

# REST Server Quickstart

---

*This page explains how to initialize the REST Server, register endpoints, and expose your own functions over HTTP.*

## Usage

To use the REST-Server module:

1. Load the the `rest` module.
2. Initialize the framework, using the `.com_kx_rest.init` function.
3. Register the endpoints, using the `.com_kx_rest.register` function.
4. (Optional) On a request to `.z.ph` or `.z.pp`, forward execution to `.com_kx_rest.process` to process the incoming request and return the result.

Note: "Note" The examples below assume a `.rest` namespace, which aliases `.com_kx_rest`.

```
```bash
.com_kx_rest:use`kx.rest;
```
```

## Initialize the REST server framework

Before registering endpoints, initialize the REST Server:

```
/ Alias namespace for convenience, typically once at beginning of file
.rest:.com_kx_rest

/ Initialize the framework with autoBind enabled
.rest.init enlist[`autoBind]!enlist[1b] / Initialize
```

Use the `.com_kx_rest.init` function to initialize the framework. Call it without arguments (for example `.com_kx_rest.init[]`) for minimal setup, or as a unary (with a dictionary argument) to perform further steps such as auto-binding to `.z.ph` and `.z.pp` (as in the above example).

With the `autoBind` option specified as `1b`, if no endpoint matches a request, the server delegates to the next handler (if applicable). Conversely if `autoBind` is not specified (or specified as `0b`), unmatched requests return an HTTP 404 code.

Note that when auto-binding is disabled, the application binds `.com_kx_rest.process` to `.z.ph` and `.z.pp`, and performs any chaining as needed.

## Register the endpoints

Endpoints map HTTP methods and URL paths to q functions.

Here is an example with two endpoints:

```
// Register endpoints
.rest.register[`get;
  "/customers";
  "Returns all customers";
  .db.getAllCustomers;
  .rest.reg.data[`i;-6h;0b;0;"Offset to first row"],
  .rest.reg.data[`cnt;-6h;0b;10;"Number of rows to return"] ]

.rest.register[`get;
  "/customers/{id}";
  "Returns one or more customers by their IDs";
  .db.getCustomersById;
  .rest.reg.data[`id;6h;1b;0;"One or more customer IDs"] ]
```

After we initialized the framework with `autoBind` set to `1b`, we created two endpoints, as follows:

`get /customers`

This endpoint returns all customers. It expects two parameters:

- ``i (integer)`` is the offset of first row to return
- ``cnt (integer)`` is the number of rows to return

Both parameters are optional, and have default values of `0` and `10` respectively. The endpoint is mapped to the `.db.getAllCustomers` function, which might be defined as follows:

```
```q
.db.getAllCustomers:{
  x[`arg;`cnt]#select from customers where i>=x[`arg;`i] }
```
```

To query the endpoint, run the following command:

```
```bash
curl 'localhost:8080/customers'
curl 'localhost:8080/customers?i=10&cnt=10'
```
```

`get /customers/{id}`

This endpoint returns one or more customers based on their IDs. It expects one parameter: `id (integer[])`, which is an input parameter in the form of a **path variable**. The value of this parameter is

determined at matching time. The endpoint is mapped to the `.db.getCustomersById` function, which might be defined as follows:

```
```q
.db.getCustomersById:{
  select from customers where id in x[`arg;`id] // should be in ID order
}
```
```

## How to register endpoints

Use the `.com_kx_rest.register` function to register an endpoint. The registration contains the following components:

- The operation and path of the endpoint
- A description summarizing the purpose of the endpoint
- The handler function
- Optional definition of user input, which includes: path variables, query string, headers, and request body (for `post/put` based endpoints)
- Optional definition of result object

The operation is arbitrary and refers to the action being performed, typically one of the standard HTTP methods defined by the REST architecture:

- `GET`
- `POST`
- `PUT`
- `DELETE`

The path is the part of the URL that follows the host and port (for example, `/customers`). It identifies the resource targeted of the operation. Paths can be customized and may include variables that act as parameters to the endpoint. For example, `/db/{table}/schema` is a valid path that has a variable named `table`.

The handler function is responsible for performing the activity of the endpoint. It works as follows:

User input definitions specify the properties of the expected input parameters which include:

- path-variables and query-string (using `.com_kx_rest.reg.data`)
- headers (using `.com_kx_rest.reg.header`)
- JSON-based request body (using `.com_kx_rest.reg.body`).

For example, the request `/db/{x}/{y}/z?i=0&cnt=10` has the following input parameters: `x`, `y`, `i`, and `cnt`. You can specify the name, data type, necessity, default value, and description of each parameter.

The framework uses this information to:

- Ensure the required parameters are supplied in the request, failing the request if any is missing
- Parse the value using the expected data type

The handler also has access to the raw user input, which is useful when some or all of the input values are unknown at the time of registration.

Output definition specifies the schema of the result (using `.com_kx_rest.reg.output`). The framework doesn't currently use the definition, but it is good practice to specify it for future purposes.

## Request processing

An HTTP request contains the following components:

- HTTP method
- Path
- Query string
- HTTP headers
- Request body (applies to `POST`, `PUT`, and `DELETE`)

The HTTP method generally specifies the REST operation; in KDB-X, it is limited to `GET` and `POST`. Thus, the framework looks for the operation in the `http-method` HTTP header, and if not present defaults to `GET` (for `.z.ph`), or `POST` (for `.z.pp`). It is thus important to use a front-end API gateway (like the AWS HTTP API gateway) that can populate `http-method` HTTP header, and convert `PUT`, and `DELETE` HTTP methods to `POST`, leaving `GET` requests as they are. Refer to the [request handling](#) for details.

When a request is received, both `operation` and `path` are combined to find a matching endpoint, favoring exact matches over ones containing variables (for example, `/a/b/c` vs `/a/{x}/c`).

