efaCloud – efa2 in der Wolke

A programmers guide
issued: 08.03.2023

www.efacloud.org

Table of Contents

Foreword

This programmers guide for the efaCloud client modules and server application is the reference manual to maintain that application part. It will cover the server PHP and Javascript code. The efa2 program is used as client.

This manual is written in English for convenience, because all programming code is commented in English to avoid trouble with non ASCII characters. But the program itself is targetted to be delivered in German. So you will find German text within the manual.

efaCloud uses a framework which was developed for different purposes by myself. Its documentation is separately provided in a tfyh.org PHP framework description.

Bonn, March 2023

Martin Glade

Introduction

efaCloud is meant to provide an efa2 logbook accessibility from anywhere. Its focus is the server side data base and interface to the standard efa2 client, boathouse or main. From there it has developed to provide an efaCloud administration application (PHP) and an efaWeb client application (Javascript).

To use it, just select the storage type efaCloud. This will extend your local storage to the cloud. The local XML storage is used in the efaCloud configuration for caching and temporary offline operation.

Envisaged usage scenario will be one to three boathouses with a efa client running e. g. on a Raspberry together with some administrator running the administrative tasks from home PC using the efa client or the efaCloud server application for this purpose. Some users may enter trips via smartphone or similar.

Therefore a server based web administration application is also part of efaCloud. Use it alternatively for profile changes in boats, persons, reservations or similar. It is built on a MySQL database and plain PHP code to access the data. No further application server framework is used.

A Javascript application is added for web based logbook usage which connects to efaCloud via the API build for the efa-connnection. It is completely separate from the efaCloud code except the menu and access management.

Programming concepts are described here as well as communication between client and server to allow for maintenance of the software.

The following description focuses on four topics:

- the data layer, i. e. the efa tables and extra server tables,
- the network layer of the API,
- the API message protocol,
- the efaCloud administration application
- the efaWeb logbook application.

Naming conventions

A few terms need to be clarified in advance, even if it is dry. What is meant here in the manual by: efa, efaCloud, efaWeb and user.

Please be sure to read this paragraph and remember the terms. If you can't distinguish between the terms, you will have a hard time understanding this manual.

efa2

efa2 is the PC programme which you can download from nmichael.de and install on any PC. It has four operating modes:

1. **efa-Bootshaus**: this is the full screen mode that everyone knows from entering the trips at the boathouse. It can be used without logging in. It is used to handle the sports operations: Enter and check sessions, news, boat damage and boat reservations.
2. **efa-Base**: the administration mode in a normal window, used to modify master data: Boats, persons, destinations, crews, logbooks and the like. It requires a login as "Admin".
3. **efa-BootshausAdmin**: it is possible to log in as Admin in the efa-Bootshaus mode. Then the same functions are available as in efa-Base, except that the menu is accessible via a mask instead of the window header. Because it is identical to efa-Base, it is not referenced in more detail.
4. **efa-Remote**: this is the mode in which one addresses the logbook programme from a remote location. This mode is built for administrative purposes and thus cannot be used in a meaningful way in parallel with efaCloud. The efa-Remote mode cannot be used with efaCloud.

The efa-PC in this user manual is the PC on which efa is installed. It is usually located in the boathouse.

efaCloud and efaWeb

efaCloud is a programme that is installed and run on a computer on the Internet.

- efaCloud-programme means the software that is installed and executed.
- efaCloud-Server means the computer on the network on which efaCloud Program is installed and executed.
- efaCloud-client means the active logbook program that exchanges data with the efaCloud server.
- efaCloud is the user interface of the efaCloud program for managing the data, similar to what efa-Base allows for efa.
- efaWeb refers to a user interface that replicates the functions of the efa-Boathouse mode and is delivered and installed as part of the efaCloud program.

efaWeb is an efaCloud client, just like efa-Bootshaus and efa-Base. But efa-Remote cannot do that. Clear? If not, you should read the definitions again.

User

In the boathouse, the use of efa is anonymous, i.e. it requires no registration, no "login". That is practical, but the possibilities must then be limited to the messages of the sports operations.

- **efa-Super-Admin**: the efa programme knows an efa-Super-Admin with the name "admin". With this role you can do everything on the efa-PC: install, configure, set up other efa-admins and any form of data change. The efa-Super-Admin is always set up and managed locally in the efa-PC. efaCloud does not know the efa-Super-Admin.
- **efa-Admin**: efa requires a login in both efa-Base and efa-BootshausAdmin mode. The non-anonymous users of efa are called efa-Admins. They are assigned permissions for the respective efa-PC. This is traditionally done in efa, but should be done in efaCloud when using efaCloud by assigning an admin name to the efaCloud users (see below).
- **efaCloud-User**: The use of efaCloud always requires a login. The users are called efaCloud users. Their authorisation is assigned via roles. They become an efa admin at the same time by being assigned an efa admin name. They can then log on to any connected efa-PC with this efa-Admin name. To do this, however, the efa permissions must be assigned in efaCloud.

  Technical efaCloud-User: every efa-PC connected to efaCloud is at the same time an efaCloud user, because it has to identify itself (authenticate) when establishing the connection. Nevertheless, the user can be anonymous on the efa-PC itself.
- **efaWeb usage**: The use of efaWeb always requires a login. Each efaCloud user can use efaWeb according to his role, there are no dedicated efaWeb users. However, for the use of efaWeb, an efaCloud user can be assigned additional differentiated rights that go beyond the role definition.

Program architecture

EfaCloud utilizes the tfyh.org PHP class framework which also defines the program architecture. Please see the corresponding manuals for details. We will just give a short overview here.

PHP-files hierarchy

Here is a two level directory providing all PHP files. The level 1 declares the function: pages/, classes/, forms/ etc. because all PHP-code sits on that level, a relative link to e.g. the efa_tables class file is always "../classes/efa_tables.php", regardless from where you start.

Obviously some are accessable from the web, some not. Class files and configuration files are never accessible to ensure that the execution logic is enforced.

The directory config/ has two levels of subdirectories, as has the directory logs/. Both do never carry PHP code.

Configuration

Different levels of configuration allow flexible framework use. Application configuration such as form layouts, predefined queries (named lists), menu definitions including the access rights are part of the application package and cannot be changed. Other parts can be changed by the application administrator.

Standard classes

Standard class files and names always start with the tfyh_-prefix to be easily spotted in the classes directory. They provide the functions for data base access, form creation and handling, user management, configuration management, data base access, session management and security, cron jobs etc.

Specific efa classes

The classes which implement efa-specific data handling are usually named efa_****.php.

Standard forms, pages

Standard forms are there for user login, user management, mail sending, configuration management, asf. Standard pages include error display, data base structure and user rights display, list display as sortable table asf.

Deployment

Deployment is simply downloading the PHP-zip package and unpacking it into the web server file tree. Configuration files are cached, if existing. If not, default configurations are used being part of the zip package.

After deployment the init_version.php “class file” is called to execute avtivities which may only have been provided with this update. Not the the loaded classes Tfyh_socket and Tfyh_toolbox will remain from the previous version, because they have been loaded before the update and will not be refreshed.

efaCloud data

efaCloud data are first of all efa2 data. It is therefore helpful to understand the efa2 data structure in order to understand the exchange of information between the client and the server.

For server side administrators and client identification efaCloud uses additional tables which are not visible to the client side.

Common efa2 and efaCloud tables

efaCloud uses the 17 efa2 data tables. These tables are identical in structure on both sides, except one to two additional fields on the server side (see below).

They are stored in the web server's data base, but they are also cached in the client for temporary offline usage and fast reboot. Local storage at the client side uses the standard XML-format without the above mentioned server side add on fields.

Again: this is for data tables only. Project or client configurations, client admins asf. are not part of the efa2 tables and efaCloud and only stores them as files and for one client as reference. They are not synchronized over the different efa-clients.

Efa2 table names in the efaCloud data base are their XML-file extensions or storage types like "efa2persons". The current logbook table name, however, is always "efa2ogbook", and not the logbook name given in the efa2 project.

