

O₂DB Access User Manual

O₂C Interface

Release 5.0 - May 1998



Information in this document is subject to change without notice and should not be construed as a commitment by O₂ Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

The software can only be used or copied in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's own use.

Copyright 1992-1998 O₂ Technology.

All rights reserved. No part of this publication can be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopy without prior written permission of O₂ Technology.

O₂, O₂API, O₂C, O₂DBAccess, O₂Engine, O₂Graph, O₂Kit, O₂Look, O₂Store, O₂Tools, and O₂Web are registered trademarks of O₂ Technology.

SQL and AIX are registered trademarks of International Business Machines Corporation.

Sun, SunOS, and SOLARIS are registered trademarks of Sun Microsystems, Inc.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

Unix is a registered trademark of Unix System Laboratories, Inc.

HPUX is a registered trademark of Hewlett-Packard Company.

BOSX is a registered trademark of Bull S.A.

IRIX is a registered trademark of Siemens Nixdorf, A.G.

NeXTStep is a registered trademark of the NeXT Computer, Inc.

Purify, Quantify are registered trademarks of Pure Software Inc.

Windows is a registered trademark of Microsoft Corporation.

All other company or product names quoted are trademarks or registered trademarks of their respective trademark holders.

Who should read this manual

This manual describes how to use O₂DBAccess. This O₂ module enables to connect O₂ applications to relational databases on remote hosts, and to import and export data from and to such systems. O₂DBAccess provides class libraries (O₂C and C++) for these tasks. The manual also describes how to invoke SQL statements from O₂. An example program is presented.

Other documents available are outlined, click below.

See [O2 Documentation set](#).



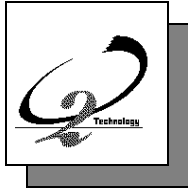


TABLE OF CONTENTS

This manual is divided into the following chapters:

- 1 - Introduction
- 2 - Utilization
- 3 - Classes
- 4 - Appendices



TABLE OF CONTENTS

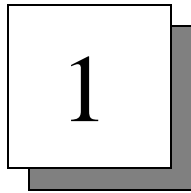
1	Introduction	9
	1.1 System Overview.....	10
	O2Engine	12
	O2 Store	13
	O2DB Access	14
	1.2 Manual Overview.....	16
2	Utilization	17
	2.1 The o2dbaccess schema	18
	Classes.....	18
	Importing the schema.....	19
	2.2 Guidelines	20
	2.3 Accessing a database.....	22
	Host connection and database log in.....	22
	Open Context	23
	2.4 Preparing a statement.....	24
	Linking statement and context	24
	Managing contexts	24
	Transferring data	25
	2.5 Run statement and fetching data.....	29
	2.6 Commit and rollback.....	30
	2.7 Ending a session.....	31
3	Classes	33
	3.1 O2DBAccess.....	34
	server_error method	35
	3.2 Connection.....	36
	connect method	37
	disconnect method.....	38
	logon method.....	39

TABLE OF CONTENTS

	logoff method	40
3.3	Session	41
	close method	42
	commit method	43
	open method	44
	rollback method	45
	sqlquery method	46
3.4	Context	47
	associate method	48
	define_bind method	49
	define_projection method	50
	exec method	51
	fetch method	52
4	Appendices	53
4.1	Example Application	54
	Define the schema	55
	Host connection and database log on	56
	Open a context	57
	Prepare the statement	58
	Run the statement	59
	Fetch the data	60
	Close the context	61
	Close database session and end host connection	61
4.2	Configuration File	62
4.3	Possible Errors	63
	INDEX	69



TABLE OF CONTENTS



Introduction

Congratulations! You are now a user of O₂DBAccess!

O₂DBAccess is the O₂ module that enables you to communicate and work with relational databases on remote hosts

This chapter introduces the O₂ system and O₂DBAccess and outlines its various features and advantages. An overview of this User Manual is then given.

1.1 System Overview

The system architecture of O₂ is illustrated in [Figure 1.1](#).

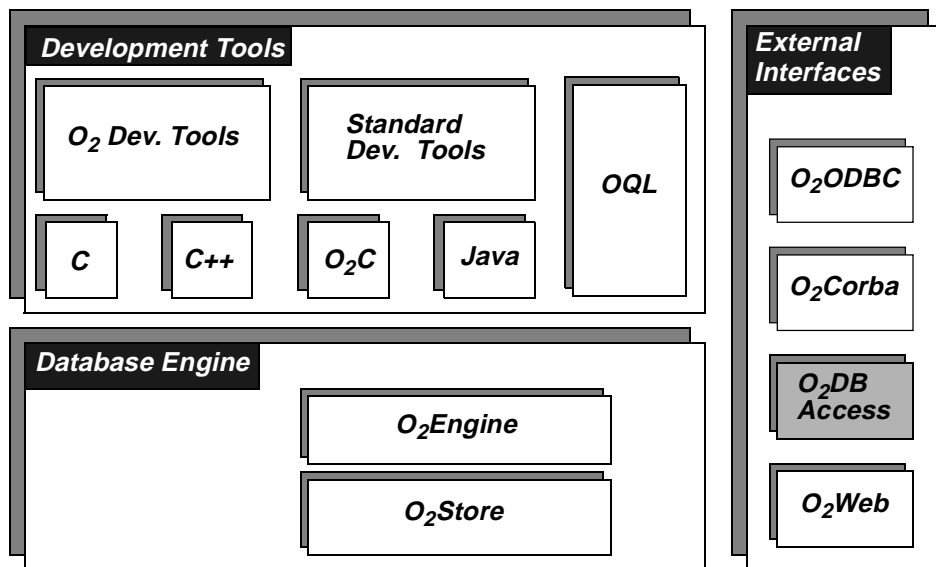


Figure 1.1: O₂ System Architecture

The O₂ system can be viewed as consisting of three components. The *Database Engine* provides all the features of a Database system and an object-oriented system. This engine is accessed with *Development Tools*, such as various programming languages, O₂ development tools and any standard development tool. Numerous *External Interfaces* are provided. All encompassing, O₂ is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

- O₂Store The database management system provides low level facilities, through O₂Store API, to access and manage a database: disk volumes, files, records, indices and transactions.
- O₂Engine The object database engine provides direct control of schemas, classes, objects and transactions, through O₂Engine API. It provides full text indexing and search capabilities with O₂Search and spatial indexing and retrieval capabilities with O₂Spatial. It includes a Notification manager for informing other clients connected to the same O₂ server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O₂ system.

System Overview

Programming Languages:

O₂ objects may be created and managed using the following programming languages, utilizing all the features available with O₂ (persistence, collection management, transaction management, OQL queries, etc.)

- C O₂ functions can be invoked by C programs.
- C++ ODMG compliant C++ binding.
- Java ODMG compliant Java binding.
- O₂C A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
- OQL ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O₂ objects and methods.

O₂ Development Tools:

- O₂Graph Create, modify and edit any type of object graph.
- O₂Look Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O₂Kit Library of predefined classes and methods for faster development of user applications.
- O₂Tools Complete graphical programming environment to design and develop O₂ database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

- O₂Corba Create an O₂/ Orbix server to access an O₂ database with CORBA.
- O₂DBAccess Connect O₂ applications to relational databases on remote hosts and invoke SQL statements.
- O₂ODBC Connect remote ODBC client applications to O₂ databases.
- O₂Web Create an O₂ World Wide Web server to access an O₂ database through the internet network.

O₂Engine

O₂Engine has all the features of a database engine providing transparent management of data persistence, data sharing and data reliability, as well as all the features of an object-oriented system including the manipulation of complex objects with identity, classes, types, methods, multiple inheritance, overriding and late binding of methods.

