

Kentico uses ObjectQuery to provide an abstraction layer over the SQL database. Developers use ObjectQuery to retrieve data from the Kentico database. The main advantages of using ObjectQuery are:

- strongly typed and enumerable results
- independence on specific versions of SQL syntax
- security

ObjectQuery call example:

```
var query = UserInfoProvider.GetUsers();
```

The example query retrieves all users stored within Kentico.

The resulting data set contains Info objects. One instance of an Info object represents one row of data from the database. In this case, the result is a set of UserInfo objects. The result allows you to iterate through the records using a foreach statement.

Similarly to users, you can use ObjectQuery to retrieve any other type of object. To do that, use the correct InfoProvider class and the GetX method, where X is the object name. The following table provides an example of the naming conventions.

Object	Info class name	InfoProvider class name	ObjectQuery method name
A/B test	ABTestInfo	ABTestInfoProvider	GetABTests
CSS Stylesheet	CssStylesheetInfo	CssStylesheetInfoProvider	GetCssStylesheets
E-mail template	EmailTemplateInfo	EmailTemplateInfoProvider	GetEmailTemplates
Page template	PageTemplateInfo	PageTemplateInfoProvider	GetPageTemplates
User	UserInfo	UserInfoProvider	GetUsers

Selecting only specific columns

ObjectQuery allows you to select only those columns that you need to work with. Selecting only the required columns is a good practice that prevents issues with performance of your code by transferring only the necessary amount of data from the database server to the application server.

```
var columnsQuery = UserInfoProvider.GetUsers()  
    .Columns("FirstName", "LastName", "UserName");
```

This example query retrieves all users but limits the data to the users' first names, last names, and usernames.

Limiting the retrieved results (SQL WHERE)

ObjectQuery provides a way to narrow down the resulting data using SQL syntax independent where conditions.

Use predefined where condition methods that provide optimized performance and make your code easily readable. The predefined where condition methods include, but are not limited to:

- WhereEquals("ColumnName", value) - checks the equality of the value in the specified column with the value specified in the second parameter.
- WhereGreaterThan("ColumnName", value) - compares the value in the specified column with the second parameter.
- WhereNull("ColumnName") - select only rows where the specified column has a NULL value.
- WhereNot(whereCondition) - negates the specified where condition.
- WhereStartsWith("ColumnName", "Value") - only works for text columns. Selects only rows whose value in the specified column starts with the given value.

You can find the full list of predefined where conditions in the [Kentico API Reference](https://docs.xperience.io).



```
// Retrieves all users whose first name is "Joe"
var whereQuery = UserInfoProvider.GetUsers().WhereEquals("FirstName", "Joe");
```



Tip: Use Visual Studio's IntelliSense to help you pick the right predefined where condition.

Logical operators

ObjectQuery allows you to join together multiple where conditions.

- By default, the conditions are combined using an AND parameter.
- To place an OR operator between conditions, call the **Or()** method between the given *Where* methods.
- You can also explicitly call the **And()** method if you wish to make the code of your conditions clearer and easier to read.

Example 1:

```
// Retrieves users whose first name is "Joe" or last name is "Smith"
var whereQuery = UserInfoProvider.GetUsers()
    .Where("FirstName", QueryOperator.Equals, "Joe")
    .Or()
    .Where("LastName", QueryOperator.Equals, "Smith");
```

Generated condition

```
DECLARE @FirstName nvarchar(max) = N'Joe';
DECLARE @LastName nvarchar(max) = N'Smith';

...

WHERE [FirstName] = @FirstName OR [LastName] = @LastName
```

Example 2:

```
// Retrieves all enabled users whose first name is "Joe" and email address ends with
"localhost.local"
var whereQuery = UserInfoProvider.GetUsers()
    .WhereEquals("FirstName", "Joe")
    .WhereEquals("UserEnabled", 1)
    .WhereEndsWith("Email", "localhost.local");
```

Generated condition

```
DECLARE @FirstName nvarchar(max) = N'John';
DECLARE @UserEnabled int = 1;
DECLARE @Email nvarchar(max) = N'%localhost.local';

...

WHERE [FirstName] = @FirstName AND [UserEnabled] = @UserEnabled AND [Email] LIKE @Email
```

Nested where conditions

If you need to construct complex where conditions with multiple AND/OR operators, use nested where conditions:

Note: You must use the **new** keyword to construct nested where conditions.

```
// Retrieves users named "Smith", whose first name is "Joe" or "John"
var nestedWhereQuery = UserInfoProvider.GetUsers()
    .WhereEquals("LastName", "Smith")
    .Where(new WhereCondition()
        .WhereEquals("FirstName", "Joe")
        .Or()
        .WhereEquals("FirstName", "John")
    );
```

Generated condition

```
DECLARE @LastName nvarchar(max) = N'Smith';
DECLARE @FirstName nvarchar(max) = N'Joe';
DECLARE @FirstName1 nvarchar(max) = N'John';

...

WHERE [LastName] = @LastName AND ([FirstName] = @FirstName OR [FirstName] =
@FirstName1)
```

Retrieving data assigned to a particular site

Many Kentico objects that are stored in the database are related to a particular site. ObjectQuery allows you to filter results by site by providing a site ID or code name.

```
// Retrieves all users assigned to the current site
var siteQuery = UserInfoProvider.GetUsers()
    .OnSite(SiteContext.CurrentSiteID);
```

```
// Retrieves all users assigned to the Corporate site
var siteQuery = UserInfoProvider.GetUsers()
    .OnSite("CorporateSite");
```

Ordering query results (SQL ORDER BY)

ObjectQuery can also sort data. Use `.OrderBy`, `.OrderByAscending`, or `.OrderByDescending`.

```
var orderedQuery = UserInfoProvider.GetUsers()
    .WhereGreaterThan("UserCreated", DateTime.Now.AddDays(-7))
    .OrderByDescending("UserCreated");
```

This example retrieves users who registered or were created in the system during the past week. The results are in descending order.