Additional server side data fields for synchronisation

For the purpose of synchronization all server side tables have an additional 'LastModification' field. A 'ClientSideKey' field is added to the tables which allow fixing of keys, i. e. for efa2logbook, efa2messages, efa2boatdamages, and efa2boatreservations.

The 'LastModification' is a small text field (8 characters) to be set to "inserted", "updated", or "deleted". The 'ClientSideKey' (64 characters) holds the mismatching local data key formatted as "<clientID>:<dataKey1>[|<dataKey2>]". Note: There is never more than one mismatching client side data key, because the mismatching data record is inserted but once.

Additional server side data field for logbook handling

Efa uses a new logbook table every year. On the server side, there is just one efa2logbook table, but it holds an additional field: the LogbookName. Each trip has not only the EntryId, but also a logbook name with it to become unique. Therefore any change of active logbook at the client side does not affect the server.

The client adds the current logbook name as last field of the efa2logbook record at the time of compiling the transaction. The server removes the field when returning logbook data to the client. So it is only visible in transaction requests.

Data keys and validity periods

Efa internally uses data keys with one or two fields to identify a data record. One of those is either a GUID identifier or a numeric key. Tables which hold a validity period entry are called versionized – for those the ValidFrom timestamp is part of the key.

In many cases the data key is however reflected in a single field. Those key columns are set AUTOINCREMENT (Logbook, Message) or UNIQUE (others) in the respective efaCloud tables. Tables with a single data key field are:

1. AutoIncrement: “Sequence”
2. BoatStatus: “BoatId”
3. Clubwork, Crews, SessionGroups, Statistics, Status, Waters: “Id”
4. Fahrtenabzeichen: “PersonId”
5. Logbook: “EntryId”
6. Messages: “MessageId”

Two fields are used for the following data tables:

7. Versionized: Boats, Destinations, Groups, Persons: “Id”, “ValidFrom”
8. BoatDamages: “BoatId”, “Damage”
9. BoatReservations: “BoatId”, “Reservation”

and for the three tables used for configuration purposes in efa, which are not copied to the server:

10. Project: “Type”, “Name”
11. Admins: “Name”
12. Config: “Category”, “Type”

Fields named “Id” carry a 36 character GUID without curly braces, including “BoatId” and “PersonId”. The latter refer to an “Id” of the respective boats and persons tables.

Versionized tables provide a the validity period actually as a doublet of “ValidForm” and “InvalidFrom” fields. The efa client handles overlaps and gaps in such periods. efaCloud does no period checking.

efaCloud record management

Before efaCloud the record creation and manipulation was fully managed by the client. This includes the records key definition. Using more than one client, this does create a considerable challenge in particular in combination with numeric autoincrementing data keys. The more clients you add, the more probable it becomes for conflicts to arise.

With efaCloud record management, up to four more data fields are added to each common table to move the record key management to the server side: a random efaCloud record Id (ecrid), the record owner (ecrown), a record history (ecrhis) and an “additional copy to” field (ecract).

The usual efa-PC will continue not to be aware of this and still run smoothly with efaCloud, since they are filtered out in data read statements.

The use cases for record management are: is coupling two efaCloud servers for rowing club cooperations where trips are copied to a second server site. Then this record management will be necessary.

efaCloud record Id

The efaCloud record Id is meant to identify a record as unique over all efaCloud Server installations. A UUID would do, but the key will be used in lists, so a shorter key is used: we take 9 bytes created by the standard “openssl_random_pseudo_bytes” PHP function map three times three of them into a four character base64 type sequence like for the transaction container, i. e. with the following special characters: “-” instead of “/” and “*” instead of “+” (e.g. “Z6K3mpORQ5y6”). As such it will not need any URL encoding (cf. Transactions container format). The ecrid thus looks like a 12 character base64 String.

Thus $2^{72} = 4{,}7 * 10^{21}$ different ecrid values are possible. The probability of getting two identical ones is appr. $2 * 10^{-14}$, if we assume 1000 efa server installations with 100.000 records each. This is far less probable than winning the jackpot whereever.

A client may also create efaCloud record Ids, but then it must use the API version 3 to make sure that these are transported with all records exchanged.

Only for efa2tables

efaCloud tables do not have an efaCloud record Id. They use instead an ID field which is the primary unique and autoincremented integer key. Their entries have no requirement to be cross-server unique and therefore do not need an additional key field.

Added upon insert and update

The ecrid key will always be created, if a record shall be inserted or updated which has an ecrid data field with a value of NULL.

The select transaction and the synchronisation process is used to trigger the generation of missing ecrids, e. g. due to a server program version upgrade which will create the ecrid columns, but not the values (that would take too long for a single PHP-page script execution).

String encryption based ecrid creation

The efaCloud record Id is generated using the PHP function “openssl_random_pseudo_bytes” yielding completely random ecrids.

Handling two keys

Yet another key does not solve a problem, if the precedence is not clear. A client requesting a data record modification SHALL provide the record with all key fields it has available. These may include an ecrid field, or not.

Here is what the server will do on whether there is an ecrid or not:

1. The record or key within the modification request contains NO VALID ecrid field: the efa client keys are used to identify a record. For logbook records the EntryId + Logbookname is used. Keys may be modified, if duplicate and then keyfixing shall be used (see Fix mismatching keys).
2. The record or key within the modification request contains A VALID valid ecrid field:

   a) the server has efaCloud record management enabled:

   - The record which shall be modified is identified using the ecrid value.
   - Uniqueness of the efa data key is not enforced.

    - If a key field of a table which allows for key fixing is left empty, the next available value (e.g. the EntryId for a trip) is set as key field.

b) the server has efaCloud record management disabled:
this scenario is not supported. All such modification requests are rejected.

Record owner and history

When switching to the server side record management, not only the ecrid, but three server side record data fields are added to each efa table record:

- Record owner (ecrown, integer): this is the efaCloudUserId of the client which created the record. It will be empty, if the record has been created at a time when efaCloud record management was disabled.
- Record history (ecrhis, text): this is a history of changes to the record. Details will come later.

Selecting records for the client

When a client requests via “select” records to be provided to it, it shall not be bothered with the specific efaCloud record management data fields, if it is not using the efaCloud record management itself.

Therefore see API versions for details how to ensure a proper “select” response.

Obsolete with version 2.3.1: In 2.3.0 versions the efaCloudUser has a flag to configure whether it gets the efaCloud record management data fields or not.

One sided tables

Not all tables are shared, since client and server application require different and specific configuration. Note that the client and server users are not linked, nor their rights synchronized. This may be improved in later releases.

Efa client only tables

Three tables with the efa client are not synchronized with the server:

- efa2admins: the local administrators.
- efa2config: the local client configuration
- efa2project: the local project configuration for the client

efaCloud server only tables

The efaCloud data base has two more tables, which do not occur in efa2:

- efaCloudUsers: the list of members who can access efaCloud, both persons and api-users.
- efaCloudLog: a log of all recent changes in the efaCloud data base.
- efaCloudArchived: the archived efa-Records, all flat in a list, all json_encoded and all without version history.
- efaCloudTrash: the deleted efa-Records, all flat in a list, all json_encoded.

These tables are not shared with the client and use a single numeric key named ID. They are not versionized. They do not have an ecrid.

Data base layout reference

The data base layout is reference within the “efa_db_layout.php” file in the classes section. It contains all layout information for all tables of all versions.

Please be aware that data base layout versions are not the same as API versions.

The layout information is compiled in a spreadsheet document which is used to create the class file. Over time we had till 2.3.2_11 ten data base layout versions. They are filed in the ‘/config/db_layout/’ folder. The efa_tools.php class file has the functions to manage the layout.

API network layer

The efa client is meant to run online using efaCloud. It will continue to run when offline, but offline operation is only intended to cover temporary network instabilities as occur in wireless environments. Write operations to the server will be queued in memory until timeout. Then retry mechanisms apply using permanent storage.

Shutting down efa before all transactions have been completed at the server side will cause data loss and may cause temporary inconsistencies between client and server.

The client host may go to standby and tear down the internet connectivity, if no transaction is pending. Since transactions time out, this will most probably not happen anyway.