User input is then processed, distinguishing between the following categories:

- User input specified using the `.com_kx_rest.reg.data` function. These contain path variables (like `id` in `/customers/{id}`) and query-string parameters. Values are parsed according to their datatype. If a parameter is missing from the request, then its default value is used. But if the missing parameter is marked required, then the request fails with 400 HTTP status code (with the names of the missing required parameters included in the response). Values of input parameters are passed to the handler function as dictionary under `arg` key. Note that raw input parameters (as they appear in the request) are also passed to the handler function under the `rawArg` key.
- Request body (where applicable) is expected to be in JSON format. It is deserialized into a KDB-X data structure using `.j.k`, and passed to the handler function under `data` key.
- Request body (in case of `post` or `put` operation) is expected to be in JSON format. If the endpoint has body input specified using the `.com_kx_rest.reg.body` function, then the elements of the defined object are parsed from the input (including nested objects) according to their datatypes and necessity settings. Similar to input parameters, the raw body is passed to the handler under `rawData` key.
- HTTP headers are passed untouched to the handler function under `hdr` key.

After input is collected, the handler function is invoked. If the function is declared to take arguments with the same names as the endpoint parameters (or `body` if endpoint is comprised only of a `body` parameter), then the function is variadically invoked with its arguments mapped from the request input. Otherwise, the function is invoked as a unary with a dictionary containing the following keys:

- REST operation of the endpoint

- The path of the endpoint
- Values of processed input parameters
- Raw input parameters present in the request
- Value of processed body object, which is typically a dictionary if the body is of an object type, but can be of any type as specified by `reg.body` function.
- Raw KDB-X form of the request body (if present)
- HTTP headers

The handler function is expected to return its response in one of the following forms:

- A KDB-X data structure (typically a dictionary or a table), which is serialized to JSON by the framework before being returned to the client
- Result of the `.com_kx_rest.util.response` function, which gives the handler control over the HTTP status code, and content type of the response (available post release 1.0.0)
- Result of the `.com_kx_rest.util.httpResponse` function, which gives the handler total control over the response
- If there is a problem with input, the handler must call `.util.throw` to signal an error

## REST Server Examples

---

*This page provides some examples on how to use the REST server in KDB-X.*

### Customers

This section demonstrates a sample REST server with various styles of REST endpoints, including static, dynamic, and API-like patterns. See how to register endpoints and handle different types of RESTful requests in KDB-X.

#### Static

resource names are predefined at endpoint registration time, for example `{/customers}`

#### Dynamic

the resource name (or part of it) is supplied at call-time, for example `/db/{tbl}/{col}`

#### API-like

same as static but uses verbs instead of names, for example `/getCustomers`

```
```q

.com_kx_rest:use`kx.rest;
.rest:.com_kx_rest; / Alias namespace for convenience

//
// @desc App initialization.
```

```
//
init:{
    .rest.init[enlist[`autoBind]!enlist[1b]]; / Initialize

    initStatic[];          / Sample static endpoints
    initDynamic[];         / Sample dynamic endpoints
    initApi[];             / Sample API-like endpoints

    .rest.register[`get;"/help"; / Returns information about registered endpoints
        "Retrieves information about registered REST endpoints";
        {.rest.t};
        ()!();

    .rest.register[`get;"/hc"; / health-check
        "health-check endpoint";
        {"ok"};
        ()!();

    .rest.register[`get;"/_ping";
        "for testing";
        {"pong"};
        ()!(); }

//
// Sample parameter subset to support paging.
//
pagingParams:.rest.reg.data[`i;-6h;0b;0;"Offset of first row"],
    .rest.reg.data[`cnt;-6h;0b;10;"Number of rows to return"];

//
// Wraps `#` to limit to data size.
//
take:{[n;d]min[(n;count d)]#d}

//
// Initialization examples.
//

//
// Static endpoints, and various styles of versioning.
//
initStatic:{
    .rest.register[`get;"/customers";
        "Returns all customers";
        {take[x[`arg;`cnt]]select from customers where i>=x[`arg;`i]};
        pagingParams
    ];

    .rest.register[`get;"/customers.2";
        "Returns all customers (version 2)";
        {take[x[`arg;`cnt]]update newCol:1 from select from customers where
```

```

i>=x[`arg;`i]];
    pagingParams
];

.rest.register[`get;"/v3/customers";
    "Returns all customers (version 3)";
    {take[x[`arg;`cnt]]update newCol:1, newCol2:10 from select from customers
where i>=x[`arg;`i]];
    pagingParams
];

.rest.register[`get;"/customers/{id}";
    "Returns one or more customers by their IDs";
    {select from customers where id in x[`arg;`id]];
    .rest.reg.data[ `id;6h;1b;0Ni;"One or more customer IDs"
]; }

initDynamic:{
    .rest.register[`get;"/db";
        "Retrieves list of table names";
        {tables[]};
        ()!()
    ];

    .rest.register[`get;"/db/{table}";
        "Retrieves a table";
        .tbl.getData;
        .rest.reg.data[ `table;-11h;1b;`; "Table name"],
        pagingParams
    ];

    .rest.register[`get;"/db/{table}/meta";
        "Retrieves metadata of a table";
        {0!meta x[`arg;`table]];
        .rest.reg.data[ `table;-11h;1b;`; "Table name"
    ];

    .rest.register[`get;"/db/{table}/{col}";
        "Retrieves a column subset of a table";
        .tbl.getData;
        .rest.reg.data[ `table;-11h;1b;`; "Table name"],
        .rest.reg.data[ `col;11h;1b;0#`; "Result columns"],
        pagingParams
    ]; }

initApi:{
    .rest.register[`get;"/getCustomers";
        "Returns all customers";
        {[[] id:til 10; nm:10?`5)}];
        ::

```

