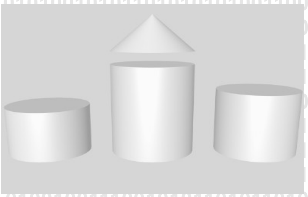


ODABA NG

ODABA HTTP server

run

run Software-Werkstatt GmbH
Winckelmannstrasse 61
12487 Berlin

Tel: +49 (30) 609 853 44
e-mail: run@run-software.com
web: www.run-software.com

Berlin, May 2023

Table of Contents

1 ODABA HTTP server

The ODABA HTTP server is a mean of communication with an ODABA database via HTTP internet protocol. When starting the server, a data source definition has to be passed to the server via configuration or ini-file. Usually, the server runs as daemon (Linux) or as Windows service. In order to debug server functions, one may also execute the server as console program.

```
OHTTPServer ini-file port (console program)
OHTTPServerD ini-file port (Linux daemon)
OHTTPWinServer ini-file port (Windows daemon)
```

The configuration or ini-file defines the data source. The port number defines the communication port.

The OHTTP server acts as a kind of database driver and should not be accessible directly via the internet. It is suggested to communicate with the internet via a back-end server. This allows performing required authentication, storing predefined queries or update procedures, which could be referenced by the application via simple URLs with a proper set of parameters e.t.c.

Examples provided in the document refer to the data model of the ODABA Sample database (see appended ODL definition).

The OHTTP server supports GET, POST, PUT, PATCH, DELETE and OSI POST requests. Since ODABA databases provide several extended database features, the semantic of POST, PATCH and PUT had to be changed slightly. Differences are described in documentation chapters for these request types. Operations supported by the server have to be defined explicitly in the configuration or ini-file passed to the server call.

Syntax elements used in different commands are described in detail in chapter **Symbol references**. Most syntax elements are JSON compatible.

Port number

The number of the communication port must be appended to each HTTP request:

```
server:port
```

This allows running servers for different databases by using different port numbers.

Initialize thread

Request are, usually running in different threads. ODABA provides for each thread a separate set of global variables. In order to use common global variables in all threads, the database event handler **`DatabaseContext::`**`doAfterInitialize()` may be overloaded in order to initialize common global variables.

1.1 HTTP Server demon

Running the service under Linux, typically the OHTTPServerD (demon application) is started, which allows running the server in the background. In order to close the server, one may simply kill the process or submit an HTTP shut down request, e.g.:

```
  http://localhost:2000/query/GET/OHTTPServerD/ShutDown
```

This is possible via a WEB browser or an an HTTP OSI command called from an Oshell.

In order to execute the request successfully, and protect the server agains unintended shut downs, one may define a ShutDown section in the ini-file:

```
[ShutDown]
administrator=admin-name
password=admin-password
```

Then, user name and password have to be added to the command.

One may also define a shut down request in the OHTTPMapper by setting

Type: `GET`
Name: `ShutDown`
Class: `OHTTPServerD`
User: *`admin-name`*
Password: *`admin-password`*

All settings are case sensitive.

2 HTTP requests

HTTP Requests may be send to `OHTTPServer` as **GET, PUT, PATCH, DELETE, POST** or **OSI** (POST) requests. For retrieving data from the database the **GET** request is used. Special update requests are supported by **PUT**,**POST** and **PATCH** request type. Deletion of database instances becomes possible by means of **DELETE** requests. Extended features are provided by the **OSI** POST request. Requests have the following common format:

```
http://server:port/location[?parameter]
[body]
```

`location` - Data to be accessed or updated. The syntax of location is described in detail in **Symbol Reference / location - HTTP URL**.

`parameter` - Several request types support parameters, which are well-defined keywords or parameters for controlling the behavior of the request. Supported parameters depend on request type.

`body` - For all HTTP requests except GET and DELETE, data may be passed in a `body,` which is, usually, interpreted as update data. Update data contains data for properties in `location` to be updated.

The `body` passed to the request may contain data for one or more properties (data items) in a JSON string as `values`.

`values` may be a single elementary `value` (`number` or `string`), but also a complex `object` or `array` (collection). Details for values are described in **Symbol reference / Values, formats and encoding**.

In case of collection source property, `body` data may be used also for locating an instance. When a collection URL has been passed in `location`, the server tries to locate an instance in the collection via values for key component properties passed in `body` (POST). This allows accessing instances with multiple key components.

The response to any request is a JSON string (`values` - UTF8 encoded), which contains either response data or an error message (see **Symbol reference / Request result (response)**).

2.1 GET request

The **GET** request is passed in the URL in `location` (see **Symbol reference / location - HTTP URL**).

`location` - is the part of URL after server and port specification. It defines an access path for the source property that addresses an elementary data item, an object instance (complex data type) or a collection or array of instances. When the source property is a collection or an array, all instances in the source property are returned in the response.

The source property may also refer to a function name, as long as the function called does not require parameters. In this case, the response contains the function result. Functions may be **`odaba::Property`** functions, but also OSI functions implemented in the class of preceding property's data type.

An error description is returned as JSON `object` (see **Error handling**) in case of key conflicts or other error(s) in the request definition.

```
// simple GET requests
// get all children of a person
http://127.0.0.1:8888/Person::Persons/P123/children

// get person LOID
http://127.0.0.1:8888/Person::Persons/P123/instanceLoid

// get number of employees for first company
http://127.0.0.1:8888/Company/0/employees/*/count
```

Response to GET request

The GET request returns a JSON `object` (syntax of `object` and other syntax elements referenced here is described in chapter **Symbol reference / Values, formats and encoding**. In case of error, a `string_value` is returned containing an error message. The response for a successful GET request is a JSON `object` with one `named_value`:

```
      { name: values }
```

`name` - is the name of the source property in `location`. In case of elementary source property (elementary attribute, **`BLOB`**, **`MEMO`**), `values` is a `value`, i.e. a string, number or **true** or **false**.

In case of a collection source, `values` is an `array`.

```
      { name: [ object, ..., object ] }
```

When `location` refers to an instance, `values` is a single `object`.

```
      { name: { named_value, ..., named_value } }
```

Within returned `object`s, each `name` is a property name as defined for the complex data type of instance and all its base types.

`values` for collection properties within an `object` return an `array` of URL instances or an empty `array`, when the collection is empty:

```
name: [ { url: LOID/number },...,{ url: LOID/number } ]
name: [ ]
```

Array attributes within an `object` return a `values` array:

```
name: [ values,...,values ]
```

Complex data type attributes within an `object` return an `object`.

```
name: { named_value,...,named_value }
```

More details for `values` are described in **Symbol reference / Values, formats and encoding**.

```
// get number of employees for first company
http://127.0.0.1:8888/Company/0/employees/count returns:
{ count: 485 }

// Get person's children
http://127.0.0.1:1234/Persons/ID123/children returns:
{ children: [
  { pid: "P123" name: "Miller", first_name_0: "Paul",
    first_name: [ "Henry", "", "" ], birth_date: "1966-11-11",
    sex: "male", married: false,
    income: 2500.00, age: 52, children_income: 3807.22,
    notes: "this might be a very long text in C-string format\n",
    address: [ { url: LOID/12346127 } ],
    children: [
      { url: LOID/12346000 },
      { url: LOID/12346002 } ],
    parents: [ { url: LOID/12346001 } ] ,
    employee: [ ]
  }, { ... next child instance .... }, ..., {...}
] }
```

2.2 PUT request

A **PUT** request is used to create new or update existing instances addressed in `location` by passing appropriate data in a `body`. The PUT request `body` contains data to be stored.

When an instance already exists at required `location`, properties of the existing instance referenced in `body` will be updated, when `replace` is passed as parameter. Properties of an existing instance not referenced in the `body` are reset to their initial state. In order to prevent single properties in existing instances from being reset, one should use PATCH rather than PUT.

In case of a collection source (no locator passed), a new instance is inserted/appended to the collection when the collection is unordered or ordered by `__AUTOIDENT` (auto-number). Otherwise, collection sources will cause an error.

