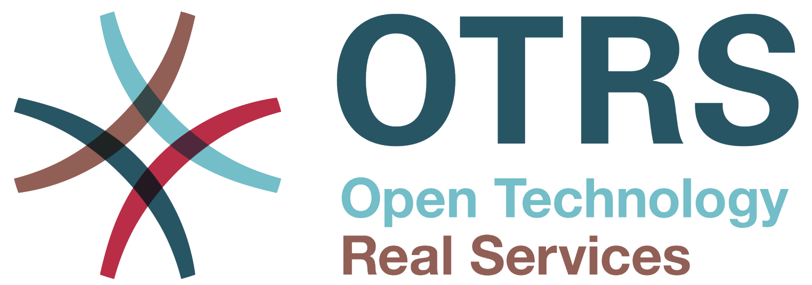


Documentation

OTRS 6 - Developer Manual

Build Date:

2016-03-14

OTRS 6 - Developer Manual

Copyright © 2003-2016 OTRS AG

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.

All trade names are used without the guarantee for their free use and are possibly registered trade marks. All products mentioned in this manual may be trade marks of the respective manufacturer.

The source code of this document can be found at github, in the repository doc-developer. Contributions are more than welcome. You can also help translating it to your language at Transifex.

Table of Contents

1. Getting Started ... 1
    1. Development Environment ... 1
        1.1. Framework checkout ... 1
        1.2. Useful Tools ... 1
        1.3. Linking Expansion Modules ... 1
    2. Architecture Overview ... 2
        2.1. Directories ... 3
        2.2. Files ... 4
        2.3. Core Modules ... 4
        2.4. Frontend Handle ... 4
        2.5. Frontend Modules ... 5
        2.6. CMD Frontend ... 5
        2.7. Generic Interface Modules ... 5
        2.8. Scheduler Task Handler Modules ... 6
        2.9. Database ... 6
2. OTRS Internals - How it Works ... 7
    1. Config Mechanism ... 7
        1.1. `Defaults.pm`: OTRS Default Configuration ... 7
        1.2. Automatically Generated Configuration Files ... 7
        1.3. XML Configuration Files ... 7
        1.4. Accessing Config Options at Runtime ... 15
    2. Database Mechanism ... 15
        2.1. How it works ... 15
        2.2. Database Drivers ... 19
        2.3. Supported Databases ... 19
    3. Log Mechanism ... 19
        3.1. System Log ... 19
        3.2. Communication Log ... 19
    4. Date and Time ... 22
        4.1. Introduction ... 22
        4.2. Creation of a DateTime object ... 22
        4.3. Time zones ... 23
        4.4. Method summary ... 23
        4.5. Deprecated package Kernel::System::Time ... 24
    5. Skins ... 24
        5.1. Skin Basics ... 24
        5.2. How skins are loaded ... 25
        5.3. Creating a New Skin ... 26
    6. The CSS and JavaScript "Loader" ... 27
        6.1. How it works ... 27
        6.2. Basic Operation ... 28
        6.3. Configuring the Loader: JavaScript ... 29
        6.4. Configuring the Loader: CSS ... 30
    7. Templating Mechanism ... 31
        7.1. Template Commands ... 31
        7.2. Using a template file ... 37
    8. Creating Your Own Themes ... 38
    9. Localization / Translation Mechanism ... 38
        9.1. Marking translatable strings in the source files ... 38
        9.2. Collecting translatable strings into the translation database ... 39
        9.3. The translation process itself ... 40
        9.4. Using the translated data from the code ... 41
3. How to Extend OTRS ... 42
    1. Writing a new OTRS frontend module ... 42
        1.1. What we want to write ... 42

1.2. Default Config File ... 42
1.3. Frontend Module ... 43
1.4. Core Module ... 44
1.5. Template File ... 46
1.6. Language File ... 46
1.7. Summary ... 46
2. Using the power of the OTRS module layers ... 46
2.1. Authentication and user management ... 47
2.2. Preferences ... 54
2.3. Other core functions ... 63
2.4. Frontend Modules ... 85
2.5. Generic Interface Modules ... 93
2.6. Daemon And Scheduler ... 113
2.7. Dynamic Fields ... 119
2.8. Email Handling ... 150
4. How to Publish Your OTRS Extensions ... 153
1. Package Management ... 153
1.1. Package Distribution ... 153
1.2. Package Commands ... 153
2. Package Building ... 154
2.1. Package Spec File ... 154
2.2. Example .sopm ... 160
2.3. Package Build ... 161
2.4. Package Life Cycle - Install/Upgrade/Uninstall ... 161
3. Package Porting ... 161
3.1. From OTRS 5 to 6 ... 161
3.2. From OTRS 4 to 5 ... 176
3.3. From OTRS 3.3 to 4 ... 178
5. Contributing to OTRS ... 186
1. Sending Contributions ... 186
2. Translating OTRS ... 186
2.1. Updating an existing translation ... 186
2.2. Adding a new frontend translation ... 187
3. Translating the Documentation ... 187
4. Code Style Guide ... 187
4.1. Perl ... 187
4.2. JavaScript ... 199
4.3. HTML ... 200
4.4. CSS ... 200
5. User Interface Design ... 201
5.1. Capitalization ... 201
6. Accessibility Guide ... 202
6.1. Accessibility Basics ... 202
6.2. Accessibility Standards ... 203
6.3. Implementation guidelines ... 203
7. Unit Tests ... 205
7.1. Creating a test file ... 205
7.2. Prerequisites for testing ... 207
7.3. Testing ... 208
7.4. Unit Test API ... 208
A. Additional Resources ... 212

List of Figures

1.1. OTRS Architecture ... 2
1.2. Generic Interface Architecture ... 3
3.1. Dashboard Widget ... 86
3.2. Dynamic Fields Architecture ... 120
3.3. Dynamic Field Interaction ... 123
3.4. Email Processing Flow ... 152
4.1. Package Life Cycle ... 161

List of Tables

4.1. Template Changes from OTRS 3.3 to 4 183

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 by using the following command:

```
# for git master
shell> git clone git@github.com:OTRS/otrs.git -b master
# for a specific branch like OTRS 3.3
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:

```
shell> git clone git@github.com:OTRS/module-tools.git
```

Please configure the OTRS system according to the installation instructions in the admin manual.

1.2. Useful Tools

There are two modules that are highly recommended for OTRS development: OTRSCodePolicy and Fred.

OTRSCodePolicy is a code quality checker that enforces the use of common coding standards also for the OTRS development team. It is highly recommended to use it if you plan to make contributions. You can use it as a standalone test script or even register it as a git commit hook that runs every time that you create a commit. Please see the module documentation for details.

Fred is a little development helper module that you can actually install or link (as described below) into your development system. It features several helpful modules that you can activate, such as an SQL logger or an STDERR console. You can find some more details in its module documentation.

By the way, these tools are also open source, and we will be happy about any improvements that you can contribute.

1.3. Linking Expansion Modules

A clear separation between OTRS and the modules is necessary for proper developing. Particularly when using a git clone, a clear separation is crucial. In order to facilitate the

OTRS access the files, links must be created. This is done by a script in the directory module tools repository. 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.

As soon as the linking is completed, the SysConfig must be rebuilt to register the module in OTRS. Additional SQL or Perl code from the module must also be executed. Example:

```
shell> ~/src/otrs/bin/otrs.Console.pl Maint::Config::Rebuild
shell> ~/src/module-tools/DatabaseInstall.pl -m Calendar.sopm -a install
shell> ~/src/module-tools/CodeInstall.pl -m Calendar.sopm -a install
```

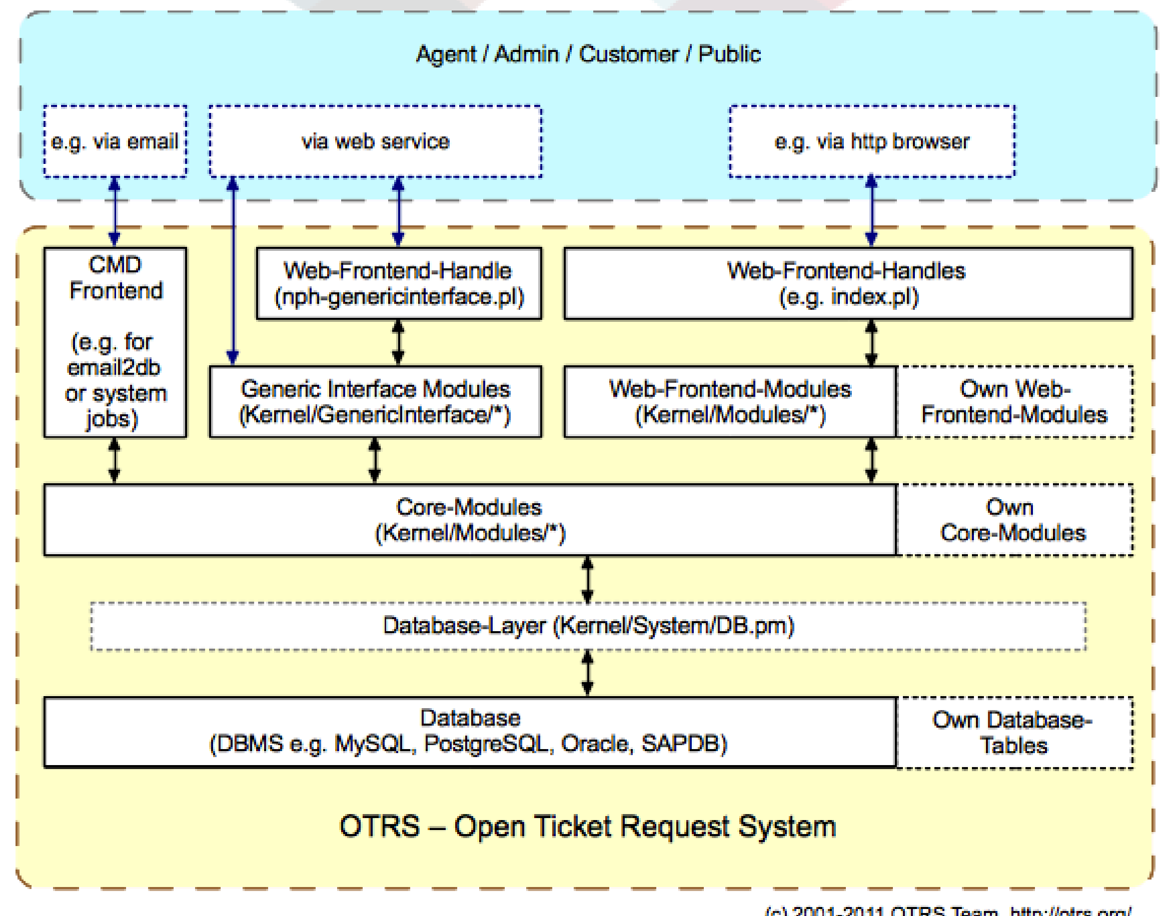
To remove links from OTRS enter the following command:

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

2. Architecture Overview

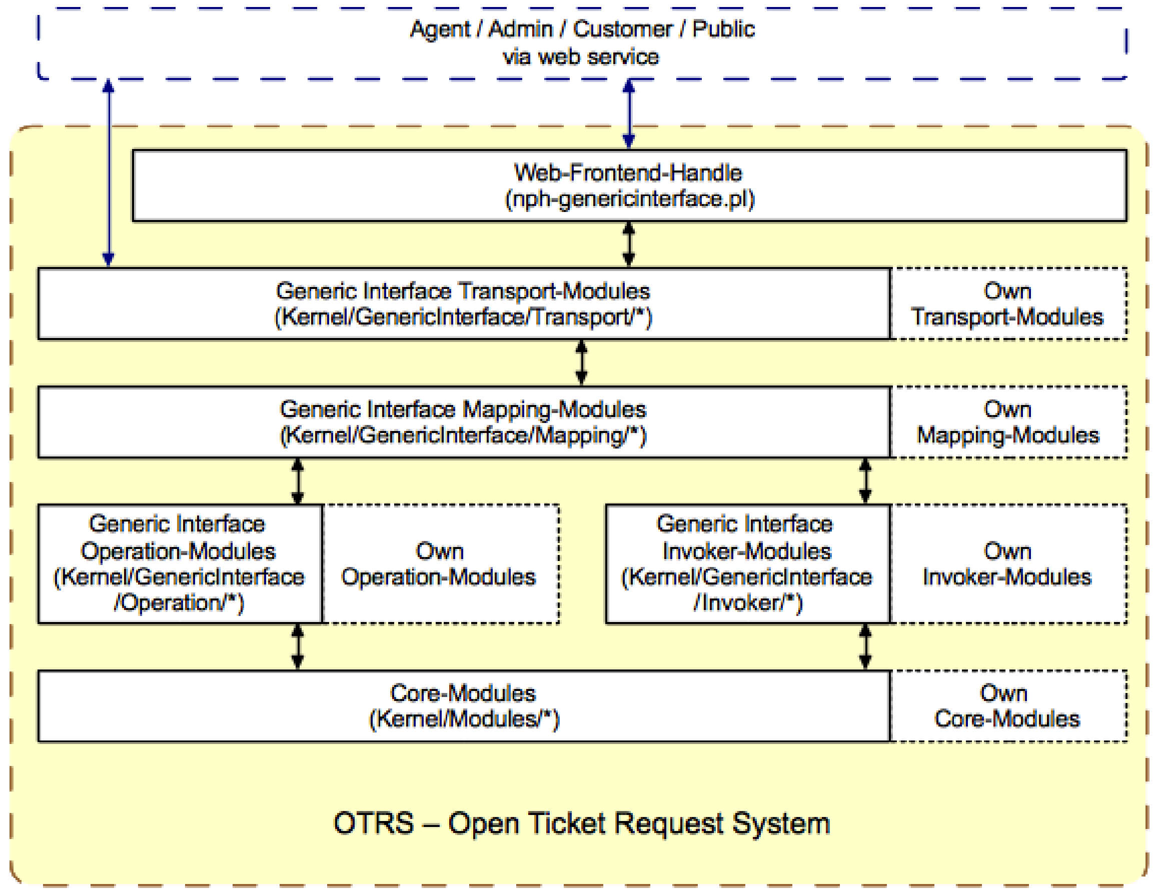
The OTRS framework is modular. The following picture shows the basic layer architecture of OTRS.

Figure 1.1. OTRS Architecture



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

Figure 1.2. Generic Interface Architecture



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

Directory	Description
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

.tt = Template::Toolkit 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 a 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 perform actions requested 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 directory 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 receive 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

OTRS comes with a dedicated mechanism to manage configuration options via a graphical interface (System Configuration). This section describes how it works internally and how you can provide new configuration options or change existing default values.

1.1. `Defaults.pm`: OTRS Default Configuration

The default configuration file of OTRS is `Kernel/Config/Defaults.pm`. This file is needed for operation of freshly installed systems without a deployed XML configuration and should be left untouched as it is automatically updated on framework updates.

1.2. Automatically Generated Configuration Files

In `Kernel/Config/Files` you can find some automatically generated configuration files:

`ZZZAAuto.pm`

Perl cache of the XML configuration's current values (default or modified by user)

`ZZZACL.pm`

Perl cache of ACL configuration from database

`ZZZACL.pm`

Perl cache of ProcessManagement configuration from database

These files are a Perl representation of the current system configuration. They should never be manually changed as they are overwritten by OTRS.

1.3. XML Configuration Files

In OTRS, configuration options that the administrator can configure via SysConfig are provided via XML files with a special format. To convert old XML's you can use 'otrs.Console.pl Dev::Tools::Migrate::ConfigXMLStructure' command. The file `Kernel/Config/Files/ZZZAAuto.pm` is a cached Perl version of the XML that contains all settings with their current value. It can be (re-)generated with `bin/otrs.Console.pl Maint::Config::Rebuild`.

Note:`$OTRS_HOME/Kernel/Config/Files/ZZZAuto.pm` does not exist any more, it has been merged into `$OTRS_HOME/Kernel/Config/Files/ZZZAAuto.pm`.

Each XML config file has the following layout:

```
<?xml version="1.0" encoding="utf-8" ?>
<otrs_config version="2.0" init="Changes">

    <!--  settings will be here -->

</otrs_config>
```

Attributes of the `otrs_config` element

`init`

The global `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).

The configuration items are written as `Setting` elements with a `Description`, a `Navigation` group (for the tree-based navigation in the GUI) and the `Value` that it represents. Here's an example:

```
<Setting Name="Ticket::Hook" Required="1" Valid="1">
    <Description Translatable="1">The identifier for a ticket, e.g. Ticket#, Call#,
 MyTicket#. The default is Ticket#.</Description>
    <Navigation>Core::Ticket</Navigation>
    <Value>
        <Item ValueType="String" ValueRegex="">Ticket#</Item>
    </Value>
</Setting>
```

Attributes of the `Setting` element

`Required`

If this is set to 1, the config setting cannot be disabled.

`Valid`

Determines if the config setting is active (1) or inactive (0) by default.

`ConfigLevel`

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 external interaction, like Sendmail, LDAP, SOAP, and others should get a config level of at least 200 (Advanced).

`Invisible`

If set to 1, the config setting is not shown in the GUI.

`Readonly`

If set to 1, the config setting cannot be changed in the GUI.

`UserModificationPossible`

If `UserModificationPossible` is set to 1, administrators can enable user modifications of this setting (in user preferences). Please note that this feature requires the **OTRS Business Solution™**.

`UserModificationActive`

If `UserModificationActive` is set to 1, user modifications of this setting is enabled (in user preferences). You should use this attribute together with `UserModificationPossible`.

UserPreferencesGroup

Use UserPreferencesGroup attribute to define under which group config variable belongs (in the UserPreferences screen). You should use this attribute together with UserModificationPossible.

Guidelines for placing settings in the right Navigation nodes

- Only create new nodes if neccessary. Avoid nodes with only very few settings if possible.
- On the first tree level, no new nodes should be added.
- Don't place new settings in Core directly. This is reserved for a few important global settings.
- Core::* can take new groups that contain settings that cover the same topic (like Core::Email) or relate to the same entity (like Core::Queue).
- All event handler registrations go to Core::Event.
- Don't add new direct child nodes within Frontend. Global front end settings go to Frontend::Base, settings only affecting a part of the system go to the respective Admin, Agent, Customer or Public sub nodes.
- Front end settings that only affect one screen should go to the relevant screen (View) node (create one if needed), for example AgentTicketZoom related settings go to Frontend::Agent::View::TicketZoom. If there are module layers within one screen with groups of related settings, they would also go to a sub group here (e. g. Frontend::Agent::View::TicketZoom::MenuModule for all ticket zoom menu module registrations).
- All global Loader settings go to Frontend::Base::Loader, screen specific Loader settings to Frontend::*::ModuleRegistration::Loader.

1.3.1. Structure of Value elements

Value elements hold the actual configuration data payload. They can contain single values, hashes or arrays.

1.3.1.1. Item

An Item element holds one piece of data. The optional ValueType attribute determines which kind of data and how it needs to be presented to the user in the GUI. If no ValueType is specified, it defaults to String.

Please see below for a definition of the different value types.

```
<Setting Name="Ticket::Hook" Required="1" Valid="1">
    <Description Translatable="1">The identifier for a ticket, e.g. Ticket#, Call#,
 MyTicket#. The default is Ticket#.</Description>
    <Navigation>Core::Ticket</Navigation>
    <Value>
        <Item ValueType="String" ValueRegex="">Ticket#</Item>
    </Value>
</Setting>
```

1.3.1.2. Array

With this config element arrays can be displayed.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Array>
            <Item Translatable="1">Value 1</Item>
            <Item Translatable="1">Value 2</Item>
            ...
        </Array>
    </Value>
</Setting>
```

1.3.1.3. Hash

With this config element hashes can be displayed.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Hash>
            <Item Key="One" Translatable="1">First</Item>
            <Item Key="Two" Translatable="1">Second</Item>
            ...
        </Hash>
    </Value>
</Setting>
```

It's possible to have nested array/hash elements (like hash of arrays, array of hashes, array of hashes of arrays, ...). Below is an example array of hashes.

```
<Setting Name="ExampleAoH">
    ...
    <Value>
        <Array>
            <DefaultItem>
                <Hash>
                    <Item></Item>
                </Hash>
            </DefaultItem>
            <Item>
                <Hash>
                    <Item Key="One">1</Item>
                    <Item Key="Two">2</Item>
                </Hash>
            </Item>
            <Item>
                <Hash>
                    <Item Key="Three">3</Item>
                    <Item Key="Four">4</Item>
                </Hash>
            </Item>
        </Array>
    </Value>
</Setting>
```

1.3.2. Value Types

The XML config settings support various types of configuration variables.

1.3.2.1. String

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="String" ValueRegex=""></Item>
```

```
    </Value>
</Setting>
```

A config element for numbers and single-line strings. Checking the validity with a regular expression is possible (optional). This is the default `ValueType`.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="String" ValueRegex="" Translatable="1">Value</Item>
    </Value>
</Setting>
```

The optional `Translatable` attribute marks this setting as translatable, which will cause it to be included in the OTRS translation files. This attribute can be placed on any tag (see also below).

1.3.2.2. Password

A config element for passwords. It's still stored as plain text in the XML, but it's masked in the GUI.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="Password">Secret</Item>
    </Value>
</Setting>
```

1.3.2.3. PerlModule

A config element for Perl module. It has a `ValueFilter` attribute, which filters possible values for selection. In the example below, user can select Perl module `Kernel::System::Log::SysLog` or `Kernel::System::Log::File`.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="PerlModule" ValueFilter="Kernel/System/Log/
*.pm">Kernel::System::Log::SysLog</Item>
    </Value>
</Setting>
```

1.3.2.4. Textarea

A config element for multiline text.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="Textarea"></Item>
    </Value>
</Setting>
```

1.3.2.5. Select

This config element offers preset values as a pull-down menu. The `SelectedID` or `SelectedValue` attributes can pre-select a default value.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="Select" SelectedID="Queue">
            <Item ValueType="Option" Value="Queue" Translatable="1">Queue</Item>
            <Item ValueType="Option" Value="SystemAddress" Translatable="1">System address</
Item>
        </Item>
    </Value>
</Setting>
```

1.3.2.6. Checkbox

This config element checkbox has two states: 0 or 1.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="Checkbox">0</Item>
    </Value>
</Setting>
```

1.3.2.7. Date

This config element contains a date value.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="Date">2016-02-02</Item>
    </Value>
</Setting>
```

1.3.2.8. DateTime

This config element contains a date and time value.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="DateTime">2016-12-08 01:02:03</Item>
    </Value>
</Setting>
```

1.3.2.9. Directory

This config element contains a directory.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="Directory">/etc</Item>
    </Value>
</Setting>
```

1.3.2.10. File

This config element contains a file path.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="File">/etc/hosts</Item>
    </Value>
</Setting>
```

1.3.2.11. Entity

This config element contains a value of a particular entity. `ValueEntityType` attribute defines the entity type. Supported entities: `DynamicField`, `Queue`, `Priority`, `State` and `Type`. Consistency checks will ensure that only valid entities can be configured and that entities used in the configuration cannot be set to invalid. Also, when an entity is renamed, all referencing config settings will be updated.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="Entity" ValueEntityType="Queue">Junk</Item>
    </Value>
</Setting>
```

1.3.2.12. TimeZone

This config element contains a time zone value.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="TimeZone">UTC</Item>
    </Value>
</Setting>
```

1.3.2.13. VacationDays

This config element contains definitions for vacation days which are repeating each year. Following attributes are mandatory: `ValueMonth`, `ValueDay`.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="VacationDays">
            <DefaultItem ValueType="VacationDays"></DefaultItem>
            <Item ValueMonth="1" ValueDay="1" Translatable="1">New Year's Day</Item>
            <Item ValueMonth="5" ValueDay="1" Translatable="1">International Workers' Day</
Item>
            <Item ValueMonth="12" ValueDay="24" Translatable="1">Christmas Eve</Item>
        </Item>
    </Value>
</Setting>
```

1.3.2.14. VacationDaysOneTime

This config element contains definitions for vacation days which occur only once. Following attributes are mandatory: `ValueMonth`, `ValueDay` and `ValueYear`.

