all browsers, and some special CSS that is inside of so-called "conditional comments" which cause it to be loaded *only* by Internet Explorer 7/8. All other browsers will ignore it.

Now we have outlined how the loader works. Let's look at how you can utilize that in your own OTRS extensions by adding configuration data to the loader, telling it to load additional or alternative CSS or JavaScript content.

5.3. Configuring the Loader: JavaScript

To be able to operate correctly, the Loader needs to know which content it has to load for a particular OTRS application module. First, it will look for JavaScript files which *always* have to be loaded, and then it looks for special files which are only relevant for the current application module.

5.3.1. Common JavaScript

The list of JavaScript files to be loaded is configured in the configuration settings `Loader::Agent::CommonJS` (for the agent interface) and `Loader::Customer::CommonJS` (for the customer interface).

These settings are designed as hashes, so that OTRS extensions can add their own hash keys for additional content to be loaded. Let's look at an example:

```
<ConfigItem Name="Loader::Agent::CommonJS###000-Framework" Required="1" Valid="1">
    <Description Translatable="1">List of JS files to always be loaded for the agent
 interface.</Description>
    <Group>Framework</Group>
    <SubGroup>Core::Web</SubGroup>
    <Setting>
        <Array>
            <Item>thirdparty/json/json2.js</Item>
            <Item>thirdparty/jquery-1.4.2/jquery.js</Item>

            ...
```

```
            <Item>Core.App.js</Item>
            <Item>Core.Agent.js</Item>
            <Item>Core.Agent.Search.js</Item>
        </Array>
    </Setting>
</ConfigItem>
```

This is the list of JavaScript files which always need to be loaded for the agent interface of OTRS.

To add new content which is supposed to be loaded always in the agent interface, just add an XML configuration file with another hash entry:

```
<ConfigItem Name="Loader::Agent::CommonJS###100-CustomPackage" Required="0" Valid="1">
    <Description Translatable="1">List of JS files to always be loaded for the agent
 interface for package "CustomPackage".</Description>
    <Group>Framework</Group>
    <SubGroup>Core::Web</SubGroup>
    <Setting>
        <Array>
            <Item>CustomPackage.App.js</Item>
        </Array>
    </Setting>
</ConfigItem>
```

Simple, isn't it?

5.3.2. Module-Specific JavaScript

Not all JavaScript is usable for all application modules of OTRS. Therefore it is possible to specify module-specific JavaScript files. Whenever a certain module is used (such as `AgentDashboard`), the module-specific JavaScript for this module will also be loaded. The configuration is done in the frontend module registration in the XML configurations. Again, an example:

```
<ConfigItem Name="Frontend::Module###AgentDashboard" Required="0" Valid="1">
    <Description Translatable="1">Frontend module registration for the agent interface.</
Description>
    <Group>Framework</Group>
    <SubGroup>Frontend::Agent::ModuleRegistration</SubGroup>
    <Setting>
        <FrontendModuleReg>
            <Description>Agent Dashboard</Description>
            <Title></Title>
            <NavBarName>Dashboard</NavBarName>
            <NavBar>
                <Description Translatable="1"></Description>
                <Name Translatable="1">Dashboard</Name>
                <Link>Action=AgentDashboard</Link>
                <NavBar>Dashboard</NavBar>
                <Type>Menu</Type>
                <Description Translatable="1"></Description>
                <Block>ItemArea</Block>
                <AccessKey>d</AccessKey>
                <Prio>50</Prio>
            </NavBar>
            <Loader>
                <JavaScript>thirdparty/flot/excanvas.js</JavaScript>
                <JavaScript>thirdparty/flot/jquery.flot.js</JavaScript>
                <JavaScript>Core.UI.Chart.js</JavaScript>
                <JavaScript>Core.UI.DnD.js</JavaScript>
                <JavaScript>Core.Agent.Dashboard.js</JavaScript>
            </Loader>
        </FrontendModuleReg>
    </Setting>
</ConfigItem>
```

It is possible to put a <Loader> tag in the frontend module registrations which may contain <JavaScript> tags, one for each file that is supposed to be loaded for this application module.

Now you have all information you need to configure the way the Loader handles JavaScript code.

There is one special case: for ToolbarModules, you can also add custom JavaScript files. Just add a JavaScript attribute to the configuration like this:

```
<ConfigItem Name="Frontend::ToolBarModule###410-Ticket::AgentTicketEmail" Required="0"
 Valid="1">
    <Description Translatable="1">Toolbar Item for a shortcut.</Description>
    <Group>Ticket</Group>
    <SubGroup>Frontend::Agent::ToolBarModule</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::Output::HTML::ToolBarLink</Item>
            <Item Key="Name">New email ticket</Item>
            <Item Key="Priority">1009999</Item>
            <Item Key="Link">Action=AgentTicketEmail</Item>
            <Item Key="Action">AgentTicketEmail</Item>
            <Item Key="AccessKey">l</Item>
            <Item Key="CssClass">EmailTicket</Item>
            <Item Key="JavaScript">OTRS.Agent.CustomToolbarModule.js</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

5.4. Configuring the Loader: CSS

The loader handles CSS files very similar to JavaScript files, as described in the previous section, and extending the settings works in the same way too.

5.4.1. Common CSS

The way common CSS is handled is very similar to the way common JavaScript is loaded. Here, the configuration settings are called Loader::Agent::CommonCSS and Loader::Customer::CommonCSS, respectively.

However, as we already noted above, Internet Explorer 7 and 8 (and for the customer interface also 6) need special treatment. That's why there are special configuration settings for them, to specify common CSS which should only be loaded in these browsers. The respective settings are Loader::Agent::CommonCSS::IE7, Loader::Agent::CommonCSS::IE8, Loader::Customer::CommonCSS::IE6, Loader::Customer::CommonCSS::IE7 and Loader::Customer::CommonCSS::IE8.

An example:

```
<ConfigItem Name="Loader::Agent::CommonCSS::IE8###000-Framework" Required="1" Valid="1">
    <Description Translatable="1">List of IE8-specific CSS files to always be loaded for the
 agent interface.</Description>
    <Group>Framework</Group>
    <SubGroup>Core::Web</SubGroup>
    <Setting>
        <Array>
            <Item>Core.OverviewSmall.IE8.css</Item>
        </Array>
    </Setting>
</ConfigItem>
```

This is the list of common CSS files for the agent interface which should only be loaded in Internet Explorer 8.

5.4.2. Module-Specific CSS

Module-specific CSS is handled very similar to the way module-specific JavaScript is handled. It is also configured in the frontend module registrations. Example:

```
<ConfigItem Name="Frontend::Module###Admin" Required="0" Valid="1">
    <Description Translatable="1">Frontend module registration for the agent interface.</
Description>
    <Group>Framework</Group>
    <SubGroup>Frontend::Admin::ModuleRegistration</SubGroup>
    <Setting>
        <FrontendModuleReg>
            <Group>admin</Group>
            <Description>Admin-Area</Description>
            <Title></Title>
            <NavBarName>Admin</NavBarName>
            <NavBar>
                <Type>Menu</Type>
                <Description Translatable="1"></Description>
                <Block>ItemArea</Block>
                <Name Translatable="1">Admin</Name>
                <Link>Action=Admin</Link>
                <NavBar>Admin</NavBar>
                <AccessKey>a</AccessKey>
                <Prio>10000</Prio>
            </NavBar>
            <NavBarModule>
                <Module>Kernel::Output::HTML::NavBarModuleAdmin</Module>
            </NavBarModule>
            <Loader>
                <CSS>Core.Agent.Admin.css</CSS>
                <CSS_IE7>Core.Agent.AdminIE7.css</CSS_IE7>
                <JavaScript>Core.Agent.Admin.SysConfig.js</JavaScript>
            </Loader>
        </FrontendModuleReg>
    </Setting>
</ConfigItem>
```

Here we have a module (the admin overview page of the agent interface) which has special JavaScript, normal CSS (tagname <CSS>) and special CSS for Internet Explorer 7 (tagname <CSS_IE7>). All of these need to be loaded in addition to the common JavaScript and CSS defined for the agent interface.

It is also possible to specify module-specific CSS for Internet Explorer 8 (tagname <CSS_IE8>) and, in the case of the customer interface, for Internet Explorer 6 (tagname <CSS_IE6>).

There is one special case: for `ToolbarModules`, you can also add custom CSS files. Just add a `CSS`, `CSS_IE7` or `CSS_IE8` attribute to the configuration like this:

```
<ConfigItem Name="Frontend::ToolBarModule###410-Ticket::AgentTicketEmail" Required="0"
 Valid="1">
    <Description Translatable="1">Toolbar Item for a shortcut.</Description>
    <Group>Ticket</Group>
    <SubGroup>Frontend::Agent::ToolBarModule</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::Output::HTML::ToolBarLink</Item>
            <Item Key="Name">New email ticket</Item>
            <Item Key="Priority">1009999</Item>
            <Item Key="Link">Action=AgentTicketEmail</Item>
            <Item Key="Action">AgentTicketEmail</Item>
            <Item Key="AccessKey">l</Item>
```

```
            <Item Key="CssClass">EmailTicket</Item>
            <Item Key="CSS">OTRS.Agent.CustomToolbarModule.css</Item>
            <Item Key="CSS_IE7">OTRS.Agent.CustomToolbarModule.IE7.css</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

6. Templating Mechanism

Internally, OTRS uses a templating mechanism to dynamically generate its HTML pages (and other content), while keeping the program logic (Perl) and the presentation (HTML) separate. Typically, a frontend module will use an own template file, pass some data to it and return the rendered result to the user.

The template files are located at: `$OTRS_HOME/Kernel/Output/HTML/Standard/*.dtl`

Inside of these templates, a set of commands for data manipulation, localization and simple logical structures can be used. This section describes these commands and shows how to use them in templates.

6.1. Template Commands

6.1.1. Data Manipulation Commands

In templates, dynamic data must be inserted, quoted etc. This section lists the relevant commands to do that.

6.1.1.1. $Data{""}

If data parameters are given to the templates by the application module, these data can be output to the template. `$Data` is the most simple, but also most dangerous one. It will insert the data parameter whose name is specified inside of the `{""}` into the template as it is, without any HTML quoting.

> **Warning**
>
> Because of the missing HTML quoting, this can result in security problems. Never output data that was input by a user without quoting in HTML context. The user could - for example - just insert a `<script>` tag, and it would be output on the HTML page generated by OTRS.
>
> Whenever possible, use `$QData{""}` (in HTML) or `$LQData{""}` (in Links) instead.

Example: Whenever we generate HTML in the application, we need to output it to the template without HTML quoting, like `<select>` elements, which are generated by the function `Layout::BuildSelection` in OTRS.

```
<label for="Dropdown">Example Dropdown</label>
$Data{"DropdownString"}
```

6.1.1.2. $QData{""}

This command has the same function as `$Data{""}`, but it performs HTML quoting on the data as it is inserted to the template.

```
The name of the author is $QData{"Name"}.
```

It's also possible specify a maximum length for the value. If, for example, you just want to show 8 characters of a variable (result will be "SomeName[...]"), use the following:

```
The first 20 characters of the author's name: $QData{"Name","20"}.
```

6.1.1.3. $LQData{""}

This command has the same function as $Data{""}, but it performs URL encoding on the data as it is inserted to the template. This should be used to output single parameter names or values of URLs, to prevent security problems. It cannot be used for complete URLs because it will also mask =, for example.

```
<a href="$Env{"Baselink"};Location=$LQData{"File"}">$QData{"File","110"}</a>
```

6.1.1.4. $Env{""}

Inserts the environment variable with the name specified in {""}. Some examples:

```
The current user name is: $Env{"UserFirstname"}

Some other common predefined variables are:

$Env{"SessionID"} --> the current session id
$Env{"Time"} --> the current time e. g.  Thu Dec 27 16:00:55 2001
$Env{"CGIHandle"} --> the current CGI handle e. g.  index.pl
$Env{"UserCharset"} --> the current site charset e. g.  iso-8859-1
$Env{"Baselink"} --> the baselink --> index.pl?SessionID=...
$Env{"UserFirstname"} --> e. g. Dirk $Env{"UserFirstname"}
$Env{"UserLogin"} --> e. g. mgg@x11.org
$Env{"UserIsGroup[users]"} = Yes --> user groups (useful for own links)
$Env{"UserIsGroup[admin]"} = Yes $Env{"Action"} --> the current action
```

Warning

Because of the missing HTML quoting, this can result in security problems. Never output data that was input by a user without quoting in HTML context. The user could - for example - just insert a <script> tag, and it would be output on the HTML page generated by OTRS.

Whenever possible, use $QEnv{""} instead.

6.1.1.5. $QEnv{""}

Works like $Env{""}, but performs HTML encoding when the data is inserted to the template.

```
The current user name is: $QEnv{"UserFirstname"}
```

6.1.1.6. $Config{""}

With this tag you can insert config variables into the template. Let's see an example Kernel/Config.pm:

```
[Kernel/Config.pm]
    # FQDN
    # (Full qualified domain name of your system.)
```

```
    $Self->{FQDN} = 'otrs.example.com';
    # AdminEmail
    # (Email of the system admin.)
    $Self->{AdminEmail} = 'admin@example.com';
[...]
```

To output values from it in the template, use:

```
The hostname is '$Config{"FQDN"}'
The admin email address is '$Config{"AdminEmail"}'
```

Warning

Because of the missing HTML quoting, this can result in security problems.

Whenever possible, use `$Quote{"$Config{""}"}`.

6.1.1.7. $Quote{""}

This tag can be used to perform quoting on HTML strings, when no other quoting is possible.

```
$Quote{"$Config{"ProductName"}"} ($Quote{"$Config{"Ticket::Hook"}"})
```

It's also possible specify a maximum length for the value. If, for example, you just want to show 8 characters of a variable (result will be "Some lon[...]"), use the following:

```
$Quote{"Some long long string", "8"})
```

6.1.2. Localization Commands

6.1.2.1. $Text{""}

Translates the enlosed string into the current user's selected language and performs HTML quoting on the resulting string. If no translation is found, the original string will be used.

```
Translate this text: $Text{"Help"}
```

When translating data coming from the application, use `$Data` inside of `$Text`, not `$QData`, to prevent double quoting:

```
Translate data from the application: $Text{"$Data{"Type"}"}
```

You can also specify parameters (%s) inside of the string which should be replaced with other data:

```
Translate this text and insert the given data: $Text{"Change %s settings", "$Data{"Type"}"}
```

6.1.2.2. $JSText{""}

Works in the same way as `$Text{""}`, but does not perform HTML encoding but JavaScript string escaping instead (all `'` characters will be encoded as `\'`. So with the help of this tag you can make sure that even dynamic strings will not break your JavaScript code.

```
window.alert('$JSText{"Some message's content"}');

// after the command was replaced in the template, this will
// result in (for an English speaking agent):

window.alert('Some message\'s content');
```

Make sure to use ' as string delimiter for strings where you want to use $JSText inside.

6.1.2.3. $TimeLong{""}

Inserts a localized date/time stamp (including a possible time zone difference of the current agent).

In different cultural areas, different convention for date and time formatting are used. For example, what is the 01.02.2010 in Germany, would be 02/01/2010 in the USA. $Time{""} abstracts this away from the templates. Let's see an example:

```
# from AgentTicketHistory.dtl
$TimeLong{"$Data{"CreateTime"}"}

# Result for US English locale:
06/09/2010 15:45:41
```

First, the data is inserted from the application module with $Data. Here always an ISO UTC timestamp (2010-06-09 15:45:41) must be passed as data to $TimeLong{""}. Then $TimeLong{""} will take that data and output it according to the date/time definition of the current locale.

The data passed to $TimeLong{""} must be UTC. If a time zone offset is specified for the current agent, it will be applied to the UTC timestamp before the output is generated.

6.1.2.4. $TimeShort{""}

Works like $TimeLong{""}, but does not output the seconds.

```
$TimeShort{"$Data{"CreateTime"}"}

# Result for US English locale:
06/09/2010 15:45
```

6.1.2.5. $Date{""}

Works like $TimeLong{""}, but outputs only the date, not the time.

```
$Date{"$Data{"CreateTime"}"}

# Result for US English locale:
06/09/2010
```

6.1.3. Template Processing Commands

6.1.3.1. Comment

The dtl comment starts with a # at the beginning of a line and will not be shown in the html output. This can be used both for commenting the DTL (=Template) code or for disabling parts of it.

```
# this section is temporarily disabled
# <div class="AsBlock">
#     <a href="...">link</a>
# </div>
```

6.1.3.2. $Include{""}

Includes another template file into the current one. The included file may also contain template commands.

```
# include Copyright.dtl
$Include{"Copyright"}
```

6.1.3.3. dtl:block

With this command, one can specify parts of a template file as a block. This block needs to be explicitly filled with a function call from the application, to be present in the generated output. The application can call the block 0 (it will not be present in the output), 1 or more times (each with possibly a different set of data parameters passed to the template).

One common use case is the filling of a table with dynamic data:

```
<table class="DataTable">
    <thead>
        <tr>
            <th>$Text{"Name"}</th>
            <th>$Text{"Type"}</th>
            <th>$Text{"Comment"}</th>
            <th>$Text{"Valid"}</th>
            <th>$Text{"Changed"}</th>
            <th>$Text{"Created"}</th>
        </tr>
    </thead>
    <tbody>
<!-- dtl:block:NoDataFoundMsg -->
        <tr>
            <td colspan="6">
                $Text{"No data found."}
            </td>
        </tr>
<!-- dtl:block:NoDataFoundMsg -->
<!-- dtl:block:OverviewResultRow -->
        <tr>
            <td><a class="AsBlock" href="$Env{"Baselink"}Action=
$Env{"Action"};Subaction=Change;ID=$LQData{"ID"}">$QData{"Name"}</a></td>
            <td>$Text{"$Data{"TypeName"}"}</td>
            <td title="$QData{"Comment"}">$QData{"Comment","20"}</td>
            <td>$Text{"$Data{"Valid"}"}</td>
            <td>$TimeShort{"$QData{"ChangeTime"}"}</td>
            <td>$TimeShort{"$QData{"CreateTime"}"}</td>
        </tr>
<!-- dtl:block:OverviewResultRow -->
    </tbody>
</table>
```

The surrounding table with the header is always generated. If no data was found, the block `NoDataFoundMsg` is called once, resulting in a table with one data row with the message "No data found."

If data was found, for each row there is one function call made for the block `OverViewResultRow` (each time passing in the data for this particular row), resulting in a table with as many data rows as results were found.

Let's look at how the blocks are called from the application module:

```
 my %List = $Self->{StateObject}->StateList(
    UserID => 1,
    Valid  => 0,
);

# if there are any states, they are shown
if (%List) {

    # get valid list
    my %ValidList = $Self->{ValidObject}->ValidList();
    for ( sort { $List{$a} cmp $List{$b} } keys %List ) {

        my %Data = $Self->{StateObject}->StateGet( ID => $_, );
        $Self->{LayoutObject}->Block(
            Name => 'OverviewResultRow',
            Data => {
                Valid => $ValidList{ $Data{ValidID} },
                %Data,
            },
        );
    }
}

# otherwise a no data found msg is displayed
else {
    $Self->{LayoutObject}->Block(
        Name => 'NoDataFoundMsg',
        Data => {},
    );
}
```

Note how the blocks have both their name and an optional set of data passed in as separate parameters to the block function call. Data inserting commands inside a block always need the data provided to the block function call of this block, not the general template rendering call.

For details, please refer to the documentation of `Kernel::Output::HTML::Layout` on otrs.github.io/doc.

6.1.4. dtl:js_on_document_complete

Marks JavaScript code which should be executed after all CSS, JavaScript and other external content has been loaded and the basic JavaScript initialization was finished. Again, let's look at an example:

```
<form title="$Text{"Move ticket to a different queue"}" action="$Env{"CGIHandle"}"
 method="get">
    <input type="hidden" name="Action" value="AgentTicketMove"/>
    <input type="hidden" name="QueueID" value="$QData{"QueueID"}"/>
    <input type="hidden" name="TicketID" value="$QData{"TicketID"}"/>
    <label for="DestQueueID" class="InvisibleText">$Text{"Change queue"}:</label>
    $Data{"MoveQueuesStrg"}
</form>
<!-- dtl:js_on_document_complete -->
<script type="text/javascript">
    $('#DestQueueID').bind('change', function (Event) {
        $(this).closest('form').submit();
    });
</script>
<!-- dtl:js_on_document_complete -->
```

This snippet creates a small form and puts an onchange-Handler on the <select> element which causes and automatical form submit.

Why is it neccessary to enclose the JavaScript code in `dtl:js_on_document_complate`? Starting with OTRS 3.0, JavaScript loading was moved to the footer part of the page for performance reasons. This means that within the <body> of the page, no JavaScript libraries are loaded yet. With `dtl:js_on_document_complete` you can make sure that this JavaScript is moved to a part of the final HTML document, where it will be executed only after the entire external JavaScript and CSS content has been successfully loaded and initialized.

Inside the `dtl:js_on_document_complete` block, you can use <script> tags to enclose your JavaScript code, but you do not have to do so. It may be beneficial because it will enable correct syntax highlighting in IDEs which support it.

6.2. Using a template file

Ok, but how to actually process a template file and generate the result? This is really simple:

```
# render AdminState.dtl
$Output .= $Self->{LayoutObject}->Output(
    TemplateFile => 'AdminState',
    Data         => \%Param,
);
```

In the frontend modules, the `Output()` function of `Kernel::Output::HTML::Layout` is called (after all the needed blocks have been called in this template) to generate the final output. An optional set of data parameters is passed to the template, for all data inserting commands which are not inside of a block.

Chapter 3. How to Extend OTRS

1. Writing a new OTRS frontend module

In this chapter, the writing of a new OTRS module is illustrated on the basis of a simple small program. Necessary prerequisite is an OTRS development environment as specified in the chapter of the same name.

1.1. What we want to write

We want to write a little OTRS module that displays the text 'Hello World' when called up. First of all we must build the directory /Hello World for the module in the developer directory. In this directory, all directories existent in OTRS can be created. Each module should at least contain the following directories:

Kernel/

Kernel/System/

Kernel/Modules/

Kernel/Output/HTML/Standard/

Kernel/Config/

Kernel/Config/Files/

Kernel/Language/

1.2. Default Config File

The creation of a module registration facilitates the display of the new module in OTRS. Therefore we create a file '/Kernel/Config/Files/HelloWorld.xml'. In this file, we create a new config element. The impact of the various settings is described in the chapter 'Config Mechanism'.

```
<?xml version="1.0" encoding="UTF-8" ?>
<otrs_config version="1.0" init="Application">
        <ConfigItem Name="Frontend::Module###AgentHelloWorld" Required="1" Valid="1">
        <Description Translatable="1">FrontendModuleRegistration for HelloWorld module.</
Description>
        <Group>HelloWorld</Group>
        <SubGroup>Frontend::Agent::ModuleRegistration</SubGroup>
        <Setting>
            <FrontendModuleReg>
                <Title>HelloWorld</Title>
                <Group>users</Group>
                <Description>HelloWorld</Description>
                <NavBarName>HelloWorld</NavBarName>
                <NavBar>
                    <Description>HelloWorld</Description>
                    <Name>HelloWorld</Name>
                    <Link>Action=AgentHelloWorld</Link>
                    <NavBar>HelloWorld</NavBar>
                    <Type>Menu</Type>
                    <Prio>8400</Prio>
```

```
                    <Block>ItemArea</Block>
                </NavBar>
            </FrontendModuleReg>
        </Setting>
    </ConfigItem>
</otrs_config>
```

1.3. Frontend Module

After creating the links and executing the Sysconfig, a new module with the name 'HelloWorld' is displayed. When calling it up, an error message is displayed as OTRS cannot find the matching frontend module yet. This is the next thing to be created. To do so, we create the following file:

```
# --
# Kernel/Modules/AgentHelloWorld.pm - frontend module
# Copyright (C) (year) (name of author) (email of author)
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Modules::AgentHelloWorld;

use strict;
use warnings;

use Kernel::System::HelloWorld;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {%Param};
    bless ($Self, $Type);

    # check needed objects
    for (qw(ParamObject DBObject TicketObject LayoutObject LogObject QueueObject
 ConfigObject EncodeObject MainObject)) {
        if ( !$Self->{$_} ) {
            $Self->{LayoutObject}->FatalError( Message => "Got no $_!" );
        }
    }

    # create needed objects
    $Self->{HelloWorldObject} = Kernel::System::HelloWorld->new(%Param);

    return $Self;
}

sub Run {
    my ( $Self, %Param ) = @_;
    my %Data = ();

    $Data{HelloWorldText} = $Self->{HelloWorldObject}->GetHelloWorldText();

    # build output
    my $Output = $Self->{LayoutObject}->Header(Title => "HelloWorld");
    $Output   .= $Self->{LayoutObject}->NavigationBar();
    $Output   .= $Self->{LayoutObject}->Output(
        Data => \%Data,
        TemplateFile => 'AgentHelloWorld',
    );
    $Output   .= $Self->{LayoutObject}->Footer();
    return $Output;
}
```

```
1;
```

1.4. Core Module

Next, we create the file for the core module "/HelloWorld/Kernel/System/HelloWorld.pm" with the following content:

```
# --
# Kernel/System/HelloWorld.pm - core module
# Copyright (C) (year) (name of author) (email of author)
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::HelloWorld;

use strict;
use warnings;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless ($Self, $Type);

    return $Self;
}

sub GetHelloWorldText {
    my ( $Self, %Param ) = @_;

    return 'Hello World';
}

1;
```

1.5. dtl Template File

The last thing missing before the new module can run is the relevant HTML template. Thus, we create the following file:

```
# --
# Kernel/Output/HTML/Standard/AgentHelloWorld.dtl - overview
# Copyright (C) (year) (name of author) (email of author)
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --
<!-- start form -->
<table border="0" width="100%" cellspacing="0" cellpadding="3">
    <tr>
        <td class="mainhead">
            $Env{"Box0"}$Text{"Overview"}: $Text{"HelloWorld"}$Env{"Box1"}
        </td>
    </tr>
    <tr>
        <td class="mainbody">
            <br>
            $Text{"$QData{"HelloWorldText"}"}!<br>
            <br>
```

```
            <br>
        </td>
    </tr>
</table>
<!-- end form -->
```

The module is working now and displays the text 'Hello World' when called.

1.6. Language File

If the text 'Hello World' is to be translated into for instance German, you can create a translation file for this language: '/HelloWorld/Kernel/Language/de_AgentHelloWorld.pm'. Example:

```
package Kernel::Language::de_AgentHelloWorld;

use strict;
use warnings;

sub Data {
    my $Self = shift;

    $Self->{Translation}->{'Hello World'} = 'Hallo Welt';

    return 1;
}
1;
```

1.7. Summary

The example given above shows that it is not too difficult to write a new module for OTRS. It is important, though, to make sure that the module and file name are unique and thus do not interfere with the framework or other expansion modules. When a module is finished, an OPM package must be generated from it (see chapter Package Building).

2. Using the power of the OTRS module layers

OTRS has a large number of so-called "module layers" which make it very easy to extend the system without patching existing code. One example is the number generation mechanism for tickets. It is a "module layer" with pluggable modules, and you can add your own custom number generator modules if you wish to do so. Let's look at the different layers in detail!

2.1. Authentication and user management

2.1.1. Agent Authentication Module

There are several agent authentication modules (DB, LDAP and HTTPBasicAuth) which come with the OTRS framework. It is also possible to develop your own authentication modules. The agent authentication modules are located under Kernel/System/Auth/*.pm. For more information about their configuration see the admin manual. Following, there is an example of a simple agent auth module. Save it under Kernel/System/Auth/Simple.pm. You just need 3 functions: new(), GetOption() and Auth(). Return the uid, then the authentication is ok.

2.1.1.1. Code Example

The interface class is called Kernel::System::Auth. The example agent authentication may be called Kernel::System::Auth::CustomAuth. You can find an example below.

```
# --
# Kernel/System/Auth/CustomAuth.pm - provides the CustomAuth authentication
# based on Martin Edenhofer's Kernel::System::Auth::DB
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# ID: CustomAuth.pm,v 1.1 2010/05/10 15:30:34 fk Exp $
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::Auth::CustomAuth;

use strict;
use warnings;

use Authen::CustomAuth;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for (qw(LogObject ConfigObject DBObject)) {
        $Self->{$_} = $Param{$_} || die "No $_!";
    }

    # Debug 0=off 1=on
    $Self->{Debug} = 0;

    # get config
    $Self->{Die} = $Self->{ConfigObject}->Get( 'AuthModule::CustomAuth::Die' .
 $Param{Count} );

    # get user table
    $Self->{CustomAuthHost} = $Self->{ConfigObject}->Get( 'AuthModule::CustomAuth::Host' .
 $Param{Count} )
        || die "Need AuthModule::CustomAuth::Host$Param{Count}.";
    $Self->{CustomAuthSecret}
        = $Self->{ConfigObject}->Get( 'AuthModule::CustomAuth::Password' . $Param{Count} )
        || die "Need AuthModule::CustomAuth::Password$Param{Count}.";

    return $Self;
}

sub GetOption {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    if ( !$Param{What} ) {
        $Self->{LogObject}->Log( Priority => 'error', Message => "Need What!" );
        return;
    }

    # module options
    my %Option = ( PreAuth => 0, );

    # return option
    return $Option{ $Param{What} };
}
```

```
sub Auth {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    if ( !$Param{User} ) {
        $Self->{LogObject}->Log( Priority => 'error', Message => "Need User!" );
        return;
    }

    # get params
    my $User       = $Param{User}      || '';
    my $Pw         = $Param{Pw}        || '';
    my $RemoteAddr = $ENV{REMOTE_ADDR} || 'Got no REMOTE_ADDR env!';
    my $UserID     = '';
    my $GetPw      = '';

    # just in case for debug!
    if ( $Self->{Debug} > 0 ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: '$User' tried to authenticate with Pw: '$Pw' ($RemoteAddr)",
        );
    }

    # just a note
    if ( !$User ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "No User given!!! (REMOTE_ADDR: $RemoteAddr)",
        );
        return;
    }

    # just a note
    if ( !$Pw ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: $User authentication without Pw!!! (REMOTE_ADDR:
 $RemoteAddr)",
        );
        return;
    }

    # Create a radius object
    my $CustomAuth = Authen::CustomAuth->new(
        Host   => $Self->{CustomAuthHost},
        Secret => $Self->{CustomAuthecret},
    );
    if ( !$CustomAuth ) {
        if ( $Self->{Die} ) {
            die "Can't connect to $Self->{CustomAuthHost}: $@";
        }
        else {
            $Self->{LogObject}->Log(
                Priority => 'error',
                Message  => "Can't connect to $Self->{CustomAuthHost}: $@",
            );
            return;
        }
    }
    my $AuthResult = $CustomAuth->check_pwd( $User, $Pw );

    # login note
    if ( defined($AuthResult) && $AuthResult == 1 ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: $User authentication ok (REMOTE_ADDR: $RemoteAddr).",
        );
        return $User;
    }

    # just a note
```

```
    else {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: $User authentication with wrong Pw!!! (REMOTE_ADDR:
 $RemoteAddr)"
        );
        return;
    }
}

1;
```

2.1.1.2. Configuration Example

There is the need to activate your custom agent authenticate module. This can be done using the perl configuration below. It is not recommended to use the xml configuration because you can lock you out via the sysconfig.

```
            $Self->{'AuthModule'} = 'Kernel::System::Auth::CustomAuth';
```

2.1.1.3. Use Case Example

A useful example of an authentication implementation could be a soap backend.

2.1.1.4. Release Availability

Name	Release
DB	1.0
HTTPBasicAuth	1.2
LDAP	1.0
Radius	1.3

2.1.2. Authentication Synchronisation Module

There is a LDAP authentication synchronisation module which come with the OTRS framework. It is also possible to develop your own authentication modules. The authentication synchronisation modules are located under Kernel/System/Auth/Sync/*.pm. For more information about their configuration see the admin manual. Following, there is an example of an authentication synchronisation module. Save it under Kernel/System/Auth/Sync/CustomAuthSync.pm. You just need 2 functions: new() and Sync(). Return 1, then the synchronisation is ok.

