ObjectBox Go

ObjectBox Go

This is the ObjectBox documentation for our Go API. We strive to provide you with the easiest and fastest solution to store and retrieve data.

Your feedback on ObjectBox and this documentation is very welcome. Use the "Was this page helpful?" smiley at the end of each page or send us your comments to contact[at]objectbox.io - thank you! :)

Changelog

v1.1.2 (2020-03-18)

- ensure Query finalizer is only executed by Go GC after a native call finishes

v1.1.1 (2020-02-14)

- use temp directories in tests to prevent failure in recent Go versions checking-out modules as read-only

v1.1.0 (2019-12-16)

- add Box `Insert` and `Update` methods with stricter semantics than `Put`
- add AsyncBox with `Put` , `Insert` , `Update` , `Remove`
- add Query order and parameter alias support - see Queries docs for more info
- Code generator improvements
  - handle type-checker errors more gracefully (don't fail on failures in unneeded imports)
  - add `clean` command-line option to remove all generated files
  - `time.Time` will automatically use a built in converter to Unix timestamp (milliseconds)
  - improve model.json file diff-level compatibility with other language bindings
  - embedded struct and to-one relations cycle detection

    - support changing property type and resetting its stored value
    - nil check for embedded pointer structs in the generated code
    - minor bug fixes when the generated code wouldn't compile in some edge cases
- update to the latest ObjectBox-C library v0.8.1
- deprecate `box.PutAsync()` in favor of `box.async().Put()` i.e. using AsyncBox
- mark `byte` properties as unsigned
- fix getters on objects with missing relations and non-existent IDs in `GetMany`
- better windows installation experience using a PowerShell script
- make `golint` happier :)

v1.0.0 (2019-07-16)

This is quite a big release, bringing some new features and cleaning up the API.

- explicit transaction support via `ObjectBox::RunInReadTx` and `ObjectBox::RunInWriteTx`
- Go Modules support
- add `objectbox` namespace to tags to align with `reflect.StructTag.Get` unofficial spec
- optional lazy loading on to-many relations - `lazy` annotation
- box additions:
    - `GetMany` , `RemoveMany` , `RemoveIds` , `ContainsIds` , `RemoveIds`
    - to-many relation auxiliary methods: `RelationIds` , `RelationPut` , `RelationRemove` , `RelationReplace`
- switch default/recommended `go:generate` entity generator command from `objectbox-gogen` to `//go:generate go run github.com/objectbox/objectbox-go/cmd/objectbox-gogen`
- quite a few internal changes, renames and other refactorings (e.g. renamed `PutAll` to `PutMany` , removed `Cursor` , aligned model JSON with other bindings, ...)

v0.9.0 (2019-04-24)

- Fixed macOS build and 32-bit query support.
- Minor refactoring/linter issues

v0.9.0-rc (2019-02-22)

As we queued up quite a few changes, we're doing a release candidate first for you to test:

- Improved relations support
- Embedded structs
- Custom value-converter to store unsupported types & structs that can't be inlined/prefixed
- Recognize and handle type aliases and named types used as entity fields
- New Box methods: CountMax(), IsEmpty(), Contains() and PutAsyncWithTimeout()
- Query support AND & OR (combine conditions)
- New Query methods: Limit, Offset
- New Query methods: Set*Params (type-based) - to run a cached query with custom parameters
- Query LT|GT|Between support for unsigned numbers
- Support numeric string ID in the entity
- Support for `[]string` as a field type
- Optional pass/return-by-value for slice-of-structs in the generated code
- Change strings to use hash-based instead of value-based indexes by default
- A new option to AlwaysAwaitAsync that can be enabled during initialization

v0.8.0 (2018-12-06)

- New Query API
- Box and Query do not require manual closing anymore
- Support for renaming entities and their properties using UIDs

v0.7.1 (2018-11-30)

- Fixed wrong mapping for Go types (u)int and (u)int8. Luckily we noticed this very early: if you used those types in previous versions, please delete old database files.
- Transactions are now safely aborted in case of panics

v0.7.0 (2018-11-29)

- Changed file name of generated code, e.g. file endings for model is now ".obx.go"

- Foundation for all query conditions (final query API will come with the next version)
- Put(object) now assigns the new id to the object itself

v0.6.0 (2018-11-28)

Initial public release

Installation

We're trying to make the installation experience smooth for everyone. In case you're getting stuck or are finding some steps hard to follow, please reach out to us, e.g. by creating a GitHub issue or through our Contact form. Thanks!

Linux/macOS

This section describe the installation on Linux or macOS, if you're using Windows, please skip to the Installation on Windows section.

The main prerequisite to using ObjectBox in Go is the ObjectBox binary library (.so, .dylib depending on your platform) which actually implements the database functionality.

We are using CGO which requires you to have a C/C++ compiler, such as `gcc` or `clang`, installed. You can try executing one of following commands in terminal to check if it's already available and working: `$CC --version` or `gcc --version` or `clang --version`. If any of the commands works fine (no need for all of them to work), you should be good to go. Otherwise, please `gcc` or `clang` according to the instructions for your system (e.g. `sudo apt install gcc` on Ubuntu).

> There's currently a known issue on some ARM platforms, see Raspberry Pi 3 & 4 fallback.

Quick installation

The fastest way to install is by using our installation script. Execute the following command in your project directory. If that doesn't work for you, you can skip to the manual installation bellow.

```
bash <(curl -s https://raw.githubusercontent.com/objectbox/objectbox-go/mast
```

→ Getting started /getting-started

Manual installation

C binary library

You can run the following `download.sh` script (press Y to install the library to a system-wide folder when it asks you). you can remove the temporary "objectboxlib" directory created by this step afterwards.

```
mkdir objectboxlib && cd objectboxlib
bash <(curl -s https://raw.githubusercontent.com/objectbox/objectbox-c/mas
```

Go package dependency

Go modules based project

If your project is using a `go.mod` file to keep track of the dependencies, you don't need to install anything else and you can just start using ObjectBox by importing it in your source code:

```
import "github.com/objectbox/objectbox-go/objectbox"
```

Legacy projects without a go.mod file

In case you're not using Go modules, you can install ObjectBox using following commands:

```
go get -u github.com/objectbox/objectbox-go/...
go get -u github.com/google/flatbuffers/go
```

A package dependency is set up automatically if you've used the "Quick installation" method using the `install.sh` script

→ Getting started /getting-started

Raspberry Pi 3 & 4 fallback

You may encounter an issue on some ARM platforms (seen this on Raspberry Pi 3 & 4) with the default native library installed by the script. Also, the issue seems to only occur when running natively, not inside docker.

