efaCloud – efa2 in der Wolke

A programmers guide

www.tfyh.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/mySQL. 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 framewor which was developed for different purposes by myself. Its documentation is separately provided in a tfyh.org PHP framework description.

Bonn, August 2021

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. 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 adminstrator running the administrative tasks from home PC using the efa client or the efaCloud server application for this purpose.

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.

An experimental Javascript application is added for web based logbook usage, but documentation is still to be done there. Use it with care.

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 administration application.

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 16 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 efaCloud and only stored locally at the client side.

Efa table names in the efaCloud data base their extensions or storage types like "efa2persons". The current logbook table name, however, is always "efa2ogbook", and not the logbook name given in the efa 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.

When enabling efaCloud record management, three 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) and record history (ecrhis).

The purpose of it is not the usual efa-PC which will continue not to be aware of this and still run smoothly with efaCloud. The first use case 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

This Id is meant to be unique over all efaCloud Server installations. A UUID would do, but the key will be used in lists, so it is shortened and compressed: we create a UUID (e.g.

05e0beda-dbc1-48fa-a7ab-59ef3e916947), drop all dashes (05e0bedadbc148faa7ab59ef3e916947) and compress the sextets of characters 3-8, 13-18 and 23-28 (e0beda, 48faa7, ef3e91) each into a four character base64 type sequence like for the transaction container, i. e. with the following special characters: "-" instead of "/" and "*" instead of "+" (Z6K3, mpOR, Q5y6 yielding 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 $4{,}7 * 10^{21}$ 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.

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. Others may need them.

Therefore the efaCloudUser has a flag to configure whether it gets the efaCloud record management data fields or not.

Enabling and disabling the efaCloud record management

The server provides means to enable or disable the efaCloud record management.

- When enabling the efaCloud record management the three columns ecrid, ecrown and ecrhis will be added to all efa tables and their values set to ecrown=NULL, ecrhis="", ecrid=new generated random id. The data base layout configuration parameter (in "../config/settings_db") will be set to 2. The data field 'UseEcrm' is added to the table "efaCloudUsers".
- When disabling the efaCloud record management the three columns ecrid, ecrown and ecrhis will be deleted from all efa tables and the data base layout configuration parameter (in "../config/settings_db") will be set to 1. The data field 'UseEcrm' is removed from the table "efaCloudUsers".

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.
- efaCloudConfig: a liust of configuration values to configure the server appplication
- efaCloudLog: a log of all recent changes in the efaCloud data base.
- efaCloudPartners: the list of efaCloud partner clubs and their efaCloud servers with which this efaCloud server communicates.

These tables are not shared with the client and use a single numeric key. They are not versionized.

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 permenant 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. It is recommended to do so every 90 days or so. It is intended to be done at the boat house location where server and client can be accessed both.

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.