```

]; }

//
// Automatic data retrieval
//

//
// @desc Generic data retrieval handler. Look for {tbl}, {id}, and {rel} arguments
//
.tbl.getData:{
  tn:x[`arg;`table];
  i_:$[`i in key x[`arg];x[`arg;`i];0];
  cnt:$[`cnt in key x[`arg];x[`arg;`cnt];0W];

  if[not tn in tables[]; `table]; / Check whether table exists
  w:$["~0N!flt:x[`data];();enlist parse flt]; / If request is a POST, then data
is expected to be the where clause
  c:$[`col in key x`arg;{x!x}x[`arg;`col];()); / Columns to retrieve
  take[cnt]select from ?[tn;w;0b;c] where i>=i_ }

init[]

//
// Test data
//
n:100
customers:([] id:1+til n; name:n?`7)
sn:`aapl`goog`msft`nke`ftse
symbols:1!([ sym:`aapl`goog`msft`nke`ftse; src:count[sn]?`8);
trades:([] ts:$[12h;.z.d]+$[`minute;1]*til n;
  sym:n?(0!symbols)[`sym];
  price:n?10.0;
  size:n?1000)
...

```

## queryclient

This section provides the full definition of an example client that interacts with the REST server. It relies on the [curl](#) module. It shows how to connect, create resources, submit queries, and check job status using q code and HTTP requests.

```

```q
.kurl:use`kx.kurl
if[not `server in key .Q.opt .z.x; "Provide -server http://host:port"]
server:first @[;`server] .Q.opt .z.x

// Wait forever until health check returns true.

```

```

while[200 <> first @[.kurl.sync;(server,"/v1/hc";`GET;::);{(-1;"")}];
system "sleep 1"]

// Create a new project folder
body: .j.j `name`dir!("myProject";"projFolder1")
headers:("http-method";"Content-Type")!("POST";"application/json")
resp:.kurl.sync (server,"/v1/projects";`POST;`body`headers!(body;headers))
if[200 <> first resp; 'last resp]
project:.j.k last resp
-1 "Created ", project `name;
projectID:string project `id

// Create an empty database folder
body: .j.j enlist[`name]!enlist "db1"

resp:.kurl.sync (server,"/v1/projects/",projectID,"/databases";
`POST;
`body`headers!(body;headers))

if[200 <> first resp; 'last resp]
database:.j.k last resp
databaseID:database `id

// Create a random partitioned table using JSON
// Note that this is just for demonstrations sake
// instead of calling this API, you may mount a db externally on the server
n:1000

t:([
  date:n?2021.01.01 2021.01.02 2021.01.03;
  x: n?100f;
  y: n?0b )

resp:.kurl.sync
(0N!server,"/v1/projects/",projectID,"/databases/",databaseID,"/tables";
`POST;
`body`headers!(.j.j `name`table!(`t;t);headers))

if[200 <> first resp; 'last resp]

// Submit a query job
body:.j.j`query`databaseID!("select from t"; databaseID)
resp:.kurl.sync (server,"/v1/projects/",projectID,"/jobs";
`POST;
`body`headers!(body;headers))
if[200 <> first resp; 'last resp]
job:.j.k last resp
jobID:string job `id

// Check on the job
resp:.kurl.sync (server,"/v1/projects/",projectID,"/jobs/",jobID;`GET;::)

```

```
if[200 <> first resp; 'last resp]
````
```

queryserver

This section demonstrates a REST server capable of running asynchronous query jobs. It explains the server's resource organization, usage patterns, and how to register endpoints for managing projects, databases, tables, and jobs.

Jobs are run in worker processes that execute arbitrary qSQL and trigger a callback when done. A REST Client is expected to poll for async job results.

Resources for the server are organized as a list of projects containing lists of databases.

Databases are organized as a list of tables.

Usage patterns:

|                                                                       |            |
|-----------------------------------------------------------------------|------------|
| GET /v1/hc                                                            | Simply     |
| health check                                                          |            |
| GET /v1/projects                                                      | List all   |
| projects                                                              |            |
| GET /v1/projects/{projectID}                                          | List one   |
| project's attributes                                                  |            |
| GET /v1/projects/{projectID}/databases                                | List all   |
| databases for projectID                                               |            |
| GET /v1/projects/{projectID}/databases/{databaseID}                   | List one   |
| database's attributes                                                 |            |
| GET /v1/projects/{projectID}/databases/{databaseID}/tables            | List all   |
| tables for projectID + databaseID                                     |            |
| GET /v1/projects/{projectID}/databases/{databaseID}/tables/{tableID}  | List table |
| attributes                                                            |            |
| POST /v1/projects                                                     | Create a   |
| new project                                                           |            |
| POST /v1/projects/{projectID}                                         | Update a   |
| project (rename/delete)                                               |            |
| POST /v1/projects/{projectID}/databases                               | Create a   |
| database                                                              |            |
| POST /v1/projects/{projectID}/databases/{databaseID}                  | Update a   |
| database                                                              |            |
| POST /v1/projects/{projectID}/databases/{databaseID}/tables           | Create a   |
| table                                                                 |            |
| POST /v1/projects/{projectID}/databases/{databaseID}/tables/{tableID} | Update a   |
| table                                                                 |            |
| GET /v1/projects/{projectID}/jobs/                                    | List all   |
| running queries                                                       |            |
| POST /v1/projects/{projectID}/jobs/                                   | Run a new  |
| query                                                                 |            |
| GET /v1/projects/{projectID}/jobs/{jobID}                             | Check on   |
| the status of a job                                                   |            |

GET /v1/projects/{projectID}/jobs/{jobID}/results  
JSONified query results

Get the

