syncgw

DEVELOPER GUIDE

Table of Content

Chapter 1: Figure reference

Chapter 2: Code reference

Chapter 3: Definitions and abbreviations

[APP]	*“Application”* is a Word Wide Web server-based application.
[DB]	*“Application Data Base”* is the place where [APP] data is stored. The layout of the [RECORD]s (and the location where the records are stored) depends on [APP] internal architecture.
[CONNECTOR]	The *“Data base connection handler*” is used by [SERVER] to access records in [DB].
[CLIENT]	Client device on which user initiates synchronization session. The [CLIENT] may be a cell phone or personnel digital assistant (PDA). On most [CLIENT], the synchronization capabilities are build-in; on other [CLIENT] it may be required to install special client software.
sync●gw	**sync●gw** server acts as a gateway between [CLIENT] and [DB]. The server provides data conversion utilities and support of a couple of data protocols enabling communication with [CLIENT].
[CONFIG]	**sync●gw** configuration is stored in configuration file `syncgw/config.ini.php`.
[BROWSER]	The standalone **sync●gw** *“Web User interface*” is available at `http://[yourdomain]/sync.php`.
[DATASTORE]	*“Data Store*” contains all related data to a specific data store (e.g. all contact data). Depending on which [CONNECTOR] is used, the data may be physically stored in MySQL tables, in flat files or somewhere else (e.g. in an Oracle data base).
[GROUP]	All records are stored in groups. These may be user defined. If no group is selected during synchronization, **sync●gw** uses a special group called *“default*” in each [DATASTORE] to synchronize. Please note this group cannot be deleted and is always automatically (re-) created. During [SYNCML] synchronization, only the “default” group is used. Other groups may be selected using [WEBDAV] synchro-nization.
[RECORD]	A *“Data Record”* contains an indefinite number of [FIELD]s. In case of e.g. a contact [RECORD], there may be one name with multiple telephone numbers assigned and multiple location information such as work and private address.
[FIELD]	A *“Data field*” is one piece of information store in a [RECORD]. In case of a contact [RECORD], each telephone number is a [FIELD].
[GUID]	The *“Global Unique Identified*” is the primary key used by **sync●gw** to access a specific [RECORD] in a specific [DATASTORE] for a specific user.
[LUID]	The *“Local Unique Identifier”* is the primary key used by the [CLIENT] to access a specific [RECORD] in a specific [DATASTORE] for a specific user.

Record mapping	During access to external [DATASTORE] all internal [FIELD] needs to be mapped to external [FIELD]. This process is called "*Record mapping*". This transformation is performed based on an XML mapping table.
Character Encoding	[SERVER] use internally the UTF-8-character set. During conversation with [CLIENT], all user data is automatically converted to the character set supported by [CLIENT]. If you want to know more about the supported character sets, please open [BROWSER] and select "*Check status*". Please search for "*Encoding handler*". Below the header line all supported character sets are listed.
[WAP]	The "*Wireless Application Protocol"* used for transferring data across 3G networks.
[WBXML]	"*WAP Binary XML*" is a binary representation of [SYNCML] protocol which is used to synchronize smaller data packages during synchronization.
[WEBDAV]	"*Web-based Distributed Authoring and Versioning*" is a set of methods based on the Hypertext Transfer Protocol (HTTP) that facilitates collaboration between users in editing and managing documents and files stored on Word Wide Web servers.
[MS-EAS]	*MicroSoft Exchange ActiveSync*
[MS-ASPROV]	*MicroSoft Exchange ActiveSync: Provisioning Protocol*
[MS-ASRM]	*MicroSoft Exchange ActiveSync: Rights Management Protocol*

Chapter 4: Introduction

sync●gw is written fully object oriented. This approach makes it easy to integrate [BROWSER] in your [APP] without running into potential conflicts with global variables and function names. To reduce allocated PHP memory, we additionally implemented late object binding - internal class definitions and class objects are only loaded as they are required. Starting from version 6 we also restrict memory allocation to singleton instance of one class object.