2.1.2.1. Code Example

The interface class is called Kernel::System::Auth. The example agent authentication may be called Kernel::System::Auth::Sync::CustomAuthSync. You can find an example below.

```
# --
# Kernel/System/Auth/Sync/CustomAuthSync.pm - provides the CustomAuthSync
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# Id: CustomAuthSync.pm,v 1.9 2010/03/25 14:42:45 martin Exp $
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
```

```
# --

package Kernel::System::Auth::Sync::CustomAuthSync;

use strict;
use warnings;
use Net::LDAP;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for (qw(LogObject ConfigObject DBObject UserObject GroupObject EncodeObject)) {
        $Self->{$_} = $Param{$_} || die "No $_!";
    }

    # Debug 0=off 1=on
    $Self->{Debug} = 0;

...

    return $Self;
}

sub Sync {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for (qw(User)) {
        if ( !$Param{$_} ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $_!" );
            return;
        }
    }
...
    return 1;
}
```

2.1.2.2. Configuration Example

You should activate your custom synchronisation module. This can be done using the perl configuration below. It is not recommended to use the xml configuration because this would allow you to lock yourself out via SysConfig.

```
$Self->{'AuthSyncModule'} = 'Kernel::System::Auth::Sync::LDAP';
```

2.1.2.3. Use Case Examples

Useful synchronisation implementation could be a soap or radius backend.

2.1.2.4. Release Availability

Name	Release
LDAP	2.4

2.1.2.5. Caveats and Warnings

Please note that the synchronisation was part of the authentication class Kernel::System::Auth before framework 2.4.

2.1.3. Customer Authentication Module

There are several customer authentication modules (DB, LDAP and HTTPBasicAuth) which come with the OTRS framework. It is also possible to develop your own authentication modules. The customer authentication modules are located under Kernel/System/CustomerAuth/*.pm. For more information about their configuration see the admin manual. Following, there is an example of a simple customer auth module. Save it under Kernel/System/CustomerAuth/Simple.pm. You just need 3 functions: new(), GetOption() and Auth(). Return the uid, then the authentication is ok.

2.1.3.1. Code Example

The interface class is called Kernel::System::CustomerAuth. The example customer authentication may be called Kernel::System::CustomerAuth::CustomAuth. You can find an example below.

```
# --
# Kernel/System/CustomerAuth/CustomAuth.pm - provides the custom Authentication
# based on Martin Edenhofer's Kernel::System::Auth::DB
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# Id: CustomAuth.pm,v 1.11 2009/09/22 15:16:05 mb Exp $
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::CustomerAuth::CustomAuth;

use strict;
use warnings;

use Authen::CustomAuth;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for (qw(LogObject ConfigObject DBObject)) {
        $Self->{$_} = $Param{$_} || die "No $_!";
    }

    # Debug 0=off 1=on
    $Self->{Debug} = 0;

    # get config
    $Self->{Die}
        = $Self->{ConfigObject}->Get( 'Customer::AuthModule::CustomAuth::Die' .
 $Param{Count} );

    # get user table
    $Self->{CustomAuthHost}
        = $Self->{ConfigObject}->Get( 'Customer::AuthModule::CustomAuth::Host' .
 $Param{Count} )
        || die "Need Customer::AuthModule::CustomAuth::Host$Param{Count} in Kernel/
Config.pm";
    $Self->{CustomAuthSecret}
        = $Self->{ConfigObject}->Get( 'Customer::AuthModule::CustomAuth::Password' .
 $Param{Count} )
        || die "Need Customer::AuthModule::CustomAuth::Password$Param{Count} in Kernel/
Config.pm";

    return $Self;
```

```
}

sub GetOption {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    if ( !$Param{What} ) {
        $Self->{LogObject}->Log( Priority => 'error', Message => "Need What!" );
        return;
    }

    # module options
    my %Option = ( PreAuth => 0, );

    # return option
    return $Option{ $Param{What} };
}

sub Auth {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    if ( !$Param{User} ) {
        $Self->{LogObject}->Log( Priority => 'error', Message => "Need User!" );
        return;
    }

    # get params
    my $User       = $Param{User}      || '';
    my $Pw         = $Param{Pw}        || '';
    my $RemoteAddr = $ENV{REMOTE_ADDR} || 'Got no REMOTE_ADDR env!';
    my $UserID     = '';
    my $GetPw      = '';

    # just in case for debug!
    if ( $Self->{Debug} > 0 ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: '$User' tried to authentificate with Pw:
 '$Pw' ($RemoteAddr)",
        );
    }

    # just a note
    if ( !$User ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "No User given!!! (REMOTE_ADDR: $RemoteAddr)",
        );
        return;
    }

    # just a note
    if ( !$Pw ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: $User Authentication without Pw!!! (REMOTE_ADDR:
 $RemoteAddr)",
        );
        return;
    }

    # Create a custom object
    my $CustomAuth = Authen::CustomAuth->new(
        Host   => $Self->{CustomAuthHost},
        Secret => $Self->{CustomAuthSecret},
    );
    if ( !$CustomAuth ) {
        if ( $Self->{Die} ) {
            die "Can't connect to $Self->{CustomAuthHost}: $@";
        }
        else {
```

```
            $Self->{LogObject}->Log(
                Priority => 'error',
                Message  => "Can't connect to $Self->{CustomAuthHost}: $@",
            );
            return;
        }
    }
    my $AuthResult = $CustomAuth->check_pwd( $User, $Pw );

    # login note
    if ( defined($AuthResult) && $AuthResult == 1 ) {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: $User Authentication ok (REMOTE_ADDR: $RemoteAddr).",
        );
        return $User;
    }

    # just a note
    else {
        $Self->{LogObject}->Log(
            Priority => 'notice',
            Message  => "User: $User Authentication with wrong Pw!!! (REMOTE_ADDR:
 $RemoteAddr)"
        );
        return;
    }
}

1;
```

2.1.3.2. Configuration Example

There is the need to activate your custom customer authenticate module. This can be done using the xml configuration below.

```
<ConfigItem Name="AuthModule" Required="1" Valid="1">
    <Description Lang="en">Module to authenticate customers.</Description>
    <Description Lang="de">Modul zum Authentifizieren der Customer.</Description>
    <Group>Framework</Group>
    <SubGroup>Frontend::CustomerAuthAuth</SubGroup>
    <Setting>
        <Option Location="Kernel/System/CustomerAuth/*.pm"
 SelectedID="Kernel::System::CustomerAuth::CustomAuth"></Option>
    </Setting>
</ConfigItem>
```

2.1.3.3. Use Case Example

Useful authentification implementation could be a soap backend.

2.1.3.4. Release Availability

Name	Release
DB	1.0
HTTPBasicAuth	1.2
LDAP	1.0
Radius	1.3

2.2. Preferences

2.2.1. Customer User Preferences Module

There is a DB customer-user preferences module which come with the OTRS framework. It is also possible to develop your own customer-user preferences modules. The customer-user preferences modules are located under Kernel/System/CustomerUser/Preferences/*.pm. For more information about their configuration see the admin manual. There is an example of a customer-user preferences module below. Save it under Kernel/System/CustomerUser/Preferences/Custom.pm. You just need 4 functions: new(), SearchPreferences(), SetPreferences() and GetPreferences().

2.2.1.1. Code Example

The interface class is called Kernel::System::CustomerUser. The example customer-user preferences may be called Kernel::System::CustomerUser::Preferences::Custom. You can find an example below.

```
# --
# Kernel/System/CustomerUser/Preferences/Custom.pm - some customer user functions
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# Id: Custom.pm,v 1.20 2009/10/07 20:41:50 martin Exp $
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::CustomerUser::Preferences::Custom;

use strict;
use warnings;

use vars qw(@ISA $VERSION);

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for my $Object (qw(DBObject ConfigObject LogObject)) {
        $Self->{$Object} = $Param{$Object} || die "Got no $Object!";
    }

    # preferences table data
    $Self->{PreferencesTable} = $Self->{ConfigObject}->Get('CustomerPreferences')->{Params}-
>{Table}
        || 'customer_preferences';
    $Self->{PreferencesTableKey}
        = $Self->{ConfigObject}->Get('CustomerPreferences')->{Params}->{TableKey}
        || 'preferences_key';
    $Self->{PreferencesTableValue}
        = $Self->{ConfigObject}->Get('CustomerPreferences')->{Params}->{TableValue}
        || 'preferences_value';
    $Self->{PreferencesTableUserID}
        = $Self->{ConfigObject}->Get('CustomerPreferences')->{Params}->{TableUserID}
        || 'user_id';

    return $Self;
}

sub SetPreferences {
    my ( $Self, %Param ) = @_;
```

```
    my $UserID = $Param{UserID} || return;
    my $Key    = $Param{Key}    || return;
    my $Value = defined( $Param{Value} ) ? $Param{Value} : '';

    # delete old data
    return if !$Self->{DBObject}->Do(
        SQL => "DELETE FROM $Self->{PreferencesTable} WHERE "
            . " $Self->{PreferencesTableUserID} = ? AND $Self->{PreferencesTableKey} = ?",
        Bind => [ \$UserID, \$Key ],
    );

    $Value .= 'Custom';

    # insert new data
    return if !$Self->{DBObject}->Do(
        SQL => "INSERT INTO $Self->{PreferencesTable} ($Self->{PreferencesTableUserID}, "
            . " $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue}) "
            . " VALUES (?, ?, ?)",
        Bind => [ \$UserID, \$Key, \$Value ],
    );

    return 1;
}

sub GetPreferences {
    my ( $Self, %Param ) = @_;

    my $UserID = $Param{UserID} || return;
    my %Data;

    # get preferences

    return if !$Self->{DBObject}->Prepare(
        SQL => "SELECT $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue} "
            . " FROM $Self->{PreferencesTable} WHERE $Self->{PreferencesTableUserID} = ?",
        Bind => [ \$UserID ],
    );
    while ( my @Row = $Self->{DBObject}->FetchrowArray() ) {
        $Data{ $Row[0] } = $Row[1];
    }

    # return data
    return %Data;
}

sub SearchPreferences {
    my ( $Self, %Param ) = @_;

    my %UserID;
    my $Key   = $Param{Key}   || '';
    my $Value = $Param{Value} || '';

    # get preferences
    my $SQL = "SELECT $Self->{PreferencesTableUserID}, $Self->{PreferencesTableValue} "
        . " FROM "
        . " $Self->{PreferencesTable} "
        . " WHERE "
        . " $Self->{PreferencesTableKey} = '"
        . $Self->{DBObject}->Quote($Key) . "'" . " AND "
        . " LOWER($Self->{PreferencesTableValue}) LIKE LOWER('"
        . $Self->{DBObject}->Quote( $Value, 'Like' ) . "')";

    return if !$Self->{DBObject}->Prepare( SQL => $SQL );
    while ( my @Row = $Self->{DBObject}->FetchrowArray() ) {
        $UserID{ $Row[0] } = $Row[1];
    }

    # return data
    return %UserID;
}
```

```
1;
```

2.2.1.2. Configuration Example

There is the need to activate your custom customer-user preferences module. This can be done using the xml configuration below.

```
<ConfigItem Name="CustomerPreferences" Required="1" Valid="1">
    <Description Lang="en">Parameters for the customer preference table.</Description>
    <Description Lang="de">Parameter für die Tabelle mit den Einstellungen für die
 Customer.</Description>
    <Group>Framework</Group>
    <SubGroup>Frontend::Customer::Preferences</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::System::CustomerUser::Preferences::Custom</Item>
            <Item  Key="Params">
                <Hash>
                    <Item Key="Table">customer_preferences</Item>
                    <Item Key="TableKey">preferences_key</Item>
                    <Item Key="TableValue">preferences_value</Item>
                    <Item Key="TableUserID">user_id</Item>
                </Hash>
            </Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.2.1.3. Use Case Example

Useful preferences implementation could be a soap or ldap backend.

2.2.1.4. Release Availability

Name	Release
DB	2.3

2.2.2. Queue Preferences Module

There is a DB queue preferences module which come with the OTRS framework. It is also possible to develop your own queue preferences modules. The queue preferences modules are located under Kernel/System/Queue/*.pm. For more information about their configuration see the admin manual. Following, there is an example of a queue preferences module. Save it under Kernel/System/Queue/PreferencesCustom.pm. You just need 3 functions: new(), QueuePreferencesSet() and QueuePreferencesGet(). Return 1, then the synchronisation is ok.

2.2.2.1. Code Example

The interface class is called Kernel::System::Queue. The example queue preferences may be called Kernel::System::Queue::PreferencesCustom. You can find an example below.

```
# --
# Kernel/System/Queue/PreferencesCustom.pm - some user functions
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# Id: PreferencesCustom.pm,v 1.5 2009/02/16 11:47:34 tr Exp $
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --
```

```
package Kernel::System::Queue::PreferencesCustom;

use strict;
use warnings;

use vars qw(@ISA $VERSION);

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for (qw(DBObject ConfigObject LogObject)) {
        $Self->{$_} = $Param{$_} || die "Got no $_!";
    }

    # preferences table data
    $Self->{PreferencesTable}        = 'queue_preferences';
    $Self->{PreferencesTableKey}     = 'preferences_key';
    $Self->{PreferencesTableValue}   = 'preferences_value';
    $Self->{PreferencesTableQueueID} = 'queue_id';

    return $Self;
}

sub QueuePreferencesSet {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for (qw(QueueID Key Value)) {
        if ( !defined( $Param{$_} ) ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $_!" );
            return;
        }
    }

    # delete old data
    return if !$Self->{DBObject}->Do(
        SQL => "DELETE FROM $Self->{PreferencesTable} WHERE "
            . "$Self->{PreferencesTableQueueID} = ? AND $Self->{PreferencesTableKey} = ?",
        Bind => [ \$Param{QueueID}, \$Param{Key} ],
    );

    $Self->{PreferencesTableValue} .= 'PreferencesCustom';

    # insert new data
    return $Self->{DBObject}->Do(
        SQL => "INSERT INTO $Self->{PreferencesTable} ($Self->{PreferencesTableQueueID}, "
            . " $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue}) "
            . " VALUES (?, ?, ?)",
        Bind => [ \$Param{QueueID}, \$Param{Key}, \$Param{Value} ],
    );
}

sub QueuePreferencesGet {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for (qw(QueueID)) {
        if ( !$Param{$_} ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $_!" );
            return;
        }
    }

    # check if queue preferences are available
    if ( !$Self->{ConfigObject}->Get('QueuePreferences') ) {
        return;
```

```
    }

    # get preferences
    return if !$Self->{DBObject}->Prepare(
        SQL => "SELECT $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue} "
            . " FROM $Self->{PreferencesTable} WHERE $Self->{PreferencesTableQueueID} = ?",
        Bind => [ \$Param{QueueID} ],
    );
    my %Data;
    while ( my @Row = $Self->{DBObject}->FetchrowArray() ) {
        $Data{ $Row[0] } = $Row[1];
    }

    # return data
    return %Data;
}

1;
```

2.2.2.2. Configuration Example

There is the need to activate your custom queue preferences module. This can be done using the xml configuration below.

```
<ConfigItem Name="Queue::PreferencesModule" Required="1" Valid="1">
    <Description Lang="en">Default queue preferences module.</Description>
    <Description Lang="de">Standard Queue Preferences Module.</Description>
    <Group>Ticket</Group>
    <SubGroup>Frontend::Queue::Preferences</SubGroup>
    <Setting>
        <String Regex="">Kernel::System::Queue::PreferencesCustom</String>
    </Setting>
</ConfigItem>
```

2.2.2.3. Use Case Examples

Useful preferences implementation could be a soap or radius backend.

2.2.2.4. Release Availability

Name	Release
PreferencesDB	2.3

2.2.3. Service Preferences Module

There is a DB service preferences module which come with the OTRS framework. It is also possible to develop your own service preferences modules. The service preferences modules are located under Kernel/System/Service/*.pm. For more information about their configuration see the admin manual. Following, there is an example of a service preferences module. Save it under Kernel/System/Service/PreferencesCustom.pm. You just need 3 functions: new(), ServicePreferencesSet() and ServicePreferencesGet(). Return 1, then the synchronisation is ok.

2.2.3.1. Code Example

The interface class is called Kernel::System::Service. The example service preferences may be called Kernel::System::Service::PreferencesCustom. You can find an example below.

```
# --
# Kernel/System/Service/PreferencesCustom - some user functions
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
```

```
# --
# Id: PreferencesCustom.pm,v 1.2 2009/02/16 11:47:34 tr Exp $
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::Service::PreferencesCustom;

use strict;
use warnings;

use vars qw(@ISA $VERSION);

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for (qw(DBObject ConfigObject LogObject)) {
        $Self->{$_} = $Param{$_} || die "Got no $_!";
    }

    # preferences table data
    $Self->{PreferencesTable}          = 'service_preferences';
    $Self->{PreferencesTableKey}       = 'preferences_key';
    $Self->{PreferencesTableValue}     = 'preferences_value';
    $Self->{PreferencesTableServiceID} = 'service_id';

    return $Self;
}

sub ServicePreferencesSet {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for (qw(ServiceID Key Value)) {
        if ( !defined( $Param{$_} ) ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $_!" );
            return;
        }
    }

    # delete old data
    return if !$Self->{DBObject}->Do(
        SQL => "DELETE FROM $Self->{PreferencesTable} WHERE "
            . "$Self->{PreferencesTableServiceID} = ? AND $Self->{PreferencesTableKey} = ?",
        Bind => [ \$Param{ServiceID}, \$Param{Key} ],
    );

$Self->{PreferencesTableValue} .= 'PreferencesCustom';

    # insert new data
    return $Self->{DBObject}->Do(
        SQL => "INSERT INTO $Self->{PreferencesTable} ($Self->{PreferencesTableServiceID}, "
            . " $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue}) "
            . " VALUES (?, ?, ?)",
        Bind => [ \$Param{ServiceID}, \$Param{Key}, \$Param{Value} ],
    );
}

sub ServicePreferencesGet {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for (qw(ServiceID)) {
        if ( !$Param{$_} ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $_!" );
```

```
            return;
        }
    }

    # check if service preferences are available
    if ( !$Self->{ConfigObject}->Get('ServicePreferences') ) {
        return;
    }

    # get preferences
    return if !$Self->{DBObject}->Prepare(
        SQL => "SELECT $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue} "
            . " FROM $Self->{PreferencesTable} WHERE $Self->{PreferencesTableServiceID}
 = ?",
        Bind => [ \$Param{ServiceID} ],
    );
    my %Data;
    while ( my @Row = $Self->{DBObject}->FetchrowArray() ) {
        $Data{ $Row[0] } = $Row[1];
    }

    # return data
    return %Data;
}

1;
```

2.2.3.2. Configuration Example

There is the need to activate your custom service preferences module. This can be done using the xml configuration below.

```
<ConfigItem Name="Service::PreferencesModule" Required="1" Valid="1">
    <Description Lang="en">Default service preferences module.</Description>
    <Description Lang="de">Standard Service Preferences Module.</Description>
    <Group>Ticket</Group>
    <SubGroup>Frontend::Service::Preferences</SubGroup>
    <Setting>
        <String Regex="">Kernel::System::Service::PreferencesCustom</String>
    </Setting>
</ConfigItem>
```

2.2.3.3. Use Case Example

Useful preferences implementation could be a soap or radius backend.

2.2.3.4. Release Availability

Name	Release
PreferencesDB	2.4

2.2.4. SLA Preferences Module

There is a DB SLA preferences module which come with the OTRS framework. It is also possible to develop your own SLA preferences modules. The SLA preferences modules are located under Kernel/System/SLA/*.pm. For more information about their configuration see the admin manual. Here we'll show an example of an SLA preferences module. Save it under Kernel/System/SLA/PreferencesCustom.pm. You just need 3 functions: new(), SLAPreferencesSet() and SLAPreferencesGet(). Make sure the function returns 1.

2.2.4.1. Code Example

The interface class is called Kernel::System::SLA. The example SLA preferences may be called Kernel::System::SLA::PreferencesCustom. You can find an example below.

```
# --
# Kernel/System/SLA/PreferencesCustom.pm - some user functions
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::SLA::PreferencesCustom;

use strict;
use warnings;

use vars qw(@ISA);

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for (qw(DBObject ConfigObject LogObject)) {
        $Self->{$_} = $Param{$_} || die "Got no $_!";
    }

    # preferences table data
    $Self->{PreferencesTable}      = 'sla_preferences';
    $Self->{PreferencesTableKey}   = 'preferences_key';
    $Self->{PreferencesTableValue} = 'preferences_value';
    $Self->{PreferencesTableSLAID} = 'sla_id';

    return $Self;
}

sub SLAPreferencesSet {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for (qw(SLAID Key Value)) {
        if ( !defined( $Param{$_} ) ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $_!" );
            return;
        }
    }

    # delete old data
    return if !$Self->{DBObject}->Do(
        SQL => "DELETE FROM $Self->{PreferencesTable} WHERE "
            . "$Self->{PreferencesTableSLAID} = ? AND $Self->{PreferencesTableKey} = ?",
        Bind => [ \$Param{SLAID}, \$Param{Key} ],
    );

$Self->{PreferencesTableValue} .= 'PreferencesCustom';

    # insert new data
    return $Self->{DBObject}->Do(
        SQL => "INSERT INTO $Self->{PreferencesTable} ($Self->{PreferencesTableSLAID}, "
            . " $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue}) "
            . " VALUES (?, ?, ?)",
        Bind => [ \$Param{SLAID}, \$Param{Key}, \$Param{Value} ],
    );
}
```

```
sub SLAPreferencesGet {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for (qw(SLAID)) {
        if ( !$Param{$_} ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $_!" );
            return;
        }
    }

    # check if service preferences are available
    if ( !$Self->{ConfigObject}->Get('SLAPreferences') ) {
        return;
    }

    # get preferences
    return if !$Self->{DBObject}->Prepare(
        SQL => "SELECT $Self->{PreferencesTableKey}, $Self->{PreferencesTableValue} "
            . " FROM $Self->{PreferencesTable} WHERE $Self->{PreferencesTableSLAID} = ?",
        Bind => [ \$Param{SLAID} ],
    );
    my %Data;
    while ( my @Row = $Self->{DBObject}->FetchrowArray() ) {
        $Data{ $Row[0] } = $Row[1];
    }

    # return data
    return %Data;
}

1;
```

2.2.4.2. Configuration Example

You need to register your custom SLA preferences module. This can be done using the xml configuration below.

```
<ConfigItem Name="SLA::PreferencesModule" Required="1" Valid="1">
    <Description Translatable="1">Default SLA preferences module.</Description>
    <Group>Ticket</Group>
    <SubGroup>Frontend::SLA::Preferences</SubGroup>
    <Setting>
        <String Regex="">Kernel::System::SLA::PreferencesCustom</String>
    </Setting>
</ConfigItem>
```

2.2.4.3. Use Case Example

Useful preferences implementation could be to store additional values on SLAs.

2.2.4.4. Release Availability

Name	Release
PreferencesDB	2.4

2.3. Other core functions

2.3.1. Log Module

There is a gobal log interface for OTRS that provides the possibility to create own log backends.

Writing an own logging backend is as easy as reimplementing the Kernel::System::Log::Log() method.

2.3.1.1. Code example: Kernel::System::Log::CustomFile

In this small example, we'll write a little file logging backend which works similar to Kernel::System::Log::File, but prepends a string to each logging entry.

```
# --
# Kernel/System/Log/CustomFile.pm - file log backend
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::Log::CustomFile;

use strict;
use warnings;

umask "002";

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # get needed objects
    for (qw(ConfigObject EncodeObject)) {
        if ( $Param{$_} ) {
            $Self->{$_} = $Param{$_};
        }
        else {
            die "Got no $_!";
        }
    }

    # get logfile location
    $Self->{LogFile} = '/var/log/CustomFile.log';

    # set custom prefix
    $Self->{CustomPrefix} = 'CustomFileExample';

    # Fixed bug# 2265 - For IIS we need to create a own error log file.
    # Bind stderr to log file, because iis do print stderr to web page.
    if ( $ENV{SERVER_SOFTWARE} && $ENV{SERVER_SOFTWARE} =~ /^microsoft\-iis/i ) {
        if ( !open STDERR, '>>', $Self->{LogFile} . '.error' ) {
            print STDERR "ERROR: Can't write $Self->{LogFile}.error: $!";
        }
    }

    return $Self;
}

sub Log {
    my ( $Self, %Param ) = @_;

    my $FH;

    # open logfile
    if ( !open $FH, '>>', $Self->{LogFile} ) {

        # print error screen
        print STDERR "\n";
        print STDERR " >> Can't write $Self->{LogFile}: $! <<\n";
        print STDERR "\n";
```

```
        return;
    }

    # write log file
    $Self->{EncodeObject}->SetIO($FH);
    print $FH '[' . localtime() . ']';
    if ( lc $Param{Priority} eq 'debug' ) {
        print $FH "[Debug][$Param{Module}][$Param{Line}] $Self->{CustomPrefix}
 $Param{Message}\n";
    }
    elsif ( lc $Param{Priority} eq 'info' ) {
        print $FH "[Info][$Param{Module}]  $Self->{CustomPrefix} $Param{Message}\n";
    }
    elsif ( lc $Param{Priority} eq 'notice' ) {
        print $FH "[Notice][$Param{Module}] $Self->{CustomPrefix} $Param{Message}\n";
    }
    elsif ( lc $Param{Priority} eq 'error' ) {
        print $FH "[Error][$Param{Module}][$Param{Line}] $Self->{CustomPrefix}
 $Param{Message}\n";
    }
    else {

        # print error messages to STDERR
        print STDERR
            "[Error][$Param{Module}] $Self->{CustomPrefix} Priority: '$Param{Priority}' not
 defined! Message: $Param{Message}\n";

        # and of course to logfile
        print $FH
            "[Error][$Param{Module}] $Self->{CustomPrefix} Priority: '$Param{Priority}' not
 defined! Message: $Param{Message}\n";
    }

    # close file handle
    close $FH;
    return 1;
}

1;
```

2.3.1.2. Configuration example

To activate our custom logging module, the administrator can either set the existing configuration item "LogModule" manually to "Kernel::System::Log::CustomFile". To realize this automatically, you can provide an XML configuration file which overrides the default setting.

```
<ConfigItem Name="LogModule" Required="1" Valid="1">
    <Description Translatable="1">Set Kernel::System::Log::CustomFile as default logging
 backend.</Description>
    <Group>Framework</Group>
    <SubGroup>Core::Log</SubGroup>
    <Setting>
        <Option Location="Kernel/System/Log/*.pm"
 SelectedID="Kernel::System::Log::CustomFile"></Option>
    </Setting>
</ConfigItem>
```

2.3.1.3. Use case examples

Useful logging backends could be logging to a web service or to encrypted files.

2.3.1.4. Caveats and Warnings

Please note that Kernel::System::Log has other methods than Log() which cannot be reimplemented, for example code for working with shared memory segments and log data caching.

2.3.2. Output Filter

Output filters allow to modify HTML on the fly. It is best practice to use output filters instead of modifying `.dtl` files directly. There are three good reasons for that. When the same adaption has to be applied to several frontend modules then the adaption only has to be implemented once. The second advantage is that when OTRS is upgraded there is a chance that the filter doesn't have to be updated, when the relevant pattern has not changed. When two extensions modify the same file there is a conflict during the installation of the second package. This conflict can be resolved by using two output filters that modify the same frontend module.

There are four different kinds of output filters. They are active at different stages of the generation of HTML content.

2.3.2.1. FilterElementPre

The content of a template can be changed by the filter before any processing by the Layout module takes place. This kind of filter should be used in most cases. Processing instructions like $Text{"..."}, $QData{"..."} can be inserted into the template content and they will be honored by the subsequent DTL processing.

2.3.2.2. FilterElementPost

The content of a template can be changed after variable substitution and translation. The kind of filter should only be used when the filter needs access to translated strings or to substituted variables.

2.3.2.3. FilterContent

This kind of filter allows to process the complete HTML output for the request right before it is sent to the browser. This can be used for global transformations. But in real life there is rarely a need to use this kind of filter.

2.3.2.4. FilterText

This kind of output filter is a plugin for the method Kernel::Output::HTML::Layout::Ascii2HTML() and is only active when the parameter LinkFeature is set to 1. Thus the FilterText output filters are currently only active for the display of the body of plain text articles. Plain text articles are generated by incoming non-HTML mails and when OTRS is configured to not use the rich text feature in the frontend.

2.3.2.5. Code example

See package TemplateModule.

2.3.2.6. Configuration example

See package TemplateModule.

2.3.2.7. Use Cases

2.3.2.7.1. Show additional ticket attributes in AgentTicketZoom.

All ticket attributes are passed to the AgentTicketZoom template. Therefore it suffices to insert e.g. the instruction $QData{"Title"} into the content. This can be achieved with a FilterElementPre output filter.

2.3.2.7.2. Add an additional CSS file.

An additional CSS file can be added to all agent frontends with an FilterElementPre filter that only modifies `Header.dtl`. Therefore it suffices to insert e.g. the instruction $QData{"Title"} into the content. This can be achieved with a FilterElementPre output filter.

2.3.2.7.3. Show the service selection as a multi level menu.

Use a FilterElementPost for this feature. The list of selectable services can be parsed from the processed template output. The multi level selection can be constructed from the service list and inserted into the template content. A FilterElementPost output filter must be used for that.

2.3.2.7.4. Create links within plain text article bodies.

A biotech company uses gene names like IPI00217472 in plain text articles. A Filter-Text output filter can be used to create links to a sequence database, e.g. http://srs.e-bi.ac.uk/srsbin/cgi-bin/wgetz?-e+[IPI-acc:IPI00217472]+-vn+2, for the gene names.

2.3.2.7.5. Prohibit active content

There is firewall rule that disallows all active content. In order to avoid rejection by the firewall the HTML tag <applet> can be filtered with an FilterContent output filter.

2.3.2.8. Caveats and Warnings

Every ElementPre and ElementPost output filter is constructed and run for every Template that is needed for the current request. Thus low performance of the output filter or a large number of filters can severely degrade performance. When that becomes an issue, the construction of needed objects can be done in the Run-method after the checks. Thus the expensive code is run only in the relevant cases.

2.3.2.9. Best Practices

In order to increase flexibility the list of affected templates should be configurable in SysConfig.

2.3.2.10. Release Availability

The four kinds of output filters are available in OTRS 2.4.

2.3.3. Stats Module

There are two different types of internal stats modules - dynamic and static. This section describes how such stats modules can be developed.

2.3.3.1. Dynamic Stats

In contrast to static stats modules, dynamic statistics can be configured via the OTRS web interface. In this section a simple statistic module is developed. Each dynamic stats module has to implement these subroutines

- new
- GetObjectName
- GetObjectAttributes
- ExportWrapper
- ImportWrapper

Furthermore the module has to implement either GetStatElement or GetStatTable. And if the header line of the result table should be changed, a sub called GetHeaderLine has to be developed.

2.3.3.1.1. Code example

In this section a sample stats module is shown and each subroutine is explained.

```
# --
# Kernel/System/Stats/Dynamic/DynamicStatsTemplate.pm - all advice functions
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::Stats::Dynamic::DynamicStatsTemplate;

use strict;
use warnings;

use Kernel::System::Queue;
use Kernel::System::State;
use Kernel::System::Ticket;
```

This is common boilerplate that can be found in common OTRS modules. The class/package name is declared via the package keyword. Then the needed modules are 'use'd.

```
sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for my $Object (
        qw(DBObject ConfigObject LogObject UserObject TimeObject MainObject EncodeObject)
        )
    {
        $Self->{$Object} = $Param{$Object} || die "Got no $Object!";
    }

    # created needed objects
    $Self->{QueueObject}    = Kernel::System::Queue->new( %{$Self} );
    $Self->{TicketObject}   = Kernel::System::Ticket->new( %{$Self} );
    $Self->{StateObject}    = Kernel::System::State->new( %{$Self} );

    return $Self;
}
```

new is the constructor for this statistic module. It creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module have to be created in "new". In lines 27 to 29 the object of the stats module is created. Lines 31 to 37 check if objects that are needed in this code - either for creating other objects or in this module - are passed. After that the other objects are created.

```
sub GetObjectName {
    my ( $Self, %Param ) = @_;

    return 'Sample Statistics';
}
```

GetObjectName returns a Name for the Statistics module. This is the label that is shown in the drop down in the configuration as well as in the list of existing statistics (column "object").