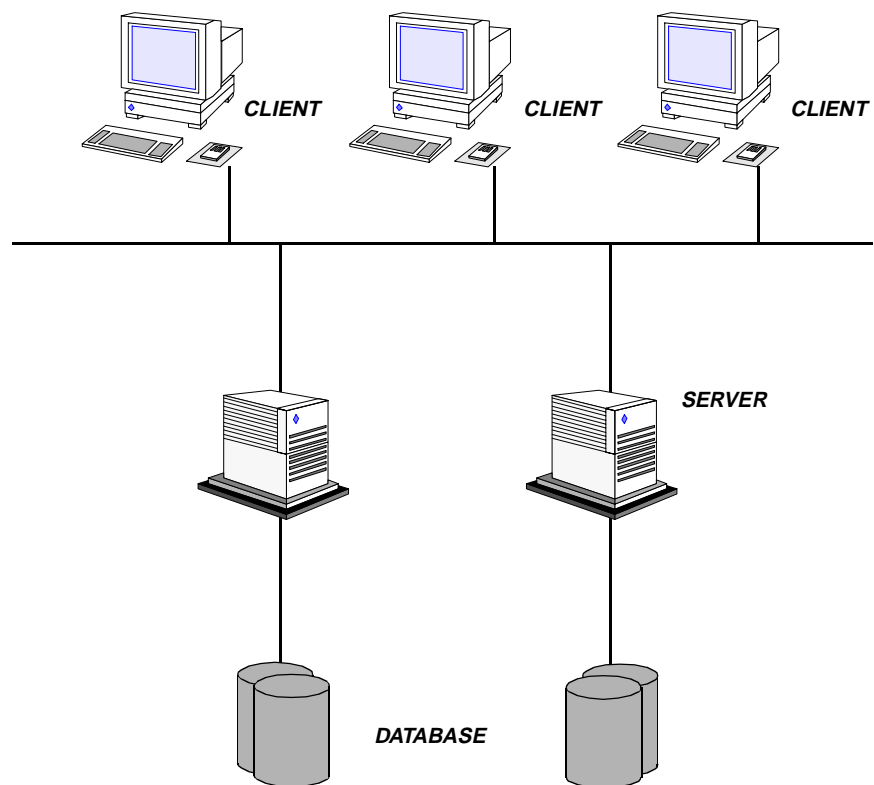


Figure 1.2: Client/server architecture

O₂ Store

The O₂Store physical storage management system offers you the following features:

- Transactional management of persistent structures.
- Client/ server architecture.
- Rollbacks and crash recovery.

O₂Store has the client/ server architecture shown in [Figure 1.2](#). The server process provides persistence, disk management, concurrency control, data recovery and database security.

The features offered by O₂Engine and O₂Store are shown in [Figure 1.3](#) below.

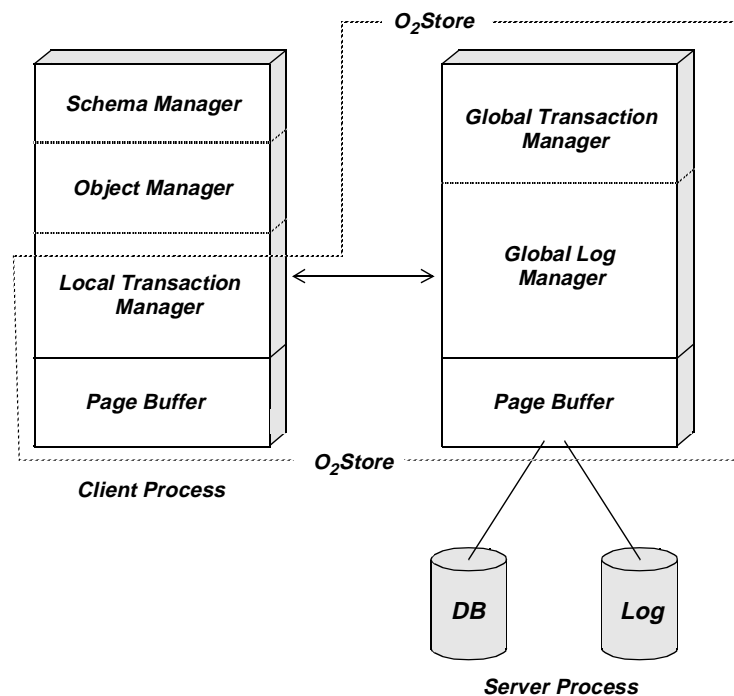


Figure 1.3: Global architecture showing O₂Store layer

O₂DB Access

O₂DBAccess is a set of O₂ classes that enables O₂ applications to communicate and work with relational databases on remote hosts.

These classes allow you to carry out the following actions from your applications:

- Connect to a server and set up a session on a remote database.
- Run any SQL statement in the SQL syntax of that database.
- Fetch data as required from the database to the user application into O₂ objects.
- Mirror the commit and rollback facilities of some databases.
- Close the database session and terminate the connection to the host.

You can also retrieve error message text corresponding to database error codes.

O₂DBAccess is based on the SequeLink protocol as shown in [Figure 1.4](#). This is a software package that enables a client application to access simultaneously different relational databases residing on different servers that are connected to one or more types of local networks.

SequeLink uniformly manages the different network protocols and the heterogeneity between platforms¹. With O₂DBAccess you can link to any platform currently supported for SequeLink.

1. For information about all possible network-host-database combinations, call O₂Line.

System Overview : O2DB Access

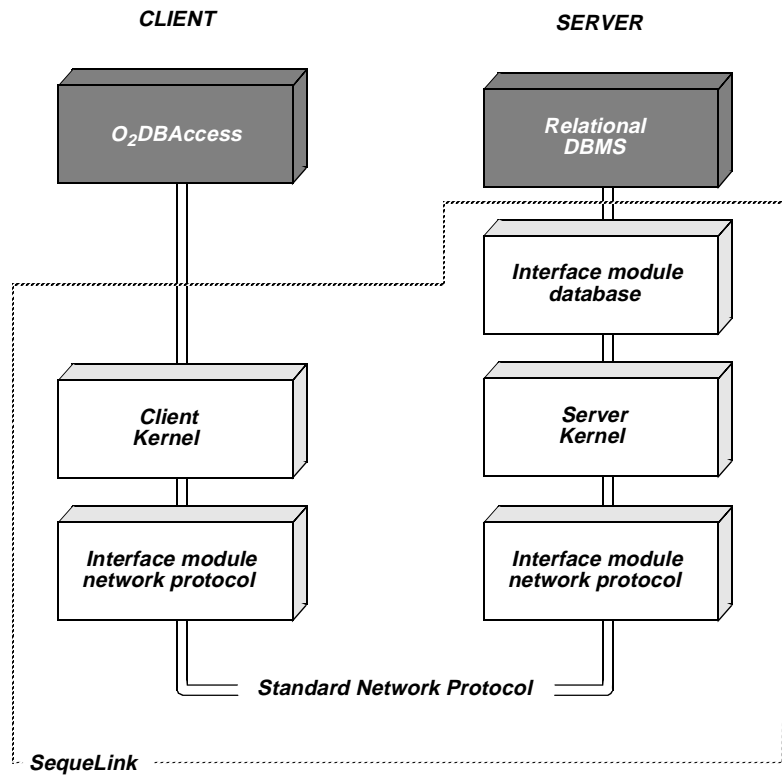


Figure 1.4: O₂DBAccess and SequeLink

1.2 Manual Overview

This manual is divided up into the following chapters:

- **Chapter 1 - Introduction**

A short introduction to the O₂ system, O₂Engine, O₂Store and O₂DBAccess.

- **Chapter 2- Utilization**

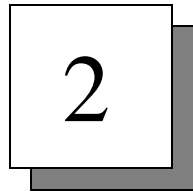
This chapter describes how to use O₂DBAccess: accessing a database, preparing a statement, fetching data, commit and rollback and running the statement.

- **Chapter 3 - Classes**

This chapter details all the various classes and methods of the O₂DBAccess schema: `o2DBAccess`, `Connection`, `Session`, and `Context`.

- **Chapter 4 - Appendices**

This chapter includes an example program. It gives possible error codes and the configuration file.



Utilization

This chapter details how to use O₂DBAccess.