The network usage is limited to user triggered actions. You can safely block all incoming connections at the client side router.

Connection security

efaCloud makes use of https based security, and authenticates the client for each transaction separately. There is no client side session management. **You MUST use https as security layer, the protocol provides no encryption nor key management to handle sessions. The credentials are sent with every transaction.**

The efaCloudUserID and password must be changed manually. There is no automatic password update procedure or password lifetime limit.

Disabling a client

If for any reason a client shall be blocked, it is sufficient to change the password at the efaCloud server. This will immediately block all coming transactions, because each transaction is separately authorised.

Client server handshake

POST requests must be issued to /api/posttx.php to push or pull information to / from the server. This allows for a larger amount of data to be transferred than using a GET request and will never show up in any browser address field.

All transaction information will be contained in a single URI-encoded encoded transaction container using the POST parameter “txc” (transaction container). URI-encoding is according RFC3986.

Transaction containers consist of a set of transaction requests or responses, see definitions in section ‘API transaction format’.

Timeout and retry

A transaction container times out at the client side after 30 seconds without a response. That moves the queues to a disconnected state. A retry will be triggered regularly. Upon the retry trigger all transactions from the busy queue are again sent to the server after incrementing their retry counter.

For details see section ‘On and off scenarios‘.

API state machine

The API has different states which shall be handled by the client. Once the client activates the efaCloud feature, the API transaction queues start working. User requests, automatic triggers or failures result in state machine changes.

On and off scenarios

The full state machine diagram is shown in Figure 1. The following on/off scenarios are envisaged.

efaCloud feature is activated

The efaCloud feature is either manually activated or by the program starting in efaCloud configuration. Manual activation clears the permanent queues. If automatically started, permanent transaction queues are read from disk.

The queue handler is constructed and started. The queues will go to AUTHENTICATING state. In this state only ‘nop’ requests can be appended to the pending queue. Such a ‘nop’ request is automatically appended to prove the connection and credentials.

Authentication success will change the state to WORKING. All transaction failures in AUTHENTICATING mode will immediately trigger efaCloud deactivation.

Transaction queues go online

Transaction queues go online either by successful authentication or be successful reconnecting in DISCONNECTED state.

The transaction pending queue will then accept transactions of all types.

The event will set the queues to WORKING state. This state is called IDLE, when the queues are empty.

Transactions queues go to synchronization

The transaction queues go synchronization either by manual or periodic trigger.

The pending transactions queue is suspended and all transactions enter the synching queue before being processed in the busy queue. Details see section ‘Data synchronization’.

The event will set the state to SYNCHRONIZING. Either when a transaction error occurs, or when the synchronization cycle is completed, The state falls back to WORKING/IDLE.

Transactions queues go offline

The transaction queues go offline either by being manually paused or a time out occurrence.

The busy and pending transaction queues will then drop all but the insert, update, delete transactions. The synching queue will drop all but the keyfixing confirmations, i. e. keyfixing transactions with a record to cleanse at the server side.

The pending queue will accept further insert, update & delete transactions and pile them up. There is no limit to the number of piled transactions in offline mode.

The event will set efaCloud either to PAUSED or DISCONNECTED state. That means that any synchronization is stopped and will need a new starting trigger.

efaCloud feature is deactivated

The efaCloud feature can manually be deactivated after pausing the queues or automatically when detecting an authentication failure.

All remaining pending, busy, and synching transactions are dropped from the queues and the queue handler is stopped and destroyed.

State machine summary

To summarize all into a state machine diagram see figure below.

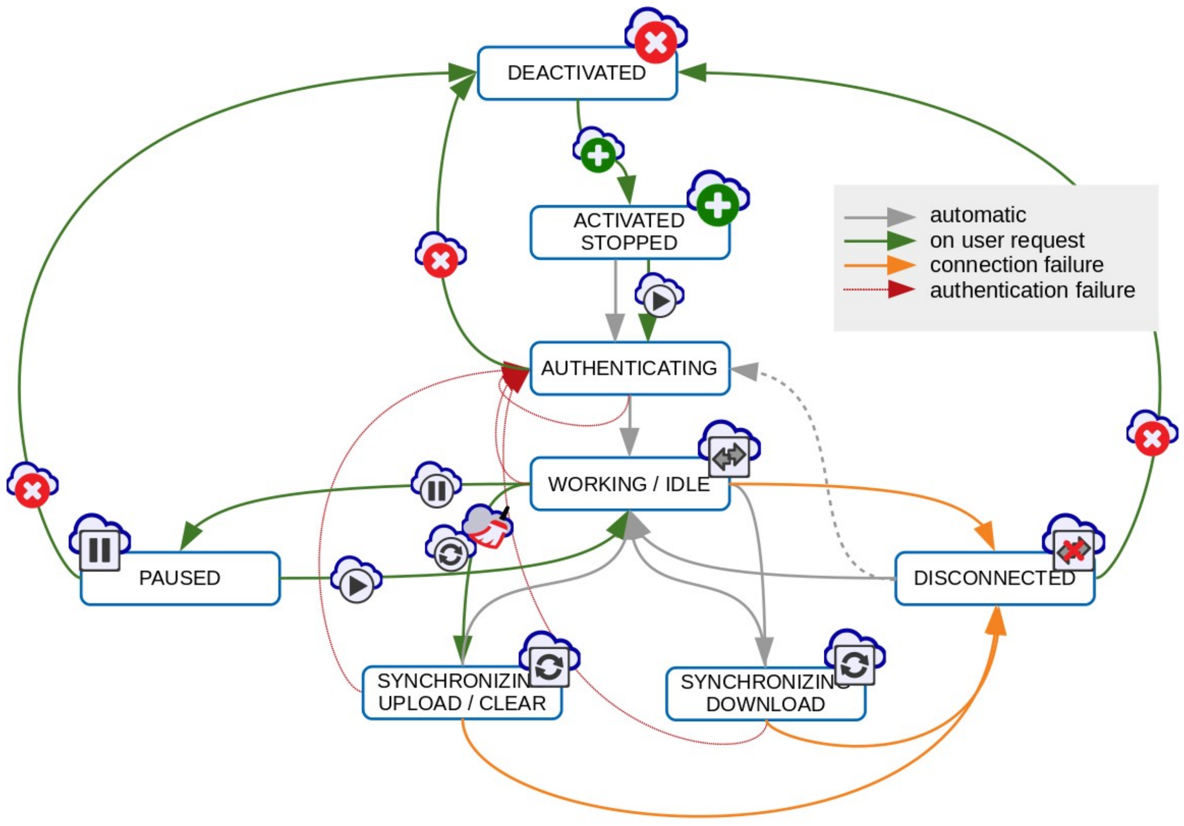

Figure 1: API State machine

Manually state transitions (green arrows) can be triggered in states: “deactivated” by pressing the “activate” button, “working” by pressing the “pause”, “synchronize” or “delete” button, and “paused” by pressing the “start” or “deactivate” button.

Automatic state transitions (gray arrows) are triggered in states: “stopped“ by the program start, “working” by auto-synchronization, “authenticating” and “synchronizing” by successful completion of tasks, and in “disconnected” by detecting a restore of the internet connection. Note that “disconnected” returns to “authenticating” rather than to “working”, if the first pending transaction is “NOP”.

For failure:

- A timeout of the transaction queue (orange arrows) trigger transitions in states “working” and “synchronizing” to “disconnected”,
- a failed transaction (as well as the synchronization completion) in “synchronizing” (gray arrow) triggers a transition to “working/idle”,
- and an authentication failure for any transaction container (dark red fine arrow) triggers a transition in “authenticating”.

Transaction queues

Transaction queues are used in the client side implementation, to handle the communication with the server.

Transactions are typically appended to a transactions-pending queue. It is checked regularly. If during such a check pending transactions are found they are moved to a busy transactions queue and forwarded to the server. Those transactions are packaged into a single transaction container for that purpose.

The protocol is meant to be used in a serial manner, i. e. no further client request container is issued as long as another one is open and neither completed nor timed out. A transaction ID ensures that a response can always be linked to the respective request, but this is not used for sequence correction, but only for debugging purposes and error notifications.