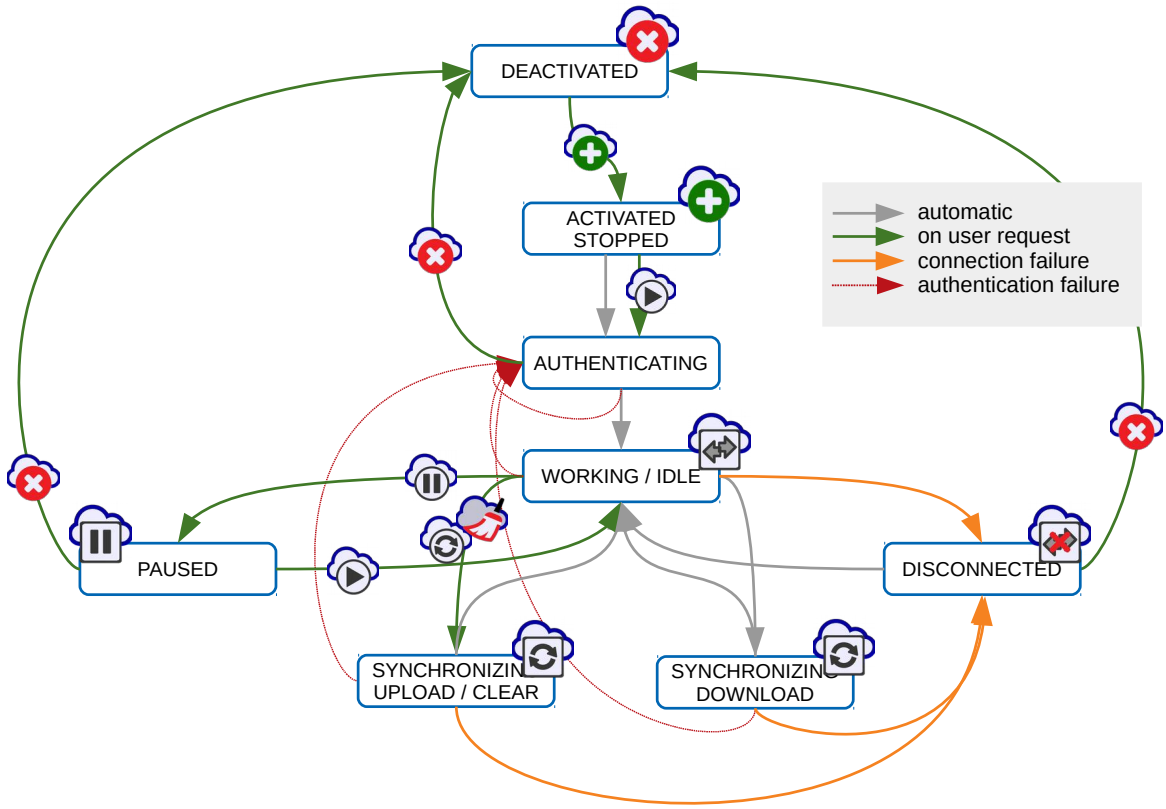

Figure 1: API State machine

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.

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.

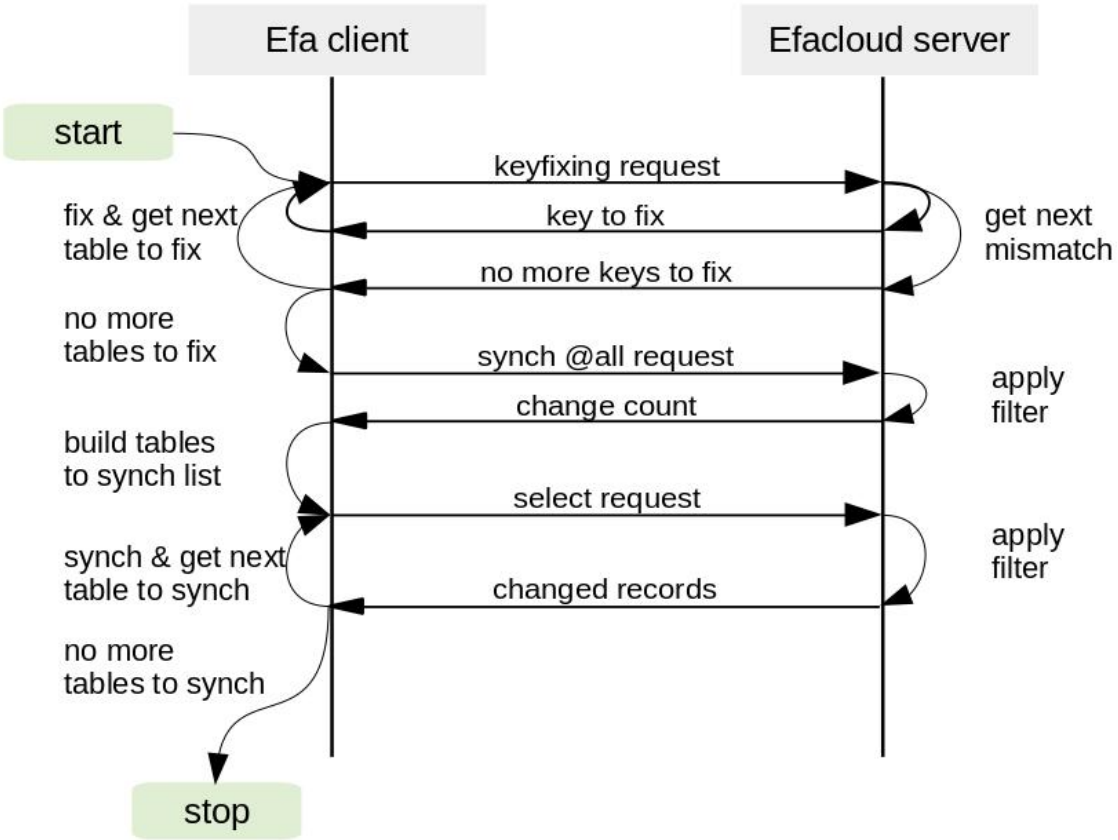

Figure 2: Server to client synchronization (download)

