OTOBO Developer Manual

Release 10.1

Rother OSS GmbH

5[illegible] 09, 2024

Contents

1 Getting Started **3**

1.1 Development Environment 3
1.1.1 Obtain the Source Code 3
1.1.2 Useful Tools 3
1.1.3 Linking Expansion Modules 4
1.2 Architecture Overview 4
1.2.1 Directories 7
1.2.2 Files 7
1.2.3 Core Modules 7
1.2.4 Front End Handle 8
1.2.5 Front End Modules 8
1.2.6 CMD Front End 8
1.2.7 Generic Interface Modules 8
1.2.8 Scheduler Task Handler Modules 10
1.2.9 Database 10

2 OTOBO Internals - How it Works **11**

2.1 Config Mechanism 11
2.1.1 `Defaults.pm`: OTOBO Default Configuration 11
2.1.2 Automatically Generated Configuration Files 11
2.1.3 XML Configuration Files 11
2.1.4 Accessing Config Options at Runtime 20
2.2 Database Mechanism 20
2.2.1 SQL 20
2.2.2 XML 22
2.2.3 Database Drivers 25
2.2.4 Supported Databases 25
2.3 Log Mechanism 25
2.3.1 System Log 25
2.3.2 Communication Log 26
2.4 Date and Time 28
2.4.1 Introduction 28
2.4.2 Creation of a `DateTime` Object 28
2.4.3 Time Zones 29
2.4.4 Method Summary 29
2.4.5 Deprecated Package `Kernel::System::Time` 30

2.5 Skins . . . 31
2.5.1 Skin Basics . . . 31
2.5.2 How Skins Are Loaded . . . 31
2.5.3 Creating a New Skin . . . 32
2.6 The CSS and JavaScript Loader . . . 34
2.6.1 How it works . . . 34
2.6.2 Basic Operation . . . 34
2.6.3 Configuring the Loader: JavaScript . . . 35
2.6.4 Configuring the Loader: CSS . . . 37
2.7 Templating Mechanism . . . 37
2.7.1 Template Commands . . . 37
2.7.2 Using a Template File . . . 45
2.8 Creating Your Own Themes . . . 45
2.9 Localization / Translation Mechanism . . . 45
2.9.1 Marking Translatable Strings in the Source Files . . . 46
2.9.2 Collecting Translatable Strings Into The Translation Database . . . 46
2.9.3 The Translation Process Itself . . . 48
2.9.4 Using The Translated Data From The Code . . . 48

3 How to Extend OTOBO 49

3.1 Writing A New OTOBO Front End Module . . . 49
3.1.1 What we want to write . . . 49
3.1.2 Default Config File . . . 49
3.1.3 Front End Module . . . 51
3.1.4 Core Module . . . 52
3.1.5 Template File . . . 54
3.1.6 Language File . . . 54
3.1.7 Summary . . . 54
3.2 Writing A New OTOBO Front End Component . . . 54
3.2.1 The Goal . . . 55
3.2.2 Using The Skeleton Command . . . 55
3.2.3 The Route Configuration . . . 55
3.2.4 Component Template Code . . . 56
3.2.5 Component Core Code . . . 57
3.2.6 Component Style Code . . . 57
3.2.7 Passing Parameters to the Route Component . . . 58
3.2.8 Component Folders . . . 58
3.2.9 Packaging Additional Vendor Modules . . . 59
3.3 Using the power of the OTOBO module layers . . . 61
3.3.1 Agent Authentication Module . . . 62
3.3.2 Authentication Synchronization Module . . . 65
3.3.3 Customer Authentication Module . . . 66
3.3.4 Customer User Preferences Module . . . 69
3.3.5 Queue Preferences Module . . . 72
3.3.6 Service Preferences Module . . . 75
3.3.7 SLA Preferences Module . . . 77
3.3.8 Log Module . . . 80
3.3.9 Output Filter . . . 82
3.3.10 Stats Module . . . 84
3.3.11 Ticket Number Generator Modules . . . 101
3.3.12 Ticket Event Module . . . 102
3.3.13 Dashboard Module . . . 104
3.3.14 Notification Module . . . 108
3.3.15 Ticket Menu Module . . . 110

3.3.16 Root Application Module Layer . . . 112
3.3.17 Network Transport . . . 113
3.3.18 Mapping . . . 118
3.3.19 Invoker . . . 123
3.3.20 Operation . . . 126
3.3.21 OTOBO Daemon . . . 136
3.3.22 OTOBO Scheduler . . . 140
3.3.23 Overview . . . 142
3.3.24 Dynamic Fields Framework . . . 142
3.3.25 Dynamic Field Interaction With Front End Modules . . . 148
3.3.26 How To Extend The Dynamic Fields . . . 152
3.3.27 Create New Dynamic Field . . . 154
3.3.28 Creating a Dynamic Field Functionality Extension . . . 179
3.3.29 Ticket Postmaster Module . . . 185
3.3.30 Process Management . . . 187
3.3.31 Autoload Modules . . . 194

4 How to Publish Your OTOBO Extensions 197
4.1 Package Management . . . 197
4.1.1 Package Distribution . . . 197
4.1.2 Package Commands . . . 197
4.2 Package Building . . . 198
4.2.1 Package Spec File . . . 198
4.2.2 Example .sopm . . . 203
4.2.3 Package Build . . . 204
4.2.4 Package Life Cycle . . . 204
4.3 Package Porting . . . 205
4.3.1 Front End Messages . . . 205
4.3.2 Styling Improvements . . . 206
4.3.3 Encode API Changed . . . 207
4.3.4 LinkObject API Changed . . . 208
4.3.5 Event Handling Changes . . . 209
4.3.6 `MojoUserAgent` Added, `WebUserAgent` Deprecated . . . 210

5 Documentation 213
5.1 Documentation Infrastructure . . . 213
5.2 reStructuredText Primer . . . 213
5.2.1 Headings . . . 214
5.2.2 Paragraphs . . . 214
5.2.3 Inline Markups . . . 214
5.2.4 Lists . . . 215
5.2.5 Literal Blocks . . . 215
5.2.6 Tables . . . 216
5.2.7 Hyperlinks . . . 216
5.2.8 Images . . . 216
5.2.9 Colored Boxes . . . 217
5.3 Style Guide . . . 217
5.3.1 Writing Content . . . 217
5.3.2 Screenshots . . . 218
5.3.3 Capitalization in Documentation . . . 220
5.3.4 Buttons and Screen Names . . . 221
5.3.5 Wording . . . 221
5.3.6 Variable Names . . . 222
5.4 Translating the Documentation . . . 222

6 Contributing to OTOBO 225
6.1 Sending Contributions . . . 225
6.2 Translating . . . 226
6.3 Code Style Guide . . . 226
6.3.1 Perl . . . 226
6.3.2 JavaScript . . . 237
6.3.3 HTML . . . 239
6.3.4 CSS . . . 239
6.4 User Interface Design . . . 240
6.4.1 Capitalization . . . 240
6.5 Accessibility Guide . . . 240
6.5.1 Accessibility Basics . . . 241
6.5.2 Accessibility Standards . . . 241
6.5.3 Implementation guidelines . . . 242
6.6 Unit Tests . . . 244
6.6.1 Creating a Test File . . . 244
6.6.2 Prerequisites for Testing . . . 246
6.6.3 Testing . . . 246
6.6.4 Unit Test API . . . 247

7 Additional Resources 251

This work is copyrighted by OTRS AG (https://otrs.com), Zimmersmühlenweg 11, 61440 Oberursel, Germany.

Copyright © for modifications and amendments 2019-2020 ROTHER OSS GmbH (https://otobo.de), Oberwalting 31, 94339 Leiblfing, Germany

Terms and Conditions OTRS: Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found on the GNU website.

Terms and Conditions Rother OSS: Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "COPYING".

Published by: Rother OSS GmbH, (https://otobo.de), Oberwalting 31, 94339 Leiblfing, Germany.

Authors: OTRS AG (original version), Rother OSS GmbH (https://otobo.de).

CHAPTER 1

Getting Started

OTOBO 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 OTOBO modules and applications based on the OTOBO styleguides.

1.1 Development Environment

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

1.1.1 Obtain the Source Code

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 clone the Git repository by using the following command:

```
shell> git clone git@github.com:RotherOSS/otobo.git -b master
```

For a specific branch like OTOBO 6:

```
shell> git clone git@github.com:RotherOSS/otobo.git -b rel-6_0
```

Please configure the OTOBO system according to the installation instructions.

1.1.2 Useful Tools

Clone the module-tools module too, for your development environment. It contains a number of useful tools:

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

OTOBOCodePolicy is a code quality checker that enforces the use of common coding standards also for the OTOBO 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.

```
shell> git clone git@github.com:RotherOSS/CodePolicy.git
```

1.1.3 Linking Expansion Modules

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

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

As soon as the linking is completed, the system configuration must be rebuilt to register the module in OTOBO. Additional SQL or Perl code from the module must also be executed.

Example:

```
shell> ~/src/otobo/bin/otobo.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 OTOBO enter the following command:

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

1.2 Architecture Overview

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

The OTOBO Generic Interface continues OTOBO modularity. The next picture shows the basic layer architecture of the Generic Interface.

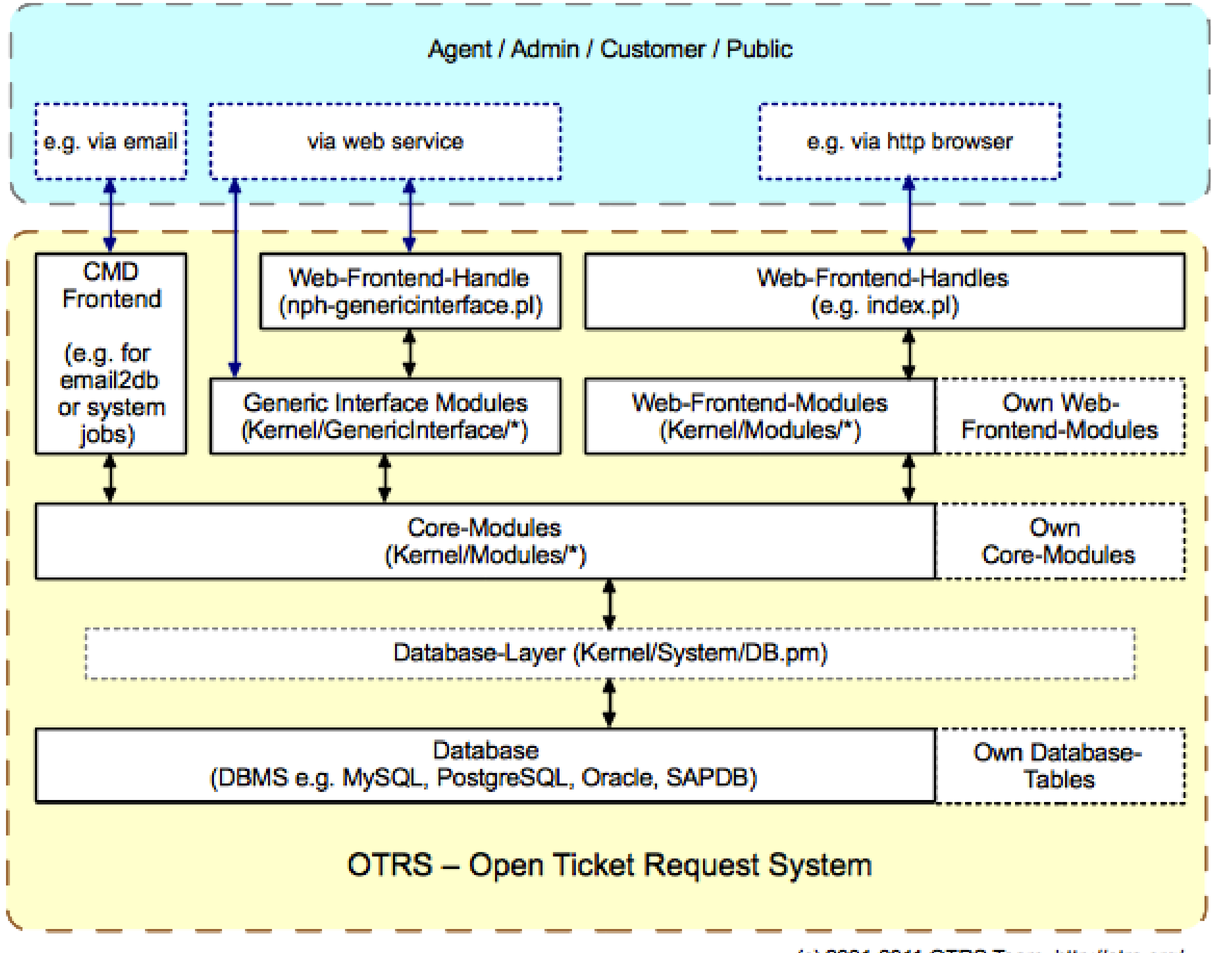

Fig. 1.1: OTOBO Architecture

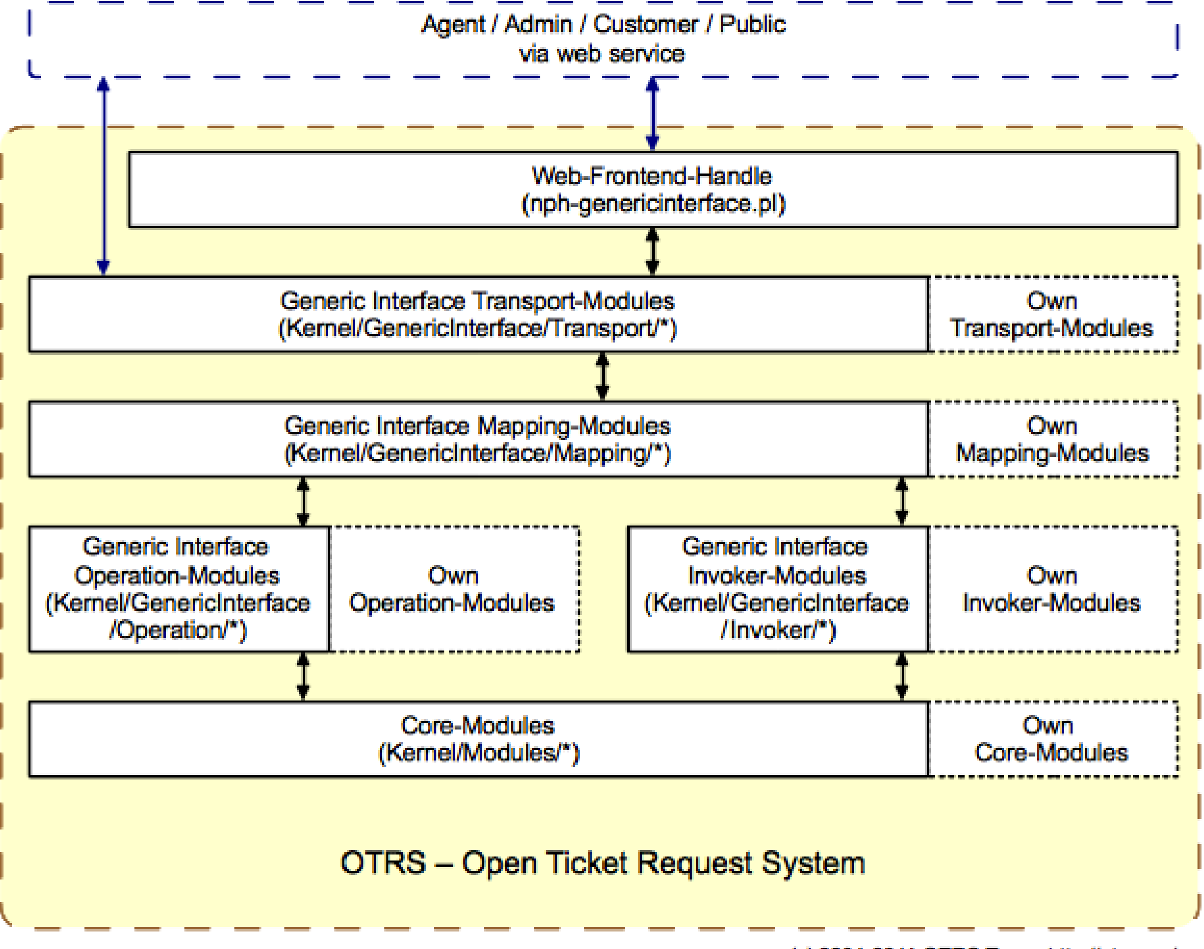

Fig. 1.2: Generic Interface Architecture

1.2.1 Directories

Directory	Description
bin/	command line tools
bin/cgi-bin/	web handle
bin/fcgi-bin/	fast CGI web handle
Kernel	application code base
Kernel/Config/	configuration files
Kernel/Config/Files	configuration files
Kernel/GenericInterface/	the Generic Interface API
Kernel/GenericInterface/Invoker/	invoker modules for Generic Interface
Kernel/GenericInterface/Mapping/	mapping modules for Generic Interface, e.g. Simple
Kernel/GenericInterface/Operation /	operation modules for Generic Interface
Kernel/GenericInterface/Transport /	transport modules for Generic Interface, e.g. "HTTP SOAP"
Kernel/Language	language translation files
Kernel/Scheduler/	Scheduler files
Kernel/Scheduler/TaskHandler	handler modules for scheduler tasks, e.g. GenericInterface
Kernel/System/	core modules, e.g. Log, Ticket
Kernel/Modules/	front end 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

1.2.2 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

1.2.3 Core Modules

Core modules are located under `$OTOBO_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 configuration options.
- `Kernel::System::Log` to log into OTOBO log back end.
- `Kernel::System::DB` to access the database back end.

- `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 the Documentation Portal.

1.2.4 Front End Handle

The interface between the browser, web server and the front end modules. A front end module can be used via the HTTP link.

```
http://localhost/otobo/index.pl?Action=Module
```

1.2.5 Front End Modules

Front end modules are located under `$OTOBO_HOME/Kernel/Modules/*.pm`. There are two public functions in there - `new()` and `run()` - which are accessed from the front end handle (e.g. `index.pl`).

`new()` is used to create a front end module object. The front end handle provides the used front end 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 see the Documentation Portal.

1.2.6 CMD Front End

The CMD (command line) front end is like the web front end handle and the web front end module in one (just without the `LayoutObject`) and uses the core modules for some actions in the system.

1.2.7 Generic Interface Modules

Generic interface modules are located under `$OTOBO_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 OTOBO as a client for the web service.
- `Kernel::GenericInterface::Provider` to use OTOBO 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 the Documentation Portal.

Generic Interface Invoker Modules

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

Generic interface invoker modules are used as a back end to create requests for remote systems to execute actions.

For more information see the Documentation Portal.

Generic Interface Mapping Modules

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

For more information see the Documentation Portal.

Generic Interface Operation Modules

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

Generic interface operation modules are used as a back end to perform actions requested by a remote system.

For more information see the Documentation Portal.

Generic Interface Transport Modules

Generic interface network transport modules are located under `$OTOBO_HOME/Kernel/GenericInterface/Transport/*`. Each transport module should be placed in a directory named as the network protocol used. For example: The HTTP SOAP transport module, located in `$OTOBO_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 the Documentation Portal.

1.2.8 Scheduler Task Handler Modules

Scheduler task handler modules are located under `$OTOBO_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 the Documentation Portal.

1.2.9 Database

The database interface supports different databases.

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

CHAPTER 2

OTOBO Internals - How it Works

2.1 Config Mechanism

OTOBO 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.

2.1.1 `Defaults.pm`: OTOBO Default Configuration

The default configuration file of OTOBO 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.

2.1.2 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.

`ZZZProcessManagement.pm` Perl cache of process management 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 OTOBO.

2.1.3 XML Configuration Files

In OTOBO, configuration options that the administrator can configure via system configuration are provided via XML files with a special format. To convert old XML's you can use the following command:

```
otobo> /opt/otobo/bin/otobo.Console.pl Dev::Tools::Migrate::ConfigXMLStructure
```

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:

```
otobo> /opt/otobo/bin/otobo.Console.pl Maint::Config::Rebuild
```

Each XML config file has the following layout:

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

    <!--  settings will be here -->

</otobo_config>
```

`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)
- `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>
```

`Required` If this is set to 1, the configuration 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).

`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`.

Structure of `Value` elements

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

`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 Value Types 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>
```

`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>
```

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, etc.). 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>
```

Value Types

The XML config settings support various types of configuration variables.