```
<Setting Name="SettingName">
    ...
    <Value>
```

```
        <Item ValueType="VacationDaysOneTime">
            <Item ValueYear="2004" ValueMonth="1" ValueDay="1">test</Item>
        </Item>
    </Value>
</Setting>
```

1.3.2.15. WorkingHours

This config element contains definitions for working hours.

```
<Setting Name="SettingName">
    ...
    <Value>
        <Item ValueType="WorkingHours">
            <Item ValueType="Day" ValueName="Mon">
                <Item ValueType="Hour">8</Item>
                <Item ValueType="Hour">9</Item>
            </Item>
            <Item ValueType="Day" ValueName="Tue">
                <Item ValueType="Hour">8</Item>
                <Item ValueType="Hour">9</Item>
            </Item>
        </Item>
    </Value>
</Setting>
```

1.3.2.16. Frontend Registration

Module registration for Agent Interface:

```
<Setting Name="SettiFrontend::Module###AgentModuleName">
    ...
    <Value>
        <Item ValueType="FrontendRegistration">
            <Hash>
                <Item Key="Group">
                    <Array>
                    </Array>
                </Item>
                <Item Key="GroupRo">
                    <Array>
                    </Array>
                </Item>
                <Item Key="Description" Translatable="1">Phone Call.</Item>
                <Item Key="Title" Translatable="1">Phone-Ticket</Item>
                <Item Key="NavBarName">Ticket</Item>
            </Hash>
        </Item>
    </Value>
</Setting>
```

1.3.3. DefaultItem in Array and Hash

The new XML structure allows us to create complex structures. Therefore we need `DefaultItem` entries to describe the structure of the Array/Hash. If it's not provided, system considers that you want simple Array/Hash with scalar values. `DefaultItem` is used as a template when adding new elements, so it can contain additional attributes, like `ValueType`, and it can define default values.

Here are few examples:

1.3.3.1. Array of Array with Select items

```
<Array>
    <DefaultItem>
        <Array>
            <DefaultItem ValueType="Select" SelectedID='option-2'>
                <Item ValueType="Option" Value="option-1">Option 1</Item>
                <Item ValueType="Option" Value="option-2">Option 2</Item>
            </DefaultItem>
        </Array>
    </DefaultItem>
    ...
</Array>
```

1.3.3.2. Hash of Hash with Date items

```
<Hash>
    <DefaultItem>
        <Hash>
            <DefaultItem ValueType="Date">2017-01-01</DefaultItem>
        </Hash>
    </DefaultItem>
    ...
</Hash>
```

1.4. Accessing Config Options at Runtime

You can read and write (for one request) the config options via the core module `Kernel::Config`.

If you want to read a config option:

```
my $ConfigOption = $Kernel::OM->Get('Kernel::Config')->Get('Prefix::Option');
```

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

```
$Kernel::OM->Get('Kernel::Config')->Set(
    Key => 'Prefix::Option'
    Value => 'SomeNewValue',
);
```

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

```
$Kernel::OM->Get('Kernel::System::DB')->Do(
    SQL=> "INSERT INTO table (name, id) VALUES ('SomeName', 123)",
```

```
);

$Kernel::OM->Get('Kernel::System::DB')->Do(
    SQL=> "UPDATE table SET name = 'SomeName', id = 123",
);

$Kernel::OM->Get('Kernel::System::DB')->Do(
    SQL=> "DELETE FROM table WHERE id = 123",
);
```

2.1.1.2. SELECT

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

$Kernel::OM->Get('Kernel::System::DB')->Prepare(SQL => $SQL, Limit => 15);

while (my @Row = $Kernel::OM->Get('Kernel::System::DB')->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 = ?";

$Kernel::OM->Get('Kernel::System::DB')->Prepare(
    SQL   => $SQL,
    Limit => 15,
    Bind  => [ $Tn, $Group ],
);

while (my @Row = $Kernel::OM->Get('Kernel::System::DB')->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 = $Kernel::OM->Get('Kernel::System::DB')->Quote("It's a problem!");
```

Integer:

```
my $QuotedInteger = $Kernel::OM->Get('Kernel::System::DB')->Quote('123', 'Integer');
```

Number:

```
my $QuotedNumber = $Kernel::OM->Get('Kernel::System::DB')->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> syntax has changed in >=2.2. Values are now used in the tag content (not longer in an attribute).

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>
```

LONGBLOB columns need special treatment. Their content needs to be Base64 transcoded if the database driver does not support the feature DirectBlob. Please see the following example:

```
my $Content = $StorableContent;
if ( !$DBObject->GetDatabaseFunction('DirectBlob') ) {
    $Content = MIME::Base64::encode_base64($StorableContent);
}
```

Similarly, when reading from such a column, the content must not automatically be decoded as UTF-8 by passing the Encode => 0 flag to Prepare():

```
return if !$DBObject->Prepare(
    SQL => '
```

```
        SELECT content_type, content, content_id, content_alternative, disposition, filename
        FROM article_data_mime_attachment
        WHERE id = ?',
    Bind   => [ \$AttachmentID ],
    Encode => [ 1, 0, 0, 0, 1, 1 ],
);

while ( my @Row = $DBObject->FetchrowArray() ) {

    $Data{ContentType} = $Row[0];

    # Decode attachment if it's e. g. a postgresql backend.
    if ( !$DBObject->GetDatabaseFunction('DirectBlob') ) {
        $Data{Content} = decode_base64( $Row[1] );
    }
    else {
        $Data{Content} = $Row[1];
    }
    $Data{ContentID}          = $Row[2] || '';
    $Data{ContentAlternative} = $Row[3] || '';
    $Data{Disposition}        = $Row[4];
    $Data{Filename}           = $Row[5];
}
```

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 = $Kernel::OM->Get('Kernel::System::DB')->SQLProcessor(
    Database => \@XMLARRAY,
);
push(@SQL, $Kernel::OM->Get('Kernel::System::DB')->SQLProcessorPost());

for (@SQL) {
    $Kernel::OM->Get('Kernel::System::DB')->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 (only for external database connections, not as OTRS database)

3. Log Mechanism

3.1. System Log

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

The Log object can be accessed and used via the `ObjectManager` like this:

```
$Kernel::OM->Get('Kernel::System::Log')->Log(
    Priority => 'error',
    Message  => 'Need something!',
);
```

Depending on the configured log level via `MinimumLogLevel` option in SysConfig, logged message will either be saved or not, based on their *Priority* flag.

If `error` is set, just errors are logged. With debug, you get all logging messages. The order of log levels is:

- `debug`
- `info`
- `notice`
- `error`

The output of the system log can be directed to either a syslog daemon or log file, depending on the configured `LogModule` option in SysConfig.

3.2. Communication Log

In addition to System Log, OTRS provides specialized logging backend for any communication related logging. Since OTRS 6, system comes with dedicated tables and frontends to track and display communication logs for easier debugging and operational overview.

To take advantage of the new system, first create a non-singleton instance of communication log object:

```
my $CommunicationLogObject = $Kernel::OM->Create(
    'Kernel::System::CommunicationLog',
    ObjectParams => {
        Transport   => 'Email',      # Transport log module
        Direction   => 'Incoming',   # Incoming|Outgoing
        AccountType => 'POP3',       # Mail account type
        AccountID   => 1,            # Mail account ID
    },
);
```

When you have a communication log object instance, you can start an object log for logging individual messages. There are two object logs currently implemented: `Connection` and `Message`.

`Connection` object log should be used for logging any connection related messages (for example: authenticating on server or retrieving incoming messages).

Simply, start the object log by declaring its type, and you can use it immediately:

```
$CommunicationLogObject->ObjectLogStart(
    ObjectLogType => 'Connection',
);

$CommunicationLogObject->ObjectLog(
    ObjectLogType => 'Connection',
    Priority      => 'Debug',                              # Trace, Debug, Info, Notice,
 Warning or Error
    Key           => 'Kernel::System::MailAccount::POP3',
    Value         => "Open connection to 'host.example.com' (user-1).",
);
```

The communication log object instance handles the current started object logs, so you don't need to remember and bring them around everywhere, but it also means that you can only start one object per type.

If you encounter an unrecoverable error, you can choose to close the object log and mark it as failed:

```
$CommunicationLogObject->ObjectLog(
    ObjectLogType => 'Connection',
    Priority      => 'Error',
    Key           => 'Kernel::System::MailAccount::POP3',
    Value         => 'Something went wrong!',
);

$CommunicationLogObject->ObjectLogStop(
    ObjectLogType => 'Connection',
    Status        => 'Failed',
);
```

In turn, you can mark the communication log as failure as well:

```
$CommunicationLogObject->CommunicationStop(
    Status => 'Failed',
);
```

Otherwise, stop the object log and in turn communication log as success:

```
$CommunicationLogObject->ObjectLog(
    ObjectLogType => 'Connection',
    Priority      => 'Debug',
    Key           => 'Kernel::System::MailAccount::POP3',
    Value         => "Connection to 'host.example.com' closed.",
);

$CommunicationLogObject->ObjectLogStop(
    ObjectLogType => 'Connection',
    Status        => 'Successful',
);

$CommunicationLogObject->CommunicationStop(
    Status => 'Successful',
);
```

Message object log should be used for any log entries regarding specific messages and their processing. It is used in a similar way, just make sure to start it before using it:

```
$CommunicationLogObject->ObjectLogStart(
    ObjectLogType => 'Message',
);

$CommunicationLogObject->ObjectLog(
    ObjectLogType => 'Message',
    Priority      => 'Error',
    Key           => 'Kernel::System::MailAccount::POP3',
    Value         => "Could not process message. Raw mail saved (report it on http://
bugs.otrs.org/)!",
);

$CommunicationLogObject->ObjectLogStop(
    ObjectLogType => 'Message',
    Status        => 'Failed',
);

$CommunicationLogObject->CommunicationStop(
    Status => 'Failed',
);
```

You also have the possibility to link the log object and later lookup the communications for a certain object type and ID:

```
$CommunicationLogObject->ObjectLookupSet(
    ObjectLogType    => 'Message',
    TargetObjectType => 'Article',
    TargetObjectID   => 2,
);

my $LookupInfo = $CommunicationLogObject->ObjectLookupGet(
    TargetObjectType => 'Article',
    TargetObjectID   => 2,
);
```

You should make sure to always stop communication and flag it as failed, if any log object failed as well. This will allow administrators to see failed communications in the overview, and take any action if needed.

It's important to preserve the communication log for duration of a single process. If your work is spanning over multiple modules and any of them can benefit from logging, make sure to pass the existing communication log instance around so all methods can use the same one. With this approach, you will make sure any log entries spawned for the same process are contained in a single communication.

If passing the communication log instance is not an option (async tasks!), you can also choose to recreate the communication log object in the same state as in previous step. Just get the communication ID and pass it to the new code, and then create the instance with this parameter supplied:

```
# Get communication ID in parent code.
my $CommunicationID = $CommunicationLogObject->CommunicationIDGet();

# Somehow pass communication ID to child code.
# ...

# Recreate the instance in child code by using same communication ID.
my $CommunicationLogObject = $Kernel::OM->Create(
    'Kernel::System::CommunicationLog',
    ObjectParams => {
        CommunicationID => $CommunicationID,
    },
);
```

You can then continue to use this instance as previously stated, start any object logs if needed, adding entries and setting status in the end.

If you need to retrieve the communication log data or do something else with it, please also take a look at `Kernel::System::CommunicationLog::DB.pm`

4. Date and Time

OTRS comes with its own package to handle date and time which ensures correct calculation and storage of date and time.

4.1. Introduction

Date and time are represented by an object of `Kernel::System::DateTime`. Every `DateTime` object holds its own date, time and time zone information. In contrast to the now deprecated `Kernel::System::Time` package, this means that you can and should create a `DateTime` object for every date/time you want to use.

4.2. Creation of a DateTime object

The object manager of OTRS has been extended by a `Create` method to support packages for which more than one instance can be created:

```
my $DateTimeObject = $Kernel::OM->Create(
    'Kernel::System::DateTime',
    ObjectParams => {
        TimeZone => 'Europe/Berlin'
    },
);
```

The example above will create a `DateTime` object for the current date and time in time zone Europe/Berlin. There are more options to create a `DateTime` object (time components, string, time stamp, cloning), see POD of `Kernel::System::DateTime`.

Note

You will get an error if you try to retrieve a `DateTime` object via `$Kernel::OM->Get('Kernel::System::DateTime')`.

4.3. Time zones

Time offsets in hours (+2, -10, etc.) have been replaced by time zones (Europe/Berlin, America/New_York, etc.). The conversion between time zones is completely encapsulated within a `DateTime` object. If you want to convert to another time zone, simply use the following code:

```
$DateTimeObject->ToTimeZone( TimeZone => 'Europe/Berlin' );
```

There is a new SysConfig option `OTRSTimeZone`. This setting defines the time zone that OTRS uses internally to store date and time within the database.

Note

You have to ensure to convert a `DateTime` object to the OTRS time zone before it gets stored in the database (there's a convenient method for this: `ToOTRSTimeZone()`). An exception could be that you explicitly want a database column to hold a date/time in a specific time zone. But be aware that the database itself won't provide time zone information by itself when retrieving it.

Note

`TimeZoneList()` of `Kernel::System::DateTime` provides a list of available time zones.

4.4. Method summary

The `Kernel::System::DateTime` package provides the following methods (this is only a selection, see source code for details).

4.4.1. Object creation methods

A `DateTime` object can be created either via the object manager's `Create()` method or by cloning another `DateTime` object with its `Clone()` method.

4.4.2. Get method

With `Get()` all data of a `DateTime` object will be returned as a hash (date and time components including day name, etc. as well as time zone).

4.4.3. Set method

With `Set()` you can either change certain components of the `DateTime` object (year, month, day, hour, minute, second) or you can set a date and time based on a given string ('2016-05-24 23:04:12'). Note that you cannot change the time zone with this method.

4.4.4. Time zone methods

To change the time zone of a `DateTime` object use method `ToTimeZone()` or as a shortcut for converting to OTRS time zone `ToOTRSTimeZone()`.

To retrieve the configured OTRS time zone or user default time zone, always use method `OTRSTimeZoneGet()` or `UserDefaultTimeZoneGet()`. Never retrieve these manually via `Kernel::Config`.

4.4.5. Comparison operators and methods

`Kernel::System::DateTime` uses operator overloading for comparisons. So you can simply compare two `DateTime` objects with <, <=, ==, !=, >= and >. `Compare()` is usable in Perl's sort context as it returns -1, 0 or 1.

4.5. Deprecated package Kernel::System::Time

The now deprecated package `Kernel::System::Time` has been extended to fully support time zones instead of time offsets. This has been done to ensure that existing code works without (bigger) changes.

However, there is a case in which you have to change existing code. If you have code that uses the old time offsets to calculate a new date/time or a difference, you have to migrate this code to use the new `DateTime` object.

Example (old code):

```
my $TimeObject     = $Kernel::OM->Get('Kernel::System::Time'); # Assume a time offset of 0
 for this time object
my $SystemTime     = $TimeObject->TimeStamp2SystemTime( String => '2004-08-14 22:45:00' );
my $UserTimeZone   = '+2'; # normally retrieved via config or param
my $UserSystemTime = $SystemTime + $UserTimeZone * 3600;
my $UserTimeStamp  = $TimeObject->SystemTime2TimeStamp( SystemTime => $UserSystemTime );
```

Example (new code):

```
my $DateTimeObject = $Kernel::OM->Create('Kernel::System::DateTime'); # This implicitly sets
 the configured OTRS time zone
my $UserTimeZone   = 'Europe/Berlin'; # normally retrieved via config or param
$DateTimeObject->ToTimeZone( TimeZone => $UserTimeZone );
my $SystemTime    = $DateTimeObject->ToEpoch(); # note that the epoch is independent from
 the time zone, it's always calculated for UTC
my $UserTimeStamp = $DateTimeObject->ToString();
```

5. 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, ...).

5.1. 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.

5.2. 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 bandwidth 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 this:

```
.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;
}
```

5.3. 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 three 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="Changes">
    <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.Console.pl Maint::Config::Rebuild). 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
```

6. 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.

6.1. 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 unnecessary whitespace and documentation
- serves it to the client in only a few HTTP requests instead of many individual ones, allowing the client to cache these snippets in the browser cache
- performs 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.

6.2. Basic Operation

With the configuration settings `Loader::Enabled::CSS` and `Loader::Enabled::JavaScript`, the loader can be turned on and off for CSS and JavaScript, 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 unnecessary 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.

6.3. 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.

6.3.1. 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:

```
<Setting 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>
    <Navigation>Frontend::Base::Loader</Navigation>
    <Value>
        <Array>
            <Item>thirdparty/jquery-3.2.1/jquery.js</Item>
            <Item>thirdparty/jquery-browser-detection/jquery-browser-detection.js</Item>

            ...

            <Item>Core.Agent.Header.js</Item>
            <Item>Core.UI.Notification.js</Item>
            <Item>Core.Agent.Responsive.js</Item>
        </Array>
    </Value>
</Setting>
```

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:

```
<Setting 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>
```

```
    <Navigation>Frontend::Base::Loader</Navigation>
    <Value>
        <Array>
            <Item>thirdparty/jquery-3.2.1/jquery.js</Item>
        </Array>
    </Value>
</Setting>
```

Simple, isn't it?

6.3.2. 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:

```
<Setting Name="Loader::Module::AgentDashboard###001-Framework" Required="0" Valid="1">
    <Description Translatable="1">Loader module registration for the agent interface.</
Description>
    <Navigation>Frontend::Agent::ModuleRegistration::Loader</Navigation>
    <Value>
        <Hash>
            <Item Key="CSS">
                <Array>
                    <Item>Core.Agent.Dashboard.css</Item>

                    ...

                </Array>
            </Item>
            <Item Key="JavaScript">
                <Array>
                    <Item>thirdparty/momentjs-2.18.1/moment.min.js</Item>
                    <Item>thirdparty/fullcalendar-3.4.0/fullcalendar.min.js</Item>
                    <Item>thirdparty/d3-3.5.6/d3.min.js</Item>
                    <Item>thirdparty/nvd3-1.7.1/nvd3.min.js</Item>
                    <Item>thirdparty/nvd3-1.7.1/models/OTRSLineChart.js</Item>
                    <Item>thirdparty/nvd3-1.7.1/models/OTRSMultiBarChart.js</Item>
                    <Item>thirdparty/nvd3-1.7.1/models/OTRSStackedAreaChart.js</Item>
                    <Item>thirdparty/canvg-1.4/rgbcolor.js</Item>
                </Array>
            </Item>
        </Hash>
    </Value>
</Setting>
```

It is possible to put a `<Item Key="JavaScript">` tag in the frontend module registrations which may contain `<Array>` and one tag `<Item>` for each JavaScript 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.

6.4. 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.

6.4.1. Common CSS

The way common CSS is handled is very similar to the way common JavaScript is loaded.

7. 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/*.tt`

OTRS relies on the Template::Toolkit rendering engine. The full Template::Toolkit syntax can be used in OTRS templates. This section describes some example use cases and OTRS extensions to the Template::Toolkit syntax.

7.1. Template Commands

7.1.1. Inserting dynamic data

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

7.1.1.1. `[% Data.Name %]`

If data parameters are given to the templates by the application module, these data can be output to the template. `[% Data.Name %]` is the most simple, but also most dangerous one. It will insert the data parameter whose name is `Name` into the template as it is, without further processing.

> **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 `[% Data.Name | html %]` (in HTML) or `[% Data.Name | uri %]` (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"]
```

If you have data entries with complex names containing special characters, you cannot use the dot (`.`) notation to access this data. Use `item()` instead: `[% Data.item('Complex-name') %]`.

7.1.1.2. `[% Data.Name | html %]`

This command has the same function as the previous one, but it performs HTML quoting on the data as it is inserted to the template.

```
The name of the author is [% Data.Name | html %].
```

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: [% Data.Name | truncate(20) | html %].
```

7.1.1.3. [% Data.Name | uri %]

This command 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=[% Data.File | uri %]">[% Data.File | truncate(110)
 | html %]</a>
```

7.1.1.4. [% Data.Name | JSON %]

This command outputs a string or another data structure as a JavaScript JSON string.

```
var Text = [% Data.Text | JSON %];
```

Please note that the filter notation will only work for simple strings. To output complex data as JSON, please use it as a function:

```
var TreeData = [% JSON(Data.TreeData) %];
```

7.1.1.5. [% Env() %]

Inserts environment variables provided by the `LayoutObject`. Some examples:

```
The current user name is: [% Env("UserFullname") %]

Some other common predefined variables are:

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

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.

Don't forget to add the `| html` filter where appropriate.

7.1.1.6. [% Config() %]

Inserts 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.

Don't forget to add the `| html` filter where appropriate.

7.1.2. Localization Commands

7.1.2.1. `[% Translate() %]`

Translates a string into the current user's selected language. If no translation is found, the original string will be used.

```
Translate this text: [% Translate("Help") | html %]
```

You can also translate dynamic data by using `Translate` as a filter:

```
Translate data from the application: [% Data.Type | Translate | html %]
```

You can also specify one or more parameters (`%s`) inside of the string which should be replaced with dynamic data:

```
Translate this text and insert the given data: [% Translate("Change %s settings", Data.Type)
 | html %]
```

Strings in JavaScript can be translated and processed with the JSON filter.

```
var Text = [% Translate("Change %s settings", Data.Type) | JSON %];
```

7.1.2.2. `[% Localize() %]`

Outputs data according to the current language/locale.

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. `[% Localize() %]` abstracts this away from the templates. Let's see an example:

```
[% Data.CreateTime | Localize("TimeLong") %]
# 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 `[% Localize() %]`. Then it will be output it according to the date/time definition of the current locale.

The data passed to `[% Localize() %]` 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.

There are different possible date/time output formats: `TimeLong` (full date/time), `TimeShort` (no seconds) and `Date` (no time).

```
[% Data.CreateTime | Localize("TimeLong") %]
# Result for US English locale:
06/09/2010 15:45:41

[% Data.CreateTime | Localize("TimeShort") %]
# Result for US English locale:
06/09/2010 15:45

[% Data.CreateTime | Localize("Date") %]
# Result for US English locale:
06/09/2010
```

Also the output of human readable file sizes is available as an option `Localize('Filesize')` (just pass the raw file size in bytes).

```
[% Data.Filesize | Localize("Filesize") %]
# Result for US English locale:
23 MB
```

7.1.2.3. `[% ReplacePlaceholders() %]`

Replaces placeholders (%s) in strings with passed parameters.

In certain cases, you might want to insert HTML code in translations, instead of placeholders. On the other hand, you also need to take care of sanitization, since translated strings should not be trusted as-is. For this, you can first translate the string, pass it through the HTML filter and finally replace placeholders with static (safe) HTML code.

```
[% Translate("This is %s.") | html | ReplacePlaceholders('<strong>bold text</strong>') %]
```

Number of parameters to `ReplacePlaceholders()` filter should match number of placeholders in the original string.

You can also use `[% ReplacePlaceholders() %]` in function format, in case you are not translating anything. In this case, first parameter is the target string, and any found placeholders in it are substituted with subsequent parameters.

```
[% ReplacePlaceholders("This string has both %s and %s.", '<strong>bold text</strong>,
 '<em>italic text</em>') %]
```

7.1.3. Template Processing Commands

7.1.3.1. Comment

Lines starting with a # at the beginning of will not be shown in the html output. This can be used both for commenting the Template code or for disabling parts of it.