When the server response is received, the busy transactions are moved to the transactions done, or depending on the response code, to the transactions failed queue.

In summary there are the following transaction queues:

1. transactions synch (in memory)
2. transactions pending (locally on disc)
3. transactions busy (locally on disc)
4. transactions done (in memory, limited size, only the last 50)
5. transactions permanently failed (locally on disc)
6. transactions dropped (in memory, limited size, only the last 50)

All queues are kept in memory. Some are additionally written to local disk at every change. They are reloaded from disk when starting the program, cf. Sections ‘efaCloud initialization‘ and ‘efaCloud specific local files’. In-memory transactions are lost when closing the efa client.

Data synchronization

The challenge of data synchronization between a client and the server is addressed by a two step approach:

1. Fix mismatching keys.
   While fixing the pending transactions queue is paused. A special fixing transactions queue is used for that purpose.
2. Synchronize data by either
   a) Data download (default): Retrieve the data records which were modified, since the last time, this step was executed and copying them to the local XML data base.

or

b) Data upload (manually triggered): Retrieve the keys of all data records from the server and insert those from the client to the server which are missing at the server side.

Running a data upload is either manually triggered via the UI or by a cron job (recommended: once per day). Both step 2a and step 2b must immediately follow step 1.

Data synchronisation will start only when both the pending transaction queue and the busy transaction queue are empty. Because the server caches all mismatching client keys it is ensured that update and delete transactions hit the correct data record at the server side, even while a key mismatch exists.

Data synchronization will fall back to the normal WORKING STATE, if a transaction is completed with an error, because the synchronization process relies on the answers and if they don't come the process will not end and continue to block the normal communication.

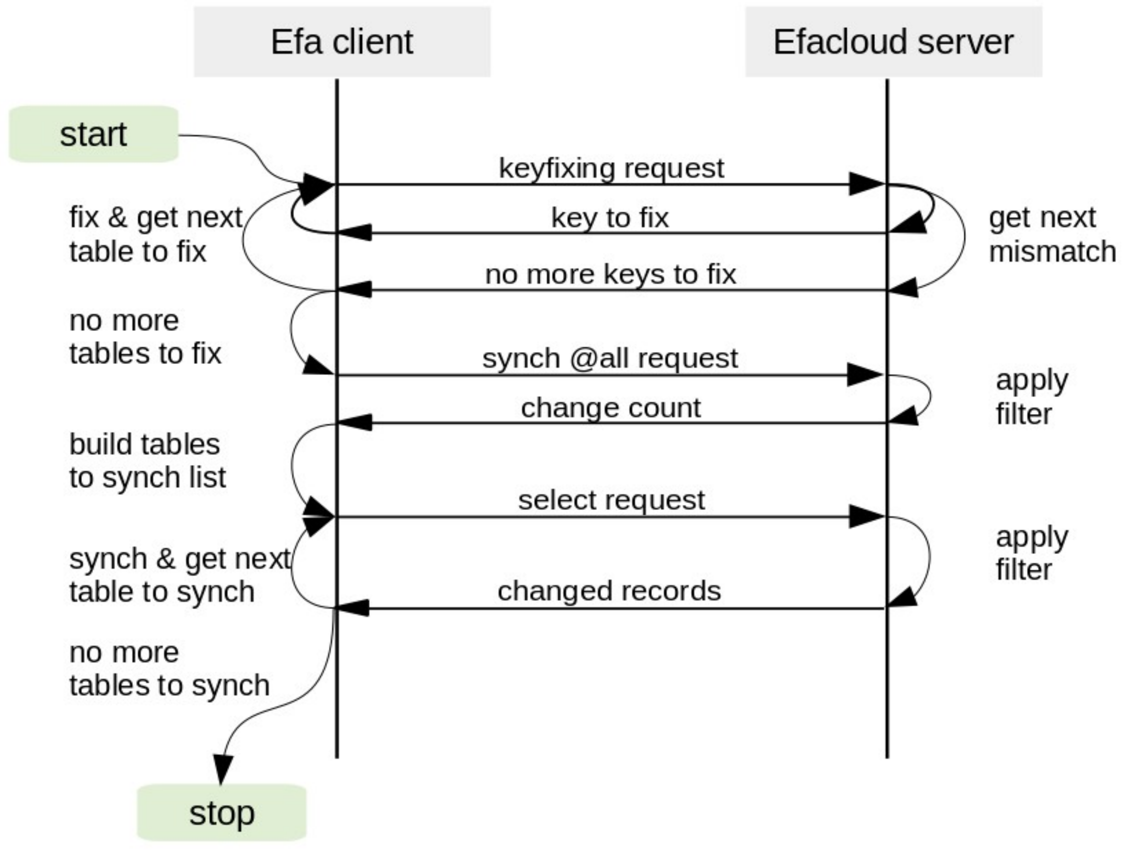

Figure 2: Server to client synchronization (download)

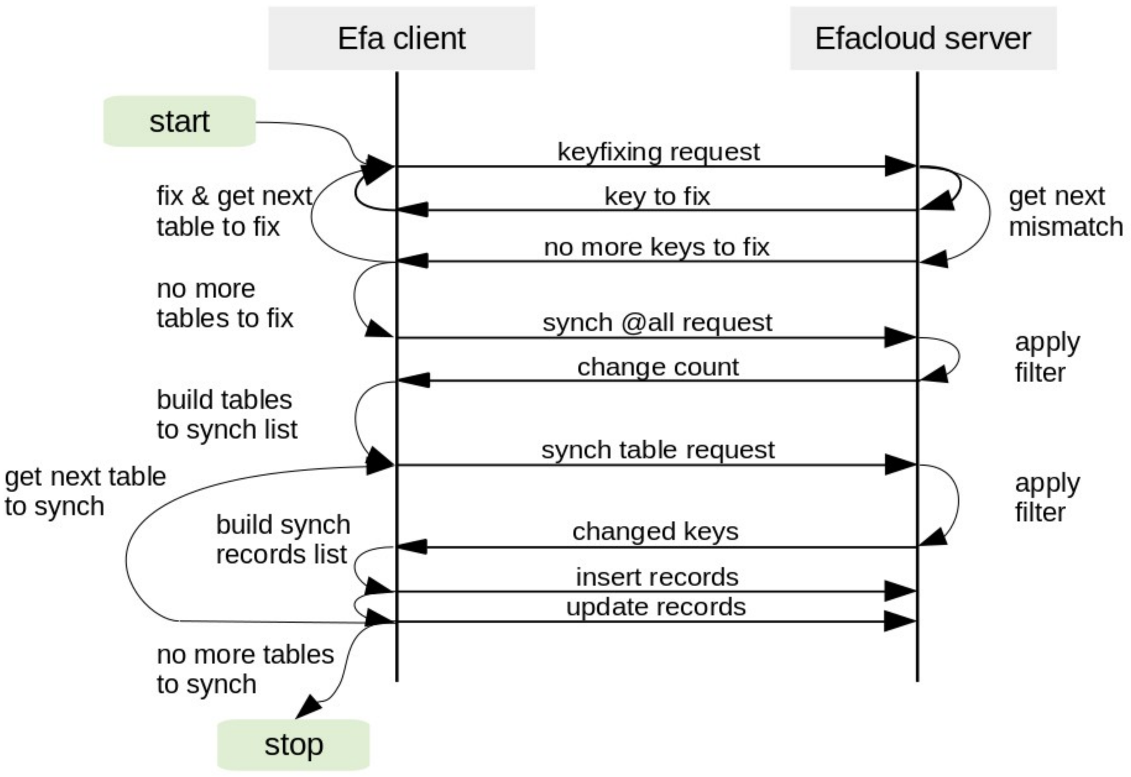

Figure 3: Client to server synchronization (upload)

Adding missing ecrid values via synch upload

In order to ensure ecrid generation for all relevant records after version upgrade to 2.3.1, the “select” transaction will return record which have no ecrid with a “LastModified” value of 230 for version 2.3.0. That will trigger an update statement during upload synchronization which will lead to the required ecrid generation (cf. efaCloud record management.