You can check your installation to see if you encounter crashes (SIGBUS/SIGSEGV) by executing `go test github.com/objectbox/objectbox-go/...`

As a workaround, you can install an ARMv6 version of the native library (instead of the default ARMv7 the script picks) using the same script as in the manual installation, just changing the arguments:

```
./download.sh 0.8.1 testing Linux armv6
```

Windows

The main prerequisite to using ObjectBox in Go is the ObjectBox binary DLL which actually implements the database functionality. We are using CGO which requires you to have MinGW `gcc` in PATH, e. g. http://tdm-gcc.tdragon.net/ - in case you don't have MinGW installed yet, please do so first.

Quick installation

You can use the following PowerShell script to help you with installation of the library to the right folders:

- download the script: (right click and "Save link as", the location doesn't matter) install.ps1
- run the script - either double click or right click and "Run in PowerShell" - depends on your settings.
- the script will guide you through the installation steps.

Manual installation

In case you couldn't install fully using the script (e.g. it didn't find your MinGW installation), you can try to finish the installation manually

1. In order to compile your program, you need to copy the downloaded `download/bjectbox.dll` to the MinGW library directory, e. g. `C:\TDM-GCC-64\lib`
2. To run the program, you either need to have the `objectbox.dll` library in the same folder as the compiled program, have its location (e. g. `C:\TDM-GCC-64\lib` in `Path` environment variable), or copy it to the system library directory `c:\Windows\System32`

Go package dependency

Go modules based project

If your project is using a `go.mod` file to keep track of the dependencies, you don't need to install anything else and you can just start using ObjectBox by importing it in your source code:

```
import "github.com/objectbox/objectbox-go/objectbox"
```

→ Getting started /getting-started

Distributing

Don't forget to include the `objectbox.dll` with your program when distributing/packaging for installer. Having it in the same directory as your program binary should be enough.

Getting started

Creating an Entity

Using ObjectBox in your project is fairly straight-forward.

First of all, let's define an Entity which is just a `struct`. It could be located basically anywhere but to keep our project structure clean, let's have it in a `internal/model/task.go` i. e. `model` package.

internal/model/task.go

```go
package model

//go:generate go run github.com/objectbox/objectbox-go/cmd/objectbox-gogen

type Task struct {
  Id          uint64
  Text        string
  DateCreated  int64
  DateFinished int64
}
```

Note that `Id uint64` is recognized by ObjectBox to contain an ID field which gives us a direct access to the stored Tasks objects by their ID.

Alternatively if your ID field is named differently, you can annotate it with `objectbox:"id"`.
For more information see Entity Annotations.

Having the entity file, we can run bindings generator to get the necessary `task.obx.go`

```
cd my-project-dir
go generate ./...
```

The generated bindings code has two main competencies:

- provide Entity model (schema) to the ObjectBox
- convert between internal object representation (FlatBuffers) and our struct `Task`

> Additionally to the `task.obx.go` there's an `objectbox-model.json` which holds information about the model and `objectbox-model.go` which defines a model initialization function. All these files should be committed in source control along with the rest of your code.

Initializing ObjectBox

To use ObjectBox in our main application code, we use `Builder` and give it model information and optionally some other settings. The code with the model information is generated by the ObjectBox and you just need to use in your code:

```go
import (
  "github.com/objectbox/objectbox-go/objectbox"
  "github.com/objectbox/objectbox-go/examples/tasks/internal/model"
)

func initObjectBox() *objectbox.ObjectBox {
  objectBox, err := objectbox.NewBuilder().Model(model.ObjectBoxModel()).B
  return objectBox
}
```

Working with Object Boxes

Bet you wondered where our name comes from :)

From ObjectBox you vend Box instances to manage your entities. While you can have multiple Box instances of the same type (for the same Entity) "open" at once, it's usually preferable to just use one instance and pass it around your code.

main.go

```go
func main() {
   // load objectbox
   ob := initObjectBox()
   defer ob.Close() // In a server app, you would just keep ob and close o

   box := model.BoxForTask(ob)

   // Create
   id, _ := box.Put(&model.Task{
      Text: "Buy milk",
   })

   task, _ := box.Get(id) // Read
   task.Text += " & some bread"
   box.Put(task)         // Update
   box.Remove(task)      // Delete
}
```

Wherever you have access to a Box, you can use it to persist objects and fetch objects from disk. **Boxes are thread safe.** Here are some of the basic operations:

- **Put:** Persist an object, which may overwrite an existing object with the same ID. In other words, use `Put` to insert or update objects. When put succeeds, an ID will be assigned to the entity.
- **Get:** When you have an object's ID, you can get to the object very efficiently using `Get`. It will return `nil` & error in case an error occurred and `nil` without any error if the object doesn't exist.
  To get all objects of a type, use `GetAll` which returns a slice.
- **Remove:** Deletes a previously persisted object from its box. Use `RemoveAll` to delete all objects and empty the box.
- **Count:** The number of objects stored in this box.

Task-list example application

To see it all put together, have a look at the Task-List application example in our git repository:
https://github.com/objectbox/objectbox-go/tree/master/examples

Transactions

Basics

ObjectBox is a fully transactional database satisfying ACID properties. A transaction can group several operations into a single unit of work that either executes completely or not at all. If you are looking for a more detailed introduction to transactions in general, please consult other resources like Wikipedia on database transactions.