`location` - is the property source which should address a single (not existing) instance (see **Symbol reference / location - HTTP URL**). Since PUT is creating a new instance, the source property for PUT must not be an elementary source property. Usually, the source property in the PUT `location` refers to a non existing instance. PUT first tries to insert the new instance referenced in `location` before updating properties with data passed in `body`.

`body` - is a JSON `object` or an `object` array passed with the request, which contains name/value pairs for properties to be updated and data to be stored (see description in **Symbol reference / body - request data for HTTP requests**). Properties passed in `body` must refer to attributes, **`MEMO`** or **`BLOB`** properties.

`replace` - In case of passing the `replace` parameter, an instance that already exists will be updated. Otherwise, the request terminates with an error message.

When terminating normally, PUT returns following response:

- `created` - when a new instance has been created
- `updated` - when data has been replaced

The identifying key value for new instances should be passed in the PUT `location` (part of URL). When passing unique key component values in the body, those should precede all other attributes and references.

An error description is returned as JSON `object` (see **Error handling**) in case of key conflicts or other error(s) in the request definition and no changes are made.

2.3 PATCH request

A **PATCH** request is used to partially update an existing instance addressed in `location` by passing appropriate data in a `body`. The PATCH request `body` contains data to be stored. When no instance exists at required `location`, a new instance is created, when `insert` has been passed as parameter. Otherwise, the request returns an error.

`location` - is the property source (see **Symbol reference / location - HTTP URL**), which must address a single instance or an elementary property (attribute, **`MEMO`** or **`BLOB`**). Since PATCH is, usually, updating existing instances, `location` must refer to an existing instance. PATCH first tries to locate the instance referenced in `location` before updating properties with data passed in `body`.

`body` - is a JSON `object` or `value`, which contains a `value` or `name`/`values` pairs for properties to be updated and data to be stored (see `body` definition in **Symbol reference / body - request data for HTTP requests**). Properties passed in `body` must refer to attributes, **`MEMO`** or **`BLOB`** properties.

`insert` - when passing the parameter `insert` (update or insert), instances are created when not yet existing. When not being passed and no instance exists at required `location`, the request fails.

When `location` refers to an elementary property, the data (`value`) passed in `body` is stored to the property. When `location` refers to an array attribute, `body` should contain a `value array` and values are stored according to position in the array. Attribute elements not referenced in the `value array` (less values than attribute elements) remain unchanged. Passing too many values will cause an error.

When `location` refers to an object instance, the data passed in `body` must be an `object` containing `name/values` pairs. Names must refer to property names defined for the object. Properties not referenced remain unchanged. Passing `null` as value will reset the property value to its initial state. Undefined property names will cause an error.

When terminating normally, PATCH returns following response:

- `updated` - when data has been updated successfully
- `created` - when a new instance has been created (`insert` passed)

The identifying key value for instances to be created should be passed in the PATCH `location` (part of URL). When passing unique key component values in the body, those should precede all other attributes and references.

An error description is returned as JSON `object` (see **Error handling**) in case of key conflicts or other error(s) in the request definition and no changes are made.

2.4 POST request

A **POST** request is used to partially update existing instances at `location` addressed by passing appropriate data in a `body`. In contrast to PUT or PATCH, post allows inserting/updating multiple instances passed in a JSON `array` in `body`. Before inserting or updating instances, POST tries to locate instances by means of key attributes passed in `body` `object`s. In contrast to PUT and PATCH, POST does not allow updating related references. In order to update references, additional POST request have to be submitted.

`location` - is the property source which must address a collection property (see **Symbol reference / location - HTTP URL**). Since data for locating instances is passed in `body`, addressing an instance or elementary property in `location` will cause an error.

`body` - is a JSON `object` or `array` (of `object`s) which contain `name`/`values` pairs for key component attributes and properties to be updated (see `body` definition in **Symbol reference / body - request data for HTTP requests**). Properties passed in `body` must refer to attributes, **`MEMO`** or **`BLOB`** properties.

`noreplace` - In case of passing the `noreplace` parameter, an instance that already exists, will not be updated.

`noinsert` - In order to avoid creating new instances, `noinsert` may be passed instead.

Since POST first tries to locate an instance by main key attribute values passed in the object, POST does not allow changing main key attribute values. In order to update main key attribute values, one should use PATCH.

When creating or updating instances conflicts with parameter passed, the request terminates with an error and no changes are made.

Properties of an existing instance not referenced in the body remain unchanged. Passing `null` for a property will reset the property value to its initial state.

When terminating normally, POST returns following response:

- `updated` - when data has been updated successfully
- `created` - when a new instance has been created

When creating new instances, unique key component values for each instance have to precede all other attributes and references in the JSON instance (part between { ... }).

An error description is returned as JSON `object` (see **Error handling**) in case of key conflicts or other error(s) in the request definition and no changes are made.

2.5 DELETE request

In order to remove an instance from a collection or to delete it completely, the **DELETE** request may be used:

`location` - is the property source (see **Symbol reference / location - HTTP URL**), which must address a single existing instance. Referring to instances via `LOID` is not allowed for a DELETE request.

Whether an instance is removed from the collection or deleted, depends on the ownership status of the source property. Instances in owning collections are deleted, always.

`delete` - In case of passing the `delete` parameter, an instance is completely deletes also when not being owned by the source property.

When terminating normally, DELETE returns following response:

- `deleted` - when data has been deleted successfully

An error description is returned as JSON `object` (see **Error handling**) in case of key conflicts or other error(s) in the request definition.

3 OSI POST Request

An **OSI POST** request executes an OSI expression or an OSI access path for evaluating the source property and returns a JSON response. An OSI request allows retrieving data but also updating database content.

```
http://server:port/osi
body
```

The URL for an OSI POST request only contains the keyword `osi` (not case sensitive). The request is passed in the `body` of the OSI POST request

`body`: The OSI POST request is passed in the `body` with following structure (see **Symbol reference / body - request data for OSI POST**):

```
source => { fieldlist }
```

`source` - The source property defines the data source for the request, i.e. the vertical dimension of data to be processed. A source property may refer to an elementary data item, to an instance or to a collection of instances. In case of including functions in the source expression, the function result represents the source property for creating the result. Following source definition formats are supported:

- `access path` - is a valid OSI access path for the database. Usually, an access path consists of extent or property names and function calls. Access path elements are separated by dot (.), e.g.

  `Persons().income('netto')` or
  `Persons("ID123").income('netto')`).

- `function` - is a valid OSI inline function enclosed in `{ ... }`.

- `expression` - Expressions are valid OSI expressions (access paths combined with operations)

Access paths, functions and expressions have to be defined within the context of the database.

`fieldlist` - An OSI POST request returns data for an elementary data item (`value`), an instance (`object`) or a collection (`array`). In case of complex data (complex data type for source property), the field list defines the horizontal result dimension (object elements). In case of referring to an elementary data type source property, field lists are not supported and will cause an error. A field list allows reducing the number of values returned from an instance but also adding derived or related information. Moreover, field definitions allow defining hierarchical response structures. A `fieldlist` contains one or more field definitions separated by comma:

```
fielddef, ..., fielddef
```

`fielddef` - Field definitions may be provided in different ways:

- `name` - Name of a property defined for the data type of the preceding (upper) `source`. In this case, the value name in the response is the same as the property name. In order to rename the value in the output, one may define `name: source`, where `name` is the value name in the response and `source` is the property name for the value to be assigned..
- `name: source` - In order to return fields in the response that are not defined for source property (related or derived information), one may pass a `source` definition as access path, function or OSI expression (see `source` definition above).
- `name: source => { fieldlist }` - In order to define hierarchical response, one may define a `source` (see definition above) appended by a field list.

Sources in field lists have to be defined within the context of the preceding (upper) `source`, i.e. when the `source` returns a **`Person`** property, field sources must refer to access paths or inline expressions valid in the context of **`Person`**.

When the defined `source` is an elementary property, a field list must not be defined and the item is returned as `value`. When source defines an instance or collection without defining a `fieldlist`, the part of the response for this source corresponds to the response of a GET request.

In case of error(s) in the request definition or while executing the request, an error description is returned as JSON `object` (see **Error handling**) in case of error(s).