API transaction format

Transactions hold the information exchanged between client and server. The logical layer is described here.

Because all data are duplicated at the efa client to allow temporary offline use there is a data synchronisation challenge.

This is greatly simplified by the fact, that client and server use exactly the same fields and table structure, the same keys to assert uniqueness and retrieve records.

The client writes transactions immediately to the server, e. g. all trip entries, boat reservations, messages asf. It will poll the server data base regularly for server side registered data updates, e. g. once a day. No server side notification of data change is provided.

CSV-encoding

All messages use csv-encoding for header and payload. The separator character is “;” and the quotation character is ‘”’. The line break character is “\n”.

Headers shall not have entries which need quotation for readability purposes.

Data entries may contain line breaks. They must be quoted if, and only if, they contain either a separator character, a quotation character, or a line break. Quotation characters within an entry are doubled prior to outer quotation.

An example with ten entries is:

- 43;;entry1; entry2a, entry2b;"""entry3""";"entry4;";entry5;"entry6 with
  a line break.";;
- [1]:43 - [2]:[empty] - [3]:entry1 - [4]:entry2a, entry2b – [5]:"entry3" – [6]:entry4; - [7]:entry5 - [8]:entry6 with
  a line break. - [9]:[empty] - [10]:[empty]

Transactions container format

Transactions are bundled into containers, either as requests from the client to the server, or as responses the other way. The server always completes handling of all transaction before it returns its response. If the client detects, that a requested transaction is missing in the response container, it shall regard the transaction as failed.

A transaction container with requests contains four header fields, delimited by a semicolon, and a set of transaction requests:

- <version>;
- <cID>;
- <efaCloudUserID>;
- <password>;
- <transaction_requests>

The transaction_requests are Strings separated by the efaCloud message separator-string ‘\n|-eFa-|\n’ (9 characters). Any value transferred should not contain this String. If so, it is changed to ‘\n|-efa-|\n’ (also 9 characters).

A transaction container with responses consists of four header fields and the responses

- <version>;
- <cID>;
- <cresult_code>;
- <cresult_message>;
- <transaction_responses>

the transaction_responses are Strings separated by the efaCloud message separator-string.

Transaction containers MUST end with the end of the last transaction message, not with an efaCloud message separator-string.

The entire container is encoded as UTF-8 String. It is then base64 encoded with the following characters replaced: “/” by “-”, “+” by “*”, “=” by “_”. As such it will not need any URL encoding.

This applies for both request and response and is meant to provide byte-safety and predictable character encoding.

API versions

The transaction container uses a version identifier to manage the communication between client and server that may have a different software update status. Three API versions are available up to now:

1. The basic API version. All transaction types except “verify”. Records are always returned without data fields efaCloud record management, i. e. without ecrid, ecrown, ecrhis and ecract fields.
2. Like version 1, but now with transaction type “verify” included, and "createtable", "addcolumns", "autoincrement", "unique" removed.
3. Like version 2, but efaCloud record management data fields are now included included in records returned except the history field “ecrhis”.

The server will use the minimum of a) the API version of the client request and b) the highest available version at the server side.

Transaction format

Transaction requests and transaction responses have a standard format which is as well a csv-type String. Requests have a header and an optional data record. Responses have a header followed by a result message which may contain one or multiple data records as csv-table.

Transaction ID

Each transaction has a numeric ID provided by the efa client. It is autoincremented, and persistent. It will be kept over the transaction lifecycle, e.g. when sending a retry.

Transaction Request

A single transaction request message consists of four header fields and a record:

- <ID>;
- <retries>;
- <type>;
- <tablename>;
- <record>

Header fields must not contain characters outside the ASCII range [32 .. 126].

The first header field is the transaction ID provided from the client for matching the result, the second is the count of retries for this request, the third is the transaction type to be used, the fourth the name of the table affected like “efa2boatreservations”.

The record format is csv: field1;value1;field2;value2;field3;....

If the record is missing, e.g. in the list transaction, there must not be a ‘;’ after the table name.

Transaction Encoding

Transaction messages are encoded as csv-lines. Requests use the delimiter: ‘;’, the text quotation: ‘”’, and unix line breaks: ‘\n’. A single csv-line MUST NOT have an ending line-break. Responses use the delimiter “;”.

A value can contain a line break, line feed or similar characters. Values are quoted, if and only if the value contains either a line break ‘\n’, a text quotation ‘”’ or a delimiter “;”.

If quoted, inner quotes ‘”’ are doubled to ‘””’. A value which starts with a blank or ends with a blank is not quoted and the blank is part of the value.

All texts are UTF-8 enoded.

Transaction Response

The transaction response from efaCloud server consists of:

- <ID>;
- <result_code>;
- <result_message>

The result_code is numeric as is the transactionID.

Transaction Result codes

Result codes are (<400: ok, >= 400: fail). The default result messages contain their meaning. The meanings are:

- 300 => "Transaction completed.",
- 301 => "Container parsed. User yet to be verified.",
- 302 => "API version of container not supported. Maximum API level exceeded.",
- 303 => "Transaction completed with key fixed.",
- 304 => "Transaction forbidden.",
- 400 => "XHTTPrequest Error.",

- 401 => "Syntax error.",
- 402 => "Unknown client.",
- 403 => "Authentication failed.",
- 404 => "Server side busy.",
- 405 => "Wrong transaction ID.",
- 406 => "Overload detected.",
- 407 => "No data base connection.",
- 500 => "Transaction container aborted.",
- 501 => "Transaction invalid.",
- 502 => "Transaction failed.",
- 503 => "Transaction missing in container.",
- 504 => "Transaction container decoding failed.",
- 505 => "Server response empty.",
- 506 => "Internet connection aborted.",
- 507 => "Could not decode server response."

Responses on error codes (400 and above) are always delayed by 3 seconds to protect against brute force guessing of the client credentials.

Transaction types

The following chapter specifies all transaction types available at the client API.

The client writes to efaCloud using “insert” and “update”. The only way to read data from efaCloud is “list”. Lists are defined at efaCloud to provide data as needed. Test the credentials and timeout with “nop”.

Structure build

Transactions to build the efaCloud tables are createtable, addcolumns, autoincrement and unique. Because this is meant to be used for setup only, no change or delete transaction is provided.

API transactions of this section are only available at API Level 1 and must no more be used from level 2.3.1 onwards.

Write transactions

The following API transaction types are available to write data to efaCloud.

insert

Insert a data set into a table. The record contains the data record to insert. If the table allows mismatching keys, the record will be inserted with a different, unique key at the server side, and the client side key will be cached.

Get the next data record pair with conflicting client side and server side key, if there is at least one. Details see section: Fix mismatching keys.

Returned result:

- Result Code: 300 => "Transaction completed." and Result Message ‘ok.’ on success.
- Result Code: 303 => “Transaction completed and data key mismatch detected.” The Result Message is a csv-formatted table like for select. This table contains two records which represent the current server side record (1.) and the same record with the current client side key (2.). The client shall delete the client side record (using the data key of 2.) and insert a copy of the server side record (using the full record 1.) to fix the key mismatch. The same procedure as with type ‘keyfixing’.
- Result Code: 502 => "Transaction failed." and as Result Message a failure description, e.g. key is already in use.

update, delete

Update or delete a data record of a table. The record contains the data record to update or to delete. If the table allows mismatching keys and the record’s key matches a ClientSideKey field, the update or delete will be executed on the record with the corresponding ClientSideKey value.

Data records of the 16 common efa2* tables are not deleted, but emptied. All fields except the data key and the LastModified and LastModification fields are set to “”. The remaining stub is used to inform an offline client afterwards about the record deletion.

Returned result:

- Result Code: 300 => "Transaction completed." and Result Message ‘ok.’ on success.
- Result Code: 502 => "Transaction failed." and as Result Message a description of the failure reasons.

keyfixing

Fix an existing key mismatch by deleting the client side key entry at the server side. Get the next data record pair with conflicting client side and server side key, if there is at least one. Details see section: Fix mismatching keys.