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 OTOBO translation files. This attribute can be placed on any tag (see also below).

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>
```

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>
```

Textarea

A config element for multiline text.

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

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>
```

Checkbox

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

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

Date

This config element contains a date value.

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

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>
```

Directory

This config element contains a directory.

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

File

This config element contains a file path.

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

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>
```

TimeZone

This config element contains a time zone value.

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

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>
```

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>
```

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>
```

Front End 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>
```

Default Item 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:

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>
```

Hash of Hash with `Date` Items

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

2.1.4 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',
);
```

2.2 Database Mechanism

OTOBO comes with a database layer that provides access to different databases.

`Kernel::System::DB` supports two ways of interacting with the database: SQL and XML.

2.2.1 SQL

The SQL interface should be used for normal database actions (`SELECT`, `INSERT`, `UPDATE`, etc.). It can be used more or less like the Perl DBI interface. One limitation is that only a single statement handle may be active per database object.

INSERT/UPDATE/DELETE statements

```
my $Name        = 'Arne Saknussemm';
my $YearOfBirth = 1662;
my $DBObject    = $Kernel::OM->Get('Kernel::System::DB');

# created
$DBObject->Do(
```

```
    SQL  => 'INSERT INTO alchemists (name, birth_year) VALUES ( ?, ? )',
    Bind => [ \$Name, \$YearOfBirth ],
);

# update
$Name =~ s/ß/S/;
$YearOfBirth++;
$DBObject->Do(
    SQL  => 'UPDATE alchemists SET name = ?, birth_year = ?',
    Bind => [ \$Name, \$YearOfBirth ],
);

# delete
$DBObject->Do(
   SQL=> 'DELETE FROM alchemists WHERE birth_year > 2022',
);
```

A special feature is the possibility to insert a timestamp that is generated on the client side. Using the client time avoids inconsistences with a divergent time on the database host.

```
# add entry
my $Name        = 'Professor Otto Lidenbrock';
my $DBObject    = $Kernel::OM->Get('Kernel::System::DB');
$DBObject->Do(
    SQL  => 'INSERT INTO geologist (name, created) VALUES (?, current_timestamp)',
    Bind => [ \$Name ],
);
```

SELECT statement

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

# get the last 15 steps for transaction 123
my $TransactionNumber = 123;
my $SQL = qq{SELECT step_id FROM transaction_steps WHERE tn = ? ORDER BY step_id DESC}
↪;
$DBObject->Prepare(
    SQL   => $SQL,
    Bind  => [ \$TransactionNumber ],
    Limit => 15,
);

my @StepIds;
while (my @Row = $DBObject->FetchrowArray()) {
    push @StepIds, $Row[0];
}

# get description for all valid steps
$DBObject->Prepare(
    SQL   => q{SELECT step_id, description FROM transaction_steps WHERE valid = 1},
);

my %StepIdToDescription;
while (my ($StepId, $Description) = $DBObject->FetchrowArray()) {
    $StepIdToDescription{$StepId} = $Description;
}
```

```
# get the number of valid steps
$DBObject->Prepare(
    SQL   => q{SELECT COUNT(*) FROM transaction_steps WHERE valid = 1},
);

# when there is number of results is known, then no loop is required
my ($NumValid) = $DBObject->FetchrowArray();

# loops can alse be avoided with SelectAll()
# SelectAll() returns a reference to an array of array references
my @ValidIds = map { $_->[0] } $DBObject->SelectAll(
    SQL => q{SELECT DESTINCT valid_id FROM transaction_steps ORDER BY valid_id ASC},
)->@*;
```

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

Note: Use the `Bind` attribute whereever you can, especially for long statements. If you use `Bind` you do not need the function `Quote()`.

Note: Beware that SelectAll() may not be used within a loop over the FetchrowArray() results.

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.

2.2.2 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.

INSERT

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

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];
}
```

DROP TABLE

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

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"/>
```

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 my $Statement (@SQL) {
    $Kernel::OM->Get('Kernel::System::DB')->Do(SQL => $Statement);
}
```

2.2.3 Database Drivers

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

2.2.4 Supported Databases

- MySQL or MariaDB
- PostgreSQL
- Oracle
- Microsoft SQL Server (only for external database connections, not as OTOBO database)

2.3 Log Mechanism

2.3.1 System Log

OTOBO comes with a system log back end that can be used for application logging and debugging.

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

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

Depending on the configured log level via `MinimumLogLevel` option in system configuration, 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 system configuration.

2.3.2 Communication Log

In addition to system log, OTOBO provides specialized logging back end for any communication related logging. The system comes with dedicated tables and front ends 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 https://
↪github.com/RotherOSS/otobo/issues/)!",
);

$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`.

2.4 Date and Time

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

2.4.1 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.

2.4.2 Creation of a `DateTime` Object

The object manager of OTOBO 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, timestamp, 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')`.

2.4.3 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 system configuration option `OTOBOTimeZone`. This setting defines the time zone that OTOBO uses internally to store date and time within the database.

Note: You have to ensure to convert a `DateTime` object to the OTOBO time zone before it gets stored in the database (there's a convenient method for this: `ToOTOBOTimeZone()`). 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.

2.4.4 Method Summary

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

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.

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).

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.

Time Zone Methods

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

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

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.

2.4.5 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 OTOBO 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();
```

2.5 Skins

The visual appearance of OTOBO 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 OTOBO (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, etc.).

2.5.1 Skin Basics

All skins are in `$OTOBO_HOME/var/httpd/htdocs/skins/Agent/$SKIN_NAME`.

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

Note: Skin support for customer interface was dropped with the new external interface. To create customized layout for external interface, use the Layout module of the admin interface.

2.5.2 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, the other one will be loaded only **after** the default skin. By loading here we mean that OTOBO 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="/otobo-web/skins/Agent/default/css-cache/CommonCSS_
↪179376764084443c181048401ae0e2ad.css" />
<link rel="stylesheet" href="/otobo-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="/otobo-web/skins/Agent/default/css/Core.Reset.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Default.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Header.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.OverviewControl.
↪css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.OverviewSmall.css
↪" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.OverviewMedium.
↪css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.OverviewLarge.css
↪" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Footer.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Grid.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Form.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Table.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Widget.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.WidgetMenu.css" /
↪>
```

```
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.TicketDetail.css
↪" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Tooltip.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Dialog.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Print.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Agent.
↪CustomerUser.GoogleMaps.css" />
<link rel="stylesheet" href="/otobo-web/skins/Agent/default/css/Core.Agent.
↪CustomerUser.OpenTicket.css" />
<link rel="stylesheet" href="/otobo-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 OTOBO 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 The CSS and JavaScript Loader section.

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;
}
```

2.5.3 Creating a New Skin

In this section, we will be creating a new agent skin which replaces the default OTOBO background color (white) with a custom company color (light grey) and the default logo by a custom 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 OTOBO 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 `$OTOBO_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 OTOBO 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 configuration file `$OTOBO_HOME/Kernel/Config/Files/XML/CustomSkin.xml`, which will contain the needed settings as follows:

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

To make this configuration active, we need to navigate to the system configuration module in the admin area of OTOBO. Alternatively, you can run the script:

```
$OTOBO_HOME/bin/otobo.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 system configuration setting `Loader::Agent::DefaultSelectedSkin` and set it to custom.

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

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

2.6 The CSS and JavaScript Loader

The CSS and JavaScript code in OTOBO 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 unnecessary whitespace and documentation) had to be addressed. To achieve these goals, the loader was invented.

2.6.1 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 OTOBO

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

2.6.2 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).

To learn about how the loader works, please turn it off in your OTOBO installation with the aforementioned configuration settings. Now look at the source code of the application module that you are currently using in this OTOBO 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="/otobo-web/js/js-cache/CommonJS_
↪d16010491cbd4faaaeb740136a8ecbfd.js"></script>

<script type="text/javascript" src="/otobo-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="/otobo-web/skins/Agent/default/css-
↪cache/CommonCSS_00753c78c9be7a634c70e914486bfbad.css" />

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

<!--[if IE 8]>
    <link rel="stylesheet" type="text/css" href="/otobo-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 OTOBO extensions by adding configuration data to the loader, telling it to load additional or alternative CSS or JavaScript content.

2.6.3 Configuring the Loader: JavaScript

To be able to operate correctly, the loader needs to know which content it has to load for a particular OTOBO 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.

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 OTOBO 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 OTOBO.

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?

Module Specific JavaScript

Not all JavaScript is usable for all application modules of OTOBO. 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 front end 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/OTOBOLineChart.js</Item>
                    <Item>thirdparty/nvd3-1.7.1/models/OTOBOMultiBarChart.js</Item>
                    <Item>thirdparty/nvd3-1.7.1/models/OTOBOStackedAreaChart.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 front end 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.

2.6.4 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.

Common CSS

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

2.7 Templating Mechanism

Internally, OTOBO 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 front end 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 `$OTOBO_HOME/Kernel/Output/HTML/Standard/*.tt`.

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

2.7.1 Template Commands

Inserting Dynamic Data

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

[% 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 OTOBO.
>
> 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 OTOBO.

```
<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') %]`.

[% 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 %].
```

[% 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>
```

[% 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) %];
```

[% 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, e. g. `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 OTOBO.
>
> Don't forget to add the `| html` filter where appropriate.

[% Config() %]

Inserts configuration 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} = 'otobo.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.

Localization Commands

`[% 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 %];
```

`[% 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
```

`[% 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>') %]
```

Template Processing Commands

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>
```

[% 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 OTOBO 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.

[% 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 OTOBO 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` 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 more information see the Documentation Portal.

[% 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 %]`? 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.

2.7.2 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 front end 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.

2.8 Creating Your Own Themes

You can create your own themes so as to use the layout you like in the OTOBO web front end. 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 Mechanism.

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.

   Note: Only copy over the files you're planning to change. OTOBO 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 system configuration 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 OTOBO, since these changes will be lost after an update. Create your own themes only by performing the steps described above.

2.9 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
- the usage of translated content within the code

2.9.1 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)
```

This 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 system configuration 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>
```

In YAML text, you can mark strings for translation by appending `# Translatable` to the end of the target string.

```
--
Title:
   - Ticket Number # Translatable
Name:
   - Alice
```

2.9.2 Collecting Translatable Strings Into The Translation Database

The console command `otobo.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 OTOBO framework and all extension modules, `.pot` and `.po` files are written. These files are used by translators to localize the software.

But OTOBO requires the translations to be in Perl files for speed reasons. These files will also be generated by `otobo.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`

Default Framework Translation File

The default framework translation file includes the basic translations. The following is an example of a default framework translation file.

```
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;
```

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.

```
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;
```

Note: The language files for the new interface are now part of the built application (static JSON). When you add a custom language file to the file system, you need to rebuild the application for the change to be considered. To trigger the rebuild, restart the server with the `--deploy-assets` option:

```
otobo> /opt/otobo/bin/otobo.WebServer.pl --deploy-assets
```

During the build process, the language files will be refreshed and will take any `*_Custom.pm` into account.

2.9.3 The Translation Process Itself

OTOBO uses Weblate to manage the translation process. Please see Translating section for details.

2.9.4 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 Templating Mechanism.

CHAPTER 3

How to Extend OTOBO

3.1 Writing A New OTOBO Front End Module

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

3.1.1 What we want to write

We want to write a little OTOBO 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 OTOBO 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
```

3.1.2 Default Config File

The creation of a module registration facilitates the display of the new module in OTOBO. 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" ?>
<otobo_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>
</otobo_config>
```

3.1.3 Front End Module

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

```
# --
# 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;
```

3.1.4 Core Module

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

```
# --
# 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 OTOBO 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;
```

3.1.5 Template File

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

```
# --
# 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.

3.1.6 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;
```

3.1.7 Summary

The example given above shows that it is not too difficult to write a new module for OTOBO. 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).

3.2 Writing A New OTOBO Front End Component

In this example, we will try to write a new OTOBO front end component. Starting with OTOBO 10, the framework supports single page application front ends written in Vue.js and based on a new JavaScript toolchain. First iteration contains the new external interface, for which we will try to write a custom component. You will need to have a running OTOBO Development Environment as specified in the chapter of the same name.

3.2.1 The Goal

We want to write a small front end component that displays the text Hello World when called up. This will be a route component, meaning it will be available in the external interface when called with a carefully crafted URL.

3.2.2 Using The Skeleton Command

To speed up the development, we should use a skeleton command to get a boilerplate template file which we can build upon.

On a running OTOBO instance, call the following command to generate the template. We will use `HelloWorld` as the name of our new component:

```
bin/otobo.Console.pl Dev::Code::Generate::VueComponent --component-directory /ws/
↪MyPackage --component-subdirectory Apps/External/Components/Route --no-docs␣
↪HelloWorld
```

In the command `--component-directory` is the directory of your module, `--component-subdirectory` path under `Frontend/` folder that will house the component file. For now, use `--no-docs` switch to skip creation of the documentation component for the design system.

This command will generate two files with following paths:

```
Generated: /ws/MyPackage/Frontend/Apps/External/Components/Route/HelloWorld.vue
Generated: /ws/MyPackage/Frontend/Tests/Apps/External/Components/Route/HelloWorld.js
Skipped creating documentation component.
```

3.2.3 The Route Configuration

In order to allow the route in the external interface application, we need to add a correct route configuration that points to our component. Therefore we create a file `Kernel/Config/Files/XML/HelloWorld.xml` with following definition:

```
<?xml version="1.0" encoding="utf-8" ?>
<otobo_config version="2.0" init="Application">
    <Setting Name="ExternalFrontend::Route###420-HelloWorld" Required="0" Valid="1">
        <Description Translatable="1">Defines the application routes for the external␣
↪interface. Additional routes are defined by adding new items and specifying their␣
↪parameters. 'Group' and 'GroupRo' arrays can be used to limit access of the route␣
↪to members of certain groups with RW and RO permissions respectively. 'Path' defines␣
↪the relative path of the route, and 'Alias' can be used for specifying an␣
↪alternative path. 'Component' is the path of the Vue component responsible for␣
↪displaying the route content, relative to the Components/Route folder in the app.
↪'IsPublic' defines if the route will be accessible for unauthenticated users and in␣
↪case this is set to '1', 'Group' and 'GroupRo' parameters will be ignored. 'Props'␣
↪can be used to signal that the path contain dynamic segments, and that their values␣
↪should be bound to the component as props (use '1' to turn on this feature).</
↪Description>
        <Navigation>Frontend::External::Route</Navigation>
        <Value>
            <Array>
                <DefaultItem ValueType="ApplicationRoute">
                    <Hash>
```

```
                    </Hash>
                </DefaultItem>
                <Item>
                    <Hash>
                        <Item Key="Group">
                            <Array>
                            </Array>
                        </Item>
                        <Item Key="GroupRo">
                            <Array>
                            </Array>
                        </Item>
                        <Item Key="Path">/hello-world/:headingText?</Item>
                        <Item Key="Alias"></Item>
                        <Item Key="Component">HelloWorld</Item>
                        <Item Key="IsPublic">1</Item>
                        <Item Key="Props">1</Item>
                    </Hash>
                </Item>
            </Array>
        </Value>
    </Setting>
</otobo_config>
```

- `Group` and `GroupRo` can be used to limit the route screen to users with certain groups. Please note that this only concerns the authenticated customer users.
- `Path` is actually the slug under which the route component will be available. The full URL in this case will be `/external/hello-world`, and any subsequent path component will be passed as a parameter named `headingText`. If your system has `Frontend::PrefixPath` configured, full URL will be prepended by it.
- `Alias` can be used to provide an alias for the same route (e. g. `/hello-world-alt`). It will point to the same component.
- `Component` is the component identifier, first part of the filename, without the `.vue` extension. In case of component folders, it's the name of the root folder. See Component Folders for more information.
- `IsPublic` defines if the route will be accessible by unauthenticated users (0/Empty - not accessible, 1 - accessible).
- `Props` defines if the route will be passed URI parameters as prop values (0/Empty - not passed, 1 - passed). See Passing Parameters to the Route Component for more information.

3.2.4 Component Template Code

Let's fire up the code editor now and take a closer look at the `HelloWorld.vue` file that our skeleton command created.

Top part of the file contains a template section which should contain Vue.js template code. For example, let's modify it so it displays a heading with a text variable:

```
<template>
    <main class="HelloWorld">
        <b-container>
            <b-row>
                <b-col>
```

```
                    <h1 class="HelloWorld__Heading">
                        {{ headingText | translate }}
                    </h1>
                </b-col>
            </b-row>
        </b-container>
    </main>
</template>
```

OTOBO supports number of filters, with `translate` being one of them. It even supports translation of string literals with placeholder values, you can use it like this:

```
{{ 'This is a %s.' | translate('string') }}
```

3.2.5 Component Core Code

Next, we add a support for a prop to our component core code block, following is a modified and abridged version suitable for an example:

```
<script>
export default {
    name: 'HelloWorld',

    props: {
        headingText: {
            type: String,
            default: translatable('Hello, world!'),
        },
    },
};
</script>
```

This adds a prop with the name `headingText` to our component, which is of type string and has a sensible default value.

Usage of `translatable()` no-op method is limited to marking translatable strings which appear in the code. Please note that this is not required for string literals which are piped to the translate filter, as this will be assumed from the start. Rule of thumb is to use the marker anywhere where the string is not translated at the place where it is defined.

3.2.6 Component Style Code

Last, but not the least, we have an option to specify styles used by the component. For this we have access to the SCSS, which is a flavor of SASS CSS extension set. To leverage it, just add a style tag at the end of the component file:

```
<style lang="scss">
.HelloWorld {
    &__Heading {
        color: $primary;
    }
}
</style>
```

Inside the style block, you will have access to certain set of global variables and mixins. Please refer to the framework code for details (take a look at the `Frontend/Styles/globals.js`).

Please note that while the styles will be loaded only when your component is referenced, these will be globally available afterwards since the CSS is inherently global for the same page. There is an option to scope the styles just to your component, you can do this via the `scoped` attribute on the style tag, but this might not be necessary with clever usage of BEM approach in designing your class names.

3.2.7 Passing Parameters to the Route Component

In the route configuration above, we defined the route path that contains a parameter placeholder (`headingText`). By activating the `Props` flag, we made sure that the value of this parameter will be bound to our component prop with the same name every time a route is entered.

For example, if we enter the route via the `/external/hello-world` URL, our component prop will be undefined and therefore will get its default value.

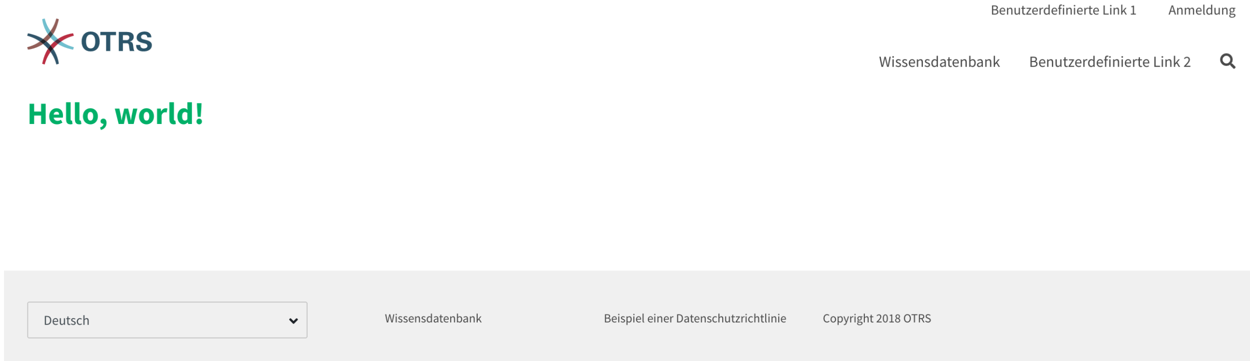

Fig. 3.1: Passing Parameters - Default Prop Value

But, if we access the route via the `/external/hello-world/Value`, the prop will be set to string `Value`, and even automatically translated in the current user language (where applicable).