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

7.1.3.2. [% InsertTemplate("Copyright.tt") %]

Warning

Please note that the `InsertTemplate` command was added to provide better backwards compatibility to the old DTL system. It might be deprecated in a future version of OTRS and removed later. If you don't use block commands in your included template, you don't need `InsertTemplate` and can use standard Template::Toolkit syntax like `INCLUDE/PROCESS` instead.

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

```
# include Copyright.tt
[% InsertTemplate("Copyright") %]
```

Please note this is not the same as Template::Toolkit's `[% INCLUDE %]` command, which just processes the referenced template. `[% InsertTemplate() %]` actually adds the content of the referenced template into the current template, so that it can be processed together. That makes it possible for the embedded template to access the same environment/data as the main template.

7.1.3.3. [% RenderBlockStart %] / [% RenderBlockEnd %]

Warning

Please note that the blocks commands were added to provide better backwards compatibility to the old DTL system. They might be deprecated in a future version of OTRS and removed later. We advise you to develop any new code without using the blocks commands. You can use standard Template::Toolkit syntax like `IF/ELSE`, `LOOP`s and other helpful things for conditional template output.

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>[% Translate("Name") | html %]</th>
            <th>[% Translate("Type") | html %]</th>
            <th>[% Translate("Comment") | html %]</th>
            <th>[% Translate("Validity") | html %]</th>
            <th>[% Translate("Changed") | html %]</th>
            <th>[% Translate("Created") | html %]</th>
        </tr>
    </thead>
    <tbody>
[% RenderBlockStart("NoDataFoundMsg") %]
        <tr>
            <td colspan="6">
                [% Translate("No data found.") | html %]
```

```
            </td>
        </tr>
[% RenderBlockEnd("NoDataFoundMsg") %]
[% RenderBlockStart("OverviewResultRow") %]
        <tr>
            <td><a class="AsBlock" href="[% Env("Baselink") %]Action=[% Env("Action")
 %];Subaction=Change;ID=[% Data.ID | uri %]">[% Data.Name | html %]</a></td>
            <td>[% Translate(Data.TypeName) | html %]</td>
            <td title="[% Data.Comment | html %]">[% Data.Comment | truncate(20) | html %]</
td>
            <td>[% Translate(Data.Valid) | html %]</td>
            <td>[% Data.ChangeTime | Localize("TimeShort") %]</td>
            <td>[% Data.CreateTime | Localize("TimeShort") %]</td>
        </tr>
[% RenderBlockEnd("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 = $Kernel::OM->Get('Kernel::System::State)->StateList(
    UserID => 1,
    Valid  => 0,
);

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

    # get valid list
    my %ValidList = $Kernel::OM->Get('Kernel::System::Valid')->ValidList();
    for my $ListKey ( sort { $List{$a} cmp $List{$b} } keys %List ) {

        my %Data = $Kernel::OM->Get('Kernel::System::State)->StateGet( ID => $ListKey );
        $Kernel::OM->Get('Kernel::Output::HTML::Layout')->Block(
            Name => 'OverviewResultRow',
            Data => {
                Valid => $ValidList{ $Data{ValidID} },
                %Data,
            },
        );
    }
}

# otherwise a no data found msg is displayed
else {
    $Kernel::OM->Get('Kernel::Output::HTML::Layout')->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.

7.1.4. `[% WRAPPER JSOnDocumentComplete %]...[% END %]`

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 action="[% Env("CGIHandle") %]" method="post" enctype="multipart/form-data"
 name="MoveTicketToQueue" class="Validate PreventMultipleSubmits" id="MoveTicketToQueue">
    <input type="hidden" name="Action"       value="[% Env("Action") %]"/>
    <input type="hidden" name="Subaction"    value="MoveTicket"/>

    ...

    <div class="Content">
        <fieldset class="TableLike FixedLabel">
            <label class="Mandatory" for="DestQueueID"><span class="Marker">*</span> [%
 Translate("New Queue") | html %]:</label>
            <div class="Field">
                [% Data.MoveQueuesStrg %]
                <div id="DestQueueIDError" class="TooltipErrorMessage" ><p>[%
 Translate("This field is required.") | html %]</p></div>
                <div id="DestQueueIDServerError" class="TooltipErrorMessage"><p>[%
 Translate("This field is required.") | html %]</p></div>
[% WRAPPER JSOnDocumentComplete %]
<script type="text/javascript">
    $('#DestQueueID').bind('change', function (Event) {
        $('#NoSubmit').val('1');
        Core.AJAX.FormUpdate($('#MoveTicketToQueue'), 'AJAXUpdate', 'DestQueueID',
 ['NewUserID', 'OldUserID', 'NewStateID', 'NewPriorityID' [% Data.DynamicFieldNamesStrg
 %]]);
    });
</script>
[% END %]
                    </div>
                    <div class="Clear"></div>
```

This snippet creates a small form and puts an onchange handler on the `<select>` element which triggers an AJAX based form update.

Why is it necessary to enclose the JavaScript code in `[% WRAPPER JSOnDocumentComplete %]...[% END %]`? 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 `[% WRAPPER JSOnDocumentComplete %]...[% END %]` 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 `[% WRAPPER JSOnDocumentComplete %]...[% END %]` 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.

7.2. Using a template file

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

```
# render AdminState.tt
$Output .= $Kernel::OM->Get('Kernel::Output::HTML::Layout')->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.

8. Creating Your Own Themes

You can create your own themes so as to use the layout you like in the OTRS web frontend. To create custom themes, you should customize the output templates to your needs. More information on the syntax and structure of output templates can be found in the templating section.

As an example, perform the following steps to create a new theme called "Company":

1. Create a directory called `Kernel/Output/HTML/Templates/Company` and copy all files that you like to change from `Kernel/Output/HTML/Templates/Standard` into the new folder.

   ### Important

   Only copy over the files you're planning to change. OTRS will automatically get the missing files from the Standard theme. This will make upgrading at a later stage much easier.

2. Customize the files in the directory `Kernel/Output/HTML/Templates/Company` and change the layout to your needs.

3. To activate the new theme, add them in SysConfig under Frontend::Themes.

Now the new theme should be usable. You can select it via your personal preferences.

Warning

Do not change the theme files shipped with OTRS, since these changes will be lost after an update. Create your own themes only by performing the steps described above.

9. Localization / Translation Mechanism

There are four steps needed to translate / localize software: marking localizable strings in the source files, generating the translation database/file, the translation process itself, and the usage of translated data within the code.

9.1. Marking translatable strings in the source files

In Perl code, all literal strings to be translated are automatically marked for translation: `$LanguageObject->Translate('My string %s', $Data)` will mark 'My string %s' for translation. If you need to mark strings, but NOT translate them in the code yet, you can use the NOOP method `Kernel::Langauge::Translatable()`.

```
package MyPackage;

use strict;
use warnings;
```

```
use Kernel::Language (qw(Translatable));

...

my $UntranslatedString = Translatable('My string %s');
```

In Template files, all literal strings enclosed in the `Translate()`-Tag are automatically marked for extraction: `[% Translate('My string %s', Data.Data )%]`.

In SysConfig and Database XML files you can mark strings for extraction with the `Translatable="1"` attribute.

```
# Database XML
<Insert Table="groups">
    <Data Key="id" Type="AutoIncrement">1</Data>
    ...
    <Data Key="comments" Type="Quote" Translatable="1">Group for default access.</Data>
    ...
</Insert>

# SysConfig XML
<Setting>
    <Option SelectedID="0">
        <Item Key="0" Translatable="1">No</Item>
        <Item Key="1" Translatable="1">Yes</Item>
    </Option>
</Setting>
```

9.2. Collecting translatable strings into the translation database

The console command `otrs.Console.pl Dev::Tools::TranslationsUpdate` is used to extract all translatable strings from the source files. These will be collected and written into the translation files.

For the OTRS framework and all extension modules that also use Transifex for managing the translations, .pot and .po files are written. These files are used to push the translatable strings to Transifex and pull the translations from there.

But OTRS requires the translations to be in Perl files for speed reasons. These files will also be generated by `otrs.Console.pl Dev::Tools::TranslationsUpdate`. There are two different translation cache file types which are used in the following order. If a word/ sentence is redefined in a translation file, the last definition will be used.

1. Default Framework Translation File

   `Kernel/Language/$Language.pm`

2. Custom Translation File

   `Kernel/Language/$Language_Custom.pm`

9.2.1. 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;
```

9.2.2. 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;
```

9.3. The translation process itself

OTRS uses Transifex to manage the translation process. Please see this section for details.

9.4. Using the translated data from the code

You can use the method `$LanguageObject->Translate()` to translate strings at runtime from Perl code, and the `Translate()`-Tag in templates.

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/Templates/Standard
Kernel/Config
Kernel/Config/Files
Kernel/Config/Files/XML/
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/XML/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="2.0" init="Application">
    <Setting Name="Frontend::Module###AgentHelloWorld" Required="1" Valid="1">
        <Description Translatable="1">FrontendModuleRegistration for HelloWorld module.</
Description>
        <Navigation>Frontend::Agent::ModuleRegistration</Navigation>
        <Value>
            <Item ValueType="FrontendRegistration">
                <Hash>
                    <Item Key="Group">
                        <Array>
                            <Item>users</Item>
                        </Array>
                    </Item>
                    <Item Key="GroupRo">
                        <Array>
                        </Array>
                    </Item>
                    <Item Key="Description" Translatable="1">HelloWorld.</Item>
                    <Item Key="Title" Translatable="1">HelloWorld</Item>
                    <Item Key="NavBarName">HelloWorld</Item>
                </Hash>
            </Item>
        </Value>
    </Setting>
    <Setting Name="Loader::Module::AgentHelloWorld###002-Filename" Required="0" Valid="1">
        <Description Translatable="1">Loader module registration for the agent interface.</
Description>
```

```
        <Navigation>Frontend::Agent::ModuleRegistration::Loader</Navigation>
        <Value>
            <Hash>
                <Item Key="CSS">
                    <Array>
                    </Array>
                </Item>
                <Item Key="JavaScript">
                    <Array>
                    </Array>
                </Item>
            </Hash>
        </Value>
    </Setting>
    <Setting Name="Frontend::Navigation###AgentHelloWorld###002-Filename" Required="0"
 Valid="1">
        <Description Translatable="1">Main menu item registration.</Description>
        <Navigation>Frontend::Agent::ModuleRegistration::MainMenu</Navigation>
        <Value>
            <Array>
                <DefaultItem ValueType="FrontendNavigation">
                    <Hash>
                    </Hash>
                </DefaultItem>
                <Item>
                    <Hash>
                        <Item Key="Group">
                            <Array>
                                <Item>users</Item>
                            </Array>
                        </Item>
                        <Item Key="GroupRo">
                            <Array>
                            </Array>
                        </Item>
                        <Item Key="Description" Translatable="1">HelloWorld.</Item>
                        <Item Key="Name" Translatable="1">HelloWorld</Item>
                        <Item Key="Link">Action=AgentHelloWorld</Item>
                        <Item Key="LinkOption"></Item>
                        <Item Key="NavBar">HelloWorld</Item>
                        <Item Key="Type">Menu</Item>
                        <Item Key="Block"></Item>
                        <Item Key="AccessKey"></Item>
                        <Item Key="Prio">8400</Item>
                    </Hash>
                </Item>
            </Array>
        </Value>
    </Setting>
</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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

package Kernel::Modules::AgentHelloWorld;
```

```
use strict;
use warnings;

# Frontend modules are not handled by the ObjectManager.
our $ObjectManagerDisabled = 1;

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

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

    return $Self;
}

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

    my $HelloWorldObject = $Kernel::OM->Get('Kernel::System::HelloWorld');
    my $LayoutObject     = $Kernel::OM->Get('Kernel::Output::HTML::Layout');

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

    # build output
    my $Output = $LayoutObject->Header(Title => "HelloWorld");
    $Output   .= $LayoutObject->NavigationBar();
    $Output   .= $LayoutObject->Output(
        Data         => \%Data,
        TemplateFile => 'AgentHelloWorld',
    );
    $Output   .= $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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

package Kernel::System::HelloWorld;

use strict;
use warnings;

# list your object dependencies (e.g. Kernel::System::DB) here
our @ObjectDependencies = (
    # 'Kernel::System::DB',
);

=head1 NAME

HelloWorld - Little "Hello World" module

=head1 DESCRIPTION
```

```
Little OTRS module that displays the text 'Hello World' when called up.

=head2 new()

Create an object. Do not use it directly, instead use:

    my $HelloWorldObject = $Kernel::OM->Get('Kernel::System::HelloWorld');

=cut

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

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

    return $Self;
}

=head2 GetHelloWorldText()

Return the "Hello World" text.

    my $HelloWorldText = $HelloWorldObject->GetHelloWorldText();

=cut

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

    # Get the DBObject from the central object manager
    # my $DBObject = $Kernel::OM->Get('Kernel::System::DB');

    my $HelloWorld = $Self->_FormatHelloWorldText(
        String => 'Hello World',
    );

    return $HelloWorld;
}

=begin Internal:

=head2 _FormatHelloWorldText()

Format "Hello World" text to uppercase

    my $HelloWorld = $Self->_FormatHelloWorldText(
        String => 'Hello World',
    );

=cut

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

    my $HelloWorld = uc $Param{String};

    return $HelloWorld;

}

=end Internal:

1;
```

1.5. 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/Templates/Standard/AgentHelloWorld.tt - 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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --
<h1>[% Translate("Overview") | html %]: [% Translate("HelloWorld") %]</h1>
<p>
    [% Data.HelloWorldText | Translate() | html %]
</p>
```

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 in `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) 2001-2021 OTRS AG, https://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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 Synchronization Module

There is an LDAP authentication synchronization module which come with the OTRS framework. It is also possible to develop your own authentication modules. The authentication synchronization 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 synchronization module. Save it under `Kernel/System/Auth/Sync/CustomAuthSync.pm`. You just need 2 functions: `new()` and `Sync()`. Return 1, then the synchronization 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) 2001-2021 OTRS AG, https://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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 synchronization 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 synchronization 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 synchronization 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) 2001-2021 OTRS AG, https://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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 authentication 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. Following, there is an example of a customer-user preferences module. 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) 2001-2021 OTRS AG, https://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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 synchronization 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) 2001-2021 OTRS AG, https://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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 mod-

ules 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 synchronization 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) 2001-2021 OTRS AG, https://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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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. Following, there is 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 SLA 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

There is the need to activate 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 global 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 `.tt` files directly. There are three good reasons for that. When the same adaptation 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 three different kinds of output filters. They are active at different stages of the generation of HTML content.

2.3.2.1. `FilterElementPost`

These filters allow to modify the output of a template after it was rendered.

To translate content, you can run `$LayoutObject->Translate()` directly. If you need other template features, just define a small template file for your output filter and use it to render your content before injecting it into the main data. It can also be helpful to use jQuery DOM operations to reorder/replace content on the screen in some cases instead of using regular expressions. In this case you would inject the new code somewhere in the page as invisible content (e. g. with the class Hidden), and then move it with jQuery to the correct location in the DOM and show it.

To make using post output filters easier, there is also a mechanism to request HTML comment hooks for certain templates/blocks. You can add in your module config XML like:

```
<ConfigItem
Name="Frontend::Template::GenerateBlockHooks###100-OTRSBusiness-ContactWithData"
Required="1" Valid="1">
    <Description Translatable="1">Generate HTML comment hooks for
the specified blocks so that filters can use them.</Description>
    <Group>OTRSBusiness</Group>
    <SubGroup>Core</SubGroup>
    <Setting>
        <Hash>
            <Item Key="AgentTicketZoom">
                <Array>
                    <Item>CustomerTable</Item>
                </Array>
```

```
            </Item>
        </Hash>
    </Setting>
</ConfigItem>
```

This will cause the block `CustomerTable` in `AgentTicketZoom.tt` to be wrapped in HTML comments each time it is rendered:

```
<!--HookStartCustomerTable-->
... block output ...
<!--HookEndCustomerTable-->
```

With this mechanism every package can request just the block hooks it needs, and they are consistently rendered. These HTML comments can then be used in your output filter for easy regular expression matching.

2.3.2.2. `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.

2.3.2.3. `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.

2.3.2.4. Code example

See package `TemplateModule`.

2.3.2.5. Configuration example

See package `TemplateModule`.

2.3.2.6. Use Cases

2.3.2.6.1. Show additional ticket attributes in `AgentTicketZoom`

This can be achieved with a `FilterElementPost` output filter.

2.3.2.6.2. 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.

2.3.2.6.3. Create links within plain text article bodies

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

2.3.2.6.4. 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 a `FilterContent` output filter.

2.3.2.7. Caveats and Warnings

Every `FilterElementPost` output filter is constructed and run for every configured 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.

2.3.2.8. Best Practices

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

2.3.2.9. Release Availability

The output filters are available since OTRS 2.4. The type `FilterElementPre` was dropped with OTRS 5.

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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 a 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 used via the use keyword.

```
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;
}
```

The 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 and a time drop down. To get the values shown in the drop down, some operations are needed. In this case `StateList` and `GetAllQueues` are called.

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

- `Name`

  The label in the web interface.

- `UseAsXvalue`

  This attribute can be used on the x-axis.

- `UseAsValueSeries`

  This attribute can be used on the y-axis.

- `UseAsRestriction`

  This attribute can be used for restrictions.

- `Element`

  The HTML field name.

- `Block`

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

- Values

  The values shown in the attribute.

Hint: If you install this sample and you configure 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 array references. 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 contain 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 hash references 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 array reference 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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="utf-8" ?>
<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;
}
```

The 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 hash references with these key-value pairs:

- `Frontend`

  The label in the web interface.

- Name

  The HTML field name.

- 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 in `%Param` a key `TypeIDs` and a key `QueueIDs` exist (see attributes in `Param` method) and their values are array references. The returned data consists of three parts: Two array references and an array. In the first array reference the title for the statistic is stored, the second array reference 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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;
```

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 ticket number 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`. A ticket event module needs only 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 available in OTRS 6.0:

- TicketCreate

- TicketDelete
- TicketTitleUpdate
- TicketUnlockTimeoutUpdate
- TicketQueueUpdate
- TicketTypeUpdate
- TicketServiceUpdate
- TicketSLAUpdate
- TicketCustomerUpdate
- TicketPendingTimeUpdate
- TicketLockUpdate
- TicketArchiveFlagUpdate
- TicketStateUpdate
- TicketOwnerUpdate
- TicketResponsibleUpdate
- TicketPriorityUpdate
- HistoryAdd
- HistoryDelete
- TicketAccountTime
- TicketMerge
- TicketSubscribe
- TicketUnsubscribe
- TicketFlagSet
- TicketFlagDelete
- EscalationResponseTimeNotifyBefore
- EscalationUpdateTimeNotifyBefore
- EscalationSolutionTimeNotifyBefore
- EscalationResponseTimeStart
- EscalationUpdateTimeStart
- EscalationSolutionTimeStart
- EscalationResponseTimeStop

- EscalationUpdateTimeStop
- EscalationSolutionTimeStop
- NotificationNewTicket
- NotificationFollowUp
- NotificationLockTimeout
- NotificationOwnerUpdate
- NotificationResponsibleUpdate
- NotificationAddNote
- NotificationMove
- NotificationPendingReminder
- NotificationEscalation
- NotificationEscalationNotifyBefore
- NotificationServiceUpdate

Article events available in OTRS 6.0:

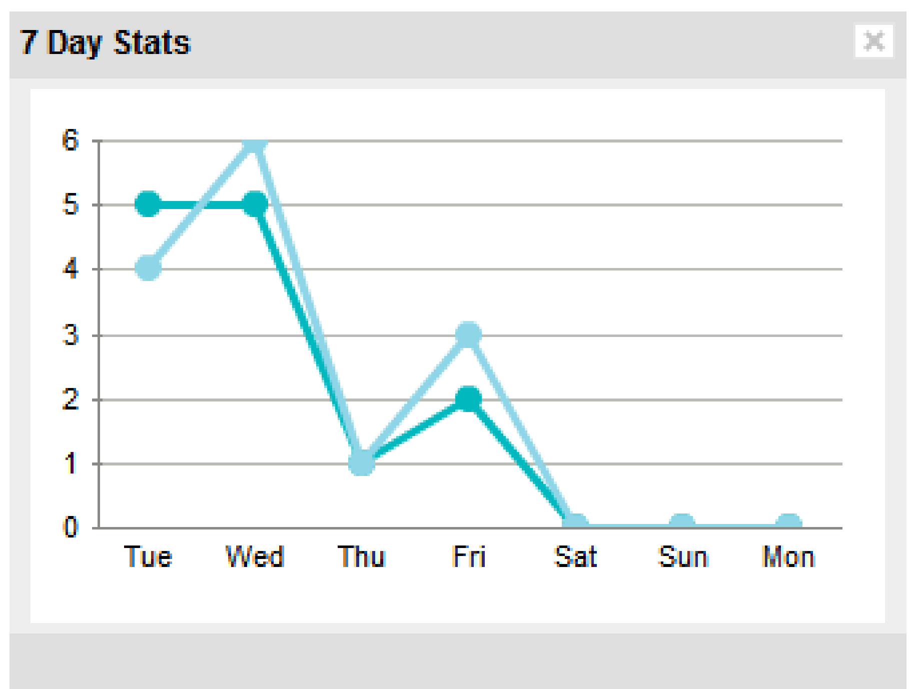
- ArticleCreate
- ArticleUpdate
- ArticleSend
- ArticleBounce
- ArticleAgentNotification
- ArticleCustomerNotification
- ArticleAutoResponse
- ArticleFlagSet
- ArticleFlagDelete
- ArticleAgentNotification
- ArticleCustomerNotification

2.4. Frontend Modules

2.4.1. Dashboard Module

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

Figure 3.1. Dashboard Widget




```
# --
# Kernel/Output/HTML/DashboardTicketStatsGeneric.pm - message of the day
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 web server (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 version 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 menus 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`. Following, there is an example of a notify module. Save it under `Kernel/Output/HTML/TicketNotificationCustom.pm`. You just need 2 functions: `new()` and `Run()`.

```
# --
# Kernel/Output/HTML/NotificationCustom.pm
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 an external tool if parameters (e.g. `FreeTextField`) have been set.

2.4.2.4. Release Availability

Name	Release
NotificationAgentOnline	2.0
NotificationAgentTicketEscalation	2.0
NotificationCharsetCheck	1.2
NotificationCustomerOnline	2.0
NotificationUIDCheck	1.2

2.4.3. Ticket Menu Module

Ticket menu modules are used to display an additional link in the menu above a ticket. You can write and register your own ticket menu module. There are 4 ticket menus (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`. Following, there is an example of a ticket menu module. Save it under `Kernel/Output/HTML/TicketMenuCustom.pm`. You just need 2 functions: `new()` and `Run()`.

```
# --
# Kernel/Output/HTML/TicketMenuCustom.pm
# Copyright (C) 2001-2021 OTRS AG, https://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 (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 deal with the data in the Remote System format. It is not recommended to do any data transformation in this modules, as the Mapping layer is the responsible to perform any data transformation needed during the communication. An exception of this is the data conversion that is required specifically by for the transport e.g. XML or JSON from / to Perl conversions.

2.5.1.1. Transport backend

Next we will show 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 and HTTP REST transports. If the planned web service can use HTTP SOAP or HTTP SOAP there is no need to create a new network transport module, instead we recommend to take a look into HTTP SOAP or HTTP REST configurations to check their settings and how it can be tuned according to the remote system.

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 loop-back interface.