More details about `source` and `fielddef` syntax are provided in **Symbol reference / body - request data for OSI POST**. How to define OSI access paths, functions and expressions is described in ODABA Script Interface.

```
// OSI POST body
Persons('P123') => {
    name: { name + ', ' + first_name; }, // inline function
    employed_at: employees(0).company(0).name, // access path
    children: children => {
      name: name + ', ' + first_name, // expression
      loid: instanceLoid,
      age
    }
}
```

Response to an OSI POST request

The OSI POST request returns a JSON `object` (syntax of `object` and other syntax elements referenced here is described in chapter **Symbol reference**). In case of error, a `string_value` is returned containing an error message. The response for a successful OSI POST request is an `object` with one `named_value`:

```
{ name: values }
```

An OSI POST `request` may define a request hierarchy. Here, the response structure for a request with one hierarchy level is discussed, which is repeated for all deeper levels.

The top element `name` is the name of the source property. For a subordinated `source` definitions, which are always part of a `fielddef` in a field list, the `name` of `fielddef` is used.

The structure of `values` following `name` depends on the complexity of `source`. `values` for collection properties and array attributes with complex data type are returned as `array` of `object`(s):

```
name: [ object, ..., object ]
name: [ ]
```

When the collection property is empty, `[]` is returned as empty `array`. For elementary array attributes an `array` of values is returned:

```
name: [ value, ..., value ]
```

For a single complex data type property an `object` is returned:

```
name: { named_value, ..., named_value }
```

For an elementary source property (elementary attribute, **`MEMO`** or **`BLOB`**), a `value` is returned:

```
name: value
```

When returning an `object`, `named_value`(s) within the object are, usually, names listed in the `fieldlist` for the source. When no field list has been defined, all properties of the complex data type are included (same as for GET). For collection properties within returned instance(s) , an `array` of URL `object`s is returned (see GET response).

4 QUERY POST Request

Because the complexity of OSI POST requests, OHTTP provides a feature of pre-defined queries, which may be executed as QUERY POST requests:

`http://server:port/query/`*`type`*`/`*`class`*`/`*`method`*`?`*`parameters`*

Mainly, query definitions provide a mapping between application and database terminology. How the mapping will be resolved depends on the request type defined in the query.

The URL for an QUERY POST request only contains the keyword `query` (not case sensitive), followed by the request type, the class name and the query definition name. Request data is passed in the `body` of the QUERY POST request. When omitting the class name, the method must be defined as global method.

`type` - Typically, QUERY requests (request type) are defined as OSI, PUT or PATCH requests. In case of PUT/PATCH requests, values to be assigned are taken from parameters passed to the request, where parameter names correspond to application defined names. For OSI QUERY definitions, one may define hierarchical requests.

`class` - defines the object type, the query definition refers to. In case of global queries, * has to be passed as class name.

`method` - The method (query) defines a request body in a formalized way. A query method is defined in global (*) or class context, i.e. `class`/`methode` do identify a query.

QUERY definitions are stored in the resource database using a system-defined type HTTP_Query, which also provides a check feature. QUERY definitions may be provided as direct, reference or hierarchical field definitions (HTTP_Field). Direct field definitions are defined for elementary data fields. Complex data type field may either refer to another QUERY definition or to a subordinated field definition list. Query definition resist on server side and are resolved there to corresponding HTTP requests discussed in previous chapters.

Because OSI POST requests allow nearly everything manipulating the database content (as `GET`, `PUT`, `PATCH` and `DELETE` do), QUERY POST request is also a mean of security. By restricting access to `QUERY`, (`allow=QUERY` in the server ini-file), only predefined queries may be executed.

4.1 Running queries

Queries are always submitted as HTTP GET requests containing server is, port, query class and query name as well as HTTP method and optional parameters:

```
http://server:port/query/type/[class|*]/method?p1=v1&p2=v2...
```

`server` – server name ir ip addressed

`port` – port number that has been passed starting the server

`type` – HTTP method to be executed internally (GET, OSI, PUT, PATCH or DELETE)

`class` – Class name or complex application data type that has defined the query. When the data source referenced is a global (static) method, '*' may be passed instead of a class name.

`method` – Query name for a query defined for the class.

`pn` – any number of parameters. Parameter names will replace the values in the resolved method and must correspond to field names or variables defined in data sources. Variable names in data sources has to be preceded by '?' (as `?id` for `id` parameter.

`vn` – parameter values contain either strings or numerical data. Typically, numerical parameters are passed as collection positions, while string data is used for identifiers. String data should be quoted ("…"). When the parameter value is not a numerical value, quotes may be omitted for string values.

4.2 GET Query

Get queries will return all data for an instance. Instance attributes are shown using attribute names (using database terminology). The mapping for a GET query would look like:

Class/type	Query/fvariable	Source	Comment
Company/GET	**Show**	Company(?id)	?id is a parameter passed with the HTTP request
*/GET	**GetByLoid**	LOID(?id)	Global LOID query (without class name)

A GET query does not define field mappings. The following (test) request will be generated:

```
Test: http://localhost:2000/query/GET/Company/Show?id=2
GET
user
pwd
/Company(?id)
```

The first four lines are generated for test purposes. The first line shows the URL for calling the query with GET method. The `/Company(?id)` line below defines the HTTP GET command executed internally.

Collection content is shown by listing links (`LOID/number`) for all instances. The `id` passed, is a collection position in this example. The request returns:

```
{ Company: { name: "My company",
employees: [ { url: LOID/7659 },
{ url: LOID/1113 },
... 102 links follow
{ url: LOID/7638 } ],
cars: [ { url: LOID/93 },
{ url: LOID/96 },
{ url: LOID/99 },
{ url: LOID/102 },
{ url: LOID/105 },
{ url: LOID/108 },
{ url: LOID/111 } ] } }
```

In order to get data for referenced instanced, a global request could be submitted for retrieving data by local object identifier(LOID).

```
Test: http://localhost:2000/query/GET/*/GetByLoid?id=1113
GET
user
pwd
/LOID(?id)
```

The query will return data for the first employee referenced in companies `em-ployees` collection:

```
{ VOID: { pid: "P127", name: "Figpk", first_name: [ "Fdvlqrjcmk","","" ],
  birth_date: 1965-03-10, sex: male, married: false, income: 3064.00,
  age: 57, children_inc: 32513.00, notes: null,
  location: [ ],
  children: [ { url: LOID/5781 },
    { url: LOID/5772 },
    { url: LOID/5763 },
    { url: LOID/5754 },
    { url: LOID/5658 },
    { url: LOID/5649 } ],
  parents: [ { url: LOID/1314 },
  { url: LOID/948 } ],
  employee: [ { url: LOID/1113 } ],
  company: [ { url: LOID/16 } ],
  used_cars: [ ] } }
```

Attention: Providing a global LOID query allows WEB application to access any kind of data in the database, which is an advantage, but might also be a risk.

4.3 OSI query

A complex mapping for retrieving data could look like (sample database):

Class/method	Query/field name	Source	Comment
Person/OSI	**Children**	Persons(?id)	?id is a parameter passed with the HTTP request
	name	name	
	firstname	first_name	All first names
	income		Source name is the same as target
	children	children()	collection
	name	name	Child name
	firstname1	first_name(0)	First first name
	Income	income	Child’s income
Company/OSI	**EmployeesCount**	Company(?id)	Calling a system function
	count	employees.count	Number of employees

For the first query (**Children**) the mapping, the mapping tool generates the following query:

```
Test: http://localhost:2000/query/OSI/Person/Income?id="P124"
POST
user
pwd
/osi
Persons(?id) ==> {
  name : name,
  firstname1 : first_name(0),
  income : income,
  children_income : children_inc,
    children : children() ==> {
    name : name,
    firstname : first_name,
    income : income } }
```

The first four lines are generated for test purposes. The first line shows the URL for calling the query with GET method. Below, starting with the `/osi` line, the generated POST request follows as executed by the OHTTP server, because OSI has been defined as request type (method) for the query (mapping). The result returned from this request looks as:

```
{ Persons: { name: "Miller", firstname: [ "Rznmgxpiwb", "", "" ],
  income: 0.00, children_income: 12643.00,
  children: [
  {name: "Ycngp", firstname1: "Dcwktgfuxj", income: 8261.00},
  {name: "Czjso", firstname1: "Ygkordeqhx", income: 4382.00} ]
} }
```

Instead of property paths, access paths may contain also function calls to system or application functions. The second query (**EmployeesCount**) returns the number of employees in the company passed in '?id'.

```
Test: http://localhost:2000/query/OSI/Company/Employees?id=2
POST
user
pwd
/osi
Company(?id) ==> {
count : employees.count }
```

Instead of a company identifier, a collection position (`2`) has been passed in `id`.