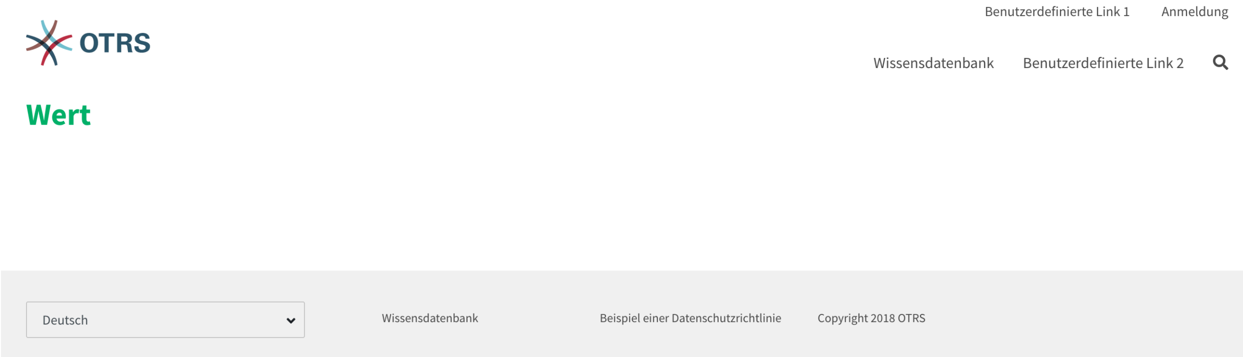

Fig. 3.2: Passing Parameters - Translated Prop Value

3.2.8 Component Folders

In case of self-enclosed components, you might want to ship some additional files with it. Sometimes it's better to modularize the code base since it's easier to maintain. In case of front end components

you have a really simple way of doing this: component folders. Instead of a single `.vue` file for a component, enclose the file named `index.vue` in a folder named as your component. Something like this:

```
HelloWorld/
HelloWorld/index.vue
```

Then, simply add new files in the same folder, following a sane structure:

```
HelloWorld/
HelloWorld/index.vue
HelloWorld/Styles/_mystyles.scss
HelloWorld/Images/foobar.png
HelloWorld/Fonts/awesome-font.woff
HelloWorld/Fonts/awesome-font.woff2
HelloWorld/ChildComponent1.vue
HelloWorld/ChildComponent2/index.vue
HelloWorld/ChildComponent2/Styles/_childstyles2.scss
```

You get the idea. It will then be possible to reference the new files via relative paths, in order to achieve something like this in the parent component (`index.vue`):

```
<template>
    <img src="./Images/foobar.png" alt="Foobar" />
</template>
```

Or, something like this:

```
<script>
export default {
    name: 'HelloWorld',

    components: {
        ChildComponent1: () => import('./ChildComponent1'),
        ChildComponent2: () => import('./ChildComponent2'),
    },
...
```

Even external styles can be referenced in the correct block:

```
<style lang="scss">
@import './Styles/mystyles';
</style>
```

With this approach you will be left with a packaged component in a single folder that follows the logical tree hierarchy, and makes all resources easily findable when needed.

3.2.9 Packaging Additional Vendor Modules

In certain cases, you might need to ship additional Node.js modules with your package. Unfortunately, both NPM and OTOBO do not support easy addition of modules to the root `node_modules/` folder, but there is a mechanism to provide pre-packaged module files.

Simply create a `Frontend/Vendor` folder in your package, and add your module resources in sub-folders within it.

For example, let us assume we want to ship a useful `vue-full-calendar` component and its dependencies as part of our package. This component has following NPM dependencies:

```
$ npm view vue-full-calendar dependencies
{ 'babel-plugin-transform-runtime': '^6.23.0', fullcalendar: '^3.4.0', 'lodash.
↪defaultsdeep': '^4.6.0' }
```

However, some of its dependencies have even more dependencies and we can inspect them too:

```
$ npm view babel-plugin-transform-runtime dependencies
{ 'babel-runtime': '^6.22.0' }

$ npm view fullcalendar dependencies
{ jquery: '2 - 3', moment: '^2.20.1' }

$ npm view lodash.defaultsdeep dependencies
```

Quick check will inform us that both `babel-runtime` and `moment` are actually part of the OTOBO framework dependencies:

```
/opt/otobo $ npm list babel-runtime
otobo-frontend@7.0.0-dev /ws/otobo7-mojo
  bootstrap-vue@2.0.0-rc.11
    opencollective@1.0.3
      babel-polyfill@6.23.0
        babel-runtime@6.26.0  deduped
  esdoc2@2.1.5
    babel-generator@6.26.0
      babel-messages@6.23.0
        babel-runtime@6.26.0  deduped
...

/opt/otobo $ npm list moment
otobo-frontend@7.0.0-dev /ws/otobo7-mojo
  moment-timezone@0.5.21
    moment@2.22.2
```

This means that we don't have to ship those modules too, since they will be available out-of-box. While it's cumbersome to check all dependencies, it will be worthwhile because our package will be smaller. We will also prevent issues with overriding framework dependencies, since `Frontend/Vendor` wins always.

Let's now install what we need and discard what we don't need. The easiest way to do it is via the following NPM command:

```
/ws/MyPackage $ npm install vue-full-calendar --no-save
+ vue-full-calendar@2.7.0
added 9 packages from 14 contributors in 1.883s

/ws/MyPackage $ ls -1 node_modules/
babel-plugin-transform-runtime
babel-runtime
core-js
fullcalendar
jquery
lodash.defaultsdeep
moment
regenerator-runtime
vue-full-calendar
```

Now we remove those modules which we know are provided by the framework:

```
/ws/MyPackage $ rm -rf node_modules/babel-runtime node_modules/core-js node_modules/
↪moment node_modules/regenerator-runtime

/ws/MyPackage $ ls -1 node_modules/
babel-plugin-transform-runtime
fullcalendar
jquery
lodash.defaultsdeep
vue-full-calendar
```

Much better. Now we move the modules to their correct place:

```
/ws/MyPackage $ mkdir -p Frontend/Vendor
/ws/MyPackage $ mv node_modules/* Frontend/Vendor/
/ws/MyPackage $ rmdir node_modules/
```

Final optimization would be to remove unneeded files from the specific module folders. This might prove to be complicated, but it's worth it since it will further reduce size of the modules and number of files that need to be included in the package.

For example, let's remove minimized JS files from the `fullcalendar` module because we identified that the Vue component uses full dist files only:

```
/ws/MyPackage $ rm Frontend/Vendor/fullcalendar/dist/*.min.*
```

It's also safe to remove jQuery source and minimized files as well, since the `fullcalendar` uses original dist files too:

```
/ws/MyPackage $ rm Frontend/Vendor/jquery/dist/*.min.*
/ws/MyPackage $ rm Frontend/Vendor/jquery/external/sizzle/dist/*.min.*
/ws/MyPackage $ rm -rf Frontend/Vendor/jquery/src
```

We are left with approx. 100+ files which we need to include in our SOPM files, like any other regular package file. Once we do this, these dependencies will be present and resolvable in the target system:

```
/ws/MyPackage $ ls -la Frontend/Vendor
Frontend/Vendor
Frontend/Vendor/vue-full-calendar
Frontend/Vendor/vue-full-calendar/.babelrc
Frontend/Vendor/vue-full-calendar/LICENSE
Frontend/Vendor/vue-full-calendar/tests
Frontend/Vendor/vue-full-calendar/tests/fullcalendar.spec.js
Frontend/Vendor/vue-full-calendar/index.js
...
```

3.3 Using the power of the OTOBO module layers

OTOBO 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!

3.3.1 Agent Authentication Module

There are several agent authentication modules (DB, LDAP and HTTPBasicAuth) which come with the OTOBO 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.

Agent Authentication Module 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.

```
# --
# Copyright (C) 2001-2020 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Agent Authentication Module 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 system configuration.

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

Agent Authentication Module Use Case Example

A useful example of an authentication implementation could be a SOAP back end.

3.3.2 Authentication Synchronization Module

There is an LDAP authentication synchronization module which come with the OTOBO 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.

Authentication Synchronization Module 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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
}
```

Authentication Synchronization Module 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 system configuration.

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

Authentication Synchronization Module Use Case Example

Useful synchronization implementation could be a SOAP or RADIUS back end.

3.3.3 Customer Authentication Module

There are several customer authentication modules (DB, LDAP and HTTPBasicAuth) which come with the OTOBO 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.

Customer Authentication Module 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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Customer Authentication Module 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 Translatable="1">Module to authenticate customers.</Description>
    <Group>Framework</Group>
    <SubGroup>Frontend::CustomerAuthAuth</SubGroup>
    <Setting>
        <Option Location="Kernel/System/CustomerAuth/*.pm" SelectedID=
↪"Kernel::System::CustomerAuth::CustomAuth"></Option>
    </Setting>
</ConfigItem>
```

Customer Authentication Module Use Case Example

Useful authentication implementation could be a SOAP back end.

3.3.4 Customer User Preferences Module

There is a DB customer-user preferences module which come with the OTOBO 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()`.

Customer User Preferences Module 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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Customer User Preferences Module 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 Translatable="1">Parameters for the customer preference table.</
↪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>
```

Customer User Preferences Module Use Case Example

Useful preferences implementation could be a SOAP or LDAP back end.

3.3.5 Queue Preferences Module

There is a DB queue preferences module which come with the OTOBO 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.

Queue Preferences 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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Queue Preferences 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 Translatable="1">Default queue preferences module.</Description>
    <Group>Ticket</Group>
```

```
    <SubGroup>Frontend::Queue::Preferences</SubGroup>
    <Setting>
        <String Regex="">Kernel::System::Queue::PreferencesCustom</String>
    </Setting>
</ConfigItem>
```

Queue Preferences Use Case Example

Useful preferences implementation could be a SOAP or RADIUS back end.

3.3.6 Service Preferences Module

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

Service Preferences 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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Service Preferences 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 Translatable="1">Default service preferences module.</Description>
    <Group>Ticket</Group>
    <SubGroup>Frontend::Service::Preferences</SubGroup>
    <Setting>
        <String Regex="">Kernel::System::Service::PreferencesCustom</String>
    </Setting>
</ConfigItem>
```

Service Preferences Use Case Example

Useful preferences implementation could be a SOAP or RADIUS back end.

3.3.7 SLA Preferences Module

There is a DB SLA preferences module which come with the OTOBO 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.

SLA Preferences 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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

SLA Preferences 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>
```

SLA Preferences Use Case Example

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

3.3.8 Log Module

There is a global log interface for OTOBO that provides the possibility to create own log back ends.

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

Log Module Code Example

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Log Module 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>
```

Log Module Use Case Example

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

Note: `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.

3.3.9 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 front end modules then the adaption only has to be implemented once. The second advantage is that when OTOBO 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 front end module.

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

`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:

```
<Setting Name="Frontend::Template::GenerateBlockHooks###100-OTOBOBusiness-
↪ContactWithData" Required="1" Valid="1">
    <Description Translatable="1">Generate HTML comment hooks for the specified blocks␣
↪so that filters can use them.</Description>
    <Navigation>Frontend::Base::OutputFilter</Navigation>
    <Value>
        <Hash>
            <Item Key="AgentTicketZoom">
                <Array>
                    <Item>CustomerTable</Item>
                </Array>
            </Item>
        </Hash>
    </Value>
</Setting>
```

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.

`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.

`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 OTOBO is configured to not use the Rich Text feature in the front end.

Output Filter Code Example

See package TemplateModule.

Output Filter Configuration Example

See package TemplateModule.

Output Filter Use Case Example

Show additional ticket attributes in `AgentTicketZoom` This can be achieved with a `FilterElementPost` output filter.

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.

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.

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.

Note: 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.

Best Practices

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

3.3.10 Stats Module

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

Dynamic Stats

In contrast to static stats modules, dynamic statistics can be configured via the OTOBO 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.

Stats Code Example

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO 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. `<OTOBO_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 OTOBO 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"?>

<otobo_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>
</otobo_stats>
```

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Stats Configuration Example

```
<?xml version="1.0" encoding="utf-8" ?>
<otobo_config version="1.0" init="Config">
    <ConfigItem Name="Stats::DynamicObjectRegistration###DynamicStatsTemplate"␣
↪Required="0" Valid="1">
        <Description Translatable="1">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>
</otobo_config>
```

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

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`

Static Stats 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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Static Stats Configuration Example

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

3.3.11 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 system configuration 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.

Ticket Number Generator Code Example

See files in `Kernel/System/Ticket/Number` folder of the source code.

Ticket Number Generator Configuration Example

See settings in `Kernel/Config/Files/XML/Ticket.xml` started with the name `Ticket::NumberGenerator`.

Ticket Number Generator Use Case Example

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.

Note: 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.

3.3.12 Ticket Event Module

Ticket event modules are running right after a ticket or an article action took 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`. The parameter `Data` is a referecne to a hash. It contains data of the ticket and, in the case of article related events, also data of the article.

Ticket Event Module Code Example

See the files in `Kernel/System/Ticket/Event` folder of the source code.

Ticket Event Module Configuration Example

See the settings in `Kernel/Config/Files/XML/Ticket.xml` that are starting with the name `Ticket::EventModulePost###`.

Ticket Event Module Use Case Example

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 system configuration entry `Ticket::EventModulePost###910-ForceUnlockOnMove`.

Perform extra cleanup action when a ticket is deleted A customized OTOBO 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.

New tickets should be twittered A ticket event module listening to `TicketCreate` can send out tweets.

Ticket and Article Events

Available ticket events:

- `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`

Available article events:

- `ArticleCreate`
- `ArticleUpdate`

- `ArticleSend`
- `ArticleBounce`
- `ArticleAgentNotification`
- `ArticleCustomerNotification`
- `ArticleAutoResponse`
- `ArticleFlagSet`
- `ArticleFlagDelete`
- `ArticleAgentNotification`
- `ArticleCustomerNotification`

3.3.13 Dashboard Module

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

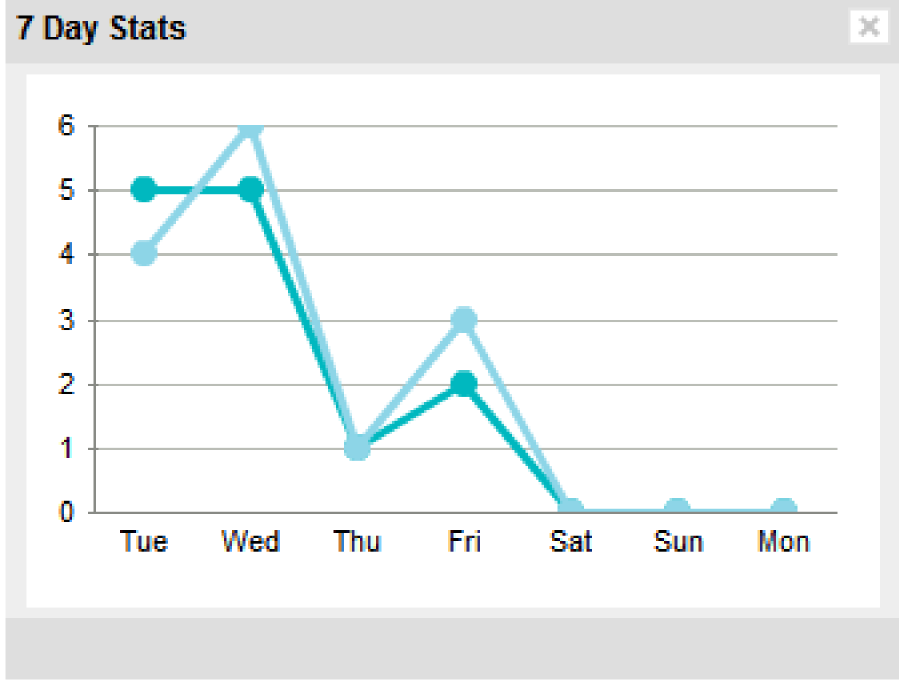

Fig. 3.3: Dashboard Widget

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 Translatable="1">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>
    <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>
```

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

3.3.14 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 OTOBO framework.

- `AgentOnline`
- `AgentTicketEscalation`
- `CharsetCheck`
- `CustomerOnline`
- `UIDCheck`

Notification Module 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()`.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Notification Module 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 Translatable="1">Module to show custom message in the agent␣
↪interface.</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>
```

Notification Module Use Case Example

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

3.3.15 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 OTOBO framework. For more information please have a look at the OTOBO admin manual.

Ticket Menu Module 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()`.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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;
```

Ticket Menu Module 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 Translatable="1">Module to show custom link in menu.</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>
```

Ticket Menu Module Use Case Example

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

Note: The ticket menu directs to an URL that can be handled. If you want to handle that request via the OTOBO framework, you have to write your own front end module.

3.3.16 Root Application Module Layer

The root application module layer enables the developers to develop and load custom front end components. Components will be lazy loaded and are omnipresent in the front end application. They can use any API that is exposed to other components within the application, including its state.

Developing a custom root application component is as easy as creating a new file.

Root Application Component Code Example

In this small example we'll write a little component, which outputs a custom footer on every page of the agent interface app.

Just create a new file `SampleFooter.vue` in `/Frontend/Apps/Agent/Components/RootApplicationModule/Modules` folder.

```
<template>
    <div class="CustomFooter">This is my custom footer.</div>
</template>

<script>
export default {
    name: 'SampleFooter',
};
</script>

<style lang="scss">
.CustomFooter {
    width: 100%;
    height: 30px;
    border-top: 1px solid #000;
    padding: 2px;
    text-align: center;
}
</style>
```

To activate our custom footer module, just rebuild the app that was modified. If a development server is used, it will need to be restarted to pick up the changes.

3.3.17 Network Transport

The network transport is used as method to send and receive information between OTOBO 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.

OTOBO as provider OTOBO uses the network transport modules to get the data from the remote system and the operation to be executed. After the operation is performed OTOBO uses them again to send the response back to the remote system.

OTOBO as requester OTOBO uses the network transport modules to send petitions to the remote system to perform a remote action along with the required data. OTOBO 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.

Transport Back End

Next we will show how to develop a new transport back end. Each transport back end 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 back ends 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.

Network Transport 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 back ends. For example the HTTP SOAP transport modules uses `SOAP::Lite` module as back end.

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.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO 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 OTOBO) 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 OTOBO 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.

Network Transport Configuration Example

There is the need to register this network transport module to be accessible in the OTOBO 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>
```

3.3.18 Mapping

The mapping is used to convert data from OTOBO 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:

Mapping from	Mapping 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.

OTOBO 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 OTOBO format.
3. OTOBO performs the operation and return the response in OTOBO format.
4. The data is mapped from the OTOBO format to the remote system format.
5. The response with the data in the remote system format is sent to the remote system.

OTOBO as requester example

1. OTOBO prepares the request to the remote system using the data in the OTOBO format.
2. The data is mapped from the OTOBO 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 OTOBO with the data in remote system format.

4. The data is mapped form remote system format (again) to the OTOBO format.
5. OTOBO processes the response.

Mapping Back End

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 back end. Each mapping back end 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 back ends are handled by the mapping module (`Kernel/GenericInterface/Mapping.pm`).

Mapping Code Example

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO 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 OTOBO 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.

Mapping Configuration Example

There is the need to register this mapping module to be accessible in the OTOBO 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>
```

3.3.19 Invoker

The invoker is used to create a request from OTOBO to a remote system. This part of the GI is in charge of perform necessary tasks in OTOBO side, to gather the necessary data in order to construct the request.

Invoker Back End

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`).

Invoker Code Example

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO 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 OTOBO 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.

Invoker Configuration Example

There is the need to register this invoker module to be accessible in the OTOBO 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>
```

3.3.20 Operation

The operation is used to perform an action within OTOBO. 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, OTOBO sends a defined confirmation to the external system.

Operation Back End

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`).

Operation Code Example

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO 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.

