ObjectBox Swift

ObjectBox Swift Usage Guide

First steps with ObjectBox for Swift:

→ Installing ObjectBox /install

→ Getting Started Using ObjectBox in Your Project /getting-started

If you have any questions, don't hesitate to get in touch! contact [at] objectbox.io

---

ObjectBox Change Log

1.3.0 - 2020-05-11

- Built with and for Swift 5.2
- Fix sporadic "errno 12" while opening a DB on iOS devices
- Lower deployment target to iOS 9.3 (seems to be still used by older iPads)
- Fix for PropertyQuery (e.g. single line construction without holding on to Query)
- API clean ups
  - Remove deprecated Query.all(); use find() instead
  - Box functions like visit(), for() and forEach() are now read-only by default. New optional writable flags changes this to use a write transaction instead.

1.2.0 - 2019-12-17

- Added support for optional unsigned property types
- Better type support for queries; e.g. unsigned and optional properties, Bool properties
- Property queries compute sums and averages more precisely (improved algorithms and wider types)

- Some Query API clean up, e.g. setting query parameters does not throw anymore, findUnique() always returns an optional result, etc.

1.1.1 - 2019-11-23

Fix for the 1.1 "realpath" error during build on machines without Homebrew coreutils.

1.1 - 2019-11-18

- Experimental Carthage support (in addition to CocoaPods)
- If a `convert` annotation on an enum has no explicit database type given, its `RawType` is used.
- Various performance optimizations
- Usability improvements, e.g. `setup.rb` now asks which project to use if there is more than one, and creates a non-empty `*.generated.swift` file to help with code completion.
- Added `put()` with variable argument list, `put(inout)` for mutable structs, and `putAndReturnIDs()`. Upgrade note: `put()` itself no longer returns IDs.
- Added variants of Box methods for `ContiguousArray` class, which are faster than the methods using standard arrays.
- Anonymous build statistics can be turned off using `--no-statistics`

1.0.1 - 2019-10-01

- Fix bug that prevented code generator from working in projects that use SwiftPM

1.0 - 2019-09-24

- Relations in queries
- Observer callbacks for data changes
- Many-to-many relations
- Asynchronous put/remove
- Edit relations through their backlinks
- API clean up; e.g. renames, simple IDs, simplifications, etc.

- If you've been using pre-releases, you can find migration instructions in the Readme on Github.

0.9.1 - 2019-08-13

iOS hotfix

- Fixed "Storage error code 78" for iOS

0.9.0 - 2019-07-22

Open Source Release

- Source code for the Swift binding is now available
- Reduced write/update time by about 25%
- Rewrote the remaining Objective-C classes in Swift
- Fix an issue reading an old entity written before a new, non-optional property was added to it
- Improvements for people using SwiftLint in their projects
- Added `findIds()`, `visit()` / `forEach()`. Added `remove()` for queries.
- Improved support for `enum`s and other user-defined types
- `Data` and `[UInt8]` are now supported as property types.
- Binaries: Swift 5 ABI only

0.8.0 - 2019-05-14

Struct Support and Performance Improvements

- Immutable structs can now be used, you are no longer restricted to classes
- We no longer throw `NSError`-based errors, they're all `enum ObjectBoxError` cases now
- Strings that were created as `NSStrings` previously are now created as Swift `String`s, which should reduce the number of unnecessary UTF-16/UTF-8 roundtrip conversions when Swift 5.1 arrives
- New projects are now set up with separate generated source files for each target by `setup.rb`.

- Binaries: Swift 5 ABI only

0.7.0 - 2019-04-02

Swift 5 and Build Improvements

- Binaries: Swift 5 ABI only
- ObjectBox setup script can now also be used without CocoaPods
- ObjectBox can now be used in frameworks

0.6.0 - 2018-12-19

Model Migration

- Your data model is now migrated transparently for you when you make changes
- Properties can be indexed
- You can require property fields to be unique
- You can specify a different name for use in the model than the instance variable's
- Use of ObjectBox without CocoaPods has been simplified
- You do not need to annotate back-links in relations anymore in clear cases
- Binaries: Swift 4 ABI only.
- **Alpha users:**
  - Use annotations `// objectbox:` instead of `// sourcery:`
  - Delete any database files you created before
    (one time only, starting with the beta we have model migrations)

0.5.5 - 2018-11-29

Alpha 6

- Remove code-generator limitations regarding order of properties vs. relations
- Support sandboxed macOS applications (see macOS Sandbox setup)
- Add "transient" annotation for skipping properties
- Binaries: Swift 4 ABI only.