```

```q
.com_kx_rest:use`kx.rest; // load the module
.rest:.com_kx_rest / Make an alias for convenience
.rest.init[enlist[`autoBind]!enlist[1b]];

// Querystring params
\d .demo
param.projectID:.rest.reg.data[`projectID;-7h;1b;0;"Project ID"]
param.databaseID:.rest.reg.data[`databaseID;10h;1b;"";"Database ID"]
param.tableID:.rest.reg.data[`tableID;10h;1b;"";"Table ID"]
param.jobID:.rest.reg.data[`jobID;-7h;1b;0;"Job ID"]

// Body params
.rest.reg.object[`project;
.rest.reg.data[`name;-11h;0b;`;"Project name"],
.rest.reg.data[`dir;10h;0b;"";"Project directory"]]
.rest.reg.object[`database;
.rest.reg.data[`name;-11h;0b;`;"Database name"]]
.rest.reg.object[`table;
.rest.reg.data[`name;-11h;0b;`;"Table name"],
.rest.reg.data[`table;98h;0b;([] x:());"Table content"]]
.rest.reg.object[`job;
param.databaseID,
.rest.reg.data[`query;10h;0b;"";"q query"]]

param.project:.rest.reg.body[`project;0b;;; "Project information"]
param.database:.rest.reg.body[`database;0b;;; "Database information"]
param.table:.rest.reg.body[`table;0b;;; "Table information"]
param.job:.rest.reg.body[`job;0b;;; "Job information"]
.rest.register[`get;"/v1/hc";"simple healthcheck";{"ok"};()!()];

// GETTER API
.rest.register[`get;
"/v1/projects";
"List all projects";
{.demo.projects};
::]

.rest.register[`get;
"/v1/projects/{projectID}";
"List one projects attributes";
{first select from .demo.projects where id = x[`arg;`projectID]};
param.projectID]

```

```
.rest.register[`get;
"/v1/projects/{projectID}/databases";
"List all databases for a project";
{.demo.findDB};
param.projectID]

.rest.register[`get;
"/v1/projects/{projectID}/databases/{databaseID}";
"List a databases attributes";
{
  db:.demo.findDB x;
  enlist[`name]!enlist db `h
};
param.projectID,param.databaseID]

.rest.register[`get;
"/v1/projects/{projectID}/databases/{databaseID}/tables";
"List all tables in a database";
{
  db:.demo.findDB x;
  key db `h
};
param.projectID,param.databaseID]

.rest.register[`get;
"/v1/projects/{projectID}/databases/{databaseID}/tables/{tableID}";
"List table attributes";
{'`notImplemented};
param.projectID,param.databaseID,param.tableID]

// CREATE APIs
.rest.register[`post;
"/v1/projects";
"Create a new project";
{
  system "mkdir -p ",di:projRoot,x[`data;`dir];
  .demo.projects,:
  `name`id`dir`created!(x[`data;`name];count .demo.projects;di; .z.p);
  :last .demo.projects
};
param.project]

.rest.register[`post;
"/v1/projects/{projectID}";
"Update a project";
{'`notImplemented};
param.projectID]

.rest.register[`post;
"/v1/projects/{projectID}/databases";
"Add a database to a project";
```

```

{
    proj:.demo.findProject x;
    dbh:hsym `$db:proj[`dir],"/",string n: x[`data;`name];
    if[() ~ key dbh; system "mkdir -p ",db];
    :enlist[`id]!enlist n
};
param.projectID,param.database]

.rest.register[`post;
"/v1/projects/{projectID}/databases/{databaseID}";
"Rename or delete a database";
{"not implemented"};
param.projectID,param.databaseID]

.rest.register[`post;
"/v1/projects/{projectID}/databases/{databaseID}/tables";
"Add a table to a database";
{ .demo.writePar x };
param.projectID,param.databaseID,param.table]

.rest.register[`post;
"/v1/projects/{projectID}/databases/{databaseID}/tables/{tableID}";
"Add or overwrite dates in a table";
{`notImplemented};
param.projectID,param.databaseID,param.tableID]

// JOB API
.rest.register[`get;"/v1/projects/{projectID}/jobs";
"List all jobs for a project";
{select from .demo.jobs where projectID = x[`arg; `projectID];};
param.projectID]

.rest.register[`post;
"/v1/projects/{projectID}/jobs";
"Submit a query job";
{
    proj:findProject x;
    avail:first .demo.workers except exec worker from .demo.jobs;
    db:proj[`dir],"/",x[`data;`databaseID];
    neg[avail] (.demo.runQuery; db; x[`data;`query]);
    .demo.jobs,:`id`worker`projectID`status!(count .demo.jobs;
    avail;proj `id;`active);
    :last .demo.jobs
};
param.projectID,param.job]

.rest.register[`get;
"/v1/projects/{projectID}/jobs/{jobID}";
"List details of a job";
{findJob x};
param.projectID,param.jobID]

```

```

.rest.register[`get;
"/v1/projects/{projectID}/jobs/{jobID}/results";
"Get results for a finished job";
{
    job:select from findJob[x] where status=`done;
    if[1 <> count job;"Job not finished"];
    job[`worker] ".demo.results"
};
param.projectID,param.jobID]

projRoot:system["cd "],"/exampleProjects/"
projects:([ name:(); id:"j"$(); dir:(); created:"p"$())

findProject:{
    proj:select from .demo.projects where id = x[`arg;`projectID];
    if[0 = count proj; '"No such project "', x[`arg, `projectID]];
    :first proj; }

findDB:{
    proj:findProject x;
    dbh:hsym `$db:proj[`dir],"/",n: x[`arg;`databaseID];
    if[( ) ~ key dbh;"Database does not exist: ", n];
    :`name`h!(n;dbh) }

findJob:{
    select from .demo.jobs where projectID = x[`arg;`projectID],
        id = x[`arg;`jobID] }

writePar:{
    db:findDB x;
    name:x[`data;`name];
    t:x[`data;`table];
    t[`date]: "D"$t `date;
    a:cols[t] except `date;
    kt:?[t;();`date;a!a];
    (`$string key[kt]) {[root;name;date;d]
        (` sv root,date,name,`) set flip d
        }[db `h;name]' value kt;
    :name }

done:{ .demo.jobs:update status:`done from .demo.jobs where worker = .z.w;}
maxWait:00:00:05
i:0
n:10
workers:()
jobs:([ id:"j"$(); worker:"I"$(); projectID:"j"$(); status:`$())
\d .

.z.po:{.demo.i+:1;}
.z.ts:{[start;now]

```

```

if [now > start + .demo.maxWait;
    -2 "Took longer than ",string[.demo.maxWait],
    " to start ",string[.demo.n]," workers";
    -2 "Exiting...";
    exit 1];
// Clear timer and uninstall .z.po
if[.demo.n = count .z.W;
    system "t 0";
    .z.po:{};
    .demo.workers:key .z.W]
}{.z.p;}

do[10; system "q queryworker.q -server ",string system "p"]
\t 1000
```

```

## queryworker

This section provides the full definition of the example REST server's worker processes. These workers execute query jobs submitted to the server and return results asynchronously.

This process should be started automatically by the REST server example

Copy this into your present working directory when running the REST server.

