Documentation

Release 5.0.6.1

XDAC - Data Access Connector

Jun. 16, 2025

Table of contents

CHAPTER 1

General information

1.1 Getting to know XDAC

XSQUARE-Data Access Connector (**XDAC**) is a standalone web server that provides APIs based on your database. XDAC allows you to automate the API publishing process by providing access to procedures and functions stored on the side of your database.

Using XDAC allows you to focus on data rather than implementing business logic, while reducing errors and duplicate code, making maintenance easier. Thus, XDAC speeds up software development, allowing the developer to avoid the mundane process of writing the same type of code.

XDAC supports the following databases:

- PostgreSQL
- Microsoft SQL Server
- Oracle
- Firebird

1.2 How does XDAC work?

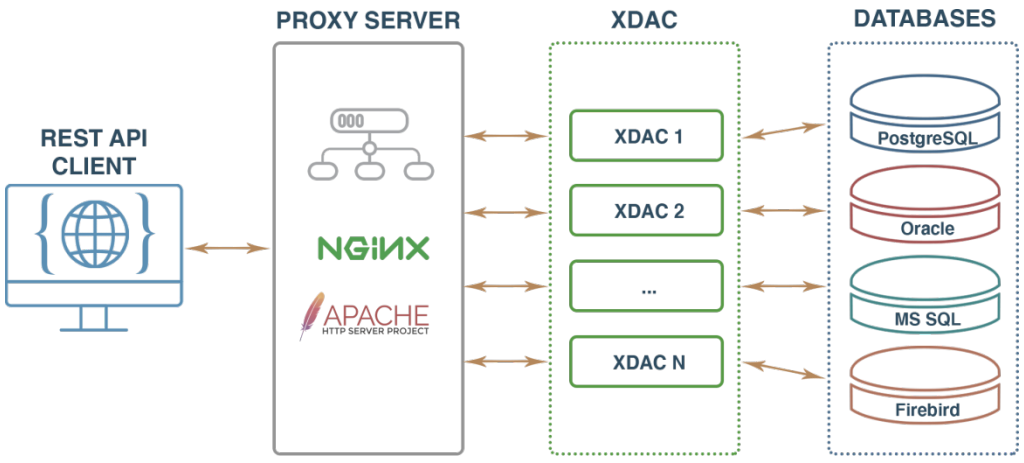
When connecting to a database, XDAC receives a set of procedures and functions. The resulting list is written to the application cache, which is then used to match the requested procedures with the parameters passed in the request. Each procedure and function in the list is published as an endpoint of the XDAC application. Access is provided by the name of the procedure or function, which is specified in the service call path. Parameters passed to a procedure or function are matched by name. The result of the function is in most cases converted into a scalar json value and given to the end user.

CHAPTER 2

Architecture and system requirements

2.1 Architecture

The application architecture is a web server that processes client HTTP requests by providing access to procedures and functions stored on the database side and returns the result to the client.



2.2 Runtime environment

Supported architectures:

- x86-64
- ARM
- Loongson

Supported OS:

- DEB-based - any
- RPM-based - any
- Debian 12 - recommended

Supported DBMSs:

- PostgreSQL
- Oracle
- Microsoft SQL Server
- Firebird

2.3 System requirements

- CPU - 1 Core
- RAM - 100 MB
- HDD - 100 MB + Logs

Installation of virtualization/containerization system, operating system, database is carried out at the discretion of the Administrator based on the needs of the Organization.

CHAPTER 3

Installation

Below you can find details on installing XDAC using Debian as an example.

All commands should be run with root privileges.

Create a directory for the XDAC distribution

```
mkdir /root/xsquare
```

Change directory

```
cd /root/xsquare
```

Download/receive the distribution to the created directory

```
wget https://xsquare.dev/files/xdac/xsquare.xdac.5.0_latest_release.deb
```

Install XDAC

```
dpkg -i xsquare.xdac.5.0_latest_release.deb
```

Check the status of the service

```
systemctl status xsquare.xdac
```

CHAPTER 4

Configuration files

4.1 Configuration file config.json

For the correct operation, a configuration file config.json must be present in the directory with the service.