Chapter 5: Data flow between [CLIENT] and sync•gw

5.1 Session initiated by [CLIENT]

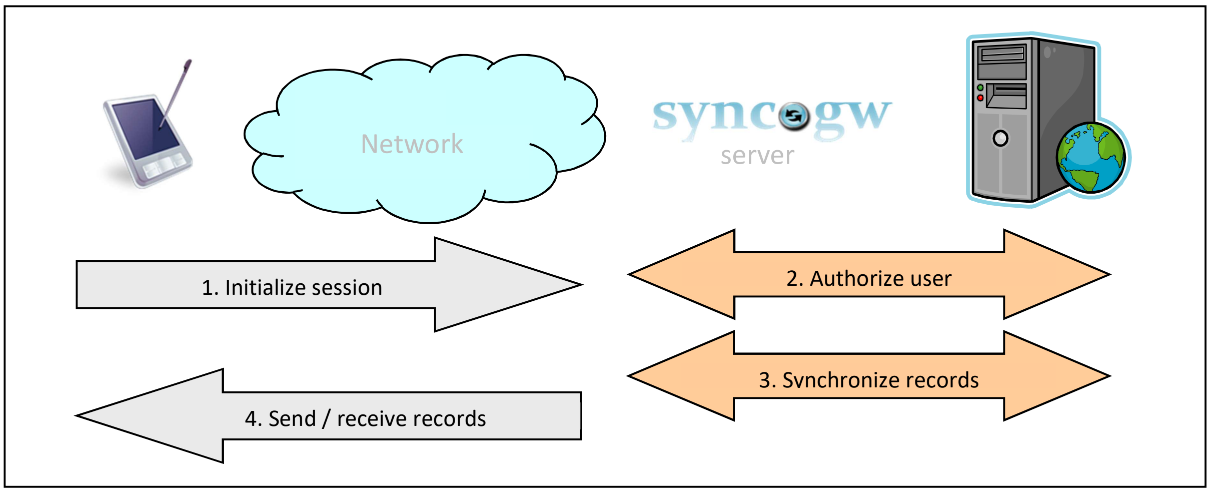

Figure 1: Session initialization by [CLIENT]

Figure above show the data flow during session initialization between session initiated by [CLIENT], sync•gw and [APP].

(1) User initiates a synchronization session on [CLIENT]. The [CLIENT] connects to sync•gw through available network (3G, WLAN).

(2) sync•gw authorizes user in [APP] (using [CONNECTOR]). If authorization fails, an appropriate return code is sent to [CLIENT] and session is terminated.

(3) The first-time sync•gw access the internal [DATASTORE] in session, sync•gw synchronize [DB] with [DATASTORE]. During this process, all internal [RECORD] are mapped to external [RECORD] (using methods provided by [CONNECTOR]). Then the records are compared with [RECORD] loaded from [DB] to discover any changes.

- If sync•gw detects any field change, the appropriate field is mapped back to internal representation and the internal [RECORD] is updated. Finally [RECORD] is flagged for synchronization with [CLIENT].

- In case [RECORD] is found in [DATASTORE] but not found in [DB], [RECORD] is flagged for deletion on [CLIENT].

- If [RECORD] is not found in [DATASTORE] (but in [DB]), then it is loaded into [DATASTORE] and flagged to be added on [CLIENT] on next synchronization.

(4) Synchronization is started. During follow-up communication all flagged [RECORD] are synchronized with [CLIENT].

5.2 Data exchange

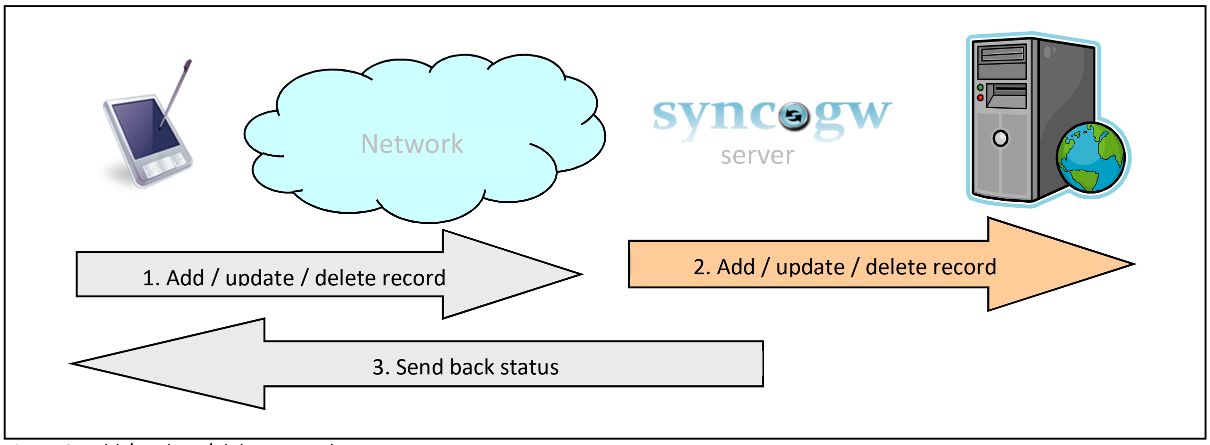
Network
sync•gw
server
1. Add / update / delete record
2. Add / update / delete record
3. Send back status

Figure 2: Add / update / delete record

Figure above show the data flow for sync•gw data exchange processing.

(1) [CLIENT] send updates to existing [RECORD], new [RECORD] or notifies sync•gw about deletion of [RECORD].

(2) sync•gw performs the requested action on the [DATASTORE] and initiates associated changes on [DB] using [CONNECTOR].

(3) The status of the command is sent back to [CLIENT].

Chapter 6: [CONNECTOR] to [APP]

There is no straightforward interface between sync•gw and [APP]. The only link is the external [DB]. This simplifies the integration of sync•gw into [APP] extremely. There is no need to make any modification to [APP] code and [APP] doesn't need to take care about sync•gw processing.

To access external [RECORD] sync•gw uses [CONNECTOR] to perform any add, update and delete to [DB].