```
sub GetObjectAttributes {
    my ( $Self, %Param ) = @_;
```

```
    # get state list
    my %StateList = $Self->{StateObject}->StateList(
        UserID => 1,
    );

    # get queue list
    my %QueueList = $Self->{QueueObject}->GetAllQueues();

    # get current time to fix bug#3830
    my $TimeStamp = $Self->{TimeObject}->CurrentTimestamp();
    my ($Date) = split /\s+/, $TimeStamp;
    my $Today = sprintf "%s 23:59:59", $Date;

    my @ObjectAttributes = (
        {
            Name             => 'State',
            UseAsXvalue      => 1,
            UseAsValueSeries => 1,
            UseAsRestriction => 1,
            Element          => 'StateIDs',
            Block            => 'MultiSelectField',
            Values           => \%StateList,
        },
        {
            Name             => 'Created in Queue',
            UseAsXvalue      => 1,
            UseAsValueSeries => 1,
            UseAsRestriction => 1,
            Element          => 'CreatedQueueIDs',
            Block            => 'MultiSelectField',
            Translation      => 0,
            Values           => \%QueueList,
        },
        {
            Name             => 'Create Time',
            UseAsXvalue      => 1,
            UseAsValueSeries => 1,
            UseAsRestriction => 1,
            Element          => 'CreateTime',
            TimePeriodFormat => 'DateInputFormat',    # 'DateInputFormatLong',
            Block            => 'Time',
            TimeStop         => $Today,
            Values           => {
                TimeStart => 'TicketCreateTimeNewerDate',
                TimeStop  => 'TicketCreateTimeOlderDate',
            },
        },
    );

    return @ObjectAttributes;
}
```

In this sample stats module, we want to provide three attributes the user can chose from: A list of queues, a list of states an a time drop down. To get the values shown in the drop down, some operations are needed. In this case call StateList and GetAllQueues.

Then the list of attributes is created. Each attribute is defined via a hashreference. You can use these keys:

- Name

  the label in the web interface

- UseAsXvalue

  Can this attribute be used on the x-axis

- UseAsValueSeries

Can this attribute be used on the y-axis

- UseAsRestriction

  Can this attribute be used for restrictions.

- Element

  the HTML fieldname

- Block

  the block name in the template file (e.g. <OTRS_HOME>/Kernel/Output/HTML/Standard/AgentStatsEditXaxis.dtl)

- Values

  the values shown in the attribute

Hint: If you install this sample an you configured a statistic with some queues - lets say 'queue A' and 'queue B' - then these queues are the only ones that are shown to the user when he starts the statistic. Sometimes a dynamic drop down or multiselect field is needed. In this case, you can set "SelectedValues" in the definition of the attribute:

```
        {
            Name             => 'Created in Queue',
            UseAsXvalue      => 1,
            UseAsValueSeries => 1,
            UseAsRestriction => 1,
            Element          => 'CreatedQueueIDs',
            Block            => 'MultiSelectField',
            Translation      => 0,
            Values           => \%QueueList,
            SelectedValues   => [ @SelectedQueues ],
        },
```

```
sub GetStatElement {
    my ( $Self, %Param ) = @_;

    # search tickets
    return $Self->{TicketObject}->TicketSearch(
        UserID     => 1,
        Result     => 'COUNT',
        Permission => 'ro',
        Limit      => 100_000_000,
        %Param,
    );
}
```

GetStatElement gets called for each cell in the result table. So it should be a numeric value. In this sample it does a simple ticket search. The hash %Param contains information about the "current" x-value and the y-value as well as any restrictions. So, for a cell that should count the created tickets for queue 'Misc' with state 'open' the passed parameter hash looks something like this:

```
    'CreatedQueueIDs' => [
        '4'
    ],
    'StateIDs' => [
        '2'
    ]
```

If the "per cell" calculation should be avoided, GetStatTable is an alternative. GetStatTable returns a list of rows, hence an array of arrayreferences. This leads to the same result as using GetStatElement

```
sub GetStatTable {
    my ( $Self, %Param ) = @_;

    my @StatData;

    for my $StateName ( keys %{ $Param{TableStructure} } ) {
        my @Row;
        for my $Params ( @{ $Param{TableStructure}->{$StateName} } ) {
            my $Tickets = $Self->{TicketObject}->TicketSearch(
                UserID     => 1,
                Result     => 'COUNT',
                Permission => 'ro',
                Limit      => 100_000_000,
                %{$Params},
            );

            push @Row, $Tickets;
        }

        push @StatData, [ $StateName, @Row ];
    }

    return @StatData;
}
```

GetStatTable gets all information about the stats query that is needed. The passed parameters contains information about the attributes (Restrictions, attributes that are used for x/y-axis) and the table structure. The table structure is a hash reference where the keys are the values of the y-axis and their values are hashreferences with the parameters used for GetStatElement subroutines.

```
    'Restrictions' => {},
    'TableStructure' => {
        'closed successful' => [
            {
                'CreatedQueueIDs' => [
                    '3'
                ],
                'StateIDs' => [
                    '2'
                ]
            },
        ],
        'closed unsuccessful' => [
            {
                'CreatedQueueIDs' => [
                    '3'
                ],
                'StateIDs' => [
                    '3'
                ]
            },
        ],
    },
    'ValueSeries' => [
        {
            'Block' => 'MultiSelectField',
            'Element' => 'StateIDs',
            'Name' => 'State',
            'SelectedValues' => [
                '5',
                '3',
                '2',
                '1',
```

```
                '4'
            ],
            'Translation' => 1,
            'Values' => {
                '1' => 'new',
                '10' => 'closed with workaround',
                '2' => 'closed successful',
                '3' => 'closed unsuccessful',
                '4' => 'open',
                '5' => 'removed',
                '6' => 'pending reminder',
                '7' => 'pending auto close+',
                '8' => 'pending auto close-',
                '9' => 'merged'
            }
        }
    ],
    'XValue' => {
        'Block' => 'MultiSelectField',
        'Element' => 'CreatedQueueIDs',
        'Name' => 'Created in Queue',
        'SelectedValues' => [
            '3',
            '4',
            '1',
            '2'
        ],
        'Translation' => 0,
        'Values' => {
            '1' => 'Postmaster',
            '2' => 'Raw',
            '3' => 'Junk',
            '4' => 'Misc'
        }
    }
```

Sometimes the headers of the table have to be changed. In that case, a subroutine called GetHeaderLine has to be implemented. That subroutine has to return an arrayreference with the column headers as elements. It gets information about the x-values passed.

```
sub GetHeaderLine {
    my ( $Self, %Param ) = @_;

    my @HeaderLine = ('');
    for my $SelectedXValue ( @{ $Param{XValue}->{SelectedValues} } ) {
        push @HeaderLine, $Param{XValue}->{Values}->{$SelectedXValue};
    }

    return \@HeaderLine;
}
```

```
sub ExportWrapper {
    my ( $Self, %Param ) = @_;

    # wrap ids to used spelling
    for my $Use (qw(UseAsValueSeries UseAsRestriction UseAsXvalue)) {
        ELEMENT:
        for my $Element ( @{ $Param{$Use} } ) {
            next ELEMENT if !$Element || !$Element->{SelectedValues};
            my $ElementName = $Element->{Element};
            my $Values      = $Element->{SelectedValues};

            if ( $ElementName eq 'QueueIDs' || $ElementName eq 'CreatedQueueIDs' ) {
                ID:
                for my $ID ( @{$Values} ) {
                    next ID if !$ID;
                    $ID->{Content} = $Self->{QueueObject}->QueueLookup( QueueID => $ID-
>{Content} );
```

```
                }
            }
            elsif ( $ElementName eq 'StateIDs' || $ElementName eq 'CreatedStateIDs' ) {
                my %StateList = $Self->{StateObject}->StateList( UserID => 1 );
                ID:
                for my $ID ( @{$Values} ) {
                    next ID if !$ID;
                    $ID->{Content} = $StateList{ $ID->{Content} };
                }
            }
        }
    }
    return \%Param;
}
```

Configured statistics can be exported into XML format. But as queues with the same queue names can have different IDs on different OTRS instances it would be quite painful to export the IDs (the statistics would calculate the wrong numbers then). So an export wrapper should be written to use the names instead of ids. This should be done for each "dimension" of the stats module (x-axis, y-axis and restrictions).

ImportWrapper works the other way around - it converts the name to the ID in the instance the configuration is imported to.

This is a sample export:

```
<?xml version="1.0" encoding="utf-8"?>

<otrs_stats>
<Cache>0</Cache>
<Description>Sample stats module</Description>
<File></File>
<Format>CSV</Format>
<Format>Print</Format>
<Object>DeveloperManualSample</Object>
<ObjectModule>Kernel::System::Stats::Dynamic::DynamicStatsTemplate</ObjectModule>
<ObjectName>Sample Statistics</ObjectName>
<Permission>stats</Permission>
<StatType>dynamic</StatType>
<SumCol>0</SumCol>
<SumRow>0</SumRow>
<Title>Sample 1</Title>
<UseAsValueSeries Element="StateIDs" Fixed="1">
<SelectedValues>removed</SelectedValues>
<SelectedValues>closed unsuccessful</SelectedValues>
<SelectedValues>closed successful</SelectedValues>
<SelectedValues>new</SelectedValues>
<SelectedValues>open</SelectedValues>
</UseAsValueSeries>
<UseAsXvalue Element="CreatedQueueIDs" Fixed="1">
<SelectedValues>Junk</SelectedValues>
<SelectedValues>Misc</SelectedValues>
<SelectedValues>Postmaster</SelectedValues>
<SelectedValues>Raw</SelectedValues>
</UseAsXvalue>
<Valid>1</Valid>
</otrs_stats>
```

Now, that all subroutines are explained, this is the complete sample stats module.

```
# --
# Kernel/System/Stats/Dynamic/DynamicStatsTemplate.pm - all advice functions
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
```

```
# --

package Kernel::System::Stats::Dynamic::DynamicStatsTemplate;

use strict;
use warnings;

use Kernel::System::Queue;
use Kernel::System::State;
use Kernel::System::Ticket;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for my $Object (
        qw(DBObject ConfigObject LogObject UserObject TimeObject MainObject EncodeObject)
        )
    {
        $Self->{$Object} = $Param{$Object} || die "Got no $Object!";
    }

    # created needed objects
    $Self->{QueueObject}    = Kernel::System::Queue->new( %{$Self} );
    $Self->{TicketObject}   = Kernel::System::Ticket->new( %{$Self} );
    $Self->{StateObject}    = Kernel::System::State->new( %{$Self} );

    return $Self;
}

sub GetObjectName {
    my ( $Self, %Param ) = @_;

    return 'Sample Statistics';
}

sub GetObjectAttributes {
    my ( $Self, %Param ) = @_;

    # get state list
    my %StateList = $Self->{StateObject}->StateList(
        UserID => 1,
    );

    # get queue list
    my %QueueList = $Self->{QueueObject}->GetAllQueues();

    # get current time to fix bug#3830
    my $TimeStamp = $Self->{TimeObject}->CurrentTimestamp();
    my ($Date) = split /\s+/, $TimeStamp;
    my $Today = sprintf "%s 23:59:59", $Date;

    my @ObjectAttributes = (
        {
            Name             => 'State',
            UseAsXvalue      => 1,
            UseAsValueSeries => 1,
            UseAsRestriction => 1,
            Element          => 'StateIDs',
            Block            => 'MultiSelectField',
            Values           => \%StateList,
        },
        {
            Name             => 'Created in Queue',
            UseAsXvalue      => 1,
            UseAsValueSeries => 1,
            UseAsRestriction => 1,
            Element          => 'CreatedQueueIDs',
```

```
            Block            => 'MultiSelectField',
            Translation      => 0,
            Values           => \%QueueList,
        },
        {
            Name             => 'Create Time',
            UseAsXvalue      => 1,
            UseAsValueSeries => 1,
            UseAsRestriction => 1,
            Element          => 'CreateTime',
            TimePeriodFormat => 'DateInputFormat',    # 'DateInputFormatLong',
            Block            => 'Time',
            TimeStop         => $Today,
            Values           => {
                TimeStart => 'TicketCreateTimeNewerDate',
                TimeStop  => 'TicketCreateTimeOlderDate',
            },
        },
    );

    return @ObjectAttributes;
}

sub GetStatElement {
    my ( $Self, %Param ) = @_;

    # search tickets
    return $Self->{TicketObject}->TicketSearch(
        UserID     => 1,
        Result     => 'COUNT',
        Permission => 'ro',
        Limit      => 100_000_000,
        %Param,
    );
}

sub ExportWrapper {
    my ( $Self, %Param ) = @_;

    # wrap ids to used spelling
    for my $Use (qw(UseAsValueSeries UseAsRestriction UseAsXvalue)) {
        ELEMENT:
        for my $Element ( @{ $Param{$Use} } ) {
            next ELEMENT if !$Element || !$Element->{SelectedValues};
            my $ElementName = $Element->{Element};
            my $Values      = $Element->{SelectedValues};

            if ( $ElementName eq 'QueueIDs' || $ElementName eq 'CreatedQueueIDs' ) {
                ID:
                for my $ID ( @{$Values} ) {
                    next ID if !$ID;
                    $ID->{Content} = $Self->{QueueObject}->QueueLookup( QueueID => $ID-
>{Content} );
                }
            }
            elsif ( $ElementName eq 'StateIDs' || $ElementName eq 'CreatedStateIDs' ) {
                my %StateList = $Self->{StateObject}->StateList( UserID => 1 );
                ID:
                for my $ID ( @{$Values} ) {
                    next ID if !$ID;
                    $ID->{Content} = $StateList{ $ID->{Content} };
                }
            }
        }
    }
    return \%Param;
}

sub ImportWrapper {
    my ( $Self, %Param ) = @_;

    # wrap used spelling to ids
```

```
    for my $Use (qw(UseAsValueSeries UseAsRestriction UseAsXvalue)) {
        ELEMENT:
        for my $Element ( @{ $Param{$Use} } ) {
            next ELEMENT if !$Element || !$Element->{SelectedValues};
            my $ElementName = $Element->{Element};
            my $Values      = $Element->{SelectedValues};

            if ( $ElementName eq 'QueueIDs' || $ElementName eq 'CreatedQueueIDs' ) {
                ID:
                for my $ID ( @{$Values} ) {
                    next ID if !$ID;
                    if ( $Self->{QueueObject}->QueueLookup( Queue => $ID->{Content} ) ) {
                        $ID->{Content}
                            = $Self->{QueueObject}->QueueLookup( Queue => $ID->{Content} );
                    }
                    else {
                        $Self->{LogObject}->Log(
                            Priority => 'error',
                            Message  => "Import: Can' find the queue $ID->{Content}!"
                        );
                        $ID = undef;
                    }
                }
            }
            elsif ( $ElementName eq 'StateIDs' || $ElementName eq 'CreatedStateIDs' ) {
                ID:
                for my $ID ( @{$Values} ) {
                    next ID if !$ID;

                    my %State = $Self->{StateObject}->StateGet(
                        Name  => $ID->{Content},
                        Cache => 1,
                    );
                    if ( $State{ID} ) {
                        $ID->{Content} = $State{ID};
                    }
                    else {
                        $Self->{LogObject}->Log(
                            Priority => 'error',
                            Message  => "Import: Can' find state $ID->{Content}!"
                        );
                        $ID = undef;
                    }
                }
            }
        }
    }
    return \%Param;
}

1;
```

2.3.3.1.2. Configuration example

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<otrs_config version="1.0" init="Config">
    <ConfigItem Name="Stats::DynamicObjectRegistration###DynamicStatsTemplate" Required="0"
 Valid="1">
        <Description Lang="en">Here you can decide if the common stats module may generate
 stats about the number of default tickets a requester created.</Description>
        <Group>Framework</Group>
        <SubGroup>Core::Stats</SubGroup>
        <Setting>
            <Hash>
                <Item Key="Module">Kernel::System::Stats::Dynamic::DynamicStatsTemplate</
Item>
            </Hash>
        </Setting>
    </ConfigItem>
</otrs_config>
```

2.3.3.1.3. Use case examples

Use cases.

2.3.3.1.4. Caveats and Warnings

If you have a lot of cells in the result table and the GetStatElement is quite complex, the request can take a long time.

2.3.3.1.5. Release Availability

Dynamic stat modules are available since OTRS 2.0.

2.3.3.2. Static Stats

The subsequent paragraphs describe the static stats. Static stats are very easy to create as these modules have to implement only three subroutines.

- new
- Param
- Run

2.3.3.2.1. Code example

The following paragraphs describe the subroutines needed in a static stats.

```
sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {%Param};
    bless( $Self, $Type );

    # check all needed objects
    for my $Needed (
        qw(DBObject ConfigObject LogObject
        TimeObject MainObject EncodeObject)
        )
    {
        $Self->{$Needed} = $Param{$Needed} || die "Got no $Needed";
    }

    # create needed objects
    $Self->{TypeObject}   = Kernel::System::Type->new( %{$Self} );
    $Self->{TicketObject} = Kernel::System::Ticket->new( %{$Self} );
    $Self->{QueueObject}  = Kernel::System::Queue->new( %{$Self} );

    return $Self;
}
```

new creates a new instance of the static stats class. First it creates a new object and then it checks for the needed objects.

```
sub Param {
    my $Self = shift;

    my %Queues = $Self->{QueueObject}->GetAllQueues();
    my %Types  = $Self->{TypeObject}->TypeList(
        Valid => 1,
    );

    my @Params = (
        {
```

```
            Frontend  => 'Type',
            Name      => 'TypeIDs',
            Multiple  => 1,
            Size      => 3,
            Data      => \%Types,
        },
        {
            Frontend  => 'Queue',
            Name      => 'QueueIDs',
            Multiple  => 1,
            Size      => 3,
            Data      => \%Queues,
        },
    );

    return @Params;
}
```

The Param method provides the list of all parameters/attributes that can be selected to create a static stat. It gets some parameters passed: The values for the stats attributes provided in a request, the format of the stats and the name of the object (name of the module).

The parameters/attributes have to be hashreferences with these key-value-pairs.

- Frontent

  the label in the web interface

- Name

  the HTML fieldname

- Data

  the values shown in the attribute

Other parameter for the BuildSelection method of the LayoutObject can be used, as it is done with "Size" and "Multiple" in this sample module.

```
sub Run {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for my $Needed (qw(TypeIDs QueueIDs)) {
        if ( !$Param{$Needed} ) {
            $Self->{LogObject}->Log(
                Priority => 'error',
                Message  => "Need $Needed!",
            );
            return;
        }
    }

    # set report title
    my $Title = 'Tickets per Queue';

    # table headlines
    my @HeadData = (
        'Ticket Number',
        'Queue',
        'Type',
    );

    my @Data;
    my @TicketIDs = $Self->{TicketObject}->TicketSearch(
        UserID     => 1,
        Result     => 'ARRAY',
```

```
        Permission => 'ro',
        %Param,
    );

    for my $TicketID ( @TicketIDs ) {
        my %Ticket = $Self->{TicketObject}->TicketGet(
            UserID => 1,
            TicketID => $TicketID,
        );
        push @Data, [ $Ticket{TicketNumber}, $Ticket{Queue}, $Ticket{Type} ];
    }

    return ( [$Title], [@HeadData], @Data );
}
```

The Run method actually generates the table data for the stats. It gets the attributes for this stats passed. In this sample it in %Param a key 'TypeIDs' and a key 'QueueIDs' exist (see attributes in Param method) and their values are arrayreferences. The returned data consists of three parts: Two arrayreferences and an array. In the first arrayreference the title for the statistic is stored, the second arrayreference contains the headlines for the columns in the table. And then the data for the table body follow.

```
# --
# Kernel/System/Stats/Static/StaticStatsTemplate.pm
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::Stats::Static::StaticStatsTemplate;

use strict;
use warnings;

use Kernel::System::Type;
use Kernel::System::Ticket;
use Kernel::System::Queue;

=head1 NAME

StaticStatsTemplate.pm - the module that creates the stats about tickets in a queue

=head1 SYNOPSIS

All functions

=head1 PUBLIC INTERFACE

=over 4

=cut

=item new()

create an object

    use Kernel::Config;
    use Kernel::System::Encode;
    use Kernel::System::Log;
    use Kernel::System::Main;
    use Kernel::System::Time;
    use Kernel::System::DB;
    use Kernel::System::Stats::Static::StaticStatsTemplate;

    my $ConfigObject = Kernel::Config->new();
    my $EncodeObject = Kernel::System::Encode->new(
        ConfigObject => $ConfigObject,
```

```
    );
    my $LogObject    = Kernel::System::Log->new(
        ConfigObject => $ConfigObject,
    );
    my $MainObject = Kernel::System::Main->new(
        ConfigObject => $ConfigObject,
        LogObject    => $LogObject,
    );
    my $TimeObject = Kernel::System::Time->new(
        ConfigObject => $ConfigObject,
        LogObject    => $LogObject,
    );
    my $DBObject = Kernel::System::DB->new(
        ConfigObject => $ConfigObject,
        LogObject    => $LogObject,
        MainObject   => $MainObject,
    );
    my $StatsObject = Kernel::System::Stats::Static::StaticStatsTemplate->new(
        ConfigObject => $ConfigObject,
        LogObject    => $LogObject,
        MainObject   => $MainObject,
        TimeObject   => $TimeObject,
        DBObject     => $DBObject,
        EncodeObject => $EncodeObject,
    );

=cut

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {%Param};
    bless( $Self, $Type );

    # check all needed objects
    for my $Needed (
        qw(DBObject ConfigObject LogObject
        TimeObject MainObject EncodeObject)
        )
    {
        $Self->{$Needed} = $Param{$Needed} || die "Got no $Needed";
    }

    # create needed objects
    $Self->{TypeObject}   = Kernel::System::Type->new( %{$Self} );
    $Self->{TicketObject} = Kernel::System::Ticket->new( %{$Self} );
    $Self->{QueueObject}  = Kernel::System::Queue->new( %{$Self} );

    return $Self;
}

=item Param()

Get all parameters a user can specify.

    my @Params = $StatsObject->Param();

=cut

sub Param {
    my $Self = shift;

    my %Queues = $Self->{QueueObject}->GetAllQueues();
    my %Types  = $Self->{TypeObject}->TypeList(
        Valid => 1,
    );

    my @Params = (
        {
            Frontend  => 'Type',
            Name      => 'TypeIDs',
```

```
            Multiple  => 1,
            Size      => 3,
            Data      => \%Types,
        },
        {
            Frontend  => 'Queue',
            Name      => 'QueueIDs',
            Multiple  => 1,
            Size      => 3,
            Data      => \%Queues,
        },
    );

    return @Params;
}

=item Run()

generate the statistic.

    my $StatsInfo = $StatsObject->Run(
        TypeIDs  => [
            1, 2, 4
        ],
        QueueIDs => [
            3, 4, 6
        ],
    );

=cut

sub Run {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for my $Needed (qw(TypeIDs QueueIDs)) {
        if ( !$Param{$Needed} ) {
            $Self->{LogObject}->Log(
                Priority => 'error',
                Message  => "Need $Needed!",
            );
            return;
        }
    }

    # set report title
    my $Title = 'Tickets per Queue';

    # table headlines
    my @HeadData = (
        'Ticket Number',
        'Queue',
        'Type',
    );

    my @Data;
    my @TicketIDs = $Self->{TicketObject}->TicketSearch(
        UserID     => 1,
        Result     => 'ARRAY',
        Permission => 'ro',
        %Param,
    );

    for my $TicketID ( @TicketIDs ) {
        my %Ticket = $Self->{TicketObject}->TicketGet(
            UserID => 1,
            TicketID => $TicketID,
        );
        push @Data, [ $Ticket{TicketNumber}, $Ticket{Queue}, $Ticket{Type} ];
    }

    return ( [$Title], [@HeadData], @Data );
```

```
}

1;

=back

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (http://otrs.org/).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see http://www.gnu.org/licenses/agpl.txt.

=cut
```

2.3.3.2.2. Configuration example

There is no configuration needed. Right after installation, the module is available to create a statistic for this module.

2.3.3.2.3. Use case examples

Use cases.

2.3.3.2.4. Caveats and Warnings

Caveats and Warnings for static stats.

2.3.3.2.5. Release Availability

Static stat modules are available since OTRS 1.3.

2.3.3.2.6. Using old static stats

Standard OTRS versions 1.3 and 2.0 already facilitated the generation of stats. Various stats for OTRS versions 1.3 and 2.0 which have been specially developed to meet customers' requirements can be used in more recent versions too.

The files must merely be moved from the `Kernel/System/Stats/` path to `Kernel/System/Stats/Static/`. Additionally the package name of the respective script must be amended by "::Static".

The following example shows how the first path is amended.

```
   package Kernel::System::Stats::AccountedTime;
```

```
   package Kernel::System::Stats::Static::AccountedTime;
```

2.3.4. Ticket Number Generator Modules

Ticket number generators are used to create distinct identifiers aka TicketNumber for new tickets. Any method of creating a string of numbers is possible, you should use common sense about the length of the resulting string (guideline: 5-10). When creating a ticket number, make sure the result is prefixed by the SysConfig-Variable SystemID in order to enable the detection of ticket numbers on inbound email responses. A ticket number generator module needs the two functions TicketCreateNumber() and GetTNByString(). The method TicketCreateNumber() is called without parameters and returns the new ticket

number. The method GetTNByString() is called with the param String which contains the string to be parsed for a ticket number and returns the ticket number if found.

2.3.4.1. Code example

See Kernel/System/Ticket/Number/UserRandom.pm in the package TemplateModule.

2.3.4.2. Configuration example

See Kernel/Config/Files/TicketNumberGenerator.xml in the package TemplateModule.

2.3.4.3. Use Cases

2.3.4.3.1. Ticket numbers should follow a specific scheme.

You will need to create a new ticket number generator if the default modules don't provide the ticket number scheme you'd like to use.

2.3.4.4. Caveats and Warnings

You should stick to the code of GetTNByString() as used in existing ticket number generators to prevent problems with ticket number parsing. Also the routine to detect a loop in TicketCreateNumber() should be kept intact to prevent duplicate ticket numbers.

2.3.4.5. Release Availability

Ticket number generators have been available in OTRS since OTRS 1.1.

2.3.5. Ticket Event Module

Ticket event modules are running right after a ticket action takes place. Per convention these modules are located in the directory "Kernel/System/Ticket/Event". An ticket event module needs only the two functions new() and Run(). The method Run() receives at least the parameters Event, UserID, and Data. Data is a hash ref containing data of the ticket, and in case of Article-related events also containing Article data.

2.3.5.1. Code example

See Kernel/System/Ticket/Event/EventModulePostTemplate.pm in the package TemplateModule.

2.3.5.2. Configuration example

See Kernel/Config/Files/EventModulePostTemplate.xml in the package TemplateModule.

2.3.5.3. Use Cases

2.3.5.3.1. A ticket should be unlocked after a move action.

This standard feature has been implemented with the ticket event module Kernel::System::Ticket::Event::ForceUnlock. When this feature is not wanted, then it can be turned off by unsetting the SysConfig entry Ticket::EventModulePost###910-ForceUnlockOnMove.

2.3.5.3.2. Perform extra cleanup action when a ticket is deleted.

A customized OTRS might hold non-standard data in additional database tables. When a ticket is deleted then this additional data needs to be deleted. This functionality can be achieved with a ticket event module listening to 'TicketDelete' events.

2.3.5.3.3. New tickets should be twittered.

A ticket event module listening to 'TicketCreate' can send out tweets.

2.3.5.4. Caveats and Warnings

No caveats are known.

2.3.5.5. Release Availability

Ticket events have been available in OTRS since OTRS 2.0.

Ticket Events for OTRS 2.0: TicketCreate, TicketDelete, TicketTitleUpdate, TicketUnlockTimeoutUpdate, TicketEscalationStartUpdate, MoveTicket, SetCustomerData, TicketFreeTextSet, TicketFreeTimeSet, TicketPendingTimeSet, LockSet, StateSet, OwnerSet, TicketResponsibleUpdate, PrioritySet, HistoryAdd, HistoryDelete, TicketAccountTime, TicketMerge, ArticleCreate, ArticleFreeTextSet, ArticleUpdate, ArticleSend, ArticleBounce, SendAgentNotification, SendCustomerNotification, SendAutoResponse, ArticleFlagSet;

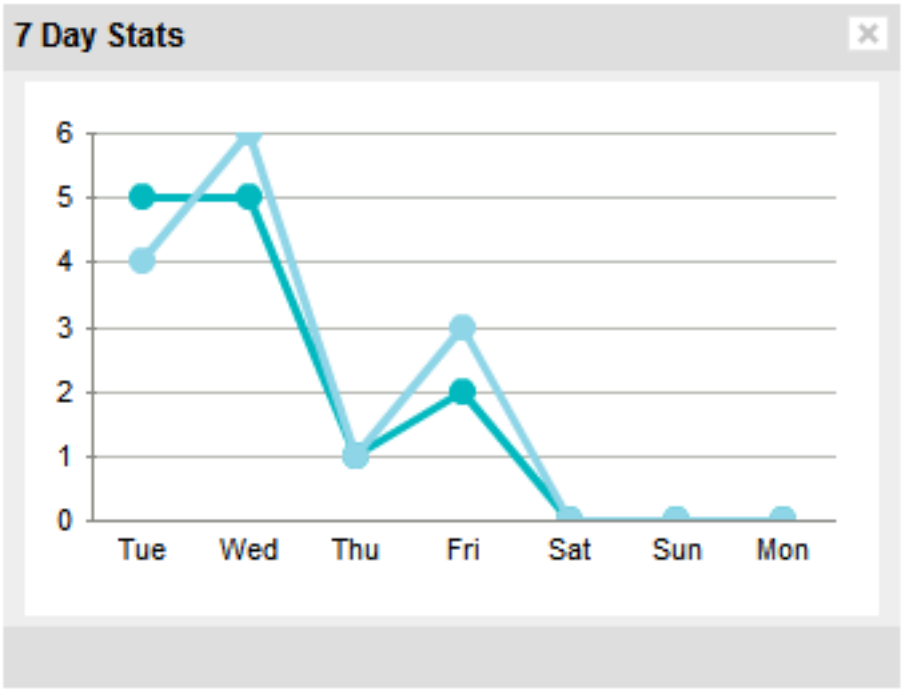
Ticket Events for OTRS 2.1 and higher: TicketCreate, TicketDelete, TicketTitleUpdate, TicketUnlockTimeoutUpdate, TicketEscalationStartUpdate, TicketQueueUpdate (MoveTicket), TicketCustomerUpdate (SetCustomerData), TicketFreeTextUpdate (TicketFreeTextSet), TicketFreeTimeUpdate (TicketFreeTimeSet), TicketPendingTimeUpdate (TicketPendingTimeSet), TicketLockUpdate (LockSet), TicketStateUpdate (StateSet), TicketOwnerUpdate (OwnerSet), TicketResponsibleUpdate, TicketPriorityUpdate (PrioritySet), TicketSubscribe, TicketUnsubscribe, HistoryAdd, HistoryDelete, TicketAccountTime, TicketMerge, ArticleCreate, ArticleFreeTextUpdate (ArticleFreeTextSet), ArticleUpdate, ArticleSend, ArticleBounce, ArticleAgentNotification (SendAgentNotification), ArticleCustomerNotification (SendCustomerNotification), ArticleAutoResponse (SendAutoResponse), ArticleFlagSet, ArticleFlagDelete;

Ticket Events for OTRS 2.4: TicketCreate, TicketDelete, TicketTitleUpdate, TicketUnlockTimeoutUpdate, TicketQueueUpdate(MoveTicket), TicketTypeUpdate, TicketServiceUpdate, TicketSLAUpdate, TicketCustomerUpdate (SetCustomerData), TicketFreeTextUpdate, TicketFreeTimeUpdate, TicketPendingTimeUpdate (TicketPendingTimeSet), TicketLockUpdate (LockSet), TicketStateUpdate (StateSet), TicketOwnerUpdate (OwnerSet), TicketResponsibleUpdate, TicketPriorityUpdate (PrioritySet), HistoryAdd, HistoryDelete, TicketAccountTime, TicketMerge, ArticleCreate, ArticleFreeTextUpdate (ArticleFreeTextSet), ArticleUpdate, ArticleSend, ArticleBounce, ArticleAgentNotification (SendAgentNotification), ArticleCustomerNotification (SendCustomerNotification), ArticleAutoResponse(SendAutoResponse), ArticleFlagSet, ArticleFlagDelete;