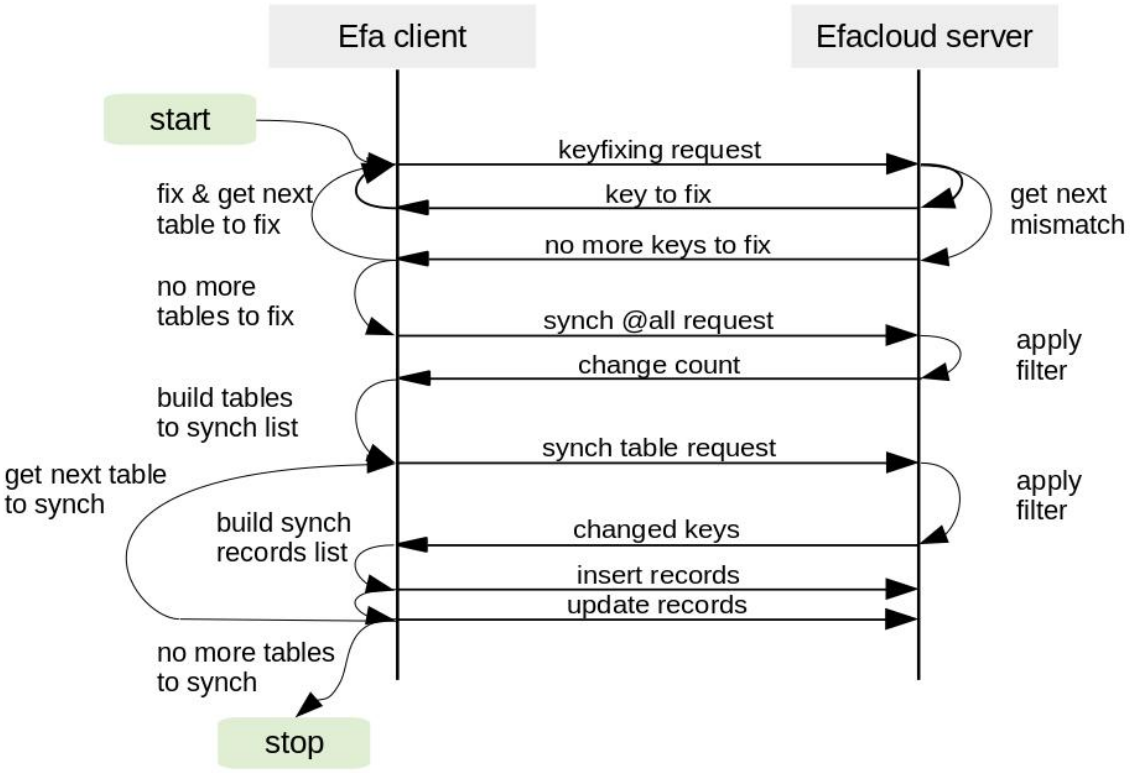

Figure 3: Client to server synchronization (upload)

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.

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):

300 => "Transaction completed.",

400 => "XHTTPrequest Error.", (client side generated error)
401 => "Syntax error.",
402 => "Unknown client.",
403 => "Authentication failed.",
404 => "Server side busy.",
405 => "Wrong transaction ID.", (client side generated error)
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", (client side generated error)
506 => "Internet connection aborted", (client side generated error)
507 => "Could not decode server response" (client side generated error)

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”.

Build structure at efaCloud

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.

createtable

Create a table.

The record contains column definitions, e.g. [“Id” => “Varchar(256) NOT NULL”, “ValidFrom” => “int(20) NOT NULL”]. The record MUST contain all key fields and may contain further fields. The list of key fields of the efa2 tables is static PHP code and not passed over the API.

If a table with the given name exists, it will be dropped. So be careful with this transaction type. Returned result:

Result Code: 300 => "Transaction completed." on success,

Result Code: 502 => "Transaction failed.", if the data base operation failed.

addcolumns

Add columns to an existing table.