You may not notice it, but almost all interactions with ObjectBox involve transactions. For example, if you call `box.Put()` a write transaction is used. Also if you `box.Get()` an object or query for objects, a read transaction is used. All of this is done under the hood and transparent to you. It may be fine to completely ignore transactions altogether in your app without running into any problems. With more complex apps however, it’s usually worth learning transaction basics to make your app more consistent and efficient.

Explicit transactions

All ObjectBox operations run in implicit transactions – unless an explicit transaction is in progress. In the latter case, multiple operations share the (explicit) transaction. In other words, with explicit transactions you control the transaction boundary. Doing so can greatly improve efficiency and consistency in your app.

The `ObjectBox::RunInReadTx()` and `ObjectBox::RunInWriteTx()` take a function as and argument and run it inside a transaction (read-only or a write transaction, respectively).

There are multiple advantages of explicit transactions:

- you can perform any number of operations and use objects of multiple boxes, while having a consistent view of the data,
- running multiple updates/inserts is faster because it doesn't involve starting an implicit transaction each time,
- being able to "roll-back" a transaction when an error occurs, potentially discarding changes from multiple updates.

Example for a write transaction which just inserts 1 000 000 objects:

```go
ob.RunInWriteTx(func() error {
  for i := 1000000; i > 0; i-- {
    box.Put(&iot.Event{})
  }
  return nil // return no error so the transaction is not rolled back
})
```

Understanding transactions is essential to mastering the database performance. If you just remember one sentence on this topic, it should be this one: a write transaction has its price, and it's the same whether it's implicit or explicit.

Committing a transaction involves syncing data to the physical storage, which is a relatively expensive operation for databases. Only when the file system confirms that all data has been stored in a durable manner (not just memory cached), the transaction can be considered successful. This file sync required by a transaction may take a couple of milliseconds. Keep this in mind and try to group several operations (e.g. `Put` calls) in one transaction.

Read Transactions

In ObjectBox, read transactions are cheap. In contrast to write transactions, there is no commit and thus no expensive sync to the file system. Operations like `Get` , `Count` , and queries run inside an implicit read transaction if they are not called when already inside an explicit transaction (read or write). Note that it is illegal to `Put` when inside a read transaction.

While read transaction are much cheaper than write transactions, there is still some overhead to start a read transaction. Thus, for a high number of reads (e.g. hundreds, in a loop), you can improve performance by grouping those reads in a single read transaction (see explicit transactions below).

Multiversion concurrency

ObjectBox gives developers Multiversion concurrency control (MVCC) semantics. This allows multiple concurrent readers (read transactions) which can execute immediately without blocking or waiting. This is guaranteed by storing multiple versions of (committed) data. Even if a write transaction is in progress, a read transaction can read the last consistent state immediately. Write transactions are executed sequentially to ensure a consistent state. Thus, it is advised to keep write transactions short to avoid blocking other pending write transactions. For example, it is usually a bad idea to do networking or complex calculations while inside a write transaction. Instead, do any expensive operation and prepare objects before entering a write transaction.

Note that you do not have to worry about making write transactions sequential yourself. If multiple threads want to write at the same time (e.g. via `Box::Put` or `ObjectBox::RunInWriteTx`), one of the treads will be selected to go first, while the other threads have to wait. It works just like a `mutex.Lock()`

Locking inside a Write Transaction

⚠ Avoid locking (e.g. via `mutex.Lock()`) when inside a write transaction when possible.

Because write transactions run exclusively, they effectively acquire a write lock internally. As with all locks, you need to pay close attention when multiple locks are involved. Always obtain locks in the same order to avoid deadlocks. If you acquire a lock “X” inside a transaction, you must ensure that your code does not start another write transaction while having the lock “X”.

Entity Annotations

ObjectBox - Database Persistence with Entity Annotations

ObjectBox is a database that persists objects. For a clear distinction, we sometimes call those persistable objects **entities**. To let ObjectBox know which structs are entities you add go:generate command to their source file and annotations (Go tags) to some fields. Then ObjectBox can do its magic with your entities.

Here is an example:

```go
//go:generate go run github.com/objectbox/objectbox-go/cmd/objectbox-gogen

type Task struct {
  Id          uint64
  Text        string
  DateCreated  int64
  DateFinished int64
}
```

When you run `go generate ./...` in your project, it finds all files with a `//go:generate` comment and executes the program in that comment, `objectbox-gogen` . This means you can include multiple entities (structs) inside a single file without repeating the comment. Having multiple entities in a single file may help increase generation performance on large projects because the generator can cache information about referenced types during runtime.

---

id - Object IDs

In ObjectBox, every object has an ID of type `uint64` to efficiently get or reference objects. ObjectBox recognizes this automatically if your entity has a `uint64` field named ID (case insensitive).

```go
type Task struct {
  Id uint64
}
```

Alternatively, you can use the `id` annotation on a `uint64` property with any name in your entity:

```go
type Group struct {
  GroupID uint64 `objectbox:"id"`
}
```

Note that in case this Id is zero on an instance you are inserting, ObjectBox considers the object as a new one during `Put()` and automatically assigns an ID.

If your application requires other ID types (such as a string UID given by a server), you can model them as standard properties. An example:

```go
type Task struct {
  Id  uint64 `objectbox:"id"`
  Uid string
}
```

---

index - property Indexes

Annotate a property with `objectbox:"index"` to create a database index for the corresponding database column. This can improve performance when querying for that property.

```go
type Task struct {
  Uid string `objectbox:"index"`
}
```

Indexing is currently not supported for `byte[]` , `float` and `double`

Index types (String)

ObjectBox can use either actual **value** of the property or its **hash** to build an index. Because `string` properties are typically taking more space than scalar values, ObjectBox is using hash for strings by default.

You can instruct ObjectBox to use a different index type :

```go
type Task struct {
  Uid string `objectbox:"index:hash64"`
}
```

ObjectBox supports these index types:

- **Not specified** Uses best index based on property type (uses `hash` for `string` , `value` for others).
- **"value"** Uses property values to build index. For `String,` this may require more storage than a hash-based index.
- **"hash"** Uses 32-bit hash of property values to build index. Occasional collisions may occur which should not have any performance impact in practice. Usually a better choice than `hash64` , as it requires less storage.
- **"hash64"** Uses long hash of property values to build the index. Requires more storage than `hash` and thus should not be the first choice in most cases.