It is divided into the following chapters:

- [The o2dbaccess schema](#)
- [Guidelines](#)
- [Accessing a database](#)
- [Preparing a statement](#)
- [Run statement and fetching data](#)
- [Commit and rollback](#)
- [Ending a session](#)

2.1 The o2dbaccess schema

O₂DBAccess is in fact a standard O₂ schema called `o2dbaccess` that you can use in any of your user-defined schemas.

Classes

The `o2dbaccess` schema, shown in [Figure 2.1](#), has classes that enable you to communicate and work with the remote database.

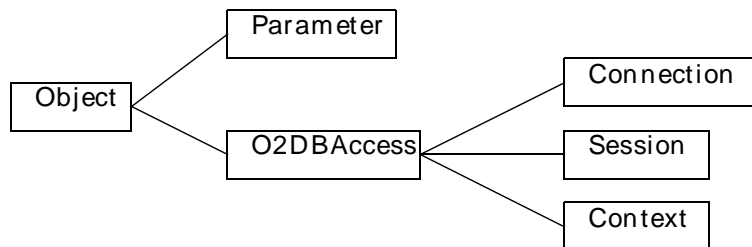


Figure 2.1: `o2dbaccess` schema

The schema classes are as follows:

- **`o2DBAccess`**

An `o2DBAccess` object defines the common resources of `O2DBAccess` classes. It contains one method, `server_error`, which you use to obtain the RDBMS-detected error codes.

- **`Connection`**

`Connection` is a subclass of `o2dbaccess`. A `Connection` object defines and maintains a connection to a remote host.

- **`Session`**

`Session` is a subclass of `o2dbaccess`. A `Session` object defines and maintains a connection to a database server. It manages the transactions of the session.

- **`Context`**

`Context` is a subclass of `o2dbaccess`. A `Context` object defines an access context to a database and contains information that is required to run an SQL statement.

- **`Parameter`**

Any classes used for the result object and parameters must be a subclass of the `Parameter` class.

Importing the schema

Import the `o2dbaccess` schema using the O₂ `import` command:

```
import schema o2dbaccess class Parameter, Connection,  
Session, Context;
```

This `import` command gives you access to any `o2dbaccess` classes.

If you are using O₂ Tools, you see the class hierarchy shown in Figure 2.2.

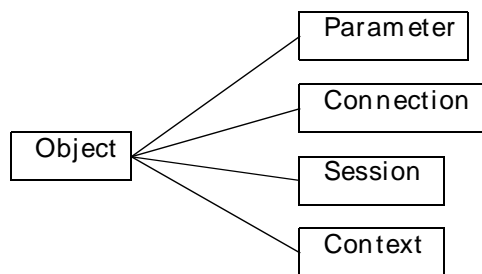


Figure 2.2: Imported O₂DBAccess classes

You can also import the `O2DBAccess` class if you are going to use generic error messages.

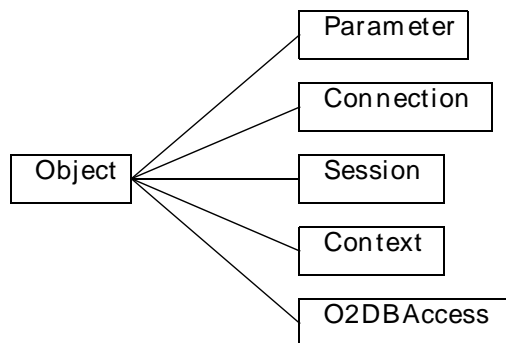


Figure 2.3: Import O₂DBAccess class

Note

For fuller details of these classes and their methods, refer to Chapter 3.

2.2 Guidelines

To send an SQL statement for processing on remote database, you need to carry out the following steps:

1. Set up a connection to the host **server** machine.
2. Set up a session on the database by logging on.
3. Open an access context for the statement.
4. Prepare the statement to be run. This means putting information about the statement in the context.
5. Run the statement.
6. If the statement is a select statement, fetch the data from the database to your application into an O₂ object.
7. If information has been inserted, updated or deleted, make the changes permanent, or undo them.
8. If you do not want to rerun the statement, close the context.
9. Close the session by logging off the database.
10. End the connection by disconnecting from the host.

Guidelines

Each step corresponds to a particular method. [Table 2.1](#) shows these methods and its corresponding step.

The class of the method is given after the method name.

These methods constitute the basic set of methods that you need to use.

Table 2.1

Method set

Methods	Step
<code>connect@Connection</code>	Set up a connection
<code>logon@Connection</code>	Set up a session
<code>open@Session</code>	Open an access context
<code>associate@Context</code> <code>define_projection@Context</code> <code>define_bind@Context</code>	Prepare a statement
<code>exec@Context</code>	Run a statement
<code>fetch@Context</code>	Fetch data from the database
<code>commit@Session</code>	Make the changes permanent
<code>rollback@Session</code>	Undo a transaction
<code>close@Session</code>	Close the context
<code>logoff@Connection</code>	End the session
<code>disconnect@Connection</code>	End the connection

All these various steps are detailed below.

For a description of each specific method refer to Chapter 3.

Note

Steps 3 to 8 can be reduced using the method `sqlquery` of the class `Session`. See [Section 3.3](#) for more details.

2.3 Accessing a database

You must first connect to the remote host and log onto the database.

Host connection and database log in

With O₂DBAccess, you do this by creating a **Connection** object. You must call the **connect** method on this object to connect to the remote host. You then call the **logon** method on this object to log onto the database and begin a session. The **logon** method returns a **Session** object.

Once connected and logged on, you can run as many SQL statements as you want.

For example:

```
run body {
  o2 Connection host = new Connection;      Create the object
  o2 Session session;
                                           Connection to the host defined by where
  host->connect ("where", "username", "password");
                                           Log on the database defined by db name
  session = host->logon ("db name", "db username/password").session;

  /* Some transactions */

  host->logoff (session);                    End session

  host->disconnect;                          Disconnect from the host
};
```

When your transactions with the database are finished, you use the **logoff** and **disconnect** methods to respectively end the session and remote host connection.

Refer to [Section 3.2](#) for a full description of all these **Connection** methods.

Accessing a database : Open Context

Open Context

Once connected to the remote host and logged on to the database, you must now open a context for each SQL statement you want to run. You do this using the `open` method from your `session` object (result of the `logon` method). The context contains the statement itself and any additional information that may be needed to run the statement. This method returns an object of class `Context`.

A `Context` object remains open until you explicitly close it using the `close` method (of your `session` object), or until you end the database session using the `logoff` method. However, you do not need to close a context in order to use it for a different statement. You simply re-use it.

For example:

```
run body {
  o2 Connection host = new Connection;
  o2 Session      session;
  o2 Context      context1, context2;

  host->connect ("where", "username", "password");
  session = host->logon ("db name", "db username/password").session;
  context1 = session->open.context;
  /* statements */
  session->close(context1);
  context2 = session->open.context;
  /* statements */
  session->close(context2);
  host->logoff(session);
  host->disconnect;
};
```

Contexts and how to manage contexts are explained fully in [Section 2.4](#) below and [Section 3.3](#) gives a description of the `open` method.

Note

The context is local to the logon session in which it is used and the number of contexts you can open at the same time is restricted to 100. However, you rarely need more than 15 and the external database or yourself can impose a lower limit.

2.4 Preparing a statement

The next step after accessing the database is to prepare the SQL statements you want to run.

Linking statement and context

You must firstly link the statement to its context using the `associate` method of the `Context` class.

This method associates the statement to the opened context in order to pass information about the statement to the database server.

The information in the context can be used for a type checking on the client side.

The `associate` also sends the statement to the RDBMS database for validation. It is at this point that any SQL syntax errors are trapped. If any are found, you can get the database error codes by calling the `server_error` method on the `Context` object (see [Section 3.1](#) for details of this method and [Section 4.3](#) for a list of possible errors).

Finally, the `associate` method stores the statement in the context.