The following picture shows the central role of [CONNECTOR] in data processing.

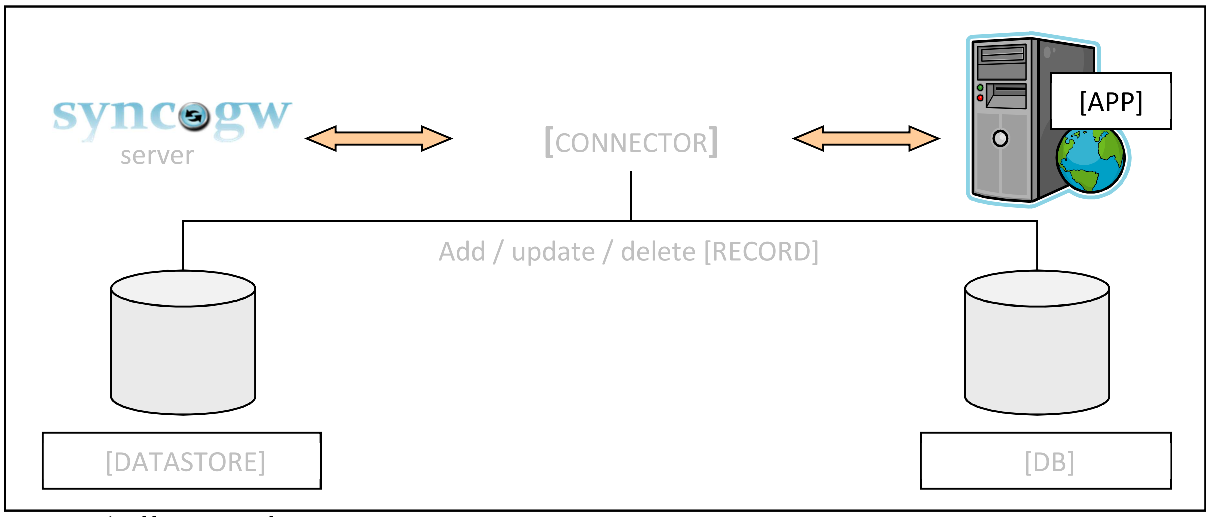

Figure 3: Role of [CONNECTOR]

6.1 Add or update [RECORD] processing

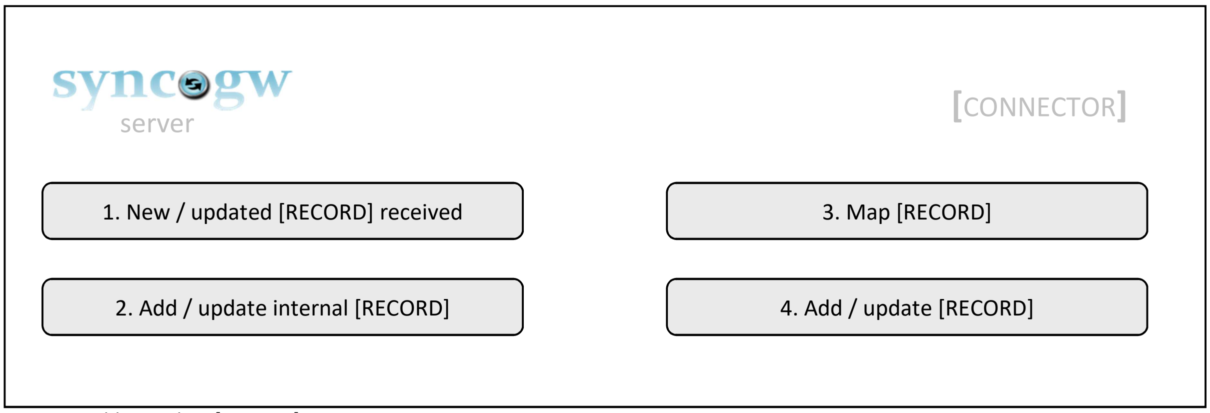
sync•gw
server
[CONNECTOR]
1. New / updated [RECORD] received
2. Add / update internal [RECORD]
3. Map [RECORD]
4. Add / update [RECORD]

Figure 4: Add or update [RECORD] processing

Figure above illustrates which steps are performed during processing of new [RECORD] (add) or updated [RECORD] received from [CLIENT].

(1) sync•gw receive a new [RECORD] (or a request for update on an existing [RECORD]) from [CLIENT]

(2) sync•gw adds new or updates an existing [RECORD] in [DATASTORE].

(3) sync•gw calls [CONNECTOR] to map [RECORD] to external record. [CONNECTOR] may use mapping methods provided sync•gw.

(4) [CONNECTOR] adds mapped record to or updates existing record in [DB].