Limits of hash-based indexes: Hashes work great for equality checks, but not for "**starts with**" type conditions. If you frequently use those, you should use value-based indexes instead.

unique - unique constraints

Annotate a property with `objectbox"unique"` to enforce that values are unique before an entity is inserted/updated:

```go
type Task struct {
  Uid string `objectbox:"unique"`
}
```

A `put()` operation will abort and return an error if the unique constraint is violated.

Complex fields (structs)

When your field contains another struct that is not a Relation, ObjectBox will, by default, store it embedded, using prefix-based field naming. The `Task` in the following example will internally end up with four fields, "id", "meta_created", "meta_modified" and "text".

```go
// NOTE this is be placed in a separate field because it's not an entity
type Metadata struct {
  Created  int64
  Modified int64
}

// this is an Entity, with code genereated using objectbox-gogen
type Task struct {
    Id   uint64
    Meta Metadata
  Text string
}
```

inline

You can modify the behavior by specifying the struct fields to be "inlined" in which case, the fields would be named "id", "created", "modified" and "text".

```
type Task struct {
    Id   uint64
    Meta Metadata `objectbox:"inline"`
  Text string
}
```

It's important to keep distinction between prefixed and inlined fields in mind if you want to move fields around between two embedded/included structs or rename them.

Other annotations

```
// `objectbox:"uid:1306759095002958910"`
type Task struct {
  Text  string `objectbox:"name:text"`
  Date  uint64 `objectbox:"date"`
  notes string `objectbox:"-"`
  DateCreated  int64 `objectbox:"date uid:7144924247938981575"`
}
```

converter

Defines converter for custom types, see Custom types docs for more information.

date

Informs ObjectBox that it should store the given property as a DateType - it expects timestamp since UNIX epoch in milliseconds.

If you use a `time.Time` field, it's automatically recognized as a date and the code generator will use the built-in converter and store the field internally as a Unix timestamp.

> Because ObjectBox stores dates internally as Unix timestamps with **millisecond** precision, the built-in converter falls-back to that precision when working with `time.Time` struct.
>
> If you require greater precision, define your own converter with any built-in supported type that can accommodate your storage format (e.g. `int`, `string`, `[]byte`, etc.).

lazy

Specifies that the "To-Many" relation on the current field should not be called right away when the object is read but manually, using GetRelated. See Relations docs for more details.

link

Declares the struct field that is itself a struct (or a pointer to one) as a relation, instructing ObjectBox to create a link between the Entity where this field is contained and the Entity of the field (type of the struct field). See Relations docs for more details on how relations work and how you can define them.

name

Lets you define under what name the property is stored in the database. This allows you to rename the Go field without affecting the property name on the database level. To rename the property in the DB, you should use the uid annotation instead.

type

Used in conjunction with converter to specify the underlying type stored in the database (type returned by the converter). See Custom types docs for more information.

"-"

Marks properties that should not be persisted (saved into DB) and are only used in your program during runtime.

Triggering generation

Once your entity schema is in place, you can trigger the code generation by running `go generate` inside the directory that contains the files with the entities.

uid

ObjectBox keeps track of entities and properties by assigning them unique IDs (UIDs) during the code-generation phase. All those UIDs are stored in a file `objectbox-model.json` in your package, which you should add to your version control system (e.g. git).

If you specify the `objectbox"uid:....."` tag on a property (or as a special comment on an entity struct), ObjectBox would be able to uniquely identify it even after you change the name and would update the database accordingly on the next application launch.

For more information, look at the following page dedicated to schema updates.

→ Schema changes /schema-changes

Additionally, If you are interested, we have in-depth documentation on UIDs and concepts in the Java/Android docs.

Queries

Using queries is simple: from your entity's `Box`, call `Query()` with conditions as arguments:

```
query := box.Query(Device_.Location.HasPrefix("US-", false))
devices, err := query.Find()
```

Building queries

The query in the code above uses a function `HasPrefix` on a device location. Where does this come from? ObjectBox generates a `Device_` struct for you to reference available properties conveniently. This also allows code completion in your IDE and avoids typos: correctness is checked at compile time (string based queries would only be checked at run-time).

Let's say you have the following entity defined in your package:

```
type Device struct {
  Id       uint64
  Name     string
  Location string
  Profile  uint32
}
```

Using this input, the ObjectBox code generator creates a variable `Device_` in the same package:

```
var Device_ = struct {
  Id       *objectbox.PropertyUint64
  Name     *objectbox.PropertyString
  Location *objectbox.PropertyString
  Profile  *objectbox.PropertyUint32
}{...}
```

You can use `Device_` to construct type-specific conditions in place and combining them, forming the full query. The following example looks for devices located in the U. S. with profile number 42.

```
box.Query(Device_.Profile.Equals(42), Device_.Location.HasPrefix("US-", fals
```

Reusing Queries and Parameters

If you frequently run a `Query` you should cache the `Query` object and re-use it. To make a `Query` more reusable you can change the values, or query parameters, of each condition you added even after the `Query` is built. Let's see how.