```
# --
# Kernel/GenericInterface/Transport/HTTP/Test.pm - GenericInterface network transport
 interface for testing
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

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

use strict;
use warnings;

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

# prevent 'Used once' warning for Kernel::OM
use Kernel::System::ObjectManager;

our $ObjectManagerDisabled = 1;
```

This is common header that can be found in common OTRS modules. The class/package name is declared via the package keyword. Transports can not be instantiated by the Object Manager.

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

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

    for my $Needed (qw( 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 only objects of other classes not handled by the object manager 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::OM->Get('Kernel::System::Web::Request');

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

    # special handling for empty post request
    if ( scalar 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 from 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 parses 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 sends the response back to the Remote System for the requested operation.

For this particular example we return a 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 its 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 {    ## no critic
    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);

    #print $Request->as_string();
    #print $Response->as_string();

    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.

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 OTRS to the external systems, and vice versa. 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 from 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 prepares 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 performs the action and sends the response back to OTRS with the data in remote system format
4. The data is mapped form remote system format (again) to the OTRS format
5. OTRS processes 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 probable that you don't need to develop a custom mapping module. Please check *Simple* mapping module (`Kernel/GenericInterface/Mapping/Simple.pm`) and its on-line documentation before continue.

If *Simple* mapping module does not match your needs then we will show 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

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

use strict;
use warnings;

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

our $ObjectManagerDisabled = 1;
```

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. Mappings can not be instantiated by the Object Manager.

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

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

    # check needed params
    for my $Needed (qw(DebuggerObject MappingConfig)) {
        if ( !$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 only objects of other classes not handled by the object manager 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 if the structure 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 ( sort keys %{ $Param{Data} } ) {
        $ReturnData->{$Key} = uc $Param{Data}->{$Key};
    }

    return $ReturnData;
}

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

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

    return $ReturnData;
}

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

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

    return $ReturnData;
}
```

This are the helper functions that actually performs the string conversions.

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 show 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

package Kernel::GenericInterface::Invoker::Test::Test;

use strict;
use warnings;

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

# prevent 'Used once' warning for Kernel::OM
use Kernel::System::ObjectManager;

our $ObjectManagerDisabled = 1;
```

This is common header that can be found in common OTRS modules. The class/package name is declared via the package keyword. Invokers can not be instantiated by the Object Manager.

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

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

    # check needed params
    if ( !$Param{DebuggerObject} ) {
        return {
            Success      => 0,
            ErrorMessage => "Got no DebuggerObject!"
        };
    }
```

```
    $Self->{DebuggerObject} = $Param{DebuggerObject};

    return $Self;
}
```

The constructor new creates a new instance of the class. According to the coding guidelines only objects of other classes not handled by the object manager 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} = $Kernel::OM->Get('Kernel::System::Time')->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 an error message as the passed Summary.

Also this example appends the word "Test" to the parameter Action and if GetSystemTime 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). The 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).

Note

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 a call to a Core Module, update the database, send an error, etc.

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 to correctly execute the action. After the action is performed, OTRS sends a defined confirmation to the external system.

2.5.4.1. Operation backend

Next we will show 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

package Kernel::GenericInterface::Operation::Test::Test;

use strict;
use warnings;

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

our $ObjectManagerDisabled = 1;
```

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. Operations can not be instantiated by the Object Manager.

```
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 only objects of other classes not handled by the object manager 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.

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>
```

2.5.4.1.3. Unit Test Example

Unit Test for Generic Interface operations does not differs from other unit tests but it is needed to consider testing locally, but also simulating a remote connection. It is a good practice to test both separately since results could be slightly different.

To learn more about unit tests, please take a look to the Unit Test Chapter.

The following is just the starting point for a unit test:

```
# --
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

## no critic (Modules::RequireExplicitPackage)
use strict;
use warnings;
use utf8;

use vars (qw($Self));

use Kernel::GenericInterface::Debugger;
use Kernel::GenericInterface::Operation::Test::Test;

use Kernel::System::VariableCheck qw(:all);

# Skip SSL certificate verification (RestoreDatabase must not be used in this test).
$Kernel::OM->ObjectParamAdd(
    'Kernel::System::UnitTest::Helper' => {
        SkipSSLVerify => 1,
    },
);
my $Helper = $Kernel::OM->Get('Kernel::System::UnitTest::Helper');

# get a random number
my $RandomID = $Helper->GetRandomNumber();

# create a new user for current test
my $UserLogin = $Helper->TestUserCreate(
    Groups => ['users'],
);
my $Password = $UserLogin;

my $UserID = $Kernel::OM->Get('Kernel::System::User')->UserLookup(
    UserLogin => $UserLogin,
```

```
);

# set web-service name
my $WebserviceName = '-Test-' . $RandomID;

# create web-service object
my $WebserviceObject = $Kernel::OM->Get('Kernel::System::GenericInterface::Webservice');
$Self->Is(
    'Kernel::System::GenericInterface::Webservice',
    ref $WebserviceObject,
    "Create web service object",
);

my $WebserviceID = $WebserviceObject->WebserviceAdd(
    Name   => $WebserviceName,
    Config => {
        Debugger => {
            DebugThreshold => 'debug',
        },
        Provider => {
            Transport => {
                Type => '',
            },
        },
    },
    ValidID => 1,
    UserID  => 1,
);
$Self->True(
    $WebserviceID,
    "Added Web Service",
);

# get remote host with some precautions for certain unit test systems
my $Host = $Helper->GetTestHTTPHostname();

my $ConfigObject = $Kernel::OM->Get('Kernel::Config');

# prepare web-service config
my $RemoteSystem =
    $ConfigObject->Get('HttpType')
    . '://'
    . $Host
    . '/'
    . $ConfigObject->Get('ScriptAlias')
    . '/nph-genericinterface.pl/WebserviceID/'
    . $WebserviceID;

my $WebserviceConfig = {
    Description =>
        'Test for Ticket Connector using SOAP transport backend.',
    Debugger => {
        DebugThreshold => 'debug',
        TestMode       => 1,
    },
    Provider => {
        Transport => {
            Type   => 'HTTP::SOAP',
            Config => {
                MaxLength => 10000000,
                NameSpace => 'http://otrs.org/SoapTestInterface/',
                Endpoint  => $RemoteSystem,
            },
        },
        Operation => {
            Test => {
                Type => 'Test::Test',
            },
        },
    },
    Requester => {
        Transport => {
```

```
            Type   => 'HTTP::SOAP',
            Config => {
                NameSpace => 'http://otrs.org/SoapTestInterface/',
                Encoding  => 'UTF-8',
                Endpoint  => $RemoteSystem,
            },
        },
        Invoker => {
            Test => {
                Type => 'Test::TestSimple'
                ,    # requester needs to be Test::TestSimple in order to simulate a request
 to a remote system
            },
        },
    },
};

# update web-service with real config
# the update is needed because we are using
# the WebserviceID for the Endpoint in config
my $WebserviceUpdate = $WebserviceObject->WebserviceUpdate(
    ID      => $WebserviceID,
    Name    => $WebserviceName,
    Config  => $WebserviceConfig,
    ValidID => 1,
    UserID  => $UserID,
);
$Self->True(
    $WebserviceUpdate,
    "Updated Web Service $WebserviceID - $WebserviceName",
);

# debugger object
my $DebuggerObject = Kernel::GenericInterface::Debugger->new(
    DebuggerConfig => {
        DebugThreshold => 'debug',
        TestMode       => 1,
    },
    WebserviceID      => $WebserviceID,
    CommunicationType => 'Provider',
);
$Self->Is(
    ref $DebuggerObject,
    'Kernel::GenericInterface::Debugger',
    'DebuggerObject instantiate correctly',
);

# define test cases
my @Tests = (
    {
        Name           => 'Test case name',
        SuccessRequest => 1,                  # 1 or 0
        RequestData    => {

            # ... add test data
        },
        ExpectedReturnLocalData => {
            Data => {

                # ... add expected local results
            },
            Success => 1,                     # 1 or 0
        },
        ExpectedReturnRemoteData => {
            Data => {

                # ... add expected remote results
            },
            Success => 1,                     # 1 or 0
        },
        Operation => 'Test',
    },
```

```
    # ... add more test cases
);

TEST:
for my $Test (@Tests) {

    # create local object
    my $LocalObject = "Kernel::GenericInterface::Operation::Test::$Test->{Operation}"->new(
        DebuggerObject => $DebuggerObject,
        WebserviceID   => $WebserviceID,
    );

    $Self->Is(
        "Kernel::GenericInterface::Operation::Test::$Test->{Operation}",
        ref $LocalObject,
        "$Test->{Name} - Create local object",
    );

    my %Auth = (
        UserLogin => $UserLogin,
        Password  => $Password,
    );
    if ( IsHashRefWithData( $Test->{Auth} ) ) {
        %Auth = %{ $Test->{Auth} };
    }

    # start requester with our web-service
    my $LocalResult = $LocalObject->Run(
        WebserviceID => $WebserviceID,
        Invoker      => $Test->{Operation},
        Data         => {
            %Auth,
            %{ $Test->{RequestData} },
        },
    );

    # check result
    $Self->Is(
        'HASH',
        ref $LocalResult,
        "$Test->{Name} - Local result structure is valid",
    );

    # create requester object
    my $RequesterObject = $Kernel::OM->Get('Kernel::GenericInterface::Requester');
    $Self->Is(
        'Kernel::GenericInterface::Requester',
        ref $RequesterObject,
        "$Test->{Name} - Create requester object",
    );

    # start requester with our web-service
    my $RequesterResult = $RequesterObject->Run(
        WebserviceID => $WebserviceID,
        Invoker      => $Test->{Operation},
        Data         => {
            %Auth,
            %{ $Test->{RequestData} },
        },
    );

    # check result
    $Self->Is(
        'HASH',
        ref $RequesterResult,
        "$Test->{Name} - Requester result structure is valid",
    );

    $Self->Is(
        $RequesterResult->{Success},
        $Test->{SuccessRequest},
```

```
        "$Test->{Name} - Requester successful result",
    );

    # ... add tests for the results
}

# delete web service
my $WebserviceDelete = $WebserviceObject->WebserviceDelete(
    ID     => $WebserviceID,
    UserID => $UserID,
);
$Self->True(
    $WebserviceDelete,
    "Deleted Web Service $WebserviceID",
);

# also delete any other added data during the this test, since RestoreDatabase must not be
 used.

1;
```

2.5.4.1.4. WSDL Extension Example

WSDL files contain the definitions of the web services and its operations for SOAP messages, in case we will extend `development/webservices/GenericTickeConnectorSOAP.wsdl` in some places:

Port Type:

```
    <wsdl:portType name="GenericTicketConnector_PortType">
        <!-- ... -->
        <wsdl:operation name="Test">
            <wsdl:input message="tns:TestRequest"/>
            <wsdl:output message="tns:TestResponse"/>
        </wsdl:operation>
    <!-- ... -->
```

Binding:

```
    <wsdl:binding name="GenericTicketConnector_Binding"
 type="tns:GenericTicketConnector_PortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <!-- ... -->
        <wsdl:operation name="Test">
            <soap:operation soapAction="http://www.otrs.org/TicketConnector/Test"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <!-- ... -->
    </wsdl:binding>
```

Type:

```
    <wsdl:types>
        <xsd:schema targetNamespace="http://www.otrs.org/TicketConnector/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <!-- ... -->
            <xsd:element name="Test">
                <xsd:complexType>
                    <xsd:sequence>
```

```
                           <xsd:element minOccurs="0" name="Param1" type="xsd:string"/>
                           <xsd:element minOccurs="0" name="Param2"
 type="xsd:positiveInteger"/>
                   </xsd:sequence>
               </xsd:complexType>
           </xsd:element>
           <xsd:element name="TestResponse">
               <xsd:complexType>
                   <xsd:sequence>
                       <xsd:element maxOccurs="unbounded" minOccurs="1" name="Attribute1"
 type="xsd:string"/>
                   </xsd:sequence>
               </xsd:complexType>
           </xsd:element>
       <!-- ... -->
       </xsd:schema>
   </wsdl:types>
```

Message:

```
   <!-- ... -->
   <wsdl:message name="TestRequest">
       <wsdl:part element="tns:Test" name="parameters"/>
   </wsdl:message>
   <wsdl:message name="TestResponse">
       <wsdl:part element="tns:TestResponse" name="parameters"/>
   </wsdl:message>
   <!-- ... -->
```

2.5.4.1.5. WADL Extension Example

WADL files contain the definitions of the web services and its operations for REST interface, add a new resource to `development/webservices/GenericTickeConnectorREST.wadl`.

```
  <resources base="http://localhost/otrs/nph-genericinterface.pl/Webservice/
GenericTicketConnectorREST">
   <!-- ... -->
   <resource path="Test" id="Test">
     <doc xml:lang="en" title="Test"/>
       <param name="Param1" type="xs:string" required="false" default="" style="query"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"/>
       <param name="Param2" type="xs:string" required="false" default="" style="query"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"/>
       <method name="GET" id="GET_Test">
         <doc xml:lang="en" title="GET_Test"/>
         <request/>
         <response status="200">
           <representation mediaType="application/json; charset=UTF-8"/>
         </response>
       </method>
     </resource>
   </resource>
   <!-- ... -->
  </resources>
```

2.5.4.1.6. Web Service SOAP Extension Example

Web services can be imported into OTRS by a YAML with a predefined structure in this case we will extend `development/webservices/GenericTickeConnectorSOAP.yml` for a SOAP web service.

```
Provider:
  Operation:
```

```
    # ...
    Test:
      Description: This is only a test
      MappingInbound: {}
      MappingOutbound: {}
      Type: Test::Test
```

2.5.4.1.7. Web Service REST Extension Example

Web services can be imported into OTRS by a YAML with a predefined structure in this case we will extend `development/webservices/GenericTickeConnectorREST.yml` for a REST web service.

```
Provider:
  Operation:
    # ...
    Test:
      Description: This is only a test
      MappingInbound: {}
      MappingOutbound: {}
      Type: Test::Test
  # ...
  Transport:
    Config:
      # ...
      RouteOperationMapping:
        # ..
        Test:
          RequestMethod:
          - GET
          Route: /Test
```

2.6. Daemon And Scheduler

2.6.1. OTRS Daemon

The OTRS Daemon is a separated process that helps OTRS to execute certain actions asynchronously and detached of the web server process, but sharing the same database.

2.6.1.1. OTRS Daemon Modules

The OTRS Daemon `bin/otrs.Daemon.pl` main purpose is to call (Daemonize) all the registered daemon modules in the System Configuration.

Each daemon module must implement a common API in order to be correctly called by the OTRS Daemon and be a semi persistent process in the system. Persistent process could grow in size and memory usage over the time and normally they do not respond to changes in the configuration. That is why the daemon modules should implement a discard mechanism to be stopped and re-spawned again from time to time, freeing system resources and re-reading the configuration.

A daemon module could be an all-in-one solution to perform a certain job, but there could be the case that a solution requires different daemon modules due to its complexity. That is exactly the case of the OTRS Scheduler Daemon that is split into several daemon modules including some daemon modules for task management and task execution.

It is not always necessary to create a new daemon module to perform certain task, usually the OTRS Scheduler Daemon can deal with the majority of them, either if it is an OTRS function that needs to be executed on a regular basis (CRON like) or if it's triggered by an OTRS event, the OTRS Scheduler should be capable to deal with it out of the box or by adding a new scheduler task worker module.

2.6.1.1.1. Creating A New Daemon Module

All daemon modules requires to be registered in the System Configuration in order to be called by the main OTRS Daemon.

2.6.1.1.1.1. Daemon Module Registration Code Example

```
<Setting Name="DaemonModules###TestDaemon" Required="1" Valid="1">
    <Description Translatable="1">The daemon registration for the scheduler generic agent
 task manager.</Description>
    <Navigation>Daemon::ModuleRegistration</Navigation>
    <Value>
        <Hash>
            <Item Key="Module">Kernel::System::Daemon::DaemonModules::TestDaemon</Item>
        </Hash>
    </Value>
</Setting>
```

2.6.1.1.1.2. Daemon Module Code Example

The following code implements a daemon module that displays the system time every 2 seconds.

```
# --
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

package Kernel::System::Daemon::DaemonModules::TestDaemon;

use strict;
use warnings;
use utf8;

use Kernel::System::VariableCheck qw(:all);

use parent qw(Kernel::System::Daemon::BaseDaemon);

our @ObjectDependencies = (
    'Kernel::Config',
    'Kernel::System::Cache',
    'Kernel::System::DB',
);
```

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

In this case we are inheriting from BaseDaemon class, and the object manager dependencies are set.

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

    # Allocate new hash for object.
    my $Self = {};
    bless $Self, $Type;

    # Get objects in constructor to save performance.
    $Self->{ConfigObject} = $Kernel::OM->Get('Kernel::Config');
    $Self->{CacheObject}  = $Kernel::OM->Get('Kernel::System::Cache');
    $Self->{DBObject}     = $Kernel::OM->Get('Kernel::System::DB');

    # Disable in memory cache to be clusterable.
```

```
    $Self->{CacheObject}->Configure(
        CacheInMemory  => 0,
        CacheInBackend => 1,
    );

    $Self->{SleepPost} = 2;          # sleep 2 seconds after each loop
    $Self->{Discard}   = 60 * 60;    # discard every hour

    $Self->{DiscardCount} = $Self->{Discard} / $Self->{SleepPost};

    $Self->{Debug}      = $Param{Debug};
    $Self->{DaemonName} = 'Daemon: TestDaemon';

    return $Self;
}
```

The constructor new creates a new instance of the class. Some used objects are also created here. It is highly recommended to disable in-memory cache in daemon modules especially if OTRS runs in a cluster environment.

In order to make this daemon module to be executed every two seconds it is necessary to define a sleep time accordingly, otherwise it will be executed as soon as possible.

Refreshing the daemon module from time to time is necessary in order to define when it should be discarded.

For the following functions (`PreRun`, Run and `PostRun`) if they return false, the main OTRS Daemon will discard the object and create a new one as soon as possible.

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

    # Check if database is on-line.
    return 1 if $Self->{DBObject}->Ping();

    sleep 10;

    return;
}
```

The `PreRun` method is executed before the main daemon module method, and the its purpose is to perform some test before the real operation. In this case a check to the database is done (always recommended), otherwise it sleeps for 10 seconds. This is needed in order to wait for DB connection to be reestablished.

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

    print "Current time " . localtime . "\n";

    return 1;
}
```

The Run method is where the main daemon module code resides, in this case it only prints the current time.

```
sub PostRun {
    my ( $Self, %Param ) = @_;
    sleep $Self->{SleepPost};
    $Self->{DiscardCount}--;

    if ( $Self->{Debug} ) {
        print "  $Self->{DaemonName} Discard Count: $Self->{DiscardCount}\n";
    }
```

```
    return if $Self->{DiscardCount} <= 0;

    return 1;
}
```

The `PostRun` method is used to perform the sleeps (preventing the daemon module to be executed too often) and also to manage the safe discarding of the object. Other operations like verification or cleanup can be done here.

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

    my %Summary = (
        Header => 'Test Daemon Summary:',
        Column => [
            {
                Name        => 'SomeColumn',
                DisplayName => 'Some Column',
                Size        => 15,
            },
            {
                Name        => 'AnotherColumn',
                DisplayName => 'Another Column',
                Size        => 15,
            },
            # ...
        ],
        Data => [
            {
                SomeColumn    => 'Some Data 1',
                AnotherColumn => 'Another Data 1',
            },
            {
                SomeColumn    => 'Some Data 2',
                AnotherColumn => 'Another Data 2',
            },
            # ...
        ],
        NoDataMesssage => '',
    );

    return \%Summary;
}
```

The `Summary` method is called by the console command `Maint::Daemon::Summary` and it's required to return `Header`, `Column`, `Data` and `NoDataMessages` keys. `Column` and `Data` needs to be an array of hashes. It is used to display useful information of what the daemon module is currently doing, or what has been done so far. This method is optional.

```
1;
```

End of file.

2.6.2. OTRS Scheduler

The OTRS Scheduler is a conjunction of daemon modules and task workers that runs together in order to perform all needed OTRS tasks asynchronously from the web server process.

2.6.2.1. OTRS Scheduler Task Managers

`SchedulerCronTaskManager` reads registered cron tasks from the OTRS SysConfig and determines the correct time to create a task to be executed.

SchedulerFutureTaskManager checks the tasks that are set to be executed just one time in the future and sets this task to be executed in time. For example, when a Generic Interface Invoker can not reach the remote server, it can self schedule to be run again 5 minutes later.

SchedulerGenericAgentTaskManager continuously reads the GenericAgent tasks that are set to be run on regular time basis and sets their execution accordingly.

Whenever these tasks managers are not enough, a new daemon module can be created. At a certain point of its Run() method it needs to call TaskAdd() from the chedulerDB object to register a task, and as soon as it is registered, it will be executed in the next free slot by the SchedulerTaskWorker.

2.6.2.2. OTRS Scheduler Task Workers

SchedulerTaskWorker execute all tasks planned by the previous tasks managers plus the ones that come directly from the code by using the Asynchronous Executor.

In order to execute each task, the SchedulerTaskWorker calls a backend module (Task Worker) to perform the specific task. The worker module is determined by the task type. If a new task type is added, it will require a new task worker.

2.6.2.2.1. Creating A New Scheduler Task Worker

All files placed under Kernel/System/Daemon/DaemonModules/SchedulerTaskWorker could potentially be task workers and they do not require any registration in the system configuration.

2.6.2.2.1.1. Scheduler Task Worker Code Example

```
# --
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

package Kernel::System::Daemon::DaemonModules::SchedulerTaskWorker::TestWorker;

use strict;
use warnings;

use parent qw(Kernel::System::Daemon::DaemonModules::BaseTaskWorker);

our @ObjectDependencies = (
    'Kernel::System::Log',
);
```

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

In this case we are inheriting from BaseTaskWorker class, and the object manager dependencies are set.

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

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

```
    $Self->{Debug}      = $Param{Debug};
    $Self->{WorkerName} = 'Worker: Test';

    return $Self;
}
```

The constructor new creates a new instance of the class.

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

    # Check task params.
    my $CheckResult = $Self->_CheckTaskParams(
        %Param,
        NeededDataAttributes => [ 'NeededAtrribute1', 'NeededAtrribute2' ],
        DataParamsRef        => 'HASH', # or 'ARRAT'
    );

    # Stop execution if an error in params is detected.
    return if !$CheckResult;

    my $Success;
    my $ErrorMessage;

    if ( $Self->{Debug} ) {
        print "    $Self->{WorkerName} executes task: $Param{TaskName}\n";
    }

    do {

        # Localize the standard error.
        local *STDERR;

        # Redirect the standard error to a variable.
        open STDERR, ">>", \$ErrorMessage;

        $Success = $Kernel::OM->Get('Kernel::System::MyPackage')->Run(
            Param1 => 'someparam',
        );
    };

    if ( !$Success ) {

        $ErrorMessage ||= "$Param{TaskName} execution failed without an error message!";

        $Self->_HandleError(
            TaskName     => $Param{TaskName},
            TaskType     => 'Test',
            LogMessage   => "There was an error executing $Param{TaskName}: $ErrorMessage",
            ErrorMessage => "$ErrorMessage",
        );
    }

    return $Success;
}
```

The Run is the main method. A call to `_CheckTaskParams()` from the base class will save some lines of code. Executing the task while capturing the STDERR is a very good practice, since the OTRS Scheduler runs normally unattended, and saving all errors to a variable will make it available for further processing. `_HandleError()` provides a common interface to send the error messages as email to the recipient specified in the System Configuration.