2.4. Frontend Modules

2.4.1. Dashboard Module

Dashboard module to display statistics in the form of a line graph.




```
# --
# Kernel/Output/HTML/DashboardTicketStatsGeneric.pm - message of the day
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Output::HTML::DashboardTicketStatsGeneric;

use strict;
use warnings;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {%Param};
    bless( $Self, $Type );

    # get needed objects
    for (
        qw(Config Name ConfigObject LogObject DBObject LayoutObject ParamObject TicketObject
 UserID)
        )
    {
        die "Got no $_!" if !$Self->{$_};
    }

    return $Self;
}

sub Preferences {
    my ( $Self, %Param ) = @_;

    return;
}

sub Config {
    my ( $Self, %Param ) = @_;
```

```
    my $Key = $Self->{LayoutObject}->{UserLanguage} . '-' . $Self->{Name};
    return (
        %{ $Self->{Config} },
        CacheKey => 'TicketStats' . '-' . $Self->{UserID} . '-' . $Key,
    );

}

sub Run {
    my ( $Self, %Param ) = @_;

    my %Axis = (
        '7Day' => {
            0 => { Day => 'Sun', Created => 0, Closed => 0, },
            1 => { Day => 'Mon', Created => 0, Closed => 0, },
            2 => { Day => 'Tue', Created => 0, Closed => 0, },
            3 => { Day => 'Wed', Created => 0, Closed => 0, },
            4 => { Day => 'Thu', Created => 0, Closed => 0, },
            5 => { Day => 'Fri', Created => 0, Closed => 0, },
            6 => { Day => 'Sat', Created => 0, Closed => 0, },
        },
    );

    my @Data;
    my $Max = 1;
    for my $Key ( 0 .. 6 ) {

        my $TimeNow = $Self->{TimeObject}->SystemTime();
        if ($Key) {
            $TimeNow = $TimeNow - ( 60 * 60 * 24 * $Key );
        }
        my ( $Sec, $Min, $Hour, $Day, $Month, $Year, $WeekDay )
            = $Self->{TimeObject}->SystemTime2Date(
            SystemTime => $TimeNow,
            );

        $Data[$Key]->{Day} = $Self->{LayoutObject}->{LanguageObject}->Get(
            $Axis{'7Day'}->{$WeekDay}->{Day}
        );

        my $CountCreated = $Self->{TicketObject}->TicketSearch(

            # cache search result 20 min
            CacheTTL => 60 * 20,

            # tickets with create time after ... (ticket newer than this date) (optional)
            TicketCreateTimeNewerDate => "$Year-$Month-$Day 00:00:00",

            # tickets with created time before ... (ticket older than this date) (optional)
            TicketCreateTimeOlderDate => "$Year-$Month-$Day 23:59:59",

            CustomerID => $Param{Data}->{UserCustomerID},
            Result     => 'COUNT',

            # search with user permissions
            Permission => $Self->{Config}->{Permission} || 'ro',
            UserID => $Self->{UserID},
        );
        $Data[$Key]->{Created} = $CountCreated;
        if ( $CountCreated > $Max ) {
            $Max = $CountCreated;
        }

        my $CountClosed = $Self->{TicketObject}->TicketSearch(

            # cache search result 20 min
            CacheTTL => 60 * 20,

            # tickets with create time after ... (ticket newer than this date) (optional)
            TicketCloseTimeNewerDate => "$Year-$Month-$Day 00:00:00",

            # tickets with created time before ... (ticket older than this date) (optional)
```

```
            TicketCloseTimeOlderDate => "$Year-$Month-$Day 23:59:59",

            CustomerID => $Param{Data}->{UserCustomerID},
            Result     => 'COUNT',

            # search with user permissions
            Permission => $Self->{Config}->{Permission} || 'ro',
            UserID => $Self->{UserID},
        );
        $Data[$Key]->{Closed} = $CountClosed;
        if ( $CountClosed > $Max ) {
            $Max = $CountClosed;
        }
    }

    @Data = reverse @Data;
    my $Source = $Self->{LayoutObject}->JSONEncode(
        Data => \@Data,
    );

    my $Content = $Self->{LayoutObject}->Output(
        TemplateFile => 'AgentDashboardTicketStats',
        Data         => {
            %{ $Self->{Config} },
            Key    => int rand 99999,
            Max    => $Max,
            Source => $Source,
        },
    );

    return $Content;
}

1;
```

To use this module add the following to the `Kernel/Config.pm` and restart your webserver (if you use mod_perl).

```
<ConfigItem Name="DashboardBackend###0250-TicketStats" Required="0" Valid="1">
    <Description Lang="en">Parameters for the dashboard backend. "Group" are used to
 restricted access to the plugin (e. g. Group: admin;group1;group2;). "Default" means if the
 plugin is enabled per default or if the user needs to enable it manually. "CacheTTL" means
 the cache time in minutes for the plugin.</Description>
    <Description Lang="de">Parameter für das Dashboard Backend. "Group" ist verwendet um
 den Zugriff auf das Plugin einzuschränken (z. B. Group: admin;group1;group2;). ""Default"
 bedeutet ob das Plugin per default aktiviert ist oder ob dies der Anwender manuell machen
 muss. "CacheTTL" ist die Cache-Zeit in Minuten nach der das Plugin erneut aufgerufen
 wird.</Description>
    <Group>Ticket</Group>
    <SubGroup>Frontend::Agent::Dashboard</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::Output::HTML::DashboardTicketStatsGeneric</Item>
            <Item Key="Title">7 Day Stats</Item>
            <Item Key="Created">1</Item>
            <Item Key="Closed">1</Item>
            <Item Key="Permission">rw</Item>
            <Item Key="Block">ContentSmall</Item>
            <Item Key="Group"></Item>
            <Item Key="Default">1</Item>
            <Item Key="CacheTTL">45</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.4.1.1. Caveats and Warnings

An excessive number of days or individual lines may lead to performance degradation.

2.4.1.2. Release Availability

from 2.4.0

2.4.2. Notification Module

Notification modules are used to display a notification below the main navigation. You can write and register your own notification module. There are currently 5 ticket menues in the OTRS framework.

- AgentOnline
- AgentTicketEscalation
- CharsetCheck
- CustomerOnline
- UIDCheck

2.4.2.1. Code Example

The notification modules are located under Kernel/Output/HTML/TicketNotification*.pm. There is an example of a notify module below. Save it under Kernel/Output/HTML/Ticket-NotificationCustom.pm. You just need 2 functions: new() and Run().

```
# --
# Kernel/Output/HTML/NotificationCustom.pm
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Output::HTML::NotificationCustom;

use strict;
use warnings;

use Kernel::System::Custom;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # get needed objects
    for my $Object (qw(ConfigObject LogObject DBObject LayoutObject TimeObject UserID)) {
        $Self->{$Object} = $Param{$Object} || die "Got no $Object!";
    }
    $Self->{CustomObject} = Kernel::System::Custom->new(%Param);
    return $Self;
}

sub Run {
    my ( $Self, %Param ) = @_;

    # get session info
    my %CustomParam      = ();
    my @Customs    = $Self->{CustomObject}->GetAllCustomIDs();
    my $IdleMinutes = $Param{Config}->{IdleMinutes} || 60 * 2;
    for (@Customs) {
        my %Data = $Self->{CustomObject}->GetCustomIDData( CustomID => $_, );
        if (
```

```
            $Self->{UserID} ne $Data{UserID}
            && $Data{UserType} eq 'User'
            && $Data{UserLastRequest}
            && $Data{UserLastRequest} + ( $IdleMinutes * 60 ) > $Self->{TimeObject}-
>SystemTime()
            && $Data{UserFirstname}
            && $Data{UserLastname}
            )
        {
            $CustomParam{ $Data{UserID} } = "$Data{UserFirstname} $Data{UserLastname}";
            if ( $Param{Config}->{ShowEmail} ) {
                $CustomParam{ $Data{UserID} } .= " ($Data{UserEmail})";
            }
        }
    }
    for ( sort { $CustomParam{$a} cmp $CustomParam{$b} } keys %CustomParam ) {
        if ( $Param{Message} ) {
            $Param{Message} .= ', ';
        }
        $Param{Message} .= "$CustomParam{$_}";
    }
    if ( $Param{Message} ) {
        return $Self->{LayoutObject}->Notify( Info => 'Custom Message: %s", "' .
 $Param{Message} );
    }
    else {
        return '';
    }
}

1;
```

2.4.2.2. Configuration Example

There is the need to activate your custom notification module. This can be done using the xml configuration below. There may be additional parameters in the config hash for your notification module.

```
<ConfigItem Name="Frontend::NotifyModule###3-Custom" Required="0" Valid="0">
    <Description Lang="en">Module to show custom message in the agent interface.</
Description>
    <Description Lang="de">Mit diesem Modul können eigene Meldungenen innerhalb des Agent-
Interfaces angezeigt werden.</Description>
    <Group>Framework</Group>
    <SubGroup>Frontend::Agent::ModuleNotify</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::Output::HTML::NotificationCustom</Item>
            <Item Key="Key1">1</Item>
            <Item Key="Key2">2</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.4.2.3. Use Case Example

Useful ticket menu implementation could be a link to a external tool if parameters (e.g. FreeTextField) have been set.

2.4.2.4. Release Availability

Name	Release
NotificationAgentOnline	2.0
NotificationAgentTicketEscalation	2.0

Name	Release
NotificationCharsetCheck	1.2
NotificationCustomerOnline	2.0
NotificationUIDCheck	1.2

2.4.3. Ticket Menu Module

Ticket menu modules are used to display an addtional link in the menu above a ticket. You can write and register your own ticket menu module. There are 4 ticket menues (Generic, Lock, Responsible and TicketWatcher) which come with the OTRS framework. For more information please have a look at the OTRS admin manual.

2.4.3.1. Code Example

The ticket menu modules are located under Kernel/Output/HTML/TicketMenu*.pm. There is an example of a ticket-menu module below. Save it under Kernel/Output/HTML/Ticket-MenuCustom.pm. You just need 2 functions: new() and Run().

```
# --
# Kernel/Output/HTML/TicketMenuCustom.pm
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# Id: TicketMenuCustom.pm,v 1.17 2010/04/12 21:34:06 martin Exp $
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Output::HTML::TicketMenuCustom;

use strict;
use warnings;

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # get needed objects
    for my $Object (qw(ConfigObject LogObject DBObject LayoutObject UserID TicketObject)) {
        $Self->{$Object} = $Param{$Object} || die "Got no $Object!";
    }

    return $Self;
}

sub Run {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    if ( !$Param{Ticket} ) {
        $Self->{LogObject}->Log(
            Priority => 'error',
            Message  => 'Need Ticket!'
        );
        return;
    }

    # check if frontend module registered, if not, do not show action
    if ( $Param{Config}->{Action} ) {
        my $Module = $Self->{ConfigObject}->Get('Frontend::Module')->{ $Param{Config}-
>{Action} };
        return if !$Module;
```

```
    }

    # check permission
    my $AccessOk = $Self->{TicketObject}->Permission(
        Type     => 'rw',
        TicketID => $Param{Ticket}->{TicketID},
        UserID   => $Self->{UserID},
        LogNo    => 1,
    );
    return if !$AccessOk;

    # check permission
    if ( $Self->{TicketObject}->CustomIsTicketCustom( TicketID => $Param{Ticket}-
>{TicketID} ) ) {
        my $AccessOk = $Self->{TicketObject}->OwnerCheck(
            TicketID => $Param{Ticket}->{TicketID},
            OwnerID  => $Self->{UserID},
        );
        return if !$AccessOk;
    }

    # check acl
    return
        if defined $Param{ACL}->{ $Param{Config}->{Action} }
            && !$Param{ACL}->{ $Param{Config}->{Action} };

    # if ticket is customized
    if ( $Param{Ticket}->{Custom} eq 'lock' ) {

        # if it is locked for somebody else
        return if $Param{Ticket}->{OwnerID} ne $Self->{UserID};

        # show custom action
        return {
            %{ $Param{Config} },
            %{ $Param{Ticket} },
            %Param,
            Name        => 'Custom',
            Description => 'Custom to give it back to the queue!',
            Link        => 'Action=AgentTicketCustom;Subaction=Custom;TicketID=
$QData{"TicketID"}',
        };
    }

    # if ticket is customized
    return {
        %{ $Param{Config} },
        %{ $Param{Ticket} },
        %Param,
        Name        => 'Custom',
        Description => 'Custom it to work on it!',
        Link        => 'Action=AgentTicketCustom;Subaction=Custom;TicketID=
$QData{"TicketID"}',
    };
}

1;
```

2.4.3.2. Configuration Example

There is the need to activate your custom ticket menu module. This can be done using the xml configuration below. There may be additional parameters in the config hash for your ticket menu module.

```
<ConfigItem Name="Ticket::Frontend::MenuModule###110-Custom" Required="0" Valid="1">
    <Description Lang="en">Module to show custom link in menu.</Description>
    <Description Lang="de">Mit diesem Modul wird der Custom-Link in der Linkleiste der
 Ticketansicht angezeigt.</Description>
```

```
    <Group>Ticket</Group>
    <SubGroup>Frontend::Agent::Ticket::MenuModule</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::Output::HTML::TicketMenuCustom</Item>
            <Item Key="Name">Custom</Item>
            <Item Key="Action">AgentTicketCustom</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.4.3.3. Use Case Example

Useful ticket menu implementation could be a link to a external tool if parameters (e.g. FreeTextField) have been set.

2.4.3.4. Caveats and Warnings

The ticket menu directs to an URL that can be handled. If you want to handle that request via the OTRS framework, you have to write your own frontend module.

2.4.3.5. Release Availability

Name	Release
TicketMenuGeneric	2.0
TicketMenuLock	2.0
TicketMenuResponsible	2.1
TicketMenuTicketWatcher	2.4

2.5. Generic Interface Modules

2.5.1. Network Transport

The network transport is used as method to send and receive information between OTRS and a Remote System. The Generic Interface configuration allows a web service to use different network transport modules for provider and requester, but the most common scenario is that the same transport module is used for both.

OTRS as provider:

OTRS uses the network transport modules to get the data from the Remote System and the operation to be executed. After the operation is performed OTRS uses them again to send the response back to the Remote System.

OTRS as requester:

OTRS uses the network transport modules to send petitions to the Remote System to perform a remote action along with the required data, OTRS waits for the Remote System response and send it back to the Requester module.

In both ways network transport modules deals with the data in the Remote System format. it is not recommended to do any data transformation in this modules, the Mapping layer is the responsible to perform any data transformation needed during the communication.

2.5.1.1. Transport backend

Next will see how to develop a new transport backend, each transport backend has to implement these subroutines:

- new
- ProviderProcessRequest
- ProviderGenerateResponse
- RequesterPerformRequest

We should implement each one of this methods in order to be able to communicate correctly with a Remote System in both ways. All network transport backends are handled by the transport module (Kernel/GenericInterface/Transport.pm).

Currently Generic Interface implements the HTTP SOAP transport, if the planned web service can use HTTP SOAP there is no need to create a new network transport module, instead we recommend to take a look into HTTP SOAP configuration to check the settings and how it can be tuned.

2.5.1.1.1. Code example

In case that the provided network transports does not match the web service needs, then in this section a sample network transport module is shown and each subroutine is explained. Normally transport modules uses CPAN modules as backends for example the HTTP SOAP transport modules uses SOAP::Lite module as backend.

For this example a custom package is used to return the data without doing a real network request to a Remote System, instead this custom module acts as a loopback interface.

```
# --
# Kernel/GenericInterface/Transport/HTTP/Test.pm - GenericInterface network transport
 interface for testing
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::GenericInterface::Transport::HTTP::Test;

use strict;
use warnings;

use LWP::UserAgent;
use LWP::Protocol;
use HTTP::Request::Common;

use Kernel::System::Web::Request;

use vars qw(@ISA $VERSION);
```

This is common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

```
sub new {
    my ( $Type, %Param ) = @_;

    my $Self = {};
    bless( $Self, $Type );

    for my $Needed (
        qw(LogObject EncodeObject ConfigObject MainObject DebuggerObject TransportConfig)
        )
```

```
    {
        $Self->{$Needed} = $Param{$Needed} || return {
            Success      => 0,
            ErrorMessage => "Got no $Needed!"
        };
    }

    return $Self;
}
```

The constructor 'new' creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module have to be created in "new".

```
sub ProviderProcessRequest {
    my ( $Self, %Param ) = @_;

    if ( $Self->{TransportConfig}->{Config}->{Fail} ) {
        return {
            Success      => 0,
            ErrorMessage => "HTTP status code: 500",
            Data         => {},
        };
    }

    my $ParamObject = Kernel::System::Web::Request->new( %{$Self} );

    my %Result;

    for my $ParamName ( $ParamObject->GetParamNames() ) {
        $Result{$ParamName} = $ParamObject->GetParam( Param => $ParamName );
    }

    # special handling for empty post request
    if ( keys %Result == 1 && exists $Result{POSTDATA} && !$Result{POSTDATA} ) {
        %Result = ();
    }

    if ( !%Result ) {
        return $Self->{DebuggerObject}->Error(
            Summary => 'No request data found.',
        );
    }

    return {
        Success   => 1,
        Data      => \%Result,
        Operation => 'test_operation',
    };

}
```

The 'ProviderProcessRequest' function gets the request form the Remote System (in this case the same OTRS) and extracts the data and the operation to perform from the request. For this example the operation is always 'test_operation'.

The way this function parse the request to get the data and the operation name, depends completely on the protocol to be implemented and the external modules that are used for.

```
sub ProviderGenerateResponse {
    my ( $Self, %Param ) = @_;

    if ( $Self->{TransportConfig}->{Config}->{Fail} ) {
        return {
            Success      => 0,
            ErrorMessage => 'Test response generation failed',
        };
```

```
    }

    my $Response;

    if ( !$Param{Success} ) {
        $Response
            = HTTP::Response->new( 500 => ( $Param{ErrorMessage} || 'Internal Server
 Error' ) );
        $Response->protocol('HTTP/1.0');
        $Response->content_type("text/plain; charset=UTF-8");
        $Response->date(time);
    }
    else {

        # generate a request string from the data
        my $Request
            = HTTP::Request::Common::POST( 'http://testhost.local/', Content =>
 $Param{Data} );

        $Response = HTTP::Response->new( 200 => "OK" );
        $Response->protocol('HTTP/1.0');
        $Response->content_type("text/plain; charset=UTF-8");
        $Response->add_content_utf8( $Request->content );
        $Response->date(time);
    }

    $Self->{DebuggerObject}->Debug(
        Summary => 'Sending HTTP response',
        Data    => $Response->as_string(),
    );

    # now send response to client
    print STDOUT $Response->as_string();

    return {
        Success => 1,
    };
}
```

This function send the response back to the Remote System for the requested operation.

For this particular example we return an standard HTTP response success (200) or not (500), along with the required data on each case.

```
sub RequesterPerformRequest {
    my ( $Self, %Param ) = @_;

    if ( $Self->{TransportConfig}->{Config}->{Fail} ) {
        return {
            Success      => 0,
            ErrorMessage => "HTTP status code: 500",
            Data         => {},
        };
    }

    # use custom protocol handler to avoid sending out real network requests
    LWP::Protocol::implementor(
        testhttp => 'Kernel::GenericInterface::Transport::HTTP::Test::CustomHTTPProtocol'
    );
    my $UserAgent = LWP::UserAgent->new();
    my $Response = $UserAgent->post( 'testhttp://localhost.local/', Content =>
 $Param{Data} );

    return {
        Success => 1,
        Data    => {
            ResponseContent => $Response->content,
        },
    };
}
```

This is the only function that is used by OTRS as requester. It sends the request to the Remote System and waits for it response.

For this example we use a custom protocol handler to avoid send the request to the real network. this custom protocol is specified below.

```
package Kernel::GenericInterface::Transport::HTTP::Test::CustomHTTPProtocol;

use base qw(LWP::Protocol);

sub new {
    my $Class = shift;
    return $Class->SUPER::new(@_);
}

sub request {
    my $Self = shift;
    my ( $Request, $Proxy, $Arg, $Size, $Timeout ) = @_;

    my $Response = HTTP::Response->new( 200 => "OK" );
    $Response->protocol('HTTP/1.0');
    $Response->content_type("text/plain; charset=UTF-8");
    $Response->add_content_utf8( $Request->content );
    $Response->date(time);

    return $Response;
}
```

This is the code for the custom protocol that we use. This approach is only useful for training or for testing environments where the Remote Systems are not available.

For a new module development we do not recommend to use this approach, a real protocol needs to be implemented.

```
1;

=end Internal:

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut

=cut
```

2.5.1.1.2. Configuration Example

There is the need to register this network transport module to be accessible in the OTRS GUI. This can be done using the xml configuration below.

```
<ConfigItem Name="GenericInterface::Transport::Module###HTTP::Test" Required="0" Valid="1">
    <Description Translatable="1">GenericInterface module registration for the transport
 layer.</Description>
    <Group>GenericInterface</Group>
    <SubGroup>GenericInterface::Transport::ModuleRegistration</SubGroup>
    <Setting>
```

```
        <Hash>
            <Item Key="Name">Test</Item>
            <Item Key="Protocol">HTTP</Item>
            <Item Key="ConfigDialog">AdminGenericInterfaceTransportHTTPTest</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.5.2. Mapping

The mapping is used to convert data from / to OTRS, to / from the external system. This data can be represented as key => value pairs, mapping modules can be developed to transform not just values but also the keys.

For example:

From	**To**
Prio => Warning	PriorityID => 3

The mapping layer is not absolutely necessary, a web service can skip it completely depending on the web service configuration and how invokers and operation are implemented. But if some data transformations are needed, is highly recommended to use an existing mapping module or create a new one.

Mapping modules can be called more than one time during a normal communication, take a look to the following examples.

OTRS as provider example:

1. The remote system sends the request with the data in the remote system format
2. The data is mapped from the remote system format to the OTRS format
3. OTRS performs the operation and return the response in OTRS format
4. The data is mapped form the OTRS format to the remote system format
5. The response with the data in the remote system format is sent to the remote system

OTRS as requester example:

1. OTRS prepare the request to the remote system using the data in the OTRS format
2. The data is mapped from the OTRS format to the remote system format
3. The request is sent to the remote system which perform the action and send the response back to OTRS with the response data in remote system format
4. The data is mapped form remote system format (again) to the OTRS format
5. OTRS process the response

2.5.2.1. Mapping backend

Generic Interface provides a mapping module called *Simple*. With this module most of the data transformations including key and value mapping can be done, and also it defines rules for to handling the default mappings for both keys and values.

So it is highly provable that you don't need to develop a custom mapping module. Please check Mapping Simple module (Kernel/GenericInterface/Mapping/Simple.pm) and its on-line documentation before continue.

If Mapping Simple module does not match your needs then we will see how to develop a new Mapping backend, each mapping backend has to implement these subroutines:

- new
- Map

We should implement each one of this methods in order to be able to map the data in the communication, handled either by the requester or provider. All mapping backends are handled by the mapping module (Kernel/GenericInterface/Mapping.pm).

2.5.2.1.1. Code example

In this section a sample mapping module is shown and each subroutine is explained.

```
# --
# Kernel/GenericInterface/Mapping/Test.pm - GenericInterface test data mapping backend
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::GenericInterface::Mapping::Test;

use strict;
use warnings;

use Kernel::System::VariableCheck qw(IsHashRefWithData IsStringWithData);

use vars qw(@ISA $VERSION);
```

This is common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

We also include VariableCheck module to perform certain validation over some variables.

```
sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed params
    for my $Needed (qw(DebuggerObject MainObject MappingConfig)) {
        if ( !$Param{$Needed} ) {
            return {
                Success      => 0,
                ErrorMessage => "Got no $Needed!"
            };
        }
        $Self->{$Needed} = $Param{$Needed};
    }

    # check mapping config
    if ( !IsHashRefWithData( $Param{MappingConfig} ) ) {
        return $Self->{DebuggerObject}->Error(
            Summary => 'Got no MappingConfig as hash ref with content!',
```

```
        );
    }

    # check config - if we have a map config, it has to be a non-empty hash ref
    if (
        defined $Param{MappingConfig}->{Config}
        && !IsHashRefWithData( $Param{MappingConfig}->{Config} )
        )
    {
        return $Self->{DebuggerObject}->Error(
            Summary => 'Got MappingConfig with Data, but Data is no hash ref with content!',
        );
    }

    return $Self;
}
```

The constructor 'new' creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module have to be created in "new".

```
sub Map {
    my ( $Self, %Param ) = @_;

    # check data - only accept undef or hash ref
    if ( defined $Param{Data} && ref $Param{Data} ne 'HASH' ) {
        return $Self->{DebuggerObject}
            ->Error( Summary => 'Got Data but it is not a hash ref in Mapping Test
 backend!' );
    }

    # return if data is empty
    if ( !defined $Param{Data} || !%{ $Param{Data} } ) {
        return {
            Success => 1,
            Data    => {},
        };
    }

    # no config means that we just return input data
    if (
        !defined $Self->{MappingConfig}->{Config}
        || !defined $Self->{MappingConfig}->{Config}->{TestOption}
        )
    {
        return {
            Success => 1,
            Data    => $Param{Data},
        };
    }

    # check TestOption format
    if ( !IsStringWithData( $Self->{MappingConfig}->{Config}->{TestOption} ) ) {
        return $Self->{DebuggerObject}->Error(
            Summary => 'Got no TestOption as string with value!',
        );
    }

    # parse data according to configuration
    my $ReturnData = {};
    if ( $Self->{MappingConfig}->{Config}->{TestOption} eq 'ToUpper' ) {
        $ReturnData = $Self->_ToUpper( Data => $Param{Data} );
    }
    elsif ( $Self->{MappingConfig}->{Config}->{TestOption} eq 'ToLower' ) {
        $ReturnData = $Self->_ToLower( Data => $Param{Data} );
    }
    elsif ( $Self->{MappingConfig}->{Config}->{TestOption} eq 'Empty' ) {
        $ReturnData = $Self->_Empty( Data => $Param{Data} );
    }
    else {
        $ReturnData = $Param{Data};
```

```
    }

    # return result
    return {
        Success => 1,
        Data    => $ReturnData,
    };
}
```

The 'map' function is the main part of each mapping module, it receives the mapping configuration (rules) and the data in the original format (either OTRS or remote system format) and converts it to a new format, even the strucure of the data can be changed during the mapping process.

In this particular example there are three rules to map the values this rules are set in the mapping configuration key "TestOption" and they are ToUpper, ToLower and Empty.

- ToUpper: converts each data value to upper case.
- ToLower: converts each data value to lower case.
- Empty: converts each data value into an empty string.

In this example no data key transformations were implemented.

```
sub _ToUpper {
    my ( $Self, %Param ) = @_;

    my $ReturnData = {};
    for my $Key ( keys %{ $Param{Data} } ) {
        $ReturnData->{$Key} = uc $Param{Data}->{$Key};
    }

    return $ReturnData;
}

sub _ToLower {
    my ( $Self, %Param ) = @_;

    my $ReturnData = {};
    for my $Key ( keys %{ $Param{Data} } ) {
        $ReturnData->{$Key} = lc $Param{Data}->{$Key};
    }

    return $ReturnData;
}

sub _Empty {
    my ( $Self, %Param ) = @_;

    my $ReturnData = {};
    for my $Key ( keys %{ $Param{Data} } ) {
        $ReturnData->{$Key} = '';
    }

    return $ReturnData;
}
```

This are the helper functions that actually performs the string conversions

```


=end Internal:

=back
```

```
=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut

=cut
```

2.5.2.1.2. Configuration Example

There is the need to register this mapping module to be accessible in the OTRS GUI. This can be done using the xml configuration below.

```
<ConfigItem Name="GenericInterface::Mapping::Module###Test" Required="0" Valid="1">
    <Description Translatable="1">GenericInterface module registration for the mapping
 layer.</Description>
    <Group>GenericInterface</Group>
    <SubGroup>GenericInterface::Mapping::ModuleRegistration</SubGroup>
    <Setting>
        <Hash>
            <Item Key="ConfigDialog"></Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.5.3. Invoker

The invoker is used to create a request from OTRS to a Remote System, this part of the GI is in charge of perform necessary tasks in OTRS side, to gather the necessary data in order to construct the request.

2.5.3.1. Invoker backend

Next we will see how to develop a new Invoker, each invoker has to implement these subroutines:

- new
- PrepareRequest
- HandleResponse

We should implement each one of this methods in order to be able to execute a request using the request handler ('Kernel/GenericInterface/Requester.pm').

2.5.3.1.1. Code example

In this section a sample invoker module is shown and each subroutine is explained.

```
# --
# Kernel/GenericInterface/Invoker/Test.pm - GenericInterface test data Invoker backend
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --
```

```
package Kernel::GenericInterface::Invoker::Test::Test;

use strict;
use warnings;

use Kernel::System::VariableCheck qw(IsString IsStringWithData);

use vars qw(@ISA $VERSION);
```

This is common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

```
sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # check needed params
    for my $Needed (qw(DebuggerObject MainObject TimeObject)) {
        if ( !$Param{$Needed} ) {
            return {
                Success      => 0,
                ErrorMessage => "Got no $Needed!"
            };
        }

        $Self->{$Needed} = $Param{$Needed};
    }

    return $Self;
}
```

The constructor 'new' creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module have to be created in "new".

```
sub PrepareRequest {
    my ( $Self, %Param ) = @_;

    # we need a TicketNumber
    if ( !IsStringWithData( $Param{Data}->{TicketNumber} ) ) {
        return $Self->{DebuggerObject}->Error( Summary => 'Got no TicketNumber' );
    }

    my %ReturnData;

    $ReturnData{TicketNumber} = $Param{Data}->{TicketNumber};

    # check Action
    if ( IsStringWithData( $Param{Data}->{Action} ) ) {
        $ReturnData{Action} = $Param{Data}->{Action} . 'Test';
    }

    # check request for system time
    if ( IsStringWithData( $Param{Data}->{GetSystemTime} ) && $Param{Data}-
>{GetSystemTime} ) {
        $ReturnData{SystemTime} = $Self->{TimeObject}->SystemTime();
    }

    return {
        Success => 1,
        Data    => \%ReturnData,
    };
}
```

The 'PrepareRequest' function is used to handle and collect all needed data to be sent into the request, here we can receive data from the request handler, use it, extend it, generate new data, and after that, we can transfer the results to the mapping layer.

For this example we are expecting to receive a ticket number, if there isn't then we use the debugger method "Error()" that creates an entry in the debug log and also returns a structure with the parameter Success as 0 and Error Message as the passed summary.

Also this example appends the word "Test" to the parameter "Action" and if GetSystem-Time is requested, it will fill the SystemTime parameter with the current system time. This part of the code is to prepare the data to be sent, on a real invoker some calls to core modules (Kernel/System/*.pm) should be made here.

If during any part of the 'PrepareRequest' function the request need to be stop without generating and error an entry in the debug log the following code can be used:

```
# stop requester communication
return {
    Success           => 1,
    StopCommunication => 1,
};
```

Using this, the Requester will understand that the request should not continue (it will not be sent to Mapping layer and will also not be sent to the Network Transport) Requester will not send an error on the debug log. it will only silently stop.

```
sub HandleResponse {
    my ( $Self, %Param ) = @_;

    # if there was an error in the response, forward it
    if ( !$Param{ResponseSuccess} ) {
        if ( !IsStringWithData( $Param{ResponseErrorMessage} ) ) {
            return $Self->{DebuggerObject}->Error(
                Summary => 'Got response error, but no response error message!',
            );
        }
        return {
            Success      => 0,
            ErrorMessage => $Param{ResponseErrorMessage},
        };
    }

    # we need a TicketNumber
    if ( !IsStringWithData( $Param{Data}->{TicketNumber} ) ) {
        return $Self->{DebuggerObject}->Error( Summary => 'Got no TicketNumber!' );
    }

    # prepare TicketNumber
    my %ReturnData = (
        TicketNumber => $Param{Data}->{TicketNumber},
    );

    # check Action
    if ( IsStringWithData( $Param{Data}->{Action} ) ) {
        if ( $Param{Data}->{Action} !~ m{ \A ( .*? ) Test \z }xms ) {
            return $Self->{DebuggerObject}->Error(
                Summary => 'Got Action but it is not in required format!',
            );
        }
        $ReturnData{Action} = $1;
    }

    return {
        Success => 1,
        Data    => \%ReturnData,
```

```
    };
}
```

The 'HandleResponse' function is used to receive and process the data from the previous request, that was made to the Remote System this data already passed by Mapping layer, to transform it from Remote System format to OTRS format (if needed).

For this particular example it checks the ticket number again and check if the action ends with the word 'Test' (as was done in the 'PrepareRequest' function).

Remember This invoker is only used for tests, a real invoker will check if the response was on the format described by the Remote System and can perform some actions like: call another invoker, perform an call to a Core Module, update the database, send an error, etc.