Assume we want to find a list of `User` with specific `FirstName` values. First, we build a regular `Query` with an `equal()` condition for `FirstName`. Because we have to pass an initial parameter value to `equal()` but plan to override it before running the `Query` later, we just pass an empty string:

```
var caseSensitive = false
var query = box.Query(User_.FirstName.Equals("", caseSensitive))
```

Now at some later point we want to actually run the `Query`. To set a value for the `FirstName` parameter we call `setStringParams()` on the `Query` and pass the `FirstName` property and the new parameter value:

```
query.SetStringParams(User_.FirstName, "Joe")
joes, _ := query.Find()
```

Alias/As

So you might already be wondering, what happens if you have more than one condition using the same property? For this purpose you can **assign each condition an alias** by calling `Alias()` right after specifying the condition:

```
var query = box.Query(
    User_.Age.GreaterThan().Alias("min age"),
    User_.Age.LessThan().Alias("max age"))

// Then use the alias when setting the parameter value
query.SetInt64Params(objectbox.Alias("min age"), 50)
query.SetInt64Params(objectbox.Alias("max age"), 100)
```

There's also an alternative, syntax for a aliases that makes it easier to maintain the code because it avoids repeating string constants:

```
var minAgeAlias = objectbox.Alias("min age")
var maxAgeAlias = objectbox.Alias("max age")
var query = box.Query(
    User_.Age.GreaterThan().As(minAgeAlias),
    User_.Age.LessThan().As(maxAgeAlias))

// Then use the alias when setting the parameter value
query.SetInt64Params(minAgeAlias, 50)
query.SetInt64Params(maxAgeAlias, 100)
```

Limit, Offset, and Pagination

Sometimes you only need a subset of a query, for example the first 10 elements. This is especially helpful (and resourceful) when you have a high number of entities and you cannot limit the result using query conditions only. The built `Query` has `.Offset()` and `.Limit()` methods to help you do that

```
query := box.Query(User_.FirstName.Equals("Joe", false))
joes, err := query.Offset(10).Limit(5).Find()
```

`Offset(n uint64):` the first `n` results are skipped.
`Limit(n uint64):` at most `n` results of this query are returned.

Ordering results

In addition to specifying conditions you can order the returned results:

```
query := box.Query(User_.FirstName.Equals("Joe", false), User_.Age.OrderDe
joes, err := query.Find()
```

You can combine multiple order parameters and options (some options are only available for certain data types, e.g. strings have case-sensitive ordering option), such as:

```
query := box.Query(
    User_.FirstName.Equals("Joe", false),
    User_.LastName.OrderDesc(false), // caseSensitive bool argument
    User_.Age.OrderAsc()
)
joes, err := query.Find()
```

Notable conditions/operators

In addition to expected conditions like `Equals()` , `NotEquals()` , `GreaterThan()` and `LessThan()` there are also conditions like:

- `Between()` to filter for values that are between the given two (inclusive)
- `In()` and `NotIn()` to filter for values that match any in the given set,
- `HasPrefix()` , `HasSuffix()` and `Contains()` for extended String filtering.

Working with query results

You have a few options how to handle the results of a query:

- `Find()` returns a slice of the matching objects,
- `FindIds()` fetches just the IDs of the matching objects as a slice, which can be more efficient in case you don't need the whole object,
- `Remove()` deletes all the matching objects from the database (in a single transaction),
- `Count()` gives you the number of the objects that match the query,
- `Limit()` and `Offset()` let you select just part of the result (e. g. for paging)
- `DescribeParams()` is a utility function which returns a human-readable representation of the query.

Querying linked objects (relations)

After creating a relation between entities, you might want to add a query condition for a property that only exists in the related entity. In SQL this is solved using JOINs. ObjectBox provides query links instead.
Let's see how this works using an example.

Assume there is a `Person` that can be associated with multiple `Address` entities:

```go
//go:generate go run github.com/objectbox/objectbox-go/cmd/objectbox-gogen

type Person struct {
  Id        uint64
  Name      string
  Address   []*Address
}

type Address struct {
  Id      uint64
  Street string
  ZIP    string
}
```

To get a `Person` with a certain name that also lives on a specific street, we need to query the associated `Address` entities of a `Person`. To do this, use the `Person_.Address.Link(cs ...Conditions)` method of the generated `Person_` variable to tell that the `addresses` relation should be queried and what conditions should be used to filter the addresses:

```
// get all Person objects named "Elmo" which have an address on "Sesame St
var query = BoxForPerson(ob).Query(
  Person_.name.Equals("Elmo", true),
  Person_.Address.Link(Address_.Street.Equals("Sesame Street", true)),
)
var elmosOnSesameStreet = query.Find()
```

What if we want to get a list of `Address` instead of `Person`? No problem, links are smart enough to know there's also an implicit relation in the opposite direction. Note the different `box` we're using here:

```
// get all Address objects on "Sesame Street" linked from a Person named "
val builder = box.query().equal(Address_.street, "Sesame Street")
var query = BoxForAddress(ob).Query(
    Address_.Street.Equals("Sesame Street", true),
  Person_.Address.Link(Person_.name.Equals("Elmo", true)),
)
var addressesSesameStreetWithElmo = query.Find()
```

More to come

ObjectBox core can do much more with the queries, such as property queries, aliases, etc. These are not yet supported by our Go API, but you can take a peek at https://docs.objectbox.io/queries to get the idea what's coming in the future releases.

Feel free to open a feature request on GitHub if you have an idea or a proposal.

Schema changes

ObjectBox manages its data model (schema) mostly automatically. The data model is defined by the entity structs you define. When you **add or remove** entities or properties of your entities, **ObjectBox takes care** of those changes without any further action from you.

For other changes like **renaming or changing the type**, ObjectBox needs **extra information** to make things unambiguous. This works using unique identifiers (UIDs) specified by the uid annotation, as we will see below.

Renaming Entities and Properties

So why do we need that UID annotation? If you simply rename an entity struct, ObjectBox only sees that the old entity is gone and a new entity is available. This can be interpreted in two ways:

- The old entity is removed and a new entity should be added, the old data is discarded. This is the **default behavior** of ObjectBox.
- The entity was renamed, the old data should be re-used.

So to tell ObjectBox to do a rename instead of discarding your old entity and data, you need to make sure it knows that this is the same entity and not a new one. You do that by attaching the internal UID to the entity.

For properties, the process is the same, but instead of the comment, you just use standard Go tags. We are showing both cases in the following example.