4.4 PUT query

In order to create new instances, one may use PUT request (queries). The field mapping is the same as for OSI request:

Class/method	**Query/field name**	**Source**	**Comment**
Person/PUT	**Create**	Persons	Person collection
	name	name	Family name for person to be created
	firstname1	first_name(0)	First first name
	firstname2	first_name(1)	Second first name
	birth	birth_date	Person's birth date
	married	married	Marital status
	id	pid	Person identifier

In order to assign values to the field names mapped, parameters with the same name have to be passed with the query HTTP GET request: The POST request generated from this mapping looks like:

```
Test: http://localhost:2000/query/PUT/Person/Create?
      id=P1115&name=Hover&firstname1=Paul&firstname2=John&
      birth="1.1.1999"&married=true
PUT
user
pwd
/Persons
{
  name : ?name,
  first_name(0) : ?firstname1,
  first_name(1) : ?firstname2,
  birth_date : ?birth,
  married : ?married,
  pid : ?id }
```

In case of success, the request returns an internal code 3 and a message:

```
{ code=3, message="Instance(s) created" }
```

The PUT query is an elementary way for creating new instances and usually not able to execute related requirements. When, however, a method has been implemented for creating a new person, which automatically creates a new person identifier, one could also call the implemented method:

Class/method	**Query/field name**	**Source**	**Comment**
Person/OSI	**NewPerson**	Persons	
	result	CreateNew(?name,?firstname,?birthdate,?married)	Create new person by method call

In order to assign values to the field names mapped, parameters with the same name have to be passed with the query HTTP GET request: The POST request generated from this mapping looks like:

```
Test: http://localhost:2000/query/OSI/Person/NewPerson?name="Miller"&
      firstname=Paul&birthdate="1.1.1999"&married=true
POST
user
pwd
/osi
Persons ==> {
  result : CreateNew(?name,?firstname,?birthdate,?married) }
```

The function result is returned as result value according to the mapping:

```
{ Persons: { result: true } }
```

4.5 PATCH query

In order to update data, a PATCH query may be defined, which allows updating specific (defined) values in an instance.

Class/method	**Query/field name**	**Source**	**Comment**
Person/PATCH	**Update**	Persons	Person collection
	name	name	Family name for person to be created
	birth	birth_date	Person's birth date
	married	married	Marital status

The PATCH request generated from this mapping looks like:

```
Test: http://localhost:2000/query/PUT/Person/Create?
      id=P124&&name=Hover&birth="1.1.1999"&married=false
PATCH
user
pwd
/Persons(?id)
{
  name : ?name,
  birth_date : ?birth,
  married : ?married,
  pid : ?id }
```

In case of success, the request returns an internal code 3 and a message:

```
{ code=3, message="Instance(s) updated" }
```

Similar to the PUT query, this is an elementary way for updating instances and usually not able to execute related actions. When, however, a method has been implemented for updating person data, one could also call the implemented method.

4.6 DELETE query

DELETE allow deleting single instances.

Class/method	Query/field name	Source	Comment
Person/DELETE	**Delete**	Persons(?id)	?id is passed as parameter

A DELETE query does not define field mappings.

```
Test: http://localhost:2000/query/DELETE/Person/Delete?id="P123"
DELETE
user
pwd
/Persons(?id)
```

In case of success, the request returns an internal code 4 and a message:

```
{ code=4, message="Instance(s) deleted" }
```

In order to perform more complex Delete requests, which will also perform related actions, one could also call an implemented deletion method (see “PUT request”).

4.7 FILE query

FILE allows transferring a file from the server. The file location may be an attribute of a data item.

Class/method	Query/field name	Source	Comment
Person/FILE	**ImagePath**	Persons(?id).image	?id is passed as parameter

A FILE request returns a fixed set of attributes

```
Test: http://localhost:2000/query/File/Person/ImagePath?id="P123"
FILE
user
pwd
/Persons(?id)
```

In case of success, the request returns a file message:

```
{ path: "../my_pic.jpg", mimetype: "image/jpeg", base64: "..." }
```

Path contains the file location read from the Person::image attribute. The mime type is derived from file extension. The file data is returned as base 64 character string.

In case of errors, the response contains an error reason passed in the error value.

5 OHTTP Mapping Tool (HTTPMapper)

ODABA provides a HTTP mapping tool in order to simplify the specification of requests. The mapping tool allows defining any number of queries. Query definitions are stored in the resource database for the application.

Query definitions support simple (one instance) queries, but also complex hierarchical queries. A set of queries may contain data retrieval, update, create or delete queries. In contrast to direct POST requests, queries allow single instance updated and creations, only.

When defining queries, a request type has to be set. Usually, different request types are used for following actions:

- retrieve data completely: GET
- retrieve selected fields: OSI
- Update instance data: PATCH
- Create new instance: PUT
- Delete an instance: DELETE
- Get file content: FILE

Usually, POST is not used as query type.

Each mapping (query) is assigned to a class, which defines a complex data type in the application database. Class name and query (mapping) name must be unique, i.e. names for mappings belonging to the same class must differ. Except GET and DELETE queries, queries require a field list, which may be hierarchically structured. The field list defines a set of target names (field names) and data sources. For simple cases, data sources are simple attribute names of the datatype (class) defined for the mapping. Data sources may be defined, however, also as access path (e.g. `employee(0).company(0).name` in order to get the company name where the person is employed.

When the data source for a field is referring to a property with complex data type (referenced instance, embedded instance) or a collection, a subordinated field list may be defined, which refers to data source properties of the complex data type for the referenced instance(s).

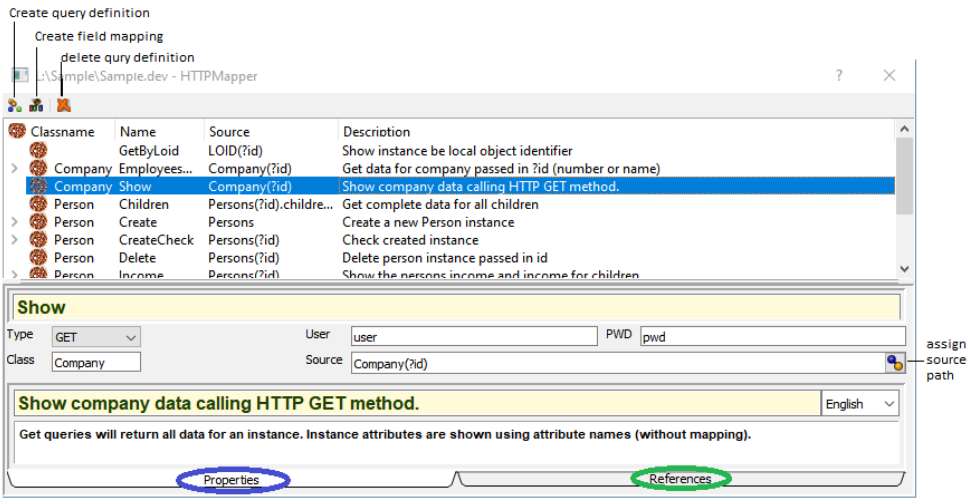
The HTTPMapper is called with an .ini or configuration file. Supposed, that ODABA has been installed in an **odaba** directory, it might be called as follows:

```
  odaba/ode ini_file -PROJECT=HTTPMapper -DATDB=Sample.dev
```

