WebApp Developers Manual

Release 7.2.1

Kopano BV

Apr 06, 2021

Contents

1 Introduction **2**
- 1.1 WebApp . . . 2
- 1.2 Requirements . . . 2
- 1.3 Document Structure Overview . . . 2

2 Architecture Overview **4**

3 Ext JS and OO-Javascript **5**
- 3.1 Classes . . . 5
- 3.2 Accessing Components . . . 6
- 3.3 Enums . . . 7
- 3.4 Singletons . . . 8

4 Plugins **10**
- 4.1 Ways of Extending WebApp . . . 10
- 4.2 What's new dialog . . . 11

5 Insertion Points **12**
- 5.1 Enable insertion points . . . 13
- 5.2 Insertion points usage example . . . 13
- 5.3 Example: adding a button . . . 13

6 Themes **16**
- 6.1 Setting the default theme . . . 16
- 6.2 Basic themes . . . 16
- 6.3 JSON themes . . . 17

7 Widgets **19**
- 7.1 Creating a widget . . . 19
- 7.2 Widget configuration . . . 19
- 7.3 Events . . . 21

8 Dialogs **22**
- 8.1 Events . . . 22
- 8.2 Example . . . 22
- 8.3 Dealing with MAPI records . . . 23

9 Bidding System **25**
- 9.1 Bidding and insertion points . . . 25
- 9.2 Working . . . 25
- 9.3 Bidding on a shared component . . . 26
- 9.4 Example . . . 26

10 Data Models **29**
10.1 Model-based architecture . . . 29
10.2 Separable model architecture . . . 30
10.3 Hierarchy model . . . 32
10.4 Settings model . . . 33

11 MAPI **36**

12 Communication **38**
12.1 Conceptual Overview . . . 38
12.2 Protocol . . . 40
12.3 Javascript Communication . . . 41
12.4 Response Handlers . . . 41
12.5 Notifications . . . 42

13 Stores and RecordFactory **44**
13.1 Stores and Substores . . . 44
13.2 RecordFactory . . . 50

14 Deployment and Build System **54**
14.1 Deployment . . . 54
14.2 Build System . . . 56

15 Translations **59**
15.1 Gettext . . . 59
15.2 Server-Side . . . 61
15.3 Client-Side . . . 61

16 Appendix A: Naming Conventions **63**
16.1 Namespace Structure . . . 63
16.2 Naming Packages and Classes . . . 63
16.3 Naming Insertion Points . . . 64
16.4 Documenting Classes . . . 65
16.5 Documenting Fields . . . 65
16.6 Documenting Constructors . . . 66
16.7 Documenting Methods . . . 66
16.8 Documenting Insertion Points . . . 67
16.9 Documenting Enumerations . . . 67

17 Appendix B: References **69**

18 Legal Notice **70**

Edition 2.1 - The Kopano Team

This document, the WebApp Developers Manual, describes how to develop and extend WebApp. In addition various code snippets are provided for an easier incorporate the WebApp code architecture.

CHAPTER 1

Introduction

1.1 WebApp

Kopano WebApp and its plugins are complete as the communication platform of the future. The WebApp is essentially a front-end for a server-side, database driven application (Kopano Core). The user interface portion of the WebApp is written using the Ext JS 3.4 framework, which provides a desktop-like UI, with a programming interface. WebApp architecture is very flexible allowing community to easily create new plugins, contexts or widgets.

Some useful links to online resources are listed in *Appendix B: References*.

1.2 Requirements

WebApp 4 and 5 requires a web server that is capable of running PHP >= 5.4. In later versions, starting with WebApp 6.0 this requirment is changed to php >= 7.2.

Corresponding php-mapi version is required and can either be used to konnect to a remote Kopano Core or an older Zarafa server.

Up to date dependencies can be found here.

For browser support read http://documentation.kopano.io/support_lifecycle_policy.

1.3 Document Structure Overview

The first part covers and represents general information about programming for Kopano WebApp. These chapters contain the things that are relevant to all plug-in developers. An overview of the WebApp architecture is given in *Architecture Overview*, and Ext JS is introduced in *Ext JS and OO-Javascript*. Starting on how to extend WebApp is given in the chapters *Dialogs* and *Bidding System*.

The second part contains advanced information about WebApp. This part explains things that probably won't be used by all plugin developers, for example, additional information about *Data Models* and *MAPI*. Some more detailed explanation of the *Communication* and *Stores and RecordFactory*. Also some advanced things to know, but not obligatory to use, such as the *Deployment and Build System* and dealing with *Translations*.

Appendix A: Naming Conventions contains coding guidelines and recommendations for the code style to use, concerning comments and documentation.

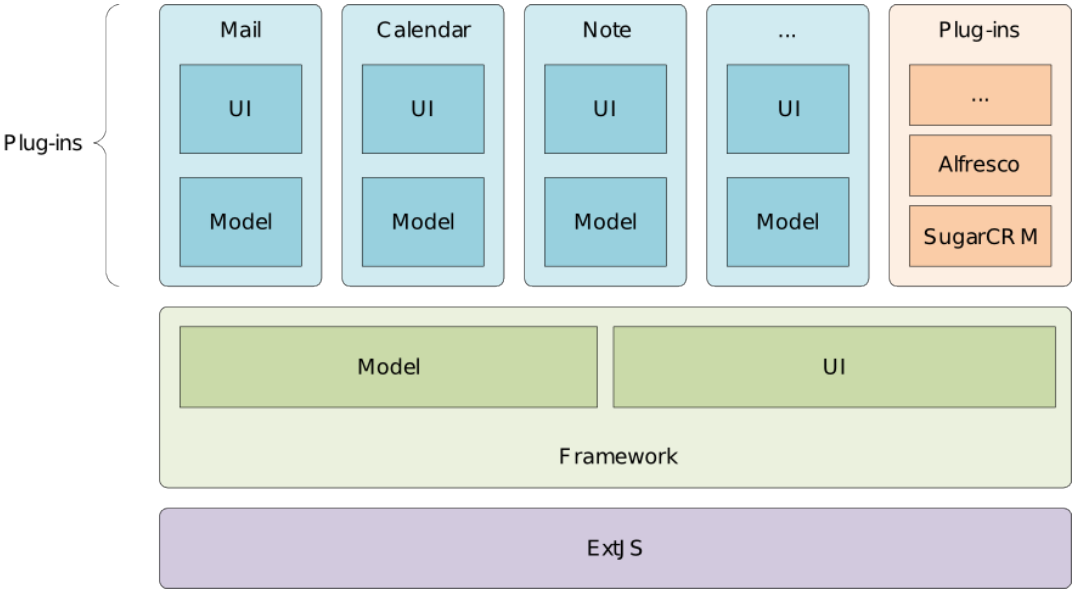
CHAPTER 2

Architecture Overview

WebApp was developed to provide possibilities for third-party developers to add new or build upon existing functionality. These developers have the choice to build a plugin or a widget, but all components are easily extensible and allow easy and fast integration with WebApp.

The base framework for WebApp is Ext JS 3.4 framework, a Javascript toolkit built by Sencha. The Ext JS toolkit was founded on a solid object-oriented inspired design, using classes and inheritance, which makes it particularly suitable for a complex application as this. WebApp is developed using the same extensible structures as those provided by Ext JS, so that the WebApp code integrates well with existing Ext JS components.

"WebApp architecture overview" shows the rough architecture of WebApp. The application framework, provided by Kopano, builds on the Ext JS library to provide all the functionality that is needed across the application. As you can see, even the Mail, Calendar, etc. contexts are plugins building on the core UI and model frameworks.



The framework also provides a user interface infrastructure, with a main screen that carries the standard components that are used by all contexts. The framework supplies a communications API that allows for both low-level and high-level interaction with the server-side back-end.

A more advanced explanation of the architecture of WebApp can be found in *MAPI*, *Ext JS and OO-Javascript* and *Data Models*.

CHAPTER 3

Ext JS and OO-Javascript

JavaScript is not a real object-oriented language, it is a prototyping language, that is the difference with other object-oriented languages. In Javascript, you don't use classes, you create objects from other objects.

When developing your own plugins, it's a good idea to place all of your classes and singletons into namespaces to avoid collisions with other developers' code. With Ext JS, it's easy to do this using Ext.namespace.

3.1 Classes

To declare a new class, one usually starts with defining the *constructor*. It is customary, but not required, to have a "configuration" object as the first parameter. Configuration parameters are object/value pairs that configure specific parts of an object's instantiation, avoiding large sparse parameter lists. The constructor calls the constructor of its parent manually in case the class doesn't have its native constructor. If it does have one, then the following code should be executed to call the constructor, depending on which parameters you need to pass:

```
My.NameSpace.ObjectClass.superclass.constructor.call(this, config)
```

or

```
My.NameSpace.ObjectClass.superclass.constructor.apply(this, arguments)
```

This is the way all the classes in WebApp should be created.

We'll use the Spreed plugin code as an example, with a snippet of the code of the class `SpreedParticipantBox`:

```
Ext.namespace('Zarafa.plugins.spreed.dialogs');

/**
 * @class Zarafa.plugins.spreed.dialogs.SpreedParticipantBox
 * @extends Zarafa.common.recipientfield.ui.RecipientBox
 * @xtype zarafa.spreedparticipantbox
 *
 * Extension to the  {@link Zarafa.common.recipientfield.ui.RecipientBox}.
 * This box offers adding moderator icon for the moderator participant.
 */
Zarafa.plugins.spreed.dialogs.SpreedParticipantBox = Ext.extend(
        Zarafa.common.recipientfield.ui.RecipientBox, {
```

```
        /**
         * @constructor
         * @param config Configuration object
         */
        constructor : function(config)
        {
                config = config || {};
                Ext.applyIf(config, { });
                Zarafa.plugins.spreed.dialogs.SpreedParticipantBox.superclass.
↪constructor.call(this, config);
        },

        // other functions
        }
);

Ext.reg('spreed.spreedparticipantbox',
Zarafa.plugins.spreed.dialogs.SpreedParticipantBox);
```

The `Ext.namespace` call on the first line ensures that there exists a JavaScript object called `Zarafa.plugins.spreed.dialogs` to act as an enclosing namespace object for our new class. The Ext JS `Ext.namespace` function declares a namespace. We place chunks of related functionality in namespaces and let the code tree reflect the namespace hierarchy.

After this come several lines of documentation: the class name, its parent class and xtype to register. It's good practice to always do this since the build system extracts this information to determine the inclusion order of JavaScript files.

The call to `Ext.extend` expresses that `Zarafa.plugins.spreed.dialogs.SpreedParticipantBox` extends (is a child class of) `Zarafa.common.recipientfield.ui.RecipientBox`. It is now possible to substitute an instance of the former for the latter. In most cases you can easily extend already existing classes to add some small functionality. If the class you're writing is not a child class of anything, simply extend it from Object.

When creating a common user interface class in the core, it should be registered using the `Ext.reg` function to allow the usage of xtype when creating the object. It is registered with the prefix `Zarafa.`[name] to prevent name clashes with Ext JS registered classes.

```
// At the bottom of the derived class, register it:
Ext.reg('spreed.spreedparticipantbox',Zarafa.plugins.spreed.dialogs.
↪SpreedParticipantBox);

// When using the derived editor in any class:
xtype : 'spreed.spreedparticipantbox'
```

New classes should always be created in a new file within the correct folder. Exceptions might be made if the new class is a helper class which only consists of a few lines of code and it is bound to the other class, defined in the same file, in such a way that moving it into a separate file is not logical.

3.2 Accessing Components

Often, when writing Panels which contain Components combined with the lazy instantation as discussed further, the problem arises that somewhere in the code of the container, a particular component must be accessed. Because the items array cannot be read (this only contains the lazy configuration objects and not the Component instantiations) other methods like `panel.findBy()`, `findById()` or `findByType()` must be used. Because these functions always search through all items, and all subitems (in case containers are embedded in the main panel), performance of these functions is quite low.

Ext JS offers an easy way to assign a component to a variable in the panel during rendering. If the *ref* option is

used in the component, the variable name can be specified of the component inside the parent panel. By using a path specifier it is even possible to assign the variable to the parent of the parent container. Consider the following example:

```
{
        xtype: 'panel',
        id: 'panel1',
        items: [{
                xtype: 'textarea',
                ref: 'myTextArea'
        }]
}
```

After rendering, the panel, with id `panel1`, will contain the field `myTextArea`, which will point to the given textarea. For specifying the path, consider the following example:

```
{
        xtype: 'panel',
        id: 'panel1',
        items: [{
                xtype: 'panel',
                id: 'panel2',
                items: [{
                        xtype: 'textarea',
                        ref: '../myTextArea'
                }]
        }]
}
```

By using the path seperator, the field `myTextArea` will now be assigned to `panel1`. Within the textarea itself, `panel1` can be accessed through the `refOwner` field.

Using the above objects, one can use the following code within the `textarea`:

```
reset : function()
{
        // The panel owns the record for which we display the data
        this.setText(this.refOwner.record.get('data'));
}
```

While in `panel1`, we can use the following code to access the `textarea` component:

```
update : function(record)
{
        // Update the textarea with the new record data
        this.myTextArea.update(record.get('data'));
}
```

This will work for any component, including components added in the `tbar` (top bar), `fbar` (footer bar) or `bbar` (bottom bar) fields. However, for these fields, additional path seperators are needed, since these are actually separate containers inside the main container.

3.3 Enums

Enumerations in Zarafa are extended from `Zarafa.core.Enum`. See the following example from the Spreed plugin:

```
Ext.namespace('Zarafa.plugins.spreed.data');

/**
 * @class Zarafa.plugins.spreed.data.DialogTypes
```

```
 * @extends Zarafa.core.Enum
 *
 * Enum containing the different types of dialogs needed to display spreed meeting.
 * @singleton
 */
Zarafa.plugins.spreed.data.DialogTypes = Zarafa.core.Enum.create({

        /**
         * The dialog with empty fields.(Brandly new)
         *
         * @property
         * @type Number
         */
        EMPTY  : 1,

        /**
         * The dialog with filled subject and participants.
         *
         * @property
         * @type Number
         */
        FILLED : 2,

        /**
         * The dialog with only participants field prefilled.
         *
         * @property
         * @type Number
         */
        PARTICIPANTS_FILLED : 3
});
```

So, this example contains, as usual, the namespace specification, then comments explaining what the purpose of this enumeration is. Then, the constructor `create` of the superclass, followed by the list of properties and their values.

You can also add extra items to an enumeration at a later stage, this might be especially helpful for plugins. This requires to extend from `Zarafa.core.data.RecordCustomObjectType`. It is connected with RecordFactory, so all custom records get a `BASE_TYPE` that is higher than 1000; so you can see it is a custom record. Thus, `BASE_TYPE` is equal to 1000, and the next one you add is then 1001, as the value of the previous highest value is incremented. See an example of its use in the class `FbEventRecord`:

```
Zarafa.core.data.RecordCustomObjectType.addProperty('ZARAFA_FACEBOOK_EVENT');
```

Now our Facebook records are of our custom type `ZARAFA_FACEBOOK_EVENT`.

3.4 Singletons

Static functions can be declared inside a *singleton* object. A singleton, in object oriented design, is a class that only has a single instance. This can be easily simulated in JavaScript by just creating one instance and re-using it. Consider the following example of the definition of a singleton, emulated in Javascript by a struct:

```
/**
 * @class Zarafa.core.XMLSerialisation
 * Functions used by Request for converting between XML and JavaScript objects
 * (JSON)
 * @singleton
 */
Zarafa.core.XMLSerialisation = {
```

```
        // snip
};
```

For singleton classes which extend an existing class, consider the following example:

```
/**
 * @class Ext.StoreMgr
 * @extends Ext.util.MixedCollection
 * The default global group of stores.
 * @singleton
 */
Ext.StoreMgr = Ext.extend(Ext.util.MixedCollection, {
        // snip
});

// Make it a singleton
Ext.StoreMgr = new Ext.StoreMgr();
```

There is now an instance of `Ext.StoreMgr` that is given the same name of the original class definition, effectively making it the only possible instance of it.

CHAPTER 4

Plugins

4.1 Ways of Extending WebApp

There are multiple ways to extend WebApp. Most common ways are via *plugins (including themes)* and *widgets*. With a plugin you can create new contexts, insert buttons and add functionality. See chapter *Insertion Points* for more information.

Contexts are implemented as plugins, and third-party developers can develop their own contexts to extend or customise the application. Three examples of contexts: the Mail, Calendar and Note contexts. Each context is a kind of special plugin that has additional functionality allowing it to *take over* the toolbar and main content section of the screen. Only a single context can be active at any given time. For example, folders in the folder hierarchy are linked to contexts that display their contents, so that when a user clicks his or her inbox, the Mail context is shown.

The main application screen consists of several areas. Some of these are independent shown in e.g. "Mail context". The UI provides a standard hierarchy panel showing a list of folders the user has access to, as well as a bottom tool bar. Each context has its own content panel and tool bar, but only the ones belonging to the currently active context are visible while all others are hidden. This is achieved by loading the toolbars and content panels of all contexts into their respective areas and using card layouts to switch between them.

Plugins can be of arbitrary complexity; they have the liberty to work with dialogs, stores and other extended Zarafa features. When you need a deeper integration with Zarafa, you will need to work with a plugin. Example plugins are the XMPP plugin and Facebook plugin.

Plugins can be shown visually as some UI components which trigger some work: buttons or context menu items to integrate events, etc.

Finally, *Widgets* appear only in the *Today* context and the side bar.

They can be be added or removed from there by the user. Each time the Today context is opened, the configured widgets are shown. Also, if you enable and lock the side bar, the widgets put in there are shown continuously. Widgets are small plugins, usually only visual, with very simple functionality. Example widgets: a simple visual shell game, a customized clock.