6.2 Delete [RECORD] processing

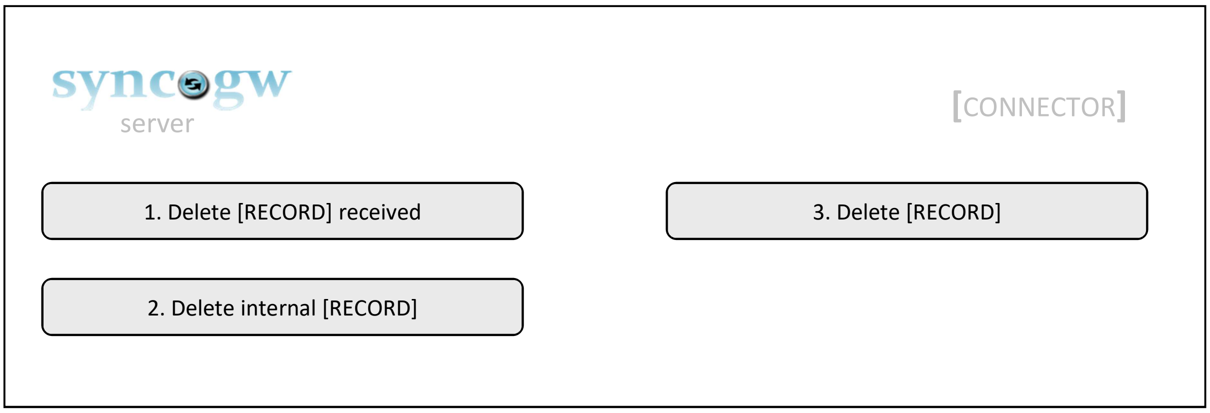

Figure 5: Delete [RECORD] processing

Figure above illustrates which steps are performed during processing of delete [RECORD] requests received from [CLIENT].

(1) **sync•gw** receives the request for record deletion and deletes [RECORD] in [DATASTORE] from [CLIENT].

(2) **sync•gw** deletes [RECORD] in [DATASTORE].

(3) **sync•gw** calls [CONNECTOR] to delete external record.

6.3 [RECORD] mapping

sync•gw assume [APP] is using its own [DB] and its own [RECORD] layout. **sync•gw** needs to map internal and external [RECORD] in various situations. This process is called "*[RECORD] mapping*".

To enable **sync•gw** to perform this mapping and add, update or delete [RECORD] in [DB], [CONNECTOR] **must** provide methods which performs the mapping of the internal and of the external [RECORD].

6.4 Data exchange options

During synchronization either on [CLIENT] or on **sync•gw** [RECORD] are added, updated or deleted.

In some **sync•gw** installations it may be required to control the exchange of data.

Imagine you want to set up a global phone directory for your employees. This directory should only be maintained on server. This requires a way to reject any changes received from [CLIENT].

Or think about your [CLIENT] data should be protected under all circumstances. This requires a way to reject any changes in [DB].

This behavior may be configured by setting the appropriate "*Synchronization options*" in [BROWSER].

Chapter 7: Creating myApp [CONNECTOR]

In the following chapters, we create a sample [CONNECTOR] named `myApp`.

The connector files were located in directory `vendor/syncgw/myapp-bundle`.

We assume

- There is an existing [APP] called `myApp` available up and running. This is not part of the sample code.
- The [APP] data is stored in a MySQL data base.
- `myApp` [CONNECTOR] will only support contact data.
- The internal **sync•gw** [DATASTORE] is in same data base.

To create the initial environment for [APP], please import file `tables.sql` in a MySQL data base of your choice. This file will create a user authorization table and a contact table used to store synchronized contact data including some sample data. At least configure **sync•gw** to use this data base.

During synchronization session, please user the sample user ID “syncgw” with password “syncgw” as credential on your [CLIENT].

You may check the interface definition files `DBAdmin.php`, `DBintHandler.php` and `DBextHandler.php` located in directory `syncgw/interfaces` which contains commented definitions of all classes and methods you need to implement in your own data base handler.

Now we’re ready to start.

First, we create your own bundle. This directory is automatically discovered by **sync•gw** during startup. **sync•gw** expects to find at least two files in this directory. The file `Admin.php` contains the administrator interface class and is only used during calls to [BROWSER]; the second file `Handler.php` contains the [CONNECTOR] class definitions and is used especially during **sync•gw** synchronization.

7.1 Administrator interface class

For the administrator interface class, we need to create PHP file `Admin.php` in `myapp` directory. This file contains a class definition including all class methods required by [BROWSER].

7.1.1 Class definition: Admin

```php
<?php
declare(strict_types=1);

/*
 *    Administration interface handler
 *
 *    @package    sync*gw
 *    @subpackage myApp handler
 *    @copyright  (c) 2008 - 2023 Florian Daeumling, Germany. All right reserved
 *    @license    LGPL-3.0-or-later
 */

namespace syncgw\interfaces\myapp;

use syncgw\interfaces\DBAdmin;
use syncgw\gui\guiHandler;
use syncgw\lib\Config;
use syncgw\lib\DataStore;
use syncgw\lib\Cnnfig;

class Admin extends \syncgw\interfaces\mysql\Admin implements DBAdmin {
}

?>
```

Table 1: Admin class definition

We define our own class which is based on the MySQL class definition and implements the `Admin` class. Using this approach, we can use all the provided MySQL class methods and only need to add our own code to handle special `myApp` configuration parameter.