Step 1: Add an empty \`objectbox:"uid"\` annotation to the entity/property you want to rename:

```
// `objectbox:"uid"`
type OldEntityName struct {
  Id  uint64
}

```

```go
type Task struct {
  Id  uint64
  OldPropertyName string `objectbox:"uid"`
}
```

Step 2: Re-generate ObjectBox code for the project using `go generate ./...` in your project directory. The generation will fail with an error message that gives you the current UID of the entity/property:

```
output for empty "uid" annotation on an entity

can't merge binding model information: uid annotation value must not be em
(model entity UID = 1306759095002958910) on entity OldEntityName
```

```
output for empty "uid" annotation on a property

can't merge binding model information: uid annotation value must not be em
    [rename] apply the current UID 9141374017424160113
    [change/reset] apply a new UID 6050128673802995827
```

Note how for a property, the output is slightly different and, besides support for renaming, it provides a newly generated UID you can use to effectively reset (clean) the stored data on the property. See Reset data - new UID on a property for more details.

Step 3: Apply the UID printed in the error message to your entity/property:

```go
// `objectbox:"uid:1306759095002958910"`
type OldEntityName struct {
  Id  uint64
}

type Task struct {
  Id  uint64
  OldPropertyName string `objectbox:"uid:9141374017424160113"`
}
```

Step 4: The last thing to do is the actual rename on the language level:

```
// `objectbox:"uid:1306759095002958910"`
type RenamedEntity struct {
  Id  uint64
}

type Task struct {
  Id  uint64
  RenamedProperty string `objectbox:"uid:9141374017424160113"`
}
```

You can now use your renamed entity/property as expected and all existing data will still be there.

Note: Instead of the above you can also find the UID of the entity/property in the `objectbox-model.json` file yourself and add it together with the @Uid annotation before renaming your entity/property.

Changing Property Types

> ObjectBox does not support migrating existing property data to a new type. You will have to take care of this yourself, e.g. by keeping the old property and adding some migration logic.

New property, different name

This solution useful if you need data migration or just want to keep the old data around.

```
type Task struct {
  Id  uint64
  OldProperty string
}

// becomes

type Task struct {
  Id  uint64
  OldProperty string
  NewProperty int
}

// Note, if the property already had an UID annotation,
//  don't add the same UID to the new property - skip the annotation inste
```

Reset data - new UID on a property

This solution useful if you don't care about the original data at all = it will be lost.

Step 1: Add an empty `objectbox:"uid"` annotation to the property you want to reset:

```
type Task struct {
  Id  uint64
  Property string `objectbox:"uid"`
}
```

Step 2: Re-generate ObjectBox code for the project using `go generate ./...` in your project directory. The generation will fail with an error message that gives you the current

UID of the property:

```
can't merge binding model information: uid annotation value must not be em
    [rename] apply the current UID 9141374017424160113
    [change/reset] apply a new UID 6050128673802995827
```

Step 3: Apply the UID printed in the error message to your property (and change its type):

```
type Task struct {
  Id  uint64
  Property int `objectbox:"uid:6050128673802995827"`
}
```

You can now use the property in your entity as if it was a new one.

The original property data isn't really removed right away on old stored objects but will be empty when read from DB and overwritten (thus finally lost) next time an object is written.

Custom types

The following built-in types, their aliases and named types based on them are recognized as ObjectBox and stored as an appropriate internal type:

```
int, int8, int16, int32, int64
uint, uint8, uint16, uint32, uint64
bool
string, []string
byte, []byte
rune
float32, float64
```

Defining a converter

To add support for a custom type, you can map properties to one of the built-in types using a `converter` annotation.

For example, you could define a color in your entity using a custom `Color` struct and map it to an `int32`. Or you can map the `time.Time` to an `int64`, though losing some precision - less than a millisecond, i. e. a thousandth of a second):

```
type Task struct {
  Id          uint64
  Text        string
  DateCreated time.Time  `objectbox:"date type:int64 converter:timeInt64"`
}
```

In the entity definition above, we instruct ObjectBox to store the `DateCreated` field as a `int64` while converting it to/from `time.Time` when using in the program. ObjectBox will generate a binding code that will call the following two functions (both start with the prefix `timeInt64` specified above):

```go
// from DB value to runtime value
func timeInt64ToEntityProperty(dbValue int64) (time.Time, error)

// from runtime value to DB value
func timeInt64ToDatabaseValue(goValue time.Time) (int64, error)
```

Just to complete the example, those functions could be implemented like this:

```go
// converts Unix timestamp in milliseconds (ObjectBox date field format) t
func timeInt64ToEntityProperty(dbValue int64) (goValue time.Time, err erro
  err = goValue.UnmarshalText([]byte(dbValue))
  if err != nil {
    err = fmt.Errorf("error unmarshalling time %v: %v", dbValue, err)
  }
  return goValue, err
}

// converts time.Time to Unix timestamp in milliseconds
// i. e. internal format expected by ObjectBox on a date field
func timeInt64ToDatabaseValue(goValue time.Time) (int64, error) {
  var ms = int64(goValue.Nanosecond()) / 1000000
  return goValue.Unix()*1000 + ms, nil
}
```

Actually this converter for `time.Time` is already part of the `objectbox` package and used automatically when you mark a `time.Time` property with `` `objectbox:"date"` ``.

Queries on custom types

When you use a converter, the actual value stored in the database is the result of the `...ToDatabaseValue()` call, e.g. `int64` in the previous example. Therefore, when you want to compare the stored data in a query condition, make sure you use the converted value as well:

```go
// Create
id, _ := box.Put(&model.Task{
  Text: "Buy milk",
  DateCreated: time.Now().UTC()
})

// Query
minTime, _ := time.Parse(time.RFC3339, "2018-11-28T12:16:42.145+07:00")
minTimeInt64, _ := objectbox.TimeInt64ConvertToDatabaseValue(minTime)
tasks, _ := box.Query(
  model.Task_.DateCreated.GreaterThan(minTimeInt64)
).Find()
```

Things to look out for

You must **not interact with the database** (such as using `Box` or `ObjectBox`) inside the converter. The converter methods are called within a transaction, so for example getting or putting entities to a box will fail.

Your converter implementation must be **thread safe** as it can be called from multiple go routines in parallel. Try to avoid using global variables.

`Query` is unaware of custom types. You have to **use the primitive DB type for queries**.