0.5.4 - 2018-11-27

Alpha 5

- Code generation fixes for optionals
- Expanded example app to demo optionals
- Fixes for Date-decoding into entities
- Binaries: Swift 4 ABI only

0.5.3 - 2018-11-26

Cleanup

- Just small things. Also an elephant.

0.5.2 - 2018-11-22

iOS Sandboxing.

- Fixed issues that could occur when deploying to device.
- Added auto-registration of your entities

0.5.1 - 2018-11-19

Alpha 2

- Fix an issue related to "duplicate index ID" error message.

0.5.0 - 2018-11-16

Alpha 1

- Initial public release for comment.

Installing ObjectBox

> Having problems installing ObjectBox using this guide?
> Please, let us know where we lost you. Thanks for your help!

Adding the ObjectBox Framework to Your Project

We recommend to setup ObjectBox using CocoaPods. Other alternatives are Carthage (experimental) or a manual setup in Xcode. Note that Swift Package Manager does not yet support binary frameworks and thus is not an option yet.

If you are familiar with CocoaPods or Carthage, here's the quick setup (details below):

CocoaPods

1. Add `pod 'ObjectBox'` to the `Podfile` in your Xcode project directory.
2. Run these **two** commands in Terminal:
   `pod install --repo-update`
   `Pods/ObjectBox/setup.rb`
3. Open your `.xcworkspace` in Xcode.

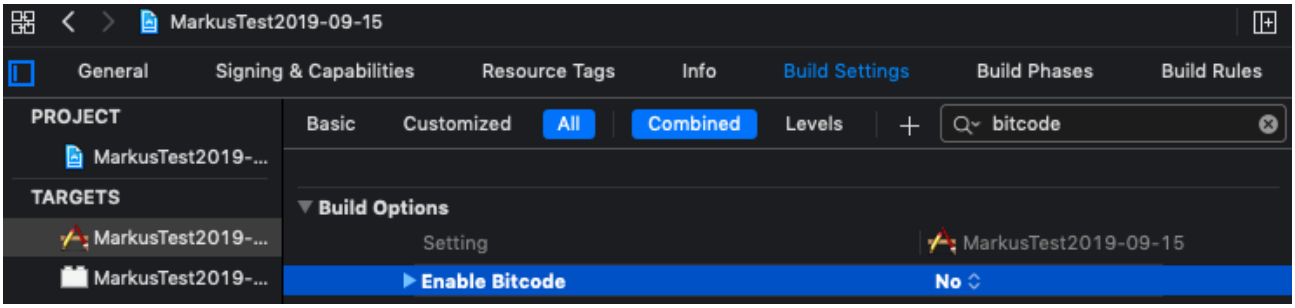
Carthage (experimental)

Carthage support is experimental and only recommended for **experienced users**. Note: Carthage may falsely report incompatible Swift 5.1.x versions. This Carthage bug prevents the framework to be installed!

1. Add `github "objectbox/objectbox-swift"` to your Cartfile
2. Run `carthage update` to download ObjectBox.
3. Add either `Carthage/Build/Mac/ObjectBox.framework` or `Carthage/Build/iOS/ObjectBox.framework` to the appropriate targets of your project
4. In the "General" build settings tab, locate the "ObjectBox.framework" and sure one of "Embed Without Signing" or "Embed & Sign" is selected.
5. Run `gem install xcodeproj` and then run the ObjectBox setup script from `Carthage/Build/Mac/OBXCodeGen.framework/setup.rb` in your project directory.