If the SQL statement does not need a result object (i.e. it is not a select statement) and it contains no parameter markers, `O2DBAccess` needs no more information.

You can therefore immediately run your statement using the `exec` method. See [Section 2.5](#) for more details.

However, if the statement a select statement and/ or it contains parameter markers, you must provide `O2DBAccess` with more information. This is explained in the remainder of this section.

Managing contexts

When you want to run a select statement or a statement that contains parameter markers or both, you need to provide more information before the statement is run.

The information needed includes the result object and its projection list which you store in the context using the `define_projection` method of the class `Context`.

You also need to store any parameters in the context using the `define_bind` method.

Preparing a statement : Transferring data

You can manage your contexts in three different ways:

1. Use one context for one specific SQL statement and close the context as soon as the statement has been run.
2. Open a context, use it for one statement, and then reuse for another statement by simply associating it to the new statement.

You can do this as many times as you want until you want to close the context.

3. Open a context for a particular statement that you want to run several times. You keep the context open and associated to the statement until you no longer wish to rerun the statement.

While the context is open, you can rerun the statement with new values for any of its parameters using the `exec` and `fetch` methods. However, you cannot redefine any objects using the `define_projection` or `define_bind` methods.

Note

Re-associating a context frees all defines.

Transferring data

The transfer of data between the application and the database involves the following steps:

1. You must firstly define the classes of objects where you want the data to be buffered. These classes must be subclasses of the `Parameter` class.
2. You must then give the relevant transfer information. This includes the objects in which the data is to be buffered and whether these objects define a parameter or the result object.

You do this by defining data buffers using the `define_projection` and `define_bind` methods described above. These methods store the transfer information in the context.

You can then run the statement using the `exec` method. If you are transferring data from the database, you must fetch it from the database to the designated result object using the `fetch` method.

This section now describes these steps in more detail.

- **Data buffers**

The data buffers are O₂ objects, the classes of which are user-defined and must be inherited from the **Parameter** class.

Define the class type as follows knowing that you can choose any collection types:

- For a parameter, the class type must be atomic and match the scalar type of the associated parameter marker in the SQL statement.
- The result of a select statement is a relation. A relation is a collection of tuples whose attributes have scalar types. If the relation has only one attribute, you can use a collection of atoms.

If you want to fetch data row by row or if the relation has only one row, you can omit the collection and use a tuple (or an atom for a one attribute relation).

Note

You can encapsulate collection elements and tuple attributes.

- **Defining buffers**

You give the necessary information about the result object and the parameters using the **define_projection** and **define_bind** methods.

The **define_projection** method has a projection mechanism with which you can define the result object.

You can declare a tuple type with more attributes, in a different order and with different names than in the relation. To do this, you must give a link between the tuple attributes and the relation attributes.

This link is called the projection list and is made up of a list of attribute names where the *i*th member of the list obtains the value of the *i*th attribute of the relation.

The other tuple attributes get their default O₂ values.

For example, if you want to obtain data from the relation:

```
Relation [A: integer, B: char(1), C: float, D: char(30)]
```

Preparing a statement : Transferring data

You then want to store the result in an object of the **Employees** class, which is defined as follows:

```
class Employees inherit Parameter type
  list (tuple (name: string,
               code: integer,
               entry_date: Date, Imported from o2kit
               dept_id: char,
               salary: real))
end;
```

Once connected, logged on and a context is opened, you must create a new **Employees** object:

```
o2 Employees result_object = new Employees;
```

Associate your statement to the opened context.

```
context->associate("SELECT A, B, C, D FROM Relation");
```

Define the result object using the **define_projection** method (the **entry_date** attribute is not be valued):

```
context->define_projection(result_object,
                          list("code", "dept_id",
                              "salary", "name"));
```

The **define_bind** method declares an object that will contain the value of a variable used in the SQL statement. A second parameter defines the position of this variable in the SQL statement.

Note

With ORACLE, the variable markers must be called: **":1"**, **":2"**, etc.

For example, you want to update the **C** attribute where the **B** attribute is **'C'** and **'T'** in the following relation:

```
Relation[A: integer, B: char(1), C: float, D: char(30)]
```

For this, you define the **Char** class as follows.

```
class Char inherit Parameter public type char end;
```

Once connected, logged on and a context is opened, you must create a new **Char** object.

```
o2 Char parameter = new Char;
```

Associate your statement to the opened context.

```
context->associate("UPDATE Relation SET C=C * 1.1 WHERE B=?");
```

Define the result object using the `define_bind` method.

```
context->define_bind(parameter, 1);
```

Fix the parameter value and run the statement.

```
*parameter = 'C';  
context->exec;
```

You can rerun the statement with another value.

```
*parameter = 'T';  
context->exec;
```

Run statement and fetching data

2.5 Run statement and fetching data

If the SQL statement does not need a result object (i.e. it is not a select statement) and it contains no parameter markers, you can run your statement using the `exec` method.

If the statement is a select and/ or it contains parameter markers, you must provide the relevant transfer information as detailed above in [Section 2.4](#), and then transfer the data using the `fetch` method.

For example, suppose you set up a single connection, a single session in which you want to run three SQL statements: a create statement, an insert statement containing three parameters, which you want run twice, and a select statement. You call the following sequence of methods in order to process each step.

```
run body {
  o2 Connection host = new Connection;
  o2 Session    session;
  o2 Context    context1, context2, context3;

  host->connect ("where", "username", "password");
  session = host->logon ("db name",
                      "db username/password").session;
  context1 = session->open.context;
  context1->associate ("CREATE...");
  context1->exec;
  session->close (context1);
  context2 = session->open.context;
  context2->associate ("INSERT...?...?...?...");
  context2->define_bind (...);
  context2->define_bind (...);
  context2->define_bind (...);
  context2->exec;
  ...
  context2->exec;
  session->commit;
  session->close (context2);
  context3->associate ("SELECT...");
  context3->define_projection (...);
  context3->exec;
  ...
  context3->fetch (...);
  session->close (context3);
  host->logoff (session);
  host->disconnect;
};
```

2.6 Commit and rollback

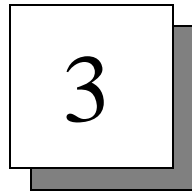
The `commit` and `rollback` methods mirror a feature of some databases in allowing you to systematically and explicitly make permanent or unroll a series of related database actions at strategic points in a session.

2.7 Ending a session

When all your transactions with the database are complete, the methods `logoff` and `disconnect` end the session and connection respectively, close the context, etc.

```
run body {
  o2 Connection host = new Connection;
  o2 Session      session;
  o2 Context      context1, context2;

  host->connect ("where", "username", "password");
  session = host->logon ("db name",
                       "db username/password").session;
  context1 = session->open.context;
  /* statements */
  session->close(context1);
  context2 = session->open.context;
  /* statements */
  session->close(context2);
  host->logoff(session);
  host->disconnect;
};
```

Classes

CLASS SET AND THEIR METHODS

O₂DBAccess is a set of O₂ classes that enables O₂ applications to communicate and work with relational databases on remote hosts.

This chapter details all these classes and their respective methods.

It is divided into the following sections:

- **O2DBAccess**
- [Connection](#)
- [Session](#)
- [Context](#)

3.1 O₂DBAccess

An `o2dbaccess` object defines the common resources of O₂DBAccess classes.

It contains one method, the `server_error` method with which you can obtain the RDBMS-detected error codes.

server_error method

Summary	Gives the RDBMS-detected error code.
Syntax	<i>receiver</i> -> server_error
Arguments	None.
Description	You use this method to obtain RDBMS-detected error codes.
Returns	The code and its description. The type of the returned value is as follows: <code>tuple (code: integer, msg: string)</code> The <code>code</code> attribute contains the RDBMS-detected error code and the <code>msg</code> attribute contains its textual description.