Now that we have seen what the possibilities are we will explain in the next parts the background and show how to implement a plugin that actually extends WebApp. We will use the plugins that were introduced in the introduction chapter.

This part is probably the most useful one to casual plugin and widget developers. We explain how to implement a widget, a dialog, and how to access globally available data. After going through this we are ready for the real

advanced topics.

Since WebApp 5.0.0 the widget context can also be disabled in WebApps config.php

Plugins should be installed in the */plugin* directory and are visible in 'settings -> plugins'. Plugin version are read from version tag in the *manifest.xml*

For an overview of plugins, which can also be used as examples see: https://stash.kopano.io/projects/KWA

4.2 What's new dialog

In KW-1241 we introduced a utilty to notify the user about new features of a specific version. Example code block:

Recommended is placing this above the *initPlugin* function.

```
whatsNew : {
        version: '0.3',
        features: [{
            title: 'Some plugin for WebApp',
            description: 'With this plugin version we introduced something awesome.
↪',
            image_url: 'img/title.jpg'
        }]
    },
```

CHAPTER 5

Insertion Points

The first thing plugin developers should really learn to start implementing new plugins are insertion points. An **insertion point** is a named location in the UI component hierarchy where plugins may add their own components. Insertion points are typically added to toolbars and context menus, and are intended to be easy hooks for adding new buttons, button groups or menuitems. The name of an insertion point is hierarchical, so most things specifically related to e-mail will start with `context.mail` and have a more precise indication of the location after that.

Note, that work with insertion points is implemented as a call to the container, which collects UI components from registered plugins and returns them. You can easily create your custom insertion points. See the listing below:

```
var toolbar = new Ext.Toolbar(
{
        items : [
                // Fixed items, always present
                {
                        iconCls: 'icon_new'
                },
                {
                        iconCls: 'icon_delete'
                },

                // Create insertion point 'example.toolbar'
                container.populateInsertionPoint('example.toolbar')
        ]
});
```

And now you have created a new insertion point, which can be used in future. By design, the names of the insertion points should reflect the structure of the application. Therefore, the proposed naming scheme follows a hierarchy separated by dots (`.`). More information and recommendations on the naming conventions are given in *Appendix A: Naming Conventions*.

In some cases, it's useful to have insertion points that provide extra information to the plugin. An example is an insertion point in a context menu, where it's useful to pass the menu object (`Ext.menu.Menu`) to the creation function. Yet another example is a toolbar in a *read mail* dialog, which might pass the entry ID of the item that is being shown to the plugin.

```
container.populateInsertionPoint('dialog.readmail.toolbar', mailEntryId);
```

A plugin is then able to register a function that uses these parameters:

```
createButton : function(insertionPoint, mailEntryId)
{
        return
        {
                xtype: 'button',
                text: 'Hello!',
                handler: function()
                {
                        alert('Message ID: ' + mailEntryId);
                }
        };
}
```

The returned component has an `xtype` field. This field is the unique identifier for that class, and is registered using `Ext.reg()` together with the corresponding constructor. However, in most cases, you need to use an already existing insertion point.

5.1 Enable insertion points

While we do not have a list of insertion points in the documentation, it is possible to get an overview of them by enabling insertion points. Doing so, you will be able to visually identify locations where WebApp can be extended: it highlights all existing insertion points, even those provided by third-party plugins.

Do this by navigating to *Settings -> Advanced (Should be enabled in config.php -> Developer tools -> Show insertion points in WebApp* Reload WebApp afterwards to view insertion points as buttons in WebApp.

5.2 Insertion points usage example

```
this.registerInsertionPoint('main.toolbar.actions', this.addCustomAction, this);

addCustomAction: function(insertionpoint) {
        return {
                xtype: 'button',
                tooltip: _('Custom Action'),
                iconCls: 'icon_customAction'
        }
}
```

In this case, the function `addCustomAction` is defined in your plug-in and returns an instance of `Ext.Button`, which is also suitable for inclusion in a tool bar like this. Other insertion points require different types of objects, please refer to the API documentation for the object type that the insertion point expects.

Thus, using the zdeveloper plug-in, you can quickly identify where you can add user interface elements to WebApp. Moreover, if you already know where to look, you can use it to get the exact name of the insertion point that you need. For all the rest, such as finding out the exact type of the object you should return when registering to an insertion point, you still need the API reference documentation.

5.3 Example: adding a button

The following code snippet shows an example with the Facebook events integration plugin:

```
Zarafa.plugins.facebook.FacebookEventsPlugin = Ext.extend(Zarafa.core.Plugin,
{
        /**
         * @constructor
```

```
         * @param {Object} config Configuration object
         *
         */
        constructor : function (config)
        {
                config = config || {};
                Zarafa.plugins.facebook.FacebookEventsPlugin.superclass.
↪constructor.call(this, config);
                this.init();
        },

        /**
         * Called after constructor.
         * Registers insertion point for facebook button.
         * @private
         */
        init : function()
        {
                this.registerInsertionPoint('navigation.south', this.
↪createFacebookButton, this);
        },

        /**
         * Creates the button by clicking on which the Facebook
         * will be imported to the Zarafa calendar.
         *
         * @return {Object} Configuration object for a {@link Ext.Button button}
         * @private
         */
        createFacebookButton:function()
        {
                var button=
                {
                        xtype             : 'button',
                        text              : _('Import Facebook events'),
                        iconCls           : 'icon_facebook_button',
                        navigationContext : container.getContextByName('calendar')
                }
                return button;
        });

        Zarafa.onReady(function()
        {
                if(container.getSettingsModel().get('zarafa/v1/plugins/facebook/
↪enable') === true)
                {
                        container.registerPlugin(new Zarafa.plugins.facebook.
                        FacebookEventsPlugin());
                }
        });
});
```

So, let's look closer on what is happening after the each function call. Just after calling the constructor we call *init()* function which registers insertion point in the navigation panel, south part.

```
init : function()
{
        this.registerInsertionPoint('navigation.south',this.createFacebookButton,
        this);
},
```

Here we initiate our plugin insertion point by *registerInsertionPoint* function. Which takes three parameters

*registerInsertionPoint(**match*, **func**, **scope**)*.

Where *match* is a string or regular expression naming the existing insertion point where new item will be added. Regular expression can be used like in the example below:

```
this.registerInsertionPoint(/context\..*?\.toolbar/, this.createButton, this);
```

func is a function that creates one or more Ext JS components at the specified insertion point and *scope* is an optional parameter of the scope in which the items will be created.

```
createFacebookButton:function()
{
        var button=
        {
                xtype             : 'button',
                text              : _('Import Facebook events'),
                iconCls           : 'icon_facebook_button',
                navigationContext : container.getContextByName('calendar')
        }
        return button;
}
```

Here comes the button (the returned component may be any other UI component (*Ext.Component* instances)) itself with specified xtype and text. We also specify the icon css style. And the last configuration property is *navigationContext* used here for our Facebook button to be displayed only when Calendar context is the active one.

After specifying all the necessary properties for items to insert we should register our plugin by calling the following function:

```
Zarafa.onReady(function() {
        if(container.getSettingsModel().get('zarafa/v1/plugins/facebook/enable')␣
↪=== true) {
                container.registerPlugin(new Zarafa.plugins.facebook.
                FacebookEventsPlugin());
        }
});
```

Zarafa.onReady function is called when all essentials have been loaded and Container is ready for work – so it is high time for plugins to start their work. *container.getSettingsModel* function is used to check in Settings if the plugin is enabled. More information about Settings can be found in the corresponding chapter *Settings model*. *container.registerPlugin* function registers the plugin itself for work.

CHAPTER 6

Themes

Themes are actually plugins that extend the *themePlugin (Zarafa.core.ThemePlugin)* Everything you want to do in a plugin can also be done in a theme.

For an example theme see: https://stash.kopano.io/projects/KWA/repos/themeexample/browse

Note: themes can be disabled in config.php since WebApp 5.0.0.

6.1 Setting the default theme

Default themes can be added in the config.php. If the login page is modified in a a theme, all users will see this by default. When a user has selected a different theme in settings, they will see this login page after login (When WebApp reads out the settings).

The theme directory name should be added in config.php, not the theme display name.

6.2 Basic themes

A common theme structure is:

```
themecompany
                js/ThemeCompany.js
                img/
                css/
                manifest.xml
                favicon.ico
```

When the theme is default or chosen by the user, the css is loaded automatically.

Example of manifest.xml

```
<?xml version="1.0"?>
<!DOCTYPE plugin SYSTEM "manifest.dtd">
<plugin version="2">
        <info>
                <version>1.0</version>
                <name>themecompany</name>
```

```
                <title>Company Theme</title>
                <author>You as new developer</author>
                <authorURL>http://www.yourwebsite.com</authorURL>
                <description>Anything you would like to have.</description>
        </info>
        <components>
                <component>
                        <files>
                                <client>
                                        <clientfile load="source">js/ThemeCompany.
↪js</clientfile>
                                        <clientfile load="debug">js/ThemeCompany.js
↪</clientfile>
                                        <clientfile load="release">js/ThemeCompany.
↪js</clientfile>
                                </client>
                        </files>
                </component>
        </components>
</plugin>
```

Example of ThemeCompany.js:

```
// Create the namespace that will be used for this plugin
Ext.namespace('Zarafa.plugins.themecompany');

/**
* A theme plugin should extend {@link Zarafa.core.ThemePlugin}. If it only changes␣
↪the css
* there is nothing to implement in this class.
* @class Zarafa.plugins.themecompany.ThemeCompany
* @extends Zarafa.core.ThemePlugin
*/
Zarafa.plugins.themecompany.ThemeCompany = Ext.extend(Zarafa.core.ThemePlugin, {});

// Register the plugin with the container after the WebApp has loaded.
Zarafa.onReady(function() {
        container.registerPlugin(new Zarafa.core.PluginMetaData({
                // To avoid problems the name of a plugin should be exactly the␣
↪same as the
                // the name of the directory it is located in.
                name : 'themecompany',
                // The displayName is what will be shown in the dropdown in which␣
↪the user can pick a theme
                displayName : _('Company Theme'),
                // Do not allow the user to disable this plugin
                allowUserDisable : false,
                // Do not show this plugin in the plugin list
                allowUserVisible : false,
                pluginConstructor : Zarafa.plugins.themecompany.ThemeCompany
        }));
});
```

6.3 JSON themes

JSON themes were added to make theming a bit easier. The downside is that JSON themes do not support Javascript changes. If you are in need of JS changes (think of new functionality or new buttons), we recommend the basic theme structure.

Related blog post: https://kopano.com/blog/new-json-themes-kopano-webapp/

A common JSON theme structure is:

```
themecompany
                theme.json
                resources/img/
                resources/css/
```

Example

```
{
        "display-name": "themecompany",
        "background-image": "resources/img/bg.jpg",
        "logo-large": "resources/img/logo_company_wide.svg",
        "spinner-image": "resources/img/spinner.svg",
        "primary-color": "#0033a0",
        "sprimary-color:hover": "#4d94ce",
        "selection-color": "#b9dcf5",
        "action-color": "#0033a0",
        "stylesheets": "resources/css/themecompany.css",
        "icons-primary-color": "#222222",
        "icons-secondary-color": "#0033a0"
}
```

CHAPTER 7

Widgets

As was mentioned in *Plugins*, widgets appear in the *Today* context or the side panel. The user can add or remove them at will. A user can also add multiple instances of the same widget but with different parameters.

Having multiple instances of the same widget is made possible because each widget's instance object has a unique identifier to keep multiple instances apart. As soon as you add a new widget to the *Today* context or the side panel, the widget receives a new identifier, a GUID, which will be used to store the widget's state. This newly added instance of the widget can maintain his own settings; its settings are stored in the settings tree in the folder *zarafa/v1/widgets/[GUID]*. When a widget is removed from the *Today* context or the side panel, this folder is deleted.

7.1 Creating a widget

We will look at the widget's architecture based on the *Facebook Widget* example. It shows the activity stream from a site that you can change in the widget's configuration.

To create a custom widget, you need to do two simple steps:

1. Create your own class that extends `Zarafa.core.ui.widget.Widget`;
2. Register this widget class in the container.

```
Zarafa.widgets.FBWidget = Ext.extend(Zarafa.core.ui.widget.Widget, {
        // The widget's code goes here
});

Zarafa.onReady(function() {
        container.registerWidget(Zarafa.widgets.FBWidget,
        'fb',
        _('Facebook'),
        'plugins/facebookwidget/resources/_static/facebook.png');
});
```

7.2 Widget configuration

To allow the widget to do something, you need to add the custom functionality inside the class. In our example, we save the URL of the site which activity we want to monitor in the widget's settings. To use them we need to

initialize parameter `hasConfig` in constructor and set it to `true`. For example, here is the part of the constructor that tells this to the widget framework:

```
constructor : function(config) {
        config = config || {};
        Ext.applyIf(config, {
                name: 'fb',
                height: 600,
                hasConfig : true
        });
}
```

After this, the "gear" icon will appear in the right top corner of the widget (see e.g. "Widget settings").

|Widget settings|

Widget settings

When you click on this icon, the `config` method of the widget will be called. In this method, we can setup the widget's settings dialog and show it.

```
/**
 * Called when a user clicks the config button on the widget panel.
 * Shows the window with url field - where the user need to put
 * new value for Facebook Activity Site.
 */
config : function()
{
        var configWindow = new Ext.Window({
                title  : _('Configure widget'),
                layout : 'fit',
                width  : 350,
                height : 120,
                items : [{
                        xtype : 'form',
                        frame : true,
                        ref : 'formPanel',
                        labelWidth : 180,
                        items : [{
                                xtype: 'textfield',
                                anchor: '100%',
                                fieldLabel: _('Site name to track the activity'),
                                allowBlank : false,
                                vtype: 'url',
                                ref : '../siteUrlField',
                                name: 'site_url',
                                value: this.get('site_url')
                        }],
                buttons : [{
                        text : _('Save'),
                        scope : this,
                        ref : '../../savebutton',
                        handler : this.saveUserUrlToSettings
                        },
                        {
                        text: _('Cancel'),
                        scope : this,
                        ref : '../../cancelbutton',
                        handler : this.closeConfigWindow
                        }
                        ]
                }]
        });
        configWindow.show(this);
```

```
},
```

To save or receive the settings value we use `this.set()` and `this.get()` methods respectively. In our example, we get the default value of the "site url field" using `this.get('site_url')`. If it was not stored before with `this.set('site_url')`, it will return `undefined`. You will need to take care of this by yourself. That is why we use `Ext.isEmpty` check the value in the constructor:

```
var siteUrl = this.get('site_url');
if( Ext.isEmpty( siteUrl ) ) {
        siteUrl = this.defaultFbActivitySite;
        this.set('site_url', siteUrl);
}
this.setTitle( _('Facebook') + ' - ' + siteUrl);
```

Don't forget to call parent constructor or the widget will not work; the parent class' constructor puts it in the proper location and takes care of loading the settings, and so on.

```
Zarafa.widgets.FBWidget.superclass.constructor.call(this, config);
```

7.3 Events

As the `Zarafa.core.ui.widget.Widget` class extends the `Ext.ux.Portlet` and finally `Ext.Panel`, you can override some helpful methods to gain more consistency and readability inside your widget. One of these methods is `initEvents`, it will be called after the panel is rendered. It is a useful place in the code to initialize the widget's events. But remember: each time you override the methods of the parent class, you need to call the superclass method explicitly.

```
initEvents : function()
{
        Zarafa.widgets.FBWidget.superclass.initEvents.apply(this, arguments);
        this.mon(this, 'bodyresize', this.reloadIframe, this);
},
```

Finally, another useful method is `onRender`, which is called after the component was rendered. You can use it to render your custom elements inside the widget. To check if the widget is visible in the current moment or not, you can use `isWidgetVisible` method. It will return `true` if the widget is visible and the `widgetPanel`, on which it resides, is not collapsed. When you click the "cross" (close) icon, then the `widgetPanel`, on which the widgets resides, will unregister the widget's GUID and it will destroy the widget. It will then fire the *destroy* event. Therefore, you can define an `onDestroy` method to do some tasks after widget is destroyed.

CHAPTER 8

Dialogs

Dialogs are used throughout the WebApp. Dialogs are an important part in the application. When you are creating an email, appointment, a contact or when opening the address book: the contents will always be shown in a dialog.

The `Zarafa.core.ui.ContentPanel` has a couple of generic subclasses that might help in managing MAPI objects (objects that contain information on e.g. e-mail, appointments, etc.). There are two basic types of dialogs in WebApp: `Zarafa.core.ui.RecordContentPanel` and `Zarafa.core.ui.ContentPanel`. Each Dialog must inherit from either one to benefit from any automatically added user interface elements and styling.

Usually, you would use `Zarafa.core.ui.ContentPanel`. If you want to handle MAPI records, then see sections *Dealing with MAPI records* and *Stores and RecordFactory* for more background information.

8.1 Events

For plugins, it is possible to detect when a `Dialog` is about to be displayed to the user, allowing it to hook into further events of the `Dialog` itself, or any of its components. To do this, the plugin must listen to the `createdialog` event from the `Zarafa.core.data.ContentPanelMgr` singleton object. This event will pass the dialog that is being displayed as argument. Note that the event is called before the dialog is rendered. The `DialogMgr` will inform the plugin that the dialog has disapeared using the `destroydialog` event.