The server will delete its client side key entry of the fixed data record and return the next available mismatching data record pair. It will always return a data record pair with a server side key which is not yet used at the client side.

The record contains the (now synchronous) data key of the record in which the client side key entry shall be deleted at the server side. It can be missing.

Returned result:

- Result Code: 300 => "Transaction completed.", if no further key mismatch exists for the table. The Result Message is empty.
- Result Code: 303 => “Transaction completed and data key mismatch detected.” The Result Message is a csv-formatted table like for select. This table contains two lines which represent the current server side record (1.) and the same record with the current client side key (2.). The client shall delete the client side record (using the data key of 2.) and insert a copy of the server side record (using the full record 1.) to fix the key mismatch. The same procedure as with type ‘insert’.
- Result Code: 502 => "Transaction failed." and as Result Message the table name followed by a reason description.

Read data from efaCloud

The following API transaction types are available to read data from efaCloud.

select

Get a set of full table records from the server side. This includes the LastModification field, because it is needed for one step download synchronization of tables.

The record contains a matching filter like [“Id” => “someGUID”, “ValidFrom” => “1567389200”, “?” => “=”] in which the “?” field contains the SQL operator applied to all filter values.

Returned result:

- Result Code: 300 => "Transaction completed." on success and as Result Message a csv-formatted table, the first line being the column names. Extra server side column ClientSideKey is excluded.

  If the no data match the filter, the result message is “none matching”.

  If the table does not exist, the result message is “no such table”.

  The transaction cannot fail, there is no other result code option.

synch

Get the keys and modification stamps of data records based on a filter. Similar to select but implemented to identify synchronisation need and trigger subsequent select or update statements. Use synch to find the tables with changes using the LastModified time stamp as filter.

The record contains a matching filter, see the select type.

If the table name is “@all” the transaction will apply the filter to all tables and return the count of matching records rather than the keys.

Returned result:

- Result Code: 300 => "Transaction completed." on success and as Result Message a csv-formatted table, the first line being the column names. Columns are the matching records data key fields together with the LastModified and LastModification data fields.

  If the table name is “@all”, the transaction will apply the filter to all efa2 tables and return as result message the count of matching records rather than the keys formatted “tablename1=count1;tablename2=count2;...”. If there are no efa2tables yet in the database, the result message is empty.

  If the no data match the filter or if there are no tables at @all, the result message is empty.

  The transaction cannot fail, there is no other result code option.

list

Get a predefined list from the server side.

The tablename contains the list’s name, the record contains a field: “setname” with the name of the list set, e.g. efaWeb, optionally a “LastModified” value (seconds) as list parameter {LastModified} and optionally further multiple listarg1, listarg2, listarg3, ... fields like “listarg1;{PersonId}=f233a7ee-acf8-4caa-aa0b-0dbc57310a3d”.

Returned result:

- Result Code: 300 => "Transaction completed." on success and as Result Message the csv-formatted list, the first line being the column names.
- Result Code: 502 => "Transaction failed." if ‘listname’ is not a name of a predefined list or “setname” is not a name of an existing set or missing.

Support functions

The following API transaction types are available for support functions such as debugging or server activity triggering.

nop (synch config, welcome message)

Returns with no further operation after the given sleep-period in seconds.

Used for credentials check, synch configuration alignment between server and client, welcome message propagation server to client and timeout testing.

The record hold the sleep-period in seconds, e.g. “sleep;3”. Value range 0 .. 100. All values < 0 are set to 0, all > 100 are set to 100.

Returned result:

- Result Code: 300 => "Transaction completed." and as result “ok.” after sleeping for sleep-period seconds. Appends some configuration values.
- Result Code: 502 => "Transaction failed." if the credentials could not be verified, after “sleeping” 3 seconds.

verify (from API V2 onwards)

Verify another user by his credentials. Used to authorize admin sessions at the client side. Is always directly triggered by a user who enters the account name and password which shall be verified. If the efaCloudUser which shall be verified has no password set, but an external authorization provider exists, the call is forwarded to the AuthProvider interface (see section Auth provider).

The record contains one of the data fields “efaAdminName” or “efaCloudUserID” to identify the user and the data field “password” of the users password (plain text, must never be stored). If both “efaAdminName” and “efaCloudUserID” are provided, “efaCloudUserID” is ignored.

Returned result:

- Result Code: 300 => "Transaction completed." and as result the user data record without history and password hash as csv table with a header plus a single data row.
- Result Code: 402, 403 and as result message a description why the verification failed.

Note that there is no session management at the server side. It is the task of the client to change the clientID for the next transaction request in order to use the privileges of the verified user.

backup

Returns after creating a data base backup into a zip archive of text files. There is a two stage backup process with 10 backups at each stage. So this gives you 10 days daily backup and 10 backups with a 10 day period between each, I. e. a 100 day backup regime.

Returned result:

- Result Code: 300 => "Transaction completed." and as result the primary and – if created - secondary backup index after backup completion, e.g. ‘p7’ or ‘p0s2’.

  The transaction cannot fail, there is no other result code option.

info

Returns an information table on the boat status, currently. The record defines the information type and format mode

Information types:

- regarding boats: onthewater, notavailable, notusable, reserved

Format modes:

- bit mask 0x1: 1= HTML, 0 = CSV
- bit mask 0x2: headline on/off
- bit mask 0x4: multicolumn on/off

Returned result:

- Result Code: 300 => "Transaction completed." and as result message the requested information.
- Result Code: 502 => "Transaction invalid.", if the type is not one of the allowed Strings.

upload

Upload a **file** to the efacloud server. Does not modify the data base. Typical usage scenario is statistics publication.

The parameters are: tablename contains the filetype, “zip”, “text” or “binary” are allowed. The record contains a filepath and a contents field. The filepath is a relative path, but must not contain the “../”-String to prevent from access to higher level directories. The file size is limited to 1.0 Mbyte. Zip and binary file contents must be base64 encoded. Zip files are extracted and the contents is saved rather than the archive.

Returned result:

- Result Code: 300 => "Transaction completed." and Result Message contains the number of bytes written on success.
- Result Code: 502 => "Transaction failed." and as Result Message a description of the failure reasons.

Fix mismatching keys

Four tables allow for key correction: the logbook, the messages, the boat reservations and the boat damages.

Key fixing is performed at the client by deleting the record with the wrong key and inserting instead the record with the corrected key.

When mismatching keys shall be fixed, the client issues a keyfixing request and as long as mismatching records are returned which shall be fixed, it will continue doing so. When no mismatching record is returned, it will continue with the next table. That will go through all four tables which allow key fixing.

Find free client side data keys

For both insert and keyfixing requests the server has to return an appropriate data key which shall be fixed. It is mandatory that this new data key is not in use at the client side, because if so, the insertion of the fixed record at the client side will fail.

Now the set of client side keys is the set of keys of all records without key mismatch plus the set of mismatching client side keys. To find a free client side key you need to iterate through the server side keys until you’ve found one, which is not in set of client side keys. Because all synchronous keys are part of both sets, they can be left out.

Said that, a free client side key can be detected at the server by iterating through the set of all server side keys of mismatching records until one is detected which is not in the set of mismatching client side keys.

Key mismatch consequences

The server will register the client side key within the server side record, if a key mismatch occurs. That will ensure that the client-server communication is always referring to the same data records.

But at the client side it is not fully safe. If a client side operation is ongoing it opens the record and releases it during the user interaction. Once the user has finished, the modification is applied to the data base. If the record’s key was changed in the meantime by a background key modification the client operation modifies none or even a wrong record.

This is quite a heavy consequence and there is nothing one can do about it without a major change at the client architecture. The risk is taken, because the scenario is quite improbable: the client must have been offline at the operation’s start and go online just during its execution. And the operation must modify a mismatching record, i. e. a record which has been inserted during the offline period.

Data content checks

The API takes data provided by the efa-PC and takes them for granted, because the efa-PC does all checks.

Now there are two more data sources: efaCloud and efaWeb. They need to apply the very same data compliance checks as efa2 does. If not, download of a record may fail upon insertion in efa2 due to some rule violation.