Example

```
#include "o2dbaccess.h"

o2 Context context;
o2 integer status;
o2 tuple(code: integer, msg: string) error;
...
status = context->associate("SELECT * FROM emp");
if (status != O2DB_OK) {
    if ( status == o2dbE_SERVER ) {
        error = host->server_error;
        printf("Server Error (%d): %s\n", error.code, error.msg);
        ...
    }
}
```

3.2 Connection

A `Connection` object defines and maintains a connection to a remote host.

The information needed to set up a link between the O₂ application and the host system is found in a configuration file. It is made up of a set of network and host-specific parameters.

Refer to [Section 4.2](#) for a full description of the configuration file.

This section now describes the methods associated to the `Connection` class:

- `connect method`
- `disconnect method`
- `logon method`
- `logoff method`

Connection : connect method

connect method

- Summary** Connects to a remote host.
- Syntax** `receiver->connect (c_name, username, password)`
- Arguments**
- `c_name` is a string that defines the link parameters used in the configuration file.
 - `username` is a string specifying the host user name.
 - `password` is a string specifying the host user password.
- Description** The `connect` method sets up a link between the workstation and a remote SequeLink server.
- Returns** 0 if the connection is successful or an error code if not.
- Example**

```
o2 Connection host = new Connection;
o2 integer status;
...
    supra_server:TCP:salome:LSPSUPRA2:::15:false
    This is the corresponding resource line in the configuration file.

status = host->connect("supra_server", "scott", "TIGER");
if (status != 0) {
    printf("error in connection (%d)\n",status);
    return;
}
```

disconnect method

Summary	Disconnects the remote host.
Syntax	<i>receiver</i> -> disconnect
Arguments	None
Description	The disconnect method disconnects the link between the workstation and the remote SequeLink server. All the current sessions of this connection are closed.
Returns	Nothing

Example

```
o2 Connection host = new Connection;
o2 integer status;
...
    supra_server:TCP:salome:LSPSUPRA2:::15:false
    This is the corresponding resource line in the configuration file.

status = host->connect("supra_server", "scott", "TIGER");
if (status != 0) {
    printf("error in connection (%d)\n",status);
    return;
}
...
host->disconnect;
```

Connection : logon method

logon method

Summary Logs onto the database.

Syntax `receiver->logon (logon1, logon2)`

Arguments `logon1, logon2` are strings that specify the parameters required in order to log onto the database you are using.

Refer to the documentation "Using SequeLink with your Database and Server" for the specific logon parameters needed for the database you want to use. The parameter length must be less than 256 characters.

Description The `logon` method passes both the `logon1` and `logon2` parameters to the database server. It then starts a new session by creating an object of class `Session`.

Returns An object defining a link with the database, or an error code if something goes wrong.

The type of returned value is:

```
tuple (retcode: integer, session: Session)
```

Example

```
o2 Connection host = new Connection;
o2 tuple (retcode : integer, session : Session) status;
o2 Session session;
...
log on SUPRA Server

status = host->logon("O2SCDEMO",
                    "connect o2tech identified by beaut");
if (status.retcode != 0) {
    printf("error in log on (%d)\n", status.retcode);
    return;
}
session = status.session;
...
```

logoff method

- Summary** Ends a logon session
- Syntax** `receiver->logoff (session)`
- Arguments** `session` is a **Session** object that represents the session to close.
- Description** The **logoff** method ends a specified logon session and releases all its resources. All contexts created during the session are closed.
- Note that the receiver object of the **logoff** method must be the same as the receiver object of the **logon** method that originally created the **Session** object.
- Returns** 0 if successfully logged off or an error code if not.

Example

```
o2 Connection host = new Connection;
o2 integer status;
o2 Session session;
...

status = host->logoff(session);
if (status != 0) {
    printf("error in log off (%d)\n", status);
    return;
}
```


3.3 Session

A `session` object defines and maintains the connection to a database server.

The `session` class has the following methods:

- `close method`
- `commit method`
- `open method`
- `rollback method`
- `sqlquery method`

close method

- Summary** Closes a context.
- Syntax** `receiver->close (context)`
- Arguments** `context` is a **Context** object that represents the context to close. The context must have been created by the receiver.
- Description** The `close` method closes a context and releases all its resources.
- Note that the receiver object of the `close` method must be the same as the receiver object of the `open` method that originally created the **Context** object.
- Returns** 0 if the method is successful or an error code if not.

Example

```
o2 Session session;
o2 Context context;
o2 integer status;
o2 tuple (retcode : integer, context : Context) open_status;
...
open_status = session->open;
if (open_status.retcode != 0) {
    printf("error in open (%d)\n", open_status.retcode);
    return;
}
context = open_status.context;
...
status = session->close(context);
...
```

Session : commit method

commit method

Summary	Commits modifications.
Syntax	<i>receiver</i> -> commit
Arguments	None
Description	The commit method commits what has been done in the session since last commit or start of the session.
Returns	0 if the method is successful or an error code if not.

Example

```
o2 Connection host = new Connection;
o2 Session session;
o2 integer status;
o2 tuple (retcode : integer, session : Session) log_status;
...
log_status = host->logon("O2SCDEMO",
                        "connect o2tech identified by beaut");
if (log_status.retcode != 0) {
    printf("error in log on (%d)\n", log_status.retcode);
    return;
}
session = log_status.session;
...
status = session->commit;
...
```

log on SUPRA Server

open method

Summary	Creates a new context.
Syntax	<i>receiver</i> -> open
Arguments	None.
Description	The open method creates a new access context for the receiver by creating an object of class Context .
Returns	An object defining the new context, or an error code if something went wrong. The type of returned value is: <code>tuple (retcode: integer, context: Context)</code>

Example

```
o2 Session session;
o2 Context context;
o2 integer status;
o2 tuple (retcode : integer, context : Context) open_status;
...
open_status = session->open;
if (open_status.retcode != 0) {
    printf("error in open (%d)\n", open_status.retcode);
    return;
}
context = open_status.context;
...
```

Session : rollback method

rollback method

- Summary** Reverses all modifications since last commit.
- Syntax** `receiver->rollback`
- Arguments** None.
- Description** The `rollback` method rolls back any modifications you have carried out in this session since last commit.
- Returns** 0 if the method is successful or an error code if not.

Example

```
o2 Connection host = new Connection;
o2 Session session;
o2 integer status;
o2 tuple (retcode : integer, session : Session) log_status;
...
log_status = host->logon("O2SCDEMO",
                        "connect o2tech identified by beaut");
if (log_status.retcode != 0) {
    printf("error in log on (%d)\n", log_status.retcode);
    return;
}
session = log_status.session;
...
status = session->rollback;
...
```

log on SUPRA Server

sqlquery method

Summary	Runs a statement and fetches the result if required.	
Syntax	<i>receiver->sqlquery (result, stmt, params, projection)</i>	
Arguments	<i>result</i>	is a Parameter object specifying where the query result is stored, or nil if not required.
	<i>stmt</i>	is a string of less than 4096 characters representing the SQL statement to be run.
	<i>params</i>	is a list of Parameter objects specifying the query parameters.
	<i>projection</i>	is a list of string specifying the projection attributes. By default (projection = list()), the query result fully matches the O ₂ class type.
Description	This method runs a statement and can fetch the result. Only projection attributes are valued (others get their O ₂ default values). By default (projection = list()), the result totally matches the O ₂ class type.	
Returns	An integer of the number of fetched rows if successful: 0 if there is no more data to fetch or a negative number representing an error code.	

Example

```

class Employees inherit Parameter type
  list(tuple(name: string, department: integer, salary: real)) end;
class Integer inherit Parameter public type integer end;
class String inherit Parameter public type string end;
run body {
  ...
  o2 Session session;
  o2 integer status;
  o2 string stmt = "SELECT ename, esalary FROM emp\
                  WHERE deptno = ? and job = ?";
  o2 Employees res = new Employees;
  o2 Integer dept = new Integer;
  o2 String job = new String;
  ...
  *dept = 80;
  *job = "tailor";
  status = session->sqlquery(res, stmt, list(dept, job),
                           list("name", "salary"));
  ...
  res->display;
  ...
}

```

3.4 Context

A `Context` object defines an access context to a database and contains information that is required to execute an SQL statement.