8.2 Example

The following code snippet shows how we can create our own custom dialog. This is just the definition of the dialog and its contents, it does not yet make it available to WebApp yet.

```
Ext.namespace('Dialogs.dialogsexample.dialogs');

/**
 * @class Dialogs.dialogsexample.dialogs.SimpleDialog
 * @extends Zarafa.core.ui.ContentPanel
 *
 * The simple dialog which contains Ext.panel.
 * @xtype simpledialog
 */
Dialogs.dialogsexample.dialogs.SimpleDialog = Ext.extend(Zarafa.core.ui.
↪ContentPanel, {
```

```
        /**
         * @constructor
         * @param config Configuration structure
         */
        constructor : function(config)
        {
                config = config || {};
                Ext.applyIf(config, {
                        defaultTitle            : _('Simple Dialog'),
                        alwaysUseDefaultTitle   : true,
                        width                   : 340,
                        height                  : 200,
                        //Add panel
                        items                   : [
                        {
                                xtype : 'panel'
                        }
                        ]
                });

        //Call superclass constructor
        Dialogs.dialogsexample.dialogs.SimpleDialog.superclass.constructor.
↪call(this, config);
        }
});

// Register the dialog xtype
Zarafa.core.ui.ContentPanel.register(Dialogs.dialogsexample.dialogs.SimpleDialog,
↪'simpledialog');
```

You can see that to create a custom dialog, we need to extend `Zarafa.core.ui.ContentPanel` and call the superclass constructor inside the dialog constructor. Finally, the last step that should be done is to register this dialog with a custom xtype:

```
Zarafa.core.ui.ContentPanel.register(Dialogs.dialogsexample.dialogs.SimpleDialog,
↪'simpledialog');
```

And that's it! In the items list, you can put the content that you wish.

It can be any subclass of `Ext.Component`. Now we can use our dialog from WebApp, just run the following from the Javascript console while WebApp is loaded:

```
Dialogs.dialogsexample.dialogs.SimpleDialog.create({
        title : _('My Custom Dialog')
});
```

This piece of code will create an instance and show the dialog on screen.

8.3 Dealing with MAPI records

When working with `MAPIRecord` instances, it is useful to use the `Zarafa.core.ui.RecordDialog`, because it has better support for managing `MAPIRecords` and has extra functionality for saving the record. For `Messages` (e.g. Mail and Meeting Requests), we have the `Zarafa.core.ui.MessageDialog` which extends the `RecordDialog` functionality to support sending the message to all recipients.

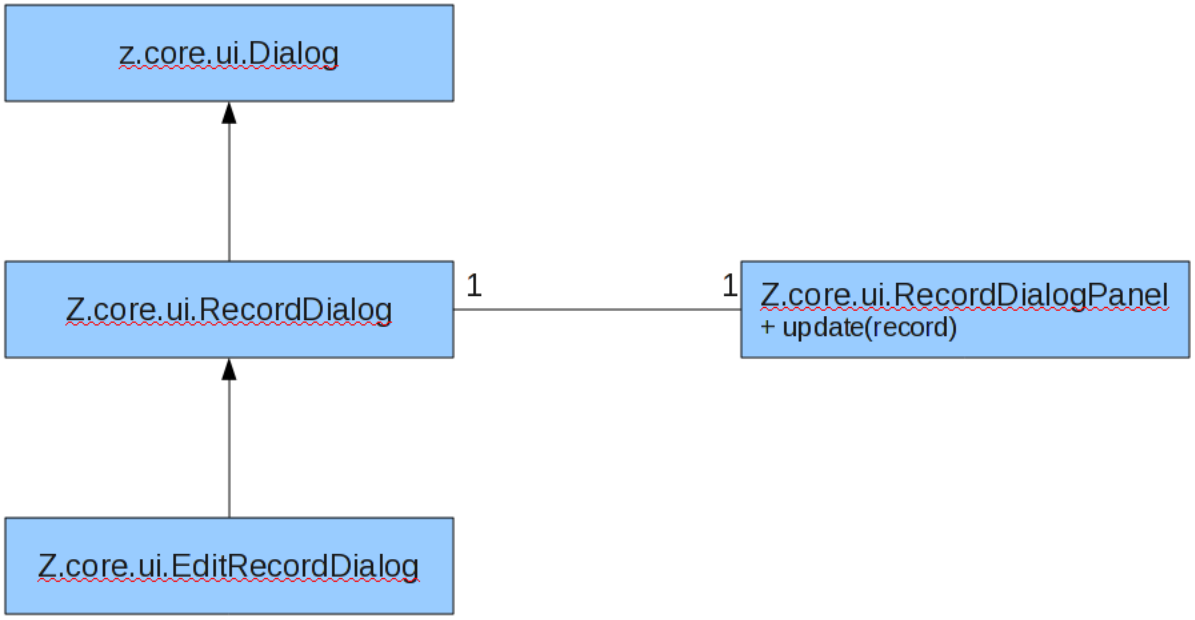
A `Dialog` that *displays* the contents of an `IPMRecord` must always inherit from `RecordDialog`. This dialog accepts the `record` configuration option which will be opened by default, otherwise it is possible to set the displayed `IPMRecord` through the `setRecord` function on this dialog. Using the `RecordDialog`, the dialog will automatically hook into the `IPMStoreManager` to listen for update events regarding the `IPMRecord`

which is opened by the dialog. Also, with the `setrecord` and `updaterecord` events, the dialog can inform all components within the dialog about record changes.

A `Dialog` that is used for *creating* or *editing* an `IPMRecord` must always inherit from `EditRecordDialog`. `EditRecordDialog` is a subclass of `RecordDialog`, and therefore offers the same features.

Additionally, the `EditRecordDialog` automatically places all edited `IPMRecords` into the `ShadowStore`. For further explanation of the shadow store and why it is relevant, see *IPM Stores and ShadowStore*.

See "RecordDialog UML diagram" for how they relate to each other.



RecordDialog UML diagram

8.3.1 Displaying a record

When adding a `Panel` to the `RecordDialog` (or `EditRecordDialog`), it is recommended to extend the `RecordDialogPanel`. This `Panel` will take the `RecordDialog` update features into account and automatically updates the `Panel` when the `RecordDialog` signals a change in the `IPMRecord` for this `Dialog`. Any subclass of `RecordDialogPanel` is required only to implement the function `update(record)` which will be called when the `Record` is updated.

The constructor of the subclass does not receive a `record` field; furthermore, the constructor must assure that any components which display particular fields of the `IPMRecord` are initialized to display a default value (i.e. `undefined` or an empty string). When the `Dialog` is rendered, the `update(record)` function will be used to set the `IPMRecord` for the first time.

Both the `RecordDialog` and `EditRecordDialog` offer default buttons for the `Toolbar` for saving and deleting the `IPMRecord` which is contained in the `Dialog`.

CHAPTER 9

Bidding System

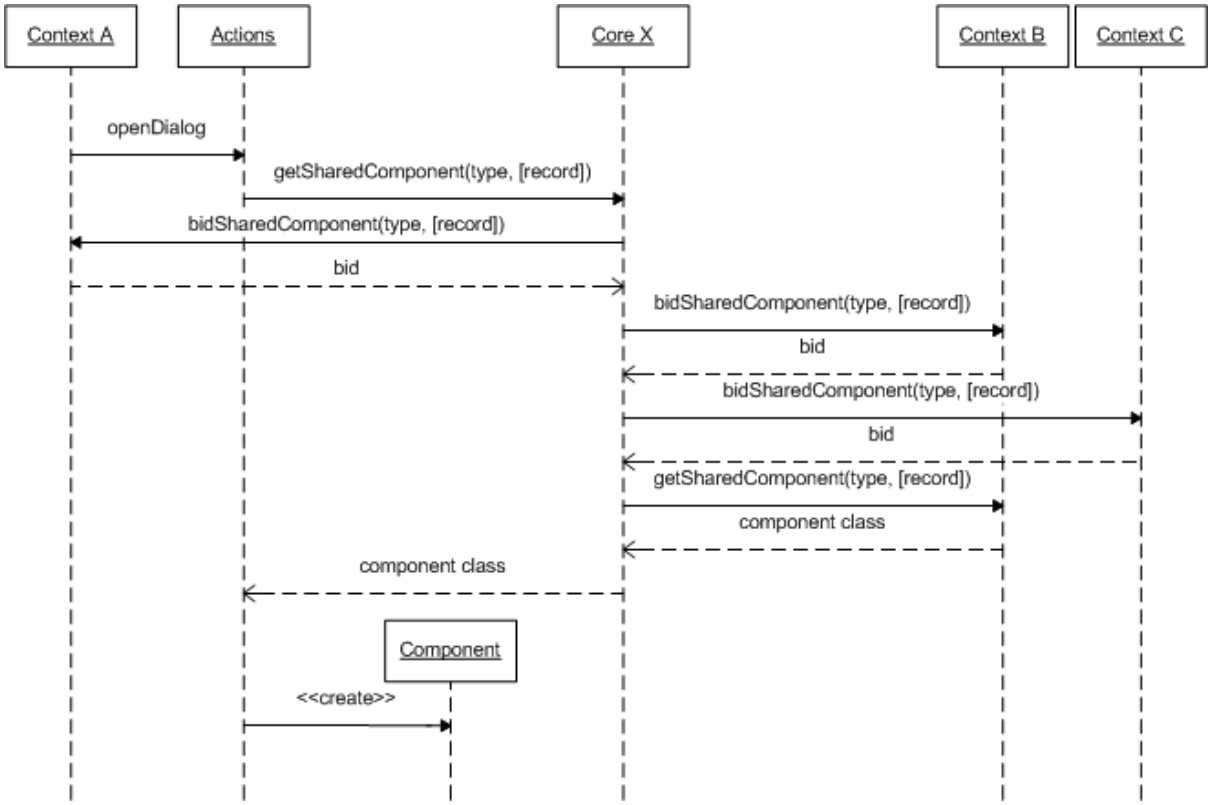
9.1 Bidding and insertion points

Insertion points, as described in *Insertion Points*, are perfect for extending WebApp in predefined places with a small additions. Next to this, another way of extending functionality is to *override* existing user interface components. The way WebApp decides which user interface class to use when presenting some kind of data is called the "bidding system": WebApp allows all components to bid on functionality. The way this works, roughly, is that WebApp will ask all registered plugins to place bids on something, and the highest bidder is chosen to deliver the functionality.

The shared component system is not meant to be used for each and every button, but for components of some magnitude. This is due to the time it takes to ask each component to place a bid, so for items that need to be shown often and quickly, the system is too slow. Therefore, if we would use the shared component system for these smaller components as well, it will have a negative effect on the performance. Dialogs and preview panels are examples of components that are suitable for the bidding system: it takes a lot of time to initialize them anyway, so the bidding rounds don't have a noticeable negative impact there. For smaller components, a more suitable approach is to use an insertion point: these are easily iterated over in rapid succession.

9.2 Working

The shared component bidding system is a mechanism to prevent tight coupling between plugins and allowing plugins to override existing dialogs. A plugin that requires a component like a dialog, context menu or a preview panel can ask the container to get it a specific type of the shared component. The container will start a bidding round across all the registered plugins, and each plugin will be able to bid and the highest bidder is asked to deliver this component. The system makes it possible to implement a dialog for reading mail by a mail context and the dialog for showing an appointment or a meeting request by the calendar context.



Shared Component bidding sequence

9.3 Bidding on a shared component

When the shared component is requested, a component type is also supplied. This can be used to bid more detailed based on the component type. The various component types are defined as valuess in the enumeration `Zarafa.core.data.SharedComponentType`. Other types can be added using the `addProperty` method of the enumeration. Next to the type of the component, a record can also be added as an argument.

The following snippet of code shows how to use the Shared Component mechanism:

```
var componentType = Zarafa.core.data.SharedComponentType[componentType];

var dialog = container.getSharedComponent(componentType, record);
if (dialog) {
        config.record = record;
        dialog.create(config);
}
```

The plugins that take part in the bidding round for the component can then base their bid on the data within the record. For example, the mail context can bid differently for records that have the `object_type` set to `MAPI_MESSAGE` and a `message_class` of `IPM.Note`, than for the records with a `message_class` of `IPM.Appointment`. On the other hand, the calendar context would bid higher for the latter one.

The hierarchy context would look for records that contain an `object_type` set to `MAPI_FOLDER` to do its bidding.

9.4 Example

The following snippet of code shows how the plugin can participate in the bidding round.

```
/**
 * Bid for the type of shared component
 * and the given record.
 * This will bid on a common.dialog.create or common.dialog.view for a
 * record with a message class set to IPM or IPM.Note.
 * @param {Zarafa.core.data.SharedComponentType} type Type of component a context
 * can bid for.
 * @param {Ext.data.Record} record Optionally passed record.
 * @return {Number} The bid for the shared component
 */
bidSharedComponent : function(type, record)
{
        var bid = -1;
        if (Ext.isArray(record)) {
                record = record[0];
        }
        if (record && record.store || record instanceof Zarafa.addressbook.
↪AddressBookRecord) {
                switch (type)
                {
                        case Zarafa.core.data.SharedComponentType['common.create']:
                        case Zarafa.core.data.SharedComponentType['common.view']:
                        case Zarafa.core.data.SharedComponentType['common.
↪contextmenu']:
                                if (record.store.customObjectType==Zarafa.core.
↪data.RecordCustomObjectType.ZARAFA_SPREED_PARTICIPANT)
                                    //|| record instanceof Zarafa.addressbook.
↪AddressBookRecord)
                                {
                                        bid = 2;
                                }
                                break;
                        case Zarafa.core.data.SharedComponentType['common.dialog.
↪attachments']:
                                if(record instanceof Zarafa.plugins.spreed.data.
↪SpreedRecord) {
                                        bid = 2;
                                }
                                break;
                }
        }
        return bid;
},
```

When the plugin is the highest bidder the function `getSharedComponent` will be called to actually deliver the class to be constructed as component.

```
/**
 * Will return the reference to the shared component.
 * Based on the type of component requested a component is returned.
 * @param {Zarafa.core.data.SharedComponentType} type Type of component a context
 * can bid for.
 * @param {Ext.data.Record} record Optionally passed record.
 * @return {Ext.Component} Component
 */
getSharedComponent : function(type, record)
{
        var component;
        switch (type)
        {
                case Zarafa.core.data.SharedComponentType['common.create']:
                case Zarafa.core.data.SharedComponentType['common.view']:
```

```
                        component = Zarafa.plugins.spreed.dialogs.
↪EditSpreedParticipantDialog;
                break;
                case Zarafa.core.data.SharedComponentType['common.contextmenu']:
                        component = Zarafa.plugins.spreed.dialogs.
↪SpreedParticipantContextMenu;
                break;
                case Zarafa.core.data.SharedComponentType['common.dialog.
↪attachments']:
                        component = Zarafa.plugins.spreed.dialogs.
↪AttachmentsDialog;
                break;
        }
        return component;
}
```

CHAPTER 10

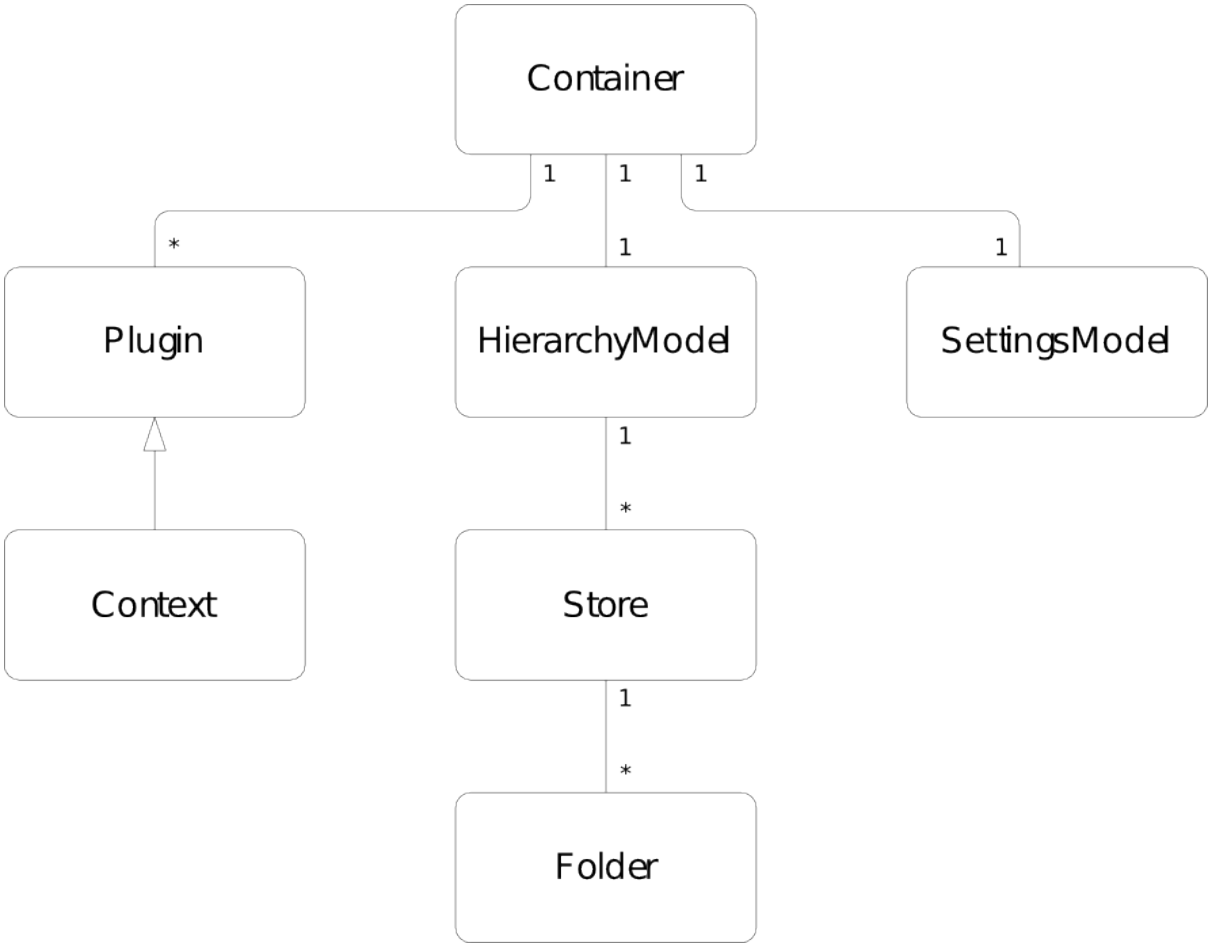
Data Models

10.1 Model-based architecture

A data model encapsulates data, providing a way for the interested parties to query, load and modify said information and get notified in case of changes. The WebApp framework contains a *global model* that provides an API for handling plugins and insertion points, the hierarchy model (store/folder hierarchy), and settings.