Project settings

Because ObjectBox does not include bitcode at this point you need to adjust "Build Settings" in Xcode accordingly. In that tab, ensure "All" is active and search for "bitcode". You will find the "Enable Bitcode" setting which you set to "No".



Otherwise, for some build targets, you will get a build error like this:

```
'.../ObjectBox.framework/ObjectBox' does not contain bitcode. You must rebuild
it with bitcode enabled (Xcode setting ENABLE_BITCODE), obtain an updated
library from the vendor, or disable bitcode for this target.
```

> **Still using Swift 5.0?** ObjectBox 1.1 supports Swift 5.1+. If you want to give ObjectBox a try but are stuck with Swift 5.0, use the ObjectBox501 pod. We recommend Swift 5.1 though.

Detailed Instructions

CocoaPods

If you don't have CocoaPods already, install it by typing `sudo gem install cocoapods` in Terminal. If you don't have a `Podfile` yet, `cd` into your Xcode project folder and create one using `pod init` as usual. Now find the `# Pods for MyTargetName` comment for each target and add the following below it:

```
pod 'ObjectBox'
```

Then run:

```
pod repo update
pod install
Pods/ObjectBox/setup.rb
```

The first call will ensure your local copy of the list of available CocoaPods is current, the second call is your usual procedure when adding a new pod to your `Podfile` that will download the newest release of ObjectBox to your `Pods` directory and install the pod in your Xcode workspace.

Once you have ObjectBox downloaded, `setup.rb` will set up the ObjectBox preprocessor in your project for every target with an executable.

You can now open your project's Xcode workspace as usual and follow the rest of this tutorial.

Troubleshooting

Something doesn't work yet (e.g. `pod install` fails)? Try updating CocoaPods related things using:

```
gem update xcodeproj && gem update cocoapods && pod repo update
```

Updating

After a new ObjectBox version is released, you need to get the new pod and run the setup script again:

```
pod repo update
pod update ObjectBox
Pods/ObjectBox/setup.rb
```

Carthage

If you don't have Carthage yet, install it e.g. using Homebrew `brew install carthage`. If you do not have a Cartfile yet, create a text file named just `Cartfile` (no `.txt` suffix) in your project's folder and write the following text into it

```
github "objectbox/objectbox-swift"
```

Then open Terminal, `cd` into the folder containing your project and run

```
carthage update
gem install xcodeproj
```

```
3  Carthage/Build/Mac/OBXCodeGen.framework/setup.rb
```

The first call is the typical way to make Carthage download a dependency specified in your Cartfile. Once you have ObjectBox downloaded, `setup.rb` will set up the ObjectBox preprocessor in your project for every target with an executable. The `gem install xcodeproj` line installs a helper tool needed by the setup script.

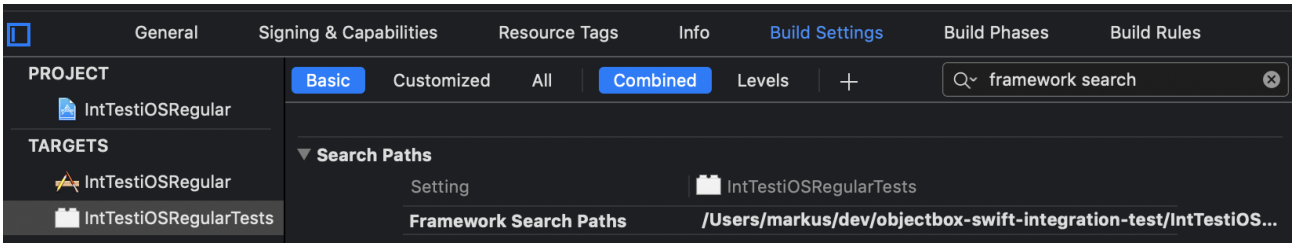
> Carthage has a bug preventing the Swift pre-built framework from working. It falsely reports that Swift 5.1.x versions are incompatible. **This will cause the setup to fail!**
> Track this Carthage issue for updates.
>
> *The error looks like this:*
> ```
> Skipped installing objectbox-swift.framework binary due to the error:
>  "Incompatible Swift version - framework was built with 5.1.2
> (swiftlang-1100.0.278 clang-1100.0.33.9) and the local version is 5.1
> (swiftlang-1100.0.270.13 clang-1100.0.33.7)."
> ```