The ini-file provides database locations for system databases. The applications resource database (DATDB) is passed in this example as parameter, but could also be defined in the ini-file:

```
// ini-file ODE.INI for ODE tools
[SYSTEM]
DICTIONARY=odaba/ode.sys
[ode]
SYSDB=odaba/ode.sys
RESDB=odaba/ode.dev
;DATDB=Poject database passed as parameter
NET=YES
SYSAPPL=YES
ONLINE_VERSION=YES
PROJECT_DLL=Designer
CTXI_DLL=AdkCtxi
[HTTP]
Server=localhost
Port=2000
```

In order to run tests, the HTTP Server and port have to be defined, too. After starting the HTTP mapper, the tool main window appears:



Classname	Name	Source	Description
	GetByLoid	LOID(?id)	Show instance be local object identifier
Company	Employees...	Company(?id)	Get data for company passed in ?id (number or name)
Company	Show	Company(?id)	Show company data calling HTTP GET method.
Person	Children	Persons(?id).childre...	Get complete data for all children
Person	Create	Persons	Create a new Person instance
Person	CreateCheck	Persons(?id)	Check created instance
Person	Delete	Persons(?id)	Delete person instance passed in id
Person	Income	Persons(?id)	Show the persons income and income for children

Show

Type	GET	User	user	PWD	pwd
Class	Company	Source	Company(?id)		

Show company data calling HTTP GET method. English

Get queries will return all data for an instance. Instance attributes are shown using attribute names (without mapping).

In order to test query definitions, one may change to the **References** tab.

5.1 Create query definition

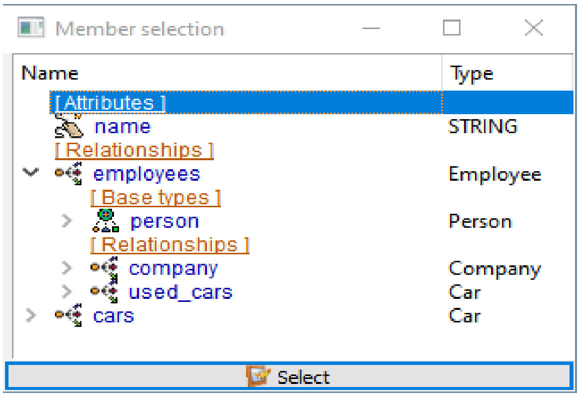
With the **create query definition** button a new query will be defined after entering data type/class name and query name. Fields below the list (**Type**, **Source**) are mandatory and must be filled. **User** and **PWD** are optionally and required for test, only.

Query definitions can be created, only, when the list is empty or a query definition is selected in the list.

The lower part is used for documentation purpose, only.

5.1.1 Assign source path

In order to assign a data source path, additional support is provided when pressing the **assign source path** button:



A selection three pops up showing the class members of the defined data type, or for fields for the data type evaluated for the parent field/query.

Pressing the select button writes the data source path for the selected property to the **Source** field.

Selecting employees in the example produces the following path:

```
Company(?id).employees(-1)
```

The placeholder `?id` is created by default as parameter for an identifying key or position in the top collection. The top collection is assumed to have the same name as the data type, which has to be changed, when this is nit the case. For referenced collections (as `employees`) either **0** (single instance reference) or **-1** is generated. In case of **-1**, the value should be replaced bay a valid locator (key or position) or removed in order to indicate, the complete collection should be processed (...`employees()`). The path could also include selection rules or any other supported **`odaba::Property`** function as wall as functions implemented in the application:

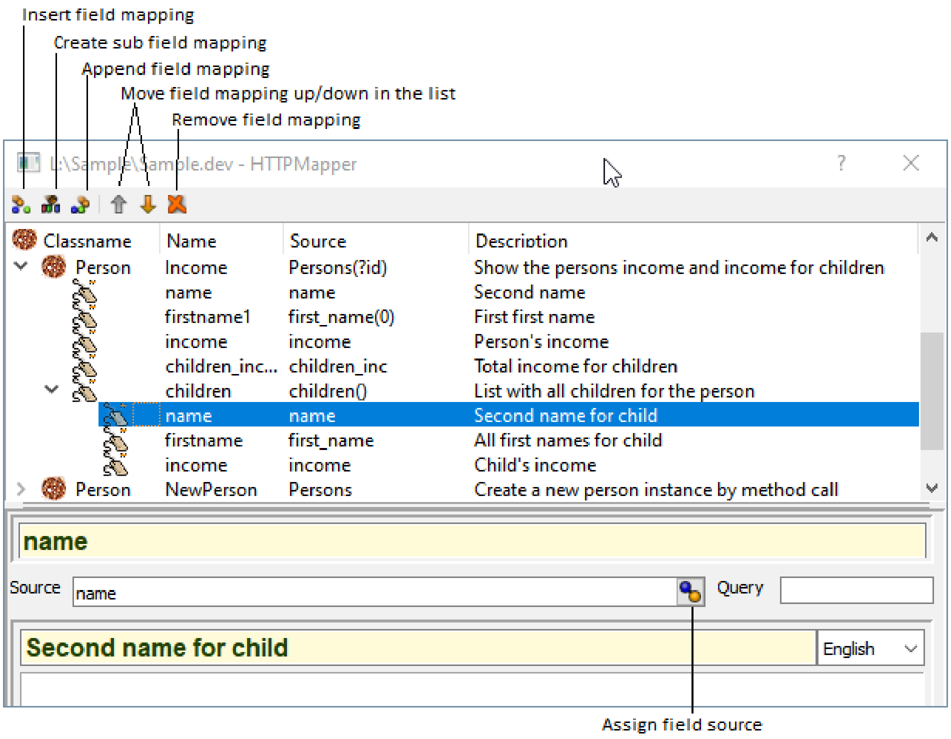
```
Company(?id).employees.where(married==true)
```

for showing married employees, only. In general, everything that defines a valid ODABA access path or operation may be passed as data source.

5.1.2 Create field mapping

Query types except GET and DELETE require specific field mappings. Field mappings may be defined as subordinated field mappings for a query definition or for a field mapping. Thus, one may define hierarchical structure reflected in the returned or passed JSON string.

Create field mapping requires a field name and shows the mapping definition afterward in the mapping tree.



Classname	Name	Source	Description
Person	Income	Persons(?id)	Show the persons income and income for children
	name	name	Second name
	firstname1	first_name(0)	First first name
	income	income	Person's income
	children_inc...	children_inc	Total income for children
	children	children()	List with all children for the person
	name	name	Second name for child
	firstname	first_name	All first names for child
	income	income	Child's income
Person	NewPerson	Persons	Create a new person instance by method call

name

Source name Query

Second name for child English

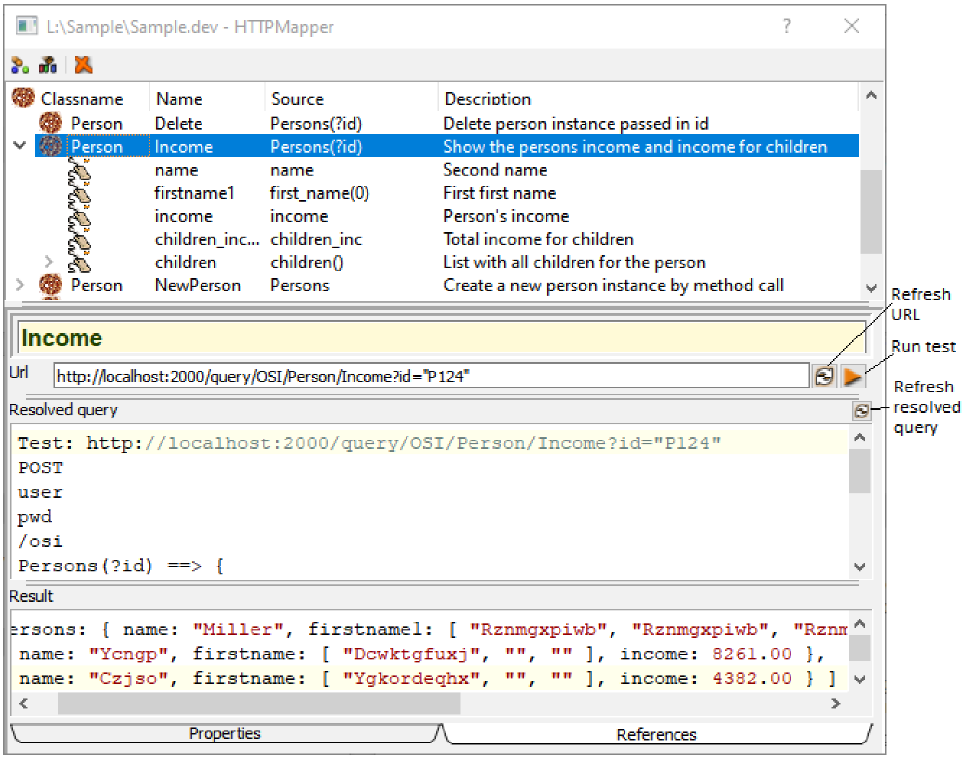
New field mappings may be inserted in front of the selected line (**Insert field mapping**) or appended at the end of the current hierarchy level (**Append field mapping**). In order to create a subordinated field mapping - typically for complex data types referenced in the selected field mapping – the **Create sub field mapping** button should be used. Field mappings may be moved up or down in the selected hierarchy level using the **Move field mapping up/down** button. The **Remove field mapping** button will remove the selected field mapping ans all subordinated field mappings.