The Ext JS library provides a way of working with server-backed data stores and standard UI components. Most notably, the data grid component (i.e. data representing a list of mails, appointments, etc.) integrates with them. The framework contains base classes for creating such data stores that use the PHP code as backend.

The *global model* stores information required across contexts and plugins. A schematic overview is shown in "Global data model".



Global data model

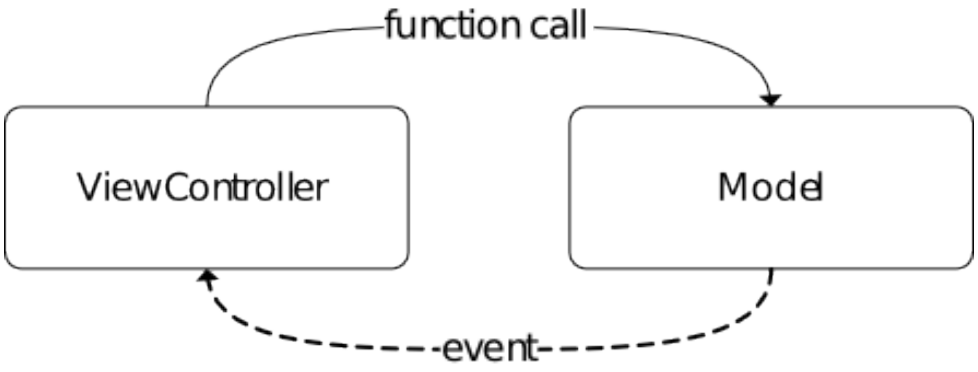
A global instance of the `Zarafa.core.Container` object, simply called `container`, maintains a list of the registered plugins, lazily-created instances of `Zarafa.core.HierarchyModel`, and `Zarafa.core.SettingsModel`.

The *hierarchy model* is a separate context, see *Hierarchy model*.

The *settings model* is also a separate context, see *Settings model*.

10.2 Separable model architecture

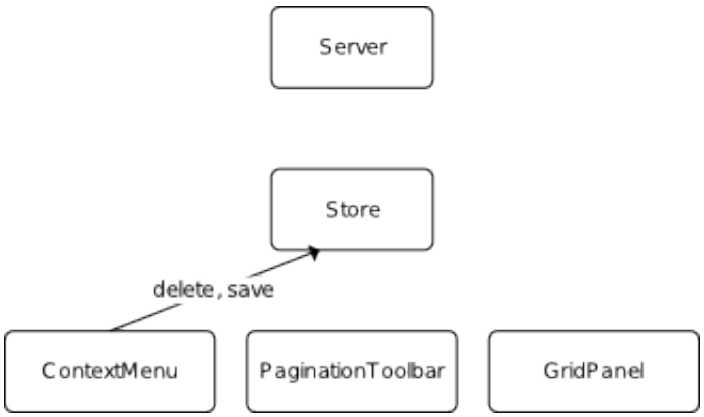
To promote code re-use and separation of concepts, a separable model architecture is used. In essence, this is just a MVC (Model-View-Controller) design with the view and controller collapsed into a single entity. The responsibility of the model is to store and validate data, and synchronize with the server. On the other side, the view/controller components present the data to the user, allow him to edit the data, and generally drive the model. The view/controller components contain a reference to one or more model objects. They interact with the models with method calls, and get notified of changes through events. This principle is shown in "Separable model architecture".



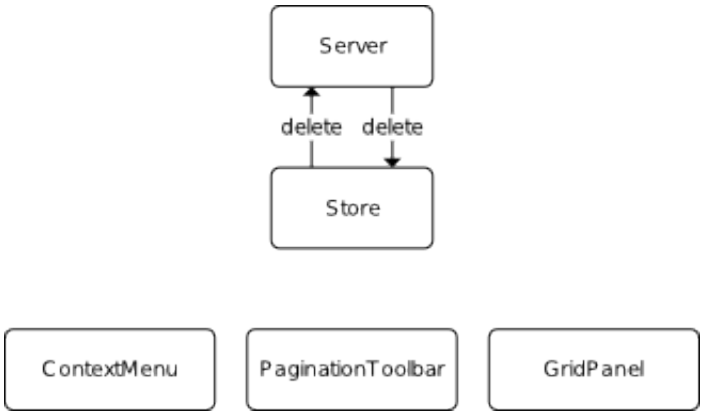
Separable model architecture

Since this design decouples data manipulation and client/server communication from the user interface, the model part of the application can be easily unit tested outside the browser.

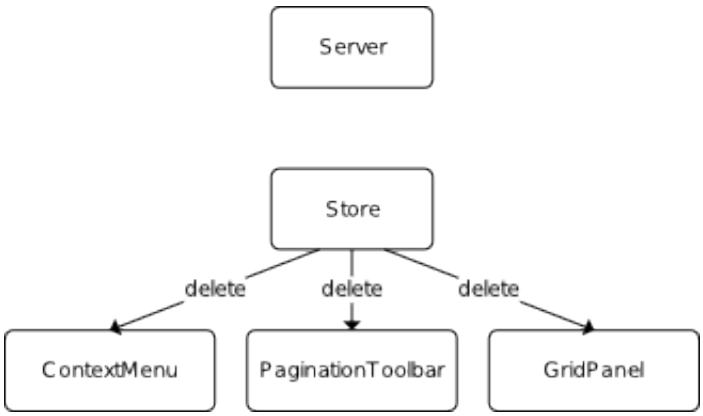
View/controller components never need communicating with each other, or even need to know of each others existence, as all data requests and modifications are done through the models. A concrete example of how models and view/controllers interact is shown in "Deleting a message item from a data store: the ContextMenu deletes item in the model" and on. Shown at the top is a store containing a set of message items, representing the model. Three UI components act as view/controllers, mutating and displaying information in the store.



Deleting a message item from a data store: the ContextMenu deletes item in the model



Deleting a message item from a data store: the model commits the change



Deleting a message item from a data store: the model fires a delete event on completion

In this example, if a user deletes an item, the `ContextMenu` object calls the `delete` method on the store, and then calls `save` ("Deleting a message item from a data store: the ContextMenu deletes item in the model") to tell the store to synchronise with the server ("Deleting a message item from a data store: the model commits the change"). When this (asynchronous) operation has finished, a `delete` event is fired to notify the three components that an item was removed ("Deleting a message item from a data store: the model fires a delete event on completion"). This in turn causes the grid and tool bar to update and reflect the change. Refer to *Communication* for more detailed information on how the change is communicated to the server.

10.3 Hierarchy model

The `HierarchyModel` is a model class containing a structured tree for all MAPI stores and MAPI folders in the application. It can be used to receive `entryid`, `store_entryid` or `parent_entryid` of any of its contained elements, but to receive the data from these stores, you should use context models.

Loading the hierarchy from the server can be done by calling the `load` method.

```
// Load the folder hierarchy.
container.getHierarchyModel().load();
```

This method triggers an asynchronous HTTP request. When the hierarchy has loaded (or when an error occurred) the model fires the `load` event.

```
// Hook the load event.
container.getHierarchyModel().on('load', function() {
        alert('Hierarchy loaded!');
});

// Load the folder hierarchy.
container.getHierarchyModel().load();
```

Once the hierarchy has been successfully loaded, it can be queried for stores, folders, default folders, etc:

```
//Get the calendar folder, used in the Facebook plugin
var calendarFolder = container.getHierarchyStore().getDefaultFolder('calendar');

// in convertToAppointmentRecord, we create a new record of appointment type
// so we need to set parent_entryid of this new record to entryid of the needed␣
↪folder
// and store_entryid of this new record to store_id, which is store id hierarchy
var calendarRecord = facebookEventRecord.
↪convertToAppointmentRecord(calendarFolder);
```

10.4 Settings model

The settings model is a tree of key/value pairs that is saved to the server. It is loaded once, when the application loads; however, writes are always committed immediately. All plugin developers should implement a settings module to enable and disable their plugins or widgets.

There is a separate settings page for plugins that becomes visible if you enable the insertion points (see *Insertion Points*). Here, you can let the user configure any options in your plugin, if the need arises to make something configurable. For widgets, the recommended approach is to create a `config` method (see *Widget configuration* for an explanation and example).

If you want your plugin to be enabled/disabled by default you may create a `config.php` file to set default values for your plugin settings.

Here is an example from the Facebook plugin's `config.php` file:

```
/** Disable the facebook plugin for all clients */
define('PLUGIN_FACEBOOK_USER_DEFAULT_ENABLE', false);
```

`PLUGIN_FACEBOOK_ENABLE` is used in the `injectPluginSettings` function in `plugin.facebook.php`; here are the contents of that file:

```
/**
 * Facebook Plugin
 * Integrates Facebook events in to the Zarafa calendar
 */
class Pluginfacebook extends Plugin {

        /**
         * Constructor
         */
        function Pluginfacebook() {}

        /**
         * Function initializes the Plugin and registers all hooks
         * @return void
         */
        function init() {
                $this->registerHook('server.core.settings.init.before');
        }

        /**
         * Function is executed when a hook is triggered by the PluginManager
         * @param string $eventID the id of the triggered hook
         * @param mixed $data object(s) related to the hook
         * @return void
         */
        function execute($eventID, &$data) {
                switch($eventID) {
                        case 'server.core.settings.init.before' :
                                $this->injectPluginSettings($data);
                        break;
                }
        }

        /**
         * Called when the core Settings class is initialized and ready to accept
         * the sysadmin's default settings. Registers the sysadmin defaults
         * for the Facebook plugin.
         * @param Array $data Reference to the data of the triggered hook
         */
        function injectPluginSettings(&$data) {
                $data['settingsObj']->addSysAdminDefaults(Array(
```

```
                        'zarafa' => Array(
                                'v1' => Array(
                                        'plugins' => Array(
                                                'facebook' => Array(
                                                        'enable' => PLUGIN_
↪FACEBOOK_ENABLE,
                                                )
                                        )
                                )
                        )
                ));
        }
}
```

An example from the Spreed plugin is the default meeting duration. When creating the time panel for the "Setup Spreed meeting" dialog, we set predefined values for the Spreed meeting duration. The duration is retrieved by getting the default value from calendar default appointment period. See "Settings of calendar".

|Settings of calendar|

Settings of calendar

The implementation is shown here:

```
var duration = container.getSettingsModel().get('zarafa/v1/contexts/calendar/
↪default_appointment_period');
```

To add your plugin settings to the Zarafa settings model, the `injectPluginSettings` function should be implemented on the server-side:

```
/**
 * Called when the core Settings class is initialized and ready to accept
 * the sysadmin's default settings. Registers the sysadmin defaults
 * for the Spreed plugin.
 * @param Array $data Reference to the data of the triggered hook
 */
function injectPluginSettings(&$data) {
        $data['settingsObj']->addSysAdminDefaults(Array(
                'zarafa' => Array(
                        'v1' => Array(
                                'plugins' => Array(
                                        'spreed' => Array(
                                                'enable' => PLUGIN_SPREED_USER_
↪DEFAULT_ENABLE,
                                                'default_timezone' => date_default_
↪timezone_get(),
                                        )
                                )
                        )
                )
        ));
}
```

You can add any other settings as you wish. For example, again in the Spreed plugin:

```
'spreed' => Array(
        'enable' => PLUGIN_SPREED_ENABLE,
        'default_timezone' => PLUGIN_SPREED_DEFAULT_TIMEZONE,
        'spreed_setup' => Array(
                'width' => 940,
                'height' => 480,
        )
)
```

Next to *enable*, there are settings for the default time zone (*default_timezone*) and the size of the dialog. The default time zone is taken from the value that is defined in `config.php` (in this case, the WebApp global one, not the one distributed by the plugin):

```
/** Default timezone of the spreed meeting  */
define('PLUGIN_SPREED_DEFAULT_TIMEZONE', 'Europe/Amsterdam');
```

Obviously, if you have something to configure on the server side, the names can differ but the functionality should be the same.

In the end, the result looks something like in "Plugin settings".

|Plugin settings|

Plugin settings

Finally, the settings model also supports simple `get` and `set` functions for getting and setting values. For quick access to your configuration values, the path to a value is delimited with the forward slash character (*/*).

```
// Read flag to 2 seconds.
container.getSettingsModel().set('zarafa/v1/contexts/mail/readflag_time', 2);
```

If you have survived until here and have written a plugin that does something with the usual interfaces, but are hungry for more, then the upcoming chapters are where you will find all the nitty-gritty details of WebApp.

We dive into MAPI, the communication protocol between WebApp and the server. This will require an understanding of how Zarafa works and how data is stored in MAPI. Finally, a few subjects such as the ant-based build system and server-side translations are handled.

CHAPTER 11

MAPI

Zarafa provides its groupware functionality by connecting the Linux-based server with clients using MAPI.

Messaging Application Programming Interface (MAPI) is a messaging architecture and a Component Object Model based API for Microsoft Windows. Simple MAPI is a subset of 12 functions which enable developers to add basic messaging functionality. Extended MAPI allows complete control over the messaging system on the client computer, creation and management of messages, management of the client mailbox, service providers, and so forth. Simple MAPI ships with Microsoft Windows as a part of Outlook Express/Windows Mail while the full Extended MAPI ships with Office Outlook and Exchange.

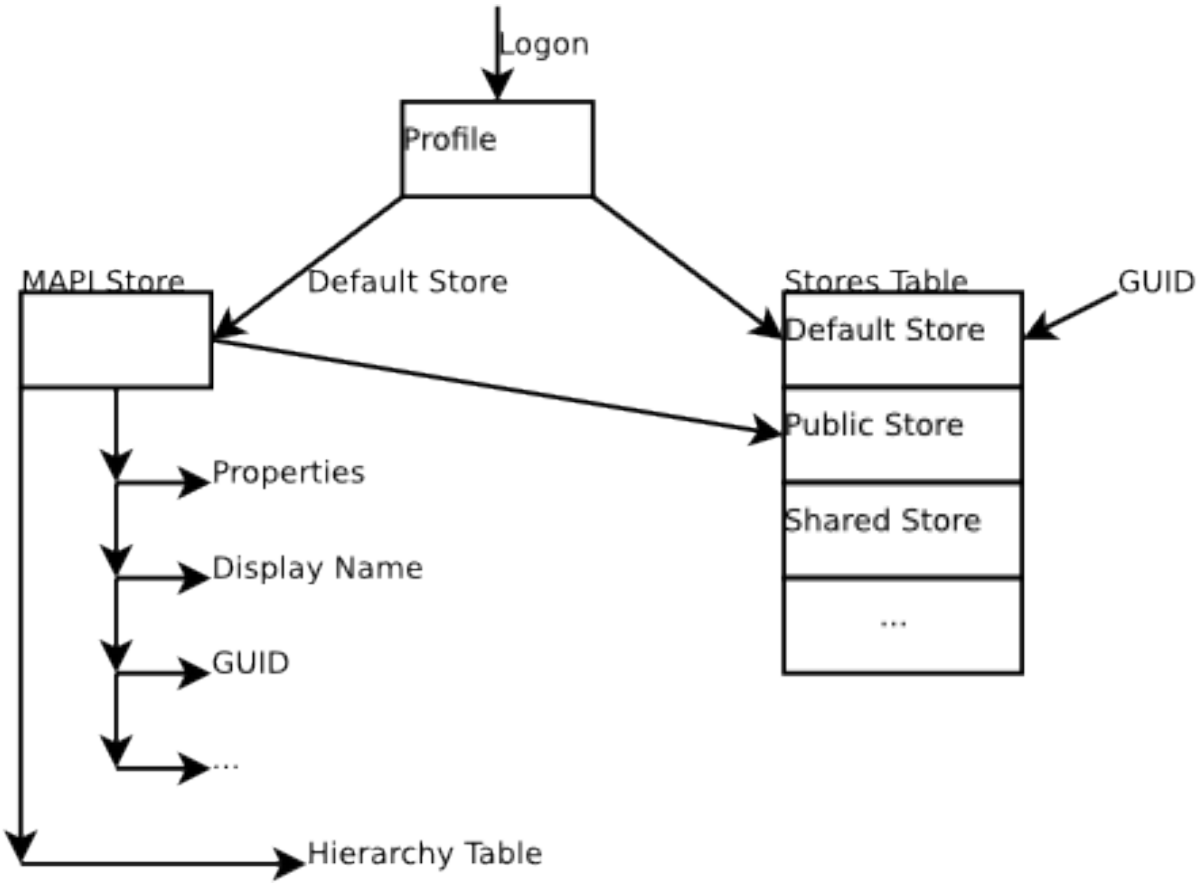
For more information about MAPI concepts refer to corresponding MSDN article.

The data model provides an API for working with client-side, and server-side data. A user has access to a set of *MAPI stores*, which contain *MAPI folders*, which in turn contain *MAPI messages*. A simplified view of the MAPI model is shown in "MAPI data model".

MAPI data model

MAPI (Messaging Application Programming Interface) Store is actually the base one, which can be extended with plugins. See "MAPI store table".

An important note here is that *Zarafa.core.data.MAPIStore* is not a *MAPI Store* as it is, but an *Ext.data.Store* used to collect *MAPI Messages*.