The record contains the column definitions, e. g. "FirstName" => "varchar(256) NOT NULL DEFAULT 'John'", "LastName" => "varchar(256) NOT NULL DEFAULT 'Doe'", "MiddleInitial" => "varchar(256) NULL DEFAULT NULL". It MUST contain at least one column. .

If a column with the given name exists, it will not be changed. That will however, not abort the operation and thus not affect the other columns provided. Returned result:

Result Code: 300 => "Transaction completed." on success,

Result Code: 502 => "Transaction failed." one or more columns with the given name exists at the efaCloud server.

autoincrement

Alter a column to become autoincremented. The column must be of integer type. It will be set to be the primary key.

The record contains the name of the column and the sql definition (ignored), e. g. “EntryId;Int(10) NULL DEFAULT NULL”. The data base command is like “ALTER TABLE efa2logbook ADD PRIMARY KEY (EntryId); ALTER TABLE Test MODIFY COLUMN EntryId INT auto_increment”

If no column with the given name exists, nothing will not be changed. Returned result:

Result Code: 300 => "Transaction completed." on success,

Result Code: 502 => "Transaction failed." on any database transaction error at the efaCloud server.

unique

Alter a column to make it unique.

The record contains the name of the column and the sql definition (ignored), e. g. “Id;Varchar(64) NULL DEFAULT NULL”. The data base command is like “ALTER TABLE efa2boatstatus ADD UNIQUE (BoatId)”

If no column with the given name exists, nothing will not be changed. Returned result:

Result Code: 300 => "Transaction completed." on success,

Result Code: 502 => "Transaction failed." column with the given name does not exists at the efaCloud server.

Write data to efaCloud

Write data transactions are insert, update and delete. Write data transactions provide the table record or data key as record within the transaction request.

The count of fields MUST be equal to the count of values. Failure to do so results in an error result code 501 => "Transaction invalid."

All key fields of the specified table MUST be provided in the record. Failure to do so results in an error result code 501 => "Transaction invalid."

Prior to any write activity the key fields are checked. If a record with matching key fields is detected, the following happens:

- Insert statements are aborted upon exact match of all key fields and the error 502 => "Transaction failed." returned.
  Exception: for tables which allow mismatch by using an extra client side Id field at the server side this case is also a success and leads to writing a new data record at the server side.
- Update and delete statements are aborted, if no record with exact match of all key fields could be found. Note: Key fields cannot be updated. updating a key field like VALIDFROM requires a delete and subsequent insert statement.

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 is ignored.

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.

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.

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.

API testing

In order to check the API it shall be testable. Testing can be performed by using the efaCloud Server application “API Test” menu option in the menu’s monitor section. Here are test records per API transaction type, using the default user ‘1142’ with the default password ‘123Test!’:

- createtable, example 1:

```
tablename = efa2status
recordfields = Id;ChangeCount;LastModified;Membership;Name;Type;Details
recorddata = varchar(64) DEFAULT NULL;Int(10) NOT NULL;varchar(64) NOT
NULL;varchar(64) NOT NULL;varchar(64) NOT NULL;varchar(64) NOT
NULL;varchar(1024) DEFAULT NULL
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7Y3JlYXRldGFibGU7ZWZhMnN0YXR1cztJZDt2YXJjaGFyKDY
0KSBERUZBVUxUIE5VTEw7Q2hhbmdlQ291bnQ7SW50KDEwKSBOT1QgTlVMTDtMYXN0TW9kaWZpZWQ7dm
FyY2hhcig2NCkgTk9UIE5VTEw7TWVtYmVyc2hpcDt2YXJjaGFyKDY0KSBOT1QgTlVMTDtOYW1lO3Zhc
mNoYXIoNjQpIE5PVCBOVUxMO1R5cGU7dmFyY2hhcig2NCkgTk9UIE5VTEw7RGV0YWlsczt2YXJjaGFy
KDEwMjQpIERFRkFVTFQgTlVMTA__
```