```
1;
```

End of file.

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 go beyond that, a new integer data type 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.

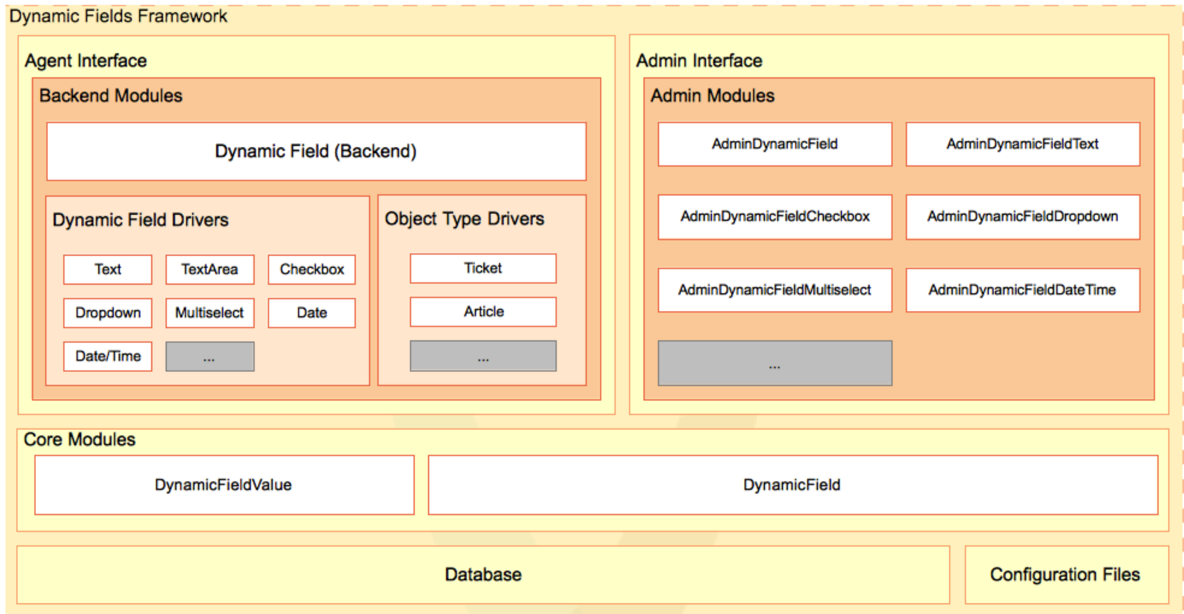
Due to its modular design each Dynamic Field type can be seen as a plug-in to a framework, and this plug-in can be an OTRS standard package to extend the available types of the Dynamic Fields or even to extend current Dynamic Field with more functions.

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.

Figure 3.2. Dynamic Fields Architecture



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 Backend 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.

These 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. These functions are triggered by the Backend object as they are needed.

These 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 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 follow 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.

These modules are located in `$OTRS_HOME/Kernel/Modules/*.pm`.

> **Note**
>
> Each Admin Dialog needs its corresponding HTML template file (`.tt`).

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. This module 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 form and into the database. This module is highly used 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 Object Type 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 need 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.

Figure 3.3. Dynamic Field Interaction

OTRS
Agent Frontend Modules
AgentTicketPhone
AgentTicketEmail
AgentTicketMove
...
AgentTicketZoom
Dynamic Fields Framework
Dynamic Field (Backend)
Dynamic Field Drivers
Text
TextArea
Checkbox
Dropdown
Multiselect
Date
Date/Time
...
Core Modules
DynamicField
DynamicFieldValue
...
Config
Database
Configuration Files

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 Backend Object function `EditFieldValueGet()` for this purpose, and can use this values to trigger ACLs. The Backend Object 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 Backend Object 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 Backend Object 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 Backend Object 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 Delegates 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 Backend object 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 it is necessary to create a new Backend extension file that defines the new functions and create also Drivers extensions that implement these new functions for each field. These new drivers will only need to implement the new functions since the original drivers takes care of the standard functions. All these new files do not need a constructor as they will be loaded as a base for the Backend object 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 and 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, 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.tt
...
|    |--/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 a 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 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 have to be created in new.

```
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, and it validates that 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 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 from 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 template to generate the Admin Dialog HTML code.

2.7.5.1.3. Dynamic Field 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 template for the password Dynamic Field is shown and explained.

```
# --
# AdminDynamicFieldPassword.tt - provides HTML form for AdminDynamicFieldPassword
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --
```

This is common header that can be found in common OTRS modules.

```
<div class="MainBox ARIARoleMain LayoutFixedSidebar SidebarFirst">
    <h1>[% Translate("Dynamic Fields") | html %] - [% Translate(Data.ObjectTypeName) |
 html %]: [% Translate(Data.Mode) | html %] [% Translate(Data.FieldTypeName) | html %] [%
 Translate("Field") | html %]</h1>
```

```
    <div class="Clear"></div>

    <div class="SidebarColumn">
        <div class="WidgetSimple">
            <div class="Header">
                <h2>[% Translate("Actions") | html %]</h2>
            </div>
            <div class="Content">
                <ul class="ActionList">
                    <li>
                        <a href="[% Env("Baselink") %]Action=AdminDynamicField"
 class="CallForAction"><span>[% Translate("Go back to overview") | html %]</span></a>
                    </li>
                </ul>
            </div>
        </div>
    </div>
```

This 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="[% Data.Mode | html %]Action" />
            <input type="hidden" name="ObjectType" value="[% Data.ObjectType | html %]" />
            <input type="hidden" name="FieldType" value="[% Data.FieldType | html %]" />
            <input type="hidden" name="ID" value="[% Data.ID | html %]" />
```

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>[% Translate("General") | html %]</h2>
                </div>
                <div class="Content">
                    <div class="LayoutGrid ColumnsWithSpacing">
                        <div class="Size1of2">
                            <fieldset class="TableLike">
                                <label class="Mandatory" for="Name"><span class="Marker">*</
span> [% Translate("Name") | html %]:</label>
                                <div class="Field">
                                    <input id="Name" class="W50pc [% Data.NameServerError |
 html %] [% Data.ShowWarning | html %]  Validate_Alphanumeric" type="text" maxlength="200"
 value="[% Data.Name | html %]" name="Name"/>
                                    <div id="NameError" class="TooltipErrorMessage"><p>[%
 Translate("This field is required, and the value should be alphabetic and numeric
 characters only.") | html %]</p></div>
                                    <div id="NameServerError"
 class="TooltipErrorMessage"><p>[% Translate(Data.NameServerErrorMessage) | html %]</p></
div>
                                    <p class="FieldExplanation">[% Translate("Must be unique
 and only accept alphabetic and numeric characters.") | html %]</p>
                                    <p class="Warning Hidden">[% Translate("Changing this
 value will require manual changes in the system.") | html %]</p>
                                </div>
                                <div class="Clear"></div>

                                <label class="Mandatory" for="Label"><span
 class="Marker">*</span> [% Translate("Label") | html %]:</label>
                                <div class="Field">
                                    <input id="Label" class="W50pc [% Data.LabelServerError
 | html %] Validate_Required" type="text" maxlength="200" value="[% Data.Label | html %]"
 name="Label"/>
```

```
                                    <div id="LabelError" class="TooltipErrorMessage"><p>[%
 Translate("This field is required.") | html %]</p></div>
                                    <div id="LabelServerError"
 class="TooltipErrorMessage"><p>[% Translate(Data.LabelServerErrorMessage) | html %]</p></
div>
                                    <p class="FieldExplanation">[% Translate("This is the
 name to be shown on the screens where the field is active.") | html %]</p>
                                </div>
                                <div class="Clear"></div>

                                <label class="Mandatory" for="FieldOrder"><span
 class="Marker">*</span> [% Translate("Field order") | html %]:</label>
                                <div class="Field">
                                    [% Data.DynamicFieldOrderSrtg %]
                                    <div id="FieldOrderError"
 class="TooltipErrorMessage"><p>[% Translate("This field is required and must be numeric.")
 | html %]</p></div>
                                    <div id="FieldOrderServerError"
 class="TooltipErrorMessage"><p>[% Translate(Data.FieldOrderServerErrorMessage) | html %]</
p></div>
                                    <p class="FieldExplanation">[% Translate("This is the
 order in which this field will be shown on the screens where is active.") | html %]</p>
                                </div>
                                <div class="Clear"></div>
                            </fieldset>
                        </div>
                        <div class="Size1of2">
                            <fieldset class="TableLike">
                                <label for="ValidID">[% Translate("Validity") | html %]:</
label>
                                <div class="Field">
                                    [% Data.ValidityStrg %]
                                </div>
                                <div class="Clear"></div>

                                <div class="SpacingTop"></div>
                                <label for="FieldTypeName">[% Translate("Field type") | html
 %]:</label>
                                <div class="Field">
                                    <input id="FieldTypeName" readonly="readonly"
 class="W50pc" type="text" maxlength="200" value="[% Data.FieldTypeName | html %]"
 name="FieldTypeName"/>
                                    <div class="Clear"></div>
                                </div>

                                <div class="SpacingTop"></div>
                                <label for="ObjectTypeName">[% Translate("Object type") |
 html %]:</label>
                                <div class="Field">
                                    <input id="ObjectTypeName" readonly="readonly"
 class="W50pc" type="text" maxlength="200" value="[% Data.ObjectTypeName | html %]"
 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>[% Translate(Data.FieldTypeName) | html %] [% Translate("Field
 Settings") | html %]</h2>
                </div>
                <div class="Content">
                    <fieldset class="TableLike">
```

```
                        <label for="DefaultValue">[% Translate("Default value") | html %]:</
label>
                        <div class="Field">
                            <input id="DefaultValue" class="W50pc" type="text"
 maxlength="200" value="[% Data.DefaultValue | html %]" name="DefaultValue"/>
                            <p class="FieldExplanation">[% Translate("This is the default
 value for this field.") | html %]</p>
                        </div>
                        <div class="Clear"></div>

                        <label for="ShowValue">[% Translate("Show value") | html %]:</label>
                        <div class="Field">
                            [% Data.ShowValueStrg %]
                            <p class="FieldExplanation">
                                [% Translate("To reveal the field value in non edit screens
 ( e.g. Ticket Zoom Screen )") | html %]
                            </p>
                        </div>
                        <div class="Clear"></div>

                        <label for="ValueMask">[% Translate("Hidden value mask") | html
 %]:</label>
                        <div class="Field">
                            <input id="ValueMask" class="W50pc" type="text" maxlength="200"
 value="[% Data.ValueMask | html %]" name="ValueMask"/>
                            <p class="FieldExplanation">
                                [% Translate("This is the alternate value to show if Show
 value is set to \"No\" ( Default: **** ).") | html %]
                            </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 can be set and it could use JavaScript 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="[% Translate("Save") | html
 %]">[% Translate("Save") | html %]</button>
                    [% Translate("or") | html %]
                    <a href="[% Env("Baselink") %]Action=AdminDynamicField">[%
 Translate("Cancel") | html %]</a>
                </div>
                <div class="Clear"></div>
            </fieldset>
        </form>
    </div>
</div>
[% WRAPPER JSOnDocumentComplete %]
<script type="text/javascript">//<![CDATA[
$('.ShowWarning').bind('change keyup', function (Event) {
    $('p.Warning').removeClass('Hidden');
});

Core.Agent.Admin.DynamicField.ValidationInit();
//]]></script>
[% END %]
```

The final part of the file contains the "Submit" button and the "Cancel" link, as well as other needed JavaScript 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 inherits 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 On-line 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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);

our @ObjectDependencies = (
    'Kernel::Config',
    'Kernel::System::DynamicFieldValue',
    'Kernel::System::Main',
);
```

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 );

    # set field behaviors
    $Self->{Behaviors} = {
        'IsACLReducible'                => 0,
        'IsNotificationEventCondition' => 1,
        'IsSortable'                   => 0,
        'IsFiltrable'                  => 0,
        'IsStatsCondition'             => 1,
        'IsCustomerInterfaceCapable'   => 1,
    };
```

```
    # get the Dynamic Field Backend custom extensions
    my $DynamicFieldDriverExtensions
        = $Kernel::OM->Get('Kernel::Config')-
>Get('DynamicFields::Extension::Driver::Password');

    EXTENSION:
    for my $ExtensionKey ( sort keys %{$DynamicFieldDriverExtensions} ) {

        # skip invalid extensions
        next EXTENSION if !IsHashRefWithData( $DynamicFieldDriverExtensions-
>{$ExtensionKey} );

        # create a extension config shortcut
        my $Extension = $DynamicFieldDriverExtensions->{$ExtensionKey};

        # check if extension has a new module
        if ( $Extension->{Module} ) {

            # check if module can be loaded
            if (
                !$Kernel::OM->Get('Kernel::System::Main')->RequireBaseClass( $Extension-
>{Module} )
                )
            {
                die "Can't load dynamic fields backend module"
                    . " $Extension->{Module}! $@";
            }
        }

        # check if extension contains more behaviors
        if ( IsHashRefWithData( $Extension->{Behaviors} ) ) {

            %{ $Self->{Behaviors} } = (
                %{ $Self->{Behaviors} },
                %{ $Extension->{Behaviors} }
            );
        }
    }

    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.

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 `BackendObject` 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 from 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(
        %Param,
        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 HTMLOutput 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.'
    }

    # HTMLOutput 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 revealed 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 = $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`.

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. For example if the instance of an 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 from 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 a structure it returns a pair for each part of the date in the hash.

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 it is 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 is 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 a 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 matches 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 Backend Object as well as 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 function 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 drivers.

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 an 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. These extended behaviors will be added to the behaviors that the Driver has out of the box, therefore a call to `HasBehavior()` to check for these 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 Backend 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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 common header that can be found in common OTRS modules. The class/package name is declared via the package keyword.

```

=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} ) {
            $Kernel::OM->Get('Kernel::System::Log')->Log(
                Priority => 'error',
                Message  => "Need $Needed!",
            );

            return;
        }
    }

    # check DynamicFieldConfig (general)
    if ( !IsHashRefWithData( $Param{DynamicFieldConfig} ) ) {
        $Kernel::OM->Get('Kernel::System::Log')->Log(
            Priority => 'error',
            Message  => "The field configuration is invalid",
        );

        return;
    }

    # check DynamicFieldConfig (internally)
    for my $Needed (qw(ID FieldType ObjectType)) {
        if ( !$Param{DynamicFieldConfig}->{$Needed} ) {
            $Kernel::OM->Get('Kernel::System::Log')->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} ) {
        $Kernel::OM->Get('Kernel::System::Log')->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 tests if the Driver can execute the function. To do that it 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.

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) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.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 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.

2.8. Email Handling

2.8.1. 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 is in the database).

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 exemplary module to match emails and set X-OTRS-Headers (see `doc/X-OTRS-Headers.txt` for more info).

`Kernel/Config/Files/MyModule.xml`:

```
<ConfigItem Name="PostMaster::PreFilterModule###1-Example" Required="0" Valid="1">
    <Description Translatable="1">Example module to filter and manipulate incoming
 messages.</Description>
    <Group>Ticket</Group>
    <SubGroup>Core::PostMaster</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::System::PostMaster::Filter::Example</Item>
            <Item Key="Match">
                <Hash>
                    <Item Key="From">noreply@</Item>
                </Hash>
            </Item>
            <Item Key="Set">
                <Hash>
                    <Item Key="X-OTRS-Ignore">yes</Item>
                </Hash>
            </Item>
        </Hash>
    </Setting>
</ConfigItem>
```

And the actual filter code in Kernel/System/PostMaster/Filter/Example.pm:

```
# --
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --

package Kernel::System::PostMaster::Filter::Example;

use strict;
use warnings;

our @ObjectDependencies = (
    'Kernel::System::Log',
);

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

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

    $Self->{Debug} = $Param{Debug} || 0;

    return $Self;
}

sub Run {
    my ( $Self, %Param ) = @_;
    # get config options
    my %Config = ();
    my %Match = ();
    my %Set = ();
    if ($Param{JobConfig} && 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}->{$_} && $Param{GetParam}->{$_} =~ /$Match{$_}/i) {
            $Matched = $1 || '1';
            if ($Self->{Debug} > 1) {
                $Kernel::OM->Get('Kernel::System::Log')->Log(
                    Priority => 'debug',
                    Message => "'$Param{GetParam}->{$_}' =~ /$Match{$_}/i matched!",
                );
            }
        }
        else {
            $MatchedNot = 1;
            if ($Self->{Debug} > 1) {
                $Kernel::OM->Get('Kernel::System::Log')->Log(
                    Priority => 'debug',
                    Message => "'$Param{GetParam}->{$_}' =~ /$Match{$_}/i matched NOT!",
                );
            }
        }
        }
        # should I ignore the incoming mail?
        if ($Matched && !$MatchedNot) {
```

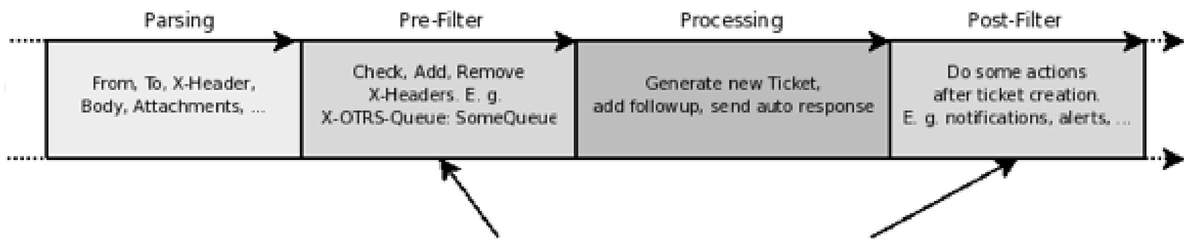
```
        for (keys %Set) {
            if ($Set{$_} =~ /\[\*\*\*\]/i) {
                $Set{$_} = $Matched;
            }
            $Param{GetParam}->{$_} = $Set{$_};
            $Kernel::OM->Get('Kernel::System::Log')->Log(
                Priority => 'notice',
                Message => "Set param '$_' to '$Set{$_}' (Message-ID: $Param{GetParam}-
>{'Message-ID'}) ",
            );
        }
    }

    return 1;
}

1;
```

The following image shows you the email processing flow.

Figure 3.4. Email Processing Flow



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 repository, just tell the OTRS framework where the location is by activating the SysConfig setting `Package::RepositoryList` and adding the new location there. Then you will have a new select option in the package manager.

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.Console.pl Dev::Package::RepositoryIndex /path/to/repository/ > /path/to/
repository/otrs.xml
```

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/otrsConsole.pl Admin::Package::Install /path/to/package.opm
```

1.2.2. Uninstall

Uninstall OPM packages.

- Web: http://localhost/otrs/index.pl?Action=AdminPackageManager
- CMD:

```
shell> bin/otrsConsole.pl Admin::Package::Uninstall /path/to/package.opm
```

1.2.3. Upgrade

Upgrade OPM packages.

- Web: http://localhost/otrs/index.pl?Action=AdminPackageManager
- CMD:

```
shell> bin/otrsConsole.pl Admin::Package::Upgrade /path/to/package.opm
```

1.2.4. List

List all OPM packages.

- Web: http://localhost/otrs/index.pl?Action=AdminPackageManager
- CMD:

```
shell> bin/otrsConsole.pl Admin::Package::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>https://otrs.com/</URL>
```

2.1.6. <License>

The license of the package (required).

```
<License>GNU GENERAL PUBLIC LICENSE Version 3, 29 June 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. Package Actions

The possible actions for the package after installation. If one of these actions is not defined on the package, it will be considered as possible.

```
<PackageIsVisible>1</PackageIsVisible>
<PackageIsDownloadable>0</PackageIsDownloadable>
<PackageIsRemovable>1</PackageIsRemovable>
```

2.1.10. <BuildHost>

This will be filled in automatically by OPM.

```
<BuildHost>?</BuildHost>
```

2.1.11. <BuildDate>

This will be filled in automatically by OPM.

```
<BuildDate>?</BuildDate>
```

2.1.12. <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>
```

2.1.13. <ModuleRequired>

Perl modules that must be installed beforehand (optional).

```
<ModuleRequired Version="1.03">Encode</ModuleRequired>
<ModuleRequired Version="5.32">MIME::Tools</ModuleRequired>
```

2.1.14. <OS>

Required OS (optional).

```
<OS>linux</OS>
<OS>darwin</OS>
<OS>mswin32</OS>
```

2.1.15. <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>
```

2.1.16. <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 Package Life Cycle.

2.1.17. <DatabaseUpgrade>

Information on which actions have to be performed in case of an upgrade (optional). Example if already installed package version is below 1.3.4 (e. g. 1.2.6), the 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 Package Life Cycle.

2.1.18. <DatabaseReinstall>

Information on which 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 Package Life Cycle.

2.1.19. <DatabaseUninstall>

Actions to be performed on package uninstall (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 Package Life Cycle.

2.1.20. <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 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 keep the new-lines and whitespace of the content).

2.1.21. <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 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 keep the newlines and whitespace of the content).

2.1.22. `<IntroReinstall>`

To show a "pre" or "post" reinstall introduction in re-installation dialog.

```
<IntroReinstall Type="post" Lang="en" Title="Some Title"><![CDATA[
Some Info formatted in 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 keep the newlines and whitespace of the content).

2.1.23. `<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 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 keep the newlines and whitespace of the content).

2.1.24. `<CodeInstall>`

Perl code to be executed when the package is installed (optional).

```
<CodeInstall><![CDATA[
# log example
$Kernel::OM->Get('Kernel::System::Log')->Log(
    Priority => 'notice',
    Message => "Some Message!",
);
# database example
$Kernel::OM->Get('Kernel::System::DB')->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 Package Life Cycle.

2.1.25. `<CodeUninstall>`

Perl code to be executed when the package is uninstalled (optional). On "pre" or "post" time of package uninstallation.

```
<CodeUninstall><![CDATA[
...
]]></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 Package Life Cycle.

2.1.26. <CodeReinstall>

Perl code to be executed when the package is reinstalled (optional).

```
<CodeReinstall><![CDATA[
...
]]></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 Package Life Cycle.

2.1.27. <CodeUpgrade>

Perl code to be executed when 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[
...
]]></CodeUpgrade>
```

You also can choose <CodeUpgrade Type="post"> or <CodeUpgrade Type="pre"> to define the time of execution separately (post is default). For more info see Package Life Cycle.

2.1.28. <PackageMerge>

This tag singals that a package has been merged into another package. In this case the original package needs to be removed from the file system and the packages database, but all data must be kept. Let's assume that PackageOne was merged into PackageTwo. Then PackageTwo.sopm should contain this:

```
<PackageMerge Name="MergeOne" TargetVersion="2.0.0"></PackageMerge>
```

If PackageOne also contained database structures, we need to be sure that it was at the latest available version of the package to have a consistent state in the database after merging the package. The attribute TargetVersion does just this: it signifies the last known version of PackageOne at the time PackageTwo was created. This is mainly to stop the upgrade process if in the user's system a version of PackageOne was found that is *newer* than the one specified in TargetVersion as this could lead to problems.

Additionally it is possible to add required database and code upgrade tags for PackageOne to make sure that it gets properly upgraded to the TargetVersion*before* merging it - to avoid inconsistency problems. Here's how this could look like:

```
<PackageMerge Name="MergeOne" TargetVersion="2.0.0">
  <DatabaseUpgrade Type="merge">
      <TableCreate Name="merge_package">
          <Column Name="id" Required="true" PrimaryKey="true" AutoIncrement="true"
 Type="INTEGER"/>
          <Column Name="description" Required="true" Size="200" Type="VARCHAR"/>
      </TableCreate>
```

```
  </DatabaseUpgrade>
</PackageMerge>
```

As you can see the attribute Type="merge" needs to be set in this case. These sections will only be executed if a package merge is possible.

2.1.29. Conditions: IfPackage and IfNotPackage

These attributes can be added to the regular Database* and Code* sections. If they are present, the section will only be executed if another package is or is not in the local package repository.

```
<DatabaseInstall IfPackage="AnyPackage">
    ...
</DatabaseInstall>
```

or

```
<CodeUpgrade IfNotPackage="OtherPackage">
    ...
</CodeUpgrade>
```

These attributes can be also set in the Database* and Code* sections inside the PackageMerge tags.

2.2. Example .sopm

This is an example spec file looks with some of the above tags.

```
<?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>https://otrs.com/</URL>
    <License>GNU GENERAL PUBLIC LICENSE Version 3, 29 June 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.tt"></
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

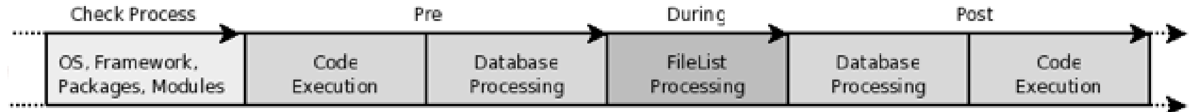
To build an .opm package from the spec opm.