7.1.2 Admin::getInstance()

```
/**
*  Get class instance handler
*
*  @return - Class object
*/
static function getInstance(): Admin {

   if (!self::$_obj)
      self::$_obj = new self();

   return self::$_obj;
}
```

Table 2: Admin::getObj ()

This function is called internally to allocate the object for this class. This reduces overall amount of allocated memory and boosts execution of class object.

7.1.3 Admin::getParms()

```
/**
 *    Show/get installation parameter
 */
public function getParms(): void {

   $gui = GUI_Handler::getObj();
   $cnf = Config::getObj();

   GuiHandler::getInstance()->putQBox('Please enter message which will be used '.
                 'during execution',
                 '<input name="MyParm" type="text" size="40" maxlength="250" value="'.
                 $cnf->getVar('Usr_Parm').'" />', 'MyApp dummy message.', FALSE);

   parent::getParms();
}
```

Table 3: Admin::getParms ()

This method is called to show any required parameter in [BROWSER]. In the `Admin` class, this method asks for all parameters required to connect to data base.

To avoid any conflicts with **sync•gw** or [BROWSER] parameter please add the prefix `Usr_` to any configuration parameter you want to use.

7.1.4 Admin::Connect()

```
/**
 *    Connect to handler
 *
 *    @return - true = ok; false = Error
*/
public function Connect(): bool {

   $gui = GUI_Handler::getObj();
   $cnf = Config::getObj();

   // connection already established?
   if ($cnf->getVar(Config::DATABASE))
      return true;

   if ($v = $gui->getVar('MyParm'))
      $cnf->updVar('Usr_Parm', $v);

   // create our own tables
   $cmds = parent::loadSQL(Util::mkPath('interfaces/myapp').'/tables.sql');
   parent::mkTable($cmds);

   return parent::Connect();
}
```
Table 4: Admin::Connect ()

This method is called to perform [CONNECTOR] installation. In the `Admin` class, this method creates all required internal **sync•gw** MySQL tables.

We want to store a [CONNECTOR] dummy configuration parameter. For this purpose [BROWSER] provide a method `updVar()` to store configuration parameter in [CONFIG] file.

To avoid any conflicts with **sync•gw** or [BROWSER] parameter please add the prefix `Usr_` to any configuration parameter you want to use.

At least we call parents `Connect()` method to ensure all required MySQL parameter are saved.

7.1.5 Admin::DisConnect()

```
/**
 *    Disconnect from handler
 *
 *    @return - TRUE=ok; FALSE=Error
 */
public function DisConnect(): bool {

   // remove parameter
   $cnf = Config::getObj();
   $cnf->updVar('Usr_Parm', '');

   // delete our own tables
   $cmds = parent::loadSQL('assets/myapp/tables.sql');
   parent::delTable($cmds);

   return parent::DisConnect();
}
```

Table 5: Admin::DisConnect ()

This method is called to perform the de-installation of [CONNECTOR] as soon as “*Drop data base connection*” in [BROWSER] is selected.

We clear configuration parameter to know our handler is not available anymore and calls parents `DisConnect()` method to ensure MySQL handler is de-installed properly.

7.1.6 Admin::SupportedHandlers()

```
/**
 *    Return list of supported data store handler
 *
 *    @return - Bit map of supported data store handler
 */
public function SupportedHandlers(): int {

   return DataStore::EXT| DataStore::NOTE;
}
```

Table 6: Admin::SuppoetedHandlers ()

This method is called to provide information about which [DATASTORE] are supported by [CONNECTOR]. Other [DATASTORE] not returned by this method are not available for synchronization or available in [BROWSER].

In `myApp` [CONNECTOR] any client request to synchronize [DATASTORE] except notes data will be rejected with an appropriate return code.

If [CONNECTOR] should support more [DATASTORE], IDs of the supported [DATASTORE] needs to concatenated using a bitwise OR (e.g. `DataStore::CONTACT|DataStore::CALENDAR`). The parameter `DataStore::EXT` specifies this data base handler supports external records.

For more information about the available handlers please refer to file `core-bundle/lib/DataStore.php`.

7.2 [CONNECTOR] handler class Handler

```
<?php
declare(strict_types=1);

/*
 *    Data base handler class
 *
 *    @package    sync*gw
 *    @subpackage myApp handler
*     @copyright  (c) 2008 - 2023 Florian Daeumling, Germany. All right reserved
 *    @license    LGPL-3.0-or-later
  */

namespace syncgw\interfaces\myapp;

use syncgw\interfaces\DBextHandler;
use syncgw\lib\Config;
use syncgw\lib\Log;
use syncgw\lib\Server;
use syncgw\lib\DataStore;
use syncgw\lib\User;
use syncgw\lib\XML;

require_once('Notes.php');

class Handler extends \syncgw\interfaces\mysql\Handler implements DBextHandler {
}

?>
```

Table 7: Handler.php skeleton

Please create a new PHP file `your-bundle/src/Handler.php`.