- createtable, example 2:

```
tablename = efa2logbook
recordfields =
EntryId;BoatId;BoatName;BoatCaptain;BoatVariant;ChangeCount;Comments;CoxId;CoxN
ame;Crew1Id;Crew1Name;Crew2Id;Crew2Name;Crew3Id;Crew3Name;Crew4Id;Crew4Name;Cre
w5Id;Crew5Name;Crew6Id;Crew6Name;Crew7Id;Crew7Name;Crew8Id;Crew8Name;Date;Desti
nationId;DestinationName;DestinationVariantName;Distance;EndDate;EndTime;LastMo
dified;SessionType;StartTime;WatersIdList;WatersNameList;LastModification
recorddata = int(10) NOT NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT
NULL;int(10) DEFAULT '1';int(10) DEFAULT NULL;int(10) NOT NULL;varchar(1024)
DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64)
DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64)
DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64)
DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64)
DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64)
DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64)
DEFAULT NULL;date DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(256) DEFAULT
NULL;varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;date DEFAULT NULL;time
DEFAULT NULL;bigint(20) NOT NULL;varchar(64) DEFAULT NULL;time DEFAULT
NULL;varchar(256) DEFAULT NULL;varchar(256) DEFAULT NULL; varchar(64) DEFAULT
NULL
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7Y3JlYXRldGFibGU7ZWZhMmxvZ2Jvb2s7RW50cnlJZDtpbnQ
oMTApIE5PVCBOVUxMO0JvYXRJZDt2YXJjaGFyKDY0KSBERUZBVUxUIE5VTEw7Qm9hdE5hbWU7dmFyY2
hhcig2NCkgREVGQVVMVCBOVUxMO0JvYXRDYXB0YWluO2ludCgxMCkgREVGQVVMVCAnMSc7Qm9hdFZhc
mlhbnQ7aW50KDEwKSBERUZBVUxUIE5VTEw7Q2hhbmdlQ291bnQ7aW50KDEwKSBOT1QgTlVMTDtDb21t
ZW50czt2YXJjaGFyKDEwMjQpIERFRkFVTFQgTlVMTDtDb3hJZDt2YXJjaGFyKDY0KSBERUZBVUxUIE5
VTEw7Q294TmFtZTt2YXJjaGFyKDY0KSBERUZBVUxUIE5VTEw7Q3JldzFJZDt2YXJjaGFyKDY0KSBERU
ZBVUxUIE5VTEw7Q3JldzFOYW1lO3ZhcmNoYXIoNjQpIERFRkFVTFQgTlVMTDtDcmV3MklkO3ZhcmNoY
XIoNjQpIERFRkFVTFQgTlVMTDtDcmV3Mk5hbWU7dmFyY2hhcig2NCkgREVGQVVMVCBOVUxMO0NyZXcz
SWQ7dmFyY2hhcig2NCkgREVGQVVMVCBOVUxMO0NyZXczTmFtZTt2YXJjaGFyKDY0KSBERUZBVUxUIE5
VTEw7Q3JldzRJZDt2YXJjaGFyKDY0KSBERUZBVUxUIE5VTEw7Q3JldzROYW1lO3ZhcmNoYXIoNjQpIE
RFRkFVTFQgTlVMTDtDcmV3NUlkO3ZhcmNoYXIoNjQpIERFRkFVTFQgTlVMTDtDcmV3NU5hbWU7dmFyY
2hhcig2NCkgREVGQVVMVCBOVUxMO0NyZXc2SWQ7dmFyY2hhcig2NCkgREVGQVVMVCBOVUxMO0NyZXc2
TmFtZTt2YXJjaGFyKDY0KSBERUZBVUxUIE5VTEw7Q3JldzdJZDt2YXJjaGFyKDY0KSBERUZBVUxUIE5
VTEw7Q3JldzdOYW1lO3ZhcmNoYXIoNjQpIERFRkFVTFQgTlVMTDtDcmV3OElkO3ZhcmNoYXIoNjQpIE
RFRkFVTFQgTlVMTDtDcmV3OE5hbWU7dmFyY2hhcig2NCkgREVGQVVMVCBOVUxMO0RhdGU7ZGF0ZSBER
UZBVUxUIE5VTEw7RGVzdGluYXRpb25JZDt2YXJjaGFyKDY0KSBERUZBVUxUIE5VTEw7RGVzdGluYXRp
b25OYW1lO3ZhcmNoYXIoMjU2KSBERUZBVUxUIE5VTEw7RGVzdGluYXRpb25WYXJpYW50TmFtZTt2YXJ
jaGFyKDY0KSBERUZBVUxUIE5VTEw7RGlzdGFuY2U7dmFyY2hhcig2NCkgREVGQVVMVCBOVUxMO0VuZE
RhdGU7ZGF0ZSBERUZBVUxUIE5VTEw7RW5kVGltZTt0aW1lIERFRkFVTFQgTlVMTDtMYXN0TW9kaWZpZ
WQ7YmlnaW50KDIwKSBOT1QgTlVMTDtTZXNzaW9uVHlwZTt2YXJjaGFyKDY0KSBERUZBVUxUIE5VTEw7
U3RhcnRUaW1lO3RpbWUgREVGQVVMVCBOVUxMO1dhdGVyc0lkTGlzdDt2YXJjaGFyKDI1NikgREVGQVV
MVCBOVUxMO1dhdGVyc05hbWVMaXN0O3ZhcmNoYXIoMjU2KSBERUZBVUxUIE5VTEw7TGFzdE1vZGlmaW
NhdGlvbjsgdmFyY2hhcig2NCkgREVGQVVMVCBOVUxM
```