The `Context` class has the following methods:

- `associate method`
- `define_bind method`
- `define_projection method`
- `exec method`
- `fetch method`

associate method

- Summary** Associates a statement with an opened context.
- Syntax** `receiver->associate (stmt)`
- Arguments** `stmt` is a string of less than 4096 characters representing the SQL statement to execute.
- Description** The `associate` method associates an SQL statement to the context receiver. You can reuse a context for a new query but the previous association is then lost.
- Returns** 0 if the method is successful or an error code if not.

Example

```
o2 Session session;
o2 Context context;
o2 integer status;
o2 tuple (retcode : integer,
          context : Context) open_status;
o2 string stmt = "SELECT ename, esalary\
                 FROM emp WHERE deptno = 80";

...
open_status = session->open;
if (open_status.retcode != 0) {
    printf("error in open (%d)\n", open_status.retcode);
    return;
}
context = open_status.context;
status = context->associate(stmt);
...
```

Context : define_bind method

define_bind method

- Summary** Stores the parameters in the context.
- Syntax** `receiver->define_bind (param, order)`
- Arguments**
- | | |
|--------------|---|
| <i>param</i> | is a Parameter object that will contain the value corresponding to a variable used in the SQL statement. |
| <i>order</i> | is an integer specifying the position of this variable in the SQL statement. Numbering begins at 1. |
- Description** The `define_bind` method declares a **Parameter** object that contains the current value corresponding to a variable used in the SQL statement. The second parameter defines the position of this variable in the SQL statement. If you use ORACLE, you must call the marker `":n"`, where n is the position of the variable in the statement.
- Returns** 0 if the method is successful or an error code if not.

Example

```
class Employees inherit Parameter type
  list(tuple(name: string, department: integer, salary: real)) end;
class Integer inherit Parameter public type integer end;
class String inherit Parameter public type string end;
run body {
  ...
  o2 Context context;
  o2 integer status;
  o2 string stmt = "SELECT ename, esalary FROM emp\
                    WHERE deptno = ? and job = ?";
  o2 Employees res = new Employees;
  o2 Integer dept = new Integer;
  o2 String job = new String;
  ...
  status = context->associate(stmt);
  status = context->define_projection(res, list("name", "salary"));
  ...
  status = context->define_bind(job, 2);
  status = context->define_bind(dept, 1);
  ...
}
```

define_projection method

- Summary** Stores the result object in the context.
- Syntax** `receiver->define_projection (result, projection)`
- Arguments**
- | | |
|-------------------|---|
| <i>result</i> | is a Parameter object specifying where the result of the query will be stored. The parameter must be not nil. |
| <i>projection</i> | is a list of strings specifying the projection attributes. By default (<code>projection=list()</code>), the query result matches the O ₂ class type. |
- Description** The `define_projection` method stores the result object in the context. The query result is assigned to the value of the result object. Only projection attributes are valued (others get their O₂ default values). By default (`projection=list()`), the query result matches the O₂ class type.
- Returns** 0 if the method is successful or an error code if not.
- Example**

```
class Employees inherit Parameter type
  list(tuple(name: string, department: integer, salary: real))
end;
run body {
  ...
  o2 Session session;
  o2 Context context;
  o2 integer status;
  o2 tuple (retcode : integer, context : Context) open_status;
  o2 string stmt = "SELECT ename, esalary\
                  FROM emp WHERE deptno = 80";
  o2 Employees res = new Employees;
  ...
  open_status = session->open;
  if (open_status.retcode != 0) {
    printf("error in open (%d)\n", open_status.retcode);
    return;
  }
  context = open_status.context;
  status = context->associate(stmt);
  ...
}
```

Context : exec method

exec method

Summary	Runs the query.
Syntax	<i>receiver</i> -> exec
Arguments	None
Description	The exec method binds the current values defined as input to the SQL statement and runs the query.
Returns	0 if the method is successful or an error code if not.

Example

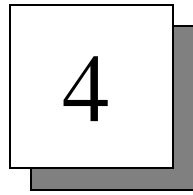
```
class Employees type
  list(tuple(name: string, department: integer, salary: real))
end;
class Integer public type integer end;
class String public type string end;
run body {
  ...
  o2 Context context;
  o2 integer status;
  o2 string stmt = "SELECT ename, esalary FROM emp
                  WHERE deptno = ? and job = ?";
  o2 Employees res = new Employees;
  o2 Integer dept = new Integer;
  o2 String job = new String;
  ...
  status = context->associate(stmt);
  status = context->define_projection(res,
                                     list("name", "salary"));
  ...
  status = context->define_bind(job, 2);
  status = context->define_bind(dept, 1);
  ...
  /* now we can run the query */
  *dept = 80;
  *job = "tailor";
  status = context->exec;
  ...
}
```

fetch method

Summary	Fetches the results.
Syntax	<i>receiver</i> -> fetch (<i>row_count</i>)
Arguments	<i>row_count</i> is an integer. The maximum number of rows to fetch. You can specify 0 (O2DB_ALL) in order to fetch as many rows as possible.
Description	The fetch method fetches the results into the result object.
Returns	An integer of the number of rows that were actually fetched. When no more rows can be fetched, it returns 0. If the fetch is unsuccessful, it returns an error code (a negative number). With o2dbw_NOT_UNIQUE, only the first row is stored in the result object.

Example

```
class Employees inherit Parameter type
  list(tuple(name: string, department: integer, salary: real)) end;
class Integer inherit Parameter public type integer end;
class String inherit Parameter public type string end;
run body {
#include "o2dbaccess.h"
  ...
  o2 Context context;
  o2 integer status;
  o2 string stmt = "SELECT ename, esalary\
                  FROM emp WHERE deptno = ? and job = ?";
  o2 Employees res = new Employees;
  o2 Integer dept = new Integer;
  o2 String job = new String;
  ...
  status = context->associate(stmt);
  status = context->define_projection(res, list("name", "salary"));
  ...
  status = context->define_bind(dept, 1);
  status = context->define_bind(job, 2);
  ... /* now you can run the query */
  *dept = 80;
  *job = "tailor";
  status = context->exec;
  status = context->fetch(O2DB_ALL);
  res->display;
  ...
  *job = "grocer";
  status = context->exec;
  status = context->fetch(10);
  res->display;
  ...
}
```



Appendices

This chapter contains the following appendices:

- [Example Application](#)
- [Configuration File](#)
- [Possible Errors](#)

4.1 Example Application

This example illustrates the various steps you must go through in order to use O₂DBAccess in order to send an SQL statement for processing on a remote database.

This section enables you to follow all these various steps and is divided up into the following sections:

- [Define the schema](#)
- [Host connection and database log on](#)
- [Open a context](#)
- [Prepare the statement](#)
- [Run the statement](#)
- [Close the context](#)
- [Fetch the data](#)
- [Close database session and end host connection](#)

Note

Commit or rollback any changes made - the **commit** and **rollback** methods depend on the database you are using are therefore not illustrated in this example.

Example Application : Define the schema

Define the schema

You begin by importing the `o2dbaccess` schema.

You then must define all the various classes (`Employees`, `Integer` and `String`) in which the result of your SQL statement and the query parameters are stored.

```
import schema o2dbaccess class Parameter, Connection,
Session, Context;

class Employees inherit Parameter type
  list (tuple (name: string,
              department: integer,
              salary: real))
end;

class Integer inherit Parameter public type integer end;

class String inherit Parameter public type string end;
```

Host connection and database log on

To connect to the host and log onto the database, you must create a `Connection` object.