```
1;

=back

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut

=cut
```

2.5.3.1.2. Configuration Example

There is the need to register this invoker module to be accessible in the OTRS GUI. This can be done using the xml configuration below.

```
<ConfigItem Name="GenericInterface::Invoker::Module###Test::Test" Required="0" Valid="1">
    <Description Translatable="1">GenericInterface module registration for the invoker
 layer.</Description>
    <Group>GenericInterface</Group>
    <SubGroup>GenericInterface::Invoker::ModuleRegistration</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Name">Test</Item>
            <Item Key="Controller">Test</Item>
            <Item Key="ConfigDialog">AdminGenericInterfaceInvokerDefault</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.5.4. Operation

The operation is used to perform an action within OTRS, this action is requested by the external system and can include special parameters in order that OTRS receives to correctly execute the action. After the action is performed, OTRS sends a defined confirmation the external system.

2.5.4.1. Operation backend

Next we will see how to develop a new Operation, each operation has to implement these subroutines

- new
- Run

We should implement each one of this methods in order to be able to execute the action handled by the provider ('Kernel/GenericInterface/Provider.pm').

2.5.4.1.1. Code example

In this section a sample operation module is shown and each subroutine is explained.

```
# --
# Kernel/GenericInterface/Operation/Test/Test.pm - GenericInterface test operation backend
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::GenericInterface::Operation::Test::Test;

use strict;
use warnings;

use Kernel::System::VariableCheck qw(IsHashRefWithData);

use vars qw(@ISA $VERSION);
```

This is common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

We also include VariableCheck module to perform certain validation over some variables.

```
sub new {
    my ( $Type, %Param ) = @_;

    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for my $Needed (qw(DebuggerObject)) {
        if ( !$Param{$Needed} ) {
            return {
                Success      => 0,
                ErrorMessage => "Got no $Needed!"
            };
        }

        $Self->{$Needed} = $Param{$Needed};
    }

    return $Self;
}
```

The constructor 'new' creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module have to be created in "new".

```
sub Run {
    my ( $Self, %Param ) = @_;

    # check data - only accept undef or hash ref
    if ( defined $Param{Data} && ref $Param{Data} ne 'HASH' ) {
        return $Self->{DebuggerObject}->Error(
```

```
            Summary => 'Got Data but it is not a hash ref in Operation Test backend)!'
        );
    }

    if ( defined $Param{Data} && $Param{Data}->{TestError} ) {
        return {
            Success      => 0,
            ErrorMessage => "Error message for error code: $Param{Data}->{TestError}",
            Data         => {
                ErrorData => $Param{Data}->{ErrorData},
            },
        };
    }

    # copy data
    my $ReturnData;

    if ( ref $Param{Data} eq 'HASH' ) {
        $ReturnData = \%{ $Param{Data} };
    }
    else {
        $ReturnData = undef;
    }

    # return result
    return {
        Success => 1,
        Data    => $ReturnData,
    };
}
```

The 'Run' function is the main part of each operation, it receives all internal mapped data from remote system needed by the provider to execute the action, it performs the action and returns the result to the provider to be external mapped and deliver back to the remote system.

This particular example returns the same data as came from the remote system, unless "TestError" parameter is passed in this case it returns an error.

```
1;

=back

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut

=cut
```

2.5.4.1.2. Configuration Example

There is the need to register this operation module to be accessible in the OTRS GUI. This can be done using the xml configuration below.

```
<ConfigItem Name="GenericInterface::Operation::Module###Test::Test" Required="0" Valid="1">
    <Description Translatable="1">GenericInterface module registration for the operation
 layer.</Description>
    <Group>GenericInterface</Group>
    <SubGroup>GenericInterface::Operation::ModuleRegistration</SubGroup>
    <Setting>
```

```
        <Hash>
            <Item Key="Name">Test</Item>
            <Item Key="Controller">Test</Item>
            <Item Key="ConfigDialog">AdminGenericInterfaceOperationDefault</Item>
        </Hash>
    </Setting>
</ConfigItem>
```

2.6. Scheduler Task Handler Modules

2.6.1. Task Handler

The task handler modules are used to perform actions within OTRS asynchronously via OTRS scheduler (a separated "daemon like" process) made specifically to execute this kind of jobs.

2.6.1.1. Task Handler backend

Next we will see how to develop a new scheduler task handler backend. Each task handler backend has to implement these subroutines:

- new
- Run

We should implement each one of this methods in order to be able to execute tasks. All task handler backends are used by main scheduler task handler module (Kernel/scheduler/TaskHandler.pm).

Currently the scheduler only has one working task handler backend named "GenericInterface". This task handler backend executes Generic Interface Invoker modules in the background.

For example:

By using the OTRS Scheduler combined with the Generic Interface a user does not need to wait in the New phone ticket screen after creating a ticket until a Remote System responds to a web service request triggered by an event like "TicketCreate", instead this task can be delegated to the OTRS scheduler to run it on the background.

There is no need to create a new task handler backend for the Generic Interface, unless a different behavior from the current one is needed. to delegate Generic Interface Invoker tasks to the OTRS Scheduler you need to configure the Invoker event trigger as asynchronous.

2.6.1.1.1. Code example

In this section a sample scheduler task handler module is shown and each subroutine is explained.

```
# --
# Kernel/Scheduler/TaskHandler/Test.pm - Scheduler task handler test backend
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Scheduler::TaskHandler::Test;
```

```
use strict;
use warnings;

use vars qw(@ISA);
```

This is common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

```
sub new {
    my ( $Type, %Param ) = @_;

    my $Self = {};
    bless( $Self, $Type );

    # check needed objects
    for my $Needed (qw(MainObject ConfigObject LogObject DBObject TimeObject)) {
        $Self->{$Needed} = $Param{$Needed} || die "Got no $Needed!";
    }

    return $Self;
}
```

The constructor 'new' creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module have to be created in "new".

```
sub Run {
    my ( $Self, %Param ) = @_;

    # check data - we need a hash ref
    if ( $Param{Data} && ref $Param{Data} ne 'HASH' ) {
        $Self->{LogObject}->Log(
            Priority => 'error',
            Message  => 'Got no valid Data!',
        );

        return {
            Success => 0,
        };
    }

    # create tmp file
    if ( $Param{Data}->{File} ) {
        my $Content = 123;
        return if !$Self->{MainObject}->FileWrite(
            Location => $Param{Data}->{File},
            Content  => \$Content,
        );
    }

    # re schedule with new time
    return {
        Success    => $Param{Data}->{Success},
        ReSchedule => $Param{Data}->{ReSchedule},
        DueTime    => $Param{Data}->{ReScheduleDueTime},
        Data       => $Param{Data}->{ReScheduleData},
    };
}
```

The 'Run' function is the main part of the module it checks the incoming data, make calls to other OTRS modules to execute the tasks, waits for the task result and based on the result information it determines if the task was successfully executed or not and if the task needs to be re-schedule itself, when and with which data.

For this testing example if the parameter "File" exists in the incoming data, a new file is to be created on the file system by the task handler, depending of the success of this,

the task handler will continue to the rest part of the code where base it decisions for the returning structure on the previously provided data.

On real task handler backends the decisions for the returning structure could be based on the task handler backend logics, the task execution results, configuration settings, etc.

```
1;

=back

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut
```

2.7. Dynamic Fields

2.7.1. Overview

Dynamic Fields are custom fields that can be added to a screen to enhance and add information to an object (e.g. a ticket or an article).

The Dynamic Fields are the evolution of the ticket and article Free Fields ("TikcetFreeText", "TicketFreeKey", "TicketFreeTime", "ArticleFreeText", "ArticleFreeKey" and "ArticleFreeTime") from older versions of OTRS.

From OTRS version 3.1 the old Free Fields has been replaced with the new Dynamic Fields, for a better backward compatibility and data preservation when updating from previous versions, a migration script has been developed to convert the existing Free Fields to Dynamic Fields and to move their values from the *ticket* and *article* tables in the database to new dynamic fields tables.

Note

Any custom development that uses Free Fields needs to be ported to the new Dynamic Fields code structure, otherwise it will not work anymore. For this reason is very important to know that only updated installations of OTRS 3.0 has the old Free Fields converted to Dynamic Fields, new or clean installations of OTRS has no Dynamic Fields defined "out of the box" and any Dynamic Field needed by the custom development needs to be added.

The restriction on the number of the fields per ticket or article has been removed, this means that a ticket or article could have as many fields as needed. and now it is also possible to use the Dynamic Fields framework for other objects rather than just ticket or article.

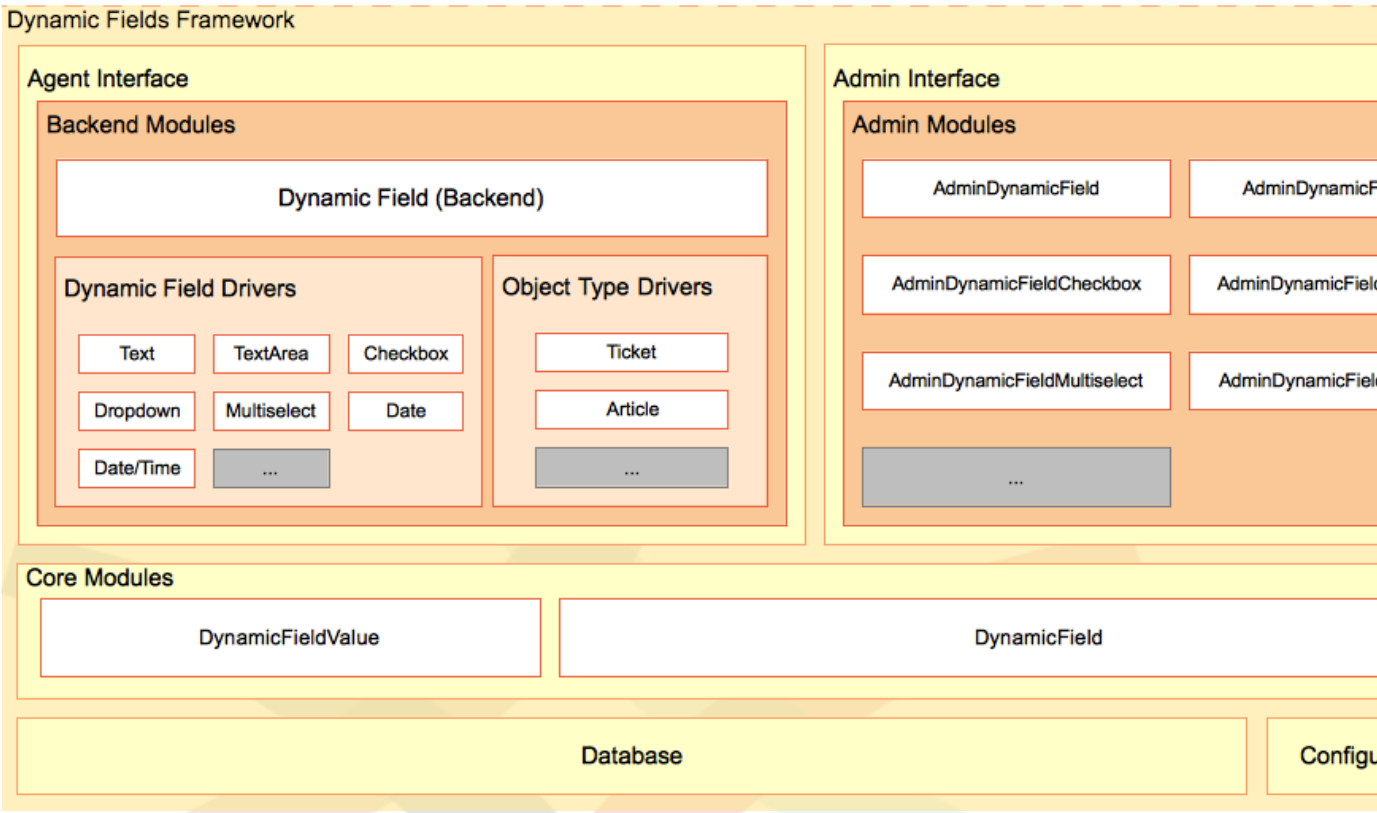
The new Dynamic Fields can store the same data types as the Free Fields (Text and Date/Time), and they can be also defined as them (Single line input, drop-down and date-time). but Dynamic Fields goes beyond that, a new data type: Integer, has been added and also new options to define the fields like Multiple-line inputs, check-boxes, Multiple-select and date (without time) fields. Each field type defines its own data type.

Due to its modular design each Dynamic Field type can be seen as a plug-in to a framework, and this plug-in can be a OTRS standard package to extend the available types of the Dynamic Fields or even to extend current Dynamic Field with more functions.

2.7.2. Dynamic Fields Framework

Before creating new Dynamic Fields is necessary to understand its framework and how OTRS screens interact with them, as well as their underlying API.

The following picture shows the architecture of the Dynamic Fields framework.



2.7.2.1. Dynamic Field Backend Modules

2.7.2.1.1. Dynamic Field (Backend)

Normally called as "BackendObject" in the frontend modules is the mediator between the frontend modules and each specific Dynamic Field implementation or Driver. It defines a Generic middle API for all Dynamic Field Drivers, and each Driver has the responsibility to implement the middle API for the specific needs for the field.

The Dynamic Field Banckend is the master controller of all the Drivers, each function in this module is responsible to check the required parameters and call the same function in the specific Driver according to the Dynamic Field Configuration parameter received.

This module is also responsible to call specific functions on each Object Type Delegate (like Ticket or Article) e.g. to add a history entry or fire an event.

This module is located in $OTRS_HOME/Kernel/System/DynamicField/Backend.pm.

2.7.2.1.2. Dynamic Field Drivers

A Dynamic Field Driver is the implementation of the Dynamic Field. Each Driver must implement all the mandatory functions specified in the Backend. (there are some functions that depends on a behavior and it is not needed to implement those if the Dynamic Field does not have that particular behavior)

A Driver is responsible to know how to get its own value or values from a web request, or from a profile (like a search profile), it also needs to know the HTML code to render

the field in edit or display screens, or how to interact with the stats module, among other functions.

This modules are located in $OTRS_HOME/Kernel/System/DynamicField/Driver/*.pm.

It exists some base drivers like Base.pm BaseText.pm, BaseSelect.pm and BaseDateTime.pm, that implements common functions for certain drivers (e.g Driver TextArea.pm uses BaseText.pm that also uses Base.pm then TextArea only needs to implement the functions that are missing in Base.pm and BateText.pm or the ones that are special cases)

The following is the Drivers inheritance tree:

- Base.pm
  - BaseText.pm
    - Text.pm
    - TextArea.pm
  - BaseSelect.pm
    - Dropdown.pm
    - Multiselect.pm
  - BaseDateTime.pm
    - DateTime.pm
    - Date.pm
  - Checkbox.pm

2.7.2.1.3. Object Type Delegate

An Object Type Delegate is responsible to perform specific functions on the object linked to the dynamic field. This functions are triggered by the BackendObject as they are needed.

This modules are located in $OTRS_HOME/Kernel/System/DynamicField/ObjectType/*.pm.

2.7.2.2. Dynamic Fields Admin Modules

To manage the Dynamic Fields (Add, Edit and List) a series of modules has been already developed. There is one specific master module (AdminDynamicField.pm) that shows the list of defined Dynamic Fields, and from within other modules are called to create new Dynamic Fields or modify an existing ones.

Normally a Dynamic Field Driver needs its own Admin Module (Admin Dialog) to define its properties, this dialog might differ from other Drivers. But this is not mandatory, Drivers can share Admin Dialogs, if they can provide add needed information for all the Drivers that are linked to them, no matter if they are from different type. What is mandatory is that each Driver must be linked to an Admin Dialog. (e.g. Text and TextArea Drivers share AdminDynamicFieldText.pm Admin Dialog, and Date and Date/Time Drivers share AdminDynamicFieldDateTime.pm Admin Dialog).

Admin Dialogs follows the normal OTRS Admin Module rules and architecture. But for standardization all configuration common parts to all Dynamic Fields should have the same look and feel among all Admin Dialogs.

This modules are located in $OTRS_HOME/Kernel/Modules/*.pm.

Note

Each Admin Dialog needs its corresponding HTML template file (.dtl).

2.7.2.3. Dynamic Fields Core Modules

This modules reads and writes the Dynamic Fields information from and to the database tables.

2.7.2.3.1. "DynamicField.pm" Core Module

This module is responsible to manage the Dynamic Field definitions, it provides the basic API for add, change, delete, list and get Dynamic Fields and is located in $OTRS_HOME/Kernel/System/DynamicField.pm.

2.7.2.3.2. "DynamicFieldValue.pm" Core Module

This module is responsible to read and write Dynamic Field values into the from and into the database this module is highly use by the Drivers and is located in $OTRS_HOME/Kernel/System/DynamicFieldValue.pm.

2.7.2.4. Dynamic Fields Database Tables

There are two tables in the database to store the dynamic field information:

dynamic_field: Used by the Core Module DynamicField.pm, it stores the Dynamic Field definitions.

dynamic_field_value: Used by the Core Module DynamicFieldValue.pm to save the Dynamic Field values for each Dynamic Field and each Object Type instance.

2.7.2.5. Dynamic Fields Configuration Files

The Backend module needs a way to know which Drivers exists and since the amount of Drivers can be easily extended, the easiest way to manage them is to use the system configuration, where the information of Dynamic Field Drivers and ObjectType Drivers can be stored and extended.

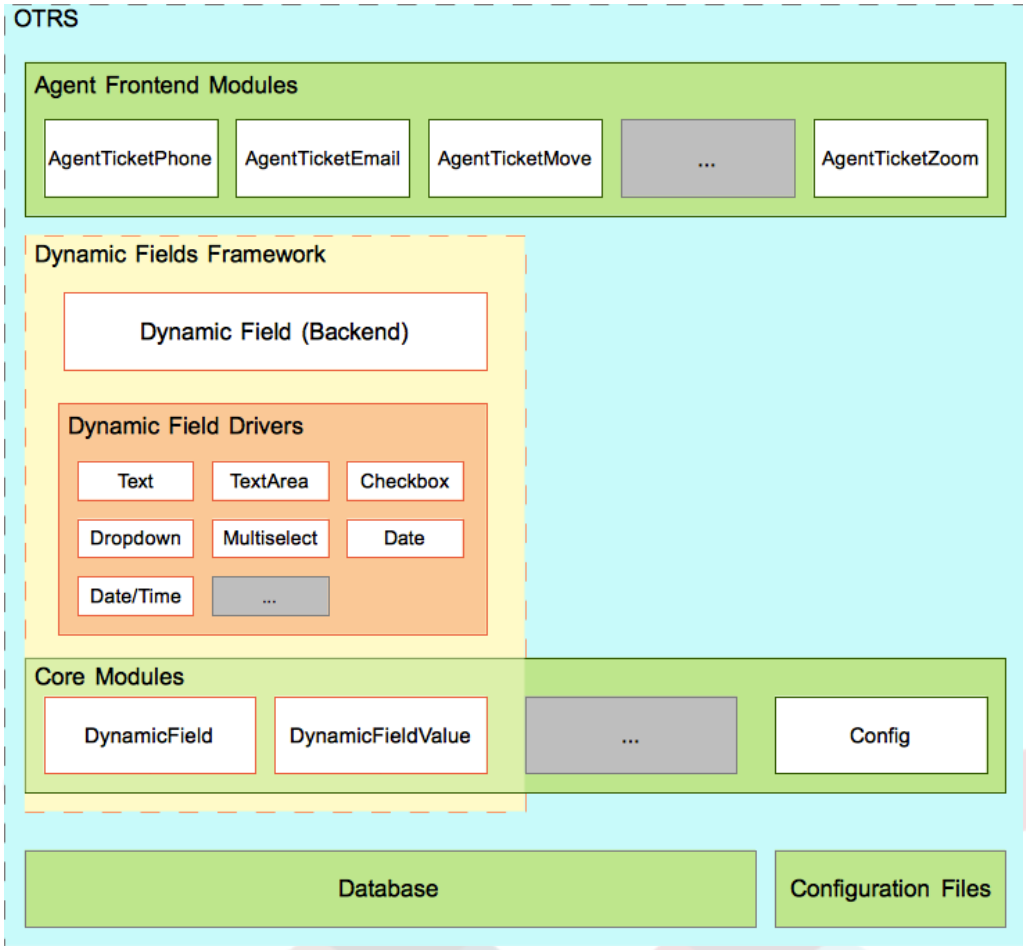
The master Admin Module also needs to know this information about the available Dynamic Field Drivers to use the Admin Dialog linked with, to create or modify the Dynamic Fields.

Frontend modules needs to read the system configuration to know which Dynamic Fields are active for each screen and which ones are also mandatory. for example: Ticket::Frontend::AgentTicketPhone###DynamicField stores the active, mandatory and inactive Dynamic Fields for New Phone Ticket Screen.

2.7.3. Dynamic Field Interaction With Frontend Modules

Knowing about how Frontend modules interact with Dynamic fields is not strictly necessary to extend Dynamic Fields for the Ticket or Article objects, since all the screens that could use the Dynamic Fields are already prepared. But in case of custom developments or to extend the Dynamic Fields to other objects is very useful to know how to access Dynamic Fields framework from a Frontend Module.

The following picture shows a simple example of how the Dynamic Fields interact with other OTRS framework parts.



The first step is that the Frontend module reads the configured Dynamic Fields for example *AgentTicketNote* should read *Ticket::Frontend::AgentTicketNote###DynamicField* setting. This setting can be used as the filter parameter for DynamicField Core Module function *DynamicFieldListGet()*. The screen can store the results of this function to have the list of the Dynamic Fields activated for this particular screen.

Next, the screen should try to get the values from the web request, it can use the BackendObject function *EditFieldValueGet()* for this purpose. and can use this values to trigger ACLs. The BackendObject will use each Driver to perform the specific actions for all functions.

To continue, the screen should get the HTML for each field to display it, the BackendObject function *EditFieldRender()* can be used to perform this action and the ACLs restriction as well as the Values from the web request can be passed to this function in order to get better results. in case of a submit the screen could also use the BackendObject function *EditFieldValueValidate()* to check the mandatory fields.

Note

Other screens could use *DisplayFieldRender()* instead of *EditFieldRender()* if the screen only shows the field value, and in such case no value validation is needed.

To store the value of the Dynamic Field is necessary to get the Object ID, for this example if the Dynamic Field is linked to a ticket object, the screen should already have the TicketID, otherwise if the field is linked to an article object in order to set the value of the field is necessary to create the article first. *ValueSet()* from the BackendObject can be used to set the Dynamic Field value.

In summary the Frontend modules does not need to know how each Dynamic Field works internally to get or set their values or to display them, It just needs to call the BackendObject module and use the fields in a generic way.

2.7.4. How To Extend The Dynamic Fields

There are many ways to extend the Dynamic Fields, the following sections will try to cover the most common scenarios.

2.7.4.1. Create a New Dynamic Field Type (for ticket or article objects)

To create a new Dynamic Field Type is necessary to:

- Create a Dynamic Field Driver:

  This is the main module of the new field.

- Create or use an existing Admin Dialog:

  To have a management interface and set its configuration options.

- Create a Configuration File:

  To register the new field in the Backend (or new Admin Dialogs in the framework if needed) and be able to create instances or it.

2.7.4.2. Create a New Dynamic Field Type (for other objects)

To create a new Dynamic Field Type for other objects is necessary to:

- Create a Dynamic Field Driver

  This is the main module of the new Field.

- Create an Object Type Delegate

  This is necessary, even if the "other object" does not require any specific data handling in its functions (e.g. after a value is set). All Object Type Delates must implement the functions that the Backend requires.

  Take a look in the current Object Type Delegates to implement the same functions, even if they just return a successful value for the "other object".

- Create or use an existing Admin Dialog

  To have a management interface and set its configuration options.

- Implement Dynamic Fields in the Frontend Modules

  To be able to use the Dynamic Fields.

- Create a Configuration File

  To register the new field in the Backend (or new Admin Dialogs in the framework if needed) and be able to create instances or it. And make the needed settings to show, hide or show the Dynamic Fields as Mandatory in the new screens.

2.7.4.3. Create a New package to use Dynamic Fields

To create a package to use existing dynamic fields is necessary to:

- Implement Dynamic Fields in the Frontend Modules

  To be able to use the Dynamic Fields.

- Create a Configuration File

  To give the end user the possibility to show, hide or show the Dynamic Fields as Mandatory in the new screens.

2.7.4.4. Extend Backend and Drivers Functionalities

It might be possible that the BackendObject does not have a needed function for custom developments, or could also be possible that it has the function needed, but the return format does not match the needs of the custom development, or that a new behavior is needed to execute the new or the old functions.

The easiest way to do this, is to extend the current field files, for this is necessary to create a new Backend extension file that defines the new functions and create also Drivers extensions that implement this new functions for each field. This new drivers will only need to implement the new functions since the original drivers takes care of the standard functions. All this new new files does not need a constructor as they will be loaded as a base for the BackendObject and the drivers.

The only restrictions are that the functions should be named different than the ones on the Backend and Drivers, otherwise they will be overwritten with current objects.

Put the new Backend extension into the DynamicField directory (e.g. /$OTRS_HOME/Kernel/System/DynamicField/NewPackageBackend.pm and its Drivers in /$OTRS_HOME/Kernel/System/DynamicField/Driver/NewPackage*.pm

New behaviors only need a small setting in the extensions configuration file.

To create new Backend functions is needed to:

- Create a New Backend extension module.

  To define only the new functions.

- Create the Dynamic Fields Driver extensions.

  To implement only the new functions.

- Implement New Dynamic Fields functions in the Frontend Modules

  To be able to use the new Dynamic Fields functions.

- Create a Configuration File

  To register the new backend and drivers extensions and behaviors.

2.7.4.5. Other Extensions

Other extensions could be a combination of the above examples.

2.7.5. Creating A New Dynamic Field

To illustrate this process a new Dynamic Field "Password" will be created. this new Dynamic Field Type will show a New password field to Ticket or Article objects. since is very similar to a Text Dynamic Field we will use the Base an BaseText Drivers as a basis to build this new field.

Note

This new password field implementation is just for educational purposes, it does not provide any level of security and is not recommended for production systems.

To create this new Dynamic Field we will create 4 files: a Configuration File (XML), to register the modules, an Admin Dialog Module (perl), to setup the field options, a Template Module (DTL), for the Admin Dialog and a Dynamic Field Driver (perl).

File Structure:

```
$HOME (e. g. /opt/otrs/)
|
...
|--/Kernel/
|   |--/Config/
|   |   |--/Files/
|   |   |   |DynamicFieldPassword.xml
...
|   |--/Modules/
|   |   |AdminDynamicFieldPassword.pm
...
|   |--/Output/
|   |   |--/HTML/
|   |   |   |--/Standard/
|   |   |   |   |AdminDynamicFieldPassword.dtl
...
|   |--/System/
|   |   |--/DynamicField/
|   |   |   |--/Driver/
|   |   |   |   |Password.pm
...
```

2.7.5.1. Dynamic Field Password files

2.7.5.1.1. Dynamic Field Configuration File Example

The configuration files are used to register the Dynamic Field Types (Driver) and the Object Type Drivers for the BackendObject. They also store standard registrations for Admin Modules in the framework.

2.7.5.1.1.1. Code Example:

In this section a configuration file for password Dynamic Field is shown and explained.

```
<?xml version="1.0" encoding="utf-8"?>
<otrs_config version="1.0" init="Application">
```

This is the normal header for a configuration file.

```
    <ConfigItem Name="DynamicFields::Driver###Password" Required="0" Valid="1">
        <Description Translatable="1">DynamicField backend registration.</Description>
        <Group>DynamicFieldPassword</Group>
        <SubGroup>DynamicFields::Backend::Registration</SubGroup>
        <Setting>
            <Hash>
                <Item Key="DisplayName" Translatable="1">Password</Item>
                <Item Key="Module">Kernel::System::DynamicField::Driver::Password</Item>
                <Item Key="ConfigDialog">AdminDynamicFieldPassword</Item>
            </Hash>
        </Setting>
    </ConfigItem>
```

This setting registers the Password Dynamic Field Driver for the Backend module so it can be included in the list of available Dynamic Fields Types, it also specify its own Admin Dialog in the key "ConfigDialog", this key is used by the Master Dynamic Field Admin Module to manage this new Dynamic Field Type.

```
   <ConfigItem Name="Frontend::Module###AdminDynamicFieldPassword" Required="0" Valid="1">
       <Description Translatable="1">Frontend module registration for the agent
interface.</Description>
       <Group>DynamicFieldPassword</Group>
       <SubGroup>Frontend::Admin::ModuleRegistration</SubGroup>
       <Setting>
           <FrontendModuleReg>
               <Group>admin</Group>
               <Description>Admin</Description>
               <Title Translatable="1">Dynamic Fields Text Backend GUI</Title>
               <Loader>
                   <JavaScript>Core.Agent.Admin.DynamicField.js</JavaScript>
               </Loader>
           </FrontendModuleReg>
       </Setting>
   </ConfigItem>
```

This is an standard module registration for the Password Admin Dialog in the Admin Interface.

```
</otrs_config>
```

Standard closure of a configuration file.

2.7.5.1.2. Dynamic Field Admin Dialog Example

The Admin Dialogs are standard Admin modules to manage (add or edit) the Dynamic Fields.