Make sure the `Notes` class definition is included at the top of the file. This is required because the class is not automatically loaded.

7.2.1 Handler::getInstance()

```
/**
 * Get class instance handler
 *
 * @return - Class object or NULL
 */
public static function getInstance() {
}
```

Table 8: Handler::_start ()

- We check if we're responsible.
- We define a message we want to use in our [CONNECTOR] and load the configuration parameter we defined in [BROWSER] configuration panel.
- We load parameter required to connect to data base and open a connection.
- We create our notes data store handler object `$_hd`.
- We register our shutdown function `delInstance()`.

At the end of the method we send a message about the configuration parameter to **sync•gw** log.

7.2.2 Handler::delInstance()

```
/**
 *    Shutdown function
 */
public function delInstance (): void {
}
```

Table 9: Handler::_stop()

In the destructor method you may specify any “shutdown” code to stop the interface to [APP] [DB].

We need to delete reference object to enable call to `getInstance()` on next allocation of object.

7.2.3 Handler::Authorize()

```
/**
 *    Authorize user in external data base
 *
 *    @param   - User name
 *    @param   - Host name
 *    @param   - User password
 *    @return  - true = Ok; false = Not authorized
 */
public function Authorize(string $user, string $host, string $passwd): bool {

   // get password
   if (!($obj = $this->_db->query('SELECT `password`, `id` FROM `myapp_usertable` '.
                                  'WHERE `username` = "'.$user.'"')))
      return false;

   // get data
   $rec = $obj->fetch_assoc();

   // user not found?
   if (!isset($rec['password']))
      return false;

   // check password
   if (strcmp($passwd, $rec['password']))
      return false;

   // save user id
   $this->_uid = intval($rec['id']);

   // load internal user object
   $usr = User::getObj();

   return $usr->loadUsr($user, $host);}
}
```

Table 10: Handler::Authorize()

This method is called to authorize user connecting from [CLIENT] trying to initiate a synchronization session.

Please note this authentication check is required, because **sync•gw** itself does not maintain any authorization scheme.

7.2.4 Handler::Query()

```
/**
 *    Perform query on external data base
 *
 *    @param   - Handler ID
 *    @param   - Query command:
 *            DataStore::ADD   Add record                             $parm= XML object
 *            DataStore::UPD   Update record                          $parm= XML object
 *            DataStore::DEL   Delete record or group                 $parm= GUID
 *                             (including sub-records)
 *            DataStore::RGID  Read single record                     $parm= GUID
 *            DataStore::GRPS  Read all group records                 $parm= None
 *            DataStore::RIDS  Read all records in group              $parm= Group ID or ''
 *                                                                    for record in base group
 *    @return  - According  to input parameter
 *            DataStore::ADD   New record ID or false  on error
 *            DataStore::UPD   true=Ok; false =Error
 *            DataStore::DEL   true =Ok; false =Error
 *            DataStore::RGID  XML object; false=Error
 *            DataStore::GRPS  [ "GUID" => Typ of record ]
 *            DataStore::RIDS  [ "GUID" => Typ of record ]
 */
public function Query(int $hid, int $cmd, $parm = NULL) {

   // we don't serve internal calls
   if (!($hid & DataStore::EXT))
      return parent::Query($hid, $cmd, $parm);

   // user ID set?
   // check handler called
   if ($this->_uid == -1 || !($hid & DataStore::EXT|DataStore::NOTE))
      return $cmd & (DataStore::RIDS|DataStore::RNOK) ? [] : FALSE;

   // perform query
   return $this->_hd->Query($this->_db, $this->_uid, $cmd, $parm);
}
```

Table 11: Handler::extQuery()

Method `Query()` is called each time **sync•gw** recognize any need to add, update or delete an external [RECORD] or to discover record status on any [RECORD] in [DB]. All internal call were redirected to MySQL handler.

The first parameter `$hid` is the [DATASTORE] handler ID to be accessed.

Parameter `$cmd` is the command to perform on [DB].

Please note this method must take care about some special conditions:

- For command `DataStore::GRPS` and `DataStore::RGIDS` method must always return the groups at top of `array()`.
- [GROUP] `array()` consists of all assign external record IDs.
- External default [GROUP] must be created and maintained by [APP].

7.2.4.1 Class definition: Notes

```
<?php
declare(strict_types=1);

/*
 *    Notes data store handler class
 *
 *    @package    sync*gw
 *    @subpackage myApp handler
 *    @copyright  (c) 2008 - 2023 Florian Daeumling, Germany. All right reserved
 *    @license    https://github.com/Toteph42/syncgw/blob/master/LICENSE
 */ LGPL-3.0-or-later

namespace syncgw\interfaces\myapp;

use syncgw\lib\DB;
use syncgw\lib\DataStore;
use syncgw\lib\XML;
use syncgw\document\field\FieldSummary;
use syncgw\document\field\FieldBody;
use syncgw\document\field\FieldCategories;

class Notes extends XML {


}

?>
```

Table 12: Notes.php skeleton

We define our class as child of the **sync•gw** `XML` class. The `XML` class provides a couple of methods we will use. Class definition is in `your-bundle/src/Notes.php`.