The **Source** path is the only optional field for the mapping. Similar to the query definition, a selection list may be shown using the **Assign field source** button.

Fields below are used for documentation purpose, only.

5.2 Test query definition

After activating the References tab, test data will be shown:



Classname	Name	Source	Description
Person	Delete	Persons(?id)	Delete person instance passed in id
Person	Income	Persons(?id)	Show the persons income and income for children
	name	name	Second name
	firstname1	first_name(0)	First first name
	income	income	Person's income
	children_inc...	children_inc	Total income for children
	children	children()	List with all children for the person
Person	NewPerson	Persons	Create a new person instance by method call

Income

Url http://localhost:2000/query/OSI/Person/Income?id="P124"

Resolved query

```
Test: http://localhost:2000/query/OSI/Person/Income?id="P124"
POST
user
pwd
/osi
Persons(?id) ==> {
```

Result

```
ersons: { name: "Miller", firstname1: [ "Rznmgxpiwb", "Rznmgxpiwb", "Rznm
 name: "Ycngp", firstname: [ "Dcwktgfuxj", "", "" ], income: 8261.00 },
 name: "Czjso", firstname: [ "Ygkordeqhx", "", "" ], income: 4382.00 } ]
```

The URL shown in the URL field may be generated using the **Refresh URL** button. In order to avoid overwriting parameters added to the URL, the field has to be cleared before refreshing. Before running the query, usually parameters have to be added to the URL after ? for variable fields or ?-names in the source path. The resolved query shows the query as being defined in an offline test script, i.e. the first four lines are not part of the request, but used for test purposes, only.

When changing URL parameters or source definitions, it may become necessary to refresh the content of the Resolved query field using the **Refresh resolved query** button. This is, however, not necessary for running the request, but for documentation, only.

With the **Run test** button, one may submit the request defined by means of the query, supposed, that an HTTP server with the address defined in the URL is running. The query result will be displayed in the **Result** field afterward.

6 Error handling

There are two kind of errors that may appear. Server errors are errors that appear when starting up the server. In this case, an error is written to the error log defined in the TRACE option (environment) variable.

Another type of errors are response errors, i.e. errors caused by incorrect HTTP requests or errors that appear when processing the request. In case of response errors, a JSON string containing error code and message is returned:

```
{ code=number, message="string" }
```

`number` – is an internal termination code. Usually, error codes returned to a request are negative. Termination codes for requests executed successfully (inserted, deleted, etc.) are positive.

`string` - more detailed error message. The error message may also contain more detailed information in case of database access errors or exceptions.

`string` and `number` follow the syntax rules described in **Symbol reference / Values, formats and encoding**.

6.1 Internal termination codes

Internal termination codes are set for different reasons.

In case of normal termination, the internal error code is set to one of the following reasons:

- 1 - Ok, but no content returned (DELETE, PUT, PATCH) ==> 200
- 2 - Instance(s) updated ==> 201
- 3 - Instance(s) created and/or updated ==> 201
- 4 - Instance(s) deleted ==> 201

Server start errors:

- -31 - Invalid authentication type (basic/none) ==> 401
- -32 - Invalid data source defined in server ini-file. No database available ==> 401

Request errors

- -10 - Syntax error in URL ==> 404
- -11 - Missing locator in URL (// not allowed)
- -20 - Invalid URL (property or method not found) ==> 404
- -21 - Invalid URL (could not locate instance) ==> 409
- -30 - Authentication error (invalid user name or password) ==> 401
- -31 - Invalid data source defined in server ini-file. No database available ==> 404
- -32 - Request type not allowed (ALLOW option in server.ini) ==> 400

- -33 : Query mapping not defined ==> 409
- -40 - Invalid parameter passed in URL ==> 400
- -41 - Too many parameter passed in URL ==> 400
- -100 - Syntax error in OSI REQUEST body ==> 400
- -101 - Error parsing request source in ... ==> 400
- -102 - Body for OSI POST request does not contain code ==> 400
- -110 - Invalid OSI REQUEST body (property or member not found) ==> 400
- -111 - Error executing OSI request (field source) ==> 409
- -112 - Error selecting data in OSI REQUEST body (field source) ==> 409
- -113 - Could not locate instance for URL or OSI field source ==> 409
- -114 - Could not assign data to complex data type property ==> 409
- -115 - URL must refer to collection property ==> 409
- -116 - Updating existing instance is not allowed (invalid request or missing parameter) ==> 409
- -117 - Cannot create new instance for LOID access ==> 409
- -118 - Creating new instance is not allowed (invalid request or missing parameter) ==> 409
- -119 - POST request must not refer to an instance (locator not allowed) ==> 409
- -120 - POST must refer to collection property (complex data type) ==> 409
- -121 - Data passed in body is not a valid JSON string ==> 409
- -122 - Property name or path not valid in current context ==> 409
- -123 - Missing data (body) for PUT, PATCH or POST request
- -900 - exception thrown ==> 500

In case of exceptions an error message is returned in response message text. Error codes are transformed later to HTTP error codes (e.g. ==> 200). In case of success error codes, an additional message is (HTTP return codes 2xx) returned in response.

6.2 HTTP error codes

Internal error codes are converted into HTTP error codes. Following error codes are created:

- 200 - Ok
- 201 - Created (PUT or PATCH when a new instance has been created)
- 204 - No content (request is ok but no content returned – PUT, PATCH, DELETE)
- 400 - Bad request in passed body
- 401 - Unauthorized (user or password invalid)
- 403 - Forbidden (user accepted but not authorized for requested operation)
- 404 - not found (url is syntactically incorrect or not a valid database path)

- 405 - Method not allowed (request type not supported by application)
- 409 - operation refused (instance not located, duplicate key etc.)
- 500 - internal server error

7 Symbol reference

Syntax for input (`location`, `body`) and response (`values`) is described in terms of OSI BNF. The following topics contain the formal description of different language elements:

`location` - defines the request source (source property). `location` is passed to GET, PUT, PATCH, POST and DELETE.

`body` - contain complex data passed to PUT, PATCH, POST and OSI POST requests.

`values` - returned as response to all request types. The complexity of `values`, depends on request type. In case of request error(s), `values` is a `string_value` containing detailed error information.

7.1 location - HTTP URL

The HTTP URL (`location`) is part of the OHTTP request and defines the location for the required action. It follows the server and port specification in a HTTP request:

```
http://server:port/location[?parameter]
```

`parameter` - Several request types support parameters, which are well-defined keywords for controlling the behavior of the request. Supported parameters depend on request type. Following parameters are supported:

```
parameter := 'insert' | 'replace' | 'delete' |
             'noinsert' | 'noreplace'
```

`location` - The syntax for location is as follows:

```
location := property_reference(*)
property_reference := '/' name [ '/' [locator] ]
locator := '*' | key | skey | number
skey := '"' key '"'
key := string [ key_component(*) ]
key_component := '|' string
```

`name` - is a property name. The name in the first property name reference is, typically, an extent name (global collection) or `LOID` (collection of all instances in the database). In order to address namespace extents, the property name may be scoped (e.g. **`Person::Persons`**). The `LOID` property provides kind of short cut to instances that are, otherwise, accessible via hierarchical path, only. `name`s for further property references must be defined in the context of the preceding property's data type. The property in the last property reference is the **source property** for the required operation.