- addcolumns

```
tablename = efa2logbook
recordfields = Crew9Id;Crew9Name;ClientSideKey
recorddata = varchar(64) DEFAULT NULL;varchar(64) DEFAULT NULL;varchar(64)
DEFAULT NULL
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7YWRkY29sdW1ucztlZmEybG9nYm9vaztDcmV3OUlkO3ZhcmN
oYXIoNjQpIERFRkFVTFQgTlVMTDtDcmV3OU5hbWU7dmFyY2hhcig2NCkgREVGQVVMVCBOVUxMO0NsaW
VudFNpZGVLZXk7dmFyY2hhcig2NCkgREVGQVVMVCBOVUxM
```

- autoincrement

```
tablename = efa2logbook
recordfields = EntryId
recorddata = int(10) NOT NULL
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7YXV0b2luY3JlbWVudDtlZmEybG9nYm9vaztFbnRyeUlkO2l
udCgxMCkgTk9UIE5VTEw_
```

- unique

```
tablename = efa2status
recordfields = Id
recorddata = varchar(64) DEFAULT NULL
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7dW5pcXVlO2VmYTJzdGF0dXM7SUQ7dmFyY2hhcig2NCkgREV
GQVVMVCBOVUxM
```

- insert, example 1:

```
tablename = efaCloudUser
recordfields = ID;Vorname;Nachname;Rolle;EMail
recorddata = 2;Peter;Pan;member;ppan@disney.com
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7aW5zZXJ0O2VmYUNsb3VkVXNlcnM7SUQ7MjtWb3JuYW1lO1B
ldGVyO05hY2huYW1lO1BhbjtSb2xsZTttZW1iZXI7RU1haWw7cHBhbkBkaXNuZXkuY29t
```

- insert, example 2:

```
tablename = efa2logbook
recordfields =
EntryId;BoatName;ChangeCount;Crew1Name;Date;DestinationName;StartTime
recorddata = 210;Single;1;Paula Pan;20.10.2020;1. Fähre;10:00
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7aW5zZXJ0O2VmYTJsb2dib29rO0VudHJ5SWQ7MjEwO0JvYXR
OYW1lO1NpbmdsZTtDaGFuZ2VDb3VudDsxO0NyZXcxTmFtZTtQYXVsYSBQYW47RGF0ZTsyMC4xMC4yMD
IwO0Rlc3RpbmF0aW9uTmFtZTsxLiBGw4PCpGhyZTtTdGFydFRpbWU7MTA6MDA_
```

- insert, example 3. You may repeat this twice to create an EntryId conflict:

```
tablename = efa2logbook
recordfields =
EntryId;BoatName;ChangeCount;Crew1Name;Date;DestinationName;StartTime
recorddata = 212;FollowMe;1;Captain Hook;21.10.2020;1. Fähre;10:20
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7aW5zZXJ0O2VmYTJsb2dib29rO0VudHJ5SWQ7MjEyO0JvYXR
OYW1lO0ZvbGxvd01lO0NoYW5nZUNvdW50OzE7Q3JldzFOYW1lO0NhcHRhaW4gSG9vaztEYXRlOzIxLj
EwLjIwMjA7RGVzdGluYXRpb25OYW1lOzEuIEbDg8KkaHJlO1N0YXJ0VGltZTsxMDoyMA__
```