7.2.5 Field mapping

```
const MAP  = [
        'title'          => FieldSummary::TAG,
        'text'           => FieldBody::TAG,
        'cats’           => FieldCategories::TAG,
];
```

Table 13: Mapping table

Starting with version 9 **sync•gw** uses its own data model. We decided to make this step forward establish clear responsibilities: From this version on, syntax check, data validation and input/output and any other field activity is in “responsibility” of each field object. If you have a field `title` and want to store in **sync•gw** data model, you check directory `core-bundle/src/documents/fields`. There all **sync•gw** supported field objects are stored.

In our handler we decided to store `title` in `FieldSummary`. Calling this object in standardized way and the object is responsible to store data within **sync•gw** and provide output in all supported protocols.

The `MAP` constant array is used to define the mapping between the internal MySQL data fields used and **sync•gw** fields.

7.2.6 Notes::Query()

```
/**
 *    Perform query on external data base
 */
public function Query(\mysqli &$db, int $uid, int $cmd, $parm) {

   …
   return $out;
}
```

Table 14: Notes::Query()

sync●gw use this method to access all external [RECORD] in [DB]. It is called by handler.

Chapter 8: User configuration options

As user performs first synchronization, a `User` object is been allocated in user data store. Within this object there are some special variables which you can use to modify synchronization behavior.

Name	Description	Options
`<Banned/>`	Allow user login	0 – login allowed (default) 1 – user is banned
`<MailSend>`	User is allowed to send mails for ActiveSync	0 – not allowed 1 – allowed (default)
`<AccountName/>`	Friendly user account name for ActiveSync	
`<DisplayName/>`	User display name in directory service for ActiveSync	
`<UserDisplayName/>`	User display name associated with the given account for ActiveSync	
`<OutOfOffice/>`	Out of office message for ActiveSync.	
`<State/>`	Status of property	0 – The property is disabled 1 – The property is enabled
`<Time/>`	Time slot definition for out-of-office	"Unix start time"./."Unix end time" or NULL for global property
`<Message>`	Message to return	
`<Audience/>`	Audience	1 – Internal 2 – Known external user 3 – Unknown external user
`<Text TYP="Text"/>`	Clear message text	
`<Text TYP="HTML"/>`	HTML message text	
`</Message>`		
`</OutOfOffice>`		
`</FreeBusy>`	Free busy array	
`<Slot/>`	Time slot definition. If nothing is specified, all time slots were of type "0"	"Unix start time/Unix end time/Type (see below)" 0 – Free 1 – Tentative 2 – Busy 3 – Out of Office (OOF) 4 – No data
`</FreeBusy>`		

In file `syncgw/activesync-bundle/assets/masPolicy.xml` you may specify server policy settings for ActiveSync. Please check all available comments in file.

In file `syncgw/activesync-bundle/assets/masRights.xml` you may specify server right management settings for ActiveSync. Please check all available comments in file.

Appendix A: External [CONNECTOR] handler testing

There is a plug-in available to test data base synchronization.

Please open [BROWSER] and configure user “debug” with password “syncgw” as “*Debug user ID*” and “*Debug user ID password*”.

Open administrator interface panel select “*Explore data*” and click on “*Run*”. Switch into a [DATASTORE] and select a group. Then one additional button is available:

Sync	Synchronize internal [DATASTORE] with external [DB].

Appendix B: sync•gw document data structure

sync•gw use its own data representation based on XML objects. These documents all have basically the same data structure.

```
<syncgw>
  <GUID>C2</GUID>
  <LUID>1243</LUID>
  <SyncStat>OK</SyncStat>
  <Type>R</Type>
  <Group>C1</Group>
  <Created>1399194107</Created>
  <LastMod>1399194107</LastMod>
  <CRC>9388af2</CRC>
  <extID>R1243</extID>
  <extGroup>G8387</extGroup>
  <Data>
      …
  </Data>
</syncgw>
```

Table 15: Document data structure

`<GUID>`	[GUID] on sync•gw.
`<LUID>`	[LUID] on [CLIENT].

`<SyncStat>`	Status of document. Allowed statuses are the value of the constants
	`DataStore::STAT_OK`
	Synchronization performed successfully
	`DataStore::STAT_ADD`
	[RECORD] need to be send to client device)
	`DataStore::STAT_REP`
	[RECORD] need to be replaced on client device)
	`DataStore::STAT_DEL`
	[RECORD] to be deleted on client device

`<Type>`	Available [RECORD] types are defined in `syncgw/lib/DB.php`. For user [RECORD] processing the relevant record types are:
	`DataStore::GROUP`
	Record is a [GROUP] record. Please note [GROUP] records are only allowed to be member of ONE parent group
	`DataStore::DGROUP`
	Special default [GROUP]
	`DataStore::DATA`
	Data record

`<Group>`	[GROUP] to which <GUID> is assigned.
`<Created>`	When [RECORD] was created - as UNIX time stamp.
`<LastMod>`	Last date and time [RECORD] was modified - as UNIX time stamp.
`<CRC>`	Unique record hash.
`<extID>`	Record id in external data base.
`<extGroup>`	External [GROUP] to which <extID> is assigned to.
`<Data>`	Data assigned to this document. The content depends on the [DATASTORE].