`locator` - is either a key value or a number (position or LOID). The locator defines a key or position for locating an instance. When `locator` is not a number, it is interpreted as key value for the main key of the collection (defined in the data model). When `locator` is a number, it is interpreted as position or LOID. In order to pass a number as key, it must be put in quotation marks. For single references, typically 0 (position) is passed as locator. For `LOID` an LOID value previously retrieved may be passed. A URL referring to a collection is called **collection URL**. The behavior of the response in case of collection URLs depends on the request type.

In order to indicate a collection iterator, '*' could be defined as locator. Usually, collection iterators are used to iterate through a collection property. In order to skip the locator, an empty locator may be passed (//). Skipping locators becomes necessary, when calling property functions (e.g. `Person::Persons/P1001/children//count`).

Elementary data type attributes and **`MEMO`** or **`BLOB`** references (**elementary property reference**) do not require a locator. For array attributes, one may pass an array index as locator in order to refer to a single element in the array. In order to refer to an attribute within a complex attribute (e.g. `city` in `address`), a position 0 has to be passed (`address/0/city`)

`skey` - a string key is a key within quotation marks. Usually, quotation marks may be omitted (as long as the key value is not recognized as number). When, however, the key value is a number, quotation marks are required in order to distinguish it from a position.

`key` - may be passed as locator, when the `name` refers to a collection property with a unique key. Components for a component key are separated by '`|`'.

A `number` (position or index) may be passed to any kind of property. In case of elementary properties (simple attributes, **`MEMO`** or **`BLOB`**), the `number` must be **0**. For collection properties or attribute arrays the `number` ranges from **0** to maximum dimension -1. For `LOID`, `number` is interpreted as value for the local object identifier (LOID) similar to a key value.

Syntactical details for string, number and name are describes below in topic **Values, formats and encodings** below.

Several characters must not appear in an URL. In general, URLs must not contain slash in names or key values. When it becomes necessary passing a key value containing '/', OSI requests may be used instead. Other forbidden characters may be pass as hexadecimal values preceded by '%'. Following characters must be passed hexadecimal:

- # -> %23
- ? -> %3F
- \ -> %5C

URL data is expected to be UTF8 encoded.

7.2 body - request data for HTTP requests

The `body` passed to a PUT, PATCH, POST request contains a list of property names and values to be updated. The `body` is always passed as `values` (see also **Values, formats and encoding** below).

```
body := values
```

For elementary source properties, the `body` just contains a `value`. Usually, when the source property refers to a complex data type instance, the `body` is passed as `object`. In case of array attributes or collection properties, the `body` is passed as `array` (see example below).

Empty values in the body will not cause any update. Thus, one may pass empty values for an array in order to update selected elements (e.g. `first_name: [ , , "Paul" ]`, which updates the value for `first_name[2]`, only).

`null` - For POST and PATCH request the `null` value can be used for resetting the property to its initial state.

```
Person:  pid, name, first_name[3], address[2]
Address: city, street, number
--> body for PATCH request:
{ pid: "ID123", name: "Miller", first_name: [ null, "Paul" ]
  address: [ { city: "Berlin", street: "Grüne Allee", nummer: 51 }, null,
null ] }
```

7.3 body - request data for OSI POST

The `body` passed to an OSI request is a kind of expression that supports filtering, update operations and field mapping. The `body` for an OSI PUT request has the following syntax:

```
body := source [ fieldlist ]
source := operand | expr_code
fieldlist := fassign '{' fielddef [fielddef_ext(*)] '}'
fassign := '==>'
fielddef_ext := ',' fielddef
fielddef := name [assignment]
assignment := ':' body
```

`name` - is the name of a `named_value` in the JSON result string (see topic **Values, formats and encoding** below)

`operand` - is a syntax element defined in OSI BNF. Semantic details are described in ODABA Script Interface - OSI. An `operand` is either an access path consisting of property and function names separated by '.' or an expression (arithmetical, Boolean etc).

`expr_code` - is an inline OSI function containing any number of statements, variable declarations etc.

```
// operand examples
Persons   // extent name
Persons() // collection property
birth_day // property name
Persons(0).children(0).name // property path
Persons.count() // access path (consists of properties and operations)
Date.now - birth_day // expression

// expr_code example (with Person as calling object)
{
VARIABLES
  int(10,2)    income;
PROCESS
  children.top();
  while ( children.next() )
    income += children.income;
FINAL
  return(income);
}
```

7.4 Request result (response)

Each request returns a JSON string with the syntax of `values` as response (syntax of `values` is described below in **Values, formats and encoding**). Most requests (except GET and OSI) simply return a `string_value`. In case of error, always a `string_value` is returned.

When terminating successfully, GET and OSI respond with an JSON `object` (see **Values, formats and encoding**) like:

```
{ name: values }
```

`name` - is the name of the source property (`source`) for the request defined in `body` (OSI) or `location` (other HTTP requests) .

`values` - Depending on the referenced property and type of request, the structure of values differs. Restrictions for values are described for each request type separately.

```
// get number of employees for first company
http://127.0.0.1:8888/Company/0/employees/count ==> JSON result:
{ count: "485" }

// Get person's children
http://127.0.0.1:1234/Persons/ID123/children ==> JSON result:
{ children: [
    { pid: "ID123" name: "Miller", first_name_0: "Paul",
      first_name: [ "Henry", "" ], birth_date: "1966-11-11", sex: "male",
      married: false, income: 2500.00, age: 52, children_income: 3807.22,
      notes: "this might be a very long text in C-string format\n",
      address: [ { url: LOID/12346127 } ],
      children: [
        { url: LOID/12346000 },
        { url: LOID/12346002 } ],
      parents: [ { url: LOID/12346001 } ],
      employee: [ ]
    },
    { ... next child instance .... }, ...
] }
```

7.5 Body for QUERY requested

The body for a QUERY request is not passed with the URL but defined in an appropriate query definition on server side (resource database). The URL for query requests has to be send as:

```
http://server:port/query/type/class/method[?parameters]
```

`server` and `port` specification is followed by the fixed value **`query`**.

`type`: defines the sub type for the query (GET, PUT, PATCH, OSI, DELETE)

`class`: The class name is the data type, the query has been defined for.

`method`: is the method name defining the query within the class.

The query definition may contain any numbers of parameters preceded by '**?**'.

```
parameters := parm [ parm_ext(*) ]
parm := name '=' value
parm_ext := '&' parm
```

`name`: is a variable name referenced in the query with preceding '?'.

`value`: is a string replacing the appropriate variable when executing the query. When value contains special characters, it should be quotes.

7.6 Values, formats and encoding

`values` - Values may be returned as result of a GET request or passed as data via a POST, PUT or PATCH request. Values represents either a single value, an object or a value array:.

```
values := value | object | array
object := '{' named_value [ named_value_ext(*) ] '}'
named_value_ext := ',' named_value
named_value := name ':' value
array := '[' value [ value_ext(*) ] ']'
value_ext := ',' value
value := string_value | number |
         'true' | 'false' | 'null'
string_value := '"' string '"'
```

`name` - A name consists of ASCII characters (lower and upper case letters, digits, underscore). Typically, names appear as property names or field names in result definitions or body data.

`number` - A number is an integer, decimal or a float point number. It depends on context (property definition), which number types are allowed.

`null` - may be passed or returned for **MEMO** and **BLOB** properties and indicates no-data, which is slightly different from an empty string (""). `null` is also passed in order to indicate, that a property should be reset.

`true, false` - are the only values allowed for Boolean properties.