Limiting the number of query results (SQL TOP N)

ObjectQuery can also retrieve only a specified number of results.

```
var topNQuery = ForumPostInfoProvider.GetForumPosts()  
    .TopN(5)  
    .OrderByDescending("PostTime");
```

This example retrieves the latest 5 forum posts with the newest post on top.


Joining multiple tables (SQL JOIN)

With ObjectQuery, you can combine data from two or more tables based on a common field.

```
var joinQuery = UserInfoProvider.GetUsers()  
    .Source(sourceItem => sourceItem.Join<UserSettingsInfo>("UserID",  
"UserSettingsUserID"))  
    .WhereEmpty("UserPassword");
```

This example retrieves users, who have not set a password, i.e., whose password is blank.


The code on the second line appends CMS_UserSettings table rows to the appropriate user records based on their ID. The code on the third line then selects only rows where the UserPassword is empty.

 By default, the *Join<TInfo>* method performs an Inner join. For other types of joins, you can call the *LeftJoin<TInfo>* or *RightJoin<TInfo>* methods. See [SQL Joins](#) for more information.

For advanced scenarios, you can use the non-generic version of the *Join* method. In this case, you need to identify the target table via a *QuerySourceTable* object (requires the exact database table name, optionally allows you to specify an alias and [SQL Hints](#)).

Working with joined data

If you need to work with data containing columns from multiple tables, convert the result of the query to a *DataSet*. The default ObjectQuery result is strongly typed, so only allows you to access the data of a single Kentico *Info* class. Use the **Result** property of ObjectQuery to get the results as a *DataSet*.

 **Limitation:** ObjectQuery joins cannot be used for tables located in different databases. For example, if you have a [separated on-line marketing database](#), you cannot join data from a table in the main database and another table in the separated database. In these cases, use separate ObjectQuery calls and then combine the resulting data manually.

Joining 3 or more tables

You can chain together multiple *Join* methods to combine rows from 3 or more database tables.

Note: When you call the *Join* method for a query source, the *left column* parameter works with columns within the given query source table by default. When using multiple joins, we recommend that you explicitly specify both the table and column by adding a table name prefix to the left column name, for example: *"CMS_User.UserID"*

The following example joins together rows from 3 tables based on shared user IDs and returns the result as a *DataSet*.

```
using System.Data;
using CMS.Membership;
using CMS.Taxonomy;

...

DataSet data = UserInfoProvider.GetUsers()
    .Source(sourceItem => sourceItem.Join<UserSettingsInfo>("CMS_User.UserID",
"UserSettingsUserID")

    .Join<CategoryInfo>("CMS_User.UserID", "CategoryUserID")
        )
    .Columns("UserID, Username, UserURLReferrer, CategoryDisplayName")
    .Result;
```

Working with ObjectQuery results

ObjectQuery results are [lazy-loaded](#). Lazy loading increases performance by executing queries only when you need to work with their results. Most of the examples on this page only constructed the proper SQL queries. The SQL queries are executed when you either enumerate the results using the foreach statement, or cast the query as a different type (for example a DataSet).

The system caches the loaded data, so if you work with the results after the query executed, the system will not perform any additional requests to the database.

The most common example of working with query results is iterating through them. Use a standard foreach loop.

```
var userQuery = UserInfoProvider.GetUsers();

foreach (UserInfo user in userQuery)
{
    // Do something with the UserInfo object
}
```



Tip: You can access the results of ObjectQuery calls in other data formats through the following properties:

- **Result** - returns a *DataSet* containing the query results. Recommended if you need to bind the data as a data source, and for data containing columns from multiple tables (joins and unions).
- **TypedResult** - returns a strongly typed collection of the query results (*InfoDataSet<TInfo>*).



LINQ support

ObjectQuery provides limited support for [LINQ](#).

Currently, we only recommend using LINQ for simple expressions, such as comparisons with constants. Any other expressions can lead to sub-optimal query execution. Use the ObjectQuery API directly to get optimal performance.

When using LINQ to process *ObjectQuery* results, always check the resulting queries using the [Kentico SQL query debug](#).

Reusing queries (mutability)

ObjectQuery instances are mutable by default. This means that any changes (method calls) made to existing instances also modify the original query.

If you wish to reuse an ObjectQuery instance without modifying the original, use one of the following approaches:



- Call the **Immutable** method for the original query. This ensures automatic cloning of the ObjectQuery instance when it is modified.
- If you want to leave a query mutable, you can call the **Clone** method to create a new instance which can then be modified without affecting the original.

Example

```
using System.Linq;
using CMS.Membership;

...

// Creates an immutable query for loading users ordered by their full name
var userQuery = UserInfoProvider.GetUsers()
    .OrderBy("FullName")
    .Immutable();

// Gets a subset of users based on the original query
var enabledUserList = userQuery
    .WhereEquals("UserEnabled", 1)
    .ToList();

// Gets a list of users using the original unmodified query
var userList = userQuery.ToList();
```