Appendix C: [BROWSER] customization

[BROWSER] use two skeleton files to display the web interface. Feel free to modify these files as required.

`syncgw/gui-bundle/assets/login.html`	Login page
`syncgw/gui-bundle/assets/interface.html`	**sync●gw** administrator interface

Appendix E: Character encoding

sync•gw use internally the UTF-8-character set. During synchronization with [CLIENT], all user data is automatically converted to the character set supported by [CLIENT]. If you want to know more about the supported character sets, please open [BROWSER] and select "*Check status*". Please search for "*Encoding handler*". Below the header line all supported character sets are listed.

```
/**
 *    Get external encoding
 *
 *    @return  - Active character set name
 */
public function getEncoding(): string {
}
```

Table 16: Encoding::getEncoding()

You may use this method to get information about which character set is in use. The method returns the name of the external character set. For more information look at `syncgw/core-bundle/assets /charset.xml` which contains all available and supported character sets.

```
/**
 *    Set external character set encoding
 *
 *    @param   - Character set ID or name
 *    @return  - Name of character set or ''
 */
public function setEncoding(string $cs): string {
}
```

Table 17: Encoding::setEncoding()

If you use a different character encoding in [APP], you may use this function to set either the character set ID (e.g. `1017` for `UTF-32`) or the name of the character set name (e.g. `US-ASCII`).

If the requested character set is not supported by PHP, a `Null` value is returned.

```
/**
 *    Encode data from internal to external encoding
 *
 *    @param   - String to encode
 *    @return  - Converted string
 */
public function export(string $str): string {
}
```

Table 18: Encoding::export()

```
/**
 *    Decode data from external to internal encoding
 *
 *    @param   - String to decode
 *    @return  - Converted string
 */
public function import(string $str): string {
}
```

Table 19: Encoding::import()

Please use these two methods to either convert a string from internal to external character set (or back).

Appendix F: Base class XML

sync•gw uses its own implementation of `PHP DOM XML` class. We decided to create our own wrapper class to make access to XML object easier and more auditable.

Appendix G: Internal configuration parameter

This chapter contains a detailed description of all available internal configuration options used in [CONFIG] file. You can only modify them in [CONFIG] file.

Max. PHP execution time (Parameter name: `MaxExecutionTime`)

This sets the maximum time in seconds a script is allowed to run before it is terminated by the parser. This helps prevent poorly written scripts from tying up the server. Defaults to 910 seconds (more information).

Default timeout for socket based streams (Parameter name: `SocketTimeout`)

Default timeout (in seconds) for socket based streams. Specifying a negative value means an infinite timeout (more information).

MySQL record size (Parameter name: `MySQLSize`)

Attachments may be very big. Sometimes MySQL has a limitation of record size which let's **sync•gw** crash. This parameter limits the size of attachments. Default is **10485760**.

MySQL retry counter (Parameter name: `MySQLRetry`)

In shared hosting environment it might happen that the MySQL server fails with error "`[2006] MySQL server has gone away`". To recover from this error, **sync•gw** waits for 300 milliseconds and tried SQL action again. These retries were limited by this parameter (defaults to **10** times).

Temporary directory (Parameter name: `TmpDir`)

Where to store temporary used files. On some hosting provider, you may encounter problems with temporary files, since the value returned by sys_get_temp_dir() is unusable.

Default system time zone (Parameter name: `Timezone`)

Time zone to use.

Maximum bytes send in one chunk (Parameter name: `SendSize`)

Defaults to 1 MB.

Force WebDAV task list synchronization (Parameter name: `TaskDAV`)

If you use CalDAV to synchronize data and you want to synchronize tasks, then you need to set this parameter to “FORCE”. Please be aware you cannot synchronize calendar and tasks from same installation.

Directory where RoundCube is installed (Parameter name: `RCDirectory`)

Normally, you should install **sync•gw** in sub directory of RoundCube. If your RoundCube installation is installed in a different directory, you may specify the path with this configuration variable.

Set SMTP debug messages level (Parameter name: `SMTPDebug`)

Set this parameter to get additional messages from PHPMAILER handler. Defaults to **0**.

0: No output
1: Client messages
2: Client and server messages
3: As SERVER plus connection status
4: Noisy, low-level data output, rarely needed

Throw external exceptions in PHPMAILER (Parameter name: `MailerError`)

Set this parameter to **1** to allow PHPMAILER to throw external exceptions. These exceptions were then catched by **sync•gw** and written to log file. Defaults to **0**.

Connection test timeout (Parameter name: `ConnectionTimeout`)

This is the SMEP connect test timeout. It is set to **5** seconds by default.

END OF DOCUMENT