2.7.5.1.2.1. Code Example:

In this section an Admin Dialog for password dynamic field is shown and explained.

```
# --
# Kernel/Modules/AdminDynamicFieldPassword.pm - provides a dynamic fields password config
 view for admins
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Modules::AdminDynamicFieldPassword;

use strict;
use warnings;

use Kernel::System::VariableCheck qw(:all);
use Kernel::System::Valid;
use Kernel::System::CheckItem;
use Kernel::System::DynamicField;
```

This is the common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

```
sub new {
    my ( $Type, %Param ) = @_;
```

```
    my $Self = {%Param};
    bless( $Self, $Type );

    for (qw(ParamObject LayoutObject LogObject ConfigObject)) {
        if ( !$Self->{$_} ) {
            $Self->{LayoutObject}->FatalError( Message => "Got no $_!" );
        }
    }

    # create additional objects
    $Self->{ValidObject} = Kernel::System::Valid->new( %{$Self} );

    $Self->{DynamicFieldObject} = Kernel::System::DynamicField->new( %{$Self} );

    # get configured object types
    $Self->{ObjectTypeConfig} = $Self->{ConfigObject}->Get('DynamicFields::ObjectType');

    # get the fields config
    $Self->{FieldTypeConfig} = $Self->{ConfigObject}->Get('DynamicFields::Backend') || {};

    $Self->{DefaultValueMask} = '****';
    return $Self;
}
```

The constructor 'new' creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module are normally created here.

```
sub Run {
    my ( $Self, %Param ) = @_;

    if ( $Self->{Subaction} eq 'Add' ) {
        return $Self->_Add(
            %Param,
        );
    }
    elsif ( $Self->{Subaction} eq 'AddAction' ) {

        # challenge token check for write action
        $Self->{LayoutObject}->ChallengeTokenCheck();

        return $Self->_AddAction(
            %Param,
        );
    }
    if ( $Self->{Subaction} eq 'Change' ) {
        return $Self->_Change(
            %Param,
        );
    }
    elsif ( $Self->{Subaction} eq 'ChangeAction' ) {

        # challenge token check for write action
        $Self->{LayoutObject}->ChallengeTokenCheck();

        return $Self->_ChangeAction(
            %Param,
        );
    }
    return $Self->{LayoutObject}->ErrorScreen(
        Message => "Undefined subaction.",
    );
}
```

Run is the default function to be called by the web request, we try to make this function as simple as possible and let the helper functions to do the "hard" work.

```
sub _Add {
```

```
    my ( $Self, %Param ) = @_;

    my %GetParam;
    for my $Needed (qw(ObjectType FieldType FieldOrder)) {
        $GetParam{$Needed} = $Self->{ParamObject}->GetParam( Param => $Needed );
        if ( !$Needed ) {
            return $Self->{LayoutObject}->ErrorScreen(
                Message => "Need $Needed",
            );
        }
    }

    # get the object type and field type display name
    my $ObjectTypeName = $Self->{ObjectTypeConfig}->{ $GetParam{ObjectType} }->{DisplayName}
 || '';
    my $FieldTypeName  = $Self->{FieldTypeConfig}->{ $GetParam{FieldType} }->{DisplayName}
  || '';

    return $Self->_ShowScreen(
        %Param,
        %GetParam,
        Mode           => 'Add',
        ObjectTypeName => $ObjectTypeName,
        FieldTypeName  => $FieldTypeName,
    );
}
```

_Add() function is also pretty simple, it just get some parameters from the web request and call the *_ShowScreen()* function, normally this function is not needed to be modified.

```
sub _AddAction {
    my ( $Self, %Param ) = @_;

    my %Errors;
    my %GetParam;

    for my $Needed (qw(Name Label FieldOrder)) {
        $GetParam{$Needed} = $Self->{ParamObject}->GetParam( Param => $Needed );
        if ( !$GetParam{$Needed} ) {
            $Errors{ $Needed . 'ServerError' }        = 'ServerError';
            $Errors{ $Needed . 'ServerErrorMessage' } = 'This field is required.';
        }
    }

    if ( $GetParam{Name} ) {

        # check if name is alphanumeric
        if ( $GetParam{Name} !~ m{\A ( ?: [a-zA-Z] | \d )+ \z}xms ) {

            # add server error error class
            $Errors{NameServerError} = 'ServerError';
            $Errors{NameServerErrorMessage} =
                'The field does not contain only ASCII letters and numbers.';
        }

        # check if name is duplicated
        my %DynamicFieldsList = %{
            $Self->{DynamicFieldObject}->DynamicFieldList(
                Valid      => 0,
                ResultType => 'HASH',
                )
        };

        %DynamicFieldsList = reverse %DynamicFieldsList;

        if ( $DynamicFieldsList{ $GetParam{Name} } ) {

            # add server error error class
            $Errors{NameServerError}        = 'ServerError';
            $Errors{NameServerErrorMessage} = 'There is another field with the same name.';
```

```
        }
    }

    if ( $GetParam{FieldOrder} ) {

        # check if field order is numeric and positive
        if ( $GetParam{FieldOrder} !~ m{\A ( ?: \d )+ \z}xms ) {

            # add server error error class
            $Errors{FieldOrderServerError}        = 'ServerError';
            $Errors{FieldOrderServerErrorMessage} = 'The field must be numeric.';
        }
    }

    for my $ConfigParam (
        qw(
        ObjectType ObjectTypeName FieldType FieldTypeName DefaultValue ValidID ShowValue
        ValueMask
        )
        )
    {
        $GetParam{$ConfigParam} = $Self->{ParamObject}->GetParam( Param => $ConfigParam );
    }

    # uncorrectable errors
    if ( !$GetParam{ValidID} ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Need ValidID",
        );
    }

    # return to add screen if errors
    if (%Errors) {
        return $Self->_ShowScreen(
            %Param,
            %Errors,
            %GetParam,
            Mode => 'Add',
        );
    }

    # set specific config
    my $FieldConfig = {
        DefaultValue => $GetParam{DefaultValue},
        ShowValue    => $GetParam{ShowValue},
        ValueMask    => $GetParam{ValueMask} || $Self->{DefaultValueMask},
    };

    # create a new field
    my $FieldID = $Self->{DynamicFieldObject}->DynamicFieldAdd(
        Name       => $GetParam{Name},
        Label      => $GetParam{Label},
        FieldOrder => $GetParam{FieldOrder},
        FieldType  => $GetParam{FieldType},
        ObjectType => $GetParam{ObjectType},
        Config     => $FieldConfig,
        ValidID    => $GetParam{ValidID},
        UserID     => $Self->{UserID},
    );

    if ( !$FieldID ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Could not create the new field",
        );
    }

    return $Self->{LayoutObject}->Redirect(
        OP => "Action=AdminDynamicField",
    );
}
```

The *_AddAction()* function gets the configuration parameters from a new dynamic field, it validates the Dynamic Field name only contains letters and numbers. This function could validate any other parameter.

Name, Label, FieldOrder, Validity are common parameters for all Dynamic Fields and they are required. Each Dynamic Field has its specific configuration that must contain at least the DefaultValue parameter, in this case it also have ShowValue and ValueMask parameters for Password field.

If the field has the ability to store a fixed list of values they need should be stored in the PossibleValues parameter inside the specific configuration hash.

As in other Admin Modules, if a parameter is not valid this function returns to the Add screen highlighting the erroneous form fields.

If all the parameters are correct it creates a new Dynamic Field.

```
sub _Change {
    my ( $Self, %Param ) = @_;

    my %GetParam;
    for my $Needed (qw(ObjectType FieldType)) {
        $GetParam{$Needed} = $Self->{ParamObject}->GetParam( Param => $Needed );
        if ( !$Needed ) {
            return $Self->{LayoutObject}->ErrorScreen(
                Message => "Need $Needed",
            );
        }
    }

    # get the object type and field type display name
    my $ObjectTypeName = $Self->{ObjectTypeConfig}->{ $GetParam{ObjectType} }->{DisplayName}
 || '';
    my $FieldTypeName  = $Self->{FieldTypeConfig}->{ $GetParam{FieldType} }->{DisplayName}
  || '';

    my $FieldID = $Self->{ParamObject}->GetParam( Param => 'ID' );

    if ( !$FieldID ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Need ID",
        );
    }

    # get dynamic field data
    my $DynamicFieldData = $Self->{DynamicFieldObject}->DynamicFieldGet(
        ID => $FieldID,
    );

    # check for valid dynamic field configuration
    if ( !IsHashRefWithData($DynamicFieldData) ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Could not get data for dynamic field $FieldID",
        );
    }

    my %Config = ();

    # extract configuration
    if ( IsHashRefWithData( $DynamicFieldData->{Config} ) ) {
        %Config = %{ $DynamicFieldData->{Config} };
    }

    return $Self->_ShowScreen(
        %Param,
        %GetParam,
        %${DynamicFieldData},
        %Config,
        ID             => $FieldID,
```

```
        Mode           => 'Change',
        ObjectTypeName => $ObjectTypeName,
        FieldTypeName  => $FieldTypeName,
    );
}
```

The "_Change" function is very similar to the "_Add" function but since this function is used to edit an existing field it needs to validated the FieldID parameter and gather the current Dynamic Field data.

```
sub _ChangeAction {
    my ( $Self, %Param ) = @_;

    my %Errors;
    my %GetParam;

    for my $Needed (qw(Name Label FieldOrder)) {
        $GetParam{$Needed} = $Self->{ParamObject}->GetParam( Param => $Needed );
        if ( !$GetParam{$Needed} ) {
            $Errors{ $Needed . 'ServerError' }        = 'ServerError';
            $Errors{ $Needed . 'ServerErrorMessage' } = 'This field is required.';
        }
    }

    my $FieldID = $Self->{ParamObject}->GetParam( Param => 'ID' );
    if ( !$FieldID ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Need ID",
        );
    }

    if ( $GetParam{Name} ) {

        # check if name is lowercase
        if ( $GetParam{Name} !~ m{\A ( ?: [a-zA-Z] | \d )+ \z}xms ) {

            # add server error error class
            $Errors{NameServerError} = 'ServerError';
            $Errors{NameServerErrorMessage} =
                'The field does not contain only ASCII letters and numbers.';
        }

        # check if name is duplicated
        my %DynamicFieldsList = %{
            $Self->{DynamicFieldObject}->DynamicFieldList(
                Valid      => 0,
                ResultType => 'HASH',
                )
        };

        %DynamicFieldsList = reverse %DynamicFieldsList;

        if (
            $DynamicFieldsList{ $GetParam{Name} } &&
            $DynamicFieldsList{ $GetParam{Name} } ne $FieldID
            )
        {

            # add server error class
            $Errors{NameServerError}        = 'ServerError';
            $Errors{NameServerErrorMessage} = 'There is another field with the same name.';
        }
    }

    if ( $GetParam{FieldOrder} ) {

        # check if field order is numeric and positive
        if ( $GetParam{FieldOrder} !~ m{\A ( ?: \d )+ \z}xms ) {

            # add server error error class
```

```
            $Errors{FieldOrderServerError}        = 'ServerError';
            $Errors{FieldOrderServerErrorMessage} = 'The field must be numeric.';
        }
    }

    for my $ConfigParam (
        qw(
        ObjectType ObjectTypeName FieldType FieldTypeName DefaultValue ValidID ShowValue
        ValueMask
        )
        )
    {
        $GetParam{$ConfigParam} = $Self->{ParamObject}->GetParam( Param => $ConfigParam );
    }

    # uncorrectable errors
    if ( !$GetParam{ValidID} ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Need ValidID",
        );
    }

    # get dynamic field data
    my $DynamicFieldData = $Self->{DynamicFieldObject}->DynamicFieldGet(
        ID => $FieldID,
    );

    # check for valid dynamic field configuration
    if ( !IsHashRefWithData($DynamicFieldData) ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Could not get data for dynamic field $FieldID",
        );
    }

    # return to change screen if errors
    if (%Errors) {
        return $Self->_ShowScreen(
            %Param,
            %Errors,
            %GetParam,
            ID   => $FieldID,
            Mode => 'Change',
        );
    }

    # set specific config
    my $FieldConfig = {
        DefaultValue => $GetParam{DefaultValue},
        ShowValue    => $GetParam{ShowValue},
        ValueMask    => $GetParam{ValueMask},
    };

    # update dynamic field (FieldType and ObjectType cannot be changed; use old values)
    my $UpdateSuccess = $Self->{DynamicFieldObject}->DynamicFieldUpdate(
        ID         => $FieldID,
        Name       => $GetParam{Name},
        Label      => $GetParam{Label},
        FieldOrder => $GetParam{FieldOrder},
        FieldType  => $DynamicFieldData->{FieldType},
        ObjectType => $DynamicFieldData->{ObjectType},
        Config     => $FieldConfig,
        ValidID    => $GetParam{ValidID},
        UserID     => $Self->{UserID},
    );

    if ( !$UpdateSuccess ) {
        return $Self->{LayoutObject}->ErrorScreen(
            Message => "Could not update the field $GetParam{Name}",
        );
    }

    return $Self->{LayoutObject}->Redirect(
```

```
        OP => "Action=AdminDynamicField",
    );
}
```

_ChangeAction() is very similar to *_AddAction()*", but adapted for the update of an existing field instead of creating a new one.

```
sub _ShowScreen {
    my ( $Self, %Param ) = @_;

    $Param{DisplayFieldName} = 'New';

    if ( $Param{Mode} eq 'Change' ) {
        $Param{ShowWarning}      = 'ShowWarning';
        $Param{DisplayFieldName} = $Param{Name};
    }

    # header
    my $Output = $Self->{LayoutObject}->Header();
    $Output .= $Self->{LayoutObject}->NavigationBar();

    # get all fields
    my $DynamicFieldList = $Self->{DynamicFieldObject}->DynamicFieldListGet(
        Valid => 0,
    );

    # get the list of order numbers (is already sorted).
    my @DynamicfieldOrderList;
    for my $Dynamicfield ( @{$DynamicFieldList} ) {
        push @DynamicfieldOrderList, $Dynamicfield->{FieldOrder};
    }

    # when adding we need to create an extra order number for the new field
    if ( $Param{Mode} eq 'Add' ) {

        # get the last element form the order list and add 1
        my $LastOrderNumber = $DynamicfieldOrderList[-1];
        $LastOrderNumber++;

        # add this new order number to the end of the list
        push @DynamicfieldOrderList, $LastOrderNumber;
    }

    my $DynamicFieldOrderSrtg = $Self->{LayoutObject}->BuildSelection(
        Data          => \@DynamicfieldOrderList,
        Name          => 'FieldOrder',
        SelectedValue => $Param{FieldOrder} || 1,
        PossibleNone  => 0,
        Class         => 'W50pc Validate_Number',
    );

    my %ValidList = $Self->{ValidObject}->ValidList();

    # create the Validity select
    my $ValidityStrg = $Self->{LayoutObject}->BuildSelection(
        Data         => \%ValidList,
        Name         => 'ValidID',
        SelectedID   => $Param{ValidID} || 1,
        PossibleNone => 0,
        Translation  => 1,
        Class        => 'W50pc',
    );

    # define config field specific settings
    my $DefaultValue = ( defined $Param{DefaultValue} ? $Param{DefaultValue} : '' );

    # create the Show value select
    my $ShowValueStrg = $Self->{LayoutObject}->BuildSelection(
        Data => [ 'No', 'Yes' ],
        Name => 'ShowValue',
```

```
        SelectedValue => $Param{ShowValue} || 'No',
        PossibleNone  => 0,
        Translation   => 1,
        Class         => 'W50pc',
    );

    # generate output
    $Output .= $Self->{LayoutObject}->Output(
        TemplateFile => 'AdminDynamicFieldPassword',
        Data         => {
            %Param,
            ValidityStrg          => $ValidityStrg,
            DynamicFieldOrderSrtg => $DynamicFieldOrderSrtg,
            DefaultValue          => $DefaultValue,
            ShowValueStrg         => $ShowValueStrg,
            ValueMask             => $Param{ValueMask} || $Self->{DefaultValueMask},
        },
    );

    $Output .= $Self->{LayoutObject}->Footer();

    return $Output;
}
1;
```

The *_ShowScreen* function is used to set and define the HTML elements and blocks from a DTL template to generate the Admin Dialog HTML code.

2.7.5.1.3. Dynamic Field DTL Template for Admin Dialog Example

The template is the place where the HTML code of the dialog is stored.

2.7.5.1.3.1. Code Example:

In this section an Admin Dialog DTL Template for the password Dynamic Field is shown and explained.

```
# --
# AdminDynamicFieldPassword.dtl - provides HTML form for AdminDynamicFieldPassword
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --
```

This is common header that can be found in common OTRS modules.

```
<div class="MainBox ARIARoleMain LayoutFixedSidebar SidebarFirst">
    <h1>$Text{"Dynamic Fields"} - $Text{"$Data{"ObjectTypeName"}"}: $Text{"$Data{"Mode"}"}
 $Text{"$Data{"FieldTypeName"}"} $Text{"Field"}</h1>

    <div class="Clear"></div>

    <div class="SidebarColumn">
        <div class="WidgetSimple">
            <div class="Header">
                <h2>$Text{"Actions"}</h2>
            </div>
            <div class="Content">
                <ul class="ActionList">
                    <li>
                        <a href="$Env{"Baselink"}Action=AdminDynamicField"
 class="CallForAction"><span>$Text{"Go back to overview"}</span></a>
                    </li>
                </ul>
            </div>
```

```
        </div>
    </div>
```

This is part of the code has the main box and also the actions side bar, no modifications are needed in this section.

```
    <div class="ContentColumn">
        <form action="$Env{"CGIHandle"}" method="post" class="Validate
 PreventMultipleSubmits">
            <input type="hidden" name="Action" value="AdminDynamicFieldPassword" />
            <input type="hidden" name="Subaction" value="$QData{"Mode"}Action" />
            <input type="hidden" name="ObjectType" value="$QData{"ObjectType"}" />
            <input type="hidden" name="FieldType" value="$QData{"FieldType"}" />
            <input type="hidden" name="ID" value="$QData{"ID"}" />
```

In this section of the code is defined the right part of the dialog, notice that the value of the Action hidden input must match with the name of the Admin Dialog.

```
            <div class="WidgetSimple">
                <div class="Header">
                    <h2>$Text{"General"}</h2>
                </div>
                <div class="Content">
                    <div class="LayoutGrid ColumnsWithSpacing">
                        <div class="Size1of2">
                            <fieldset class="TableLike">
                                <label class="Mandatory" for="Name"><span class="Marker">*</
span> $Text{"Name"}:</label>
                                <div class="Field">
                                    <input id="Name" class="W50pc $QData{"NameServerError"}
 $QData{"ShowWarning"}  Validate_Alphanumeric" type="text" maxlength="200"
 value="$QData{"Name"}" name="Name"/>
                                    <div id="NameError" class="TooltipErrorMessage"><p>
$Text{"This field is required, and the value should be alphabetic and numeric characters
 only."}</p></div>
                                    <div id="NameServerError"
 class="TooltipErrorMessage"><p>$Text{"$Data{"NameServerErrorMessage"}"}</p></div>
                                    <p class="FieldExplanation">$Text{"Must be unique and
 only accept alphabetic and numeric characters."}</p>
                                    <p class="Warning Hidden">$Text{"Changing this value
 will require manual changes in the system."}</p>
                                </div>
                                <div class="Clear"></div>

                                <label class="Mandatory" for="Label"><span
 class="Marker">*</span> $Text{"Label"}:</label>
                                <div class="Field">
                                    <input id="Label" class="W50pc
 $QData{"LabelServerError"} Validate_Required" type="text" maxlength="200"
 value="$QData{"Label"}" name="Label"/>
                                    <div id="LabelError" class="TooltipErrorMessage"><p>
$Text{"This field is required."}</p></div>
                                    <div id="LabelServerError"
 class="TooltipErrorMessage"><p>$Text{"$Data{"LabelServerErrorMessage"}"}</p></div>
                                    <p class="FieldExplanation">$Text{"This is the name to
 be shown on the screens where the field is active."}</p>
                                </div>
                                <div class="Clear"></div>

                                <label class="Mandatory" for="FieldOrder"><span
 class="Marker">*</span> $Text{"Field order"}:</label>
                                <div class="Field">
                                    $Data{"DynamicFieldOrderSrtg"}
                                    <div id="FieldOrderError"
 class="TooltipErrorMessage"><p>$Text{"This field is required and must be numeric."}</p></
div>
                                    <div id="FieldOrderServerError"
 class="TooltipErrorMessage"><p>$Text{"$Data{"FieldOrderServerErrorMessage"}"}</p></div>
```

```
                                <p class="FieldExplanation">$Text{"This is the order in
which this field will be shown on the screens where is active."}</p>
                            </div>
                            <div class="Clear"></div>
                        </fieldset>
                    </div>
                    <div class="Size1of2">
                        <fieldset class="TableLike">
                            <label for="ValidID">$Text{"Validity"}:</label>
                            <div class="Field">
                                $Data{"ValidityStrg"}
                            </div>
                            <div class="Clear"></div>

                            <div class="SpacingTop"></div>
                            <label for="FieldTypeName">$Text{"Field type"}:</label>
                            <div class="Field">
                                <input id="FieldTypeName" readonly="readonly"
class="W50pc" type="text" maxlength="200" value="$QData{"FieldTypeName"}"
name="FieldTypeName"/>
                                <div class="Clear"></div>
                            </div>

                            <div class="SpacingTop"></div>
                            <label for="ObjectTypeName">$Text{"Object type"}:</label>
                            <div class="Field">
                                <input id="ObjectTypeName" readonly="readonly"
class="W50pc" type="text" maxlength="200" value="$QData{"ObjectTypeName"}"
name="ObjectTypeName"/>
                                <div class="Clear"></div>
                            </div>
                        </fieldset>
                    </div>
                </div>
            </div>
        </div>
```

This first widget contains the common form attributes for the Dynamic Fields, for consistency with other Dynamic Fields is recommended to leave this part of the code unchanged.

```
        <div class="WidgetSimple">
            <div class="Header">
                <h2>$Text{"$Data{"FieldTypeName"}"} $Text{"Field Settings"}</h2>
            </div>
            <div class="Content">
                <fieldset class="TableLike">

                    <label for="DefaultValue">$Text{"Default value"}:</label>
                    <div class="Field">
                        <input id="DefaultValue" class="W50pc" type="text"
maxlength="200" value="$QData{"DefaultValue"}" name="DefaultValue"/>
                        <p class="FieldExplanation">$Text{"This is the default value for
this field."}</p>
                    </div>
                    <div class="Clear"></div>

                    <label for="ShowValue">$Text{"Show value"}:</label>
                    <div class="Field">
                        $Data{"ShowValueStrg"}
                        <p class="FieldExplanation">
                            $Text{"To reveal the field value in non edit screens ( e.g.
Ticket Zoom Screen )"}
                        </p>
                    </div>
                    <div class="Clear"></div>

                    <label for="ValueMask">$Text{"Hidden value mask"}:</label>
                    <div class="Field">
                        <input id="ValueMask" class="W50pc" type="text" maxlength="200"
value="$QData{"ValueMask"}" name="ValueMask"/>
```

```
                            <p class="FieldExplanation">
                                $Text{"This is the alternate value to show if Show value is
 set to "No" ( Default: **** )."}
                            </p>
                        </div>
                        <div class="Clear"></div>

                    </fieldset>
                </div>
            </div>
```

The second widget has the Dynamic Field specific form attributes. This is the place where new attributes cat be set and it could use Java Script and AJAX technologies to make it more easy or friendly for the end user.

```
            <fieldset class="TableLike">
                <div class="Field SpacingTop">
                    <button type="submit" class="Primary" value="$Text{"Save"}">
$Text{"Save"}</button>
                    $Text{"or"}
                    <a href="$Env{"Baselink"}Action=AdminDynamicField">$Text{"Cancel"}</a>
                </div>
                <div class="Clear"></div>
            </fieldset>
        </form>
    </div>
</div>
<!--dtl:js_on_document_complete-->
<script type="text/javascript">//<![CDATA[
$('.ShowWarning').bind('change keyup', function (Event) {
    $('p.Warning').removeClass('Hidden');
});

Core.Agent.Admin.DynamicField.ValidationInit();
//]]></script>
<!--dtl:js_on_document_complete-->
```

The final part of the file contains the "Submit" button and the "Cancel" link, as well as other needed Java Script code.

2.7.5.1.4. Dynamic Field Driver Example

The driver *is* the Dynamic Field. It contains several functions that are used wide in the OTRS framework. A driver can inherit some functions form base classes, for example TextArea driver inherit most of the functions from Base.pm and BaseText.pm and it only implements the functions that requires different logic or results. Checkbox field driver only inherits from Base.pm as all other functions are very different from any other Base driver.

> **Note**
>
> Please refer to the Perl Online Documentation (POD) of the module /Kernel/System/DynmicField/Backend.pm to have the list of all attributes and possible return data for each function.

2.7.5.1.4.1. Code Example:

In this section the Password Dynamic Field driver is shown and explained. This driver inherits some functions from Base.pm and BaseText.pm and only implements the functions that needs different results.

```
# --
# Kernel/System/DynamicField/Driver/Password.pm - Driver for DynamicField Password backend
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
```

```
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::DynamicField::Driver::Password;

use strict;
use warnings;

use Kernel::System::VariableCheck qw(:all);
use Kernel::System::DynamicFieldValue;

use base qw(Kernel::System::DynamicField::Driver::BaseText);
```

This is the common header that can be found in common OTRS modules. The class/package name is declared via the package keyword. Notice that BaseText is used as base class.

```
sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless( $Self, $Type );

    # get needed objects
    for my $Needed (qw(ConfigObject EncodeObject LogObject MainObject DBObject)) {
        die "Got no $Needed!" if !$Param{$Needed};

        $Self->{$Needed} = $Param{$Needed};
    }

    # create additional objects
    $Self->{DynamicFieldValueObject} = Kernel::System::DynamicFieldValue->new( %{$Self} );

    # set field behaviors
    $Self->{Behaviors} = {
        'IsACLReducible'               => 0,
        'IsNotificationEventCondition' => 1,
        'IsSortable'                   => 0,
        'IsFiltrable'                  => 0,
        'IsStatsCondition'             => 1,
        'IsCustomerInterfaceCapable'   => 1,
    };

    return $Self;
}
```

The constructor *new* creates a new instance of the class. According to the coding guidelines objects of other classes that are needed in this module should be created here.

It is important to define the behaviors correctly as the field might or might not be used in certain screens, functions that depends on behaviors that are not active for this particular field might not be needed to be implemented.

Note

Drivers are created only by the BankendObject and not directly from any other module.

```
sub EditFieldRender {
    my ( $Self, %Param ) = @_;

    # take config from field config
    my $FieldConfig = $Param{DynamicFieldConfig}->{Config};
    my $FieldName   = 'DynamicField_' . $Param{DynamicFieldConfig}->{Name};
```

```
    my $FieldLabel  = $Param{DynamicFieldConfig}->{Label};

    my $Value = '';

    # set the field value or default
    if ( $Param{UseDefaultValue} ) {
        $Value = ( defined $FieldConfig->{DefaultValue} ? $FieldConfig->{DefaultValue} :
 '' );
    }
    $Value = $Param{Value} if defined $Param{Value};

    # extract the dynamic field value form the web request
    my $FieldValue = $Self->EditFieldValueGet(
        %Param,
    );

    # set values from ParamObject if present
    if ( defined $FieldValue ) {
        $Value = $FieldValue;
    }

    # check and set class if necessary
    my $FieldClass = 'DynamicFieldText W50pc';
    if ( defined $Param{Class} && $Param{Class} ne '' ) {
        $FieldClass .= ' ' . $Param{Class};
    }

    # set field as mandatory
    $FieldClass .= ' Validate_Required' if $Param{Mandatory};

    # set error css class
    $FieldClass .= ' ServerError' if $Param{ServerError};

    my $HTMLString = <<"EOF";
<input type="password" class="$FieldClass" id="$FieldName" name="$FieldName"
 title="$FieldLabel" value="$Value" />
EOF

    if ( $Param{Mandatory} ) {
        my $DivID = $FieldName . 'Error';

        # for client side validation
        $HTMLString .= <<"EOF";
    <div id="$DivID" class="TooltipErrorMessage">
        <p>
            \$Text{"This field is required."}
        </p>
    </div>
EOF
    }

    if ( $Param{ServerError} ) {

        my $ErrorMessage = $Param{ErrorMessage} || 'This field is required.';
        my $DivID = $FieldName . 'ServerError';

        # for server side validation
        $HTMLString .= <<"EOF";
    <div id="$DivID" class="TooltipErrorMessage">
        <p>
            \$Text{"$ErrorMessage"}
        </p>
    </div>
EOF
    }

    # call EditLabelRender on the common Driver
    my $LabelString = $Self->EditLabelRender(
        DynamicFieldConfig => $Param{DynamicFieldConfig},
        Mandatory          => $Param{Mandatory} || '0',
        FieldName          => $FieldName,
    );
```

```
    my $Data = {
        Field => $HTMLString,
        Label => $LabelString,
    };

    return $Data;
}
```

This function is the responsible to create the HTML representation of the field and its label, and is used in the edit screens like "AgentTicketPhone", "AgentTicketNote", etc

```
sub DisplayValueRender {
    my ( $Self, %Param ) = @_;

    # set HTMLOuput as default if not specified
    if ( !defined $Param{HTMLOutput} ) {
        $Param{HTMLOutput} = 1;
    }

    my $Value;
    my $Title;

    # check if field is set to show password or not
    if (
        defined $Param{DynamicFieldConfig}->{Config}->{ShowValue}
        && $Param{DynamicFieldConfig}->{Config}->{ShowValue} eq 'Yes'
        )
    {

        # get raw Title and Value strings from field value
        $Value = defined $Param{Value} ? $Param{Value} : '';
        $Title = $Value;
    }
    else {

        # show the mask and not the value
        $Value = $Param{DynamicFieldConfig}->{Config}->{ValueMask} || '';
        $Title = 'The value of this field is hidden.'
    }

    # HTMLOuput transformations
    if ( $Param{HTMLOutput} ) {
        $Value = $Param{LayoutObject}->Ascii2Html(
            Text => $Value,
            Max => $Param{ValueMaxChars} || '',
        );

        $Title = $Param{LayoutObject}->Ascii2Html(
            Text => $Title,
            Max => $Param{TitleMaxChars} || '',
        );
    }
    else {
        if ( $Param{ValueMaxChars} && length($Value) > $Param{ValueMaxChars} ) {
            $Value = substr( $Value, 0, $Param{ValueMaxChars} ) . '...';
        }
        if ( $Param{TitleMaxChars} && length($Title) > $Param{TitleMaxChars} ) {
            $Title = substr( $Title, 0, $Param{TitleMaxChars} ) . '...';
        }
    }

    # create return structure
    my $Data = {
        Value => $Value,
        Title => $Title,
    };

    return $Data;
}
```

DisplayValueRender() function returns the field value as a plain text as well as its title (both can be translated). for this particular example we are checking if the password should be reveled or display a predefined mask by a configuration parameter in the Dynamic Field.

```
sub ReadableValueRender {
    my ( $Self, %Param ) = @_;

    my $Value;
    my $Title;

    # check if field is set to show password or not
    if (
        defined $Param{DynamicFieldConfig}->{Config}->{ShowValue}
        && $Param{DynamicFieldConfig}->{Config}->{ShowValue} eq 'Yes'
        )
    {

        # get raw Title and Value strings from field value
        $Value = defined $Param{Value} ? $Param{Value} : '';
        $Title = $Value;
    }
    else {

        # show the mask and not the value
        $Value = $Param{DynamicFieldConfig}->{Config}->{ValueMask} || '';
        $Title = 'The value of this field is hidden.'
    }

    # cut strings if needed
    if ( $Param{ValueMaxChars} && length($Value) > $Param{ValueMaxChars} ) {
        $Value = substr( $Value, 0, $Param{ValueMaxChars} ) . '...';
    }
    if ( $Param{TitleMaxChars} && length($Title) > $Param{TitleMaxChars} ) {
        $Title = substr( $Title, 0, $Param{TitleMaxChars} ) . '...';
    }

    # create return structure
    my $Data = {
        Value => $Value,
        Title => $Title,
    };

    return $Data;
}
```

This function is similar to *DisplayValueRender()* but is used in places where there is no LayoutObject.

```
1;

=back

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut

=cut
```

The normal OTRS footer for a Perl module file.

2.7.5.1.4.2. Other Functions:

The following are other functions that are might needed if the new Dynamic Field does not inherit from other classes, To see the complete code of this functions please take a look directly into the files Kernel::System::DynamicField::Driver::Base.pm and Kernel::System::DynamicField::Driver::BaseText.pm

```
sub ValueGet {
.
.
.
}
```

This function retrieves the value from the field on a specified Object. In this case we are returning the first text value, since the field only stores one text value at time.

```
sub ValueSet {
.
.
.
}
```

ValueSet() is used to store a Dynamic Field value, in this case this field only stores one text type value. Other fields could store more than one value on either ValueText, ValueDateTime, or ValueInt format.

```
sub ValueDelete {
.
.
.
}
```

This function is used to delete one field value attached to a particular object ID, is used for example if the instance of object is to be deleted, then there is no reason to have the field value stored in the database for that particular object instance.

```
sub AllValuesDelete {
.
.
.
}
```

AllValuesDelete() function is used to delete all values from a certain Dynamic Field. This function is very useful when a Dynamic Field is going to be deleted.

```
sub ValueValidate {
.
.
.
}
```

The *ValueValidate()* function is used to check if the value is consistent to its type.

```
sub SearchSQLGet {
.
```

```
.
.
}
```

This function is used by TicketSearch core module to build the internal query to search for a ticket based on this field as a search parameter.

```
sub SearchSQLOrderFieldGet {
.
.
.
}
```

The *SearchSQLOrderFieldGet* is also a helper for TicketSearch module, "$Param{TableAlias}." should be kept and "value_text" could be replaced with "value_date" or "value_int" depending on the field.

```
sub EditFieldValueGet {
.
.
.
}
```

EditFieldValueGet() is a function used in the edit screens of OTRS and its purpose is to get the value of the field, either form a template like generic agent profile or from a web request. This function gets the web request in the $Param{ParamObject}, that is a copy of the ParamObject of the Frontend Module or screen.

There are two return formats for this function, the normal: that is just the raw value or a structure: that is the pair field name => field value. for example a Date Dynamic Field returns normally the date as string, and if it should return the an structure it returns a pair for each part of the the date in the hash.

The if the result should be a structure then, normally this is used to store its values in a template, like a generic agent profile. For example a date field uses several HTML components to build the field, like the "Used" check-box and selects for year, month, day etc.

```
sub EditFieldValueValidate {
.
.
.
}
```

This function should provide at least, a method to validate if the field is empty, and return an error if the field is empty and mandatory, but it can also do more validations for other kind of fields, like if the option selected is valid, or if a date should be only in the past etc. It can provide a custom error message also.

```
sub SearchFieldRender {
.
.
.
}
```

This function is used by ticket search dialog and its similar to *EditFieldRander()*, but normally on a search screen small changes has to be done for all fields, for this example we use a HTML text input instead of a password input. In other fields like Dropdown field

is displayed as a Multiple select in order to let the user search for more than one value at a time.

```
sub SearchFieldValueGet {
.
.
.
}
```

Very similar to *EditFieldValueGet()*, but uses a different name prefix, adapted for the search dialog screen.

```
sub SearchFieldParameterBuild {
.
.
.
}
```

SearchFieldParameterBuild() is used also by the ticket search dialog to set the correct operator and value to do the search on this field. It also returns how the value should be displayed in the used search attributes, in the results page.

```
sub StatsFieldParameterBuild {
.
.
.
}
```

This function is used by the stats modules, it includes the field definition in the stats format. For fields with fixed values it also includes all this possible values and if they can be translated, take a look to the BaseSelect driver code for an example how to implement those.

```
sub StatsSearchFieldParameterBuild {
 .
 .
 .
 }
```

StatsSearchFieldParameterBuild() is very similar to the *SearchFieldParameterBuild()*, the difference id that the *SearchFieldParameterBuild()* gets the value from the search profile and this one gets the value directly from its parameters.