MAPI store table

One of the main MAPI concepts is a *folder*. Folders are MAPI objects that serve as the basic unit of organization for messages. Arranged hierarchically, folders can contain messages and other folders. Folders make it easier to locate and work with messages.

MAPI store uses such parameters to build a hierarchy as `entryid`, `parent_entryid` and `store_entryid` `entryid` is an id of current object in MAPI structure - for example, of the store with calendar appointments. `parent_entryid` is used to unite this object with hierarchy - so, for a store with calendar appointments, it would be an `entryid` of a folder with calendar store. `store_entryid` is a value that represent an `entryid` of current item's store. For example, for a store with calendar appointments it would be an `entryid` of IPFStore - the main store where all stores belong.

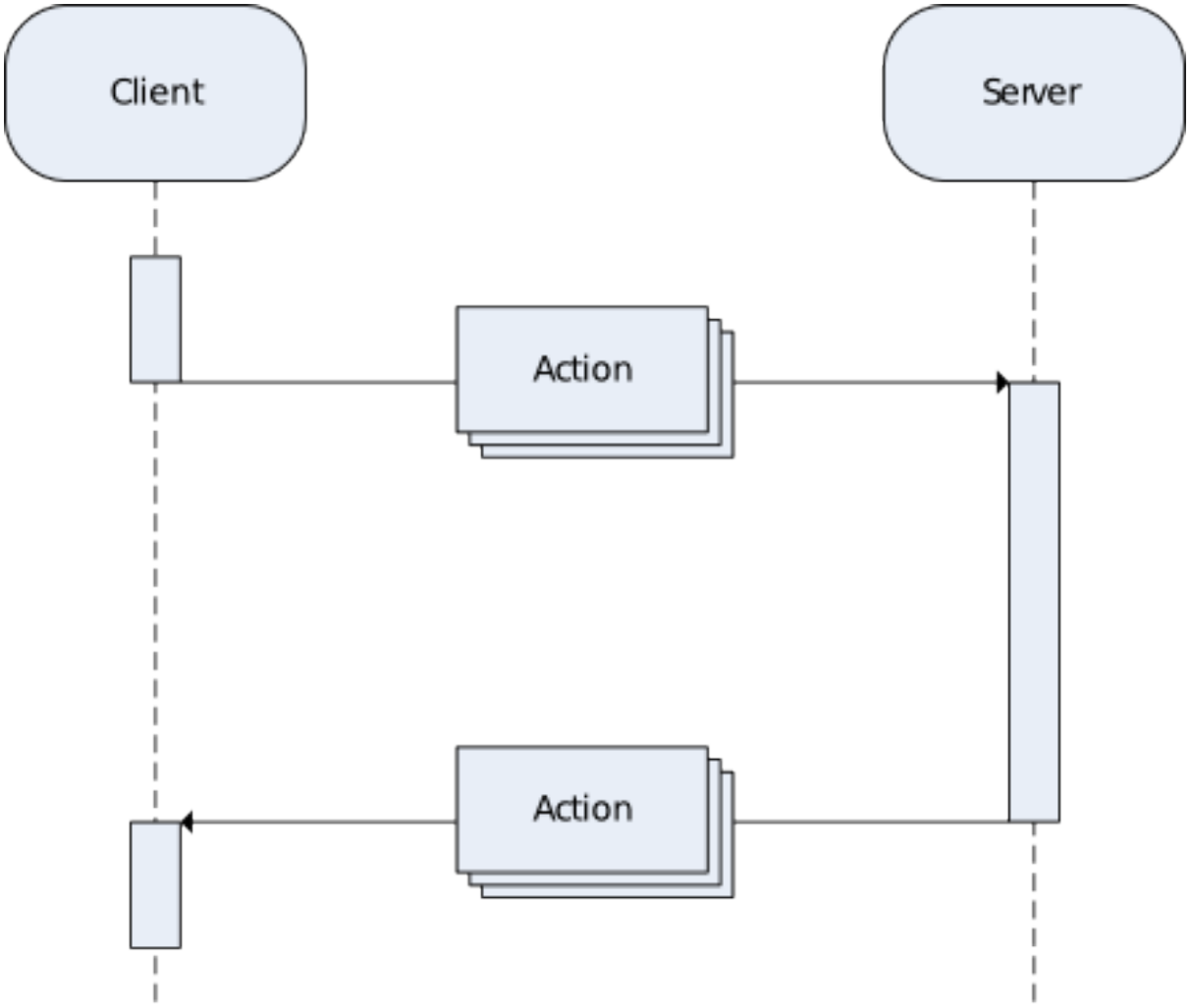
A MAPI folder can be seen as a flat list of items, much like a table in a database. Instead of directly issuing, action requests from the client to list or mutate items, a high-level API is provided that exposes a MAPI folder as an Ext JS store (Ext.data.Store).

CHAPTER 12

Communication

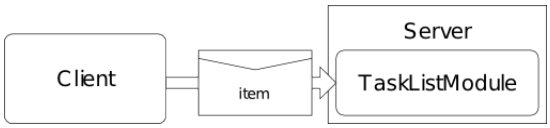
12.1 Conceptual Overview

The back-end has a pluggable architecture, with functionality divided among a set of *modules*. Each module is a named entity, such as `maillistmodule` or `taskitemmodule` (or for Spreed plugin - `class.spreedmodule`), and exposes a specific part of the Zarafa server functionality. The client communicates with a module by sending it one or more *actions*. Each request may contain one or more actions for one or more modules. Several modules may implement the same actions (such as `list`) so actions are grouped by target module. The server processes each of the actions in a sequence and formulates a response, also containing actions. The process is shown in "Message exchange".

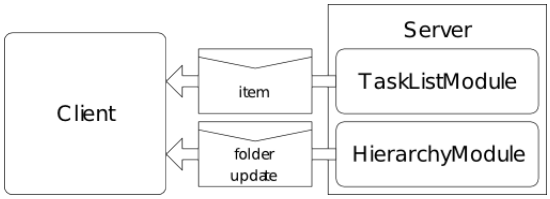


Message exchange

Although there is usually a one-to-one mapping between request and response actions, this does not neccesarily have to hold. As an example consider the situation in "Message exchange for creating a new item: client request" and "Message exchange for creating a new item: server response". In this exchange the client wants to create a new task and sends `save` action to the `tasklistmodule` on the server, as shown in "Message exchange for creating a new item: client request". If successful, the module on the server-side responds with an `item` action with information about the created task, such as the generated entry ID, but it will also send a `folderupdate` action containing updated information about the folder the item was created in. This is shown in "Message exchange for creating a new item: server response". This last action is to notify the client that there is now one more item in the folder.

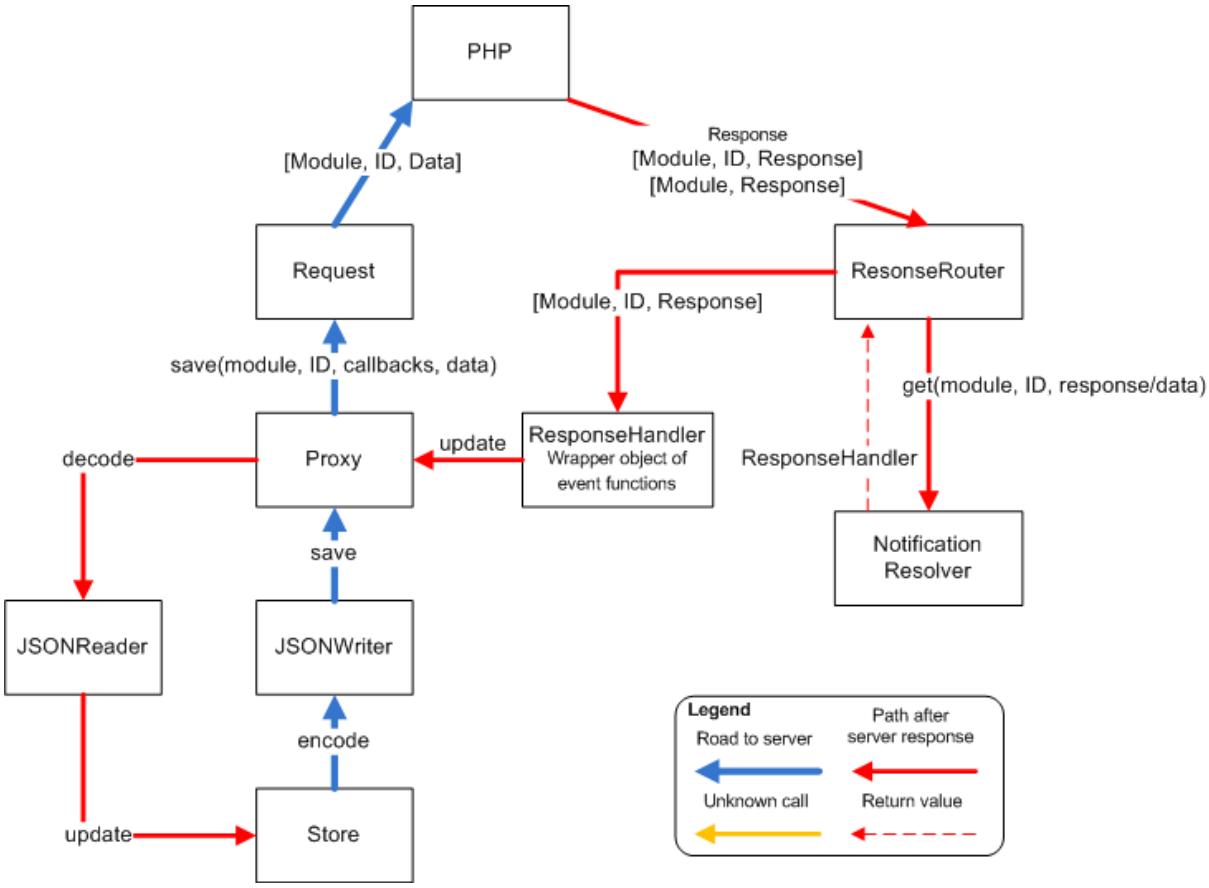


Message exchange for creating a new item: client request



Message exchange for creating a new item: server response

The exact data flow through the various classes is shown in "Request-Response flow". The important components inside this diagram will be explained in details further down in this chapter.



Request-Response flow

12.2 Protocol

Communication with the back-end starts with the client initiating a request. The data exchange format is JSON (JavaScript Object Notation). As explained in the previous section, a request may contain multiple actions for multiple modules.

The following listing shows a minimal client request for listing mails in a folder. First object is `zarafa` to define the correct scope for our request. The objects beneath there are different modules which are being accessed. Beneath each module can be one or more identifiers. Each request receives its own unique identifier. This is later used to match a response to the corresponding request. Beneath the identifier object the `action` tag is found which may contain a forest of key-value pairs. The server will respond with a similarly constructed reply.

```
{
        zarafa : {
                maillistmodule : {
                        maillistmodule1 : {
                                list : {
                                        store : [store MAPI ID]
                                        entryid : [folder MAPI ID]
                                }
                        }
                }
        }
}
```

A possible response from the server is shown below.

```
zarafa : {
        previewmailitemmodule : {
                previewmailitemmodule1 : {
                        item : {
                                [mail item details would be included here]
                        }
                }
        },
        mailnotificationmodule : {
                mailnotificationmodule1 : {
                        newitem : {
                                [mail item details would be included here]
                        }
                }
        }
}
```

12.3 Javascript Communication

To avoid having to construct HTTP requests manually, all client requests are made through a global instance of the `Zarafa.core.Request` class. This object provides JavaScript <-> JSON (de)serialisation, and allows other components to react to server actions via either events or callbacks.

Consider the code example listed below. A call is made to the request object (a global instance of which can be obtained from the container using `getRequest()`) to retrieve a single message. The first two arguments are *module name* and *action type*, which are taken from enumeration objects `Zarafa.core.ModuleNames` and `Zarafa.core.Actions` respectively (note that the unique request identification as mentioned earlier is auto-generated by the `Zarafa.core.Request` class). The third argument is a Javascript object containing the parameters to the action, which in this case are the message's MAPI store Id and MAPI entry Id.

```
container.getRequest().singleRequest(
        'pluginNAMEmodule', // e.g.'REMINDER'
        'actionToPerform', // e.g. 'reminderlistmodule'
        {
                //data inclufed in action tag
                actions: actions
        }
)
```

12.4 Response Handlers

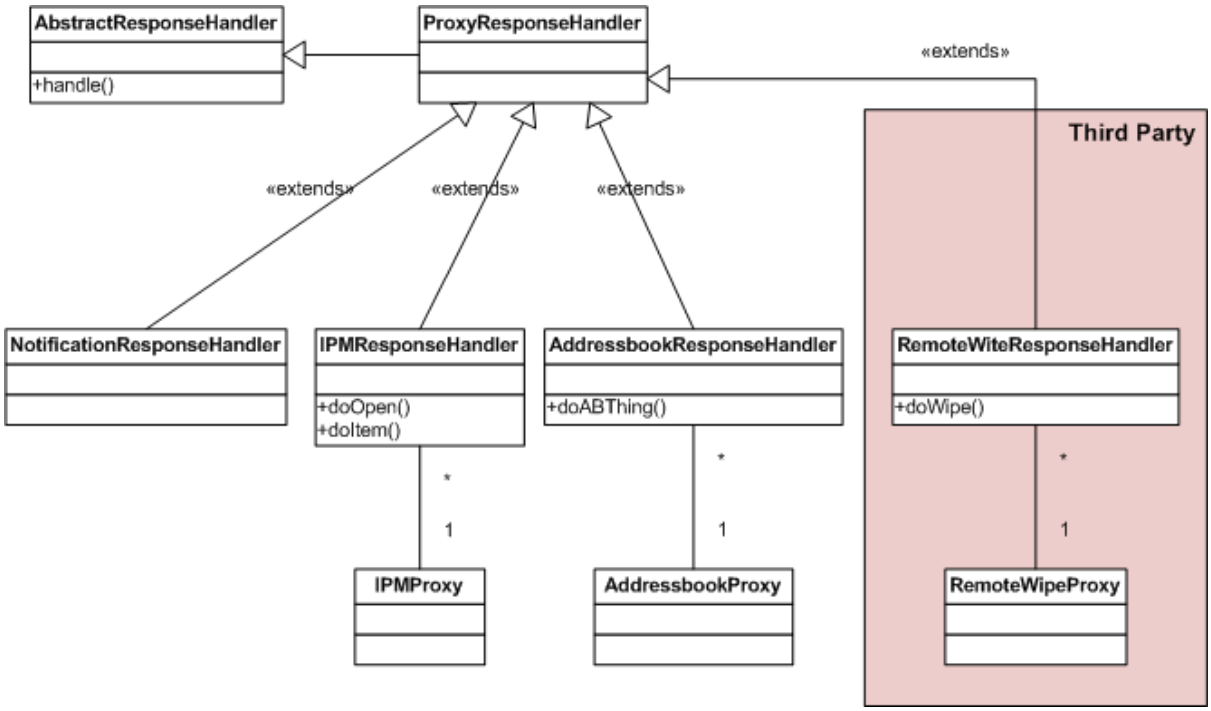
As described in the previous section a client should be able to handle the response data which is being send back from the server. For this we use `Zarafa.core.ResponseRouter` in combination with `Zarafa.core.data.AbstractResponseHandler` objects. Whenever the `Zarafa.core.Request` has received a response from the server, it will send this response to `Zarafa.core.ResponseRouter` which can then send out some events, and process the response. To notify the requestee of the initial request (for which the response was received), a `Zarafa.core.data.AbstractResponseHandler` is used. As described in the previous section a request can be created using the `singleRequest` function, this function has however a fourth argument; The `Zarafa.core.data.AbstractResponseHandler` object.

```
container.getRequest().singleRequest(
        Zarafa.core.ModuleNames.getListName('moduleName'),
        'actionToPerform',
        {
                'setting' : parameters
```

```
        },
        new Zarafa.core.data.ProxyResponseHandler({
                proxy: this,
                action: Ext.data.Api.actions['update'],
                options: parameters
        })
);
```

When a Request is made with the help of `ResponseHandler` object, the `Zarafa.core.Request` object will first generate a new Request idenfitication number, and then register the ID together with the `ResponseHandler` to the `Zarafa.core.ResponseRouter`. After receiving a new response the Router will search for all ID's and use the registered `ResponseHandler` for processing this response ("Different class diagrams").

The `ResponseHandler` which is used in the above example is a basic implementation which provides the basic communication with the `Ext.data.DataProxy` which made the request. Its primary task is checking for errors and firing the appropriate event in the `Ext.data.DataProxy` when a request has failed. See the class Diagram below for the various Response handlers:



Different class diagrams

12.5 Notifications

In some situations the user might want to be notified of particular events, for example Errors, Warnings, New mail, etc. There are various methods to display such a message to the user. To streamline the interface, the `Zarafa.core.ui.notifier.Notifier` class is provided, which is accessible (as singleton object) through the `Container` class using:

```
container.getNotifier()
```

The `Zarafa.core.ui.notifier.Notifier` class holds references to various `Zarafa.core.ui.notifier.NotifyPlugin` plugins. Each plugin provides a single method for displaying the message to the user. For example, the `ConsolePlugin` will send all messages to the browser console, the `MessageBoxPlugin` will show an `Ext.MessageBox` for each message. Creating a new plugin only requires that `Zarafa.core.ui.notifier.NotifyPlugin` is extended (which only requires the `notify`

function to be overridden). The plugin itself must then be registered with a unique name. For example:

```
container.getNotifier().registerPlugin('console', new Zarafa.core.ui.notifier.
↪ConsolePlugin());
```

When using the Notifier class to display a message to the user, one simply has to call the `notify` function on the Notifier:

```
container.getNotifier().notify('error', 'Panic', 'This is a very serious panic');
```

The first argument is the `category`. This category should start with *error*, *warning*, *info* or *debug*, but subtypes are also possible (e.g *error.json*, *info.newmail*); Based on the `category` the Notifier class will determine which plugin must be loaded to display the message. This decision is fully configurable through the Settings. When a notification arrives for a certain category, the settings which belong to that category will be requested. For the category *error* the setting:

```
'zarafa/v1/main/notifier/error'
```

for the *warning* category, the setting will be:

```
'zarafa/v1/main/notifier/warning'
```

When using subtypes, the . will be converted into a /, thus a category named *info.newmail* will result in a request to the setting:

```
'zarafa/v1/main/notifier/info/newmail'
```

The value of this setting, is the unique name of the NotifyPlugin which is registered. So when newmail notifications should be send to the ConsolePlugin, then the setting will be:

```
'zarafa/v1/main/notifier/info/newmail/value' = 'console'
```

When a subtype is not found in the Settings, for example *error.json* is requested, but the setting *zarafa/v1/main/notifier/error/json* does not exist, then the system will automatically fallback to *zarafa/v1/main/notifier/error*, when that also is not defined, then the final fallback will be *zarafa/v1/main/notifier/default*.