`string` - String values must not contain special characters as newline or tab. Special characters have to be escaped as follows (see also JSON syntax):

- \ ==> \\
- line feed (10) ==> \n
- carriage return (13) ==> \r
- backspace (8) ==> \b
- form feed (12) ==> \f
- tab (9) == \t
- " ==> \"

All string values exchanged via OHTTP are UTF8 encoded. Binary data (**BLOB** properties) is always passed as BASE64 string (which is UTF8 compatible),

```
// POST, PUT, PATCH data block
{ name: "Müller", age: 45, comment: "Höhen und \nTiefen"
  image: "......" }

// JSON result instance
{ name: "Müller", age: 45, comment: "\Höhen und \nTiefen",
  image: "......" }
```

8 Testing HTTP requests

ODABA provides a little test framework for testing OHTTP requests. Tests may be prepared in a test script containing any number of tests. The test script consists of any number of test blocks, where each test block has the following structure:

- line 1: test header (fix text 'TEST:' followed by test title)
- line 2: request type (PUT, PATCH, POST, DELETE, QUERY)
- line 3: user name
- line 4: password
- line 5: URL without server:port (e.g. /Persons::Person/0/children)
- line 6 and following (until end of file or next empty line or next test header): body

Empty lines or comment lines (beginning with //) between test blocks and block lines will be ignored.

In order to run the test script, a command line server or server daemon has to be started and an OSI scrip as shown below has to be called:

```
OShell.exe OShell.ini TestHTTP.osh
```

Since the function `OHTTPExecute(testscript,server,port)` for running the test script is implemented in **ode.dev**, the database has to be defined as `RESOURCES` database in the data source (here `Sample`).

While running the script, Test case head line, HTTP return code and result string are written to console.

```
// test block
TEST: Create instance with pid P1001
PUT
user
upwd
/Person::Persons/P1001
{ name:"Hops0", first_name: [ "Sala0", "Tralala0"], married: true,
location: {zip: 1000, city: City1000, street: "Short Lane", number:
1000}, notes: "temporary notes", birth_date: "2000-10-31" }

// TestHTTP.osh (run with OShell)
cd Sample                          // open data source (see OShell.ini)
osi 'dictionary.loadOSILibraries' // load login function when not defined
in .dev

osi begin
  Option("HTTP.timeout").assign("300"); // increase timeout
  OHTTPExecute("Test.txt","localhost","8888"); // run test script
end

// OShell.ini for running OShell
[SYSTEM]
DICTIONARY=ode.sys

[OShell]
DSC_Language=English

[OSI]
Library=./*.osi

[Sample]
DICTIONARY=Sample.dev
RESOURCES=ode.dev
DATABASE=Sample.dat
PLATFORM_INDEPENDENT=YES
SHARE=YES
ACCESS_MODE=Write
ONLINE_VERSION=YES
```

8.1 Test with internally called server

In order to support automated tests, one may also start an internal server. The procedure is the same as described above, except, that the server need not to be started and the **TestHTTP.osh** script looks a bit different (see example below).

Moreover, the **OShell.ini** file needs slight modifications. The Allow and Authentication option from the server.ini file have to be moved to **OShell.ini** (see below).

A Server.ini file with an `OHTTPServer` section has to be provided at location defined in `OHTTP.ServerIni`. The Port number defined in `OHTTP.Port` has to be passed also to `OHTTPExecute()` function.

```
// TestHTTP.osh for internal server start (run with OShell)
set OHTTP.Port=8888
set OHTTP.ServerIni=Server.ini

cd Sample
osi 'dictionary.loadOSILibraries'

osi begin
  HTTPGlobal::Start();
  Option("HTTP.timeout").assign("300");
  OHTTPExecute("Test.txt",%OHTTP.Port%);
  HTTPGlobal::Stop();
end
q

// Modified section in OShell.ini
...
[OShell]
DSC_Language=English
ALLOW=all
Authentication=basic;Person::Login
...
```

9 Database model for ODABA sample database

The model below defines the sample database model in ODL language. ODABA ODL is an extension of the ODMG ODL standard. With the server, also a test library is delivered, which contains a number of test requirements (and examples).

```
UPDATE SCHEMA Sample {

// Car class definition
  CLASS Car PERSISTENT (  KEY IDENT_KEY pk(cid); ) {
    ATTRIBUTE {
            CHAR(10)   cid;
            STRING(40) type;
            INT(2)     number_of_seats = 4;
    };

    RELATIONSHIP {
            Company       SECONDARY  company  INVERSE cars;
            SET<Employee> SECONDARY  users    ORDERED_BY (pk UNIQUE)
INVERSE used_cars;
    };
  };

// Person class definition
  CLASS Person PERSISTENT
  ( KEY { IDENT_KEY pk (pid); sk (name); };
    EXTENT MULTIPLE_KEY owner Persons ORDERED_BY (pk UNIQUE NOT_EMPTY,
sk); )
  {
    ENUM Sex {
                male = 1,
                female = 2,
                undefined = 0
    };

    STRUCT Address PERSISTENT {
                STRING(6)  zip;
                STRING(40) city;
                STRING(80) street;
                STRING(6)  number;
    };
    ATTRIBUTE {
      NOT_EMPTY CHAR(16)   pid;
                STRING(40) name;
                STRING(40) first_name[3];
                DATE       birth_date;
                Sex        sex = male;
                bool       married = false;
                INT(10,2)  income;
      TRANSIENT INT(3)     age SOURCE( (Date().now() - birth_date)/365.25
);
      TRANSIENT INT(10,2)  children_inc SOURCE ( children.sum(income) );
    };
REFERENCE   Address  location;
    REFERENCE   STRING   notes[4000];

    RELATIONSHIP {
                SET<Person>         children      BASED_ON Persons
                                                  INVERSE parents;
                Person   SECONDARY  parents[2]    BASED_ON Persons
                                                  INVERSE children;
                Employee SECONDARY  employee      INVERSE person;
    };
  };

// Employee class definition
  CLASS Employee PERSISTENT : Person person BASED_ON Person::Persons
INVERSE employee
  ( KEY  IDENT_KEY pk(pid); )
  {
    RELATIONSHIP {
                Company SECONDARY   company       BASED_ON Company
                                                  ORDERED_BY (pk UNIQUE)
                                                  INVERSE employees;
                Car     NO_CREATE   used_cars[2]  BASED_ON .company.cars
                                                  ORDERED_BY (pk UNIQUE)
                                                  INVERSE users;
    };
  };

// Company class definition
  CLASS Company PERSISTENT ( KEY IDENT_KEY pk(name); ) {
    ATTRIBUTE   NOT_EMPTY STRING(200)  name;

    RELATIONSHIP {
                SET<Employee>       employees BASED_ON Employee
                                              ORDERED_BY (pk UNIQUE)
                                              INVERSE company;
                SET<Car>      OWNER cars      ORDERED_BY (pk UNIQUE)
                                              INVERSE company;
    };
  };
```

```
// View definition
  VIEW ChildrenIncome
    FROM( SET<Person> persons = Person::Persons  )
    WHERE( children.count > 0 )
    GROUP BY( STRING incGroup = (income <= 1000 ? 'low'    :
                                 income <= 3000 ? 'medium' :
                                 income <= 7000 ? 'high'   :
'very_high' ),
              Sex sex = sex)
    ( KEY IDENT_KEY pk( incGroup, sex ); )
  {
    ATTRIBUTE {
                STRING(10) incGroup ;
                Sex        sex ;
                INT(10,2)  ci_sum = sum(children_inc);
                INT(10,2)  ci_avr = average(children_inc);
                INT(10,2)  ci_dev = deviation(children_inc);
                INT(10,2)  ci_min = minimum(children_inc);
                INT(10,2)  ci_max = maximum(children_inc);
      TRANSIENT INT(10,2)  diff   = ci_max-ci_min;
    };
  };

  // Global extent definition
  EXTENT Company UPDATE MULTIPLE_KEY OWNER Company
               ORDERED_BY ( pk UNIQUE NOT_EMPTY );
  EXTENT Employee UPDATE MULTIPLE_KEY OWNER Employee
               ORDERED_BY ( pk UNIQUE NOT_EMPTY );
  EXTENT ChildrenIncome TRANSIENT OWNER ChildrenIncome
               ORDERED_BY ( pk UNIQUE NOT_EMPTY );

};
```