This function is used by statistics module.

```
sub TemplateValueTypeGet {
.
.
.
}
```

The *TemplateValueTypeGet()* function is used to know how the Dynamic Field values stored on a profile should be retrieved, as an SCALAR or as an ARRAY, and it also defines the correct name of the field in the profile.

```
sub RandomValueSet {
.
```

```
.
.
}
```

This function is used by otrs.FillDB.pl script to populate the database with some test and random data. The value inserted by this function is not really relevant. The only restriction is that the value must be compatible with the field value type.

```
sub ObjectMatch {
.
.
.
}
```

Used by the notification modules this function returns 1 if the field is present in the $Param{ObjectAttributes} parameter and if it match the given value.

2.7.6. Creating a Dynamic Field Functionality Extension

To illustrate this process a new Dynamic Field functionality extension for the function *Foo* will be added to the Bakend Object as well as in in the Text field driver.

To create this extension we will create 3 files: a Configuration File (XML), to register the modules, a Backend extension (perl), to define the new function, and a Text field Driver extension (perl), that implements the new funtion for Text fields.

File Structure:

```
$HOME (e. g. /opt/otrs/)
|
...
|--/Kernel/
|   |--/Config/
|   |   |--/Files/
|   |   |   |DynamicFieldFooExtension.xml
...
|   |--/System/
|   |   |--/DynamicField/
|   |   |   FooExtensionBackend.pm
|   |   |   |--/Driver/
|   |   |   |   |FooExtensionText.pm
...
```

2.7.6.1. Dynamic Field Foo Extension files

2.7.6.1.1. Dynamic Field Extension Configuration File Example

The configuration files are used to register the extensions for the Backend and Drivers as well as new behaviors for each driver.

> **Note**
>
> If a driver is extended with a new function, the backend will need also an extension for that function.

2.7.6.1.1.1. Code Example:

In this section a configuration file for Foo Extension is shown and explained.

```
<?xml version="1.0" encoding="utf-8"?>
<otrs_config version="1.0" init="Application">
```

```
```

This is the normal header for a configuration file.

```
    <ConfigItem Name="DynamicFields::Extension::Backend###100-Foo" Required="0" Valid="1">
        <Description Translatable="1">Dynamic Fields Extension.</Description>
        <Group>DynamicFieldFooExtension</Group>
        <SubGroup>DynamicFields::Extension::Registration</SubGroup>
        <Setting>
            <Hash>
                <Item Key="Module">Kernel::System::DynamicField::FooExtensionBackend</Item>
            </Hash>
        </Setting>
    </ConfigItem>
```

This setting registers the extension in the Backend Object, The module will be loaded from Backend as a base class.

```
    <ConfigItem Name="DynamicFields::Extension::Driver::Text###100-Foo" Required="0"
 Valid="1">
        <Description Translatable="1">Dynamic Fields Extension.</Description>
        <Group>DynamicFieldFooExtension</Group>
        <SubGroup>DynamicFields::Extension::Registration</SubGroup>
        <Setting>
            <Hash>
                <Item Key="Module">Kernel::System::DynamicField::Driver::FooExtensionText</
Item>
                <Item Key="Behaviors">
                    <Hash>
                        <Item Key="Foo">1</Item>
                    </Hash>
                </Item>
            </Hash>
        </Setting>
    </ConfigItem>
```

This is the registration for a extension in the "Text" Dynamic Field Driver the module will be loaded as a base class in the Driver. Notice also that new behaviors can be specified, this extended behaviors will be added to the behaviors that the Driver has out of the box, therefore a call to HasBehavior() to check for this new behaviors will be totally transparent.

```
</otrs_config>
```

Standard closure of a configuration file.

2.7.6.1.2. Dynamic Field Backend Extension Example

Backend extensions will be loaded transparently into the Backed itself as a base class, all defined object and properties from the Backend will be accessible in the extension.

> **Note**
>
> All new functions defined in the Backend extension should be implemented in a Driver extension.

2.7.6.1.2.1. Code Example:

In this section the Foo extension for Backend is shown and explained. The extension only defines the function Foo().

```
# --
```

```
# Kernel/System/DynamicField/FooExtensionBackend.pm - Extension for DynamicField backend
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::DynamicField::FooExtensionBackend;

use strict;
use warnings;

use Kernel::System::VariableCheck qw(:all);

=head1 NAME

Kernel::System::DynamicField::FooExtensionBackend

=head1 SYNOPSIS

DynamicFields Extension for Backend

=head1 PUBLIC INTERFACE

=over 4

=cut
```

This is the common header that can be found in common OTRS modules. The class/package name is declared via the package keyword. Notice that BaseText is used as base class.

```
=item Foo()

Testing function: returns 1 if function is available on a Dynamic Field driver.

    my $Success = $BackendObject->Foo(
        DynamicFieldConfig   => $DynamicFieldConfig,      # complete config of the
 DynamicField
    );

Returns:
    $Success = 1;       # or undef

=cut

sub Foo {
    my ( $Self, %Param ) = @_;

    # check needed stuff
    for my $Needed (qw(DynamicFieldConfig)) {
        if ( !$Param{$Needed} ) {
            $Self->{LogObject}->Log( Priority => 'error', Message => "Need $Needed!" );
            return;
        }
    }

    # check DynamicFieldConfig (general)
    if ( !IsHashRefWithData( $Param{DynamicFieldConfig} ) ) {
        $Self->{LogObject}->Log(
            Priority => 'error',
            Message  => "The field configuration is invalid",
        );
        return;
    }

    # check DynamicFieldConfig (internally)
    for my $Needed (qw(ID FieldType ObjectType)) {
        if ( !$Param{DynamicFieldConfig}->{$Needed} ) {
            $Self->{LogObject}->Log(
```

```
                Priority => 'error',
                Message  => "Need $Needed in DynamicFieldConfig!"
            );
            return;
        }
    }

    # set the dynamic field specific backend
    my $DynamicFieldBackend = 'DynamicField' . $Param{DynamicFieldConfig}->{FieldType} .
 'Object';

    if ( !$Self->{$DynamicFieldBackend} ) {
        $Self->{LogObject}->Log(
            Priority => 'error',
            Message  => "Backend $Param{DynamicFieldConfig}->{FieldType} is invalid!"
        );
        return;
    }

    # verify if function is available
    return if !$Self->{$DynamicFieldBackend}->can('Foo');

    # call HasBehavior on the specific backend
    return $Self->{$DynamicFieldBackend}->Foo(%Param);
}
```

The function *Foo()* is only used for test purposes, first it checks the Dynamic Field configuration, then it checks if the Dynamic Field Driver (type) exists and was already loaded. To prevent the function call on a driver where is not defined it first check if the driver can execute the function, then executes the function in the driver passing all parameters.

Note

It is also possible to skip the step that test if the Driver can execute the function to do that is necessary to implement a mechanism in the Frontend module to require a special behavior on the Dynamic Field, and only after call the function in the Backend object.

```
1;

=back

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut

=cut
```

The normal OTRS footer for a Perl module file.

2.7.6.1.3. Dynamic Field Driver Extension Example

Driver extensions will be loaded transparently into the Driver itself as a base class, all defined object and properties from the Driver will be accessible in the extension.

Note

All new functions implemented in the Driver extension should be defined in a Backend extension, as every function is called from the Backend Object.

2.7.6.1.3.1. Code Example:

In this section the Foo extension for Text field driver is shown and explained. The extension only implements the function Foo().

```
# --
# Kernel/System/DynamicField/Driver/FooExtensionText.pm - Extension for DynamicField Text
 Driver
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::DynamicField::Driver::FooExtensionText;

use strict;
use warnings;
=head1 NAME

Kernel::System::DynamicField::Driver::FooExtensionText

=head1 SYNOPSIS

DynamicFields Text Driver Extension

=head1 PUBLIC INTERFACE

This module extends the public interface of L<Kernel::System::DynamicField::Backend>.
Please look there for a detailed reference of the functions.

=over 4

=cut
```

This is the common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

```
sub Foo {
    my ( $Self, %Param ) = @_;
    return 1;
}
```

The function *Foo()* has no special logic it is only for testing and it always returns 1.

```
1;

=back

=head1 TERMS AND CONDITIONS

This software is part of the OTRS project (L<http://otrs.org/>).

This software comes with ABSOLUTELY NO WARRANTY. For details, see
the enclosed file COPYING for license information (AGPL). If you
did not receive this file, see L<http://www.gnu.org/licenses/agpl.txt>.

=cut
```

The normal OTRS footer for a Perl module file.

2.8. Old Module Descriptions

Please remove these old sections if newer ones were created.

2.8.1. Navigation Module

In this module layer you can create dynamic navigation bar items with dynamic content (Name and Description). Navigation Module are located under Kernel/Output/HTML/NavBar*.pm.

Format:

```
# --
# Kernel/Output/HTML/NavBarABC.pm - shows a navbar item dynamicaly
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Output::HTML::NavBarABC;

use strict;
use warnings;

# --
sub new {
    my ( $Type, %Param ) = @_;
    [...]
    return $Self;
}
# --
sub Run {
    my ( $Self, %Param ) = @_;
    my %Return = ();
    $Return{'0999989'} = {
        Block       => 'ItemPersonal',
        Description => 'Some Desctipton',
        Name        => 'Text',
        Image       => 'new-message.png',
        Link        => 'Action=AgentMailbox&amp;Subaction=New',
        AccessKey   => 'j',
    };
    return %Return;
}
# --
1;
```

To use this module add the following code to the Kernel/Config.pm and restart your webserver (if you use mod_perl).

```
[Kernel/Config.pm]
# agent interface notification module
$Self->{'Frontend::NavBarModule'}->{'99-ABC'} = {
    Module => 'Kernel::Output::HTML::NavBarABC',
};
```

2.8.2. Customer Navigation Module

In this module layer you can create dynamic navigation bar items with dynamic content (Name and Description).

The format is the same as in the Navigation Module.

Just the config setting key is different. To use this module, add the following to the Kernel/Config.pm and restart your webserver (if you use mod_perl).

```
[Kernel/Config.pm]
# customer notifiacation module
$Self->{'CustomerFrontend::NavBarModule'}->{'99-ABC'} = {
    Module => 'Kernel::Output::HTML::NavBarABC',
};
```

2.8.3. Ticket PostMaster Module

PostMaster modules are used during the PostMaster process. There are two kinds of PostMaster modules. PostMasterPre (used after parsing an email) and PostMasterPost (used after an email is processed and in the database) modules.

If you want to create your own postmaster filter, just create your own module. These modules are located under "Kernel/System/PostMaster/Filter/*.pm". For default modules see the admin manual. You just need two functions: new() and Run():

The following is an examplary module to match emails and set X-OTRS-Headers (see doc/X-OTRS-Headers.txt for more info).

Kernel/Config.pm:

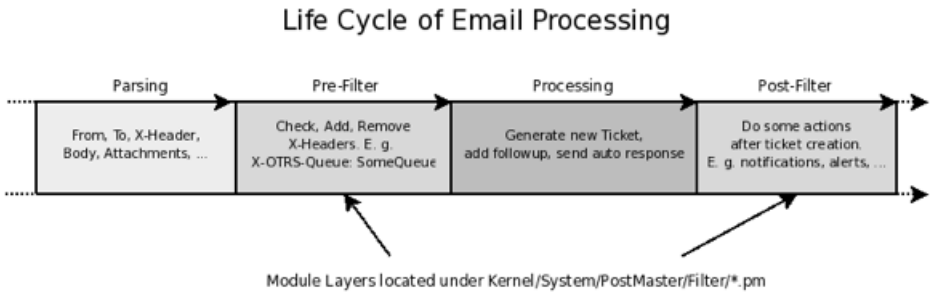
```
# Job Name: 1-Match
# (block/ignore all spam email with From: noreply@)
$Self->{'PostMaster::PreFilterModule'}->{'1-Example'} = {
    Module => 'Kernel::System::PostMaster::Filter::Example',
    Match => {
        From => 'noreply@',
    },
    Set => {
        'X-OTRS-Ignore' => 'yes',
    },
};
```

Format:

```
# --
# Kernel/System/PostMaster/Filter/Example.pm - a postmaster filter
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::System::PostMaster::Filter::Example;

use strict;
use warnings;

use vars qw(@ISA $VERSION);

sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
```

```
    bless ($Self, $Type);

    $Self->{Debug} = $Param{Debug} || 0;

    # get needed objects
    for (qw(ConfigObject EncodeObject LogObject DBObject)) {
        $Self->{$_} = $Param{$_} || die "Got no $_!";
    }

    return $Self;
}

sub Run {
    my ( $Self, %Param ) = @_;
    # get config options
    my %Config = ();
    my %Match = ();
    my %Set = ();
    if ($Param{JobConfig} &amp;&amp; ref($Param{JobConfig}) eq 'HASH') {
        %Config = %{$Param{JobConfig}};
        if ($Config{Match}) {
            %Match = %{$Config{Match}};
        }
        if ($Config{Set}) {
            %Set = %{$Config{Set}};
        }
    }
    # match 'Match => ???' stuff
    my $Matched = '';
    my $MatchedNot = 0;
    for (sort keys %Match) {
        if ($Param{GetParam}->{$_} &amp;&amp; $Param{GetParam}->{$_} =~ /$Match{$_}/i) {
            $Matched = $1 || '1';
            if ($Self->{Debug} > 1) {
                $Self->{LogObject}->Log(
                    Priority => 'debug',
                    Message => "'$Param{GetParam}->{$_}' =~ /$Match{$_}/i matched!",
                );
            }
        }
        else {
            $MatchedNot = 1;
            if ($Self->{Debug} > 1) {
                $Self->{LogObject}->Log(
                    Priority => 'debug',
                    Message => "'$Param{GetParam}->{$_}' =~ /$Match{$_}/i matched NOT!",
                );
            }
        }
    }
    # should I ignore the incoming mail?
    if ($Matched &amp;&amp; !$MatchedNot) {
        for (keys %Set) {
            if ($Set{$_} =~ /\[\*\*\*\]/i) {
                $Set{$_} = $Matched;
            }
            $Param{GetParam}->{$_} = $Set{$_};
            $Self->{LogObject}->Log(
                Priority => 'notice',
                Message => "Set param '$_' to '$Set{$_}' (Message-ID: $Param{GetParam}-
>{'Message-ID'}) ",
            );
        }
    }

    return 1;
}

1;
```

The following image shows you the email processing flow.



2.8.4. Ticket Menu Module

To add links in the ticket menu, just use ticket menu modules.

If you want to create your own ticket menu link, just create your own module. These modules are located under "Kernel/Output/HTML/TicketMenu*.pm". For default modules see the admin manual. You just need two functions: new() and Run():

The following example shows you how to show a lock or a unlock ticket link.

Kernel/Config.pm:

```
# for ticket zoom menu
$Self->{'Ticket::Frontend::MenuModule'}->{'100-Lock'} =  {
    Action => 'AgentTicketLock',
    Module => 'Kernel::Output::HTML::TicketMenuLock',
    Name   => 'Lock'
};

# for ticket preview menu
$Self->{'Ticket::Frontend::PreMenuModule'}->{'100-Lock'} =  {
    Action => 'AgentTicketLock',
    Module => 'Kernel::Output::HTML::TicketMenuLock',
    Name   => 'Lock'
};
```

Format:

```
# --
# Kernel/Output/HTML/TicketMenuLock.pm
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

package Kernel::Output::HTML::TicketMenuLock;

use strict;
use warnings;

use vars qw(@ISA $VERSION);

# --
sub new {
    my ( $Type, %Param ) = @_;

    # allocate new hash for object
    my $Self = {};
    bless ($Self, $Type);

    # get needed objects
    for (qw(ConfigObject LogObject DBObject LayoutObject UserID TicketObject)) {
```

```
        $Self->{$_} = $Param{$_} || die "Got no $_!";
    }

    return $Self;
}
# --
sub Run {
    my ( $Self, %Param ) = @_;
    # check needed stuff
    if (!$Param{Ticket}) {
        $Self->{LogObject}->Log(Priority => 'error', Message => 'Need Ticket!');
        return;
    }

    # check permission
    if ($Self->{TicketObject}->LockIsTicketLocked(TicketID => $Param{TicketID})) {
        my $AccessOk = $Self->{TicketObject}->OwnerCheck(
            TicketID => $Param{TicketID},
            OwnerID => $Self->{UserID},
        );
        if (!$AccessOk) {
            return $Param{Counter};
        }
    }

    $Self->{LayoutObject}->Block(
        Name => 'Menu',
        Data => { },
    );
    if ($Param{Counter}) {
        $Self->{LayoutObject}->Block(
            Name => 'MenuItemSplit',
            Data => {  },
        );
    }
    if ($Param{Ticket}->{Lock} eq 'lock') {
        $Self->{LayoutObject}->Block(
            Name => 'MenuItem',
            Data => {
                %{$Param{Config}},
                %{$Param{Ticket}},
                %Param,
                Name => 'Unlock',
                Description => 'Unlock to give it back to the queue!',
                Link => 'Action=AgentTicketLock&amp;Subaction=Unlock&amp;TicketID=
$QData{"TicketID"}',
            },
        );
    }
    else {
        $Self->{LayoutObject}->Block(
            Name => 'MenuItem',
            Data => {
                %{$Param{Config}},
                %Param,
                Name => 'Lock',
                Description => 'Lock it to work on it!',
                Link => 'Action=AgentTicketLock&amp;Subaction=Lock&amp;TicketID=
$QData{"TicketID"}',
            },
        );
    }
    $Param{Counter}++;
    return $Param{Counter};
}
# --
1;
```

Chapter 4. How to Publish Your OTRS Extensions

1. Package Management

The OPM (OTRS Package Manager) is a mechanism to distribute software packages for the OTRS framework via http, ftp or file upload.

For example, the OTRS project offers OTRS modules like a calendar, a file manager or web mail in OTRS packages via online repositories on our ftp servers. The packages can be managed (install/upgrade/uninstall) via the admin interface.

1.1. Package Distribution

If you want to create an OPM online repositiory, just tell the OTRS framework where the location is. Then you will have a new select option in the admin interface.

```
[Kernel/Config.pm]

# Package::RepositoryList
# (repository list)
$Self->{'Package::RepositoryList'} = {
    'ftp://ftp.example.com/packages/' => '[Example-Repository]',
};

[...]
```

1.1.1. Package Repository Index

In your repository, create an index file for your OPM packages. OTRS just reads this index file and knows what packages are available.

```
shell> bin/otrs.PackageManager.pl -a index -d /path/to/repository/ > /path/to/repository/
otrs.xml
shell>
```

1.2. Package Commands

You can use the following OPM commands over the admin interface or over bin/otrs.PackageManager.pl to manage admin jobs for OPM packages.

1.2.1. Install

Install OPM packages.

- Web: http://localhost/otrs/index.pl?Action=AdminPackageManager
- CMD:

```
shell> bin/otrs.PackageManager.pl -a install -p /path/to/package.opm
```

1.2.2. Uninstall

Uninstall OPM packages.

- Web: http://localhost/otrs/index.pl?Action=AdminPackageManager
- CMD:

```
shell> bin/otrs.PackageManager.pl -a uninstall -p /path/to/package.opm
```

1.2.3. Upgrade

Upgrade OPM packages.

- Web: http://localhost/otrs/index.pl?Action=AdminPackageManager
- CMD:

```
shell> bin/otrs.PackageManager.pl -a upgrade -p /path/to/package.opm
```

1.2.4. List

List all OPM packages.

- Web: http://localhost/otrs/index.pl?Action=AdminPackageManager
- CMD:

```
shell> bin/otrs.PackageManager.pl -a list
```

2. Package Building

If you want to create an OPM package (.opm) you need to create a spec file (.sopm) which includes the properties of the package.

2.1. Package Spec File

The OPM package is XML based. You can create/edit the .sopm via a text or xml editor. It contains meta data, a file list and database options.

2.1.1. Name

The package name (required).

```
<Name>Calendar</Name>
```

2.1.2. Version

The package version (required).

```
<Version>1.2.3</Version>
```

2.1.3. Framework

The targeted framework version (3.2.x means e.g. 3.2.1 or 3.2.2) (required).

```
<Framework>3.2.x</Framework>
```

Can also be used several times.

```
<Framework>3.0.x</Framework>
<Framework>3.1.x</Framework>
<Framework>3.2.x</Framework>
```

2.1.4. Vendor

The package vendor (required).

```
<Vendor>OTRS AG</Vendor>
```

2.1.5. URL

The vendor URL (required).

```
<URL>http://otrs.org/</URL>
```

2.1.6. License

The license of the package (required).

```
<License>GNU AFFERO GENERAL PUBLIC LICENSE Version 3, November 2007</License>
```

2.1.7. ChangeLog

The package change log (optional).

```
<ChangeLog Version="1.1.2" Date="2013-02-15 18:45:21">Added some feature.</ChangeLog>
<ChangeLog Version="1.1.1" Date="2013-02-15 16:17:51">New package.</ChangeLog>
```

2.1.8. Description

The package description in different languages (required).

```
<Description Lang="en">A web calendar.</Description>
<Description Lang="de">Ein Web Kalender.</Description>
```

2.1.9. BuildHost

This will be filled in automatically by OPM (auto).

```
<BuildHost>?</BuildHost>
```

2.1.10. BuildDate

This will be filled in automatically by OPM (auto).

```
<BuildDate>?</BuildDate>
```

2.1.11. PackageRequired

Packages that must be installed beforehand (optional). If PackageRequired is used, a version of the required package must be specified.

```
<PackageRequired Version="1.0.3">SomeOtherPackage</PackageRequired>
<PackageRequired Version="5.3.2">SomeotherPackage2</PackageRequired>
```

2.1.12. ModuleRequired

Perl modules that must be installed beforehand (optional).

```
<ModuleRequired Version="1.03">Encode</ModuleRequired>
<ModuleRequired Version="5.32">MIME::Tools</ModuleRequired>
```

2.1.13. OS (^M)

Required OS (optional).

```
<OS>linux</OS>
<OS>darwin</OS>
<OS>mswin32</OS>
```

2.1.14. Filelist

This is a list of files included in the package (optional).

```
<Filelist>
    <File Permission="644" Location="Kernel/Config/Files/Calendar.pm"/>
    <File Permission="644" Location="Kernel/System/CalendarEvent.pm"/>
    <File Permission="644" Location="Kernel/Modules/AgentCalendar.pm"/>
    <File Permission="644" Location="Kernel/Language/de_AgentCalendar.pm"/>
</Filelist>
```

2.1.15. DatabaseInstall

Database entries that have to be created when a package is installed (optional).

```
<DatabaseInstall>
    <TableCreate Name="calendar_event">
    <Column Name="id" Required="true" PrimaryKey="true" AutoIncrement="true" Type="BIGINT"/>
    <Column Name="title" Required="true" Size="250" Type="VARCHAR"/>
    <Column Name="content" Required="false" Size="250" Type="VARCHAR"/>
    <Column Name="start_time" Required="true" Type="DATE"/>
    <Column Name="end_time" Required="true" Type="DATE"/>
    <Column Name="owner_id" Required="true" Type="INTEGER"/>
    <Column Name="event_status" Required="true" Size="50" Type="VARCHAR"/>
    </TableCreate>
</DatabaseInstall>
```

You also can choose <DatabaseInstall Type="post"> or <DatabaseInstall Type="pre"> to define the time of execution separately (post is default). For more info see chapter "Package Life Cycle".

2.1.16. DatabaseUpgrade

Information on which actions have to be performed in case of an upgrade (subject to version tag), (optional). Example (if already installed package version is below 1.3.4 (e. g. 1.2.6), defined action will be performed):

```
<DatabaseUpgrade>
    <TableCreate Name="calendar_event_involved" Version="1.3.4">
        <Column Name="event_id" Required="true" Type="BIGINT"/>
        <Column Name="user_id" Required="true" Type="INTEGER"/>
    </TableCreate>
</DatabaseUpgrade>
```

You also can choose <DatabaseUpgrade Type="post"> or <DatabaseUpgrade Type="pre"> to define the time of execution separately (post is default). For more info see chapter "Package Life Cycle".

2.1.17. DatabaseReinstall

Information on what actions have to be performed if the package is reinstalled, (optional).

```
<DatabaseReinstall></DatabaseReinstall>
```

You also can choose <DatabaseReinstall Type="post"> or <DatabaseReinstall Type="pre"> to define the time of execution separately (post is default). For more info see chapter "Package Life Cycle".

2.1.18. DatabaseUninstall

Uninstall (if a package gets uninstalled), (optional).

```
<DatabaseUninstall>
    <TableDrop Name="calendar_event" />
</DatabaseUninstall>
```

You also can choose <DatabaseUninstall Type="post"> or <DatabaseUninstall Type="pre"> to define the time of execution separately (post is default). For more info see chapter "Package Life Cycle".

2.1.19. IntroInstall

To show a "pre" or "post" install introduction in installation dialog.

```
<IntroInstall Type="post" Lang="en" Title="Some Title"><![CDATA[
Some Info formatted in dtl/html....
]]></IntroInstall>
```

You can also use the "Format" attribute to define if you want to use "html" (which is default) or "plain" to use automatically a "<pre></pre>" tag when intro is shown (to use the new lines and spaces of the content).

2.1.20. IntroUninstall

To show a "pre" or "post" uninstall introduction in uninstallation dialog.

```
```

```
<IntroUninstall Type="post" Lang="en" Title="Some Title"><![CDATA[
Some Info formatted in dtl/html....
]]></IntroUninstall>
```

You can also use the "Format" attribute to define if you want to use "html" (which is default) or "plain" to use automatically a "<pre></pre>" tag when intro is shown (to use the new lines and spaces of the content).

2.1.21. IntroReinstall

To show a "pre" or "post" reinstall introduction in reinstallation dialog.

```
<IntroReinstall Type="post" Lang="en" Title="Some Title"><![CDATA[
Some Info formatted in dtl/html....
]]></IntroReinstall>
```

You can also use the "Format" attribute to define if you want to use "html" (which is default) or "plain" to use automatically a "<pre></pre>" tag when intro is shown (to use the new lines and spaces of the content).

2.1.22. IntroUpgrade

To show a "pre" or "post" upgrade introduction in upgrading dialog.

```
<IntroUpgrade Type="post" Lang="en" Title="Some Title"><![CDATA[
Some Info formatted in dtl/html....
]]></IntroUpgrade>
```

You can also use the "Format" attribute to define if you want to use "html" (which is default) or "plain" to use automatically a "<pre></pre>" tag when intro is shown (to use the new lines and spaces of the content).

2.1.23. CodeInstall

To execute perl code when the package is installed (optional).

```
<CodeInstall><![CDATA[
    # example
    if (1) {
        print STDERR "Some info to STDERR\n";
    }
    # log example
    $Self->{LogObject}->Log(
        Priority => 'notice',
        Message => "Some Message!",
    );
    # database example
    $Self->{DBObject}->Do(SQL => "SOME SQL");
]]></CodeInstall>
```

You also can choose <CodeInstall Type="post"> or <CodeInstall Type="pre"> to define the time of execution separately (post is default). For more info see chapter "Package Life Cycle".

2.1.24. CodeUninstall

To execute perl code if the package is uninstalled (optional). On "pre" or "post" time of package uninstallation.

```
<CodeUninstall><![CDATA[
    # example
    if (1) {
        print STDERR "Some info to STDERR\n";
    }
]]></CodeUninstall>
```

You also can choose <CodeUninstall Type="post"> or <CodeUninstall Type="pre"> to define the time of execution separately (post is default). For more info see chapter "Package Life Cycle".

2.1.25. CodeReinstall

To execute perl code if the package is reinstalled (optional).

```
<CodeReinstall><![CDATA[
    # example
    if (1) {
        print STDERR "Some info to STDERR\n";
    }
]]></CodeReinstall>
```

You also can choose <CodeReinstall Type="post"> or <CodeReinstall Type="pre"> to define the time of execution separately (post is default). For more info see chapter "Package Life Cycle".

2.1.26. CodeUpgrade

To execute perl code if the package is upgraded (subject to version tag), (optional). Example (if already installed package version is below 1.3.4 (e. g. 1.2.6), defined action will be performed):

```
<CodeUpgrade Version="1.3.4"><![CDATA[
    # example
    if (1) {
        print STDERR "Some info to STDERR\n";
    }
]]></CodeUpgrade>
```

You also can choose <CodeUpgrade Type="post"> or <CodeUpgrade Type="pre"gt; to define the time of execution separately (post is default).

2.2. Example .sopm

This is a whole example spec file.

```
<?xml version="1.0" encoding="utf-8" ?>
<otrs_package version="1.0">
    <Name>Calendar</Name>
    <Version>0.0.1</Version>
    <Framework>3.2.x</Framework>
    <Vendor>OTRS AG</Vendor>
    <URL>http://otrs.org/</URL>
    <License>GNU AFFERO GENERAL PUBLIC LICENSE Version 3, November 2007</License>
    <ChangeLog Version="1.1.2" Date="2013-02-15 18:45:21">Added some feature.</ChangeLog>
    <ChangeLog Version="1.1.1" Date="2013-02-15 16:17:51">New package.</ChangeLog>
    <Description Lang="en">A web calendar.</Description>
    <Description Lang="de">Ein Web Kalender.</Description>
    <IntroInstall Type="post" Lang="en" Title="Thank you!">Thank you for choosing the
 Calendar module.</IntroInstall>
```

```
    <IntroInstall Type="post" Lang="de" Title="Vielen Dank!">Vielen Dank fuer die Auswahl
 des Kalender Modules.</IntroInstall>
    <BuildDate>?</BuildDate>
    <BuildHost>?</BuildHost>
    <Filelist>
        <File Permission="644" Location="Kernel/Config/Files/Calendar.pm"></File>
        <File Permission="644" Location="Kernel/System/CalendarEvent.pm"></File>
        <File Permission="644" Location="Kernel/Modules/AgentCalendar.pm"></File>
        <File Permission="644" Location="Kernel/Language/de_AgentCalendar.pm"></File>
        <File Permission="644" Location="Kernel/Output/HTML/Standard/AgentCalendar.dtl"></
File>
        <File Permission="644" Location="Kernel/Output/HTML/NotificationCalendar.pm"></File>
        <File Permission="644" Location="var/httpd/htdocs/images/Standard/calendar.png"></
File>
    </Filelist>
    <DatabaseInstall>
        <TableCreate Name="calendar_event">
            <Column Name="id" Required="true" PrimaryKey="true" AutoIncrement="true"
 Type="BIGINT"/>
            <Column Name="title" Required="true" Size="250" Type="VARCHAR"/>
            <Column Name="content" Required="false" Size="250" Type="VARCHAR"/>
            <Column Name="start_time" Required="true" Type="DATE"/>
            <Column Name="end_time" Required="true" Type="DATE"/>
            <Column Name="owner_id" Required="true" Type="INTEGER"/>
            <Column Name="event_status" Required="true" Size="50" Type="VARCHAR"/>
        </TableCreate>
    </DatabaseInstall>
    <DatabaseUninstall>
        <TableDrop Name="calendar_event"/>
    </DatabaseUninstall>
</otrs_package>
```

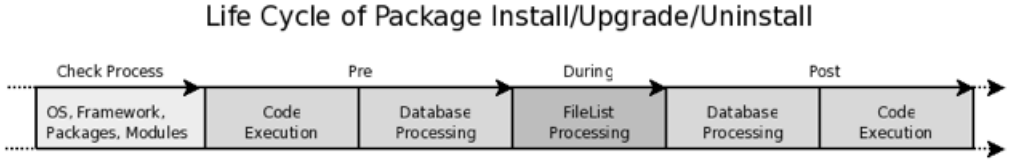
2.3. Package Build

To build an .opm package from the spec opm.

```
shell> bin/otrs.PackageManager.pl -a build -p /path/to/example.sopm
writing /tmp/example-0.0.1.opm
shell>
```

2.4. Package Life Cycle - Install/Upgrade/Uninstall

The following image shows you how the life cycle of a package instalation/upgrade/uninstallation works in the backend step by step.