Using this system with settings allows 3rd-party developers to easily develop new notification systems, while the user is capable of configuring the notifications exactly as he wants them.

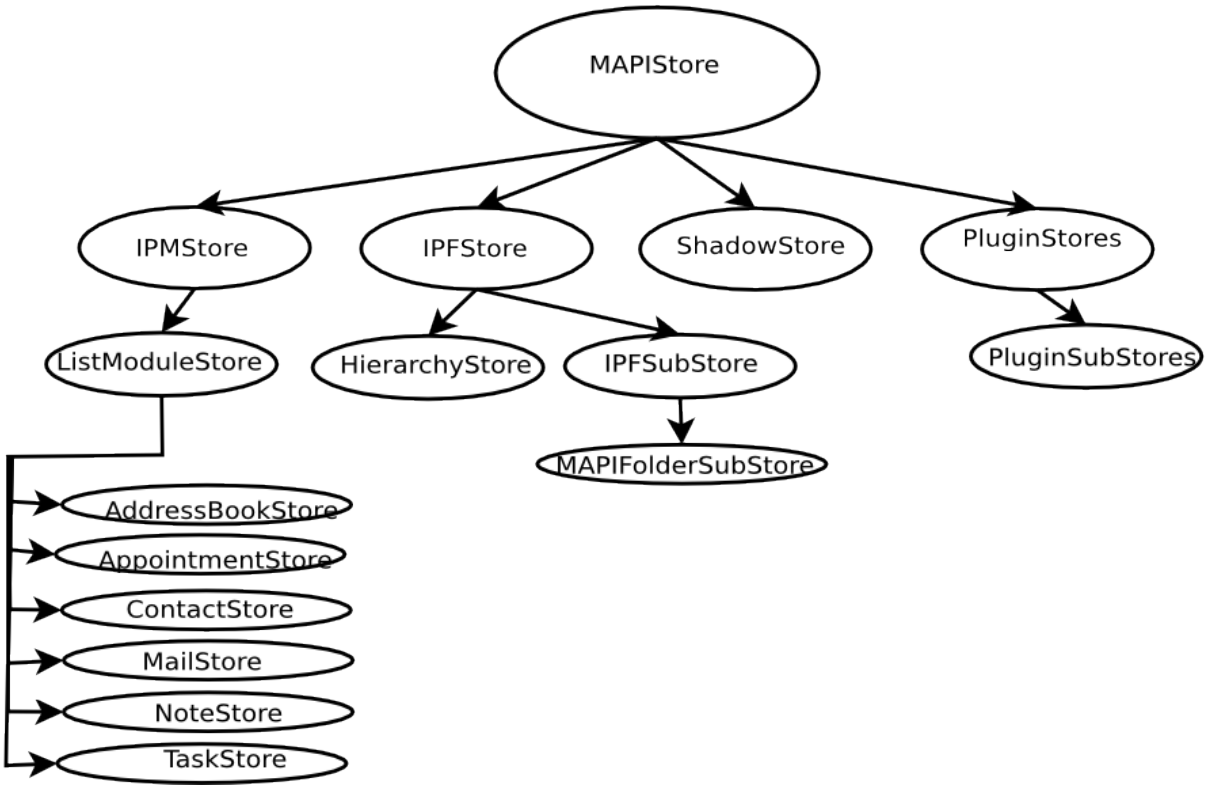
Notifications cannot be processed by the ResponseHandler provided by the requestee (that response handler is dedicated to handling the response for the request),that's why an alternative route is required. When a response is received for which no ResponseHandler has been registered, the ResponseRouter will give the response data to the `Zarafa.core.data.NotificationResolver`. This NotificationResolver will analyze the response, and return a ResponseHandler which is suitable for handling this response (Figure fig:rspnsrtr[]). With this ResponseHandler, the ResponseRouter can then continue its normal procedure to deserialize objects, handling the actions, and updating `Ext.data.Store` objects.

CHAPTER 13

Stores and RecordFactory

13.1 Stores and Substores

When you need to save and keep some information you should use stores. “Store types” shows different types of stores at their hierarchy.



Store types

13.1.1 MAPI Store

Zarafa MAPi Store is an extension of the Ext JS store which adds support for the *open* command, which is used by MAPI to request additional data for a record.

An important note here is that *Zarafa.core.data.MAPIStore* is not a store itself, but is used to collect *MAPI Messages*.

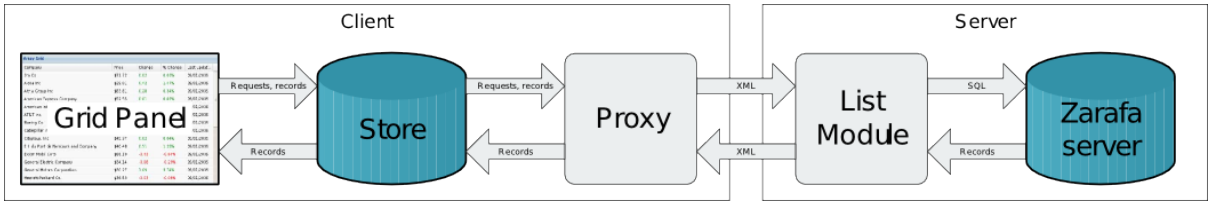
For example,

```
Ext.namespace('Zarafa.plugins.facebook.data');

/**
 * @class Zarafa.plugins.facebook.data.FbEventStore
 * @extends Zarafa.core.data.MAPIStore
 *
 * This class extends MAPIStore to configure the
 * proxy and reader in custom way.
 * Instead of defining the records dynamically, reader will
 * create {@link Zarafa.plugins.facebook.data.fbEventRecord} instance.
 *
 */
Zarafa.plugins.facebook.data.FbEventStore = Ext.extend(Zarafa.core.data.MAPIStore,
{
        /**
         * @constructor
         * @param {Object} config Configuration object
         *
         */
        constructor : function(config)
        {
                config = config || {};

                Ext.applyIf(config, {
                        reader : new Zarafa.plugins.facebook.data.
↪FbEventJSONReader({
                        id : 'id',
                        idProperty : 'id',
                        dynamicRecord : false
                }),
                        writer : new Zarafa.core.data.JsonWriter(),
                        proxy  : new Zarafa.core.data.IPMProxy()
                });

                Zarafa.plugins.facebook.data.FbEventStore.superclass.constructor.
↪call(this, config);
        }
});

Ext.reg('facebook.fbeventstore', Zarafa.plugins.facebook.data.FbEventStore);
```

This is a code example of Facebook plugin. We use the MAPI interface for compatibility with Microsoft Outlook. As far as MAPI provides full control over the messaging system, it is convenient to use it in Kopano WebApp.

A store is a client-side cache of items that exist on the server, and provides a clean interface for loading and CRUD (create, read, update and delete) operations. Many standard Ext JS components use stores to manage the data they display and operate on. In MVC (Model–View–Controller) terms, a store is the default model for many UI components. A common example is the grid panel (Ext.grid.GridPanel).

Displaying a list of tasks in a tasks folder is a matter of constructing a store instance, connecting it to the grid, and calling the load method with a Folder object as a parameter. The grid will automatically issue a generic load command to the store to populate it with data which is then displayed.



Data flow

"Data flow" shows how the various components connect to get data from the server to display in a data grid in the browser. A grid panel is connected to a store, which acts as a local cache containing a set of records. The store uses a proxy to talk to the server, which in turn tasks with a server-side list module using the Zarafa communication scheme "Request-Response flow". The server has different list modules for each type of data (tasks, mail, etc), and there are corresponding stores and proxies on the client.

Stores can do more than just plain data loading. They support pagination and sorting, and it's very easy to get this to work with the standard Ext JS components. Records can be added, removed, or updated. Changes made to the data in a store can be committed to the server by calling the save method.

A MAPI message consists of properties, but in some cases also a contents table. A spreed meting request could, for example, contain *Recipients* or *Attachments*. Distribution lists on the other hand have a list of Members. This data does not fit into the Ext JS model by default. But in the Zarafa.core.data.MAPIRecord support for SubStores has been added.

Each MAPIRecord can contain multiple SubStores which are all serialized/deserialized to/from JSON during the communication with the server. The implementation has been generalized in such a way, that plugins are able to define their own SubStores for records. The contents of a SubStore is serialized/deserialized using the JsonReader/ JsonWriter which have been configured on the SubStore itself. This means that for plugin developers they can easily create their custom table by registering the name and the type of the SubStore on the RecordFactory, and make sure a custom JsonReader and JsonWriter are set on the SubStore.

Example of plugin JSONreader is Facebook event JSON reader:

```
/*
 * #dependsFile client/zarafa/core/data/RecordCustomObjectType.js
 */
Ext.namespace('Zarafa.plugins.facebook.data');

/**
 * @class Zarafa.plugins.facebook.data.FbEventJSONReader
 * @extends Zarafa.core.data.JsonReader
 */
Zarafa.plugins.facebook.data.FbEventJSONReader = Ext.extend(Zarafa.core.data.
↪JsonReader,
{
        /**
         * @cfg {Zarafa.core.data.RecordCustomObjectType} customObjectType The
         * custom object type which represents the {@link Ext.data.Record
         * records} which should be created using {@link Zarafa.core.data.
         * RecordFactory#createRecordObjectByCustomType}.
         */
        customObjectType : Zarafa.core.data.RecordCustomObjectType.ZARAFA_FACEBOOK_
↪EVENT,

                /**
                 * @constructor
                 * @param {Object} config Configuration options.
                 */
                constructor : function(meta, recordType)
                {
                        meta = Ext.applyIf(meta || {}, {
                                id : 'id',
                                idProperty : 'id',
```

```
                                dynamicRecord : false
                        });

                        // If no recordType is provided, force the type to be a␣
↪recipient
                        if (!Ext.isDefined(recordType)) {
                                recordType = Zarafa.core.data.RecordFactory.
↪getRecordClassByCustomType(meta.customObjectType || this.customObjectType);
                        }

                        Zarafa.plugins.facebook.data.FbEventJSONReader.superclass.
↪constructor.call(this, meta, recordType);
                }
});
```

Plugin JSON writer can be found, for example, in Spreed attachments JSON writer:

```
Ext.namespace('Zarafa.plugins.spreed.data');

/**
 * @class Zarafa.plugins.spreed.data.SpreedJsonAttachmentWriter
 * @extends Zarafa.core.data.JsonAttachmentWriter
 */
Zarafa.plugins.spreed.data.SpreedJsonAttachmentWriter = Ext.extend(Zarafa.core.
↪data.JsonAttachmentWriter,
{
        /**
         * Similar to {@link Zarafa.core.data.JsonAttachmentWriter#toHash}.
         * Here we serializing only the data of the records in spreed attachment␣
↪store.
         * Note that we serialize all the records - not only removed or modified.
         *
         * @param {Ext.data.Record} record The record to hash
         * @return {Object} The hashed object
         * @override
         * @private
         */
        toPropHash : function(record)
        {
                var attachmentStore = record.getAttachmentStore();
                var hash = {};

                if (!Ext.isDefined(attachmentStore))
                        return hash;

                // Overwrite previous definition to something we can work with.
                hash.attachments = {};
                hash.attachments.dialog_attachments = attachmentStore.getId();

                var attachmentRecords = attachmentStore.getRange();
                Ext.each(attachmentRecords, function(attach) {
                        if (!Ext.isDefined(hash.attachments.add)) {
                                hash.attachments.add = [];
                        }
                        var data = attach.data;
                        hash.attachments.add.push(data);
                }, this);

                return hash;
        }
});
```

The name of the SubStore is used for the Json Data. The contents of the SubStore will be serialized with this name

into the Json Object.

Let's consider spreed substores code for example:

```
/**
 * #dependsFile client/zarafa/core/data/IPMRecipientStore.js
 */
Ext.namespace('Zarafa.plugins.spreed.data');

/**
* @class Zarafa.plugins.spreed.data.SpreedParticipantStore
* @extends Zarafa.core.data.IPMRecipientStore
*/
Zarafa.plugins.spreed.data.SpreedParticipantStore=Ext.extend(Zarafa.core.data.
↪IPMRecipientStore,
{
        /**
        * @constructor
        * @param config {Object} Configuration object
        */
        constructor : function(config) {
                config = config || {};

                Ext.applyIf(config, {
                        writer : new Zarafa.plugins.spreed.data.
↪SpreedJsonParticipantWriter(),
                        customObjectType : Zarafa.core.data.RecordCustomObjectType.
↪ZARAFA_SPREED_PARTICIPANT,
                        reader: new Zarafa.core.data.JsonRecipientReader({
                                id : 'entryid',
                                idProperty : 'entryid',
                                dynamicRecord : false
                        })
                });

                Zarafa.plugins.spreed.data.SpreedParticipantStore.superclass.
↪constructor.call(this, config)
        }
});

Ext.reg('spreed.spreedparticipantstore', Zarafa.plugins.spreed.data.
↪SpreedParticipantStore);
```

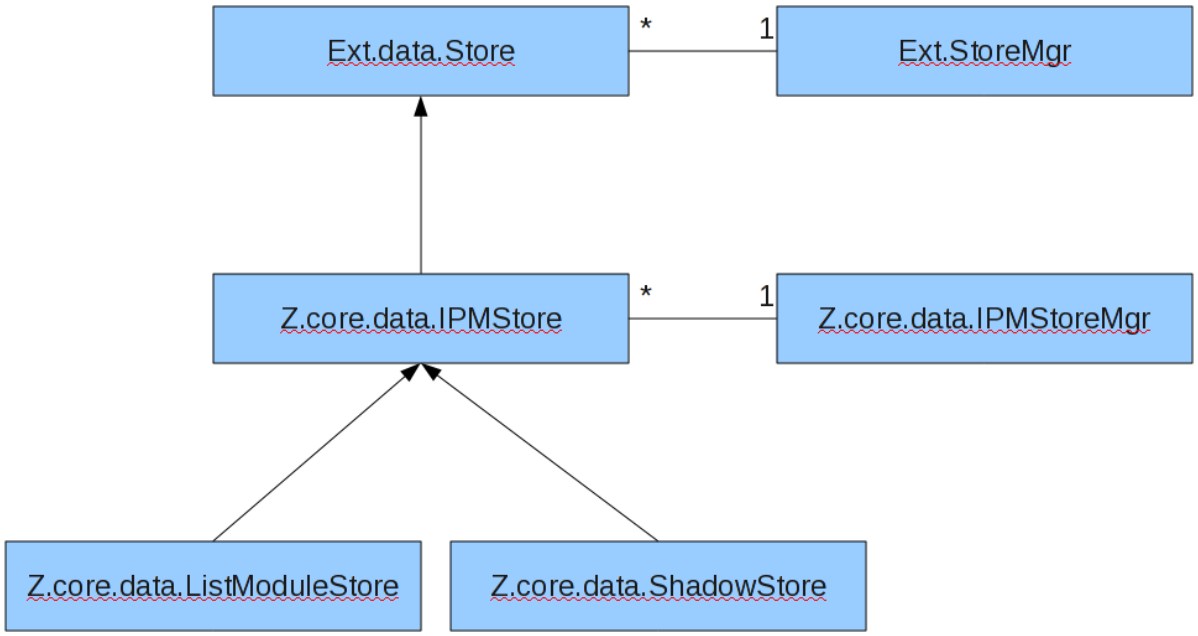
This is how it is reflected in SpreedRecord class, in constructor:

```
constructor: function(data)
{
        data = data || {};

        this.initialRecordData.participants = new Ext.util.MixedCollection();
        Zarafa.plugins.spreed.data.SpreedRecord.superclass.constructor.call(this,␣
↪data);
        this.collectedDataRecords = [];
        this.subStoresTypes =
        {
                'recipients'  : Zarafa.plugins.spreed.data.SpreedParticipantStore
        };
}
```

Where initialRecordData is an Object of recipients taken from the mailRecords, if they are opened. An important note regarding SubStores is that the SubStore is guaranteed to be available when the Record has been opened (or when it is a phantom record). When the Record has not yet been opened, the SubStore will only have been allocated if the original JsonData contains the data for the SubStore (which is not recommended).