```

\d .demo
h:-1i
lastresult:()
.z.pc:{if[x ~ h; exit 0];}

if[not `server in key args:.Q.opt .z.x;
    -2 "Server port required with -server <port>";
    exit 1]

h:hopen `"$":localhost:", first args `server

runQuery:{[dbpath;query]
    system "l ",dbpath;
    lastResult::value query;
    .z.w (`.demo.done; 1b); }

\d .

```

## OpenAPI example

This section contains an OpenAPI 3.0 specification for the REST server shown in the example on how to [expose a RESTful interface](#). You can use this specification to import the API into tools that support OpenAPI, making it easier to explore and test the endpoints.

To use this in examples, copy the content below to a local JSON file, and import it into the relevant portal.

Replace the `servers.url` example with your server URL.

You can probably edit this after the fact in the portal's UI

```
```json
{
  "openapi": "3.0.1",
  "info": {
    "title": "kdbx-rest-test-api",
    "description": "",
    "version": "1.0"
  },
  "servers": [
    {
      "url": "https://example.azure-api.net"
    }
  ],
  "paths": {
    "/customers": {
      "get": {
        "summary": "customers",
        "description": "Returns all customers",
        "operationId": "customers",
        "parameters": [
          {
            "name": "i",
            "in": "query",
            "description": "Offset of first row",
            "schema": {
              "type": ""
            }
          },
          {
            "name": "cnt",
            "in": "query",
            "description": "Number of rows to return",
            "schema": {
              "type": ""
            }
          }
        ],
        "responses": {
          "200": {
            "description": null
          }
        }
      }
    }
  }
}
```

```

    },
    "/customers.2": {
      "get": {
        "summary": "customersdotv2",
        "description": "Returns all customers (v2)",
        "operationId": "customersdotv2",
        "parameters": [
          {
            "name": "i",
            "in": "query",
            "description": "Offset of first row",
            "schema": {
              "type": ""
            }
          },
          {
            "name": "cnt",
            "in": "query",
            "description": "Number of rows to return",
            "schema": {
              "type": ""
            }
          }
        ],
        "responses": {
          "200": {
            "description": null
          }
        }
      }
    },
    "/v3/customers": {
      "get": {
        "summary": "customersv3",
        "description": "Returns all customers v3",
        "operationId": "customersv3",
        "parameters": [
          {
            "name": "i",
            "in": "query",
            "description": "Offset of first row",
            "schema": {
              "type": ""
            }
          },
          {
            "name": "cnt",
            "in": "query",
            "description": "Number of rows to return",
            "schema": {
              "type": ""
            }
          }
        ]
      }
    }
  }
}

```

```

        }
    },
    "responses": {
        "200": {
            "description": null
        }
    }
},
"/customers/{id}": {
    "get": {
        "summary": "customerbyid",
        "operationId": "customerbyid",
        "parameters": [
            {
                "name": "id",
                "in": "path",
                "description": "Customer ID",
                "required": true,
                "schema": {
                    "type": "string"
                }
            }
        ],
        "responses": {
            "200": {
                "description": null
            }
        }
    },
    "db": {
        "get": {
            "summary": "db",
            "description": "Retrieve list of table names",
            "operationId": "db",
            "responses": {
                "200": {
                    "description": null
                }
            }
        }
    },
    "db/{table}": {
        "get": {
            "summary": "tablebyid",
            "operationId": "tablebyid",
            "parameters": [
                {
                    "name": "table",

```

```

        "in": "path",
        "description": "Table name",
        "required": true,
        "schema": {
            "type": "string"
        }
    },
    {
        "name": "i",
        "in": "query",
        "description": "Offset of first row",
        "schema": {
            "type": ""
        }
    },
    {
        "name": "cnt",
        "in": "query",
        "description": "Number of rows to return",
        "schema": {
            "type": ""
        }
    }
],
"responses": {
    "200": {
        "description": null
    }
}
},
"/db/{table}/meta": {
    "get": {
        "summary": "tablemeta",
        "description": "Get table meta",
        "operationId": "tablemeta",
        "parameters": [
            {
                "name": "table",
                "in": "path",
                "description": "Table name",
                "required": true,
                "schema": {
                    "type": "string"
                }
            }
        ],
        "responses": {
            "200": {
                "description": null
            }
        }
    }
}

```

```

    }
  },
  "/db/{table}/{col}": {
    "get": {
      "summary": "column",
      "description": "Get a subset of a column",
      "operationId": "column",
      "parameters": [
        {
          "name": "table",
          "in": "path",
          "description": "Table name",
          "required": true,
          "schema": {
            "type": ""
          }
        },
        {
          "name": "col",
          "in": "path",
          "description": "Column name",
          "required": true,
          "schema": {
            "type": ""
          }
        },
        {
          "name": "i",
          "in": "query",
          "description": "Offset of first row",
          "schema": {
            "type": ""
          }
        },
        {
          "name": "cnt",
          "in": "query",
          "description": "Number of rows to return",
          "schema": {
            "type": ""
          }
        }
      ],
      "responses": {
        "200": {
          "description": null
        }
      }
    }
  },
},

```

```

    "/help": {
      "get": {
        "summary": "help",
        "description": "List all endpoints",
        "operationId": "help",
        "responses": {
          "200": {
            "description": null
          }
        }
      }
    },
    "/getCustomers": {
      "get": {
        "summary": "getCustomers",
        "description": "API like example endpoint that returns all
customers",
        "operationId": "getcustomers",
        "responses": {
          "200": {
            "description": null
          }
        }
      }
    }
  }
}
...