```
shell> bin/otrs.Console.pl Dev::Package::Build /path/to/example.sopm /tmp
Building package...
Done.
shell>
```

2.4. Package Life Cycle - Install/Upgrade/Uninstall

The following image shows you how the life cycle of a package installation/upgrade/uninstallation works in the backend step by step.

Figure 4.1. Package Life Cycle



3. Package Porting

With every new minor or major version of OTRS, you need to port your package(s) and make sure they still work with the OTRS API.

3.1. From OTRS 5 to 6

This section lists changes that you need to examine when porting your package from OTRS 5 to 6.

3.1.1. Date and time calculation

In OTRS 6, a new module for date and time calculation was added: `Kernel::System::DateTime`. The module `Kernel::System::Time` is now deprecated and should not be used for new code anymore.

The main advantage of the new `Kernel::System::DateTime` module is the support for real time zones like `Europe/Berlin` instead of time offsets in hours like +2. Note that also the old `Kernel::System::Time` module has been improved to support time zones. Time offsets have been completely dropped. This means that any code that uses time

offsets for calculations has to be ported to use the new `DateTime` module instead. Code that doesn't fiddle around with time offsets itself can be left untouched in most cases. You just have to make sure that upon creation of a `Kernel::System::Time` object a valid time zone will be given.

Here's an example for porting time offset code to time zones:

```
my $TimeObject     = $Kernel::OM->Get('Kernel::System::Time'); # Assume a time offset of 0
 for this time object
my $SystemTime     = $TimeObject->TimeStamp2SystemTime( String => '2004-08-14 22:45:00' );
my $UserTimeZone   = '+2'; # normally retrieved via config or param
my $UserSystemTime = $SystemTime + $UserTimeZone * 3600;
my $UserTimeStamp  = $TimeObject->SystemTime2TimeStamp( SystemTime => $UserSystemTime );
```

Code using the new `Kernel::System::DateTime` module:

```
my $DateTimeObject = $Kernel::OM->Create('Kernel::System::DateTime'); # This implicitly sets
 the configured OTRS time zone
my $UserTimeZone   = 'Europe/Berlin'; # normally retrieved via config or param
$DateTimeObject->ToTimeZone( TimeZone => $UserTimeZone );
my $SystemTime    = $DateTimeObject->ToEpoch(); # note that the epoch is independent from
 the time zone, it's always calculated for UTC
my $UserTimeStamp = $DateTimeObject->ToString();
```

Please note that the returned time values with the new `Get()` function in the `Kernel::System::DateTime` module are without leading zero instead of the old `SystemTime2Date()` function in the `Kernel::System::Time` module. In the new `Kernel::System::DateTime` module the function `Format()` returns the date/time as string formatted according to the given format.

3.1.2. Adding the drag & drop multiupload

For OTRS 6, a multi attachment upload functionality was added. To implement the multi attachment upload in other extensions it is necessary to remove the attachment part from the template file, also the `JSOnDocumentComplete` parts (`AttachmentDelete` and `AttachmentUpload`). Please keep in mind, in some cases the JavaScript parts are already outsourced in `Core.Agent.XXX` files.

Note

Please note that this is currently only applicable for places where it actually makes sense to have the possibility to upload multiple files (like `AgentTicketPhone`, `AgentTicketCompose`, etc.). This is not usable out of the box for admin screens.

To include the new multi attachment upload in the template, replace the existing `input type="file"` with the following code in your `.tt` template file:

```
<label>[% Translate("Attachments") | html %]:</label>
<div class="Field">
[% INCLUDE "FormElements/AttachmentList.tt" %]
</div>
<div class="Clear"></div>
```

It is also necessary to remove the `IsUpload` variable and all other `IsUpload` parts from the Perl module. Code parts like following are not needed anymore:

```
my $IsUpload = ( $ParamObject->GetParam( Param => 'AttachmentUpload' ) ? 1 : 0 );
```

Additional to that, the Attachment Layout Block needs to be replaced:

```
$LayoutObject->Block(
    Name => 'Attachment',
    Data => $Attachment,
);
```

Replace it with this code:

```
push @{ $Param{AttachmentList} }, $Attachment;
```

If the module where you want to integrate multi upload supports standard templates, make sure to add a section to have a human readable file size format right after the attachments of the selected template have been loaded (see e.g. `AgentTicketPhone` for reference):

```
for my $Attachment (@TicketAttachments) {
    $Attachment->{Filesize} = $LayoutObject->HumanReadableDataSize(
        Size => $Attachment->{Filesize},
    );
}
```

When adding selenium unit tests for the modules you ported, please take a look at `Selenium/Agent/MultiAttachmentUpload.t` for reference.

3.1.3. Improvements to administration screens

3.1.3.1. Add breadcrumbs to administration screens

In OTRS 6, all admin modules should have a breadcrumb. The breadcrumb only needs to be added on the `.tt` template file and should be placed right after the h1 headline on top of the file. Additionally, the headline should receive the class `InvisibleText` to make it only *visible* for screen readers.

```
<div class="MainBox ARIARoleMain LayoutFixedSidebar SidebarFirst">
    <h1 class="InvisibleText">[% Translate("Name of your module") | html %]</h1>
[% BreadcrumbPath = [
        {
            Name => Translate('Name of your module'),
        },
    ]
%]
[% INCLUDE "Breadcrumb.tt" Path = BreadcrumbPath %]
...
```

Please make sure to add the correct breadcrumb for all levels of your admin module (e.g. `Subactions`):

```
[% BreadcrumbPath = [
        {
            Name => Translate('Module Home Screen'),
            Link => Env("Action"),
        },
        {
            Name => Translate("Some Subaction"),
        },
    ]
%]
```

```
[% INCLUDE "Breadcrumb.tt" Path = BreadcrumbPath %]
```

3.1.3.2. Add *Save* and *Save and finish* buttons to administration screens

Admin modules in OTRS 6 should not only have a *Save* button, but also a *Save and finish* button. *Save* should leave the user on the same edit page after saving, *Save and finish* should lead back to the overview of the entity the user is currently working on. Please see existing OTRS admin screens for reference.

```
<div class="Field SpacingTop SaveButtons">
    <button class="Primary CallForAction" id="SubmitAndContinue" type="submit" value="[%
 Translate("Save") | html %]"><span>[% Translate("Save") | html %]</span></button>
    [% Translate("or") | html %]
    <button class="Primary CallForAction" id="Submit" type="submit" value="[%
 Translate("Save") | html %]"><span>[% Translate("Save and finish") | html %]</span></
button>
    [% Translate("or") | html %]
    <a href="[% Env("Baselink") %]Action=[% Env("Action") %]"><span>[% Translate("Cancel") |
 html %]</span></a>
</div>
```

3.1.4. Migrate configuration files

3.1.4.1. XML configuration file format

OTRS 6 uses a new XML configuration file format and the location of configuration files moved from `Kernel/Config/Files` to `Kernel/Config/Files/XML`. To convert existing XML configuration files to the new format and location, you can use the following tool that is part of the OTRS framework:

```
bin/otrs.Console.pl Dev::Tools::Migrate::ConfigXMLStructure --source-directory Kernel/
Config/Files
Migrating configuration XML files...
Kernel/Config/Files/Calendar.xml -> Kernel/Config/Files/XML/Calendar.xml... Done.
Kernel/Config/Files/CloudServices.xml -> Kernel/Config/Files/XML/CloudServices.xml... Done.
Kernel/Config/Files/Daemon.xml -> Kernel/Config/Files/XML/Daemon.xml... Done.
Kernel/Config/Files/Framework.xml -> Kernel/Config/Files/XML/Framework.xml... Done.
Kernel/Config/Files/GenericInterface.xml -> Kernel/Config/Files/XML/GenericInterface.xml...
 Done.
Kernel/Config/Files/ProcessManagement.xml -> Kernel/Config/Files/XML/
ProcessManagement.xml... Done.
Kernel/Config/Files/Ticket.xml -> Kernel/Config/Files/XML/Ticket.xml... Done.

Done.
```

3.1.4.2. Perl configuration file format

OTRS 6 speeds up configuration file loading by dropping support for the old configuration format (1) that just used sequential Perl code and had to be run by `eval` and instead enforcing the new package-based format (1.1) for Perl configuration files. OTRS 6+ can only load files with this format, please make sure to convert any custom developments to it (see `Kernel/Config/Files/ZZZ*.pm` for examples). Every Perl configuration file needs to contain a package with a `Load()` method.

In the past, Perl configuration files were sometimes misused as an autoload mechanism to override code in existing packages. This is not necessary any more as OTRS 6 features a dedicated `Autoload` mechanism. Please see `Kernel/Autoload/Test.pm` for a demonstration on how to use this mechanism to add a method in an existing file.

3.1.5. Perldoc structure changed

The structure of POD in Perl files was slightly improved and should be adapted in all files. POD is now also enforced to be syntactically correct.

What was previously called SYNOPSIS is now changed to DESCRIPTION, as a synopsis typically provides a few popular code usage examples and not a description of the module itself. An additional synopsis can be provided, of course. Here's how an example:

```
=head1 NAME

Kernel::System::ObjectManager - Central singleton manager and object instance generator

=head1 SYNOPSIS

# In toplevel scripts only!
local $Kernel::OM = Kernel::System::ObjectManager->new();

# Everywhere: get a singleton instance (and create it, if needed).
my $ConfigObject = $Kernel::OM->Get('Kernel::Config');

# Remove singleton objects and all their dependencies.
$Kernel::OM->ObjectsDiscard(
    Objects            => ['Kernel::System::Ticket', 'Kernel::System::Queue'],
);

=head1 DESCRIPTION

The ObjectManager is the central place to create and access singleton OTRS objects (via
 C<L</Get()>>)
as well as create regular (unmanaged) object instances (via C<L</Create()>>).
```

In case the DESCRIPTION does not add any value to the line in the NAME section, it should be rewritten or removed altogether.

The second important change is that functions are now documented as =head2 instead of the previously used =item.

```
=head2 Get()

Retrieves a singleton object, and if it not yet exists, implicitly creates one for you.

my $ConfigObject = $Kernel::OM->Get('Kernel::Config');

# On the second call, this returns the same ConfigObject as above.
my $ConfigObject2 = $Kernel::OM->Get('Kernel::Config');

=cut

sub Get { ... }
```

These changes lead to an improved online API documentation as can be seen in the ObjectManager documentation for OTRS 5 and OTRS 6.

3.1.6. Improvements to templating and working with JavaScript

3.1.6.1. JavaScript removed from templates

With OTRS 6, all JavaScript - especially located in JSOnDocumentComplete blocks - is removed from template files and moved to JavaScript files instead. Only in very rare con-

ditions JavaScript needs to be placed within template files. For all other occurrences, place the JS code in module-specific JavaScript files. An `Init()` method within such a JavaScript file is executed automatically on file load (for the initialization of event bindings etc.) if you register the JavaScript file at the OTRS application. This is done by executing `Core.Init.RegisterNamespace(TargetNS, 'APP_MODULE');` at the end of the namespace declaration within the JavaScript file.

3.1.6.2. Template files for rich text editor removed

Along with the refactoring of the JavaScript within template files (see above), the template files for the rich text editor (`RichTextEditor.tt` and `CustomerRichTextEditor.tt`) were removed as they are no longer necessary.

Typically, these template files were included in the module-specific template files within a block:

```
[% RenderBlockStart("RichText") %]
[% InsertTemplate("RichTextEditor.tt") %]
[% RenderBlockEnd("RichText") %]
```

This is no longer needed and can be removed. Instead of calling this block from the Perl module, it is now necessary to set the needed rich text parameters there. Instead of:

```
$LayoutObject->Block(
    Name => 'RichText',
    Data => \%Param,
);
```

you now have to call:

```
$LayoutObject->SetRichTextParameters(
    Data => \%Param,
);
```

Same rule applies for customer interface. Remove RichText blocks from `CustomerRichTextEditor.tt` and apply following code instead:

```
$LayoutObject->CustomerSetRichTextParameters(
    Data => \%Param,
);
```

3.1.6.3. Translations in JavaScript files

Adding translatable strings in JavaScript was quite difficult in OTRS. The string had to be translated in Perl or in the template and then sent to the JavaScript function. With OTRS 6, translation of strings is possible directly in the JavaScript file. All other workarounds, especially blocks in the templates only for translating strings, should be removed.

Instead, the new JavaScript translation namespace `Core.Language` should be used to translate strings directly in the JS file:

```
Core.Language.Translate('The string to translate');
```

It is also possible to handover JS variables to be replaced in the string directly:

```
Core.Language.Translate('The %s to %s', 'string', 'translate');
```

Every %s is replaced by the variable given as extra parameter. The number of parameters is not limited.

3.1.6.4. Handover data from Perl to JavaScript

To achieve template files without JavaScript code, some other workarounds had to be replaced with an appropriate solution. Besides translations, also the handover of data from Perl to JavaScript has been a problem in OTRS. The workaround was to add a JavaScript block in the template in which JavaScript variables were declared and filled with template tags based on data handed over from Perl to the template.

The handover process of data from Perl to JavaScript is now much easier in OTRS 6. To send specific data as variable from Perl to JavaScript, one only has to call a function on Perl-side. The data is than automatically available in JavaScript.

In Perl you only have to call:

```
$Self->{LayoutObject}->AddJSData(
    Key   => 'KeyToBeAvailableInJS',
    Value => $YourData,
);
```

The `Value` parameter is automatically converted to a JSON object and can also contain complex data.

In JavaScript you can get the data with:

```
Core.Config.Get('KeyToBeAvailableInJS');
```

This replaces all workarounds which need to be removed when porting a module to OTRS 6, because JavaScript in template files is now only allowed in very rare conditions (see above).

3.1.6.5. HTML templates for JavaScript

OTRS 6 exposes new JavaScript template API via `Core.Template` class. You can use it in your JavaScript code in a similar way as you use `TemplateToolkit` from Perl code.

Here's an example for porting existing `jQuery` based code to new template API:

```
var DivID = 'MyDiv',
    DivText = 'Hello, world!';

$('<div />').addClass('CSSClass')
    .attr('id', DivID)
    .text(DivText)
    .appendTo('body');
```

First, make sure to create a new template file under `Kernel/Output/JavaScript/Templates/Standard` folder. In doing this, you should keep following in mind:

- Create a subfolder with name of your `Module`.

- You may reuse any existing subfolder structure but only if it makes sense for your component (e.g. `Agent/MyModule/` or `Agent/Admin/MyModule/`).
- Use `.html.tmpl` as extension for template file.
- Name templates succinctly and clearly in order to avoid confusion (i.e. good: `Agent/MyModule/SettingsDialog.html.tmpl`, bad: `Agent/SettingsDialogTemplate.html.tmpl`).

Then, add your HTML to the template file, making sure to use placeholders for any variables you might need:

```
<div id="{{ DivID }}" class="CSSClass">
    {{ DivText | Translate }}
</div>
```

Then, just get rendered HTML by calling `Core.Template.Render` method with template path (without extension) and object containing variables for replacement:

```
var DivHTML = Core.Template.Render('Agent/MyModule/SettingsDialog', {
    DivID: 'MyDiv',
    DivText: 'Hello, world!'
});

$(DivHTML).appendTo('body');
```

Internally, `Core.Template` uses Nunjucks engine for parsing templates. Essentially, any valid Nunjucks syntax is supported, please see their documentation for more information.

Here are some tips:

- You can use `| Translate` filter for string translation to current language.
- All `{{ VarName }}` variable outputs are HTML escaped by default. If you need to output some existing HTML, please use `| safe` filter to bypass escaping.
- Use `| urlencode` for encoding URL parameters.
- Complex structures in replacement object are supported, so feel free to pass arrays or hashes and iterate over them right from template. For example, look at `{% for %}` syntax in Nunjucks documentation.

3.1.7. Checking user permissions

Before OTRS 6, user permissions were stored in the session and passed to the `LayoutObject` as attributes, which were then in turn accessed to determine user permissions like `if ($LayoutObject->{'UserIsGroup[admin]'}) { ... }`.

With OTRS 6, permissions are no longer stored in the session and also not passed to the `LayoutObject`. Please switch your code to calling `PermissionCheck()` on `Kernel::System::Group` (for agents) or `Kernel::System::CustomerGroup` (for customers). Here's an example:

```
my $HasPermission = $Kernel::OM->Get('Kernel::System::Group')->PermissionCheck(
UserID    => $UserID,
GroupName => $GroupName,
Type      => 'move_into',
);
```

3.1.8. Ticket API changes

3.1.8.1. `TicketGet()`

For OTRS 6, all extensions need to be checked and ported from `$Ticket{SolutionTime}` to `$Ticket{Closed}` if `TicketGet()` is called with the `Extended` parameter (see bug#11872).

Additionally, the database column `ticket.create_time_unix` was removed, and likewise the value `CreateTimeUnix` from the `TicketGet()` result data. Please use the value `Created` (database column `ticket.create_time`) instead.

3.1.8.2. `LinkObject` Events

In OTRS 6, old ticket-specific `LinkObject` events have been dropped:

- `TicketSlaveLinkAdd`
- `TicketSlaveLinkDelete`
- `TicketMasterLinkDelete`

Any event handlers listening on these events should be ported to two new events instead:

- `LinkObjectLinkAdd`
- `LinkObjectLinkDelete`

These new events will be triggered any time a link is added or deleted by `LinkObject`, regardless of the object type. `Data` parameter will contain all information your event handlers might need for further processing, e.g.:

`SourceObject`

Name of the link source object (e.g. `Ticket`).

`SourceKey`

Key of the link source object (e.g. `TicketID`).

`TargetObject`

Name of the link target object (e.g. `FAQItem`).

`TargetKey`

Key of the link target object (e.g. `FAQItemID`).

`Type`

Type of the link (e.g. `ParentChild`).

`State`

State of the link (`Valid` or `Temporary`).

With these new events in place, any events specific for custom `LinkObject` module implementations can be dropped, and all event handlers ported to use them instead. Since source and target object names are provided in the event itself, it would be trivial to make them run only in specific situations.

To register your event handler for these new events, make sure to add a registration in the configuration, for example:

```
<!-- OLD STYLE -->
<ConfigItem Name="LinkObject::EventModulePost###1000-SampleModule" Required="0" Valid="1">
    <Description Translatable="1">Event handler for sample link object module.</Description>
    <Group>Framework</Group>
    <SubGroup>Core::Event::Package</SubGroup>
    <Setting>
        <Hash>
            <Item Key="Module">Kernel::System::LinkObject::Event::SampleModule</Item>
            <Item Key="Event">(LinkObjectLinkAdd|LinkObjectLinkDelete)</Item>
            <Item Key="Transaction">1</Item>
        </Hash>
    </Setting>
</ConfigItem>

<!-- NEW STYLE -->
<Setting Name="LinkObject::EventModulePost###1000-SampleModule" Required="0" Valid="1">
    <Description Translatable="1">Event handler for sample link object module.</Description>
    <Navigation>Core::Event::Package</Navigation>
    <Value>
        <Hash>
            <Item Key="Module">Kernel::System::LinkObject::Event::SampleModule</Item>
            <Item Key="Event">(LinkObjectLinkAdd|LinkObjectLinkDelete)</Item>
            <Item Key="Transaction">1</Item>
        </Hash>
    </Value>
</Setting>
```

3.1.9. Article API changes

In OTRS 6, changes to Article API have been made, in preparations for new *Omni Channel* infrastructure.

3.1.9.1. Meta Article API

Article object now provides top-level article functions that do not involve back-end related data.

Following methods related to articles have been moved to `Kernel::System::Ticket::Article` object:

- `ArticleFlagSet()`
- `ArticleFlagDelete()`
- `ArticleFlagGet()`
- `ArticleFlagsOfTicketGet()`
- `ArticleAccountedTimeGet()`
- `ArticleAccountedTimeDelete()`
- `ArticleSenderTypeList()`
- `ArticleSenderTypeLookup()`
- `SearchStringStopWordsFind()`
- `SearchStringStopWordsUsageWarningActive()`

If you are referencing any of these methods via `Kernel::System::Ticket` object in your code, please switch to Article object and use it instead. For example:

```
my $ArticleObject = $Kernel::OM->Get('Kernel::System::Ticket::Article');
```

```
    my %ArticleSenderTypeList = $ArticleObject->ArticleSenderTypeList();
```

New `ArticleList()` method is now provided by the article object, and can be used for article listing and locating. This method implements filters and article numbering and returns article meta data only as an ordered list. For example:

```
my @Articles = $ArticleObject->ArticleList(
    TicketID             => 123,
    CommunicationChannel => 'Email',            # optional, to limit to a certain
 CommunicationChannel
    SenderType           => 'customer',         # optional, to limit to a certain article
 SenderType
    IsVisibleForCustomer => 1,                  # optional, to limit to a certain visibility
    OnlyFirst            => 1,                  # optional, only return first match, or
    OnlyLast             => 1,                  # optional, only return last match
);
```

Following methods related to articles have been dropped all-together. If you are using any of them in your code, please evaluate possibility of alternatives.

- `ArticleFirstArticle()` (use `ArticleList( OnlyFirst => 1)` instead)
- `ArticleLastCustomerArticle()` (use `ArticleList( SenderType => 'customer', OnlyLast => 1)` or similar)
- `ArticleCount()` (use `ArticleList()` instead)
- `ArticlePage()` (reimplemented in `AgentTicketZoom`)
- `ArticleTypeList()`
- `ArticleTypeLookup()`
- `ArticleIndex()` (use `ArticleList()` instead)
- `ArticleContentIndex()`

To work with article data please use new article backend API. To get correct backend object for an article, please use:

- `BackendForArticle(%Article)`
- `BackendForChannel( ChannelName => $ChannelName )`

`BackendForArticle()` returns the correct back end for a given article, or the invalid back end, so that you can always expect a back end object instance that can be used for chain-calling.

```
my $ArticleBackendObject = $ArticleObject->BackendForArticle( TicketID => 42, ArticleID =>
 123 );
```

`BackendForChannel()` returns the correct back end for a given communication channel.