You must then call the `connect` and `logon` methods on this object.

```
run body {
    o2 Connection host = new Connection;
    o2 Session sess;
    o2 Context  ctxt;
    o2 tuple (retcode: integer, sess: Session) log_status;
    o2 tuple (retcode: integer, ctxt: Context) open_status;
    o2 integer status;
    o2 string stmt = "SELECT ename, esalary\
                    FROM emp\
                    WHERE deptno = ? and job = ?";
    o2 Employees res = new Employees;
    o2 Integer dept = new Integer;
    o2 String job = new String;
    status = host->connect("supra_server", "scott", "TIGER");
    if (status != 0) {
        printf("error in connection (%d)\n", status);
        return;
    }
    log_status = host->logon("O2SCDEMO",
                          "connect o2tech identified by beaut");
    if (log_status.retcode != 0) {
        printf("error in log on (%d)\n", log_status.retcode);
        host->disconnect;
        return;
    }
}
```

Refer to [Section 2.3](#) and [Section 3.2](#) for more details.

Example Application : Open a context

Open a context

You open a context for each statement using the `open` method of the `Session` object.

This object contains the statement and other information needed for running the statement.

```
sess = log_status.sess;

open_status = sess->open;
if (open_status.retcode != 0) {
    printf("error in open (%d)\n", open_status.retcode);
    host->disconnect;

    Close all sessions and disconnect

    return;
}
```

Refer to [Section 2.3](#) and [Section 3.3](#) for further details.

Prepare the statement

You must associate a statement by firstly associating it with the opened context using the `associate` method.

The statement is a select statement and has parameters. You must therefore define and store in the context, the result object and its projection list using the `define_projection` method in connection with the class `Employees`. You define and store the parameters using the `define_bind` method in connection with the classes `Integer` and `String`.

```
ctxt = open_status.ctxt;
status = ctxt->associate(stmt);
if (status != 0) {
    printf("error in associate (%d)\n", status);
    host->disconnect;
    return;
}
status = ctxt->define_projection(res, list("name", "salary"));
if (status != 0) {
    printf("error in define_projection (%d)\n", status);
    host->disconnect;
    return;
}
status = ctxt->define_bind(job, 2);
if (status != 0) {
    printf("error in define_bind(2) (%d)\n", status);
    host->disconnect;
    return;
}
status = ctxt->define_bind(dept, 1);
if (status != 0) {
    printf("error in define_bind(1) (%d)\n", status);
    host->disconnect;
    return;
}
```

Refer to [Section 2.4](#) and [Section 3.4](#) for full details.

Example Application : Run the statement

Run the statement

All the relevant information has been given.

You can now run the statement using the `exec` method.

```
*dept = 80;
*job = "tailor";

status = ctxt->exec;
if (status != 0) {
    printf("error in exec(1) (%d)\n", status);
    host->disconnect;

    return;
}
```

Refer to [Section 2.5](#) and [Section 3.4](#) for further details of this method.

Fetch the data

As the statement is a select statement, you can transfer data from database to your application into an O₂ object using the `fetch` method.

```
status = ctxt->fetch(O2DB_ALL);
if (status < 0) {
    printf("error in fetch(1) (%d)\n", status);
    host->disconnect;
    return;
}
else
    printf("%d fetched rows\n", status);
res->display;
*job = "grocer";
status = ctxt->exec;
if (status != 0) {
    printf("error in exec(2) (%d)\n", status);
    host->disconnect;
    return;
}
status = ctxt->fetch(10);
if (status < 0) {
    printf("error in fetch(2) (%d)\n", status);
    host->disconnect;
    return;
}
else
    printf("%d fetched rows\n", status);
res->display;
```

Refer to [Section 2.5](#) and [Section 3.4](#) for more details.

Example Application : Close the context

Close the context

As the statement is not re-run, you can now close the context using the `close` method of the `Session` object.

```
status = sess->close(ctxt);
if (status != 0) {
    printf("error in close (%d)\n", status);
    host->disconnect;
    return;
}
```

Refer to [Section 2.7](#) and [Section 3.3](#).

Close database session and end host connection

All the your transactions with database are finished. The method `logoff` ends the session and the method `disconnect` ends the connection with the host.

```
status = host->logoff(sess);
if (status != 0) {
    printf("error in log off (%d)\n", status);
    host->disconnect;
    return;
}
host->disconnect;
}
```

Refer to [Section 2.7](#) and [Section 3.2](#) for more details.

4.2 Configuration File

The configuration file `o2dbaccess.cf` contains the information needed to link up your O₂ application and a remote host, in the form of a set of network and host-specific parameters.

You can change the file name by specifying a new name and its path in the environment variable `O2DBACCESS`. This variable must contain the full path of the file.

In order of priority, the file is first taken from the working directory, then `$HOME` and then the O₂ installation directory. It is an ASCII file where each line corresponds to a named link description. It contains one entry for each named link with comments beginning with `#`. The line format is:

```
c_name:network:host:service:lu_name:mode:max_contexts:type_checking
```

Each field is described below.

<i>c_name</i>	Link name. Specify a link name each time you invoke the <code>connect</code> method.
<i>network</i>	Type of network protocol used. Possible values: TCP (MacTCP on Macintosh), ADSP, APPC, DECnet, NetBIOS and AppleTalk.
<i>host</i>	RDBMS remote host/ node/ zone name.
<i>service</i>	SequeLink (database) service name that you want to connect to. You can find this name in the servermap file on the remote host. See SequeLink manual.
<i>lu_name</i>	Physical LU Name in APPC network protocol (optional).
<i>mode</i>	APPC mode in APPC network protocol (optional).
<i>max_contexts</i>	Integer (from 0 to 100) specifying maximum number of contexts that can be opened at the same time during a session. If 0, the default value (15) is used.
<i>type_checking</i>	Boolean specifying whether to enable (<code>true</code>) or disable (<code>false</code>) type checking in the <code>define_projection</code> and <code>define_bind</code> methods. Default value is <code>false</code> .

Warning !

The type checking is not supported for all RDBMS, call O₂Line for more details. An example configuration file is as follows:

```
supra on salome:TCP:salome:LSPSUPRA2:::15:false
db2 on sgph11:DECnet:sgph11:MVSDB2:::0:true
oracle:TCP:o2tech:oracle_services:::
```

Possible Errors

4.3 Possible Errors

Each method returns an internal error code.

This section describes these error codes. You can obtain RDBMS-detected error codes using the `server_error` method. See [Section 3.1](#) for a description of this method.

- **-1001**

Code: `o2dbE_SERVER (-1001)`
Call: All.
Cause: RDBMS-detected error has occurred.
Action: Consult error code and message text calling the `server_error` method.

- **-2001**

Code: `o2dbE_SQLNK (-2001)`
Call: All.
Cause: A SequeLink error has occurred. Should usually only be issued on a `connect` method call.
Action: Check the link parameters, user name and password. The connect failure reason is in `<network>srv.log` file on the remote host.

- **-3001**

Code: `o2dbE_STILL_CONNECT (-3001)`
Call: `Connection@connect`
Cause: You are still connected to a host.
Action: The `connect` method was called but not the `disconnect` method.

- **-3002**

Code: `o2dbE_NOT_CONNECT (-3002)`
Call: `Connection@logon`, `Connection@logoff`
Cause: Not connected.
Action: Check completion of the previous `connect` method call.

- **-3003**

Code: `o2dbE_NOT_MEMBER (-3003)`

Call: `Connection@logoff, Session@close`

Cause: The Session/ Context object has not been created by the receiver.

Action: Check the method call syntax.

- **-3004**

Code: `o2dbE_NOT_LOGON (-3004)`

Call: `Session@open, Session@close, Session@commit, Session@rollback, Session@sqlquery`

Cause: Not logged on.

Action: Check completion of the previous `logon` method call.

- **-3005**

Code: `o2dbE_TOOLONG (-3005)`

Call: `Connection@logon, Context@associate`

Cause: The parameters are too long.

Action: Decrease parameter length.

- **-3006**

Code: `o2dbE_NOT_OPEN (-3006)`

Call: `Context@associate, Context@define_projection, Context@define_bind, Context@exec, Context@fetch`

Cause: The context has not been opened.

Action: Check the completion of the previous `open` method call.