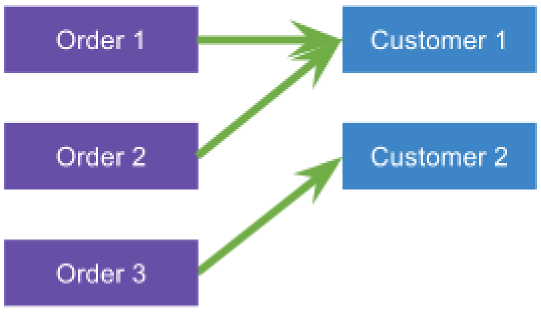
Relations

Objects may reference other objects, for example using a simple reference or a list of objects. In database terms, we call those references **relations**. The object defining the relation we call the **source** object, the referenced object we call **target** object. So the relation has a direction.

If there is one target object, we call the relation **to-one.** And if there can be multiple target objects, we call it **to-many**.

> Relations are initialized eagerly by default - i.e. the targets are loaded & as soon as the source object is read from the database. Lazy/manual loading of to-many relations is possible, using an annotation.

To-One Relations



To-One Relations

You define a to-one relation using `link` annotation on a field that is a pointer or value type of another entity. Consider the following example - the Order entity has a to-one relation to the Customer entity.

```
model.go
```

```go
type Order struct {
  Id         uint64
  Customer   *Customer `objectbox:"link"`
  Notes      string
}

type Customer struct {
    Id    uint64
    Name  string
}
```

Now let's add a customer with a few orders.

main.go

```go
// note that here we're creating a new customer
// but we could have also reused an existing one
var customer = &model.Customer{Name: "ACME Inc."}

var box = model.BoxForOrder(ob)

// Insert a new order. ObjectBox also inserts the customer automatically
// because it's new (customer.Id == 0 at this point)
box.Put(&model.Order{
    Notes: "first order, new customer",
    Customer: customer,
})

...

// Add another order. Now the customer.Id is already > 0
// so it's not inserted again, just referenced
box.Put(&model.Order{
    Text: "second order, existing customer",
    Customer: customer,
})
```

After the `box.Put` has been executed on the first order, the `customer.Id` would be `1` because we're using pointers (`Customer *Customer` field) so Put could update the variable when it has inserted the Customer. Note that this wouldn't be possible if we were using copies (`Customer Customer` field) and in that case you should insert the customer manually into it's box first (or use an existing customer selected from the database).

We can also **read**, **update** or **remove** the relationship to a customer:

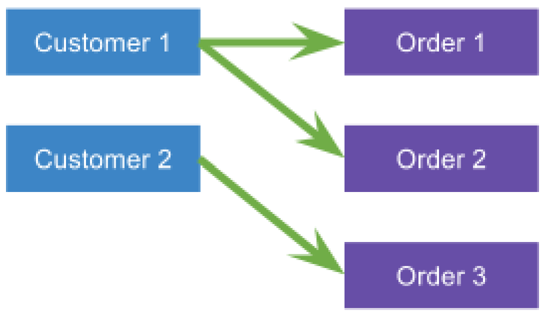
```
main.go

var box = model.BoxForOrder(ob)

order, _ := box.Get(1) // Read
// at this point, order.Customer is already loaded automatically (eager-lo

order.Customer = nil    // Remove the relation
box.Put(order)          // Update

// or do an update to a different customer
customers, _ := model.BoxForCustomer(ob).GetAll()
order.Customer = customers[2]
box.Put(order)
```

Note that removing the relation does not remove the customer from the database, it removes only the link between this specific order and the customer.

To-Many Relations

There is a slight difference if you require a one-to-many (1:N) or many-to-many (N:M) relation.
A 1:N relation is like the example above where a customer can have multiple orders, but an order is only associated with a single customer. An example for an N:M relation are students and teachers: students can have classes by several teachers but a teacher can also instruct several students.

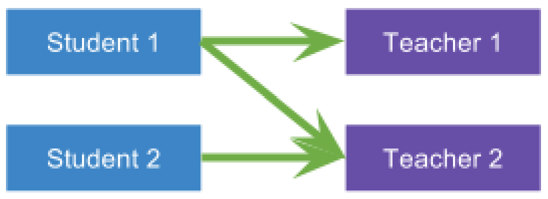
One-to-Many (1:N)



One-to-Many (1:N)

Currently, one-to-many relations are defined implicitly as an opposite relation to a to-one relation as defined above. This is useful for queries, e.g. to select all customers with an order placed within the last seven days.

Many-to-Many (N:M)



Many-to-Many (N:M)

To define a to-many relation, you can use a slice of entities - no need to specify the `link` annotation this time because ObjectBox wouldn't know how to store a slice of structs by itself anyway so it assumes it must be a many-to-may relation. They're stored when you put the source entity and loaded when you read it from the database, unless you specify a `lazy` annotation in which case, they're loaded manually, using `Box::GetRelated()` .

Assuming a students and teachers example, this is how a simple student class that has a to-many relation to teachers can look like:

model.go

```go
type Teacher struct {
  Id    uint64
  Name  string
}

type Student struct {
    Id       uint64
    Name     string
    Teachers []*Teacher
}
```

Adding the teachers of a student works exactly like with a list:

```go
main.go

var teacher1 = &model.Teacher{Name: "John Wise"}
var teacher2 = &model.Teacher{Name: "Peter Clever"}

var student1 = &model.Student{
    Name: "Martin Curious",
    // we can create the slice in place
    Teachers: []*model.Teacher{teacher1, teacher2},
}

// or append to it
var student2 = &model.Student{Name: "Earl Eager"}
student2.Teachers = append(student2.Teachers, teacher2)

// puts students and teachers
var box = model.BoxForStudent(ob)
box.Put(student1)
box.Put(student2)
```

Similar to the to-one relations, related entities are inserted automatically if they are new. If the teacher entities do not yet exist in the database, the to-many will also put them. If they already exist, the to-many will only create the relation (but not put them).