The checks which are applied by efa2 are therefore rebuilt into efaCloud to support those two data sources.

All data content checks are implemented in the efa_record.php class and addressed by writing data only through the Efa_record::modify_record() function.

API checks

All data provided for insert, update or delete over the API are checked for their compliance and an error is returned (result code 304, result message gives the cause). Most of these checks are performed for API level 3 and more, while API level 1 and 2, which are used only by efa2, does skip all checks except period consistency for logbook and clubworkbook records and insertion against existing non recent records.

efaCloud application checks

The server application uses the very same checks as for the API level 3+ before any data is modified. The corresponding error is displayed in the form. For that reason efaCloud data manipulation forms add some javascript code to provide proper error display,

Efa client programming considerations

The efa client is a development over many years. No attempt is made to explain the code, but rather an attempt to explain the efaCloud adaptations. There are two parts to it: the way how it integrates with the existing client and the add-on classes, which do not use any efa client parts.

Efa client integration

In order to integrate with the efa client, both at the GUI and at the storage interface adaptations were needed. Efa client basic capabilities were reused as much as possible.

GUI

The GUI integration was chosen to be as lean as possible. The project setup dialog gets a third storage option “efaCloud”. The only specific parameter is the efaCloud URL. Credentials are entered as for efa remote, the local storage is as with the default configuration XML, so no need for any GUI change. More configuration options may follow.

Chaged classes are Project, PreojectRecord, BaseDialog (only for icon reference), NewProjectDialog, and EfaMenuButton (Synch option).

Storage interface

Efa2 uses a set of storage packages and an abstract IDataAccess interface, to access all permanent storage. The default is local XML-file based storage. efaCloud extends this local storage by a link into the efaCloud API. Within the modofyRecord function it checks the storage ty and, if it is efaCloud, it creates a transaction to the efaCloud server in parallel to the local storage update.

This implementation was chosen, because the local storage is needed for offline operation and fast access to data, which do not change frequently, like person or boat records. The will only be looked up locally. Regular background synchronisation replaces online access for reading such data.

In order to link into the storage interface minor modifications within the following classes were needed: Daten, MetaData, EfaConfig, Audit, DataAccess, IDataAccess, DataRecord, Datafile, XMLfile.

For the duplicate Id problem, fields were added to LogbookRecord and MessageRecord (see below).

Logging, error handling, internationalization

Efa catches all errors for robustness reasons and logs only very basic activities like communication startup and tear down within the normal efa log. efaCloud adaptations make use thus of the efa capabilities. The same applies for progress display, logging or internationalization. However, most activities are logged in different files to avoid that errors throw a message which then will be appended to the transactions queue which again may throw a message asf.

efaCloud specific local files

efaCloud uses files for permanent queue storage and logging.

The permanent queues sit within the data section of the local storage, i. e. the location of the XML files. There are two directories:

a) efacloudlogs with the efacloud log of all transactional and synchronisation activities plus API and internet access statistics logging. Log files hav a max size of 200kByte and are copied to the efacloud.prevous which is replaced the next time.

b) efacloudqueues. Which stores transactions for the queues.

Log files are uploaded to the server to give the user a possibility to see the client activities from remote.

efaCloud initialization

efaCloud activities are all controlled by the TxRequestQueue instance. efaCloud uses the efa native storage instantiation to trigger the TxRequestQueue instantiation and initialization. Upon instantiation of the queues permanent queues are reloaded from the local disk.

The TxRequestQueue is instantiated by and only by the EfaCloudStorage constructor. This constructor itself has a single caller: the ‘DataAccess.createDataAccess’ function. That function has two callers, one specific for efa remote which will not create an efaCloud storage. The other is the abstract StorageObject constructor.

The StorageObject constructor is called as “super()” call from its implementing classes. Three of them are the configuration classes (Admins, EfaConfig, EfaTypes), 16 are the efa2 table classes (AutoIncrement, BoatDamages, etc), and two further the Project and Clubwork class. The “super()” call in all of their constructors uses either a non-efaCloud storage type index or the Projects default storage type index to decide on the storage type to create. So, finally, the efaCloud objects are only instantiated, if the project type is efaCloud.

Thus no efaCloud objects will be instantiated, if the project is not an efaCloud project. Switching from XML to efaCloud and vice versa therefore needs closing and reopening the project.

efaCloud specific classes

Again, two pieces were needed: a set of class files within the native efa packages to integrate plus an efaCloud specific package, located in the storage path.

Classes within the efa native structure

The link to the data storage is with the “de.nmichael.efa.data.storage.EfaCloudStorage” which extends XMLfile to push transactions to the server and support synchronisation.

The class “de.nmichael.efa.gui.EfaCloudConfigDialog” provides, based on the “de.nmichael.efa.gui.BaseTabbedDialog”, the menu to activate or deactivate the efaCloud feature and, if activated to pause and resume the client server handshake as well as to trigger a client to server synchronisation.

The DataRecord and DataFile classes are modified to forward transactions to the server. A DataRecord has two more fields: LastModification and serverCopy which are added outside the dynamic field set, because they are only teporary and will never be stored at the client side. The DataFile class hanldes the forwarding of server modifications within the modifyRecord function.

All other classes could be put to the efaCloud package.

Classes in the efaCloud package

Within the de.nmichael.efa.data.efaCloud package the following functions are programmed: API transaction encoding and decoding, transaction queue handling, internet access handling, table structure mapping and synchronization.

API transaction encoding and decoding

Class Transaction: transaction encoding and decoding. Class CsvCodec: little helper for transaction and container decoding and encoding.

Transaction queue handling

Class TxRequestQueue holds all five queues, the queue poll timer and queue shift functions. Class TxResponseHandler provides container decoding and triggers response related activities. Class TextResource: Used to exchange queue data with the local disc for permanent queues like done and failed.

Internet access handling

Class InternetAccessManager: Send and receive transaction container. This is also performed by queueing the container and polling the queue, as is done with the transaction in the TxRequestQueue. Class TaskManager: provides the framework to send and receive messages over the internet. Ensure decoupling of internet activities into a separate thread. This is done with both the request and the response container.

Table structure mapping and synchronization

Class TableBuilder: Provides a reading and mapping of the data base structure of the native efa and a set of constants describing the exceptions from rules. Maps data type definitions to SQL-type meta data information. Holds all table definitions of the efaCloud server for building and reading the server side data base.

Class SynchControl provides the functions to steer up- and download of data to and from the efaCloud server. Is triggered by the EfaCloudConfigDialog or a 12h autonomous synchronisation job.

The NOP transaction carries the server table layout so that the TableBuilder can check the local table layout and adjust the data field length, if needed. Any Adjustment or mismatch of local tables will be reported in the auditinfo.txt.

Admin to efaCloudUser mapping

Two classes in the de.nmichael.efa.core.config package provide the record and functions for the mapping of efa admins to efaCloudUsers: EfaCloudUserRecord and EfaCloudUsers.

CLI

The cli package which handles the efaCLI commands also has a MenuEfaCloud class to trigger the upload or the houskeeping via the automatic processes from a boathous installation. The class is just a hook into the basic efaCloud classes.

efaCloud server administration application

The efaCloud server side application is build on plain PHP using an SQL database, but no further framework except the included tfyh.org PHP classes are there (see section Program architecture). This keeps things simple and the need for software update low.

It is strongly encourage to use the code embedded comments and documentation to understand the details of any class, form or page file. I will not be able to maintain comments both there and her – preference is always the code. You may use doxygen or similar tools to extract the comments which use the javadoc conventions.

Efa Admins and efaCloudUsers

The efaCloud server has an own user list. This is technically different from the efa admins, but it is strongly recommended to put all admins of the log book into the efaCloudUser list and use only the “super-admin” name ‘admin’ on the client side.

All authorisation should then be done by verifying the server record. This has the disadvantage, that such admins can not log on when offline. But it is the only way to ensure conststence of user rights management.

User privileges

In order to be able to reflect the user privileges which are defined in efa, the user record has a set of Workflows and Concessions which reflect those. Mapping is done within the efa client.