Operation Configuration Example

There is the need to register this operation module to be accessible in the OTOBO 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>
```

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.

See also:

To learn more about unit tests, please take a look to the Unit Tests chapter.

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

```
# --
# Copyright (C) 2001-2020 OTOBO AG, https://otobo.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://otobo.org/SoapTestInterface/',
                Endpoint  => $RemoteSystem,
            },
        },
        Operation => {
```

```
            Test => {
                Type => 'Test::Test',
            },
        },
    },
    Requester => {
        Transport => {
            Type    => 'HTTP::SOAP',
            Config => {
                NameSpace => 'http://otobo.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;
```

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.otobo.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.otobo.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>
<!-- ... -->
```

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/otobo/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>
```

Web Service SOAP Extension Example

Web services can be imported into OTOBO 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
```

Web Service REST Extension Example

Web services can be imported into OTOBO 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
```

3.3.21 OTOBO Daemon

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

OTOBO Daemon Modules

The OTOBO daemon `bin/otobo.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 OTOBO 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 OTOBO 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 OTOBO scheduler daemon can deal with the majority of them, either if it is an OTOBO function that needs to be executed on a regular basis (CRON like) or if it's triggered by an OTOBO event, the OTOBO scheduler should be capable to deal with it out of the box or by adding a new scheduler task worker module.

Creating A New Daemon Module

All daemon modules requires to be registered in the system configuration in order to be called by the main OTOBO daemon.

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>
```

Daemon Module Code Example

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 common OTOBO 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 OTOBO 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 OTOBO 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.

3.3.22 OTOBO Scheduler

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

OTOBO Scheduler Task Managers

SchedulerCronTaskManager This reads registered cron tasks from the OTOBO system configuration and determines the correct time to create a task to be executed.

SchedulerFutureTaskManager This 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 This continuously reads the generic agent 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`.

OTOBO Scheduler Task Workers

SchedulerTaskWorker This executes 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 back end 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.

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.

Scheduler Task Worker Code Example

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 common OTOBO 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 OTOBO 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.

3.3.23 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).

Tickets or articles could have as many fields as needed. It is also possible to use the dynamic fields framework for other objects rather than just ticket or article.

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 OTOBO standard package to extend the available types of the dynamic fields or even to extend current dynamic field with more functions.

3.3.24 Dynamic Fields Framework

Before creating new dynamic fields is necessary to understand its framework and how OTOBO screens interact with them, as well as their underlying API.

In OTOBO 10 two dynamic fields API are present.

The new API is used in the `External` and `Agent` interface while the legacy is still heavily used in the `Admin` interface and core modules `$OTOBO_HOME/Kernel/System`.

Dynamic Fields Framework (Legacy API)

This is the old and well known dynamic fields API.

The following picture shows the architecture of the framework.

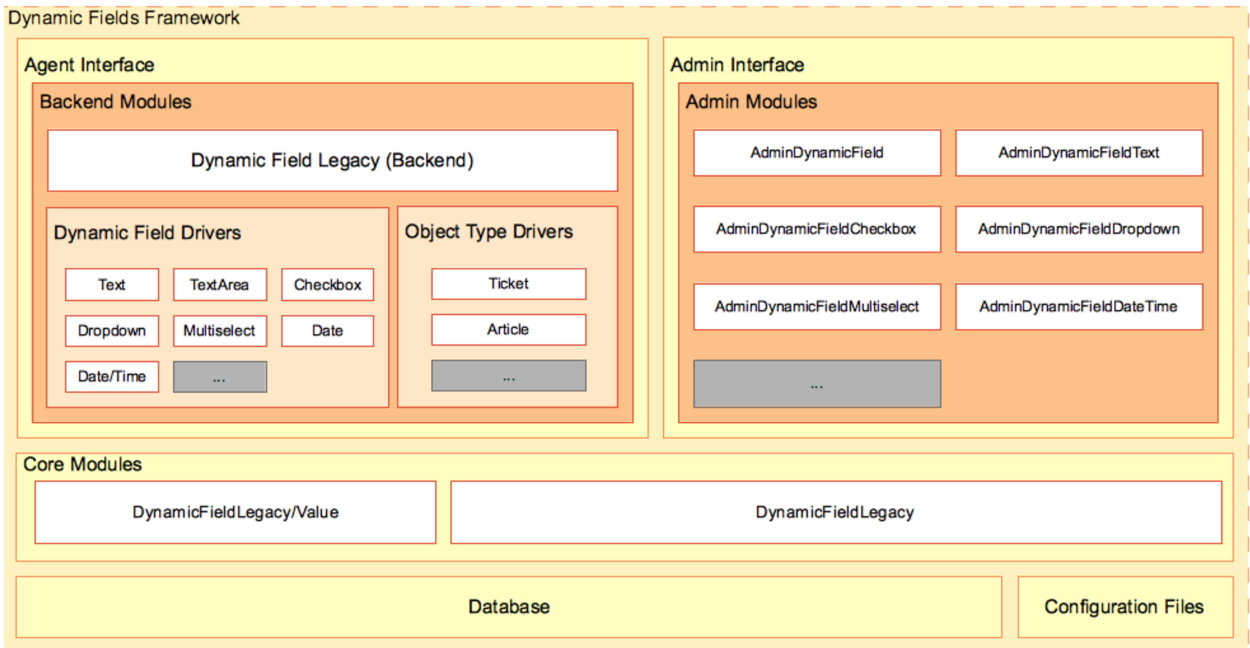

Fig. 3.4: Dynamic Fields Framework

Dynamic Field Back End Modules (Legacy API)

Dynamic Field Back End (Legacy API)

Normally called as `BackendObject` in the front end modules is the mediator between the front end 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 back end 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 `$OTOBO_HOME/Kernel/System/DynamicFieldLegacy/Backend.pm`.

Dynamic Field Drivers (Legacy API)

A dynamic field driver is the implementation of the dynamic field. Each driver must implement all the mandatory functions specified in the back end (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 `$OTOBO_HOME/Kernel/System/DynamicFieldLegacy/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`

Object Type Delegate (Legacy API)

An object type delegate is responsible to perform specific functions on the object linked to the dynamic field. These functions are triggered by the back end object as they are needed.

These modules are located in `$OTOBO_HOME/Kernel/System/DynamicFieldLegacy/ObjectType/*.pm`.

Dynamic Fields Admin Modules (Legacy API)

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 OTOBO 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 `$OTOBO_HOME/Kernel/Modules/*.pm`.

Note: Each admin dialog needs its corresponding HTML template file (`.tt`).

Dynamic Fields Core Modules (Legacy API)

This modules reads and writes the dynamic fields information from and to the database tables.

`DynamicFieldLegacy.pm` 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 `$OTOBO_HOME/Kernel/System/DynamicFieldLegacy.pm`.

`DynamicFieldLegacy/Value.pm` 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 `$OTOBO_HOME/Kernel/System/DynamicFieldLegacy/Value.pm`.

Dynamic Fields Database Tables (Legacy API)

There are two tables in the database to store the dynamic field information:

`dynamic_field` Used by the core module `DynamicFieldLegacy.pm`, it stores the dynamic field definitions.

`dynamic_field_value` Used by the core module `DynamicFieldLegacy/Value.pm` to save the dynamic field values for each dynamic field and each object type instance.

Dynamic Fields Configuration Files (Legacy API)

The back end 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.

Front end 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.

Dynamic Fields Framework (New API)

This is the new dynamic field API, which is slightly different from the legacy API in terms of the field structure.

The following picture shows the architecture of the framework.

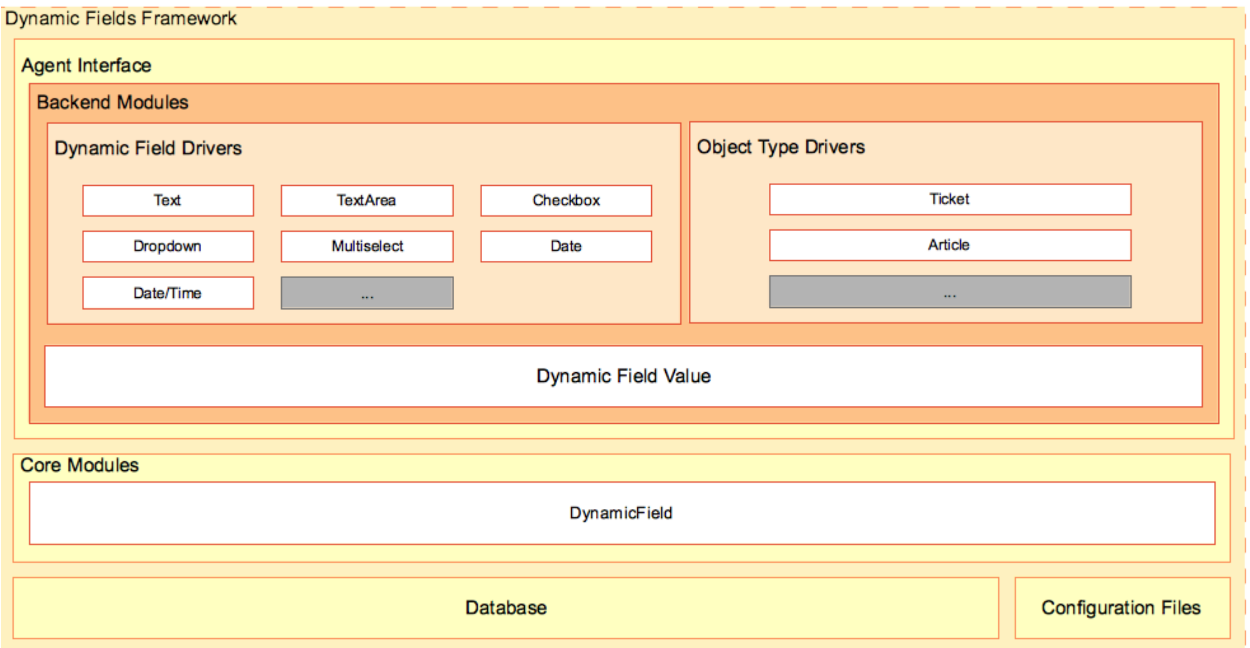

Fig. 3.5: Dynamic Fields Framework

Dynamic Field Back End Modules (New API)

Dynamic Field Drivers (New API)

Every dynamic field driver represents a dynamic field, containing data like the related name, configurations and stored values. Each driver should inherit from the base driver `Driver/Base.pm` and implement all the mandatory functions specified in it.

A driver provides necessary information about, how it must be rendered in the front end applications (displaying in forms or screens), or how to interacts with modules and APIs, like the statistics back end among other functions.

These modules are located in `$OTOBO_HOME/Kernel/System/DynamicField/Driver/*.pm`.

To provide often needed and common functions for certain driver types, there are global base drivers available (`Base/Text.pm`, `Base/Select.pm` and `Base/DateTime.pm`) from which dynamic field driver modules may inherit. Those base drivers themselves inherits from the mandatory base module.

For example driver `TextArea.pm` uses `Base/Text.pm` that also uses `Base.pm` then `TextArea` only needs to implement the functions that are missing in `Base.pm` and `Base/Text.pm` or the ones that are special cases.

The following is the drivers inheritance tree:

```
Base.pm
|
|-- Base/Text.pm
    |-- Text.pm
    |-- TextArea.pm
    |
    Base/Select.pm
```

```
    |-- Dropdown.pm
    |-- Multiselect.pm
    |
    Base/DateTime.pm
    |-- DateTime.pm
    |-- Date.pm
    |
    Checkbox.pm
    |
    ContactWithData.pm
```

Roles (New API)

The dynamic field roles provide a simple way to extend the driver functionality.

The `HasValueType` roles indicate, which kind of data the driver will store and in which database column has to be used.

The `Behaviour` roles extend the drivers with extra funcionality, for example `SupportsACLs.pm` indicates that the possible values of the drivers could be restricted by ACL permissions.

These modules are located in `$OTOBO_HOME/Kernel/System/DynamicField/Driver/Role/*.pm`.

Here is a list of the current roles:

```
HasValueType
- Text.pm
- Int.pm
- DateTime.pm

Behavior
- IsFilterable.pm
- IsSortable.pm
- IsHTMLContent.pm
- SupportsACLs.pm
- SupportsExternalInterface.pm
- SupportsLikeOperator.pm
- SupportsMultipleRecordsPerValue.pm
- SupportsNotificationEvents.pm
- SupportsOutputInTicketInformation.pm
- SupportsStatsSearchField.pm
```

Object Type Delegation (New API)

An object type delegation is responsible to perform specific functions on the object linked to a certain dynamic field. These functions are triggered by the driver as they are needed.

These modules are located in `$OTOBO_HOME/Kernel/System/DynamicField/ObjectType/*.pm`.

Dynamic Fields Admin Modules (New API)

The management of the dynamic fields is done in the administrator interface, which still uses the legacy API.

Dynamic Fields Core Modules (New API)

These modules read and write the dynamic fields information from and to the database tables.

DynamicField.pm This module is responsible to manage the dynamic field definitions. It provides the basic API for add, change, delete, list and get dynamic fields. It is located in `$OTOBO_HOME/Kernel/System/DynamicField.pm`.

DynamicField/Value.pm This module is responsible to read and write dynamic field values into the database. It is frequently used by the driver modules and is located in `$OTOBO_HOME/Kernel/System/DynamicField/Value.pm`.

Dynamic Fields Database Tables (New API)

There are three 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_obj_id_name Used by the core module `DynamicField.pm` to save the relationship between objects (with ID and name) and an available object type.

dynamic_field_value Used by the core module `DynamicField/Value.pm` to save the dynamic field values for each dynamic field and each object type instance.

Dynamic Fields Configuration Files (New API)

The master admin, as well as the dynamic field management module needs to know which drivers exists.

The easiest way to manage them is to use the system configuration, where the information of dynamic field drivers, their paths and meta information and object type modules can be stored and extended.

Front end 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 the New Phone Ticket screen.

3.3.25 Dynamic Field Interaction With Front End Modules

Knowing about how front end 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 front end module.

The following pictures are showing a simple example of how the dynamic fields interact with other OTOBO framework parts.

Dynamic Field Interaction (Legacy API)

The first step is, that the front end 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

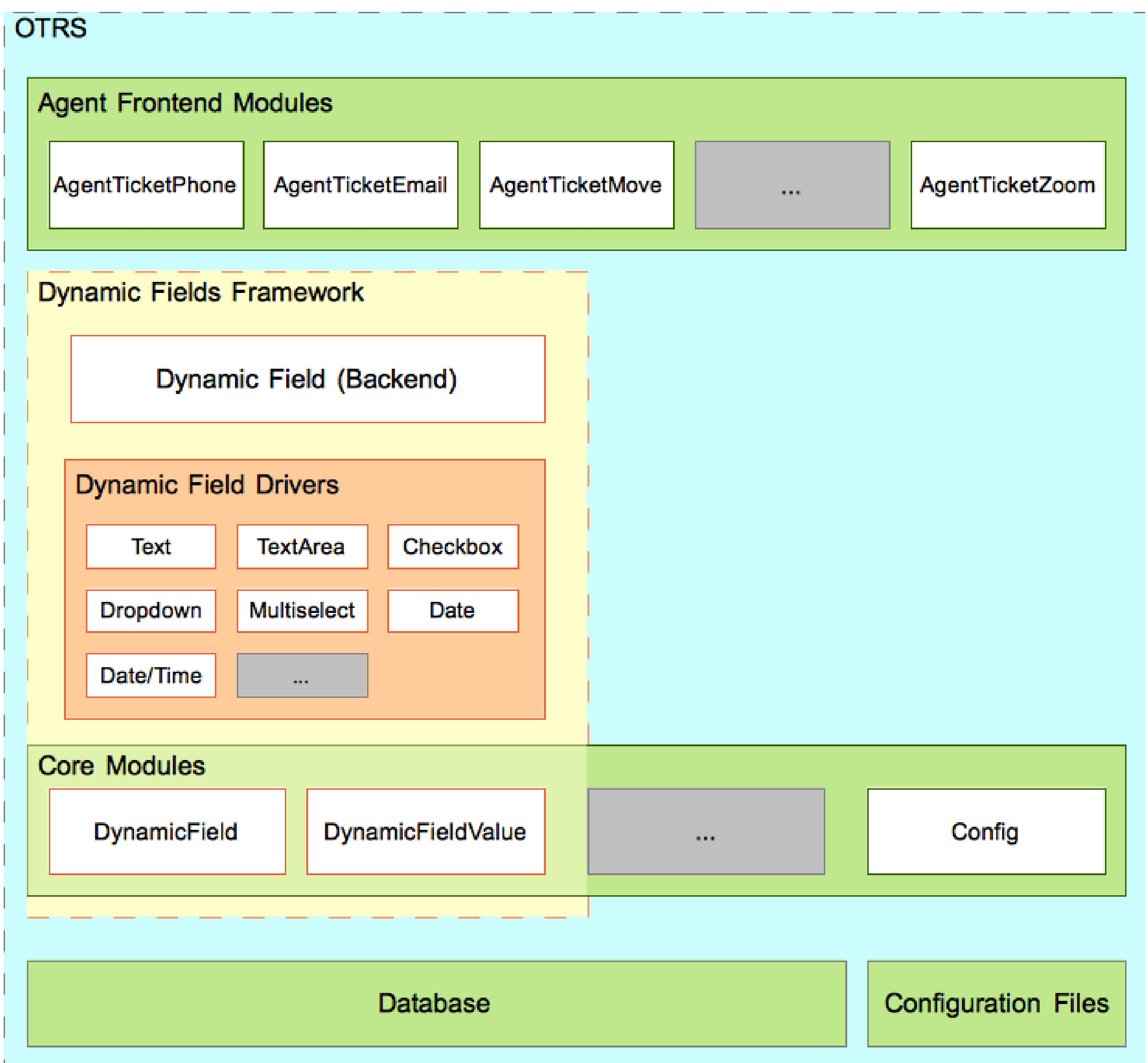

Fig. 3.6: Dynamic Field Interaction

`DynamicFieldListGet()`. The screen can store the results of this function, to have the list of the dynamic fields activated for this particular screen.

The screen should try to get the values from the web request subsequently. It can use the back end object function `EditFieldValueGet()` for this purpose and might use this values to trigger ACLs. The back end object will use each driver, to perform the specific actions for all functions.

Afterwards the screen should get the HTML for each field to display. The back end object function `EditFieldRender()` can be used to perform this action and the ACLs restrictions. On top of that, 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 back end object 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, it 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, it is necessary to create the article first. `ValueSet()` from the back end object can be used to set the dynamic field value.

In summary, the front end 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 back end object module and use the fields in a generic way.

Dynamic Field Interaction (New API)

The first step is, that the front end 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 `FieldListGet()`. The screen can store the results of this function, to have the list of the dynamic fields activated for this particular screen.

After that, the screen should load the role `ProvidesForm` and define the method `FormSchema()`, to build the full schema, including the dynamic fields. For each dynamic field, we should call the function `FormFieldSchema()` that will return the specific schema for the field.

In case of a submit, the screen could also use the dynamic field driver instance function `ValueValidate()`, to check the submitted value according to the field configuration.

Note: Other screens could use `DisplayFieldData()` instead of `FormFieldSchema()`, 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, it 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, it is necessary to create the article first. `ValueSet()` from the dynamic field driver can be used to set the dynamic field value.