```

## Expose a RESTful interface

This section provides an example of exposing a RESTful API to a KDB-X-based system.

We load the REST server library and create a simple API and async query server.

### Example customers API

Create `customers.q` from the [source in the customers example](#).

Run `q` in the current directory and load the REST server module followed by `customers.q`:

```

q -p 8080
.com_kx_rest:use`kx.rest;
\l customers.q

```

In another terminal run:

```
curl http://localhost:8080/customers
[{"id":1,"name":"milglie"}, {"id":2,"name":"igfbage"}, {"id":3,"name":"kaodhbe"},
{"id":4,"name":"bafclbi"}, {"id":5,"name":"kfhogjn"}, {"id":6,"name":"jecpaen"},
{"id":7,"name":"kfmohpi"}, {"id":8,"name":"lklcoi"}, {"id":9,"name":"kfifpag"},
{"id":10,"name":"fglgofj"}]

curl http://localhost:8080/db/customers/meta
[{"c":"id","t":"j","f":"","a":""}, {"c":"name","t":"s","f":"","a":""}]

curl http://localhost:8080/getCustomers
[{"id":0,"nm":"mpnan"}, {"id":1,"nm":"nogel"}, {"id":2,"nm":"holpj"},
{"id":3,"nm":"kkfpn"}, {"id":4,"nm":"pegin"}, {"id":5,"nm":"jcgnm"},
{"id":6,"nm":"dlhep"}, {"id":7,"nm":"cmej1"}, {"id":8,"nm":"bfmjf"},
{"id":9,"nm":"lcicg"}]
```

For a full overview of the server and the concepts introduced, see [the customer example server](#).

## Protecting a backend API

We can provision a protected public gateway to allow access to the private HTTP backend.

With the [customers API example](#), you should have your server exposed on port 8080.

The guides below will look for an HTTP backend endpoint, this is the host and port of the Customers example.

To follow along, run the HTTP customers example on a VM instance and expose port 8080. We will then use a gateway to secure access to the REST server running on 8080.

In practice, you will want 8080 exposed to the virtual network the gateway is on, but not externally to the wider Web.

Tip:"Custom HTTP headers"

Your HTTP backend and Gateway should also consider requiring custom headers of your own design to quickly ignore foreign requests. This is not covered by the guides below.

## Amazon API Gateway

Amazon API Gateway can be used to secure a REST API backend.

Start up a VM instance and run the [customers example](#) on port 8080 and expose port 8080 on the VM's network.

Now use Amazon API Gateway to secure access to the customers API backend by importing an API configuration:

### [Setting up Amazon API Gateway](#)

A [sample OpenAPI 2.0 configuration](#) for the customers API is included from the appendix.

After importing the API configuration, create a stage and name it *kxce-stage*.

The *kxce-stage* should define a single stage variable `httpUrl`, and the value should be the URL of the customers backend.

Detail: "Set this variable to the URL without the protocol prefix"

For example, set `httpUrl` to `myserver.com`, not `http://myserver.com`.

After deploying the stage, your API will be accessible over HTTPS using the Stage URL; however it will not require authentication.

To require authentication to use an API, see See [Amazon API Gateway Authentication](#).

To use IAM for authentication, attach an IAM Authenticator under *Develop* > *Routes*.

## Azure API Management

Warning: "Azure Active Directory is now known as Microsoft Entra ID"

Azure API Management service instances can be used to create a gateway to allow access to one or more REST endpoints.

See [Azure API Management Key Concepts](#)

To protect your API backend, we recommend using *Microsoft Entra ID* as the OAuth2 provider when using the API Management Service.

If you do not already have a Microsoft Entra tenant, and an Azure API Management service instance running, create new ones. Note that this might take a while. The name of your Azure API Management service instance becomes part of the gateway URL.

Start up a VM instance and run the [customers example](#) on port 8080 and expose port 8080 on the VM's network.

We use Azure API Management to secure access to the customers API backend.

Once the Azure API Management Instance is up, import the [sample OpenAPI JSON](#) from the appendix.

After importing the API, click on the API's *Design* tab, and enter the Customer examples HTTP backend endpoint URL for all operations.

With this alone, you should now be able to access your REST server from the API Management gateway URL, which at the time of writing is shown in the *API Management Overview*.

The APIs are still unauthenticated at this point, however the Gateway should be functional.

Once you have a Microsoft Entra tenant running, go to *App Registrations* and follow the instructions for [making an API and a client app](#).

Be sure to follow the [last step](#): without it, OAuth2 is confusingly enabled, yet not required.

Note: "Note" At the time of writing, that last step mentions editing a policy in raw XML. You may also edit the policy visually in the current UI, under *Inbound Processing*, in the *Design* tab, adding a `validate-jwt` type policy.

After you have the backend protected as described in the Azure walkthrough, you might want to replace the step where you treat the Azure API Portal as the app, instead using something like `kur1` as the client app.

## Google Cloud API Gateway

Google API Gateway can be used to secure a REST API backend.

Start up a VM instance and run the [customers example](#) on port 8080 and expose port 8080 on the VM's network.

We will now use Google API Gateway to secure access to the customers API backend by importing a configuration.

See [Setting up Google API Gateway](#)

Note: "Installing Google API Gateway in the Console"

If you have not done so before, opening Google API Gateway in the Cloud Console will have you "install" and accept a user agreement.

Once you have Google API Gateway opened in the Google Console, you can import an API config.