Chapter 5. Contributing to OTRS

This chapter will show how you can contribute to the OTRS framework, so that other users will be able to benefit from your work.

1. Sending Contributions

The source code of OTRS and additional public modules can be found on github. From there you can get to the listing of all available repositories. It also describes the currently active branches and where contributions should go to (stable vs. development branches).

The easiest way to send in your contributions to the OTRS developer's team is by creating a "pull request" in github. Please take a look at the instructions on github, specifically about forking a repository and sending pull requests.

The basic workflow would look like this:

- Register at github, if you have no account yet.
- Fork the repository you want to contribute to, and checkout the branch that the changes should go in.
- Create a new development branch for your fix/feature/contribution, based off the current branch.
- After you finished your changes and committed them, push your branch to github.
- Create a pull request. The OTRS dev team will be notified about this, check your pull request and either merge it or give you some feedback about possible improvements.

It might sound complicated, but once you have this workflow set up you'll see that making contributions is extremely easy.

2. Translating OTRS

The OTRS framework allows for different languages to be used in the frontend.

2.1. How it works

There are three different translation file types which are used in the following order. If a word/sentence is redefined in a translation file, the latest definition will be used.

1. Default Framework Translation File

   `Kernel/Language/$Language.pm`

2. Custom Translation File

   `Kernel/Language/$Language_Custom.pm`

2.1.1. Default Framework Translation File

The Default Framework Translation File includes the basic translations. The following is an example of a Default Framework Translation File.

Format:

```
package Kernel::Language::de;

use strict;
use warnings;

use vars qw(@ISA $VERSION);

sub Data {
    my $Self = shift;

    # $$START$$

    # possible charsets
    $Self->{Charset} = ['iso-8859-1', 'iso-8859-15', ];
    # date formats (%A=WeekDay;%B=LongMonth;%T=Time;%D=Day;%M=Month;%Y=Jear;)
    $Self->{DateFormat} = '%D.%M.%Y %T';
    $Self->{DateFormatLong} = '%A %D %B %T %Y';
    $Self->{DateFormatShort} = '%D.%M.%Y';
    $Self->{DateInputFormat} = '%D.%M.%Y';
    $Self->{DateInputFormatLong} = '%D.%M.%Y - %T';

    $Self->{Translation} = {
    # Template: AAABase
    'Yes' => 'Ja',
    'No' => 'Nein',
    'yes' => 'ja',
    'no' => 'kein',
    'Off' => 'Aus',
    'off' => 'aus',
    };
    # $$STOP$$
    return 1;
}

1;
```

2.1.2. Custom Translation File

The Custom Translation File is read out last and so its translation which will be used. If you want to add your own wording to your installation, create this file for your language.

Format:

```
package Kernel::Language::xx_Custom;

use strict;
use warnings;

use vars qw(@ISA $VERSION);

sub Data {
    my $Self = shift;

    # $$START$$

    # own translations
    $Self->{Translation}->{'Lock'} = 'Lala';
    $Self->{Translation}->{'Unlock'} = 'Lulu';

    # $$STOP$$
    return 1;
}

1;
```

2.2. Updating an existing translation

Updating an existing translation is easy:

1. To coordinate translation efforts, avoiding duplicate translation efforts, and to send in your translated files, please use the i18n mailing list of OTRS.
2. Take the current translation file (Kernel/Language/$Language.pm) from git (github) and save it in your `Kernel/Language/` directory, overwriting the current one in your filesystem.
3. Now you can update the file.
4. When you are finished and satisfied with the tranlsation, please send it to the i18n mailing list.

2.3. Adding a new frontend translation

If you want to translate the OTRS framework into a new language, you have to follow these steps:

1. To coordinate translation efforts, avoiding duplicate translation efforts, and to send in your translated files, please use the i18n mailing list of OTRS.
2. Take the current German translation (Kernel/Language/de.pm) from git (github). Use the German version because this is always up to date.
3. Change the package name (e.g. "package Kernel::Language::de;" to "package Kernel::Language::fr;") and translate each word/sentence.
4. Add the new language translation to the framework by adding it to your `Kernel/Config.pm`.

```
$Self->{DefaultUsedLanguages}->{fr} = 'French';
```

5. If you use mod_perl, restart your webserver and the new language will be shown in your preferences selection.
6. When you are finished and satisfied with the tranlsation, please send it to the i18n mailing list.

3. Translating the Documentation

The OTRS admin manual can be translated. The workflow is like this:

1. The original documentation is written in English, in docbook XML format.
2. The tool po4a extracts gettext translation files from that (.po). These files can be edited with existing helpful tools like Poedit.
3. These files can be translated and will be updated on changes in the English source files without loosing existing translations.
4. From the translation files, translated docbook XML filed can be generated for the different languages (again with po4a).

This process has the benefit of keeping track of updates in the original source files. Only additions and updates have to be retranslated.

The git repository for the admin documentation can be found on github. Inside of this repository, there is the directory with the translation files.

To start a new translation, download the translation template file doc-admin.pot from git, save it and rename the file to `doc-admin.$Lanugage.po` (e. g. `doc-admin.de.po` for German).

To improve an existing translation, you can download the `doc-admin.$Lanugage.po` file for this lanugage and work on this. If there are questions on how the translation should be made, you can use `doc-admin.de.po` as a reference.

It is important that the structure of the generated XML stays intact. So if the original string is "Edit <filename>Kernel/Config.pm</filename>", then the German translation has to be "<filename>Kernel/Config.pm</filename> bearbeiten", keeping the the XML tags. Scripts and examples usually do not have to be translated (so you can just copy the source text to the translation text field in this case).

To coordinate translation efforts and to send in your translated .po files, please use the i18n mailing list of OTRS. The OTRS team will take care of regenerating the XML documenation for your language.

Important

Especially if you add a new translation, it is recommended to send in snapshots of your file regularly to make sure that your translation has the correct structure and that the docbook documentation can be successfully generated from it, producing valid XML documents.

If you want to test your translation file locally, you can checkout the doc-admin repository and use po4a directly to generate the translation files. Please see the file `po4a.conf` in the top directory of that module for usage hints. For additional questions, please use the i18n mailing list.

4. Code Style Guide

In order to preserve the consistent development of the OTRS project, we have set up guidelines regarding style for the different programming languages.

4.1. Perl

4.1.1. Formatting

4.1.1.1. Whitespace

TAB: We use 4 spaces. Examples for braces:

```
if ($Condition) {
    Foo();
}
else {
    Bar();
}

while ($Condition == 1) {
    Foo();
}
```

4.1.1.2. Length of lines

Lines should not be longer than 100 characters.

4.1.1.3. Spaces and parentheses

To gain more readability, we use spaces between keywords and opening parenthesis.

```
if ()...
for ()...
```

If there is just one single variable, the parenthesis enclose the variable with no spaces inside.

```
if ($Condition) { ... }

# instead of

if ( $Condition ) { ... }
```

If the condition is not just one single variable, we use spaces between the parenthesis and the condition. And there is still the space between the keyword (e.g. if) and the opening parens.

```
if ( $Condition && $ABC ) { ... }
```

Note that for Perl builtin functions, we do not use parentheses:

```
chomp $Variable;
```

4.1.1.4. Source code header and charset

Attach the following header to every source file. Source files are saved in UTF-8 charset.

```
# --
# (file name) - a short description what it does
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --
```

Executable files (*.pl) have a special header.

```
#!/usr/bin/perl
# --
# (file name) - a short description what it does
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU AFFERO General Public License as published by
# the Free Software Foundation; either version 3 of the License, or
# any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU Affero General Public License
# along with this program; if not, write to the Free Software
```

```
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
# or see http://www.gnu.org/licenses/agpl.txt.
# --
```

4.1.2. Using the Perl language

4.1.2.1. Control flow

4.1.2.1.1. Conditions

Conditions can be quite complex and there can be "chained" conditions (linked with logical 'or' or 'and' operations). When coding for OTRS, you have to be aware of several situations.

Perl Best Practices says, that high precedence operators ('&&' and '||') shouldn't mixed up with low precedence operators ('and' and 'or'). To avoid confusion, we always use the high precedence operators.

```
if ( $Condition1 && $Condition2 ) { ... }

# instead of

if ( $Condition and $Condition2 ) { ... }
```

This means that you have to be aware of traps. Sometimes you need to use parenthesis to make clear what you want.

If you have long conditions (line is longer than 120 characters over all), you have to break it in several lines. And the start of the conditions is in a new line (not in the line of the 'if').

```
if (
    $Condition1
    && $Condition2
    )
{ ... }

# instead of

if ( $Condition1
    && $Condition2
    )
{ ... }
```

Also note, that the right parenthesis is in a line on its own and the left curly bracket is also in a new line an with the same indentation as the 'if'. The operators are at the beginning of a new line! The subsequent examples show how to do it...

```
if (
    $XMLHash[0]->{otrs_stats}[1]{StatType}[1]{Content}
    && $XMLHash[0]->{otrs_stats}[1]{StatType}[1]{Content} eq 'static'
    )
{ ... }

if ( $TemplateName eq 'AgentTicketCustomer' ) { ... }

if (
    ( $Param{Section} eq 'Xaxis' || $Param{Section} eq 'All' )
    && $StatData{StatType} eq 'dynamic'
    )
{ ... }

if (
```

```
    $Self->{TimeObject}->TimeStamp2SystemTime( String => $Cell->{TimeStop} )
    > $Self->{TimeObject}->TimeStamp2SystemTime(
        String => $ValueSeries{$Row}{$TimeStop}
    )
    || $Self->{TimeObject}->TimeStamp2SystemTime( String => $Cell->{TimeStart} )
    < $Self->{TimeObject}->TimeStamp2SystemTime(
        String => $ValueSeries{$Row}{$TimeStart}
    )
    )
{ ... }
```

4.1.2.1.2. Postfix "if"

Generally we use "postfix if" statements to reduce the number of levels. But we don't use it for multiline statments and is only allowed when involves return statements in functions or to end a loop or to go next iteration.

Ok:

```
next ITEM if !$ItemId;
```

Not ok:

```
return $Self->{LogObject}->Log(
    Priority => 'error',
    Message  => 'ItemID needed!',
) if !$ItemId;
```

This is less maintainable than

```
if( !$ItemId ) {
    $Self->{LogObject}->Log( ... );
    return;
}
```

Ok:

```
for my $Needed (1..10) {
    next if $Needed == 5;
    last  if $Needed == 9;
}
```

Not ok:

```
my $Var = 1 if $Something == 'Yes';
```

4.1.2.2. Restrictions for the use of some Perl builtins

Some builtin subroutines of Perl may not be used in every place:

- Don't use "die" and "exit" in .pm-files.
- Don't use the "Dumper" function in released files.

- Don't use "print" in .pm files.
- Don't use "require", use Main::Require() instead.
- Use the functions of the TimeObject instead of the builtin functions like time(), local-time() etc.

4.1.2.3. Regular Expressions

For regular expressions we always use the m// operator with curly braces as delimiters. We also use the modifiers x, m and s. The x modifiers allows you to comment your regex and use spaces to "group" logical groups.

```
$Date =~ m{ \A \d{4} - \d{2} - \d{2} \z }xms
$Date =~ m{
    \A    # beginning of the string
    \d{4} - # year
    \d{2} - # month
    [^\n]   # everything but newline
    #..
}xms;
```

As the space no longer has a special meaning, you have to use a single character class to match a single space ([ ]). If you want to match any whitespace you can use \s.

In the regex, the dot ('.') includes the newline (whereas in regex without s modifier the dot means 'everything but newline'). If you want to match anything but newline, you have to use the negated single character class ([^\n]).

```
$Text =~ m{
    Test
    [ ]    # there must be a space between 'Test' and 'Regex'
    Regex
}xms;
```

4.1.2.4. Naming

Names and comments are written in English. Variables, objects and methods must be descriptive nouns or noun phrases with the first letter set upper case (CamelCase).

Names should be as descriptive as possible. A reader should be able to say what is meant by a name without digging too deep into the code. E.g. use $ConfigItemID instead of $ID.

```
e. g. @TicktIDs or $Output or StateSet()
```

4.1.2.5. Variables

4.1.2.5.1. Declaration

If you have several variables, you can declare them in on line if the "belong together":

```
my ($Minute, $Hour, $Year);
```

and break it into seperate lines otherwise:

```
my $Minute;
my $ID;
```

Do not set to "undef" or '' in the declaration as this might hide mistakes in code.

```
my $Variable = undef;

#is the same as

my $Variable;
```

You can set a variable to '' if you want to concatenate strings:

```
my $SqlStatement = '';
for my $Part ( @Parts ) {
    $SqlStatement .= $Part;
}
```

Otherwise you would get an "uninitialized" warning.

4.1.2.6. Subroutines

4.1.2.6.1. Handling of parameters

To fetch the parameters passed to subroutines, OTRS normally uses the hash %Param (not %Params). This leads to more readable code as every time we use %Param in the subroutine code we know it is the parameter hash passed to the subroutine.

Just in some exceptions a regular list of parameters should be used. So we want to avoid something like this:

```
sub TestSub {
    my ( $Self, $Param1, $Param2 ) = @_;
}
```

We want to use this instead:

```
sub TestSub {
    my ( $Self, %Param ) = @_;
}
```

This has several advantages: We do not have to change the code in the subroutine when a new parameter should be passed, and calling a function with named parameters is much more readable.

4.1.2.6.2. Multiple named parameters

If a function call requires more than one named parameter, split them into multiple lines:

```
    $Self->{LogObject}->Log(
        Priority => 'error',
        Message  => "Need $Needed!",
    );
```

```
```

instead of:

```
    $Self->{LogObject}->Log( Priority => 'error', Message  => "Need $Needed!", );
```

4.1.2.6.3. return statements

Subroutines have to have a return statement. The explicit return statement is preferred over the implicit way (result of last statement in subroutine) as this clarifies what the subroutine returns.

```
sub TestSub {
    ...
    return; # return undef, but not the result of the last statement
}
```

4.1.2.6.4. Explicit return values

Explicit return values means that you should not have a return statement followed by a subroutine call.

```
return $Self->{DBObject}->Do( ... );
```

The following example is better as this says explicitly what is returned. With the example above the reader doesn't know what the return value is as he might not know what Do() returns.

```
return if !$Self->{DBObject}->Do( ... );
return 1;
```

If you assign the result of a subroutine to a variable, a "good" variable name indicates what was returned:

```
my $SuccessfulInsert = $Self->{DBObject}->Do( ... );
return $SuccessfulInsert;
```

4.1.2.7. Packages

4.1.2.7.1. "use" statements

use strict and use warnings have to be the first two "use"s in a module. This is correct:

```
package Kernel::System::ITSMConfigItem::History;

use strict;
use warnings;

use Kernel::System::User;
use Kernel::System::Time;
```

This is wrong:

```
package Kernel::System::ITSMConfigItem::History;

use Kernel::System::User;
use Kernel::System::Time;

use strict;
use warnings;
```

4.1.2.7.2. Passing of required objects

In core modules we use $Self to create instances of other classes, because it is more safe. For example:

```
$Self->{TimeObject} = Kernel::System::Time->new( %{$Self} );
```

In frontend modules we use the %Param hash to create the new instances as there are more variables and other information stored in the hash:

```
$Self->{TimeObject} = Kernel::System::Time->new( %Param );
```

4.1.2.7.3. Objects and their allocation

In OTRS many objects are available. But you should not use every object in every file to keep the frontend/backend separation.

- Don't use the LayoutObject in core modules (Kernel/System).
- Don't use the ParamObject in core modules (Kernel/System).
- Don't use the DBObject in frontend modules (Kernel/Modules).

4.1.3. Writing good documentation

4.1.3.1. Perldoc

4.1.3.1.1. Documenting subroutines

Subroutines should always be documented. The documentation contains a general description about what the subroutine does, a sample subroutine call and what the subroutine returns. It should be in this order. A sample documentation looks like this:

```
    =item LastTimeObjectChanged()

    calculates the last time the object was changed. It returns a hash reference with
        information about the object and the time.

        my $Info = $Object->LastTimeObjectChanged(
            Param => 'Value',
        );

    This returns something like:

        my $Info = {
            ConfigItemID    => 1234,
            HistoryType     => 'foo',
            LastTimeChanged => '08.10.2009',
        };
```

```
    =cut
```

You can copy and paste a Data::Dumper output for the return values.

4.1.3.2. Commenting

In general, you should try to write your code as readable and self-explaining as possible. Don't write a comment to explain what obvious code does, this is unneccessary duplication. Good comments should explain WHY there is some code, possible side effects and anything that might be special or unusually complicated about the code.

4.1.3.2.1. Special comments

There are two special kinds of comments.

Example 1 - comment for a longer block of code

```
# ---------------------------------#
# here is the start of a special area
# ---------------------------------#
```

Example 2 - comment for customizing standard OTRS files

```
# --- customizing for some project
```

4.1.4. Database interaction

4.1.4.1. Declaration of SQL statements

If there is no chance for changing the SQL statement, it should be used in the Prepare function. The reason for this is, that the SQL statement and the bind parameters are closer to each other.

The SQL statement should be written as one nicely indented string without concatenation like this:

```
return if !$Self->{DBObject}->Prepare(
    SQL => '
        SELECT art.id
        FROM article art, article_sender_type ast
        WHERE art.ticket_id = ?
            AND art.article_sender_type_id = ast.id
            AND ast.name = ?
        ORDER BY art.id',
    Bind => [ \$Param{TicketID}, \$Param{SenderType} ],
);
```

This is easy to read and modify, and the whitespace can be handled well by our supported DBMSs. For auto-generated SQL code (like in TicketSearch), this indentation is not neccessary.

4.1.4.2. Returning on errors

Whenever you use database functions you should handle errors. If anything goes wrong, return from subroutine:

```
return if !$Self->{DBObject}->Prepare( ... );
```

4.1.4.3. using Limit

Use Limit => 1 if you expect just one row to be returned.

```
$Self->{DBObject}->Prepare(
    SQL   => 'SELECT id FROM users WHERE username = ?',
    Bind  => [ \$Username ],
    Limit => 1,
);
```

4.1.4.4. Using the while loop

Always use the while loop, even when you expect one row to be returned, as some databases do not release the statement handle and this can lead to weird bugs.

4.2. JavaScript

4.2.1. Browser Handling

All JavaScript is loaded in all browsers (no browser hacks in the .dtl files). The code is responsible to decide if it has to skip or execute certain parts of itself only in certain browsers.

4.2.2. Directory Structure

Directory structure inside the `js/` folder:

```
* js
    * thirdparty              # thirdparty libs always have the version number inside the
 directory
        * ckeditor-3.0.1
        * jquery-1.3.2
    * Core.Agent.*            # stuff specific to the agent interface
    * Core.Customer.*         # customer interface
    * Core.*                  # common API
```

4.2.2.1. Thirdparty Code

Every thirdparty module gets its own subdirectory: "module name"-"version number" (e.g. ckeditor-3.0.1, jquery-1.3.2). Inside of that, file names should not have a version number or postfix included (wrong: `jquery/jquery-1.4.3.min.js`, right: `jquery-1.4.3/jquery.js`).

4.2.3. Variables

- Variable names should be CamelCase, just like in Perl.
- Variables that hold a jQuery object should start with `$`, for example: `$Tooltip`.

4.2.4. Functions

- Function names should be CamelCase, just like in Perl.

4.2.5. Namespaces

- TODO...

4.2.6. Comments

- Single line comments are done with `//`.
- Longer comments are done with `/* ... */`
- If you comment out parts of your JavaScript code, only use `//` because `/* .. */` can cause problems with Regular Expressions in the code.

4.2.7. Event Handling

- Always use `$.bind()` instead of the event-shorthand methods of jQuery for better readability (wrong: `$SomeObject.click(...)`, right: `$SomeObject.bind('click', ...)`).
- Do not use `$.live()`! We had severe performance issues with `$.live()` in correlation with mouse events. Until it can be verified that `$.live()` works with other event types without problems.
- If you `$.bind()` events, make sure to `$.unbind()` them beforehand, to make sure that events will not be bound twice, should the code be executed another time.

4.3. CSS

- Minimum resolution is 1024x768px.
- The layout is liquid, which means that if the screen is wider, the space will be used.
- Absolute size measurements should be specified in px to have a consistent look on many platforms and browsers.
- Documentation is made with CSSDOC (see CSS files for examples). All logical blocks should have a CSSDOC comment.

4.3.1. Architecture

- We follow the Object Oriented CSS approach. In essence, this means that the layout is achieved by combining different generic building blocks to realize a particular design.
- Wherever possible, module specific design should not be used. Therefore we also do not work with IDs on the body element, for example, if it can be avoided.

4.3.2. Style

- All definitions have a { in the same line as the selector, all rules are defined in one row per rule, the definition ends with a row with a single } in it. See the following example:

```
#Selector {
    width: 10px;
    height: 20px;
    padding: 4px;
}
```

- Between : and the rule value, there is a space
- Every rule has an indent of 4 spaces.
- If multiple selectors are specified, separate them with comma and put each one on an own line:

```
#Selector1,
#Selector2,
#Selector3 {
    width: 10px;
}
```

- If rules are combinable, combine them (e.g. combine `background-position`, `background-image`, ... into `background`).
- Rules should be in a logical order within a definition (all color specific rule together, all positioning rules together, ...).
- All IDs and Names are written in CamelCase notation:

```
<div class="NavigationBar" id="AdminMenu"></div>
```

5. User Interface Design

5.1. Capitalization

This section talks about how the different parts of the English user interface should be capitalized. For further information, you may want to review this helpful page.

- Headings (h1-h6) and Titles (Names, such as `Queue View`) are set in "title style" capitalization, that means all first letters will be capitalized (with a few exceptions such as "this", "and", "or" etc.).

  Examples: `Action List`, `Manage Customer-Group Relations`.

- Other structural elements such as buttons, labels, tabs, menu items are set in "sentence style" capitalization (only the first letter of a phrase is capitalized), but no final dot is added to complete the phrase as a sentence.

  Examples: `First name`, `Select queue refresh time`, `Print this ticket`.

- Descriptive texts and tooltip contents are written as complete sentences.

  Example: `This value is required.`

- For translations, it has to be checked if the title style capitalization is also appropriate in the target language, it might have to be changed to sentence style capitalization or something else.

6. Accessibility Guide

This document is supposed to explain basics about accessibility issues and give guidelines for contributions to OTRS.

6.1. Accessibility Basics

6.1.1. What is Accessibility?

Accessibility is a general term used to describe the degree to which a product, device, service, or environment is accessible by as many people as possible. Accessibility can be

viewed as the "ability to access" and possible benefit of some system or entity. Accessibility is often used to focus on people with disabilities and their right of access to entities, often through use of assistive technology.

In the context of web development, accessibility has a focus on enabling people with impariments full access to web interfaces. For example, this group of people can include partially visually impaired or completely blind people. While the former can still partially use the GUI, the latter have to completely rely on assistive technologies such as software which reads the screen to them (screen readers).

6.1.2. Why is it important for OTRS?

To enable impaired users access to OTRS systems is a valid goal in itself. It shows respect.

Furthermore, fulfilling accessibility standards is becoming increasingly important in the public sector (government institutions) and large companies, which both belong to the target markets of OTRS.

6.1.3. How can I successfully work on accessibility issues even if I am not disabled?

This is very simple. Pretend to be blind.

Don't

- use the Mouse and
- look at the screen.

Then try to use OTRS with the help of a screen reader and your keyboard only. This should give you an idea of how it will feel for a blind person.

6.1.4. Ok, but I don't have a screen reader!

While commercial screen readers such as JAWS (perhaps the best known one) can be extremely expensive, there are OpenSource screen readers which you can install and use:

- NVDA, a screen reader for Windows. (Use the 2010.beta1 or any later version as this has better support for the web accessibility standards).
- ORCA, a screen reader for Gnome/Linux.

Now you don't have an excuse any more. ;)

6.2. Accessibility Standards

This section is included for reference only, you do not have to study the standards themselves to be able to work on accessibility issues in OTRS. We'll try to extract the relevant guidelines in this document.

6.2.1. Web Content Accessibility Guidelines (WCAG)

This W3C standard gives general guidelines for how to create accessible web pages.

- WCAG 2.0
- How to Meet WCAG 2.0
- Understanding WCAG 2.0

WCAG has different levels of accessibility support. We currently plan to support level A, as AA and AAA deal with matters that seem not relevant for OTRS.

6.2.2. Accessible Rich Internet Applications (WAI-ARIA) 1.0

This standard (ist is still in draft status, in fact) deals with the special issues arising from the shift away from static content to dynamic web applications. It deals with questions like how a user can be notified of changes in the user interface resulting from AJAX requests, for example.

- WAI-ARIA 1.0

6.3. Implementation guidelines

6.3.1. Provide alternatives for non-text content

Goal: *All non-text content that is presented to the user has a text alternative that serves the equivalent purpose* (WCAG 1.1.1)

It is very important to understand that screen readers can only present textual information and available metadata to the user. To give you an example, whenever a screen reader sees `<a href="#" class="CloseLink"></a>`, it can only read "link" to the user, but not the purpose of this link. With a slight improvement, it would be accessible: `<a href="#" class="CloseLink" title="Close this widget"></a>`. In this case the user would hear "link close this widget", voila!

It is important to always formulate the text in a most "speaking" way. Just imagine it is the only information that you have. Will it help you? Can you understand its purpose just by hearing it?

Please follow these rules when working on OTRS:

- *Rul:*: Wherever possible, use speaking texts and formulate in real, understandable and precise sentences. "Close this widget" is much better than "Close", because the latter is redundant.
- *Rule*: Links always must have either text content that is spoken by the screen reader (`<a href="#" >Delete this entry</a>`), or a title attribute (`<a href="#" title="Close this widget"></a>`).
- *Rule*: Images must always have an alternative text that can be read to the user (`<img src="house.png" alt="Image of a house" />`).

6.3.2. Make navigation easy

Goal: *allow the user to easily navigate the current page and the entire application.*

The `title` tag is the first thing a user hears from the screen reader when opening a web page. For OTRS, there is also always just one h1 element on the page, indicating the current page (it contains part of the information from `title`). This navigational information helps the user to understand where they are, and what the purpose of the current page is.

- *Rule*: Always give a precise title to the page that allows the user to understand where they currently are.

Screen readers can use the built-in document structure of HTML (headings h1 to h6) to determine the structure of a document and to allow the user to jump around from section to section. However, this is not enough to reflect the structure of a dynamic web applica-

tion. That's why ARIA defines several "landmark" roles that can be given to elements to indicate their navigational significance. To keep the validity of the HTML documents, the role attributes (ARIA landmark roles) are not inserted into the source code directly, but instead by classes which will later be used by the JavaScript functions in OTRS.UI.Accessibility to set the corresponding role attributes on the node.

- *Rule*: Use WAI ARIA Landmark Roles to structure the content for screenreaders
  - Banner: `<div class="ARIARoleBanner"></div>` will become `<div class="ARIARoleBanner" role="banner"></div>`
  - Navigation: `<div class="ARIARoleNavigation"></div>` will become `<div class="AriaRoleNavigation" role="navigation"></div>`
  - Search function: `<div class="ARIARoleSearch"></div>` will become `<div class="ARIARoleSearch" role="search"></div>`
  - Main application area: `<div class="ARIARoleMain"></div>` will become `<div class="ARIARoleMain" role="main"></div>`
  - Footer: `<div class="ARIARoleContentinfo"></div>` will become `<div class="ARIARoleContentinfo" role="contentinfo"></div>`

For navigation inside of `<form<` elements, it is neccessary for the impaired user to know what each input elements purpose is. This can be achieved by using standard HTML `<label>` elements which create a link between the label and the form element. When an input element gets focus, the screen reader will usually read the connected label, so that the agent can hear its exact purpose. An additional benefit for seeing users is that they can click on the label, and the input element will get focus (escpecially helpful for checkboxes, for example).

- *Rule*: Provide `<label>` elements for *all* form element (input, select, textarea) fields.

  Example: `<label for="date">Date:</label><input type="text" name="date" id="date"/>`

6.3.3. Make interaction possible

Goal: *Allow the user to perform all interactions just by using the keyboard.*

While it is technically possible to create interactions with JavaScript on arbitrary HTML elements, this must be limited to elements that a user can interact with by using the keyboard. Specifically, they need to be able to give focus to the element and to interact with it. For example, a push button to toggle a widget should not be realized by using a span element with an attached JavaScript `onclick` event listener, but it should be (or contain) an a tag to make it clear to the screen reader that this element can cause interaction.

- *Rule*: For interactions, always use elements that can receive focus, such as a, `input`, `select` and `button`.
- *Rule*: Make sure that the user can always identify the nature of the interaction (see rules about non-textual content and labelling of form elements).

Goal: *Make dynamic changes known to the user.*

A special area of accessibility problems are dynamic changes in the user interface, either by JavaScript or also by AJAX calls. The screen reader will not tell the user about changes without special precautions. This is a difficult topic and cannot yet be completely explained here.

- *Rule*: Always use the validation framework `OTRS.Validate` for form validation.

This will make sure that the error tooltips are being read by the screen reader. That way the blind agent a) knows the item which has an error and b) get a text describing the error.

- *Rule*: Use the function `OTRS.UI.Accessibility.AudibleAlert()` to notify the user about other important UI changes.
- *Rule*: Use the `OTRS.UI.Dialog` framework to create modal dialogs. These are already optimized for accessibility.

6.3.4. General screen reader optimizations

Goal: *Help screen readers with their work.*

- *Rule*: Each page must identify its own main language so that the screenreader can choose the right speech synthesis engine.

  Example: `<html lang="fr">...</html>`

7. Unit Tests

OTRS provides unit tests for core modules.

7.1. Creating a test file

The test files are stored in .t files under /scripts/test/*.t. For example the file /scripts/test/Calendar.t for the Calendar Module.

A test file consists of the function call of the function to be tested and the analysis of the return value. Example (/scripts/test/Calendar.t):

```
# --
# Calendar.t - Calendar
# Copyright (C) 2001-2014 OTRS AG, http://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (AGPL). If you
# did not receive this file, see http://www.gnu.org/licenses/agpl.txt.
# --

use strict;
use warnings;
use utf8;

use vars qw($Self);

use Kernel::System::User;
use Kernel::System::CalendarEvent;

$Self->{UserObject} = Kernel::System::User->new(%{$Self});
$Self->{EventObject} = Kernel::System::CalendarEvent->new(%{$Self}, UserID => 1);


my $EventID = $Self->{EventObject}->EventAdd(
    Title => 'Some Test',
    StartTime => '1977-10-27 20:15',
    EndTime => '1977-10-27 21:00',
    State => 'public',
    UserIDs => [1],
);

$Self->True(
    $EventID,
```

```
    'EventAdd()',
);

[..]
```

7.2. Testing

To check your tests, just use "bin/otrs.UnitTest.pl -n Calendar" to use /scripts/test/Calendar.t.

```
shell:/opt/otrs> bin/otrs.UnitTest.pl -n Calendar
+-------------------------------------------------------------------+
/opt/otrs/scripts/test/Calendar.t:
+-------------------------------------------------------------------+
ok 1 - EventAdd()
=====================================================================
Product:   OTRS 3.2.x git
Test Time: 0 s
Time:      2010-04-02 12:58:37
Host:      yourhost.example.com
Perl:      5.8.9
OS:        linux
TestOk:    1
TestNotOk: 0
=====================================================================
shell:/opt/otrs>
```

7.3. True()

This function tests whether the return value of the function 'EventAdd()' in the variable $EventID is valid.

```
$Self->True(
    $EventID,
    'EventAdd()',
);
```

7.4. False()

This function tests whether the return value of the function 'EventAdd()' in the variable $EventID is invalid.

```
$Self->False(
    $EventID,
    'EventAdd()',
);
```

7.5. Is()

This function tests whether the variables $A and $B are equal.

```
$Self->Is(
    $A,
    $B,
    'Test Name',
);
```

Appendix A. Additional Resources

1. OTRS.org

The OTRS project website with source code, documentation and news is available at:

http://otrs.org/

2. Online API Library

The OTRS developer API documentation is available at:

http://otrs.github.io/doc/

3. Developer Mailing List

The OTRS developer mailing list is available at:

http://lists.otrs.org/

4. Commercial Support

For services (support, consulting, development, and training) you can contact the company behind OTRS, OTRS AG. They have offices in Germany, USA, Mexico, the Netherlands and other countries. Look at their website for contact information: http://www.otrs.com/en/corporate-navigation/contact/