- **-3008**

Code: `o2dbE_NOSTMT (-3008)`

Call: `Context@associate, Session@sqlquery`

Cause: The SQL statement is empty.

Action: Check the method call syntax.

Possible Errors

- **-3009**

Code: `o2dbE_FILE_NOTFOUND (-3009)`
Call: `Connection@connect`
Cause: The configuration file was not found.
Action: Check that the configuration file exists.

- **-3010**

Code: `o2dbE_RC_NOTFOUND (-3010)`
Call: `Connection@connect`
Cause: No description of the link parameters for this link name.
Action: Check configuration file contents and the method call syntax.

- **-3011**

Code: `o2dbE_INVALID_RC (-3011)`
Call: `Connection@connect`
Cause: Invalid line in the configuration file.
Action: Check the configuration file contents.

- **-3012**

Code: `o2dbE_UNKN_NETWORK (-3012)`
Call: `Connection@connect`
Cause: Unknown network protocol.
Action: Check configuration file contents.

- **-3013**

Code: `o2dbE_OPEN_FILE (-3013)`
Call: `Connection@connect`
Cause: The configuration file cannot be opened.
Action: Check that the configuration file exists and check its access rights.

- **-3014**

Code: `o2dbE_NOT_SELECT (-3014)`
Call: `Context@define_projection, Context@fetch, Session@sqlquery`
Cause: The SQL statement is not a select statement.
Action: Check the syntax of the SQL statement.

- **-3015**

Code: `o2dbE_RANGE (-3015)`
Call: `Context@define_bind`
Cause: The order number is out of range.
Action: Check the order number.

- **-3016**

Code: `o2dbE_DEFINED (-3016)`
Call: `Context@define_projection, Context@define_bind`
Cause: The parameter or the result object is yet defined.
Action: Check the order number.

- **-3017**

Code: `o2dbE_MISMATCH (-3017)`
Call: `Context@define_projection, Context@define_bind, Session@sqlquery`
Cause: Type checking failed.
Action: Check the result or parameter type.

- **-3018**

Code: `o2dbE_NOT_SUPPORTED (-3018)`
Call: `Context@define_projection, Context@define_bind, Session@sqlquery`
Cause: One of the atomic types used is not supported in the current version of O₂DBAccess.
Action: Check the type of result object.

Possible Errors

- **-3021**
Code: `o2dbE_INVALID_NAME (-3021)`
Call: `Context@define_projection, Session@sqlquery`
Cause: One of the names is not an attribute name.
Action: Check the list of attribute names.
- **-3022**
Code: `o2dbE_NOT_EXECUTED (-3022)`
Call: `Context@fetch`
Cause: The statement hasn't been executed.
Action: Check the completion of the previous `exec` method call.
- **-3023**
Code: `o2dbE_NO_MORE_CONTEXTS (-3023)`
Call: `Context@open, Session@sqlquery`
Cause: An attempt was made to exceed the maximum number open contexts allowed.
Action: Close some contexts.
- **-3024**
Code: `o2dbE_NILREF (-3024)`
Call: `Context@define_projection, Context@define_bind, Context@exec, Context@fetch, Session@sqlquery`
Cause: The result object or a parameter is `nil`.
Action: Check the method call syntax.
- **-3025**
Code: `o2dbE_NOMEM (-3025)`
Call: All.
Cause: Not enough memory.
Action: Close some contexts.

- **-4001**

Code: `o2dbe_INTERNAL (-4001)`

Call: All.

Cause: Internal error. It should not normally be issued.

Action: Contact O₂Line.

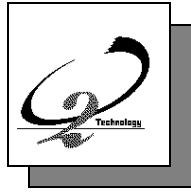
- **-5001**

Code: `o2dbw_NOT_UNIQUE (-5001)`

Call: `Context@fetch, Session@sqlquery`

Cause: This a warning. There is more than one row to fetch whereas the result type is not a collection.

Action: Nothing. Only the first row has been fetched.



INDEX



INDEX

A

Application
 Example [54–61](#)
 Information transfer [25](#)

Architecture
 Client/ server [13](#)
 O₂ [10](#)

associate
 Example [58](#)
 Method [24, 48](#)

B

Buffers
 Data [26](#)
 Definition [26](#)

C

C [11](#)

C++
 Interface [11](#)

c_name [62](#)

Class [18](#)
 Connection [22, 36](#)
 Context [47](#)
 O2DBAccess [34](#)
 Session [22, 41](#)

Client/ server architecture [13](#)

close
 Example [61](#)
 Method [23, 42](#)

commit
 Method [30, 43, 54](#)

Configuration
 File [62](#)

connect
 Example [56](#)
 Method [22, 37](#)

Connection
 Class [18, 36](#)
 Creation [22, 56](#)
 Import [55](#)
 Methods [22, 36–40](#)

Context
 Class [18, 23, 47](#)
 Close [23, 61](#)
 Import [55](#)
 Link to statement [24](#)
 Management [24](#)
 Maximum number [23](#)
 Methods [47](#)
 Open [23, 57](#)

D

Data
 Access [22](#)
 Buffers [26](#)
 Fetching [29, 60](#)
 Transfer [25](#)

Database
 Access [22–23](#)
 Information transfer [25](#)
 Log off [22, 61](#)
 Log on [22, 56](#)

define_bind
 Example [58](#)
 Method [24–27, 49](#)

define_projection
 Example [58](#)
 Method [24–26, 50](#)

INDEX

disconnect
 Example 61
 Method 22, 31, 38

E

Environment variable 62
Error codes 24, 63–68
Example application 54–61
exec
 Example 59
 Method 29, 51

F

fetch
 Example 60
 Method 29, 52
File
 Configuration 62

H

Host
 Connection 22, 56
 Disconnect 22, 31, 61
host 62

I

import schema 55
Internal error codes 63

J

Java 11

L

logoff
 Example 61
 Method 22, 31, 40
logon
 Example 56
 Method 22, 39
lu_name 62

M

Managing contexts 24
max_contexts 62



INDEX

Method [21](#)
 associate [24, 48](#)
 close [23, 42](#)
 commit [30, 43](#)
 connect [22, 37](#)
 define_bind [24–27, 49](#)
 define_projection [24–26, 50](#)
 disconnect [22, 31, 38](#)
 exec [29, 51](#)
 fetch [29, 52](#)
 logoff [22, 31, 40](#)
 logon [22, 39](#)
 open [23, 44](#)
 rollback [30, 45](#)
 server_error [35](#)
 sqlquery [46](#)

mode [62](#)

N

network [62](#)

O

O₂
 Architecture [10](#)
O₂C [11](#)
O₂Corba [11](#)
O2DBACCESS [62](#)
O2DBAccess
 Class [18, 34](#)
 Methods [35](#)
O2DBAccess [11](#)
o2dbaccess
 Schema [18, 55](#)
o2dbaccess.cf [62](#)
O₂Engine [10, 12](#)

O₂Graph [11](#)
O₂Kit [11](#)
O₂Look [11](#)
O₂ODBC [11](#)
O₂Store [10](#)
 Overview [13](#)
O₂Tools [11](#)
O₂Web [11](#)
open
 Example [57](#)
 Method [23, 44](#)
OQL [11](#)

P

Parameter
 Buffers [26](#)
 Class [18, 26](#)
 Import [55](#)
 Subclass [55](#)

R

rollback
 Method [30, 45, 54](#)

S

Schema
 Definition [18, 55](#)

INDEX

server_error
 Error codes [63](#)
 Method [24, 35](#)

service [62](#)

Session

 Class [18, 22–23, 41](#)

 Close [23, 31, 61](#)

 Example [57, 61](#)

 Import [55](#)

 Methods [41](#)

 Open [23, 57](#)

sqlquery

 Method [46](#)

Statement

 Link to context [24](#)

 Preparation [24, 58](#)

 Running [29, 59](#)

System

 Architecture [10](#)

T

Transferring data [25](#)

type_checking [62](#)



INDEX