When asked to import a configuration, you can use the [sample OpenAPI 2.0 configuration](#) for the customers API. Make sure to replace the `x-google-backend` address with the address of your VM running the REST server demo.

When asked to create a Gateway, you name it `kxce-gateway`, and choose a region.

Once Google finishes uploading your API, make a request to

```
https://GATEWAY_URL/customers
```

to use the customers API, where `GATEWAY_URL` is the URL listed in the Google Console's *Gateway* tab.

For example, the generated gateway URL would look like:

```
https://kxce-gateway-5uoiifib.ue.gateway.dev/customers
```

Your API is now accessible over HTTPS using the Gateway; however it is not yet secured.

To secure the API, read about the options Google provides.

- For simplicity, and to complete this demo, follow the [steps for securing with an API Key](#)
- For further authentication options, instead of using API keys, see [Authenticating with a Service Account](#)

## REST-Server Library API Reference

---

*This page introduces the REST-Server module API Reference for the `.com_kx_rest` namespace. It explains the available functions, utilities, and registration methods for working with the REST server API in KDB-X. Find details on initialization, endpoint registration, request processing, and utility functions, along with usage examples.*

`.com_kx_rest`.

**Initialization** [init](#) initialize the namespace

**Registration** [register](#) register an endpoint [reg.data](#) define a data item: input parameter or object member [reg.header](#) define a HTTP-header based input parameter [reg.body](#) define the expected POST body in the request [reg.output](#) define the output of the endpoint [reg.object](#) define an object for use as data element, body, or output [reg.default](#) get object populated with default values

**Utilities** [util.throw](#) throw an error [util.response](#) construct response [util.httpResponse](#) construct HTTP response

**Request processing** [process](#) process an incoming HTTP request

The examples on this page assume the following namespace alias has been created for convenience.

```
.rest:.com_kx_rest
```

### `.com_kx_rest.init`

*Initialize the library namespace*

Parameters:

```
opts    {dict}  Optional dictionary containing the following:
                autoBind {bool}          Whether to automatically bind `.z.ph` and
                `.z.pp` handlers.
                appendNewline {bool}      Whether to append newline at the end of
                every JSON response (except when the handler returns custom response).
```

Where called as a nullary, minimally initializes the namespace.

Where called as a unary with dictionary `opts`, initializes the namespace with further steps according to entries in `opts`:

```
autoBind  whether to automatically bind .z.ph and .z.pp handlers (boolean)
```

Note: "autoBind"

When true (``1b``) an incoming request is delegated to the next handler (if present) if it cannot be matched to an endpoint.

Otherwise a request that does not match an endpoint is rejected with HTTP code 404.

Examples:

```
.com_kx_rest.init[]  
.com_kx_rest.init enlist[`autoBind]!enlist[1b]
```

## `.com_kx_rest.process`

*Process an incoming HTTP request*

```
.com_kx_rest.process[method;request]
```

Where

- `method` is one of ``GET`POST` (symbol atom)
- `request` is an HTTP request (list of strings)

processes the HTTP request according to its `http-method` custom header (if present), otherwise according to the value of `method`

Example: set the REST processor as the kdb+ handlers for HTTP GET and POST calls:

```
.z.ph:.rest.process[`GET;]  
.z.pp:.rest.process[`POST;]
```

Refer to:

- `.z.ph`,
- `.z.pp`

## `.com_kx_rest.reg.body`

*Define the input request body expected by the post-based endpoint*

```
.com_kx_rest.reg.body[typ;isReq;dfv;dscr]
```

## Where

|       |                                                                               |
|-------|-------------------------------------------------------------------------------|
| typ   | name of object (defined using <code>.com_kx_rest.reg.object</code> ) (symbol) |
| isReq | whether the body is required (bool)                                           |
| dfv   | default value, of a type compatible with the object (any)                     |
| dscr  | human-readable description (string)                                           |

defines the input request body expected by the post-based endpoint.

Example: create a new customer object

```
.rest.reg.object[`customerObj;
  .rest.reg.data[`id;-6h;1b;0N;"Customer ID"],
  .rest.reg.data[`name;10h;1b;"";"Customer name"] ]

.rest.register[`post;"/customers";
  "Creates one or more customers";
  {`customers upsert cols[customers]#x`data;count customers};
  .rest.reg.body[`customerObj;1b;;; "One or more customer object"] ]
```

## .com\_kx\_rest.reg.data

*Register a data item*

```
.com_kx_rest.reg.data[nm;typ;isReq;dfv;dscr]
```

## Where

|       |                                                                                                 |
|-------|-------------------------------------------------------------------------------------------------|
| nm    | name of parameter/item (symbol)                                                                 |
| typ   | q datatype, or object name (defined using <code>.com_kx_rest.reg.object</code> ) (short symbol) |
| isReq | whether parameter is required (boolean)                                                         |
| dfv   | default value, which must be of typ (any)                                                       |
| dscr  | human readable description (string)                                                             |

defines a data item that can be used as one of

- a path-parameter or query-string based input parameter (when invoked in the context of specifying `params` argument of `.com_kx_rest.register` function)
- an element of an object (when invoked in the context of specifying `items` argument of `.com_kx_rest.reg.object` function)

## .com\_kx\_rest.reg.default

*Construct an object using the default values of its elements*

```
.com_kx_rest.reg.default nm
```

Where **nm** (symbol) is the name of an object, constructs it using the default values of its elements.

This is useful as the default value of an input parameter object.

See `.com_kx_rest.reg.object` for an example.

## `.com_kx_rest.reg.header`

*Define a header-based input parameter*

```
.com_kx_rest.reg.header[nm;typ;isReq;dfv;dscr]
```

Where

|       |                                                                                                 |
|-------|-------------------------------------------------------------------------------------------------|
| nm    | name of parameter/item (symbol)                                                                 |
| typ   | q datatype, or object name<br>defined using <code>.com_kx_rest.reg.object</code> (short symbol) |
| isReq | whether parameter is required (boolean)                                                         |
| dfv   | default value, which must be of a type that is compatible with <code>`typ`</code> (any)         |
| dscr  | human-readable description (string)                                                             |

defines a header-based input parameter.