Default admin

After installation a default admin with efaCloudUserID “1142”, efaAdminName “alexa” and password “123Test!” is set.

Do never use this Default admin, but change it immediately after installation.

Auth provider

The application uses a login procedure. If you want to ask a different authentication engine (e.g. your club administration software) to authorize your user, you can implement the Auth_provider in the authentication folder. It will be asked for a password hash based on the efaCloudUserID, if the efaCloud user has no password provided in the efaCloudUser table.

If the Auth_Provider returns a password hash to verify, the user provided password will be verified by the standard PHP call: password_verify($entered_data["Passwort"], $passwort_hash);

This procedure is also called for the API client authorisation, if the provided client does not have a password provided in the efaCloudUser table.

Data editing

The efaCloud Server administration application provides a capability to edit records. In order to provide the same lookup functionality for persons, boats, destinations asf. The data editing form uses Javascript code that was originally developed for efaWeb. See efaWeb description for details.

efaWeb application

efaCloud provides a efaWeb Javascript based efaWeb application which is called by reading the /pages/efaWeb.php script. The script passes configuration data as Javascript variables within the php source code to the Javascript client to be used by the efaWeb application.

efaWeb is run as a client, i. e. it **interacts only via the API** with the efaCloud server.

Basic application concept

The Javascript implementation uses jQuery as browser abstraction layer. It executes by defining a set of classes as vars, holding all data, the API transaction queue, form handler and configuration. It is controlled by few direct scripts.

The entire code is held within the js_xx folder. With each update the xx-number is increased to ensure, that no browser cache will start to use old code parts with the next release.

The menu displayed uses the very same logic as for the server side application, just with a different menu configuration file, the wmenu. The norma menu loads a new page on every menu action click, here the generic event is triggered by using a target of “event:function_argument” instead of a link to a php page.

Common code

The common code for use at efaCloud server and efaWeb is within the c****.js files

cAutocomplete.js

Manage all autocompletion of input where names shall be entered and UUIDs are to be matched.

cConfig.js

A class to read all common configuration from the php file’s variables. Such configuration variables are global to Javascript and start with a “$_” for immediate recognition.

cGlobals.js

A class to provide more global fields which do not depend on the efaCloud Server side.

cHelp.js

A class to provide help texts for modal display.

cList.js

The central web app data base class. It holds all tables and all indices for fast data retrieval. Data are always kept in memory, but once retrieved they are also cached in the localStorage except the persons table. This provides faster startup.

A list is not the same as the efaCloud table, but uses only those data fields which are needed to run efaWeb. A list usually corresponds to a list of the efaWeb list set at the efaCloud Server side. The only difference ist with the efaWeb_own_sessions and efaWeb_opentrips lists which filter some session entries and provide more information on those. They will therefor be merged into the efaWeb_logbook list.

cModal.js

Provides a modal which superposes the UI to enter data or display text or other stuff.

cToolbox.js

A toolbox for things like date formatting etc.

efaWeb specific code

All scripts start with a “b” for boathouse intead of “c” for common.

bConfig.js

All efaWeb configuration information: form layouts etc.

bEvents.js

The main application control triggering all activity.

bForm.js

Provide a form based on a form layout definition.

bFormHandler.js

The primary application logic to enter data. For each form there is a “<form>_do” and a “<form>_done” function to check and stored the entered data.

bLog.js

A very small logging facility.

bPanel.js

Provides the basic panel with the boat and boat status trees

bStatistic.js

A class to provide statistics including the logbook contents for display

bTxHandler.js

The API queue response handler and transaction formatter. It will process the returned transaction’s result according to the transaction type and call the postprocessing method afterwards.

bTxQueue.js

The API queue manager pulling the queue, packaging containers and sending them to the efaCloud server. When sending an API container you can provide a function with the transaction which is called when returning. This is then carried out by the bTxHandler.js script.

Object container classes

oBoat.js

Holds a boat record and provides appropriate formatting, boat status control etc.

oBoatstatus.js

Holds a boat status record and provides appropriate formatting.

oDamage.js

Holds a damage record and provides appropriate formatting.

oPerson.js

Holds a person record and provides appropriate formatting.

oReservation.js

Holds a boat reservation record and provides appropriate formatting and checks.

oSession.js

Holds a session record and provides appropriate formatting, workflow etc.

efaCloud server application scripts

sEvents.js

The main application control triggering all activity.

sForm.js

Provide a form based on a form layout definition.

sMenu.js

The efaCloud menu unfolding and folding script.

Use cases

There is no documentation of the efa client to tell what actually happens on a specific use case. They are, however, described in the following, based on what actually efa 2.3.2 does.

In order to understand what happens on a specific activity please use the debugging facility of your browser. This is far more instructive and always up to date than any description here. Just two examples are given.

Start a session

To start a session the following information must be provided by the client to the server via the API. Values are example values. That changes both the logbook and the boat status table.

Insert session

INSERT	**efa2logbook**
EntryId	2145
Date	2021-12-22
BoatId	752db431-1e30-4b2b-9111-4ee5e97d6c59
BoatVariant	1
AllCrewNames	Glade, Martin
Crew1Id	5ee42ad7-3fdf-423b-9547-86bbedd3cf6a
BoatCaptain	1
StartTime	12:25:00
DestinationName	1. Fähre - Unisteg
WatersIdList	f79b57a9-fb83-46b5-a739-447504679a11
Comments	Testeintrag
SessionType	NORMAL
Open	true
ChangeCount	1
LastModified	1642875611636
Logbookname	2021

Update boatstatus

Grey fields are written by efa, but not modified by efaWeb. If no boat status record exists for the given BoatId, the record is not updated, but inserted.

UPDATE	**efa2boatstatus**
BoatId	752db431-1e30-4b2b-9111-4ee5e97d6c59
BoatText	*Sahneschnittchen*
UnknownBoat	
BaseStatus	*AVAILABLE*
CurrentStatus	ONTHEWATER
ShowInList	
OnlyInBoathouseId	
Logbook	2021
EntryNo	2145
Comment	unterwegs nach 1. Fähre - Unisteg seit 22.12.2021 um 12:25 mit Glade, Martin
ChangeCount	582
LastModified	1642875611637

Close a session

Close trip

UPDATE	**efa2logbook**
EntryId	2145
Date	2021-12-22
EndDate	
BoatId	752db431-1e30-4b2b-9111-4ee5e97d6c59
BoatVariant	1
BoatName	
AllCrewNames	Glade, Martin
CoxId	
CoxName	

```
Crew1Id                 5ee42ad7-3fdf-423b-9547-86bbedd3cf6a
Crew1Name
[...]
Crew24Id
Crew24Name
BoatCaptain             1
StartTime               12:25:00
EndTime                 19:15:00
DestinationId
DestinationName         1. Fähre - Unisteg
DestinationVariantName
WatersIdList            f79b57a9-fb83-46b5-a739-447504679a11
WatersNameList
Distance                13 km
Comments                Testeintrag
SessionType             NORMAL
SessionGroupId
EfbSyncTime
Open                    false
ChangeCount             2
LastModified            1642875623413
Logbookname             2021
```

Reset boatstatus

Grey fields are written by efa, but not modified by efaWeb.

```
UPDATE                  efa2boatstatus
BoatId                  752db431-1e30-4b2b-9111-4ee5e97d6c59
BoatText                Sahneschnittchen
UnknownBoat
BaseStatus              AVAILABLE
CurrentStatus           AVAILABLE
ShowInList
OnlyInBoathouseId
Logbook
EntryNo
Comment
ChangeCount             583
LastModified            1642875623414
```

Appendix

Licence consideration

efaCloud is published under the GNU public license, latest version. See http://www.gnu.org/copyleft/gpl.html.

System prerequisites

efaCloud requires at the server site a mySQL data base and a PHP-interpreter 8.0 or higher with 32-Bit native integer (PHP_INT_SIZE = 4) together with the usual Apache web server. The web hoster should provide for an appropriate performance. Most challenging is uploading and backup.

If you decide to run efaCloud using a local web server, a raspberry may not be suited due to the mySQL program weight and overhead. And since this is not the purpose of that software, we will not support it.