The configuration file contains 2 sections:

1. **app** – application operation parameters
2. **db** – parameters for connection to the database

```
{
  "app":
  {
    "dbType": "postgresql",
    "port": "8887",
    "parsingSchema": "api",
    "reloadCacheInterval": 300,
    "debug": true
  },
  "db":
  {
    "login": "app_user",
    "password": "app_password",
    "domain": "10.150.117.209",
    "port": 3050,
    "dbName": "xdac_demo",
    "minCons": 1,
    "maxCons": 10,
    "runtimeOptions":
    {
      "LC_NUMERIC": "en_US.UTF-8"
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The following application settings are set in the **app** section:

- **dbType** – string. Defines to which database the connection will be made. Supported values: postgresql, oracle, mssql, firebird
- **port** - string. Specifies the number of the network port on which the service will be started (default is 8887)
- **parsingSchema** – String. Defines:
  - **schema name in PostgreSQL and MS SQL Server database, in which procedures and functions will be scanned.**
  - **package name in Oracle and Firebird database, in which procedures and functions will be scanned.**
- **reloadCacheInterval** – number. Defines the time interval in seconds for updating the cache of procedures and functions. Default value: 0. If the value is zero, the cache refresh mechanism is not started.
- **debug** – logical variable. Determines whether the extended application logging mode is enabled.

The **db** section specifies the settings of the database to be connected:

- **login** - user name
- **password** - password
- **domain** - domain name or IP address of the server
- **port** - port
- **minCons** - minimum number of connection pool sessions
- **maxCons** - maximum number of connection pool sessions
- **dbName** - database name
- **LC_NUMERIC** - locale for number formatting

Note: Example configuration files for the various databases can be found in the examples directory of the XDAC distribution.

CHAPTER 5

XDAC - Data Access Connector

5.1 Operation

XDAC is installed as a service, to start which you need to run the command:

```
systemctl start xsquare.xdac
```

Note: When installing from a deb or rpm package, the service starts automatically when the installation is complete.

To run XDAC, make sure you have an xdac.lic license file and a properly configured config.json configuration file.

If the parameters of the configuration file are changed, the service must be restarted:

```
systemctl restart xsquare.xdac
```

You can check the status of the service by using the command:

```
systemctl status xsquare.xdac
```

Errors in the application server will be logged; you can display the messages using the command:

```
journalctl -u xsquare.xdac
```

It is also possible to check the server status remotely, using a GET request to the status endpoint:

```
curl http://localhost:8887/status
```

In response, XDAC will send the following json format:

```
{
  "status": "UP",
  "version": "5.0.6.1",
  "build": "05.06.2025_17:32:09",
  "license": "license information"
}
```

5.2 DBMS customization. User rights.

Installation of the operating system, database, reverse-proxy server, and locale is carried out at the discretion of the Administrator based on the needs of the Organization. An example of the advanced setup can be found in the documentation 'Administering XSQUARE-PGHS 5', CHAPTER 4 'Advanced Setup'.

Each database scan requires a user with the ability to create a session and EXECUTE rights for the corresponding package/function. Access rights to dependent objects are determined at the discretion of the Administrator. To display errors related to the rights, it is necessary to set the flag "debug":true in the configuration file to display the extended error log.

Note: Examples of how to create a user with the necessary permissions can be found in the 'example' directory of XDAC distribution.

5.3 Scanning procedures and functions

At startup, XDAC establishes a connection to the database according to the parameters specified in the config.json configuration file.

After a successful connection, the application server scans for procedures and functions in the specified **schema** or database **package** and forms a **cache of procedures and functions**.

Only procedures and functions that meet the following conditions are written to the cache:

- Procedures with input parameters or without input parameters but without a return value.
- Functions with or without input parameters, with a single return value.

5.4 Calling procedures and functions

Each procedure and function resulting from the schema scan is published by the service with the prefix **/xdac/rpc/**:

```
http://localhost:8887/xdac/rpc/function_name
```

In general, GET and POST requests can be used to call a procedure or function. When using a GET request, the parameters to be passed are specified in the URL using a query string:

```
curl "http://localhost:8887/xdac/rpc/function_name?first_param=value&second_
↪param=value"
```

When using POST-request, the passed parameters should be included in the request body json, the name of which fields will correspond to the name of arguments, or passed as form data (multipart/form-data):

- Parameters in the body of json:

```
curl -d '{ "first_param": value, "second_param": value  }' http://localhost:8887/xdac/
↪rpc/function_name
```

- Parameters as form data:

```
curl --form first_param=value --form second_param=value localhost:8887/xdac/rpc/
↪function_name
```

As an example, let us create a function for a PostgreSQL database to add two numbers of the following type:

```
CREATE OR REPLACE FUNCTION api.add_numbers (
    first integer,
    second integer
)
RETURNS integer AS
$body$
BEGIN
    return first + second;
END;
$body$
LANGUAGE 'plpgsql'
```

An example of calling the add_numbers function using a GET request:

```
curl "http://localhost:8887/xdac/rpc/add_numbers?first=20&second=30"
```

The answer is 50

An example of calling the add_numbers function using a POST request with parameters in the body of json:

```
curl -d '{ "second": 5, "first": 6  }' http://localhost:8887/xdac/rpc/add_numbers
```

The answer is 11

An example of calling the add_numbers function using a POST request with multipart/form-data parameters:

```
curl --form first=12 --form second=34 http://localhost:8887/xdac/rpc/add_numbers
```

The answer is 46

In case of incorrect parameter name, missing parameter or passing an extra parameter, an error message in json format will be received in response. For example:

```
curl -d '{ "second1": 5, "first": 6  }' http://localhost:8887/xdac/rpc/add_numbers
```

Response:

```
{
  "code": "XDAC005",
  "hint": "Make sure that routine name is correct or try to reload application",
  "message": "Routine add_numbers(first,second1) doesn't exists in app cache",
  "details": ""
}
```

If the procedure or function parameters have default values, these parameters will be considered optional and may not be present in the query. For example, if our function add_numbers has the following format:

```
create function add_numbers(first integer DEFAULT 1, second integer DEFAULT 2)
```

in this case the function can be called without parameters:

```
curl http://localhost:8887/xdac/rpc/add_numbers
```

The answer is 3

Note: Each call to a procedure or function is made in its own transaction. The presence of the operator `COMMIT` in the body of the function will result in an error.

5.5 Naming rules

For PostgeSQL, MS SQL Server databases, all names (procedure and function names, argument names) are **case-sensitive**.

To call a function named StRaNgeNamE, call the method as follows:

```
curl "http://localhost:8887/xdac/rpc/StRaNgeNamE"
```

A call in a different format will result in an error that no such function was found:

```
{
  "code": "XDAC001",
  "hint": "Make sure that routine name is correct or try to reload applicaion",
  "message": "Routine strangename was not found in app cache",
  "details": ""
}
```

For Oracle and Firebird databases**, all names will be automatically converted to uppercase**.

5.6 Passing JSON as a parameter

If you want to pass json as a parameter to a function or procedure, you must pass the headers *Content-Type: application-json* и *Prefer:params=single-object* in the request:

```
curl -H "Content-Type: application/json" -H "Prefer: params=single-object" -d '{"text
↪":"Test message","input":[1,2,3]}' "http://localhost:8887/xdac/rpc/jsonInput"
```

The procedure or function itself should have only one argument with a data type suitable for json processing.
For example, for PostgreSQL:

```
CREATE OR REPLACE FUNCTION api."jsonInput" (
    "myPar" json
)
RETURNS json AS
$body$
```

(continues on next page)

(continued from previous page)

```
begin
return "myPar";
END;
$body$
LANGUAGE 'plpgsql'
```

5.7 Passing a binary file as a parameter

If a binary file is required as a parameter of a function or procedure, the **Content-Type: application/octet-stream** header must be passed in the request:

```
curl  "http://localhost:8887/xdac/rpc/upload_binary"  \
    -X POST -H "Content-Type: application/octet-stream" \
    --data-binary "@file_name.ext"
```

With such requests, the called procedure or function must have one mandatory parameter to handle binary data.

For example, for PostgreSQL:

```
CREATE OR REPLACE PROCEDURE api."upload_binary" (
    bytea
)
AS
$body$
BEGIN
    insert into api.files(blob)
    values ($1);
END;
$body$
LANGUAGE 'plpgsql'
```

You can also pass a binary file using a multipart/form-data request. XDAC will search for a procedure or function according to the name of the parameters passed, except for parameters that contain a file. All files that are passed in the request will be placed in the 'service' **xdac_files**.

This table contains the following fields:

- **name** - name of the parameter in the request
- **file_name** - file name
- **file_mime** - MIME type of file
- **file_content** - file content

Depending on the database type, the xdac_files table will be created in different ways:

- PostgreSQL – temporary table pgtemp.xdac_files

```
CREATE TEMP TABLE IF NOT EXISTS pg_temp.xdac_files
```

- Oracle – global temporary table XDAC_FILES

```
CREATE GLOBAL TEMPORARY TABLE XDAC_FILES
```

- MS SQL Server – temporary table xdac_files

```
CREATE TABLE #xdac_files
```

- Firebird – global temporary table XDAC_FILES

```
CREATE GLOBAL TEMPORARY TABLE XDAC_FILES
```

Note: When the current request (transaction) is completed, the information about the uploaded files will be deleted.

5.7.1 Example of working with files for PostgreSQL database

Let us create a save_document function in the api schema:

```
CREATE OR REPLACE FUNCTION api.save_document (
  document varchar,
  document_date varchar
)
RETURNS jsonb LANGUAGE 'plpgsql'
AS
$body$
declare
  l_rec record;
BEGIN
  select f.file_name, f.file_mime, length(f.file_content) file_length
  into l_rec
  from pg_temp.xdac_files f
  where f.name = 'document_scan';

  return jsonb_build_object(
          'document','document",
          'document_date','document_date",
          'file_name',l_rec.file_name,
          'file_mime',l_rec.file_mime,
          'file_length',l_rec.file_length
    );
END;
$body$;
```

Make a request to the application server that transfers the file my_image.svg:

```
curl --form document=passport --form document_date=10.01.2002 --form document_scan=
↪'@my_image.svg'  localhost:8887/xdac/rpc/save_document
```

Response:

```
{
  "document": "passport",
  "document_date": "10.01.2002",
  "file_length": 9710,
  "file_mime": "image/svg+xml",
  "file_name": "my_image.svg"
}
```

5.8 Working with headers

All headers sent to the service are recorded in the session parameters.

The following parameters are available for access:

- **request.headers** – request headers
- **request.method** – request method
- **request.path** – the path on which the service was called

In order to set the response headers and response code, the parameters **response.headers** and **response.status** are used.

The **Content-Type** header is set by XDAC itself. Any value passed into this header will be overwritten.

The storage of parameter data is implemented depending on the `database`:

- **PostgreSQL – parameters** are written to **Grand Unified Configuration (GUC),** thus working with parameters is done through **set_config** and **current_settings** calls**.**
  For example:

```
-- Get all the headers. They are converted to json
SELECT current_setting('request.headers', true)::json;

-- The method by which the service was called
SELECT current_setting('request.method', true); --GET/POST

-- The path from which the service was called
SELECT current_setting('request.path', true); --/xdac/rpc/test

— To set the response headers, you need to pass an array of json objects
SELECT set_config('response.headers','[{"Content-Type":"application/json"},{
↔"Set-Cookie":"foo=bar"}]',true);

-- sets the response code to 405
SELECT set_config('response.status','405',true);
```

- **Oracle** – parameters are written to the **XDAC_VARIABLES** package, the **SET_CONFIG** procedure is used for writing, the **GET_CONFIG** function is used for reading, which work with the **CONFIG_TABLE** table:

```
CREATE OR REPLACE PACKAGE XDAC_VARIABLES AS
TYPE config_record IS RECORD (
    key_name CLOB,
    value_name CLOB
);
TYPE config_table IS TABLE OF config_record INDEX BY PLS_INTEGER;
g_config config_table;
PROCEDURE SET_CONFIG(p_key CLOB, p_value CLOB);
FUNCTION GET_CONFIG(p_key CLOB) RETURN CLOB;
END XDAC_VARIABLES
```

- **MS SQL Server** – parameters are written to the session context via **sp_set_session_context**, values can be retrieved via **SELECT SESSION_CONTEXT**
- **Firebird** – parameters are written to **USER_TRANSACTION** context of user session with **rdb$set_context**, for reading **rdb$get_context** is used.

For example:

```
SELECT rdb$set_context('USER_TRANSACTION', 'request.headers', '') FROM rdb
↪$database
SELECT rdb$get_context('USER_TRANSACTION', 'request.headers'') FROM rdb
↪$database
```

5.9 Error handling

If an error occurs, the service will return it in the format:

```
{
  "code": "",
  "hint": "",
  "message": "",
  "details": ""
}
```

For PostgreSQL database, the service provides the ability to handle user errors caused by `the raise` operator

```
raise 'This is postgresql error'
      USING ERRCODE = 'SRV01',
      HINT = 'Try GET method',
      DETAIL = 'This is detail';
```

XDAC converts the error to the following form:

```
{
  "code": "SRV01",
  "hint": "Try GET method",
  "message": "This is postgresql error",
  "details": "This is detail"
}
```

In this case, the transaction will be completed, and all changes will be canceled using the `ROLLBACK` operator. If it is necessary to pass a user error to the service, while saving the results of the function execution, you can use the response status setting via GUC.

```
perform set_config('response.status','405',true);
perform    set_config('response.headers','[{"Content-Type":"application/json"}]',true);
return json_build_object('message','Only GET method is allowed');
```

5.10 Updating the cache of procedures and functions

All procedures and functions in a schema or package are scanned at application startup and placed in the cache. If you change the signature of procedures or create new functions, they will not be available for API calls. For updated procedures or functions to become available, the cache must be refreshed. This can be accomplished as follows:

1. Reboot the application
2. Set in config.json the parameter reloadCacheInterval in seconds, which is responsible for the time interval for cache refresh

3. Contact endpoint RELOAD to force a cache update:

```
curl http://localhost:8887/reload
```

There is also a mechanism available for PostgreSQL database to refresh the cache using NOTIFY notifications. To do this, you need to send a cache reload notification to the xdac channel:

```
NOTIFY xdac, 'reload';
```

To automate the sending of notifications to the channel you can create an **event trigger on ddl operation**. A SUPERUSER can create the following trigger:

```
CREATE OR REPLACE FUNCTION public.xdac_drop_watch () RETURNS event_trigger
LANGUAGE plpgsql
AS
$body$
DECLARE
  obj record;
BEGIN
 FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
  LOOP
   IF obj.object_type in ('function','procedure') AND not obj.is_temporary then
     NOTIFY xdac, 'reload';
   end if;
 END LOOP;
END;
$body$;



CREATE OR REPLACE FUNCTION public.xdac_ddl_watch () RETURNS event_trigger
LANGUAGE plpgsql
$body$
DECLARE
  cmd record;
BEGIN
 FOR cmd IN SELECT * FROM pg_event_trigger_ddl_commands()
 LOOP
    IF cmd.command_tag IN ('CREATE FUNCTION', 'ALTER FUNCTION','CREATE PROCEDURE',
↔'ALTER  PROCEDURE')
      and cmd.schema_name != 'pg_temp'
   then
     NOTIFY xdac, 'reload';
   end if;
 END LOOP;
END;
$body$;


CREATE EVENT TRIGGER xdac_ddl_watch  ON ddl_command_end  EXECUTE PROCEDURE public.
↔xdac_ddl_watch();
CREATE EVENT TRIGGER xdac_drop_watch ON sql_drop  EXECUTE PROCEDURE public.xdac_drop_
↔watch();
```