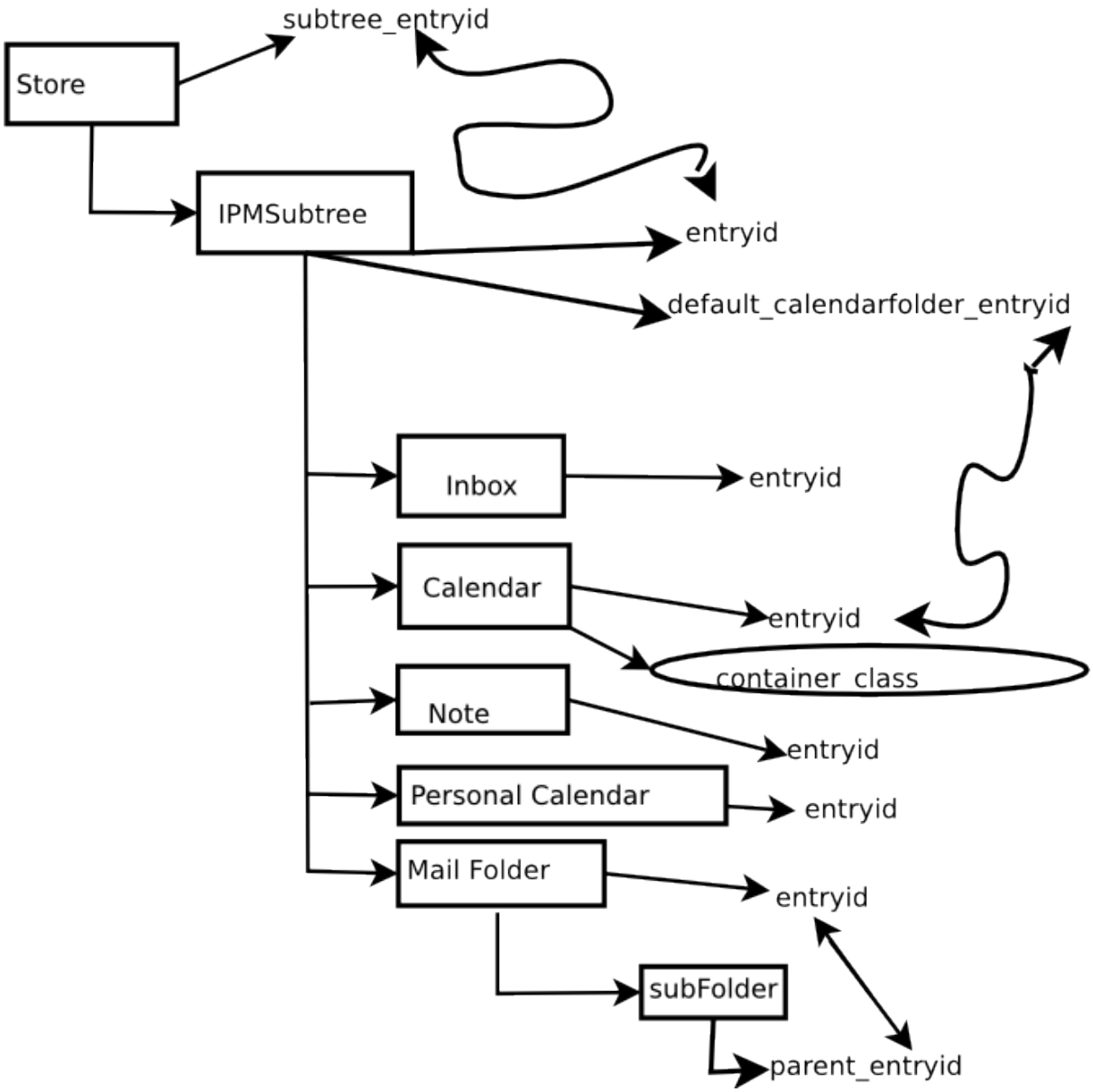
13.1.2 IPM Stores and ShadowStore

The Zarafa implementation of the Ext.data.Store is the IPMStore. Refer to "IPM Store explanation". This Store contains IPMRecords, and any IPMRecord must always belong to an IPMStore. Each IPMStore is managed by the IPMStoreMgr singleton. The IPMStore has two base classes, the first one is the ListModuleStore which is used in each Context to list all IPMRecords which must be displayed in the Context.



Store UML diagram

The second one is ShadowStore contains IPMRecords which are currently being edited within a Dialog, this includes new IPMRecords which must still be created on the server side.



IPM Store explanation

When a Dialog starts editing a IPMRecord it must copy the IPMRecord to the ShadowStore and work on that copy. When a Dialog closes it must remove record from the ShadowStore (after optionally saving the IPMRecord to the server).

Any events from a IPMStore regarding the update for IPMRecords will always be caught by the IPMStoreMgr and raised as separate event from this manager. Any UI Component which contains an IPMRecord must listen to the IPMStoreMgr for events to determine if the IPMRecord has changed and the component has to be updated. Note that listening to the IPMStore to which the Record belongs is not sufficient, because Dialogs place a copy of the IPMRecord into the ShadowStore. In which case the same Message with the same EntryId is represented by two IPMRecords in two different IPMStores.

13.2 RecordFactory

Record definitions are managed by the *Zarafa.core.data.RecordFactory*. Within the RecordFactory two groups of Record definitions exists, the first group is based on the message class (*IPM.Note*, *IPM.Contact*, etc) while the other group is based on the object type (MAPI_MAILUSER, MAPI_DISTLIST, etc). The reason for having two groups comes from MAPI which does not define the PR_MESSAGE_CLASS property for all possible objects

(most notably for Address Book items they are missing), while the PR_OBJECT_TYPE is too global to be used in all cases (There is no different value for a Mail and a Contact for instance). Ext JS allows Records to be defined using a list of possible fields, these fields have a name (the property name) and possible serialiation/deserialization information (conversion from text to Integer, Date, Boolean, etc). When a Record is created, this list is used to define which properties will be serialized/deserialized during the communication with the server. As a result, if a Plugin wishes to send an extra property it somehow has to tell Ext JS the property is valid for this Record. This is where the RecordFactory is utilized.

The RecordFactory contains the complete field definitions for all possible Record definitions. During loading, Contexts and Plugins can tell the RecordFactory which fields they wish to use for a particular Message class or object type. For example:

```
Zarafa.core.data.RecordFactory.addFieldToMessageClass('IPM.Note', [
        {name: 'from'},
]);
```

This adds the field *from* to the Record definition for any Record with messageclass *IPM.Note*. Or, according to Spreed plugin:

```
Zarafa.core.data.RecordFactory.addFieldToCustomType(Zarafa.core.data.
↪RecordCustomObjectType.ZARAFA_SPREED_ATTACHMENT, Zarafa.plugins.spreed.data.
↪SpreedAttachmentRecordFields);
```

Where *Zarafa.plugins.spreed.data.SpreedAttachmentRecordFields* is a set of fields to add to our record:

```
Zarafa.plugins.spreed.data.SpreedAttachmentRecordFields = [
        {name : 'original_record_entryid', type : 'string', defaultValue : ''},
        {name : 'original_record_store_entryid', type : 'string', defaultValue : '
↪'},
        {name : 'original_attach_num', type : 'int'},
        {name : 'original_attachment_store_id', type : 'string', defaultValue : ''}
];
```

When adding a new object type it should be added to RecordCustomObjectType enum. Refer to *Enums* for more information.

```
Zarafa.core.data.RecordCustomObjectType.addProperty('ZARAFA_SPREED_ATTACHMENT');
```

When the user creates a new Record, it must call the RecordFactory to request the Record object.

```
Zarafa.core.data.RecordFactory.createRecordObjectByMessageClass('IPM.Note');
```

This will create a new phantom record, for the messageclass *IPM.Note*. This Record now only has a single field which is allowed; namely *from*.

The plugin variant - as we used in SpreedRecord in conversion function:

```
convertToSpreedAttachment : function(attachmentRecord)
{
        return Zarafa.core.data.RecordFactory.
↪createRecordObjectByCustomType(Zarafa.core.data.RecordCustomObjectType.ZARAFA_
↪SPREED_ATTACHMENT, attachmentRecord.data);
},
```

Here we obtain a new Record object, of the custom type ZARAFA_SPREED_ATTACHMENT, with field values taken from the second parameter object. If the second parameter is not specified, values are taken from the field default value will be assigned, e.g. for *'original_attachment_store_id'* default value will be *''*.

If we want that a new phantom Records for *IPM.Note* contain a default value, we can also instruct the RecordFactory:

```
Zarafa.core.data.RecordFactory.addDefaultValueToMessageClass('IPM.Note', 'from',
↪'me');
```

Now when we create a new Record for messageclass *IPM.Note* the record will automatically have the property *from* initialized to *me*.

By default all Record instances which are created by the Record factory use the Ext.data.Record as base-class. This however is also configurable using the function *setBaseClassToMessageClass*. To force the usage of Zarafa.core.data.IPMRecord to *IPM.Note* we can use the statement as follows:

```
Zarafa.core.data.RecordFactory.setBaseClassToMessageClass('IPM.Note', Zarafa.core.
↪data.IPMRecord);
```

Now all new Record instances of Zarafa.core.data.IPMRecord for *IPM.Note* will inherit from Zarafa.core.data.IPMRecord.

The same example with Spreed plugin:

```
Zarafa.core.data.RecordFactory.setBaseClassToCustomType(Zarafa.core.data.
↪RecordCustomObjectType.ZARAFA_SPREED_ATTACHMENT, Zarafa.core.data.
↪IPMAttachmentRecord);
```

Inheritence for the messageclass works quite simple. The inheritence tree for Message class is:

```
'IPM' -> 'IPM.Note' -> 'IPM.Note.test'
      -> 'IPM.Contact'
```

When a field, default value or base class is assigned to *IPM* it is automatically propogated to *IPM.Note* and *IPM.Contact*. These values can simply be overridden within such a subdefinition. For example:

```
Zarafa.core.data.RecordFactory.addFieldToMessageClass('IPM', [
        {name: 'body'},
]);

Zarafa.core.data.RecordFactory.addFieldToMessageClass('IPM.Note', [
        {name: 'from'},
]);
```

Now all Record instances of *IPM*, *IPM.Note* and *IPM.Contact* will contain the field *body*, but only *IPM.Note* will have the additional *from* field. If we now extend our example with default values:

```
Zarafa.core.data.RecordFactory.addDefaultValueToMessageClass('IPM', 'body', 'test
↪');
Zarafa.core.data.RecordFactory.addDefaultValueToMessageClass('IPM.Contact', 'body',
↪ 'contact');
```

The record instances of *IPM* and *IPM.Note* will by default have the value *test* in the *body* property. However *IPM.Contact* will have the value *contact* in that property.

To add support for a SubStore, the following function can be called:

```
Zarafa.core.data.RecordFactory.setSubStoreToMessageClass('IPM.Note', 'recipients',␣
↪Zarafa.core.data.IPMRecipientStore);
```

This will register the SubStore with the name *recipients* to all Records which have the MessageClass *IPM.Note*. The SubStore will be created using the passed constructor (Zarafa.core.data.IPMRecipientStore). The name can be used on a MAPIRecord to detect if it supports this particular SubStore:

```
var record = Zarafa.core.data.RecordFactory.createRecordObjectByMessageClass('IPM.
↪Note');
record.supportsSubStore('recipients');
// True if the record supports the subStore
record.getSubStore('recipients');
// this returns the Zarafa.core.data.IPMRecipientStore allocated for this record
```

A final warning, adding fields, default values or base classes can only be done during initial loading. This means that these statements must be done outside any function or class. Calling Zarafa.core.data.RecordFactory.createRecordObjectByMessageClass can only be made safely after the initial loading (thus it can be done safely in functions and classes).

CHAPTER 14

Deployment and Build System

14.1 Deployment

All plugins files should be created in a special separate *plugin* directory - `webapp/plugins`.

Each plugin or widget should have one separate directory. For example,*Spreed* plugin has the following items inside:

- **js** - folder.
  - **data** - folder //all classes realted to working with data are stored here.
  - **dialogs** - folder //all classes realted to dialogs and ui should be placed here.
  - **SpreedPlugin.js** //cannot be sorted properly being the main plugin class and is separate.
- **php** - folder.
  - **data** - folder //data stored here like list of languages and timezones.
  - **inc** - folder //php classes stored here.
- **resources** - folder.
  - **css** - folder //css resources places here.
  - **icons** - folder // different icons stored here.
- **build.xml** //for building spreed plugin (discussed further in chapter *Build System*).
- **config.php** //defined `php` constants.
- **manifest.xml** //manifest file.

Example widget hierarchy let be *end of world* widget:

- **js** - folder.
  - **FBWidget.js** //widget class.
- **resources** - folder.
  - **css** - folder //css files for widget placed here.
- **images** - folder //images for widget go here.
  - **facebook.png**

- **build.xml**
- **manifest.xml**

If you want your plugin to be enabled/disabled using Zarafa Settings menu you should create a new folder in *[WebApp root folder]/plugins* directory with the name of your plugin e.g. *plugins/facebook*, *plugins/spreed*. There you should create at least two files:

- manifest.xml
- plugin.facebook.php

You may also create *config.php* file to set default value for your plugin settings. Here goes the example of FB plugin *config.php* file.

The subfolder names and hierarchy can be changed, but it still should stay intuitively understandable.

Manifest is a file, where all the files of your plugin are described. The structure should be strict:

```
<?xml version="1.0"?>
<!DOCTYPE plugin SYSTEM "manifest.dtd">
<plugin version="2">

        //First block represents the common info about plugin
        <info>
                <version>0.1</version>
                <name>facebook</name>
                <title>Zarafa Facebook Integration</title>
                <author>Zarafa</author>
                <authorURL>http://www.zarafa.com</authorURL>
                <description>Zarafa Facebook Integration, allows you to import␣
↪your Facebook events to Zarafa calendar.</description>
        </info>

        //Second block describes where to find config file of your plugin
        <config>
                <configfile>config.php</configfile>
        </config>

        //Third block describes files that are used in your plugin
        <components>
                <component>
                        <files>
                        <server>
                                <serverfile>php/plugin.facebook.php</serverfile>
                        </server>
                        <client>
                                // loaded client files should have an attribute
↪'load', which can take one of three values:
                                //source, debug and release. Source are files that␣
↪you created during development, debug - the one file,
                                //compiled from all your files, and the release is␣
↪the compressed debug file.
                                <clientfile load="release">js/facebook.js</
↪clientfile>
                                <clientfile load="debug">js/facebook-debug.js</
↪clientfile>
                                <clientfile load="source">js/FacebookEventsPlugin.
↪js</clientfile>
                                <clientfile load="source">js/
↪FbEventSelectionDialog.js</clientfile>
                                <clientfile load="source">js/FbIntegrationDialog.js
↪</clientfile>
                                <clientfile load="source">js/FbEventSelectionGrid.
↪js</clientfile>
```

```
                                <clientfile load="source">js/FbIntegrationPanel.js
↪</clientfile>
                                <clientfile load="source">js/data/
↪FbEventDataReader.js</clientfile>
                                <clientfile load="source">js/data/FbEventProxy.js</
↪clientfile>
                                <clientfile load="source">js/data/FbEventRecord.js
↪</clientfile>
                                <clientfile load="source">js/data/FbEventStore.js</
↪clientfile>
                        </client>
                        <resources>
                                //here are css files needed in your plugin
                                <resourcefile load="release">resources/css/
↪facebook.css</resourcefile>
                                <resourcefile load="source">resources/css/plugin.
↪facebookstyles.css</resourcefile>
                        </resources>
                </files>
                </component>
        </components>
</plugin>
```

14.2 Build System

The WebApp is a relatively complex application consisting of a server part, written in PHP, and a client part written in JavaScript. Tasks such as building, deploying, and testing the application, and also generating documentation are automated in a build system. The tool we use is Ant.

An important note here is that the build system is something that Zarafa company use, but it is not really mandatory for all plugin developers to use it too. Zarafa provides the required tools in the source package, but not in the normal installation packages. Plugin developers do not require the source package to create plugins, so the presence of *build.xml* is also completely optional and only can be present if plugin developers want to use Zarafa building system.

14.2.1 Getting Started

Prerequisites for using the build system is a Java 6 JDK and Ant. Other libraries and tools are kept in svn either as binaries or as source.

Linux

On most distributions both Java 6 and Ant should be in the repositories. Make sure you don't get GNU Java (gcj) but get the sun jdk or openjdk. On Ubuntu, simply type:

```
sudo apt-get install sun-java6-jdk ant ant-optional
update-alternatives --auto java
```

Especially the last line is important since multiple versions of Java can be installed side by side on a system, and the default has to be selected explicitly. Ant should be installed and configured properly by your package manager.

Windows, Mac OSX

These platforms have not been tested, but should work. The Java 8 JDK can be downloaded from Oracle.

To test if everything works as it should, go to the root of you WebApp 7 folder and simply type ant. Your system should build the tools, build the client application, and run the test suite.

14.2.2 Ant Introduction

Ant is an XML-based build system written in Java, and is quite well known in the Java community. We have chosen Ant to build our application because many of the tools that we use to process our JavaScript code were written in Java, such as the ext-doc documentation tool, our own JsConcat concatenation tool and others.

Ant build files are XML files, usually called build.xml. They define a project with several nested properties, typedefs, and tasks. Here's a partial example.

```
<project name="example" default="all">
        <property name="foo" value="bar"/>

        <target name="all" depends="compile"/>

        <target name="init">
                <!-- do something here -->
        </target>

        <target name="compile" depends="init">
                <!-- do something here -->
        </target>
</project>
```

The `target` tag defines a *target*. Targets have nested *tasks*, like calling a compiler, creating an output directory etc. The property tag acts like `key=value` in make, with the exception that properties are defined only once and become immutable. Any redefinition of the property is ignored. The following example illustrates the use of properties and how redefinitions are ignored:

```
<project name="example2" default="all">
        <property name="foo" value="bar"/>
        <property name="foo" value="not bar"/>
        <!-- Will output 'bar' -->
        <echo message="${foo}"/>
</project>
```

This mechanism is used by ant build files calling other build files. If an ant file calls another ant file, to build a sub project for instance, the caller may override properties in the callee to set configuration switches, output directories, and so forth. Properties in ant build files may also be overridden from the command line. Running the above example with and -Dfoo=foo would cause ant to echo *foo*.