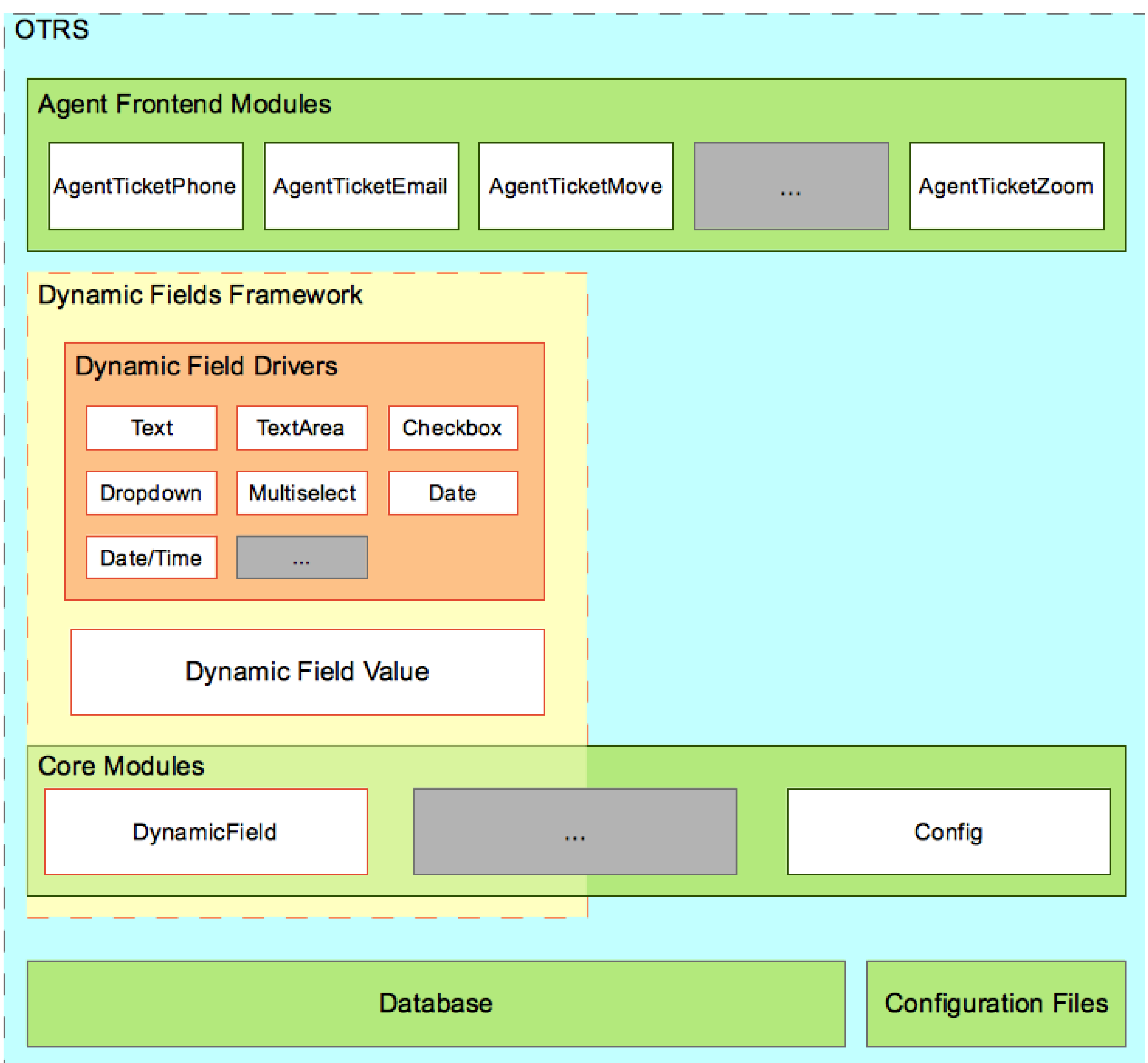

Fig. 3.7: Dynamic Field Interaction

3.3.26 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.

Create a New Dynamic Field Type (for ticket or article objects)

To create a new dynamic field type is necessary to:

1. Create a dynamic field driver. This is the main module of the new field.
2. Create or use an existing admin dialog to have a management interface and set its configuration options.
3. Create a configuration file to register the new field in the back end (or new admin dialogs in the framework if needed) and be able to create instances or it.

Create a New Dynamic Field Type (for other objects)

To create a new dynamic field type for other objects is necessary to:

1. Create a dynamic field driver. This is the main module of the new field.
2. 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 back end 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.
3. Create or use an existing admin dialog to have a management interface and set its configuration options.
4. Implement dynamic fields in the front end modules to be able to use the dynamic fields.
5. Create a configuration file to register the new field in the back end (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.

Create a New package to Use Dynamic Fields

To create a package to use existing dynamic fields is necessary to:

1. Implement dynamic fields in the front end modules to be able to use the dynamic fields.
2. 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.

Extend Back End and Drivers Functionalities (Legacy API)

It might be possible that the back end 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 back end 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 back end object and the drivers.

The only restrictions are that the functions should be named different than the ones on the back end and drivers, otherwise they will be overwritten with current objects.

Put the new back end extension into the `DynamicFieldLegacy` directory (e.g. `/$OTOBO_HOME/Kernel/System/DynamicFieldLegacy/NewPackageBackend.pm` and its drivers in `/$OTOBO_HOME/Kernel/System/DynamicFieldLegacy/Driver/NewPackage*.pm`).

New behaviors only need a small setting in the extensions configuration file.

To create new back end functions is needed to:

1. Create a new back end extension module to define only the new functions.
2. Create the dynamic fields driver extensions to implement only the new functions.
3. Implement new dynamic fields functions in the front end modules to be able to use the new dynamic fields functions.
4. Create a configuration file to register the new back end and drivers extensions and behaviors.

Extend Back End and Drivers Functionalities (New API)

It might be possible that the current drivers or the main module doesn't provide the needed functions or behavior for custom developments.

The easiest way to extend the main module is to create a new role file that defines the new functions or overrides the standard ones. Either a completely new driver module can be created from scratch or an existent one can be extended, using a role file.

Put the files into the `DynamicField` directory:

1. Main module role in `$OTOBO_HOME/Kernel/System/DynamicField/Role/*.pm` (create it if does not exists).
2. New driver in `$OTOBO_HOME/Kernel/System/DynamicField/Driver/*.pm`.
3. Driver role in `$OTOBO_HOME/Kernel/System/DynamicField/Driver/Role/*.pm`.

Finally, to load the new roles, create a new `Mojolicious` plugin in `$OTOBO_HOME/Kernel/WebApp/Plugin/*.pm`, applying the roles to the modules, for example:

```
package Kernel::WebApp::Plugin::4100DynamicFieldExtensions;
use strict;
use warnings;

use Moose::Util;

sub register {    ## no critic

    Moose::Util::ensure_all_roles(
        'Kernel::System::DynamicField',
        'Kernel::System::DynamicField::Role::ExtendedFieldDelete',
    );

    Moose::Util::ensure_all_roles(
        'Kernel::System::DynamicField::Driver::AnyAvailableDriver*',
        'Kernel::System::DynamicField::Driver::Role::ExtendedValueDelete',
    );
```

```
    return 1;
}

1;
```

Other Extensions

Other extensions could be a combination of the above examples.

3.3.27 Create 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 (for the legacy API) and `Base/Text` (for the new API) as a basis to build this new field.

> **Warning:** 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 5 files:

1. A configuration file (XML) to register the modules.
2. An admin dialog module (Perl) to setup the field options.
3. A template module for the admin dialog.
4. A dynamic field legacy driver (Perl).
5. A dynamic field driver (Perl).

File structure:

```
$HOME (e. g. /opt/otobo/)
|
...
|--/Kernel/
|    |--/Config/
|    |    |--/Files/
|    |    |    |--/XML/
|    |    |    |    |DynamicFieldPassword.xml
...
|    |--/Modules/
|    |    |AdminDynamicFieldPassword.pm
...
|    |--/Output/
|    |    |--/HTML/
|    |    |    |--/Standard/
|    |    |    |    |AdminDynamicFieldPassword.tt
...
|    |--/System/
|    |    |--/DynamicFieldLegacy/
|    |    |    |--/Driver/
|    |    |    |    |Password.pm
```

```
...
|   |   |--/DynamicField/
|   |   |   |--/Driver/
|   |   |   |   |Password.pm
...
```

Dynamic Field Password Files

Dynamic Field Configuration File Example

The configuration files are used to register the dynamic field types (driver) and the object type drivers for the back end object. They also store standard registrations for admin modules in the framework.

In this section a configuration file for password dynamic field is shown and explained.

```
<?xml version="1.0" encoding="utf-8" ?>
<otobo_config version="2.0" init="Application">
```

This is the normal header for a configuration file.

```
<ConfigItem Name="DynamicFieldsLegacy::Driver###Password" Required="0" Valid="1">
    <Description Translatable="1">DynamicField legacy 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::DynamicFieldLegacy::Driver::Password</
↪Item>
            <Item Key="ConfigDialog">AdminDynamicFieldPassword</Item>
        </Hash>
    </Setting>
</ConfigItem>

<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>
```

These settings registers the password dynamic field driver for the back end module (legacy and new API) 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.

```
</otobo_config>
```

Standard closure of a configuration file.

Dynamic Field Admin Dialog Example

The admin dialogs are standard admin modules to manage (add or edit) the dynamic fields.

In this section an admin dialog for password dynamic field is shown and explained.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO 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.

Dynamic Field Template for Admin Dialog Example

The template is the place where the HTML code of the dialog is stored.

In this section an admin dialog template for the password dynamic field is shown and explained.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO 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 Save button and the Cancel link, as well as other needed JavaScript code.

Dynamic Field Driver Example

The driver represents the dynamic field. It contains several functions that are used wide in the OTOBO framework.

A driver can inherit some functions from base classes, for example the `TextArea` driver inherits most of the functions from `Base.pm` and `BaseText.pm` (`Base/Text.pm` in the new API) and it only implements the functions that requires different logic or results. The checkbox field driver only inherits from `Base.pm`, as all other functions are very different from any other base driver.

See also:

Please refer to the Perl online documentation (POD) of the module `/Kernel/System/DynmicFieldLegacy/Backend.pm` and `/Kernel/System/DynmicFieldLegacy/Driver/Base.pm` to have the list of all attributes and possible return data for each function.

In this section the password dynamic field driver is shown and explained. This driver inherits some functions from `Base.pm` and `BaseText.pm` (`Base/Text.pm` in the new API) and only implements the functions that needs different results.

Dynamic Field Driver Example (Legacy API)

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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::DynamicFieldLegacy::Driver::Password;

use strict;
use warnings;

use parent qw(Kernel::System::DynamicFieldLegacy::Driver::BaseText);

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

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

This is the common header, that can be found in common OTOBO modules. The class/package name is declared via the `package` keyword. Note that `BaseText` is used as the 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(
↪'DynamicFieldsLegacy::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 subroutine `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 responsible to create the HTML representation of the field and its label. It 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;
}
```

The `DisplayValueRender()` function returns the field value as 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 it is used in locations, where no `LayoutObject` is available.

Other Functions (Legacy API)

The following are other functions, that might be 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 file `Kernel/System/DynamicFieldLegacy/Driver/Base.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 { ... }
```

This function 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 { ... }
```

This 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 { ... }
```

This 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 { ... }
```

This function 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 { ... }
```

This function is used in the edit screens of OTOBO 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 front end 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 checkbox 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 drop-down 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 { ... }
```

This function 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 { ... }
```

This function is very similar to the `SearchFieldParameterBuild()`. The difference is that the latter 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 { ... }
```

This 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 `otobo.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.

Dynamic Field Driver Example (New API)

```
# --
# Kernel/System/DynamicField/Driver/Password.pm - Driver for DynamicField backend
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 Moose;

extends 'Kernel::System::DynamicField::Driver::Base::Text';

our @ObjectDependencies = ();
```

This is the common header that can be found in the new OTOBO modules since OTOBO 10.

The class/package name is declared via the `package` keyword. Note that the `Base::Text` is used as the base class.

There's no need to declare the constructor `BUILD`, unless you need to do something really specific to the driver, during the creation of a new class instance.

Note: Drivers are created only by the `DynamicFieldObject` and not directly from any other module.

```
override 'FormFieldSchema' => sub {
    return {
```

```
        %{ super() },
        Type => 'FormPassword',
    };
};
```

This function is responsible to return the schema of the field, that should be used in edit screens, like `AgentTicketPhone`, `AgentTicketNote`, etc.

Other Functions (New API)

The following are other functions, that are might be needed to implement or override, according to the specification of the driver. Please refer to the file `Kernel/System/DynamicField/Driver/Base.pm`.

```
sub ValueGet { ... }
```

This function retrieves the value from the field, on a specific object. In this case, we are returning the first text value, since the field only stores one text value at a time.

```
sub ValueSet { ... }
```

This function 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 `Text`, `DateTime` or `Integer` format.

```
sub ValueDelete { ... }
```

This function is used to delete the values of a certain dynamic field. If no filter is passed, all the values will be deleted. To execute the deletion only for one object, we can pass the filter `Filters => { ObjectID => '...' }`.

```
sub ValueValidate { ... }
```

This function is used to check, if the value is consistent to its type and dynamic field instance.

```
sub ValueList { ... }
```

This function is used to get the list of the values for the dynamic field instance.

```
sub SearchSQLGet { ... }
```

This function is used by `TicketSearch` core module, to build the internal query that searches for a ticket, based on this field as a search parameter.

```
sub SearchSQLOrderFieldGet { ... }
```

This function is also a helper for `TicketSearch` module. `$Param{TableAlias}` should be kept and `_ValueDBColumn()` changed in case the driver uses multiple columns, to store it's data.

```
sub SearchConditionGet { ... }
```

This function is used by the ticket search and statistics module, to set the correct operator and value to the search on this field.

```
sub RandomValueSet { ... }
```

This function is used by `otobo.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.

3.3.28 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 back end object as well as in the text field driver.

To create this extension we will create 4 files:

1. A configuration file (XML) to register the modules.
2. A back end extension (Perl) to define the new function (legacy API).
3. A text field driver extension (Perl) that implements the new function for text fields (legacy API).
4. A text field driver role (Perl) that implements the new function for the text fields (new API).

File structure:

```
$HOME (e. g. /opt/otobo/)
|
...
|--/Kernel/
|    |--/Config/
|    |    |--/Files/
|    |    |    |--/XML/
|    |    |    |DynamicFieldFooExtension.xml
...
|    |--/System/
|    |    |--/DynamicFieldLegacy/
|    |    |    FooExtensionBackend.pm
|    |    |    |--/Driver/
|    |    |    |    |FooExtensionText.pm
...
|    |--/System/
|    |    |--/DynamicField/
|    |    |    |--/Driver/
|    |    |    |    |--/Role/
|    |    |    |    |    |FooExtensionText.pm
```

Dynamic Field Foo Extension files

Dynamic Field Extension Configuration File Example (Legacy API)

The configuration files are used to register the extensions for the back end and drivers as well as new behaviors for each drivers.

Note: If a driver is extended with a new function, the back end will need also an extension for that function.

In this section a configuration file for `Foo` extension is shown and explained.

```
<?xml version="1.0" encoding="utf-8" ?>
<otobo_config version="2.0" 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 back end 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.

```
</otobo_config>
```

Standard closure of a configuration file.

Dynamic Field Back End Extension Example (Legacy API)

Back end extensions will be loaded transparently into the back end itself as a base class. All defined object and properties from the back end will be accessible in the extension.

Note: All new functions defined in the back end extension should be implemented in a driver extension.

In this section the `Foo` extension for back end is shown and explained. The extension only defines the function `Foo()`.

```
# --
# Copyright (C) 2001-2020 OTOBO AG, https://otobo.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 OTOBO 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 front end module to require a special behavior on the dynamic field, and only after call the function in the back end object.

Dynamic Field Driver Extension Example (Legacy API)

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 back end extension, as every function is called from the back end object.

In this section the `Foo` extension for text field driver is shown and explained. The extension only implements the function `Foo()`.

```
# --
# Copyright (C) 2001-2020 OTOBO AG, https://otobo.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 DESCRIPTION

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.

=cut
```

This is common header that can be found in common OTOBO 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.

Dynamic Field Driver Extension Example (New API)

```
# --
# Kernel/System/DynamicField/Driver/Role/FooExtensionText.pm - Extension for␣
↪DynamicField Text Driver
# Copyright (C) 2001-2019 OTOBO AG, https://otobo.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::Role::FooExtensionText;

use strict;
use warnings;

use Moose::Role;

=head1 NAME

Kernel::System::DynamicField::Driver::Role::FooExtensionText

=head1 DESCRIPTION

DynamicFields Text Driver Extension

=head1 PUBLIC INTERFACE

This module extends the public interface of L
↪<Kernel::System::DynamicField::Driver::Text>.
Please look there for a detailed reference of the functions.

=cut
```

This is common header that can be found in common OTOBO role 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.

Now, to apply the role to the driver we need to create an `Mojolicious` plugin that will take care of that.

```
# --
# Kernel/WebApp/Plugin/4500DynamicFields.pm - Dynamic fields extensions.
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 Moose::Util;

sub register {    ## no critic

    Moose::Util::ensure_all_roles(
        'Kernel::System::DynamicField::Driver::Text',
        'Kernel::System::DynamicField::Driver::Role::FooExtensionText',
```

```
    );

    return 1;
}
```

3.3.29 Ticket Postmaster Module

Postmaster modules are used during the postmaster process. There are two kinds of postmaster modules:

- `PostMasterPre`: used after parsing an email.
- `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-OTOBO-Headers (see `doc/X-OTOBO-Headers.txt` for more info).

`Kernel/Config/Files/XML/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-OTOBO-Ignore">yes</Item>
                </Hash>
            </Item>
        </Hash>
    </Setting>
</ConfigItem>
```

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

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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.

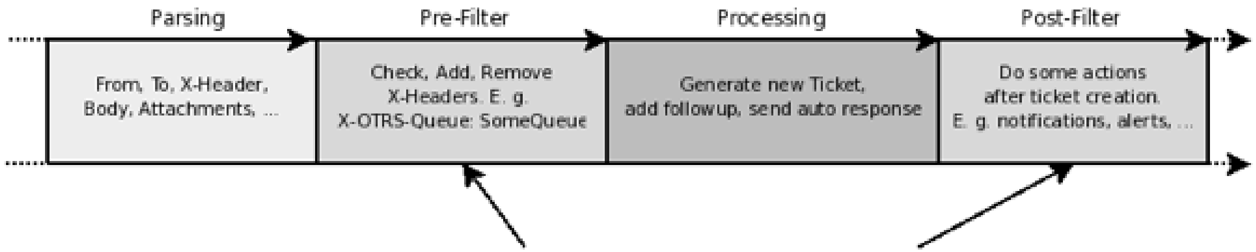

Fig. 3.8: Email Processing Flow

3.3.30 Process Management

Since OTOBO 10 process management can use script modules to perform activities (script tasks) and/or sequence flow actions (know before as transition actions). This modules are located in `Kernel::System::ProcessManagement::Modules`.

Process Management Modules

The process management modules are scripts written in Perl language to perform certain action or actions over the process ticket like set a dynamic field or change a queue. Or to any other part of the system like create new tickets.

By default modules use a set of key value pairs to use them as parameters for the action e.g. to change queue of the process ticket, the queue or queue ID and its corresponding value is needed.

Some scripts might require more than a simple key value pairs as parameters or its configuration might need to have a more user friendly GUI. In such cases OTOBO provides some configuration field types that can be also extended if needed.

Current field types:

Dropdown Shows a drop-down list with predefined values.

Key-value list Shows a list of simple key value pairs (text inputs). Pairs can be added or deleted.

Multi language Rich Text Shows a Rich Text editor associated to a system language, also shows a language selector to add Rich Text fields for the proper selected language.

Recipients Shows a multi select field pre-filled with agents to be used as email recipients, also displays a free input field to be used to specify external email addresses to be added to the recipients list.

Rich Text Shows a single Rich Text field.

Creating a New Process Management Module

The following is an example of how to create a process management module. It includes a section where all possible fields are defined as a reference. To create a new module only one field type is needed but consider that by convention the parameter user ID is used to impersonate all the actions in the module with another user than the one that triggers the action. Then it is a good practice to always include the key-value list field type along with any other needed field.