To **get** the teachers of a student we just access the list:

```go
main.go

var student1 = model.BoxForStudent(ob).Get(1);
```

```go
for _, teacher := range student1.Teachers {
    fmt.PrintLn(teacher.Name)
}
```

Remove and **update** work similar to insert - you just change the `student.Teachers` slice to reflect the new state (i.e. remove element, add elements, etc) and `box.Put(student)`. Note that if you want to change actual teacher data (e.g. change teachers name), you need to update the teacher entity itself, not just change it in one of the student.Teachers slice.

Lazy loading

In case the slices might contain many objects and you don't need to access the slice of the related objects each time you work with the source object, you may consider enabling the so called lazy-loading. You do that by specifying the `lazy` annotation on the field. Consider the updated model of the previous example:

model.go

```go
type Teacher struct {
  Id    uint64
  Name  string
}

type Student struct {
    Id        uint64
    Name      string
    Teachers  []*Teacher `objectbox:"lazy"`
}
```

This way, when you read a `Student` object, the `Teachers` field would be `nil` and you can work with the student as you wish, changing it and saving and the list of assigned teachers wouldn't change as long as the `Teachers` field stays `nil`. If it wasn't `nil`, but a slice of Teachers instead, ObjectBox would recognize this as an update of the field and replace the relational links.

Reading a lazy-loaded slice

To access the list of `Teachers`, we need to first load them. ObjectBox has generated a helper method just for that

```
main.go

var box = model.BoxForStudent(ob)

var student1 = box.Get(1);

// at this point `student1.Teachers == nil`, so if we need it, we must loa
box.GetRelated(student1) // loads all lazy-loaded relations

// or alternatively load just the Teachers property
// (useful if there were other lazy-loaded relations we didn't care about
box.GetRelated(student1, Student_.Teachers)

// now the teachers are loaded and we can access them as usual
for _, teacher := range student1.Teachers {
    fmt.PrintLn(teacher.Name)
}
```

Updating a lazy-loaded slice

To update the list of `Teachers`, we can either overwrite the slice with completely new data (new slice), or if we want to keep the original data and update it, e.g. change a few items, we need to load them first the same way as when reading (above).

```
main.go

var box = model.BoxForStudent(ob)

var student1 = box.Get(1);

// propagate student1.Teachers based on the current data in DB
box.GetRelated(student1, Student_.Teachers)

// add a new teacher to the existing
student1.Teachers = append(student1.Teachers, &model.Teacher{Name: "Peter

// save the updated list, including a new teacher
box.Put(student1)
```

FAQ

How is ObjectBox different from BoltDB (bolt/bbolt) or Badger?

BoltDB and Badger are **key/value stores**. These are database primitives using bytes for keys and values. Many databases, including ObjectBox, build on top of a K/V layer to provide a higher level interface than "just bytes". One approach to do that is a separate ORM layer (like GORM, Storm, etc.). ObjectBox Go takes a slightly different approach, integrating both together to provide a concise and type-safe interface with great performance all in one package.

Therefore, we call ObjectBox an **object database**. You interact with it using objects; the same structs you use in your Go code. Just like that, no tearing apart for SQL whatsoever required.

Also, ObjectBox knows the "inside" of objects; and allows to query for struct fields ("properties"). It also manages indexing for you, you just have to specify which properties should be indexed.

Couldn't I just use JSON to store data? (or anything file-based)

Note: The same also applies other simple file based approaches using CSV, XML, or object collections stored using binary serializations like Protocol Buffers, BSON, MessagePack, ...

It's perfectly fine to store data as JSON if you have a limited number of objects to manage. The more scalable approach to manage data, however, is to use a database like ObjectBox:

- ObjectBox uses a binary encoding which is faster
- Individual object changes: Let's say you have a list of objects and you change a single value. Using JSON, you typically write the entire list. ObjectBox touches a single object only.

- Random access: In ObjectBox you can get single objects efficiently without need to parse the entire JSON file.
- Memory efficiency: In close collaboration with the OS, ObjectBox can "page" through large data sets which would not fit into memory
- Queries & Indexing: ObjectBox comes with automatic indexing; which will drastically improve queries.
- ObjectBox offers ACID transactions to keep your data safe and consistent.

Is ObjectBox ACID compliant? Is it an in-memory database?

ObjectBox comes with ACID transactions. It has hard durability (the "D" in ACID) semantics. For synchronous transactions (e.g. what happens under the hood for Box.Put()), data is stored durable once it returns. Unlike ObjectBox, many NoSQL DBs have relaxed durability semantics (e.g. a time window of second where data can be lost).

ObjectBox is not an in-memory database. The latter have high RAM requirements because they have to keep ALL data in memory. ObjectBox usage of RAM is flexible; it doesn't need much but makes use of RAM if it is available. That is why ObjectBox is usually as fast as an in-memory database.

I'm getting "exit status 3221225781" on Windows

Exit status 3221225781 is a secret code :) for a missing DLL on windows. So even if your code has been compiled successfully, Windows can't find the DLL when launching the application (or tests). The DLL needs to be copied somewhere Windows will recognize it. See Installation on Windows for instructions.

I'm getting errors during installation

You may encounter various seemingly unrelated errors when trying to install ObjectBox on a system without a C/C++ compiler (or when trying to cross-compile). These may be, for example:

- `... objectbox-go/objectbox/condition.go:21:14: undefined: QueryBuilder`
- `cannot find module for path github.com/objectbox/objectbox-go/internal/generator`
- `pkg/mod/github.com/objectbox/objectbox-go@v1.0.0/objectbox/model.go:30:2: cannot find package`

These errors are caused by Go failing to compile the package correctly. Because ObjectBox is using a native library, it requires CGO, which in turn needs a C/C++ compiler. Please make sure you have a working C/C++ compiler installed. See the installation instructions for further details.

> Trying to cross-compile ObjectBox may also be giving you similar errors. The reason is the same - not having the right C/C++ cross-compiler set up.
>
> Currently, cross-compilation is not tested/supported so please use native platform compilation for now and upvote this GitHub issue if cross-compilation is important for you.