- update, example 1:

```
tablename = efaCloudUser
recordfields = ID;efaCloudUserID
recorddata = 2;1143
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7dXBkYXRlO2VmYUNsb3VkVXNlcnM7SUQ7MjtlZmFDbG91ZFV
zZXJJRDsxMTQz
```

- update, example 2:

```
tablename = efa2logbook
recordfields = EntryId;EndTime
recorddata = 210;12:07
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7dXBkYXRlO2VmYTJsb2dib29rO0VudHJ5SWQ7MjEwO0VuZFR
pbWU7MTI6MDc_
```

- delete (you may also try a non existing EntryId):

```
tablename = efa2logbook
recordfields = EntryId
recorddata = 212
txc = MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7ZGVsZXRlO2VmYTJsb2dib29rO0VudHJ5SWQ7MjEy
```

- keyfixing, example 1:

```
tablename = efa2logbook
recordfields =
recorddata =
txc = MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7a2V5Zml4aW5nO2VmYTJsb2dib29r
```

- keyfixing, example 2:

```
tablename = efa2logbook
recordfields = EntryId
recorddata = 213
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7a2V5Zml4aW5nO2VmYTJsb2dib29rO0VudHJ5SWQ7MjEz
```

- select

```
tablename = efa2logbook
recordfields = EntryId;?
Recorddata = 210;>
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7c2VsZWN0O2VmYTJsb2dib29rO0VudHJ5SWQ7MjEwOz87Pg_
_
```

- synch, example 1:

```
tablename = efa2logbook
recordfields = EntryId;?
recorddata = 210;>
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7c3luY2g7ZWZhMmxvZ2Jvb2s7RW50cnlJZDsyMTA7Pzs*
```

- synch, example 2:

```
tablename = @all
recordfields = LastModified;?
Recorddata = 0;>
txc = MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7c3luY2g7QGFsbDtMYXN0TW9kaWZpZWQ7MDs-Oz4_
```

- synch, example 3 as duplicate request, synch & select:

```
for each sync and select:
    tablename = efa2logbook
    recordfields = EntryId;?
    Recorddata = 210;>
txc =
MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7c3luY2g7ZWZhMmxvZ2Jvb2s7RW50cnlJZDsyMTA7Pzs*Cnw
tZUZhLXwKNDQ7MDtzZWxlY3Q7ZWZhMmxvZ2Jvb2s7RW50cnlJZDsyMTA7Pzs*
```

- list

```
tablename = trips
recordfields =
```

```
recorddata =
txc = MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7bGlzdDt0cmlwcw__
```

- nop

```
tablename =
recordfields = sleep
recorddata = 3
txc = MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7bm9wOztzbGVlcDsz
```

- backup

```
tablename =
recordfields =
recorddata =
txc = MTs0MzsxMTQyOzEyM1Rlc3QhOzQzOzA7YmFja3VwOw__
```

Partner server communication

The server normally takes the listening part.

Starting with efaCoud V2.3.0_16 communication between efaCloud servers of different clubs is possible. Then the efaCloud server of club ARC takes the role of a client to the efaCloud server of club BRC.

From a configuration perspective a table is added which holds the client information and credentials at the ARC server, whereas for the BRC server a simple user entry is sufficient.

From a programming perspective the API must be addressed as a client rather than as a server. A special class EfaCloud_Api provides the client side interface.

To make sure, all write transactions are appropriately forwarded, the Socket class gets a “Socket_listener” interface as add-on and listeners can be added to the socket after instantiation. The on_socket_transaction() function of the listener is called for insert, update and delete call to the Socket. If yo do not want this to be called for a specific transaction (e.g. to prevent from regressive looping) you will have to manage this within the listener implementation. An example is in the EfaCloud_Partner class.

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 efaclient parts.

Efa client integration

In order to integrate with the efa clinet, 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.

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

efaCloud at the server side is lightweight programming, meant to have a minimum feature set for server based administration. This is both for performance and maintainability reasons. So please do not expect extensive functionality. Prime goal is robustness. You are always encouraged to add on your own pieces. PHP can be written with any text editor.

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. This keeps things simple and the need for software update low.

The technology is reused from a club software brg-intern and described at a different place – in German for that purpose. Please send drop request at efacloud.org, if you would need help at that point.

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);

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 7 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.