Process Management Module Code Example

The following code implements a dummy process management module that can be used in script task activities or sequence flow actions.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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::ProcessManagement::Modules::Test;

use strict;
use warnings;
use utf8;

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

use parent qw(Kernel::System::ProcessManagement::Modules::Base);

our @ObjectDependencies = ( );
```

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

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

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

    # Allocate new hash for object.
    my $Self = {};
```

```
    bless( $Self, $Type );

    $Self->{Config} = {
        Description => 'Description for the module.',
        ConfigSets  => [
            {
                Type        => 'Dropdown',
                Description => 'Description for Dropdown.',
                Mandatory   => 1,
                Config      => {
                    Data => {
                        Internal1 => 'Display 1',
                        Internal2 => 'Display 2',
                    },
                    Name         => 'Test-DropDown',
                    Multiple     => 1,
                    PossibleNone => 1,
                    Sort         => 'AlphanumericValue',
                    Translation  => 1,
                },
            },
            {
                Type        => 'KeyValueList',
                Description => 'Description of the Key/Value pairs',
                Defaults    => [
                    {
                        Key       => 'Test-Param1',
                        Value     => 'Hello',
                        Mandatory => 1,
                    },
                    {
                        Key       => 'Test-Param2',
                        Mandatory => 1,
                    },
                    {
                        Key       => 'Test-Param3',
                        Value     => 'World',
                        Mandatory => 0,
                    },
                ],
            },
            {
                Type        => 'MultiLanguageRichText',
                Description => "Description for Mutli-Language Rich Text.",
                Defaults    => [
                    {
                        Key   => 'en_Subject',
                        Value => 'Hello',
                    },
                    {
                        Key   => 'en_Body',
                        Value => 'World',
                    },

                ],
            },
            {
                Type        => 'Recipients',
```

```
                Description => "Description for Recipients."
            },
            {
                Type        => 'RichText',
                Description => "Description for Rich Text.",
                Defaults    => [
                    {
                        Value     => 'Hello World',
                        Mandatory => 1,
                    },
                ],
            },
        ],
    };

    return $Self;
}
```

The constructor `new()` creates a new instance of the class. The configuration fields are defined here and they are set in `$Self->{Config}`.

The configuration has two main entries:

Description This is used to explain the administrators what does the module do and/or considerations for its configuration.

ConfigSets This is just a container for the actual configuration fields.

All configuration fields requires a type that defines the kind of field and they could also have an internal description to be used as the title of the field widget. If it is not defined, a default description is used.

Each field defines its configuration parameters and capabilities. The following is a small reference for the fields provided by OTOBO out of the box.

- `Dropdown`

  Mandatory Used to define if a value is required to be set.

  Config Holds the information to display the drop-down field.

    Data Simple hash that defines the options for the drop-down. The keys are used internally, and the values are the options that the user sees in the screen.

    Name The name of the parameter.

    Multiple To define if only one or multiple values can be selected.

    PossibleNone Defines if the list of values offer an empty value or not.

    Sort Defines how the options will be sorted when the field is rendered. The possible values are: `AlphanumericValue`, `NumericValue`, `AlphanumericKey` and `NumericKey`.

    Translation: Set if the displayed values should be translated.

- `KeyValueList`

  Defaults Array of hashes that holds the default configuration for its key value pairs.

    Key The name of a parameter.

    Value The default value of the parameter (optional).

    Mandatory Mandatory parameters can not be renamed or removed (optional).

- `MultiLanguageRichText`

  Defaults Array of hashes that holds the default configuration each language and field part.

  Key This is composed by language such as `en` or `es_MX`, followed by a _ (underscore character) and then `Subject` or `Body` for the corresponding part of the field.

  Value The default value of the field part (optional).

- `Recipients`

  No further configuration is provided for this kind of field.

- `RichText`

  Defaults Array of hashes that holds the default configuration field (only the first element is used).

  Value The default value of the field.

  Mandatory Used to define if a value is required to be set.

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

    # Define a common message to output in case of any error.
    my $CommonMessage = "Process: $Param{ProcessEntityID} Activity: $Param
↪{ActivityEntityID}";

    # Add SequenceFlowEntityID to common message if available.
    if ( $Param{SequenceFlowEntityID} ) {
        $CommonMessage .= " SequenceFlow: $Param{SequenceFlowEntityID}";
    }

    # Add SequenceFlowActionEntityID to common message if available.
    if ( $Param{SequenceFlowActionEntityID} ) {
        $CommonMessage .= " SequenceFlowAction: $Param{SequenceFlowActionEntityID}";
    }

    # Check for missing or wrong params.
    my $Success = $Self->_CheckParams(
        %Param,
        CommonMessage => $CommonMessage,
    );
    return if !$Success;

    # Override UserID if specified as a parameter in the TA config.
    $Param{UserID} = $Self->_OverrideUserID(%Param);

    # Use ticket attributes if needed.
    $Self->_ReplaceTicketAttributes(%Param);
    $Self->_ReplaceAdditionalAttributes(%Param);

    # Get module configuration.
    my $ModuleConfig = $Param{Config};

    # Add module logic here!

    return 1;
}
```

The `Run` method is the main part of the module. First sets a common message that can be used in error logs or any other purpose. For consistency its highly recommended to use it as described above.

Next step is to check if the global parameters was sent correctly.

By convention all modules should be able to override the current user ID by one is provided in the parameters (if any). This passed user ID should be used in any function call that requires it.

User defined attribute values can use current ticket values by using OTOBO smart tags. `_ReplaceTicketAttributes` is used for normal text attributes, while `_ReplaceAdditionalAttributes` for Rich Texts. For more complex parameters it might need customized functions to replace this smart tags.

The following is the proper logic of the module.

If everything was OK it must return 1.

Creating a New Process Management Module Configuration Field

The following is an example of how to create a process management module configuration field. This field can be used by any process management module after its configuration.

Process Management Module Configuration Field Code Example

The following code implements a simple input process management module configuration field (test). The name of the field and its default value can be set trough a process management module `ConfigSets`.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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::ProcessManagement::ModuleConfiguration::Test;

use strict;
use warnings;

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

our @ObjectDependencies = (
    'Kernel::Output::HTML::Layout',
    'Kernel::System::Web::Request',
);
```

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

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

    # allocate new hash for object
    my $Self = {%Param};
```

```
    bless( $Self, $Type );

    return $Self;
}
```

The constructor `new` creates a new instance of the class.

Every configuration field requires to implement at least 2 main methods: `Render` and `GetParams`.

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

    my $ConfigSet = $Param{ConfigSet} // {};
    my $EntityConfig = $Param{EntityData}->{Config}->{Config}->{ConfigTest} // {};

    my %Data;

    $Data{Description} = $ConfigSet->{Description} || 'Config Parameters (Test)';
    $Data{Name} = $ConfigSet->{Config}->{Name} // 'Test';
    $Data{Value} = $EntityConfig->{ $ConfigSet->{Config}->{Name} } // $ConfigSet->
↪{Defaults}->[0]->{Value} // '';

    return {
        Success => 1,
        HTML    => $Kernel::OM->Get('Kernel::Output::HTML::Layout')->Output(
            TemplateFile => 'ProcessManagement/ModuleConfiguration/Test',
            Data         => \%Data,
        ),
    };
}
```

`Render` method is responsible to create the required HTML for the field.

In this example it first localize some parameters for more easy read and maintain.

The following lines set the data to display. The field widget title `Description` is gather from the `ConfigSet` if defined, otherwise it uses a default text. Similar to the field `Name`, for the `Value` it first checks if the activity or sequence flow action already have a stored value, if not it tries to use the default value from the `ConfigSet`, or use empty otherwise.

At the end it returns a structure with the HTML code from a template filled with the gathered data.

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

    my %GetParams;
    my $Config = $Param{ConfigSet}->{Config} // 'Test';

    $GetParams{ $Config->{Name} } = $Kernel::OM->Get('Kernel::System::Web::Request')->
↪GetParam( Param => $Config->{Name} );

    return \%GetParams;
}
```

For this example the `GetParams` method is very straight forward. It gets the name of the field from the `ConfigSet` or use a default, and gets the value from the web request.

Process Management Module Configuration Field Template Example

The following code implements a basic HTML template for the test process management module configuration field.

```
# --
# Copyright (C) 2019-2021 Rother OSS GmbH, https://otobo.de/
# --
# 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 OTOBO modules.

```
<div id="TestConfig" class="WidgetSimple Expanded">
    <div class="Header">
        <div class="WidgetAction Toggle">
            <a href="#" title="[% Translate("Show or hide the content") | html %]"><i␣
↪class="fa fa-caret-right"></i><i class="fa fa-caret-down"></i></a>
        </div>
        <h2 for="Config">[% Translate(Data.Description) | html %]</h2>
    </div>
    <div class="Content" id="TestParams">
        <fieldset class="TableLike">
            <label for="[% Data.Name | html %]">[% Data.Name | html %]</label>
            <div class="Field">
                <input type="text" value="[% Data.Value | html %]" name="[% Data.Name␣
↪| html %]" id="[% Data.Name | html %]" />
            </div>
        </fieldset>
    </div>
</div>
```

The template shows a simple text input element with its associated label.

3.3.31 Autoload Modules

Autoload modules are a measure of last resort when the regular module layers do not suffice. These modules live in the namespace `Kernel::Autoload`. They are required whenever a new config is loaded. This assures that they are available in the web interface and in just about all scripts. The primary use of autoload modules is to add, or to override, core functionality. Please handle with care!

Agent Authentication Module Configuration Example

In order to activate an autoload module it must be declared in the system configuration. See for example the activation of our sample module `Kernel::Autoload::Test` in `Kernel/Config/Files/XML/Framework.xml`.

```
<Setting Name="AutoloadPerlPackages###1000-Test" Required="0" Valid="0">
    <Description Translatable="1">Example package autoload configuration.</Description>
    <Navigation>Core::Autoload</Navigation>
    <Value>
        <Array>
            <Item ValueType="String">Kernel::Autoload::Test</Item>
```

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

Adding Methods to Core Modules

Methods are added simple by declaring a subroutine in the target namespace.

```
use Kernel::System::Valid;    ## no critic (Modules::RequireExplicitPackage)

package Kernel::System::Valid;    ## no critic␣
↪(Modules::RequireFilenameMatchesPackage)

use strict;
use warnings;
use v5.24;
use utf8;

sub AutoloadTest {
    return 1;
}
```

See `scripts/test/Config/Autoload.t` for an example of how the new subroutine can be used.

Because of the line `use Kernel::System::Valid` we could also override existing methods. But that is not recommended.

Modifying Existing Methods of Core Modules ———————————-~~~~~~~~~~~~~-

Often it suffices to modify existing code. This can by overriding the relevant subroutine.

```
package Kernel::Autoload::Test;    ## no critic qw(Modules::ProhibitMultiplePackages)

use strict;
use warnings;
use v5.24;
use utf8;

use Kernel::System::State;

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

{
    no warnings 'redefine';    ## no critic qw(TestingAndDebugging::ProhibitNoWarnings)

    # keep reference to the original StateLookup()
    my $Orig = \&Kernel::System::State::StateLookup;

    # redefine StateLookup
    *Kernel::System::State::StateLookup = sub {
        my $Self = shift;

        $Kernel::OM->Get('Kernel::System::Log')->Log(
            Priority => 'info',
            Message  => 'Calling the modified method Kernel::System::State::StateLookup
↪',
```

```
        );

        my $Result = $Orig->( $Self, @_ );

        # return a default value
        return $Result // 'unknown state';
    };
}
```

CHAPTER 4

How to Publish Your OTOBO Extensions

4.1 Package Management

The OPM (OTOBO Package Manager) is a mechanism to distribute software packages for the OTOBO framework via HTTP, FTP or file upload.

For example, the OTOBO project offers OTOBO modules like a calendar, a file manager or web mail in OTOBO packages via online repositories on our FTP servers. The packages can be managed (install, upgrade and uninstall) via the admin interface.

4.1.1 Package Distribution

If you want to create an OPM online repository, just tell the OTOBO framework where the location is by activating the system configuration 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. OTOBO just reads this index file and knows what packages are available.

```
shell> bin/otobo.Console.pl Dev::Package::RepositoryIndex /path/to/repository/ > /
↪path/to/repository/otobo.xml
```

4.1.2 Package Commands

You can use the following OPM commands over the admin interface or over `bin/otobo.Console.pl` to manage admin jobs for OPM packages.

Install

Install OPM packages.

- Web: http://localhost/otobo/index.pl?Action=AdminPackageManager
- CMD: `bin/otobo.Console.pl Admin::Package::Install /path/to/package.opm`

Uninstall

Uninstall OPM packages.

- Web: http://localhost/otobo/index.pl?Action=AdminPackageManager
- CMD: `bin/otobo.Console.pl Admin::Package::Uninstall /path/to/package.opm`

Upgrade

Upgrade OPM packages.

- Web: http://localhost/otobo/index.pl?Action=AdminPackageManager
- CMD: `bin/otobo.Console.pl Admin::Package::Upgrade /path/to/package.opm`

List

List all OPM packages.

- Web: http://localhost/otobo/index.pl?Action=AdminPackageManager
- CMD: `bin/otobo.Console.pl Admin::Package::List`

4.2 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.

4.2.1 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.

`<Name>` * The package name.

```
<Name>Calendar</Name>
```

`<Version>` * The package version.

```
<Version>1.2.3</Version>
```

`<Framework>` * The targeted framework version (7.0.x means e.g. 7.0.1 or 7.0.2).

```
<Framework>7.0.x</Framework>
```

Can also be used several times.

```
<Framework>5.0.x</Framework>
<Framework>6.0.x</Framework>
<Framework>7.0.x</Framework>
```

<Vendor> * The package vendor.

```
<Vendor>Rother OSS GmbH</Vendor>
```

<URL> * The vendor URL.

```
<URL>https://otobo.com/</URL>
```

<License> * The license of the package.

```
<License>GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007</License>
```

<ChangeLog> The package change log.

```
<ChangeLog Version="1.1.2" Date="2018-11-15 18:45:21">Added some feature.</
↪ChangeLog>
<ChangeLog Version="1.1.1" Date="2018-11-15 16:17:51">New package.</ChangeLog>
```

<Description> * The package description in different languages.

```
<Description Lang="en">A web calendar.</Description>
<Description Lang="de">Ein Web Kalender.</Description>
```

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>
```

A special package action is `PackageAllowDirectUpdate`. Only if it is defined on the package and set to true, a package can be upgraded from a lower major version (earlier than the last one) to the latest version. (e.g. a package for OTOBO 5 updated to OTOBO 7).

```
<PackageAllowDirectUpdate>1</PackageAllowDirectUpdate>
```

<BuildHost> This will be filled in automatically by OPM.

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

<BuildDate> This will be filled in automatically by OPM.

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

<PackageRequired> Packages that must be installed beforehand. 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>
```

<ModuleRequired> Perl modules that must be installed beforehand.

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

<OS> Required OS.

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

<Filelist> This is a list of files included in the package.

```
<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>
```

<DatabaseInstall> Database entries that have to be created when a package is installed.

```
<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.

<DatabaseUpgrade> Information on which actions have to be performed in case of an upgrade.

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.

<DatabaseReinstall> Information on which actions have to be performed if the package is reinstalled.

```
<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.

`<DatabaseUninstall>` Actions to be performed on package uninstall.

```
<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.

`<IntroInstall>` To show a pre or post install introduction in installation dialog.

```
<IntroInstall Type="post" Lang="en" Title="Some Title"><![CDATA[
    Some information 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 newlines and whitespace of the content).

`<IntroUninstall>` To show a pre or post uninstall introduction in uninstallation dialog.

```
<IntroUninstall Type="post" Lang="en" Title="Some Title"><![CDATA[
    Some information 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).

`<IntroReinstall>` To show a pre or post reinstall introduction in re-installation dialog.

```
<IntroReinstall Type="post" Lang="en" Title="Some Title"><![CDATA[
    Some information 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).

`<IntroUpgrade>` To show a pre or post upgrade introduction in upgrading dialog.

```
<IntroUpgrade Type="post" Lang="en" Title="Some Title"><![CDATA[
    Some information 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).

`<CodeInstall>` Perl code to be executed when the package is installed.

```
<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.

<CodeUninstall> Perl code to be executed when the package is uninstalled. On pre or post time of package uninstallation.

```
<CodeUninstall><![CDATA[
# Some Perl code.
]]></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.

<CodeReinstall> Perl code to be executed when the package is reinstalled.

```
<CodeReinstall><![CDATA[
# Some Perl code.
]]></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.

<CodeUpgrade> Perl code to be executed when the package is upgraded (subject to `version` tag).

Example if already installed package version is below 1.3.4 (e. g. 1.2.6), the defined action will be performed:

```
<CodeUpgrade Version="1.3.4"><![CDATA[
# Some Perl code.
]]></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.

<PackageMerge> This tag signals 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.

Package Conditions `IfPackage` and `IfNotPackage` 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.

4.2.2 Example .sopm

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

```
<?xml version="1.0" encoding="utf-8" ?>
<otobo_package version="1.0">
    <Name>Calendar</Name>
    <Version>10.0.1</Version>
    <Framework>10.0.x</Framework>
    <Vendor>Rother OSS GmbH</Vendor>
    <URL>https://otobo.com/</URL>
    <License>GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007</License>
    <ChangeLog Version="1.1.2" Date="2018-11-15 18:45:21">Added some feature.</
↪ChangeLog>
    <ChangeLog Version="1.1.1" Date="2018-11-15 16:17:51">New package.</ChangeLog>
    <Description Lang="en">A customer package.</Description>
    <Description Lang="de">Ein kundenspezifisches Paket.</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>
</otobo_package>
```

4.2.3 Package Build

To build an .opm package from the spec opm.

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

4.2.4 Package Life Cycle

The following image shows you how the life cycle of a package installation, upgrade and uninstallation works in the back end step by step.

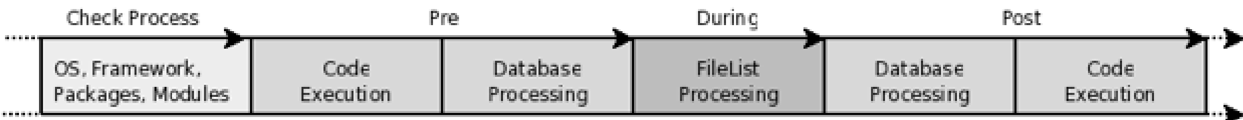

Fig. 4.1: Package Life Cycle

4.3 Package Porting

With every new minor or major version of OTOBO, you need to port your packages and make sure they still work with the OTOBO API.

This section lists changes that you need to examine when porting your package from OTOBO 7 to 8.

4.3.1 Front End Messages

In an effort to improve consistency and enforce new UX guidelines, a new and common front end messages API has been introduced in OTOBO 8. Developers should strive to use only this way of informing users of application changes. A front end component has been created for this feature, and it's included by default in every application.

The API has been reused from a previous integration, albeit with some changes. By emitting an event on the global event bus it is still possible to trigger display of a message on the user screen. For example, in order to show a toast style message, you can just emit an event like so:

```
this.$bus.$emit('showToastMessage', {
    id: 'aSampleToastMessage'
    heading: 'This is a %s heading',
    headingPlaceholders: [ 'toast' ],
    text: 'This is a %s message text.',
    textPlaceholders: [ 'toast' ],
    variant: 'warning',
});
```

If your use case is to prevent user from interacting with the app until they make a choice or acknowledge a message, you can trigger display of a blocking modal message. The interface is similar, please note the different name of the event:

```
this.$bus.$emit('showModalMessage', {
    id: 'aSampleModalMessage'
    heading: 'This is a %s heading',
    headingPlaceholders: [ 'modal' ],
    text: 'This is a %s message text.',
    textPlaceholders: [ 'modal' ],
    buttonBehavior: 'yesNo',
});
```

Both message styles provide numerous configuration options, so make sure to consult the component documentation in the design system for complete API description.

Renamed Message Events

Front end message component improvements meant that some event names had to be renamed, it will be expected from you to port existing packages that use them to conform to the new format. Please find a table below containing all affected events.

Old name	New name
showNotification	showToastMessage
clearNotification	hideToastMessage

4.3.2 Styling Improvements

In OTOBO 7, new front end stack has been introduced. However, since only one major front end application was shipped (for the external interface), there was no need to make sure that components living in the shared namespace have support for different styling. With OTOBO 8, this was bound to change, and now OTOBO supports app-specific styling in components.

Application Specific Styling in Shared Components

Usage is quite simple: whenever you need to apply an app-specific style, wrap your SCSS code in the following mixin:

```
@include Mixin__StylesForApp('External') {
    .ElementName {
        &__SubElement {
            color: $Color__External__Primary;

            &:hover {
                color: $Color__External__Highlight;
            }
        }
    }
}
```

Usage of the mixin guarantees your block will only be loaded where appropriate (in this example in the external interface app). In the block you are free to use application specific global variables and functions, without any restrictions. Everything that is available within the app will be allowed.

Note: Design system will load appropriate style depending on current choice. For example, there will be a drop-down menu shown above the component example, allowing the user to switch the styles. All you have to do is make sure your component lives in the shared namespace and implements the mixin above, as needed.

In order to provide this mechanism, it was required to refactor all the styles to conform to the BEM specification. This now applies to any variable names, mixins, functions, etc. By looking at their names, now it will be perfectly clear where they are coming from, and the chance for any possible collisions is decreased to the minimum.

Renamed SCSS Literals

Since some SCSS literals had to be renamed, it will be expected from you to port existing packages that use them to conform to the new format. Please find a table below containing all affected literals.

Namespace	Origin	Old name	New name
Shared	_colors	`$base`	`$Color__Base`
Shared	_colors	`$base`	`$Color__Base`
Shared	_colors	`$white`	`$Color__White`
Shared	_colors	`$alert`	`$Color__Alert`
Shared	_colors	`$warning`	`$Color__Warning`
Shared	_colors	`$success`	`$Color__Success`
Shared	_colors	`$shadow`	`$Color__Shadow`

Continued on next page

Table 4.1 – continued from previous page

Namespace	Origin	Old name	New name
Shared	_colors	$black100	$Color__Black100
Shared	_colors	$black90	$Color__Black90
Shared	_colors	$black80	$Color__Black80
Shared	_colors	$black70	$Color__Black70
Shared	_colors	$black50	$Color__Black50
Shared	_colors	$black30	$Color__Black30
Shared	_colors	$black20	$Color__Black20
Shared	_colors	$black10	$Color__Black10
Shared	_colors	$black4	$Color__Black4
Shared	_functions	calculateRem	Function__CalculateRem
Shared	_mixins	border-radius	Mixin__BorderRadius
Shared	_mixins	list-reset	Mixin__ListReset
Shared	_mixins	FontSize	Mixin__FontSize
Shared	_mixins	MarginBottom--Responsive	Mixin__MarginBottom--Responsive
Shared	_mixins	shadow	Mixin__Shadow
Shared	_mixins	placeholder	Mixin__Placeholder
Shared	_mixins	float-label-container	Mixin__FloatLabel__Container
Shared	_mixins	float-label	Mixin__FloatLabel
Shared	_mixins	float-label-input	Mixin__FloatLabel__Input
Shared	_mixins	float-label-scaled	Mixin__FloatLabel--Scaled
Shared	_mixins	linearGradient	Mixin__LinearGradient
Shared	_mixins	linearGradientoverlay	Mixin__LinearGradient--Overlay
External	_variables	$container-max-width	$Variable__External__ContainerMaxWidth
External	_variables	$spacing-small	$Variable__External__Spacing--Small
External	_variables	$spacing-medium	$Variable__External__Spacing--Medium
External	_variables	$font-small	$Variable__External__FontSize--Small
External	_variables	$font-smaller	$Variable__External__FontSize--Smaller
External	_variables	$font-smallest	$Variable__External__FontSize--Smallest
External	_colors	$primary	$Color__External__Primary
External	_colors	$primary-darker	$Color__External__Primary--Darker
External	_colors	$primary-dark	$Color__External__Primary--Dark
External	_colors	$primary-lighter	$Color__External__Primary--Lighter
External	_colors	$primary-light	$Color__External__Primary--Light
External	_colors	$highlight	$Color__External__Highlight
External	_colors	$highlight-darker	$Color__External__Highlight--Darker
External	_colors	$highlight-dark	$Color__External__Highlight--Dark
External	_colors	$highlight-lighter	$Color__External__Highlight--Lighter
External	_colors	$highlight-light	$Color__External__Highlight--Light
External	_colors	$gray-light	$Color__External__Gray--Light

4.3.3 Encode API Changed

The legacy method `Convert2CharsetInternal()` was dropped. Please replace any usages of this with `Convert()` and a `To => 'utf-8'` parameter like this:

```
$EncodeObject->Convert2CharsetInternal(
    Text  => $BodyStrg,
    From  => $Self->GetCharset(),
    Check => 1,
);
```

Replace this by:

```
$EncodeObject->Convert(
    Text  => $BodyStrg,
    From  => $Self->GetCharset(),
    To    => 'utf-8',
    Check => 1,
);
```

4.3.4 LinkObject API Changed

The method `LinkAdd()` has a slightly changed return value. Instead of a boolean return value it returns now the `LinkID` of the added link. You need to save the `LinkID` in order to delete a link later.

```
$True = $LinkObject->LinkAdd(
    SourceObject => 'Ticket',
    SourceKey    => '321',
    TargetObject => 'FAQ',
    TargetKey    => '5',
    Type         => 'ParentChild',
    State        => 'Valid',
    UserID       => 1,
);
```

Replace this by:

```
my $LinkID = $LinkObject->LinkAdd(
    SourceObject => 'Ticket',
    SourceKey    => '321',
    TargetObject => 'FAQ',
    TargetKey    => '5',
    Type         => 'ParentChild',
    State        => 'Valid',
    UserID       => 1,
);
```

The method `LinkDelete()` has a changed signature and return value. Instead of a boolean return value it returns now the `LinkData` as a hash. The parameter list now only requires the `LinkID` and the `UserID`.

```
$True = $LinkObject->LinkDelete(
    Object1 => 'Ticket',
    Key1    => '321',
    Object2 => 'FAQ',
    Key2    => '5',
    Type    => 'Normal',
    UserID  => 1,
);
```

Replace this by:

```
my %LinkData = $LinkObject->LinkDelete(
    LinkID => 4,
    UserID => 1,
);
```

4.3.5 Event Handling Changes

The event handling was changed from the previous `Kernel::System::Event` and `Kernel::System::EventHandler` modules to the Moose role based `Kernel::System::Event::Handler` which handles all event types and modules in dedicated event queues.

Back end files emitting events (i.e. containing `EventHandler()` calls) have to be modified to use the new event handling role.

Remove code like this:

```
use Kernel::System::EventHandler;

@ISA = qw(
    Kernel::System::EventHandler
);

sub new {
    ...
    $Self->EventHandlerInit(
        Config => 'AppointmentCalendar::EventModulePost',
    );
    ...
}

sub DESTROY {
    ...
    $Self->EventHandlerTransaction();
    ...
}
```

Replace this by:

```
# Unless already used in module.
use Moose;

with 'Kernel::System::Role::EmitsEvents';

sub EventModuleType {
    # Same module type definition as in previous EventHandlerInit.
    return 'AppointmentCalendar::EventModulePost';
}

sub EmitsEventObjectTypes {
    # All relevant object types (as per system configuration definition).
    return ['Calendar'];
}

# Unless already used in module.
no Moose;
```

In order to ensure the correct behavior it is imperative that all possible events for an object type are known via the system configuration (e.g. `Events###Ticket` for all ticket events). **This configuration is now required**. Exceptions for dynamically created events like those of dynamic fields have to be added to `Kernel::System::Event::Handler::_EventListBuild`.

A configuration like this:

```
<Setting Name="Events###AnObjectType" Required="0" Valid="1">
    <Description Translatable="1">List of all AnObjectType events to be displayed in␣
↪the GUI.</Description>
    <Navigation>Frontend::Admin</Navigation>
    <Value>
        <Array>
            <Item>ObjectCreate</Item>
            <Item>ObjectDelete</Item>
        </Array>
    </Value>
</Setting>
```

Should be modified and amended as necessary, like this:

```
<Setting Name="Events###AnObjectType" Required="1" Valid="1">
    <Description Translatable="1">List of all AnObjectType events to be displayed in␣
↪the GUI.</Description>
    <Navigation>Frontend::Admin</Navigation>
    <Value>
        <Array>
            <Item>ObjectCreate</Item>
            <Item>ObjectDelete</Item>
            <Item>ObjectUpdate</Item>
        </Array>
    </Value>
</Setting>
```

As the generic interface provides event filters for every object type, it is now **mandatory** to provide a module for every object type which retrieves object data for the filter. These modules reside in `Kernel/GenericInterface/Event/ObjectType`.

If an event list is required in your code and you have an occurrence of this:

```
my %RegisteredEvents = $Kernel::OM->Get('Kernel::System::Event')->EventList( ... );
```

Replace this by:

```
my %RegisteredEvents = $Kernel::OM->Get('Kernel::System::Event::Handler')->
↪EventListGet( ... );
```

4.3.6 `MojoUserAgent` Added, `WebUserAgent` Deprecated

The legacy `Kernel::System::WebUserAgent` was deprecated and `Kernel::System::MojoUserAgent` provided as a modern alternative.

Now it is possible to use the full Mojo::UserAgent API on instances of the new HTTP user agent.

Note: The legacy `WebUserAgent` will be removed in a future version of OTOBO. Please update all code that used it to the new object.

You can replace old code like:

```
$Kernel::OM->ObjectParamAdd(
    'Kernel::System::WebUserAgent' => {
        Timeout => $RequestTimeout,
```

```
        Proxy    => $RequestProxy,
    },
);

my %Response = $Kernel::OM->Get('Kernel::System::WebUserAgent')->Request(
    Type => 'POST',
    URL  => $Self->{CloudServiceURL},
    Data => {
        Action       => 'PublicCloudService',
        RequestData  => $RequestData,
        UniqueIDAuth => $UniqueIDAuth,
        OTOBOIDAuth   => $OTOBOIDAuth,
    },
);

if ( $Response{Status} eq '200 OK' && $Response{Content} ) { ... }
```

Replace this by:

```
my $MojoUserAgent = Kernel::System::MojoUserAgent->new(
    Timeout => $RequestTimeout,
    Proxy   => $RequestProxy,
);

my $Response = $MojoUserAgent->post(
    $Self->{CloudServiceURL},
    form => {
        Action       => 'PublicCloudService',
        RequestData  => $RequestData,
        UniqueIDAuth => $UniqueIDAuth,
        OTOBOIDAuth   => $OTOBOIDAuth,
    },
)->res();

if ( $Response->code() == 200 && $Response->body() ) { ... }
```

CHAPTER 5

Documentation

In OTOBO 10 the documentations are available in reStructuredText format. Various outputs are available on the OTOBO documentation page, like HTML, EPUB and PDF.

The documentation is written in English and translated into many languages.

5.1 Documentation Infrastructure

OTOBO uses Sphinx to generate the outputs. The HTML output is generated using the Read the Docs theme.

Note: All outputs are customized on the build server. If you want to setup a developer environment in your local machine for testing and writing the documentation, the outputs can be different.

5.2 reStructuredText Primer

The documentation format name is **reStructuredText** (one word, this is the correct spelling). This is an easy to read documentation format using plain text and small inline markers.

This short tutorial will guide you through to create or update documentations. To give a full featured tutorial about how to use the reStructuredText format is beyond the scope of this document, and many tutorials (e. g. Sphinx reStructuredText primer and reStructuredText user documentation) and on-line editors are available on the internet.

The following examples shows the most commonly used documentation elements.

5.2.1 Headings

To use heading in the documentation, you have to underline the titles with special characters. The underline must start from the first letter of the title and end at the last letter of the title. The hierarchy of the special characters are the following: =, -, ~, ^, .

The following example shows the usage of the headings:

```
Chapter title
=============

This is the heading 1 title. It has numbering like 1.

Section title
-------------

This is the heading 2 title. It has numbering like 1.1.

Subsection title
~~~~~~~~~~~~~~~~

This is the heading 3 title. It has numbering like 1.1.1.

Subsubsection title
^^^^^^^^^^^^^^^^^^^

This is the heading 4 title. It has numbering like 1.1.1.1.

Subsubsubsection title
......................

This is the heading 5 title. It has numbering like 1.1.1.1.1. Please don't use this␣
↪level of heading.
```

5.2.2 Paragraphs

For writing paragraphs, you have to start sentences at the beginning of the line. To create a new paragraph, just leave a blank line between the paragraphs. Example:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed dictum imperdiet enim.␣
↪Curabitur
nisi diam, lobortis facilisis quam ut, porttitor consequat lectus. Nam elementum,␣
↪ipsum id
feugiat vestibulum, dolor ante dictum quam, ac bibendum ipsum felis in orci.

Vestibulum maximus egestas orci, eget consequat nibh imperdiet eget. Suspendisse␣
↪sagittis tempus
sapien, sit amet tincidunt tortor efficitur et. Etiam ac lacus sem. Sed ut magna␣
↪imperdiet,
viverra quam vitae, consequat mauris.
```

5.2.3 Inline Markups

The standard inline markup is quite simple:

- One asterisk: *text* for emphasis (italics).

- Two asterisks: **text** for **strong emphasis** (boldface).
- Grave accents: ``text`` for `literal texts` (code samples).

If asterisks or grave accents appear in running text which could be confused with inline markup delimiters, they have to be escaped with a backslash, like *.

5.2.4 Lists

To create unordered lists, start a line with asterisk (*) or dash (-). To create ordered list, start a line with numbers or hash mark (#). If you need nested lists, leave a blank line between the list items and use indentation with 3 spaces. Example:

```
* This is a bulleted list.
* It has two items, the second
  item uses two lines.

1. This is a numbered list.
2. It has two items too.

#. This is a numbered list.
#. It has two items too.
```

Nested list example:

```
- this is
- a list

  - with a nested list
  - and some subitems

- and here the parent list continues
```

5.2.5 Literal Blocks

Literal blocks are texts that should be displayed as verbatim. To create literal blocks, do the following:

1. Type 2 colons (::) in a new line.
2. Leave a blank line.
3. Write the text with indentation of 3 spaces.

Use literal blocks for code snippets, terminal outputs, configuration files, etc. Example:

```
::

   $Self->{DatabaseHost} = '127.0.0.1';
   $Self->{Database} = 'otobo';
   $Self->{DatabaseUser} = 'otobo';
```

If the language of the code snippet is known, you can specify it for syntax highlighting:

```
.. code-block:: perl

   $Self->{DatabaseHost} = '127.0.0.1';
   $Self->{Database} = 'otobo';
   $Self->{DatabaseUser} = 'otobo';
```

```
.. code-block:: xml

   <Setting Name="FAQ::Agent::StateTypes" Required="1" Valid="1">
       <Description Translatable="1">List of state types which can be used in the␣
↪agent interface.</Description>
       <Navigation>Core::FAQ</Navigation>
       <Value>
           <Array>
               <Item>internal</Item>
               <Item>external</Item>
               <Item>public</Item>
           </Array>
       </Value>
   </Setting>
```

5.2.6 Tables

To create grid tables, you have to draw the table. Example:

```
+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | ...        | ...      |          |
+------------------------+------------+----------+----------+
```

5.2.7 Hyperlinks

Hyperlinks can be used inline or referenced. For inline use, encapsulate the text of the link and the URL with grave accents and two trailing underscore characters.

```
Visit `OTOBO website <https://otobo.de>`__ for more information.
```

The link above will display as: OTOBO website.

To create referenced links, you have to separate the text and the link. Example:

```
The documentations are available in the `OTOBO documentation portal`_.

.. _OTOBO documentation portal: https://doc.otobo.de/
```

5.2.8 Images

To insert an image into the documentation:

1. Put the image in the `images` folder.
2. Create a reference to the image with:

```
.. figure:: images/admin-general-catalog-management-class.png
   :alt: Admin General Catalog

   Admin General Catalog
```

5.2.9 Colored Boxes

These boxes have special meanings and will be highlighted as default.

Warning box:

```
.. warning::

   This is a warning box.
```

Warning: This is a warning box.

Note box:

```
.. note::

   This is a note box.
```

Note: This is a note box.

See also box:

```
.. seealso::

   This is a see also box.
```

See also:

This is a see also box.

5.3 Style Guide

This part of the documentation is only for visual style and wording.

5.3.1 Writing Content

There is an internet slang **TL;DR**, which means too long, didn't read (see more information on Wikipedia). Many people don't like reading long texts, so please keep the documentation as short as possible. Use step-by-step tutorials instead of writing wall of text.

For example this is a **wrong** example for writing content:

```
The agents are able to change the interface language of OTOBO. To change the
interface language, click on your avatar on the top left corner, then select
Personal Settings menu item. A new screen will be displayed. On this screen
click on the User Profile, and then find a widget named Language. Select the
desired language in the drop-down menu. Please make sure to click on the
Save button next to the language widget.
```

The same content in **suggested** human understandable format:

```
To change the interface language of OTOBO:

1. Click on your avatar on the top left corner.
2. Select *Personal Settings*.
3. Click the *User Profile* in the new screen.
4. Choose a language from the drop-down menu of the *Language* widget.
5. Click the *Save* button next to the widget.
```

The latter is easier to translate, because 6 short sentences will be included in the language file. If a content is changed in one of the sentences, only the changed sentence need to be reviewed and translated again. The first wrong example puts only one huge string to the language file, and if some changes will be made in the source string, the translator needs to review and re-translate the whole string.

5.3.2 Screenshots

Don't use the native resolution of your machine. Usually it is full HD or bigger, so creating a screenshot with this resolution will became unreadable in some output, because all the images have to be shrink to the width of A4 paper in case of PDF. OTOBO uses responsive design, so 1025 pixels is the minimum, that OTOBO assumes it is a large display. Please use this as width of your screenshots.

See also:

The resolution can be set in all web browser with a feature called mobile mode or responsive design. Check your browser user manual for the usage of the feature and set the width of the screen to 1025 pixels.

This is an example for a **wrong** screenshot, as it has resolution of full HD. Due to the automatic shrinking the texts on the screenshot are hard to read:

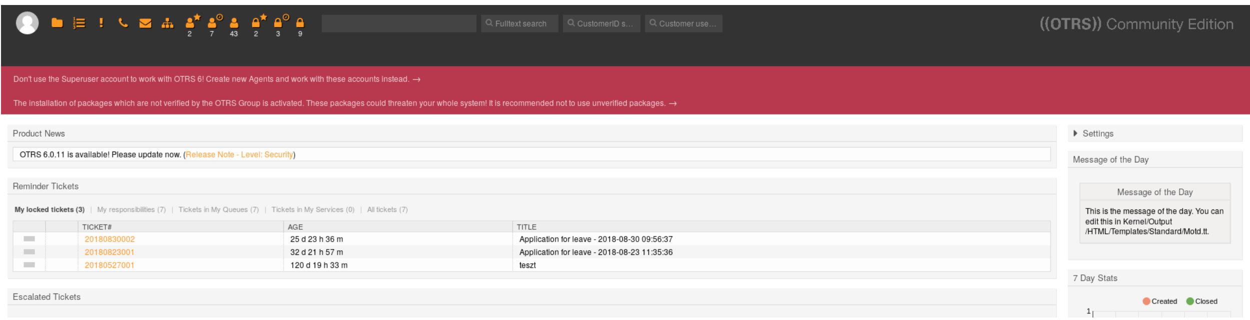

Fig. 5.1: Agent Dashboard (1920 pixels width)

The same screenshot with **suggested** resolution. The texts are much easier to read:

It is also wrong, if the screenshot has good resolution in pixels, but with high DPI. For example this screenshot is **wrong**, because the texts on it is much bigger than the other texts in the documentation:

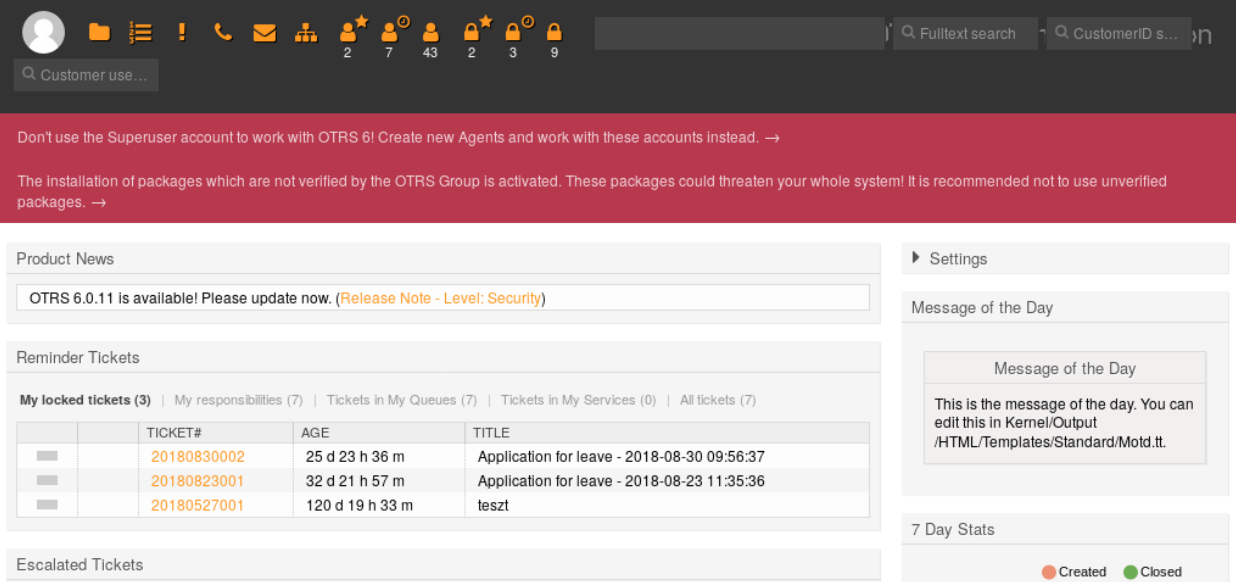

Fig. 5.2: Agent Dashboard (1025 pixels width)

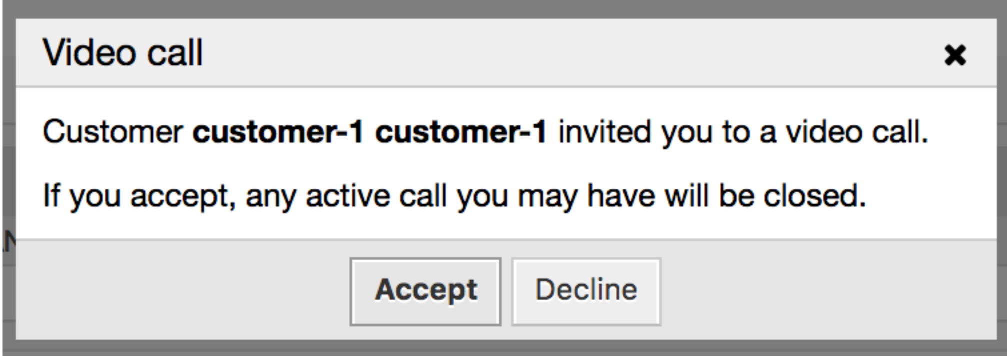

Fig. 5.3: Video Invitation Dialog (756 pixels width but with high DPI)

Create Screenshots with Firefox

If only a part of the screenshot is required, the screenshot needs to be cropped. The administrator interface of OTOBO consist of a left sidebar and a main content column. To create screenshots with Firefox:

1. Right click on an element in the browser and select Inspect element.
2. Select the element in the DOM, if it was not selected.
3. Right click on the node and select Screenshot Node.

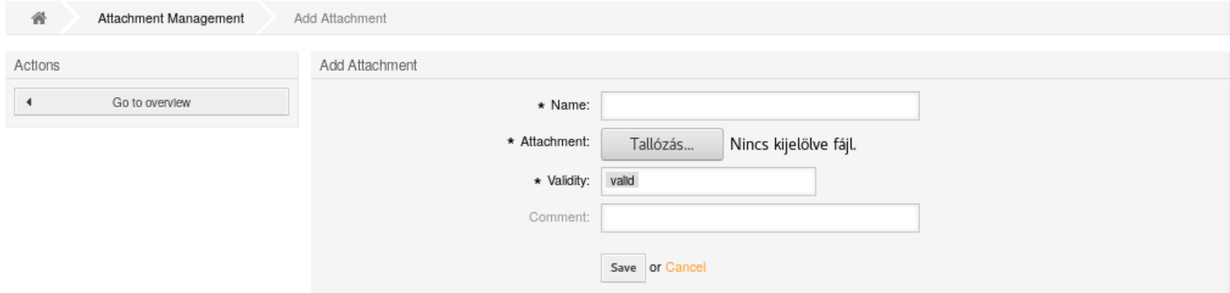

Fig. 5.4: Example screenshot for the main content

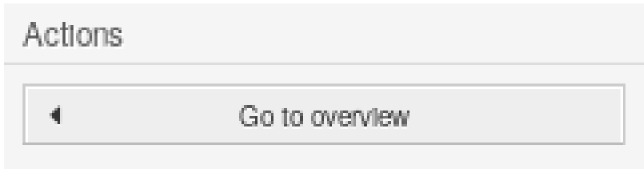

Fig. 5.5: Example screenshot for the left sidebar

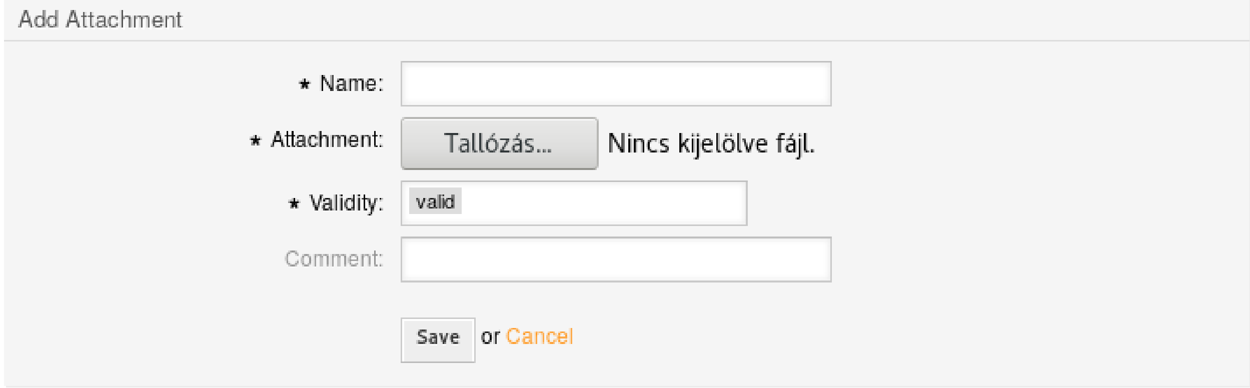

Fig. 5.6: Example screenshot for the main content column

5.3.3 Capitalization in Documentation

For titles always have to use sentence case capitalization, which means, that in titles always capitalize:

- Nouns (man, bus, book).
- Adjectives (angry, lovely, small).
- Verbs (run, eat, sleep).
- Adverbs (slowly, quickly, quietly).
- Pronouns (he, she, it).
- Subordinating conjunctions (as, because, that).

In titles do not capitalize:

- Articles: a, an, the.
- Coordinating conjunctions: and, but, or, for, nor, etc.
- Prepositions (fewer than five letters): on, at, to, from, by, etc.

In normal sentences don't capitalize any words, only names and reference to titles have to be capitalized. This is a **wrong** example:

```
An Agent is a user, who handles Tickets in the Ticket Zoom screen.
```

The **suggested** sentence with proper capitalization. Besides, Ticket Zoom is the name of the screen, so it should be emphasized:

```
An agent is a user, who handles tickets in the *Ticket Zoom* screen.
```

5.3.4 Buttons and Screen Names

In the content sentences all buttons and screens should be emphasized and should be written with capital letters or in sentence case. Don't use apostrophes or quotation marks for emphasizing.

This sentence is **wrong**, because apostrophes are used for emphasizing:

```
If you click the 'Save and Finish' button, you will be redirected to the 'Ticket Zoom
↪' screen.
```

The **suggested** way is to use asterisks for emphasizing:

```
If you click the *Save and Finish* button, you will be redirected to the *Ticket␣
↪Zoom* screen.
```

5.3.5 Wording

Don't use variable names in sentences. This sentence is **wrong**, because a variable name is meaningless for some people:

```
Add a new widget to AgentTicketZoom.
```

The same sentence without variable name, this is **suggested**:

```
Add a new widget to the *Ticket Zoom* screen of the agent interface.
```

5.3.6 Variable Names

Variable names should always marked as `literal` content. This is useful for translators, as they can exactly know, that the string mustn't be translated. If a string is not marked as literal content, it usually should be translated. For example:

```
The ``ObjectManager`` object has an ``Init()`` function. Additional configuration can␣
↪be set in ``Kernel::Config::Config.pm`` file.
```

5.4 Translating the Documentation

All translations of the OTOBO GUI, the public extension modules and the documentations are managed using Weblate.

In OTOBO 10 the documentations are available in reStructuredText format. Be careful not to break the structure while translating the documentation.

Here are some examples.

Emphases Emphasized texts are between two asterisks. The text should be translated. This is usually used for screen names, titles, buttons and labels. Please check the user interface translations to find and use the same wording in the documentation.

Example original sentence:

```
Use the *Ticket Zoom* screen to see the ticket details.
```

Example translation into Hungarian:

```
Használja a *Jegynagyítás* képernyőt a jegy részleteinek megtekintéséhez.
```

Strong Strong texts are between two double asterisks. The text should be translated. This is usually used for important information.

Example original sentence:

```
**Don't continue** the update if you get an error message.
```

Example translation into Hungarian:

```
**Ne folytassa** a frissítést, ha hibaüzenetet kap.
```

Literal texts Literal texts are between two double back-tick characters. This is usually used for variable names, configuration names and file paths, and **must not** be translated, otherwise it will break the structure.

Example original sentence:

```
Activate ``group`` in system configuration ``ExamplePermission###100``.
```

Example translation into Hungarian:

```
Aktiválja a ``group`` értéket az ``ExamplePermission###100`` rendszerbeállításban.
```

Internal links Internal links point to other pages or headings of the pages. `:doc:`page-name`` is used for referring to a page and `:ref:`Heading Title`` is used for referring to a heading. There is

a custom tag ``:sysconfig:`System Configuration Name` `` for referring to a system configuration. The texts page-name, Heading Title and System Configuration Name **must not** be translated, otherwise it will break the structure.

Example original sentence:

```
See page :doc:`queues` to add a queue, especially section :ref:`Queue Settings`.
```

Example translation into Hungarian:

```
Nézze meg a :doc:`queues` oldalt, különösen a :ref:`Queue Settings` szakaszt.
```

External links External links consist of a visible text and an URL in form ‘visible text <https://example.com>‘__. The visible text should be translated.

Example original sentence:

```
See `OTOBO website <https://otobo.de/>`__ for more information.
```

Example translation into Hungarian:

```
Nézze meg az `OTOBO weboldalát <https://otobo.de/>`__ a további információkért.
```

CHAPTER 6

Contributing to OTOBO

This chapter will show how you can contribute to the OTOBO framework, so that other users will be able to benefit from your work.

6.1 Sending Contributions

The source code of OTOBO and additional 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 OTOBO code quality checker OTOBOCodePolicy as described in the Useful Tools 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 OTOBO 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:

1. Register at GitHub, if you have no account yet.
2. Fork the repository you want to contribute to, and checkout the branch that the changes should go in.
3. Create a new development branch for your fix/feature/contribution, based on the current branch.
4. After you finished your changes and committed them, push your branch to GitHub.
5. Create a pull request. The OTOBO 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.

6.2 Translating

The translations are contributed and maintained mainly by OTOBO users, so your help is needed.

All translations of the OTOBO GUI, the public extension modules and the documentations are managed using Weblate.

To contribute to translations:

1. Sign up for a free translator account on Weblate.
2. Select a translation component and your language for translation.
3. Start updating your translation. No additional software or files required.

Note: If your language is not listed in the dashboard, you can request a language. After it is approved, you can start translating.

In OTOBO 10 the documentations are available in reStructuredText format. Be careful not to break the structure while translating the documentation.

See also:

You can find some examples in the Documentation chapter.

The OTOBO developers will download the translations from time to time into the OTOBO source code repositories, you don't have to submit them anywhere.

6.3 Code Style Guide

In order to preserve the consistent development of the OTOBO project, we have set up guidelines regarding style for the different programming languages.

6.3.1 Perl

Whitespace

TAB: We use 4 spaces. Examples for braces:

```
if ($Condition) {
    Foo();
}
else {
    Bar();
}

while ($Condition == 1) {
    Foo();
}
```

Length of Lines

Lines should generally not be longer than 120 characters, unless it is necessary for special reasons.

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;
```

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 Rother OSS GmbH, https://otobo.de/
# --
# 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 Rother OSS GmbH, https://otobo.de/
# --
# 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.
# --
```

Conditions

Conditions can be quite complex and there can be chained conditions (linked with logical or or and operations). When coding for OTOBO, 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]->{otobo_stats}[1]{StatType}[1]{Content}
    && $XMLHash[0]->{otobo_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}
    )
    )
{ ... }
```

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';
```

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.

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 system configuration 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.

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.

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)";
```

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.

Variable 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.

Handling of Parameters

To fetch the parameters passed to subroutines, OTOBO 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.
- Calling a function with named parameters is much more readable.

Multiple Named Parameters

If a function call requires more than one named parameter, split them into multiple lines.

Use this:

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

Instead of this:

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

`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
}
```

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;
```

`use` Statements

`use strict` and `use warnings` have to be the first two uses 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;
```

Objects and Their Allocation

In OTOBO many objects are available. But you should not use every object in every file to keep the front end/back end 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 front end modules (`Kernel/Modules`).

Documenting Back End 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
```

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.

Code Comments in Perl

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.
```

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.

Returning on Errors

Whenever you use database functions you should handle errors. If anything goes wrong, return from subroutine:

```
return if !$Self->{DBObject}->Prepare( ... );
```

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,
);
```

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.

6.3.2 JavaScript

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.

Directory Structure

Directory structure inside the `var/httpd/htdocs/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
```

Thirdparty Code

Every thirdparty module gets its own subdirectory: module name-version number (e.g. ckeditor-4.7.0, jquery-3.2.1). Inside of that, file names should not have a version number or postfix included (wrong: `jquery/jquery-3.2.1.min.js`, right: `jquery-3.2.1/jquery.js`).

JavaScript Variables

Variable names should be CamelCase, just like in Perl.

Variables that hold a jQuery object should start with `$`, for example: `$Tooltip`.

Functions

Function names should be CamelCase, just like in Perl.

Namespaces

Code Comments in JavaScript

The Code Comments in Perl 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.

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>')`.

6.3.3 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 templates. All needed JavaScript should be part of the proper library for a certain front end module or of a proper global library. If you need to pass JavaScript data to the front end, use `$LayoutObject->AddJSData()`.

6.3.4 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.

CSS 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.

CSS 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`, etc. into `background`).
- Rules should be in a logical order within a definition (all color specific rule together, all positioning rules together, etc.).
- All IDs and names are written in CamelCase notation:

```
<div class="NavigationBar" id="AdminMenu"></div>
```

6.4 User Interface Design

6.4.1 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.

6.5 Accessibility Guide

This document is supposed to explain basics about accessibility issues and give guidelines for contributions to OTOBO.

6.5.1 Accessibility Basics

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).

Why is it important for OTOBO?

To enable impaired users access to OTOBO 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 OTOBO.

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.
- Don't look at the screen.

Then try to use OTOBO 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.

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. ;)

6.5.2 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 OTOBO. We'll try to extract the relevant guidelines in this document.

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 OTOBO.

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

6.5.3 Implementation guidelines

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 , it can only read link to the user, but not the target of this link. With a slight improvement, it would be accessible: . 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 OTOBO:

- 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 (Delete this entry), or a title attribute ().
- Rule: Images must always have an alternative text that can be read to the user ().

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 OTOBO, 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 `OTOBO.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"/>`

Make interaction possible

Goal: Allow the user to perform all interactions just by using the keyboard.

While it is technically possible to create interactions with JavaScript on arbitrary HTML elements, this must be limited to elements that a user can interact with by using the keyboard. Specifically, they need to be able to give focus to the element and to interact with it. For example, a push button to toggle a widget should not be realized by using a `span` element with an attached JavaScript `onclick` event listener, but it should be (or contain) an `a` tag to make it clear to the screen reader that this element can cause interaction.

- Rule: For interactions, always use elements that can receive focus, such as `a`, `input`, `select` and `button`.
- Rule: Make sure that the user can always identify the nature of the interaction (see rules about non-textual content and labelling of form elements).

Goal: Make dynamic changes known to the user.

A special area of accessibility problems are dynamic changes in the user interface, either by JavaScript or also by AJAX calls. The screen reader will not tell the user about changes without special precautions. This is a difficult topic and cannot yet be completely explained here.

- Rule: Always use the validation framework `OTOBO.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 `OTOBO.UI.Accessibility.AudibleAlert()` to notify the user about other important UI changes.
- Rule: Use the `OTOBO.UI.Dialog` framework to create modal dialogs. These are already optimized for accessibility.

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>`

6.6 Unit Tests

OTOBO provides a test suite which can be used to develop and run unit tests for all system related code.

6.6.1 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 Rother OSS GmbH, https://otobo.de/
# --
# 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, OTOBO 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",
        );
    }
}
```

6.6.2 Prerequisites for Testing

To be able to run the unit tests, you need to have all optional environment dependencies (Perl modules and other modules) installed, except those for different database back ends than what you are using. Run `bin/otobo.CheckEnvironment.pl` to verify your module installation.

You also need to have an instance of the OTOBO web front end running on the FQDN that is configured in your local OTOBO's `Config.pm` file. This OTOBO instance must use the same database that is configured for the unit tests.

6.6.3 Testing

To run your tests, just use `bin/otobo.Console.pl Dev::UnitTest::Run --test Calendar` to use `scripts/test/Calendar.t`.

```
shell:/opt/otobo> bin/otobo.Console.pl Dev::UnitTest::Run --test Calendar
+-------------------------------------------------------------------+
/opt/otobo/scripts/test/Calendar.t:
+-------------------------------------------------------------------+
.....................................................................................
↪...........
=====================================================================
yourhost ran tests in 2s for OTOBO 10.0.x git
All 97 tests passed.
shell:/opt/otobo>
```

You can even run several tests at once, just supply additional `--test` arguments to the command:

```
shell:/opt/otobo> bin/otobo.Console.pl Dev::UnitTest::Run --test Calendar --test␣
↪Appointment
+-------------------------------------------------------------------+
/opt/otobo/scripts/test/Calendar.t:
+-------------------------------------------------------------------+
.....................................................................................
↪...........
+-------------------------------------------------------------------+
/opt/otobo/scripts/test/Calendar/Appointment.t:
+-------------------------------------------------------------------+
.....................................................................................
↪...........................
=====================================================================
yourhost ran tests in 5s for OTOBO 10.0.x git
All 212 tests passed.
shell:/opt/otobo>
```

If you execute `bin/otobo.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.

6.6.4 Unit Test API

OTOBO 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 <https://otobo.github.io/doc/api/otobo/8.0/Perl/Kernel/System/UnitTest.pm.html>`__.

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 <https://doc.otobo.com/doc/api/otobo/8.0/Perl/Kernel/System/UnitTest/Helper.pm.html>`__.

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=otobo_test;host=127.0.0.1;',
    DatabaseUser => 'otobo_test',
    DatabasePw   => 'otobo_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) OTOBO database XML schema to␣
↪execute
                                     # or
    DatabaseXMLFiles => [            # (optional) List of XML files to load and␣
↪execute
        '/opt/otobo/scripts/database/otobo-schema.xml',
        '/opt/otobo/scripts/database/otobo-initial_insert.xml',
    ],
);
```

CHAPTER 7

Additional Resources

otobo.de The OTOBO website with source code, documentation and news is available at www.otobo.de. Here you can also find information about professional services and OTOBO administrator training seminars from Rother OSS GmbH, the creator of OTOBO.