Ant is a build system that can be both powerful and frustrating, and it is certainly quite different from tools like make or scons. For more information, please refer to the Ant manual.

14.2.3 Tools

The build system uses several tools to accomplish various tasks. Some of these are third-party, and some of these are custom.

JsConcat

Large Ext JS applications tend to be made up of many individual files. Our naming convention dictates that most classes are defined in their own files. When the application is deployed all these files are bundled together by concatenating them into a single JavaScript file, which is then minimised.

A standard *concat* ant task exists that concatenates a set of files. The file order is important because files may depend on one or more global variables declared other files. Specifically in Ext JS applications, if some class *Foo* extends another class *Bar*, *bar.js* must be included before *foo.js*. It is possible to specify this inclusion order by hand, but considering the large number of files to manage, this is a maintenance nightmare. Therefore JsConcat was created, a concatenation task that scans the input files and determines the proper inclusion order automatically.

Inclusion Order

There are three ways to influence the inclusion order. First, there are explicit dependencies that you can enter in your file:

```
/*
 * This file needs to be included after bar.js
 * #dependsFile /foo/bar.js
 */
```

Dependencies are also extracted from `@class` and `@extends`:

```
/**
 * Foo extends Bar, so the file in which Bar is defined should come before this␣
↪file.
 * @class Foo
 * @extends Bar
 */
```

Finally, formatted as a comma-separated list of regular expressions, the *prioritise* argument can be used to move groups of files up the list. Files that have classes defined in them have those full class names matched against the regexps. Files that match the first priority group have the highest priority, files that match the second group come after that, and so on. We use this mainly to move the *core* and *common* packages to the top of the list.

For example,"+, Foo.bar.*" will move all files which have classes in the *root* package (i.e. *Date*, *Foo*) to the top of the list, after wich come all files which have classes in *Foo.bar* or any of its descending packages, and finally all the files that match neither of these criteria.

Example Build File

```
<!-- Define the 'jsconcat' task -->
<taskdef name="jsconcat" classname="com.zarafa.jsconcat.JsConcatTask">
        <classpath path="JsConcat.jar"/>
</taskdef>

<!-- Concatenates JavaScript files with automatic dependency generation -->
<target name="concat">
        <jsconcat verbose="false" destfile="${debugfile}" prioritize="\w+, Foo.bar.
↪*">
                <fileset dir="${sourcedir}" includes="**/*.js"/>
        </jsconcat>
</target>
```

Ext-doc

Ext JS comes with a JavaDoc inspired code documentation tool that generates nice HTML documentation. There is a fork of this tool in the svn that supports extra tags to document insertion points. More information can be found at the ext-doc google project page.

Closure Compiler

The Closure Compiler is a Javascript minimisation tool from Google. It will compile a concatenated JavaScript file. It supports multiple compile modes, only *whitespace* guarantees to only change the whitespaces and minimise the Javascript file without changing the code itself. The other compilation modes will change the code itself (for example, rename variables, make functions inline, etc).

CHAPTER 15

Translations

WebApp provides a multilanguage interface. The language can be switched in settings. We use GNU gettext for translating. In your JS code just rememer to translate each string label using _(*your string label*):

Example:

```
/**
 * Create all buttons which should be added by default the the `Actions` {@link␣
↪Ext.ButtonGroup ButtonGroup}.
 * This will create {@link Ext.ButtonGroup ButtonGroup} element with Spreed setup␣
↪button.
 *
 * @return {Array} The {@link Ext.ButtonGroup ButtonGroup} elements which should␣
↪be added
 * in the Actions section of the {@link Ext.Toolbar Toolbar}.
 * @private
 */
createActionButtons : function()
{
        return [{
                xtype : 'button',
                text : _('Setup Meeting'), // button text translation
                tooltip : {
                        title : _('Setup Meeting'), //tooltip title translation
                        text : _('Setup meeting with provided details')  //tooltip␣
↪text translation
                },
                iconCls : 'icon_spreed_setup',
                handler : this.setUpMeeting,
                scope : this
        }];
},
```

15.1 Gettext

Information about the GNU gettext project can be found at http://www.gnu.org/software/gettext/

GNU gettext allows us to supply the entire WebApp in the language of the user's choosing. It also allows the developer to implement plural translations and context-based translations.

With plural translations you can have gettext handle whether you should use a singular or plural translation.

PHP:

```
<?php echo sprintf(ngettext('Deleted %s file','Deleted %s files', $count), $count);
↪ ?>
```

JavaScript:

```
ngettext('Deleted %s file','Deleted %s files', count).sprintf(count);
```

Context-based translations allows you to translate the same word in English, but different in other languages. You add a context in which you specify where you use the translatable text and gettext can distinquish the different strings.

PHP:

```
<?php
        echo pgettext('Menu', 'Open');
        echo pgettext('Dialog.toolbar', 'Open');
?>
```

JavaScript:

```
pgettext('Menu', 'Open');
pgettext('Dialog.toolbar', 'Open');
```

For developers it is also possible to add commentary for the translators. You can do this by adding the following comment to the code just above the gettext function.

PHP:

```
<?php
        /* TRANSLATORS: Comment
        * Extra comment
        */
        _('English text');

        // TRANSLATORS: Comment
        // Extra comment
        _('English text');
?>
```

JavaScript:

```
/* # TRANSLATORS: Comment
* # Extra comment */
_('English text');

// # TRANSLATORS: Comment
// # Extra comment
_('English text');
```

For the extraction of the translations from the JavaScript files gettext offers the utility xgettext. However, this utility cannot read a JavaScript file. To make it work we set the language to Python (yes, that is not a joke). The downside of this is that comments directed at the translators need to be formatted a certain way. Python does not understand the JavaScript comments so we need to add a # on every line preceeding the gettext function. Also when you are using the multiline comment (/* */) the last line must contain the # character.

15.2 Server-Side

On the server-side the PHP has by default some gettext functions implemented already. It still misses a few though. That is why the of the WebApp server-side implements the following functions: pgettext, npgettext, dpgettext and dcpgettext.

More information about the default PHP functions can be found at http://php.net/gettext/.

15.2.1 Implemented Functions on the Server-Side

PHP misses the context gettext functions and that is what the WebApp implements. The $msgctxt argument supplies the context of the translation. The $msgid is the translatable string. The different functions implement the context together with the normal gettext function, plural, domain and category gettext functions.

```
<?php
        pgettext($msgctxt, $msgid);
        npgettext($msgctxt, $msgid, $msgid_plural, $num);
        dpgettext($domain, $msgctxt, $msgid);
        dcpgettext($domain, $msgctxt, $msgid, $category);
?>
```

15.3 Client-Side

15.3.1 Reading Translations and Passing it to the Client

The language class is the server-side PHP class that reads the binary .mo file in the language folder of the WebApp and returns a list of those translations. The client requests the page index.php?load=translations.js which includes the file client/translations.js.php. In this file the Translations class is defined and the translations are added inside this global Javascript object.

The plural-forms are defined inside the "first defintion". This first defition is an empty string with the value containing information like the following.

```
msgid ""
msgstr ""
"Project-Id-Version: Weblang\n"
"POT-Creation-Date: 2015-07-16 16:16:06+0100\n"
"PO-Revision-Date: 2015-07-16 16:16:06+0100\n"
"Last-Translator: Weblang automatic translator <foo@bar.foobar>\n"
"Language-Team: Kopano Team <development@kopano.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=iso-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=3; plural=n==1 ? 0 : n==2 ? 1 : 2;"
```

As you can see the last line defines the plural forms. Based on this string gettext will understand what singular or plural form it needs to use. In English there are only two, but in other languages you might have a different one when dealing with zero. Some other languages make it even more complex with different cases for numbers ending with one, except for eleven.

15.3.2 Implemented Functions on the Client-Side

The client-side implements the same function we have on the server-side. Except for the functions that implement the gettext category (the c-functions). The msgid argument is the translatable string. The msgctxt is the context argument.

```
_(key, domain)
dgettext(domain, msgid)
ngettext(msgid, msgid_plural, count)
dngettext(domain, msgid, msgid_plural, count)
pgettext(msgctxt, msgid)
dpgettext(domain, msgctxt, msgid)
npgettext(msgctxt, msgid, msgid_plural, count)
dnpgettext(domain, msgctxt, msgid, msgid_plural, count)
```

The Translations object will use the plural forms string that was extracted from the binary .mo file and passed on to the client. It is put into a function to be called when dealing with a plural translation.

CHAPTER 16

Appendix A: Naming Conventions

16.1 Namespace Structure

The bulk of the code lives in client/zarafa, representing the Zarafa namespace.

- **Zarafa** - the root namespace
  - **core** - the application *core* framework. Includes global models such as the folder hierarchy, the plugin, contexts, and widgets system, etc.
    - **ui** - UI components used in the core framework. Navigation panel, main viewport, etc.
    - **dialog** - core dialogs.
  - **common** - code that is common to all contexts and plugins.
    - **ui** - common UI components: value pickers, form panels.
    - **dialog** - dialogs used by multiple contexts.
  - **[context]** - context specific code, i.e. freebusy, mail, contact, etc.
    - **ui** - UI components used exclusively in this context.
  - **dialog** - context specific dialogs.
  - **plugins** - Zarafa standard plugins.
  - **widgets** - Zarafa standard widgets.

16.2 Naming Packages and Classes

Package names are lower case, except the top level *Zarafa* package. Class, method, and field names follow Java conventions; classes are camel case starting with a capital letter (i.e. *CalendarMultiView*, *MailStore*), methods and fields start with a lower case letter (i.e. *getParentView()*, *load()*, *folderList*).

The package/class structure follows the folder/file structure, much like with a Java project. For instance, *Zarafa.calendar.ui.CalendarPanel* can be found in zarafa/calendar/ui/CalendarPanel.js. Most complex classes will have their own file. Small/trivial classes that are used from only one place may be placed inside another class's file.

16.3 Naming Insertion Points

Insertion points also follow a hierarchy. Top level:

- Main application - *main*
- Hierarchy panel - *hierarchy*
- Contexts - *context.[name]*, i.e. *context.task*, *context.mail*
- Plug-ins - *plugin.[name]*, i.e. *plugin.folderstatus*, *plugin.sugarcrm*
- Widgets - *widget.[name]*, i.e. *widget.clock*, *widget.news*

Common node types:

- Dialog - *dialog.[name]*, i.e. *dialog.edit*
- Context menu - *contextmenu*
- Tool bar - *toolbar*
- Status bar - *status*

Most common insertion points' names are of the following structure

```
{main|hierarchy|context.[context name]|plugin.[plugin name]|widget.[widget name]}.
{dialog.[dialog name]|contextmenu|toolbar|status}*
```

16.3.1 Coding Style Guidelines

In order to maintain consistency a few guidelines shall be followed in the WA projects.

Placement of brackets.

Function declarations shall always have brackets on a new line.

```
getModel : function()
{
        return this.model;
},
```

`If-else` statements shall use a condensed style.

```
if (item.isXType('zarafa.recipientfield'))
{
        item.setRecipientStore(record.getRecipients());
} else {
        item.setValue(value);
}
```

When an if-statement scope only consists of one row then the brackets should also be present.

```
if (!this.bodyInitialised)
{
        this.initBody();
}
```

Never use single line if-statements (WRONG).

```
if (!this.bodyInitialised) this.initBody();
```

For the ternary operator one line is ok.

```
iconClass = record.isRead() ? 'icon_mail_read' : 'icon_mail_unread';
```

It's also highly recommended to **use tabs** instead of usual four spaces characters in indentation, as long as each developer can determine whether he wants his editor to show the appropriate number of spaces for each indentation of the code.

After the first character spaces are used.

16.3.2 Documentation

The application is documented using the `ext-doc` documentation tool provided by the Ext JS people. It allows documenting JavaScript code much like Javadoc or Doxygen. This section describes code is documented. Since Javascript is a very dynamic language it's pretty much impossible to detect class, method, and field definitions, and the relationships between them. Therefore documentation is quite explicit. One has to declare classes, methods, and fields manually. This section briefly describes the most important aspects of documenting with ext-doc. Please refer to the ext-doc documentation wiki for more information.

16.4 Documenting Classes

A class is declared using the `@class` statement inside a `/** */` multi-line comment. One can use `@extends` to indicate that the class is a subclass of another class. A description of the class follows these two statements. Optionally the `@singleton` statement can be used to declare the class a singleton. For classes which inherit directly or indirectly from Ext.Component, should use the `@xtype` to document which xtype can be used to automatically instantiate the object through the xtype. The constructor will be documented separately and is documented much like a method. Finally use `@cfg` to declare configuration options for this class. Note that parameters and configuration options are typed.

Plugin example:

```
/**
 * @class Zarafa.plugins.spreed.SpreedPlugin
 * @extends Zarafa.core.Plugin
 *
 * This class integrates Spreed plugin in existing system.
 * It allows user to setup spreed web meeting settings.
 */
Zarafa.plugins.spreed.SpreedPlugin = Ext.extend(Zarafa.core.Plugin, {
        //class code
}
```

16.5 Documenting Fields

Within a class configuration fields (which are configured using the `config` parameter in the constructor) must be added and documented. These fields must be documented using the `@cfg` within a `/** */` multi-line comment.

```
Zarafa.plugins.spreed.SpreedPlugin = Ext.extend(Zarafa.core.Plugin, {

        /**
         * Contains link to the spreedStore class
         * initialized once when plugin is created.
         *
         * @property
         * @type Object
         * @private
         */
        spreedStore : null,
```

```
        /**
         * Unique id which works instead entryid
         * for SpreedRecord.
         *
         * @property
         * @type Integer
         * @private
         */
        sequenceId : 0,

        //code

});
```

16.6 Documenting Constructors

Constructors must be documented inside a `/** */` multi-line comment. Use `@constructor` to mark the function as a constructor. Other than that the function is simly documented as a regular function. It is recommended, but not mandatory, to use `//` single-line comments (which will not be parsed for online documentation) to document configuration value overrides for the superclass.

```
/**
 * @constructor
 * @param {Object} config Configuration object
 *
 */
constructor : function (config)
{
        config = config || {};
        Ext.applyIf(config, {
                name : "spreed"
        });

        Zarafa.plugins.spreed.SpreedPlugin.superclass.constructor.call(this,␣
↪config);
        this.init();
},
```

16.7 Documenting Methods

The following listing shows how to document methods. Parameters can be specified using `@param` with a type, name, and description. The method name will be extracted automatically, but if for any reason this fails, adding `@method [name]` will solve that. If an argument is an optional - one can make this explicitly by using `(optional)` as exemplified by the `errorCallBack` parameter. For method which are private, only visible for the own class, then `@private` annotation shall be added. Do not use the "old" way of describing private (private) in the method description and/or with one less * in the comment opening. If this old documenting format is found in the code base it shall be updated to use @private.

```
/**
 * Similar to {@link Ext.data.JsonWriter#toHash}
 *
 * Convert recipients into a hash. Recipients exists as
 * {@link Zarafa.core.data.IPMRecipientRecord IPMRecipientRecord} within
 * a {@link Zarafa.core.data.IPMRecord IPMRecord} and thus must be serialized
 * seperately into the hash object.
 *
```

```
 * @param {Ext.data.Record} record The record to hash
 * @return {Object} The hashed object
 * @override
 * @private
 */
toPropHash : function(record)
{
        //code
        return hash;
}
```

16.8 Documenting Insertion Points

Insertion points should be documented just after the class declaration. The name of the insertion point can be specified using `@insert` with a name of the insertion point. Parameters can be specified using `@param` with a type, name, and description.

For example, populating insertion point for new menu item:

```
Zarafa.core.ui.MainToolbar = Ext.extend(Zarafa.core.ui.Toolbar, {
        // Insertion points for this class
        /**
        * @insert main.maintoolbar.new.item
        * Insertion point for populating the "New item" menu. It will be placed in␣
↪the item part of the
        * list. Each item inserted to this list is accessible from all contexts.
        * @param {Zarafa.core.ui.MainToolbar} toolbar This toolbar
        */

        ...

        //population of insertion point itself
        var itemMenu = container.populateInsertionPoint('main.maintoolbar.new.item
↪', this) || [];

        ...
}
```

16.9 Documenting Enumerations

An example of documenting enumeration is *Spreed Dialog types* enumeration. An enumeration is declared using the `@class` statement inside a `/** */` multi-line comment. One can use `@extends` to indicate that the enum extends `Zarafa.core.Enum`. A description of the class follows these two statements. Optionally the `@singleton` statement can be used to declare the class a singleton. Each property of the enum should be documented, telling what it means and what type it is.

```
/**
 * @class Zarafa.plugins.spreed.data.DialogTypes
 * @extends Zarafa.core.Enum
 *
 * Enum containing the different types of dialogs needed to display spreed meeting.
 * @singleton
 */
Zarafa.plugins.spreed.data.DialogTypes = Zarafa.core.Enum.create({

        /**
         * The dialog with empty fields.(Brandly new)
```

```
         *
         * @property
         * @type Number
         */
        EMPTY  : 1,

        /**
         * The dialog with filled subject and participants.
         *
         * @property
         * @type Number
         */
        FILLED : 2,


        /**
         * The dialog with only participants field prefilled.
         *
         * @property
         * @type Number
         */
        PARTICIPANTS_FILLED : 3

});
```

CHAPTER 17

Appendix B: References

Since many resources such as the API references would not make sense to cover in this document, these and other helpful references should help in getting started in development with WebApp.

- Kopano product page
- Kopano WebApp API reference
- Kopano WebApp user manual
- Sencha Ext JS 3.4.0 API documentation

CHAPTER 18

Legal Notice