## `.com_kx_rest.reg.object`

*Register an object*

```
.com_kx_rest.reg.object[nm;items]
```

Where

|       |                                                                                  |
|-------|----------------------------------------------------------------------------------|
| nm    | name of object, globally unique (symbol)                                         |
| items | one or more data elements, defined by <code>.com_kx_rest.reg.data</code> (table) |

registers an object, which can then be used as the datatype of an input parameter, request body, or output.

An example to define an object that contains another object, and an endpoint that takes the containing object as input, and returns the nested one:

```

.rest.reg.object[`nestedObj;
  .rest.reg.data[`prop1;6h;1b;0#0Ni;"property 1"],
  .rest.reg.data[`prop2;11h;1b;0#`;""],
  .rest.reg.data[`prop3;0h;0b;("v1";"v2");""],
  .rest.reg.data[`prop4;11h;0b;1#`a_value;"" ]

.rest.reg.object[`containerObj;
  .rest.reg.data[`id;-6h;0b;100;"id"],
  .rest.reg.data[`name;10h;0b;"xyz";"a name"],
  .rest.reg.data[`properties;`nestedObj;0b;.rest.reg.default`nestedObj;
    "A nested object" ] ]

.rest.register[`post;"/nested";
  "demonstrates nested object";
  {x[`data]`properties}; // Returns nested object
  .rest.reg.body[`containerObj;0b;.rest.reg.default`containerObj;
    "One or more container objects"],
  .rest.reg.output[`nestedObj;1b;"Resulting object" ] ]

```

## .com\_kx\_rest.reg.output

*Define the output of an endpoint*

```
.com_kx_rest.reg.output[typ;isReq;dscr]
```

Where

|       |                                                               |
|-------|---------------------------------------------------------------|
| typ   | name of object, defined with .com_kx_rest.reg.object (symbol) |
| isReq | whether output is required (bool)                             |
| dscr  | human-readable description (string)                           |

defines the output of the endpoint.

## .com\_kx\_rest.register

*Register an endpoint*

```
.com_kx_rest.register[op;path;dscr;fn;params]
```

Where

|      |                                                                          |
|------|--------------------------------------------------------------------------|
| op   | REST operation, typically one of get, post, put, or delete (symbol atom) |
| path | path: supports variables using {var} syntax (e.g. "/users/{id}")         |

```
(string)
dscr      human-readable description (string)
fn        handler function (fn) see below
params    one or more user input parameter definitions, or empty list
           if no parameters are defined, see .com_kx_rest.reg.data (dict|table)
```

registers an endpoint

```
.rest.register[`get;"/customers/{id}";
  "Returns one or more customers by their IDs";
  {[id] select from customers where uid in id};
  .rest.reg.data[`id;6h;1b;0;"One or more customer IDs" ] ]
```

Handler function:

- if the function is defined with arguments named as the keys or columns of `params` (or `body` if `params` has only a `body` key or column) then the function is variadically invoked with its arguments mapped from the request input
- otherwise, the function is invoked as a unary on a dictionary:

```
op        operation of the endpoint (symbol)
path      path of the endpoint (string)
arg       input parameters (dict)
           as defined by params argument of .com_kx_rest.register
rawArg    input parameters as received in the request, without parsing (dict)
data      value of processed body, typically a dictionary
           if the body is of an object type, but can be of any type
           as specified in call to .com_kx_rest.reg.body
rawData   raw kdb+ form of the request body (if present) (any)
hdr       HTTP headers as received (dict)
```

and returns one of

- a kdb+ data structure (typically a dictionary or a table), serialized to JSON by the framework
- result of the `.com_kx_rest.util.response` function, which gives the handler control over the HTTP status code, and content type of the response (available with release 1.0.0)
- result of the `.com_kx_rest.util.httpResponse` function, which gives the handler total control over the response

In the event of a problem (possibly with input) the handler must call `.com_kx_rest.util.throw` to signal an error.

## `.com_kx_rest.util.httpResponse`

*Construct and return endpoint's HTTP response*

```
.com_kx_rest.util.httpResponse[code;headers;cbt]
```

Where

|         |                                                                         |
|---------|-------------------------------------------------------------------------|
| code    | HTTP status code (string)                                               |
| headers | headers dictionary: keys and values are strings (dict)                  |
| cbt     | Content body, encoded according to Content-Type set in headers (string) |

returns endpoint's HTTP response, allowing control over the HTTP headers.

When this function is used, the handler must set the proper **Content-Type** header and encode **cnt** accordingly.

## **.com\_kx\_rest.util.response**

*Constructs and return endpoint's HTTP response given HTTP status code*

```
.com_kx_rest.util.response[code;cnttype;cnt]
```

Where

|         |                                                                  |
|---------|------------------------------------------------------------------|
| code    | HTTP status code (e.g. "201") (string)                           |
| cnttype | content type (one of the keys of <code>`.h.ty`</code> ) (symbol) |
| cnt     | content body, already encoded according to cnttype (string)      |

returns the endpoint's HTTP response.

Use this function when you need to control the HTTP status code, as well as the type of the content.

If the success status code is 200 and the content type is JSON, the endpoint handler can return its result directly (e.g. dict or table); the framework will encode this as JSON and return it along with 200 status code.

## **.com\_kx\_rest.util.throw**

*Signal an error*

```
.com_kx_rest.util.throw[msg;subj]
```

Where

```
msg    error message (string)
subj   subject of the error,e.g. names of input parameters (string)
```

signals an error formatted as `error-message|error-subject`.

Use this function in an endpoint handler to signal an error in a format that distinguishes between the subject of the error, and the error message itself.