```
my $ArticleBackendObject = $ArticleObject->BackendForChannel( ChannelName => 'Email' );
```

3.1.9.2. Article Backend API

All other article data and related methods have been moved to separate backends. Every communication channel now has a dedicated backend API that handles article data and can be used to manipulate it.

OTRS 6 Free ships with some default channels and corresponding backends:

- Email (equivalent to old `email` article types)
- Phone (equivalent to old `phone` article types)
- Internal (equivalent to old `note` article types)
- Chat (equivalent to old `chat` article types)

Note

While chat article backend is available in OTRS 6 Free, it is only utilized when system has a valid **OTRS Business Solution™** installed.

Article data manipulation can be managed via following backend methods:

- `ArticleCreate()`
- `ArticleUpdate()`
- `ArticleGet()`
- `ArticleDelete()`

All of these methods have dropped article type parameter, which must be substituted for `SenderType` and `IsVisibleForCustomer` parameter combination. In addition, all these methods now also require `TicketID` and `UserID` parameters.

Note

Since changes in article API are system-wide, any code using the old API must be ported for OTRS 6. This includes any web service definitions which leverage these methods directly via GenericInterface for example. They will need to be re-assessed and adapted to provide all required parameters to the new API during requests and manage subsequent responses in new format.

Please note that returning hash of `ArticleGet()` has changed, and some things (like ticket data) might be missing. Utilize parameters like `DynamicFields => 1` and `RealNames => 1` to get more information.

In addition, attachment data is not returned any more, please use combination of following methods from the article backends:

- `ArticleAttachmentIndex()`
- `ArticleAttachment()`

Note that `ArticleAttachmentIndex()` parameters and behavior has changed. Instead of old strip parameter use combination of new `ExcludePlainText`, `ExcludeHTMLBody` and `ExcludeInline`.

As an example, here is how to get all article and attachment data in the same hash:

```
my @Articles = $ArticleObject->ArticleList(
    TicketID => $TicketID,
);

ARTICLE:
for my $Article (@Articles) {
```

```
    # Make sure to retrieve backend object for this specific article.
    my $ArticleBackendObject = $ArticleObject->BackendForArticle( %{$Article} );

    my %ArticleData = $ArticleBackendObject->ArticleGet(
        %{$Article},
        DynamicFields => 1,
        UserID        => $UserID,
    );
    $Article = \%ArticleData;

    # Get attachment index (without attachments).
    my %AtmIndex = $ArticleBackendObject->ArticleAttachmentIndex(
        ArticleID => $Article->{ArticleID},
        UserID    => $UserID,
    );
    next ARTICLE if !%AtmIndex;

    my @Attachments;
    ATTACHMENT:
    for my $FileID ( sort keys %AtmIndex ) {
        my %Attachment = $ArticleBackendObject->ArticleAttachment(
            ArticleID => $Article->{ArticleID},
            FileID    => $FileID,
            UserID    => $UserID,
        );
        next ATTACHMENT if !%Attachment;

        $Attachment{FileID} = $FileID;
        $Attachment{Content} = encode_base64( $Attachment{Content} );

        push @Attachments, \%Attachment;
    }

    # Include attachment data in article hash.
    $Article->{Atms} = \@Attachments;
}
```

3.1.9.3. Article Search Index

To make article indexing more generic, article backends now provide information necessary for properly indexing article data. Index will be created similar to old `StaticDB` mechanism and stored in a dedicated article search table.

Since now every article backend can provide search on arbitrary number of article fields, use `BackendSearchableFieldsGet()` method to get information about them. This data can also be used for forming requests to `TicketSearch()` method. Coincidentally, some `TicketSearch()` parameters have changed their name to also include article backend information, for example:

Old parameter	New parameter
From	MIMEBase_From
To	MIMEBase_To
Cc	MIMEBase_Cc
Subject	MIMEBase_Subject
Body	MIMEBase_Body
AttachmentName	MIMEBase_AttachmentName

Additionally, article search indexing will be done in an async call now, in order to off-load index calculation to a separate task. While this is fine for production systems, it might create new problems in certain situations, e.g. unit tests. If you are manually creating articles in your unit test, but expect it to be searchable immediately after created, make sure to manually call the new `ArticleSearchIndexBuild()` method on article object.

3.1.10. SysConfig API changes

Note that in OTRS 6 SysConfig API was changed, so you should check if the methods are still existing. For example, `ConfigItemUpdate()` is removed. To replace it you should use combination of the following methods:

- `SettingLock()`
- `SettingUpdate()`
- `ConfigurationDeploy()`

In case that you want to update a configuration setting during a `CodeInstall` section of a package, you could use `SettingsSet()`. It does all previously mentioned steps and it can be used for multiple settings at once.

Note

Do not use `SettingSet()` in the SysConfig GUI itself.

```
my $Success = $SysConfigObject->SettingsSet(
    UserID   => 1,                                     # (required) UserID
    Comments => 'Deployment comment',                  # (optional) Comment
    Settings => [                                      # (required) List of settings to
 update.
        {
            Name                   => 'Setting::Name',  # (required)
            EffectiveValue         => 'Value',          # (optional)
            IsValid                => 1,                # (optional)
            UserModificationActive => 1,                # (optional)
        },
        ...
    ],
);
```

3.1.11. `LinkObject` API changes

Note that `LinkObject` was slightly modified in the OTRS 6 and methods `LinkList()` and `LinkKeyList()` might return different result if `Direction` parameter is used. Consider changing `Direction`.

Old code:

```
my $LinkList = $LinkObject->LinkList(
    Object    => 'Ticket',
    Key       => '321',
    Object2   => 'FAQ',
    State     => 'Valid',
    Type      => 'ParentChild',
    Direction => 'Target',
    UserID    => 1,
);
```

New code:

```
my $LinkList = $LinkObject->LinkList(
    Object    => 'Ticket',
    Key       => '321',
    Object2   => 'FAQ',
    State     => 'Valid',
    Type      => 'ParentChild',
```

```
    Direction => 'Source',
    UserID    => 1,
);
```

3.1.12. Communication Log support for additional PostMaster Filters

As part of email handling improvements for OTRS 6, a new logging mechanism was added to OTRS 6, exclusively used for incoming and outgoing communications. All PostMaster filters were enriched with this new Communication Log API, which means any additional filters coming with packages should also leverage the new log feature.

If your package implements additional PostMaster filters, make sure to get acquainted with API usage instructions. Also, you can get an example of how to implement this logging mechanism by looking the code in the `Kernel::System::PostMaster::NewTicket`.

3.1.13. Process MailQueue for unit tests

As part of email handling improvements for OTRS 6, all emails are now sent asynchronously, that means they are saved in a queue for future processing.

To the unit tests that depend on emails continue to work properly is necessary to force the processing of the email queue.

Make sure to start with a clean queue:

```
                my $MailQueueObject = $Kernel::OM->Get('Kernel::System::MailQueue');
                $MailQueueObject->Delete();
```

If for some reason you can't clean completely the queue, e.g. selenium unit tests, just delete the items created during the tests:

```
                my $MailQueueObject = $Kernel::OM->Get('Kernel::System::MailQueue');
                my %MailQueueCurrentItems = map { $_->{ID} => $_ } @{ $MailQueueObject->List() || [] };

                my $Items = $MailQueueObject->List();
                MAIL_QUEUE_ITEM:
                for my $Item ( @{$Items} ) {
                    next MAIL_QUEUE_ITEM if $MailQueueCurrentItems{ $Item->{ID} };
                    $MailQueueObject->Delete(
                        ID => $Item->{ID},
                    );
                }
```

Process the queue after the code that you expect to send emails:

```
                my $MailQueueObject = $Kernel::OM->Get('Kernel::System::MailQueue');
                my $QueueItems      = $MailQueueObject->List();
                for my $Item ( @{$QueueItems} ) {
                    $MailQueueObject->Send( %{$Item} );
                }
```

Or process only the ones created during the tests:

```
                my $MailQueueObject = $Kernel::OM->Get('Kernel::System::MailQueue');
                my $QueueItems      = $MailQueueObject->List();
```

```
                MAIL_QUEUE_ITEM:
                for my $Item ( @{$QueueItems} ) {
                    next MAIL_QUEUE_ITEM if $MailQueueCurrentItems{ $Item->{ID} };
                    $MailQueueObject->Send( %{$Item} );
                }
```

Depending on your case, you may need to clean the queue after or before processing it.

3.1.14. Widget Handling in Ticket Zoom Screen

The widgets in the ticket zoom screen have been improved to work in a more generic way. With OTRS 6, it is now possible to add new widgets for the ticket zoom screen via the SysConfig. It is possible to configure the used module, the location of the widget (e.g. Sidebar) and if the content should be loaded synchronously (default) or via AJAX.

Here is an example configuration for the default widgets:

```
<Setting Name="Ticket::Frontend::AgentTicketZoom###Widgets###0100-TicketInformation"
 Required="0" Valid="1">
    <Description Translatable="1">AgentTicketZoom widget that displays ticket data in the
 side bar.</Description>
    <Navigation>Frontend::Agent::View::TicketZoom</Navigation>
    <Value>
        <Hash>
            <Item Key="Module">Kernel::Output::HTML::TicketZoom::TicketInformation</Item>
            <Item Key="Location">Sidebar</Item>
        </Hash>
    </Value>
</Setting>
<Setting Name="Ticket::Frontend::AgentTicketZoom###Widgets###0200-CustomerInformation"
 Required="0" Valid="1">
    <Description Translatable="1">AgentTicketZoom widget that displays customer information
 for the ticket in the side bar.</Description>
    <Navigation>Frontend::Agent::View::TicketZoom</Navigation>
    <Value>
        <Hash>
            <Item Key="Module">Kernel::Output::HTML::TicketZoom::CustomerInformation</Item>
            <Item Key="Location">Sidebar</Item>
            <Item Key="Async">1</Item>
        </Hash>
    </Value>
</Setting>
```

Note

With this change, the template blocks in the widget code have been removed, so you should check if you use the old widget blocks in some output filters via `Frontend::Template::GenerateBlockHooks` functionality, and implement it in the new fashion.

3.2. From OTRS 4 to 5

This section lists changes that you need to examine when porting your package from OTRS 4 to 5.

3.2.1. `Kernel/Output/HTML` restructured

In OTRS 5, `Kernel/Output/HTML` was restructured. All Perl modules (except `Layout.pm`) were moved to subdirectories (one for every module layer). Template (theme) files were also moved from `Kernel/Output/HTML/Standard` to `Kernel/Output/HTML/Templates/Standard`. Please perform this migration also in your code.

3.2.2. Pre-Output-Filters

With OTRS 5 there is no support for `pre` output filters any more. These filters changed the template content before it was parsed, and that could potentially lead to bad performance issues because the templates could not be cached any more and had to be parsed and compiled every time.

Just switch from `pre` to `post` output filters. To translate content, you can run `$LayoutObject->Translate()` directly. If you need other template features, just define a small template file for your output filter and use it to render your content before injecting it into the main data. It can also be helpful to use jQuery DOM operations to reorder/replace content on the screen in some cases instead of using regular expressions. In this case you would inject the new code somewhere in the page as invisible content (e. g. with the class `Hidden`), and then move it with jQuery to the correct location in the DOM and show it.

To make using post output filters easier, there is also a new mechanism to request HTML comment hooks for certain templates/blocks. You can add in your module config XML like:

```
<ConfigItem
Name="Frontend::Template::GenerateBlockHooks###100-OTRSBusiness-ContactWithData"
Required="1" Valid="1">
    <Description Translatable="1">Generate HTML comment hooks for
the specified blocks so that filters can use them.</Description>
    <Group>OTRSBusiness</Group>
    <SubGroup>Core</SubGroup>
    <Setting>
        <Hash>
            <Item Key="AgentTicketZoom">
                <Array>
                    <Item>CustomerTable</Item>
                </Array>
            </Item>
        </Hash>
    </Setting>
</ConfigItem>
```

This will cause the block `CustomerTable` in `AgentTicketZoom.tt` to be wrapped in HTML comments each time it is rendered:

```
<!--HookStartCustomerTable-->
... block output ...
<!--HookEndCustomerTable-->
```

With this mechanism every package can request just the block hooks it needs, and they are consistently rendered. These HTML comments can then be used in your output filter for easy regular expression matching.

3.2.3. IE 8 and IE 9

Support for IE 8 and 9 was dropped. You can remove any workarounds in your code for these platforms, as well as any old <CSS_IE7> or <CSS_IE8> loader tags that might still lurk in your XML config files.

3.2.4. GenericInterface API change in "Ticket" connector

The operation `TicketGet()` returns dynamic field data from ticket and articles differently than in OTRS 4. Now they are cleanly separated from the rest of the static ticket and article fields - they are now grouped in a list called `DynamicField`. Please adapt any applications using this operation accordingly.

```
# changed from:

Ticket => [
{
    TicketNumber       => '20101027000001',
    Title              => 'some title',
    ...
    DynamicField_X     => 'value_x',
},
]

# to:

Ticket => [
{
    TicketNumber       => '20101027000001',
    Title              => 'some title',
    ...
    DynamicField => [
        {
            Name  => 'some name',
            Value => 'some value',
        },
    ],
},
]
```

3.2.5. Preview functions in dynamic statistics

The new statistics GUI provides a preview for the current configuration. This must be implemented in the statistic modules and usually returns fake / random data for speed reasons. So for any dynamic (matrix) statistic that provides the method `GetStatElement()` you should also add a method `GetStatElementPreview()`, and for every dynamic (table) statistic that provides `GetStatTable()` you should accordingly add `GetStatTablePreview()`. Otherwise the preview in the new statistics GUI will not work for your statistics. You can find example implementations in the default OTRS statistics.

3.2.6. HTML print discarded

Until OTRS 5, the Perl module `PDF::API2` was not present on all systems. Therefore a fallback HTML print mode existed. With OTRS 5, the module is now bundled and HTML print was dropped. `$LayoutObject->PrintHeader()` and `PrintFooter()` are not available any more. Please remove the HTML print fallback from your code and change it to generate PDF if necessary.

3.2.7. Translation string extraction improved

Until OTRS 5, translatable strings could not be extracted from Perl code and Database XML definitions. This is now possible and makes dummy templates like `AAA*.tt` obsolete. Please see this section for details.

3.3. From OTRS 3.3 to 4

This section lists changes that you need to examine when porting your package from OTRS 3.3 to 4.

3.3.1. New Object Handling

Up to OTRS 4, objects used to be created both centrally and also locally and then handed down to all objects by passing them to the constructors. With OTRS 4 and later versions, there is now an `ObjectManager` that centralizes singleton object creation and access.

This will require you first of all to change all top level Perl scripts (.pl files only!) to load and provide the `ObjectManager` to all OTRS objects. Let's look at `otrs.CheckDB.pl` from OTRS 3.3 as an example:

```
use strict;
use warnings;

use File::Basename;
use FindBin qw($RealBin);
use lib dirname($RealBin);
use lib dirname($RealBin) . '/Kernel/cpan-lib';
use lib dirname($RealBin) . '/Custom';

use Kernel::Config;
use Kernel::System::Encode;
use Kernel::System::Log;
use Kernel::System::Main;
use Kernel::System::DB;

# create common objects
my %CommonObject = ();
$CommonObject{ConfigObject} = Kernel::Config->new();
$CommonObject{EncodeObject} = Kernel::System::Encode->new(%CommonObject);
$CommonObject{LogObject}    = Kernel::System::Log->new(
    LogPrefix    => 'OTRS-otrs.CheckDB.pl',
    ConfigObject => $CommonObject{ConfigObject},
);
$CommonObject{MainObject} = Kernel::System::Main->new(%CommonObject);
$CommonObject{DBObject}   = Kernel::System::DB->new(%CommonObject);
```

We can see that a lot of code is used to load the packages and create the common objects that must be passed to OTRS objects to be used in the script. With OTRS 4, this looks quite different:

```
use strict;
use warnings;

use File::Basename;
use FindBin qw($RealBin);
use lib dirname($RealBin);
use lib dirname($RealBin) . '/Kernel/cpan-lib';
use lib dirname($RealBin) . '/Custom';

use Kernel::System::ObjectManager;

# create common objects
local $Kernel::OM = Kernel::System::ObjectManager->new(
'Kernel::System::Log' => {
    LogPrefix => 'OTRS-otrs.CheckDB.pl',
},
);

# get database object
my $DBObject = $Kernel::OM->Get('Kernel::System::DB');
```

The new code is a bit shorter than the old. It is no longer necessary to load all the packages, just the `ObjectManager`. Subsequently `$Kernel::OM->Get('My::Perl::Package')` can be used to get instances of objects which only have to be created once. The `LogPrefix` setting controls the log messages that `Kernel::System::Log` writes, it could also be omitted.

From this example you can also deduce the general porting guide when it comes to accessing objects: don't store them in `$Self` any more (unless needed for specific reasons). Just fetch and use the objects on demand like `$Kernel::OM->Get('Kernel::System::Log')->Log(...)`. This also has the benefit that the Log object will only be created

if something must be logged. Sometimes it could also be useful to create local variables if an object is used many times in a function, like `$DBObject` in the example above.

There's not much more to know when porting packages that should be loadable by the `ObjectManager`. They should declare the modules they use (via `$Kernel::OM->Get()`) like this:

```
our @ObjectDependencies = (
'Kernel::Config',
'Kernel::System::Log',
'Kernel::System::Main',
);
```

The `@ObjectDependencies` declaration is needed for the `ObjectManager` to keep the correct order when destroying the objects.

Let's look at `Valid.pm` from OTRS 3.3 and 4 to see the difference. Old:

```
package Kernel::System::Valid;

use strict;
use warnings;

use Kernel::System::CacheInternal;

...

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 EncodeObject MainObject)) {
    $Self->{$Object} = $Param{$Object} || die "Got no $Object!";
}

$Self->{CacheInternalObject} = Kernel::System::CacheInternal->new(
    %{$Self},
    Type => 'Valid',
    TTL  => 60 * 60 * 24 * 20,
);

return $Self;
}

...

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

# read cache
my $CacheKey = 'ValidList';
my $Cache = $Self->{CacheInternalObject}->Get( Key => $CacheKey );
return %{$Cache} if $Cache;

# get list from database
return if !$Self->{DBObject}->Prepare( SQL => 'SELECT id, name FROM valid' );

# fetch the result
my %Data;
while ( my @Row = $Self->{DBObject}->FetchrowArray() ) {
    $Data{ $Row[0] } = $Row[1];
}
```

```
# set cache
$Self->{CacheInternalObject}->Set( Key => $CacheKey, Value => \%Data );

return %Data;
}
```

New:

```
package Kernel::System::Valid;

use strict;
use warnings;

our @ObjectDependencies = (
'Kernel::System::Cache',
'Kernel::System::DB',
'Kernel::System::Log',
);

...

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

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

$Self->{CacheType} = 'Valid';
$Self->{CacheTTL}  = 60 * 60 * 24 * 20;

return $Self;
}

...

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

# read cache
my $CacheKey = 'ValidList';
my $Cache    = $Kernel::OM->Get('Kernel::System::Cache')->Get(
    Type => $Self->{CacheType},
    Key  => $CacheKey,
);
return %{$Cache} if $Cache;

# get database object
my $DBObject = $Kernel::OM->Get('Kernel::System::DB');

# get list from database
return if !$DBObject->Prepare( SQL => 'SELECT id, name FROM valid' );

# fetch the result
my %Data;
while ( my @Row = $DBObject->FetchrowArray() ) {
    $Data{ $Row[0] } = $Row[1];
}

# set cache
$Kernel::OM->Get('Kernel::System::Cache')->Set(
    Type  => $Self->{CacheType},
    TTL   => $Self->{CacheTTL},
    Key   => $CacheKey,
    Value => \%Data
);

return %Data;
}
```

You can see that the dependencies are declared and the objects are only fetched on demand. We'll talk about the `CacheInternalObject` in the next section.

3.3.2. `CacheInternalObject` removed

Since `Kernel::System::Cache` is now also able to cache in-memory, `Kernel::System::CacheInternal` was dropped. Please see the previous example for how to migrate your code: you need to use the global Cache object and pass the Type settings with every call to `Get()`, `Set()`, `Delete()` and `CleanUp()`. The TTL parameter is now optional and defaults to 20 days, so you only have to specify it in `Get()` if you require a different TTL value.

Warning

It is especially important to add the Type to `CleanUp()` as otherwise not just the current cache type but the entire cache would be deleted.

3.3.3. Scheduler backend files moved

The backend files of the scheduler moved from `Kernel/Scheduler` to `Kernel/System/Scheduler`. If you have any custom Task Handler modules, you need to move them also.

3.3.4. Update code sections in SOPM files

Code tags in SOPM files have to be updated. Please do not use `$Self` any more. In the past this was used to get access to OTRS objects like the `MainObject`. Please use the `ObjectManager` now. Here is an example for the old style:

```
<CodeInstall Type="post">

# define function name
my $FunctionName = 'CodeInstall';

# create the package name
my $CodeModule = 'var::packagesetup::' . $Param{Structure}->{Name}->{Content};

# load the module
if ( $Self->{MainObject}->Require($CodeModule) ) {

# create new instance
my $CodeObject = $CodeModule->new( %{$Self} );

if ($CodeObject) {

    # start method
    if ( !$CodeObject->$FunctionName(%{$Self}) ) {
        $Self->{LogObject}->Log(
            Priority => 'error',
            Message  => "Could not call method $FunctionName() on $CodeModule.pm."
        );
    }
}

# error handling
else {
    $Self->{LogObject}->Log(
        Priority => 'error',
        Message  => "Could not call method new() on $CodeModule.pm."
    );
}
}

</CodeInstall>
```

Now this should be replaced by:

```
<CodeInstall Type="post"><![CDATA[
$Kernel::OM->Get('var::packagesetup::MyPackage')->CodeInstall();
]]></CodeInstall>
```

3.3.5. New Template Engine

With OTRS 4, the DTL template engine was replaced by Template::Toolkit. Please refer to the Templating section for details on how the new template syntax looks like.

These are the changes that you need to apply when converting existing DTL templates to the new Template::Toolkit syntax:

Table 4.1. Template Changes from OTRS 3.3 to 4

<table>
<tr><th>DTL Tag</th><th>Template::Toolkit tag</th></tr>
<tr><td>$Data{"Name"}</td><td>[% Data.Name %]</td></tr>
<tr><td>$Data{"Complex-Name"}</td><td>[% Data.item("Complex-Name") %]</td></tr>
<tr><td>$QData{"Name"}</td><td>[% Data.Name | html %]</td></tr>
<tr><td>$QData{"Name", "$Length"}</td><td>[% Data.Name | truncate($Length) | html %]</td></tr>
<tr><td>$LQData{"Name"}</td><td>[% Data.Name | uri %]</td></tr>
<tr><td>$Quote{"Text", "$Length"}</td><td>cannot be replaced directly, see examples below</td></tr>
<tr><td>$Quote{"$Config{"Name"}"}</td><td>[% Config("Name") | html %]</td></tr>
<tr><td>$Quote{"$Data{"Name"}", "$Length"}</td><td>[% Data.Name | truncate($Length) | html %]</td></tr>
<tr><td>$Quote{"$Data{"Content"}","$QData{"MaxLength"}"}</td><td>[% Data.Name | truncate(Data.MaxLength) | html %]</td></tr>
<tr><td>$Quote{"$Text{"$Data{"Content"}"}","$QData{"MaxLength"}"}</td><td>[% Data.Content | Translate | truncate(Data.MaxLength) | html %]</td></tr>
<tr><td>$Config{"Name"}</td><td>[% Config("Name") %]</td></tr>
<tr><td>$Env{"Name"}</td><td>[% Env("Name") %]</td></tr>
<tr><td>$QEnv{"Name"}</td><td>[% Env("Name") | html %]</td></tr>
<tr><td>$Text{"Text with %s placeholders", "String"}</td><td>[% Translate("Text with %s placeholders", "String") | html %]</td></tr>
<tr><td>$Text{"Text with dynamic %s placeholders", "$QData{Name}"}</td><td>[% Translate("Text with dynamic %s placeholders", Data.Name) | html %]</td></tr>
<tr><td>'$JSText{"Text with dynamic %s placeholders", "$QData{Name}"}'</td><td>[% Translate("Text with dynamic %s placeholders", Data.Name) | JSON %]</td></tr>
<tr><td>"$JSText{"Text with dynamic %s placeholders", "$QData{Name}"}"</td><td>[% Translate("Text with dynamic %s placeholders", Data.Name) | JSON %]</td></tr>
<tr><td>$TimeLong{"$Data{"CreateTime"}"}</td><td>[% Data.CreateTime | Localize("TimeLong") %]</td></tr>
<tr><td>$TimeShort{"$Data{"CreateTime"}"}</td><td>[% Data.CreateTime | Localize("TimeShort") %]</td></tr>
<tr><td>$Date{"$Data{"CreateTime"}"}</td><td>[% Data.CreateTime | Localize("Date") %]</td></tr>
</table>

DTL Tag	Template::Toolkit tag
`<-- dtl:block:Name -->...<-- dtl:block:Name -->`	`[% RenderBlockStart("Name") %]...[% RenderBlockEnd("Name") %]`
`<-- dtl:js_on_document_complete -->...<-- dtl:js_on_document_complete -->`	`[% WRAPPER JSOnDocumentComplete %]...[% END %]`
`<-- dtl:js_on_document_complete_placeholder -->`	`[% PROCESS JSOnDocumentCompleteInsert %]`
`$Include{"Copyright"}`	`[% InsertTemplate("Copyright") %]`

There is also a helper script `bin/otrs.MigrateDTLtoTT.pl` that will automatically port the DTL files to Template::Toolkit syntax for you. It might fail if you have errors in your DTL, please correct these first and re-run the script afterwards.

There are a few more things to note when porting your code to the new template engine:

- All language files must now have the `use utf8;` pragma.
- `Layout::Get()` is now deprecated. Please use `Layout::Translate()` instead.
- All occurrences of `$Text{""}` in Perl code must now be replaced by calls to `Layout::Translate()`.

  This is because in DTL there was no separation between template and data. If DTL-Tags were inserted as part of some data, the engine would still parse them. This is no longer the case in Template::Toolkit, there is a strict separation of template and data.

  Hint: should you ever need to interpolate tags in data, you can use the `Interpolate` filter for this (`[% Data.Name | Interpolate %]`). This is not recommended for security and performance reasons!

- For the same reason, dynamically injected JavaScript that was enclosed by `dtl:js_on_document_complete` will not work any more. Please use `Layout::AddJSOnDocumentComplete()` instead of injecting this as template data.

  You can find an example for this in `Kernel/System/DynamicField/Driver/BaseSelect.pm`.

- Please be careful with `pre` output filters (the ones configured in `Frontend::Output::FilterElementPre`). They still work, but they will prevent the template from being cached. This could lead to serious performance issues. You should definitely not have any `pre` output filters that operate on all templates, but limit them to certain templates via configuration setting.

  The `post` output filters (`Frontend::Output::FilterElementPost`) don't have such strong negative performance effects. However, they should also be used carefully, and not for all templates.

3.3.6. New FontAwesome version

With OTRS 4, we've also updated FontAwesome to a new version. As a consequence, the icons CSS classes have changed. While previously icons were defined by using a schema like `icon-{iconname}`, it is now `fa fa-{iconname}`.

Due to this change, you need to make sure to update all custom frontend module registrations which make use of icons (e.g. for the top navigation bar) to use the new schema. This is also true for templates where you're using icon elements like `<i class="icon-{iconname}"></i>`.

3.3.7. Unit Tests

With OTRS 4, in Unit Tests `$Self` no longer provides common objects like the `MainObject`, for example. Please always use `$Kernel::OM->Get('...')` to fetch these objects.

3.3.8. Custom Ticket History types

If you use any custom ticket history types, you have to take two steps for them to be displayed correctly in `AgentTicketHistory` of OTRS 4+.

Firstly, you have to register your custom ticket history types via SysConfig. This could look like:

```
<ConfigItem Name="Ticket::Frontend::HistoryTypes###100-MyCustomModule" Required="1"
 Valid="1">
<Description Translatable="1">Controls how to display the ticket history entries as readable
 values.</Description>
<Group>Ticket</Group>
<SubGroup>Frontend::Agent::Ticket::ViewHistory</SubGroup>
<Setting>
    <Hash>
        <Item Key="MyCustomType" Translatable="1">Added information (%s)</Item>
    </Hash>
</Setting>
</ConfigItem>
```

The second step is to translate the English text that you provided for the custom ticket history type in your translation files, if needed. That's it!

If you are interested in the details, please refer to this commit for additional information about the changes that happened in OTRS.

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).

It is highly recommended that you use the OTRS code quality checker OTRSCodePolicy as described in the development environment chapter even before sending in your contributions. If your code does not validate against this tool, it will likely not be accepted.

The easiest way to send 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 on 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. The translations are contributed and maintained mainly by OTRS users, so *your* help is needed.

2.1. Updating an existing translation

Starting with OTRS 4, all translations of the OTRS GUI and the public extension modules are managed via Transifex. The OTRS project on Transifex can be found at https://www.transifex.com/otrs/OTRS/.

To contribute to a translation of the OTRS GUI, an extension module or a manual, please sign up for a free translators account on Transifex. Then you can join your language team and start updating your translation. No additional software or files required. The OTRS developers will download the translations from time to time into the OTRS source code repositories, you don't have to submit them anywhere.

2.2. Adding a new frontend translation

If you want to translate the OTRS framework into a new language, you can propose a new language translation on the Transifex OTRS project page. After it is approved, you can just start translating.

3. Translating the Documentation

The OTRS admin manual can be translated via Transifex as described in the section on translating OTRS. You can join a language team on Transifex to improve an existing translation or even suggest a new language to translate the admin manual to.

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 XML tags intact. Regular < and > signs that are escaped in the source text must also be escaped in the translations (like <someone@example.com>). 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).

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 generally not be longer than 120 characters, unless it is necessary for special reasons.

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 parenthesis.

```
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.

```
# --
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --
```

Executable files (`*.pl`) have a special header.

```
#!/usr/bin/perl
# --
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) 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 General Public License
# along with this program. If not, see https://www.gnu.org/licenses/gpl-3.0.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 and 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 statements and is only allowed when involves return statements in functions or to end a loop or to go next iteration.

This is correct:

```
next ITEM if !$ItemId;
```

This is wrong:

```
return $Self->{LogObject}->Log(
    Priority => 'error',
    Message  => 'ItemID needed!',
) if !$ItemId;
```

This is less maintainable than this:

```
if( !$ItemId ) {
    $Self->{LogObject}->Log( ... );
    return;
}
```

This is correct:

```
for my $Needed ( 1 .. 10 ) {
    next if $Needed == 5;
    last  if $Needed == 9;
}
```

This is wrong:

```
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 DateTimeObject instead of the builtin functions like time(), localtime(), etc.

4.1.2.3. Regular Expressions

For regular expressions *in the source code*, we always use the m// operator with curly braces as delimiters. We also use the modifiers x, m and s by default. The x modifier allows you to comment your regex and use spaces to visually separate 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;
```

An exception to the convention above applies to all cases where regular expressions are not written statically in the code but instead are *supplied by users* in one form or another (for example via SysConfig or in a PostMaster filter configuration). Any evaluation of such a regular expression has to be done without any modifiers (e.g. $Variable =~ m{$Regex}) in order to match the expectation of (mostly inexperienced) users and also to be backwards compatible.

If modifiers are strictly necessary for user supplied regular expressions, it is always possible to use embedded modifiers (e.g. (?:(?i)SmAlL oR lArGe)). For details, please see perlretut.

Usage of the r modifier is encouraged, e.g. if you need to extract part of a string into another variable. This modifier keeps the matched variable intact and instead provides the substitution result as a return value.

Example: Use this...

```
my $NewText = $Text =~ s{
    \A
    Prefix
    (
        Text
    )
}
{NewPrefix$1Postfix}xmsr;
```

instead of this...

```
my $NewText = $Text;
$NewText =~ s{
    \A
    Prefix
    (
        Text
    )
}
{NewPrefix$1Postfix}xms;
```

If you want to match for start and end of a *string*, you should generally use \A and \z instead of the more generic ^ and $ unless you really need to match start or end of *lines* within a multiline string.

```
$Text =~ m{
    \A      # beginning of the string
    Content # some string
    \z      # end of the string
}xms;

$MultilineText =~ m{
    \A                          # beginning of the string
    .*
    (?: \n Content $ )+          # one or more lines containing the same string
    .*
    \z                          # end of the string
}xms;
```

Usage of named capture groups is also encouraged, particularly for multi-matches. Named capture groups are easier to read/understand, prevent mix-ups when matching more than one capture group and allow extension without accidentally introducing bugs.

Example: Use this...

```
$Contact =~ s{
    \A
    [ ]*
    (?'TrimmedContact'
        (?'FirstName' \w+ )
        [ ]+
        (?'LastName' \w+ )
    )
    [ ]+
    (?'Email' [^ ]+ )
    [ ]*
    \z
}
{$+{TrimmedContact}}xms;
my $FormattedContact = "$+{LastName}, $+{FirstName} ($+{Email})";
```

instead of this...

```
$Contact =~ s{
    \A
    [ ]*
    (
        ( \w+ )
        [ ]+
        ( \w+ )
    )
    [ ]+
```

```
    ( [^ ]+ )
    [ ]*
    \z
}
{$1}xms;
my $FormattedContact = "$3, $2 ($4)";
```

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`. Examples: `@TicketIDs`, `$Output`, `StateSet()`, etc.

4.1.2.5. Variables

4.1.2.5.1. Declaration

If you have several variables, you can declare them in one line if they "belong together":

```
my ( $Minute, $Hour, $Year );
```

Otherwise break it into separate lines:

```
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::DateTime;
```

This is wrong:

```
package Kernel::System::ITSMConfigItem::History;

use Kernel::System::User;
use Kernel::System::DateTime;

use strict;
use warnings;
```

4.1.2.7.2. 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 backend modules

'NAME' section

This section should include the module name, ' - ' as separator and a brief description of the module purpose.

```
=head1 NAME

Kernel::System::MyModule - Functions to read from and write to files
```

'SYNOPSIS' section

This section should give a short usage example of commonly used module functions.

Usage of this section is optional.

```
=head1 SYNOPSIS

my $Object = $Kernel::OM->Get('Kernel::System::MyModule');

Read data

    my $FileContent = $Object->Read(
        File => '/tmp/testfile',
    );

Write data

    $Object->Write(
        Content => 'my file content',
        File    => '/tmp/testfile',
    );
```

'DESCRIPTION' section

This section should give more in-depth information about the module if deemed necessary (instead of having a long 'NAME' section).

Usage of this section is optional.

```
=head1 DESCRIPTION

This module does not only handle files.

It is also able to:
- brew coffee
- turn lead into gold
- bring world peace
```

'PUBLIC INTERFACE' section

This section marks the begin of all functions that are part of the API and therefore meant to be used by other modules.

```
=head1 PUBLIC INTERFACE
```

'PRIVATE FUNCTIONS' section

This section marks the begin of private functions.

Functions below are not part of the API, to be used only within the module and therefore not considered stable.

It is advisable to use this section whenever one or more private functions exist.

```
=head1 PRIVATE FUNCTIONS
```

4.1.3.1.2. 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:

```
=head2 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. Code Comments

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 unnecessary 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.

Please adhere to the following guidelines:

Make the code so readable that comments are not needed, if possible.

It's always preferable to write code so that it is very readable and self-explaining, for example with precise variable and function names.

Don't say what the code says (DRY -> Don't repeat yourself).

Don't repeat (obvious) code in the comments.

```
# WRONG:

# get config object
my $ConfigObject = $Kernel::OM->Get('Kernel::Config');
```

Document *why* the code is there, not how it works.

Usually, code comments should explain the *purpose* of code, not how it works in detail. There might be exceptions for specially complicated code, but in this case also a refactoring to make it more readable could be commendable.

Document pitfalls.

Everything that is unclear, tricky or that puzzled you during development should be documented.

Use full-line sentence-style comments to document algorithm paragraphs.

Always use full sentences (uppercase first letter and final period). Subsequent lines of a sentence should be indented.

```
# Check if object name is provided.
if ( !$_[1] ) {
    $_[0]->_DieWithError(
        Error => "Error: Missing parameter (object name)",
    );
}

# Record the object we are about to retrieve to potentially build better error messages.
# Needs to be a statement-modifying 'if', otherwise 'local' is local
#   to the scope of the 'if'-block.
local $CurrentObject = $_[1] if !$CurrentObject;
```

Use short end-of-line comments to add detail information.

These can either be a complete sentence (capital first letter and period) or just a phrase (lowercase first letter and no period).

```
$BuildMode = oct $Param{Mode};   # *from* octal, not *to* octal

# or

$BuildMode = oct $Param{Mode};   # Convert *from* octal, not *to* octal.
```

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 necessary.

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 template 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. Code Comments

The commenting guidelines for Perl code also apply to JavaScript.

- 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 `$.on()` instead of the event-shorthand methods of jQuery for better readability (wrong: `$SomeObject.click(...)`, right: `$SomeObject.on('click', ...)`.
- If you `$.on()` events, make sure to `$.off()` them beforehand, to make sure that events will not be bound twice, should the code be executed another time.
- Make sure to use `$.on()` with namespacing, such as `$.on('click.<Name>')`.

4.3. HTML

- Use HTML 5 notation. Don't use self-closing tags for non-void elements (such as div, span, etc.).
- Use proper intendation. Elements which contain other non-void child elements should not be on the same level as their children.
- Don't use HTML elements for layout reasons (e.g. using `br` elements for adding space to the top or bottom of other elements). Use the proper CSS classes instead.
- Don't use inline CSS. All CSS should either be added by using predefined classes or (if necessary) using JavaScript (e.g. for showing/hiding elements).
- Don't use JavaScript in TT templates. All needed JavaScript should be part of the proper library for a certain frontend module or of a proper global library. If you need to pass JavaScript data to the frontend, use `$LayoutObject->AddJSData()`.

4.4. 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.

4.4.1. 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.

4.4.2. 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 impairments 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 open source screen readers which you can install and use:

- NVDA, a screen reader for Windows.
- 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 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 target 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:

- *Rule*: 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 application. 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 screen readers.
  - 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 necessary 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 user 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 (especially 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 key-

board. 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 user 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 screen reader can choose the right speech synthesis engine.

  Example: `<html lang="fr">...</html>`

7. Unit Tests

OTRS provides a test suite which can be used to develop and run unit tests for all system related code.

7.1. Creating a test file

The test files are stored in .t files under `scripts/test/*.t`. For example, let's take a look at the file `scripts/test/Calendar.t` for the Calendar class.

Every test file should ideally instantiate unit test helper object at the start, so it can benefit from some built-in methods provided by it:

```
# --
# Copyright (C) 2001-2021 OTRS AG, https://otrs.com/
# --
# This software comes with ABSOLUTELY NO WARRANTY. For details, see
# the enclosed file COPYING for license information (GPL). If you
# did not receive this file, see https://www.gnu.org/licenses/gpl-3.0.txt.
# --
```

```
use strict;
use warnings;
use utf8;

use vars (qw($Self));

$Kernel::OM->ObjectParamAdd(
    'Kernel::System::UnitTest::Helper' => {
        RestoreDatabase => 1,
    },
);
my $Helper = $Kernel::OM->Get('Kernel::System::UnitTest::Helper');
```

By providing `RestoreDatabase` parameter to helper constructor, any database statement executed during the unit test will be rolled back at the end, making sure no permanent change has been done.

Like any other test suite, OTRS provides assertion methods which can be used to test conditions. For example, this is how we create a test user and test that it has been indeed created:

```
my $UserLogin = $Helper->TestUserCreate();
my $UserID = $UserObject->UserLookup( UserLogin => $UserLogin );

$Self->True(
    $UserID,
    "Test user $UserID created"
);
```

Please consult API section below for complete list of assertion methods.

It's always good practice to create random data in unit tests, which can help distinguish it from previously added data. Use random methods from API to get the strings and include them in your parameters:

```
my $RandomID = $Helper->GetRandomID();

# create test group
my $GroupName = 'test-calendar-group-' . $RandomID;
my $GroupID   = $GroupObject->GroupAdd(
    Name    => $GroupName,
    ValidID => 1,
    UserID  => 1,
);

$Self->True(
    $GroupID,
    "Test group $GroupID created"
);
```

Good developers make their unit test easy to maintain. Consider putting all test cases in an array and then iterate over them with some code. This will provide an easy way to extend the test later:

```
#
# Tests for CalendarCreate()
#
my @Tests = (
    {
        Name    => 'CalendarCreate - No params',
        Config  => {},
```

```
        Success => 0,
    },
    {
        Name   => 'CalendarCreate - All required parameters',
        Config => {
            CalendarName => "Calendar-$RandomID",
            Color        => '#3A87AD',
            GroupID      => $GroupID,
            UserID       => $UserID,
        },
        Success => 1,
    },
    {
        Name   => 'CalendarCreate - Same name',
        Config => {
            CalendarName => "Calendar-$RandomID",
            Color        => '#3A87AD',
            GroupID      => $GroupID,
            UserID       => $UserID,
        },
        Success => 0,
    },
);

for my $Test (@Tests) {

    # make the call
    my %Calendar = $CalendarObject->CalendarCreate(
        %{ $Test->{Config} },
    );

    # check data
    if ( $Test->{Success} ) {
        for my $Key (qw(CalendarID GroupID CalendarName Color CreateTime CreateBy ChangeTime
 ChangeBy ValidID)) {
            $Self->True(
                $Calendar{$Key},
                "$Test->{Name} - $Key exists",
            );
        }

        KEY:
        for my $Key ( sort keys %{ $Test->{Config} } ) {
            next KEY if $Key eq 'UserID';

            $Self->IsDeeply(
                $Test->{Config}->{$Key},
                $Calendar{$Key},
                "$Test->{Name} - Data for $Key",
            );
        }
    }
    else {
        $Self->False(
            $Calendar{CalendarID},
            "$Test->{Name} - No success",
        );
    }
}
```

7.2. Prerequisites for testing

To be able to run the unit tests, you need to have all optional Perl modules installed, except those for different database backends than what you are using. Run `bin/otrs.CheckModules.pl` to verify your module installation.

You also need to have an instance of the OTRS web frontend running on the FQDN that is configured in your local OTRS's `Config.pm` file. This OTRS instance must use the same database that is configured for the unit tests.

7.3. Testing

To run your tests, just use `bin/otrs.Console.pl Dev::UnitTest::Run --test Calendar` to use `scripts/test/Calendar.t`.

```
shell:/opt/otrs> bin/otrs.Console.pl Dev::UnitTest::Run --test Calendar
+-------------------------------------------------------------------+
/opt/otrs/scripts/test/Calendar.t:
+-------------------------------------------------------------------+
.................................................................................................
=====================================================================
yourhost ran tests in 2s for OTRS 6.0.x git
All 97 tests passed.
shell:/opt/otrs>
```

You can even run several tests at once, just supply additional test arguments to the command:

```
shell:/opt/otrs> bin/otrs.Console.pl Dev::UnitTest::Run --test Calendar --test Appointment
+-------------------------------------------------------------------+
/opt/otrs/scripts/test/Calendar.t:
+-------------------------------------------------------------------+
.................................................................................................
+-------------------------------------------------------------------+
/opt/otrs/scripts/test/Calendar/Appointment.t:
+-------------------------------------------------------------------+
...............................................................................................................
=====================================================================
yourhost ran tests in 5s for OTRS 6.0.x git
All 212 tests passed.
shell:/opt/otrs>
```

If you execute `bin/otrs.Console.pl Dev::UnitTest::Run` without any argument, it will run all tests found in the system. Please note that this can take some time to finish.

Provide `--verbose` argument in order to see messages about successful tests too. Any errors encountered during testing will be displayed regardless of this switch, provided they are actually raised in the test.

7.4. Unit Test API

OTRS provides API for unit testing that was used in the previous example. Here we'll list the most important functions, please also see the online API reference of `Kernel::System::UnitTest`.

`True()`

This function tests whether given scalar value is a true value in Perl.

```
$Self->True(
    1,
    'Scalar 1 is always evaluated as true'
);
```

`False()`

This function tests whether given scalar value is a false value in Perl.

```
$Self->False(
```

```
    0,
    'Scalar 0 is always evaluated as false'
);
```

`Is()`

This function tests whether the given scalar variables are equal.

```
$Self->Is(
    $A,
    $B,
    'Test Name',
);
```

`IsNot()`

This function tests whether the given scalar variables are unequal.

```
$Self->IsNot(
    $A,
    $B,
    'Test Name'
);
```

`IsDeeply()`

This function compares complex data structures for equality. $A and $B have to be references.

```
$Self->IsDeeply(
    $A,
    $B,
    'Test Name'
);
```

`IsNotDeeply()`

This function compares complex data structures for inequality. $A and $B have to be references.

```
$Self->IsNotDeeply(
    $A,
    $B,
    'Test Name'
);
```

Besides this, unit test helper object also provides some helpful methods for common test conditions. For full reference, please see the online API reference of `Kernel::System::UnitTest::Helper`.

`GetRandomID()`

This function creates a random ID that can be used in tests as a unique identifier. It is guaranteed that within a test this function will never return a duplicate.

Note

Please note that these numbers are not really random and should only be used to create test data.

```
my $RandomID = $Helper->GetRandomID();
# $RandomID = 'test6326004144100003';
```

`TestUserCreate()`

This function creates a test user that can be used in tests. It will be set to invalid automatically during the destructor. It returns the login name of the new user, the password is the same.

```
my $TestUserLogin = $Helper->TestUserCreate(
    Groups   => ['admin', 'users'],          # optional, list of groups to add this user
 to (rw rights)
    Language => 'de',                        # optional, defaults to 'en' if not set
);
```

`FixedTimeSet()`

This functions makes it possible to override the system time as long as this object lives. You can pass an optional time parameter that should be used, if not, the current system time will be used.

Note

All calls to methods of `Kernel::System::Time` and `Kernel::System::DateTime` will use the given time afterwards.

```
$HelperObject->FixedTimeSet(366475757);         # with Timestamp
$HelperObject->FixedTimeSet($DateTimeObject);   # with previously created DateTime
 object
$HelperObject->FixedTimeSet();                  # set to current date and time
```

`FixedTimeUnset()`

This functions restores the regular system time behavior.

`FixedTimeAddSeconds()`

This functions adds a number of seconds to the fixed system time which was previously set by `FixedTimeSet()`. You can pass a negative value to go back in time.

`ConfigSettingChange()`

This functions temporarily changes a configuration setting system wide to another value, both in the current instance of the `ConfigObject` and also in the system configuration on disk. This will be reset when the `Helper` object is destroyed.

Note

Please note that this will not work correctly in clustered environments.

```
$Helper->ConfigSettingChange(
    Valid => 1,            # (optional) enable or disable setting
    Key   => 'MySetting',  # setting name
    Value => { ... } ,     # setting value
);
```

`CustomCodeActivate()`

This function will temporarily include custom code in the system. For example, you may use this to redefine a subroutine from another class. This change will persist for remainder of the test. All code will be removed when the Helper object is destroyed.

Note

Please note that this will not work correctly in clustered environments.

```
$Helper->CustomCodeActivate(
    Code => q^
use Kernel::System::WebUserAgent;
package Kernel::System::WebUserAgent;
use strict;
use warnings;
{
    no warnings 'redefine';
    sub Request {
        my $JSONString = '{"Results":{},"ErrorMessage":"","Success":1}';
        return (
            Content => \$JSONString,
            Status  => '200 OK',
        );
    }
}
1;^,
    Identifier => 'News',   # (optional) Code identifier to include in file name
);
```

`ProvideTestDatabase()`

This function will provide a temporary database for the test. Please first define test database settings in `Kernel/Config.pm`, i.e:

```
$Self->{TestDatabase} = {
    DatabaseDSN  => 'DBI:mysql:database=otrs_test;host=127.0.0.1;',
    DatabaseUser => 'otrs_test',
    DatabasePw   => 'otrs_test',
};
```

The method call will override global database configuration for duration of the test, i.e. temporary database will receive all calls sent over system `DBObject`.

All database contents will be automatically dropped when the `Helper` object is destroyed.

This method returns `undef` in case the test database is not configured. If it is configured, but the supplied XML cannot be read or executed, this method will `die()` to interrupt the test with an error.

```
$Helper->ProvideTestDatabase(
    DatabaseXMLString => $XML,      # (optional) OTRS database XML schema to execute
                                    # or
    DatabaseXMLFiles => [           # (optional) List of XML files to load and execute
        '/opt/otrs/scripts/database/otrs-schema.xml',
        '/opt/otrs/scripts/database/otrs-initial_insert.xml',
    ],
);
```

Appendix A. Additional Resources

otrs.com

The OTRS website with source code, documentation and news is available at www.otrs.com. Here you can also find information about professional services and OTRS Administrator training seminars from OTRS Group, the creator of OTRS.

Online API Library

The OTRS developer API documentation is available for Perl and JavaScript.

Developer Mailing List

The OTRS developer mailing list is available at https://lists.otrs.org/.