Like for all Carthage projects, you now need to open your project in Xcode and add the framework to your target. You do this by dragging either the `Carthage/Build/iOS/ObjectBox.framework` or `Carthage/Build/Mac/ObjectBox.framework` (depending on whether your project is for iOS or macOS) to the "Frameworks, Libraries and Embedded Content" list of the "General" tab of your target's settings. Make sure you choose "Embed and Sign" to its right.

You can now open your project's Xcode workspace as usual and follow the tutorial.

> Ignore the `OBXCodeGen.framework` that you'll also see in `Carthage/Build`. It is *not* intended to be linked into your application and only exists to hold the code generator and setup script so you have them while you build.

Secondary targets, e.g. unit tests



If your unit test target does not find the ObjectBox library, please verify that the "Framework Search Paths" build settings of that secondary target are set up. Check the screenshot how to find the setting, and compare the value to the one from your main target. To fix it, press enter and ensure that the following values are present.

For iOS targets:

```
$(inherited) $(PROJECT_DIR)/Carthage/Build/iOS
```

For macOS targets:

```
$(inherited) $(PROJECT_DIR)/Carthage/Build/macOS
```

Otherwise, you may get errors like this:

```
ld: warning: Could not find or use auto-linked framework 'ObjectBox'
Undefined symbols for architecture x86_64: "type metadata accessor for
ObjectBox.ToManyProperty" ...
```

Updating

After a new ObjectBox version is released, you need to update via Carthage and run the setup script again:

```
carthage update
Carthage/Build/Mac/OBXCodeGen.framework/setup.rb
```

Getting Started Using ObjectBox in Your Project

To see ObjectBox in action, you need to

1. **Install ObjectBox** if you haven't already.
2. **Define your Entities**, i.e. the classes you want to persist.
3. import ObjectBox into your source file

Define your Entities

ObjectBox uses code generation to read and write your objects instead of forcing you to inherit from a base object. To mark an object as an entity, you have two options:

- Conform to the `Entity` protocol

  or
- add `// objectbox:entity` before the type as an annotation for the code generator.

Then supply an ID property and parameter-less initializer `init()`, and you're good to go:

```
import ObjectBox

class ExampleEntity: Entity {
    var id: Id = 0

    required init() {
        // no properties, so nothing to do here, ObjectBox calls this
    }
}
```

That is all you need to get going. Build your project and ObjectBox's code generator will generate the boilerplate code to persist and read objects of this type.

> You initialize an `Id` with 0 for all entities. This tells ObjectBox your object isn't persisted, yet. When you persist objects, the ID will be changed to the actual value. And once an entity has a non-zero ID, persisting changes will update the existing record instead of creating a new one.
>
> So the ID is managed by ObjectBox. Conversely, you do not want to modify it yourself.

Initialize a Store

To create or access a database on disk, use the `ObjectBox.Store` class. The `Store` behaves much like a database connection: you keep the instance around to maintain an open connection to the data in a folder on disk. Usually for the lifetime of your app.

```
let store = try Store(directoryPath: "/Users/jenna/Documents/mydatabase/")
```

> Getting "**Missing argument for parameter 'model'**..." here?
> **Build the project first.**
>
> Background: The ObjectBox code generator creates a second Store() initializer without the model parameter. It does this after parsing all your entities, which happens at build time. Thus, make sure to build your project at least once after declaring your entities. Otherwise Xcode's CodeSense auto-completion will not propose the right initializer in its list.

Of course, you would usually save your database in one of the standard system directories, like `.applicationSupportDirectory`, `.documentsDirectory` or `.cachesDirectory`:

```
let databaseName = "notes"
let appSupport = try FileManager.default.url(for: .applicationSupportDirec
                                             in: .userDomainMask,
                                             appropriateFor: nil,
                                             create: true)
                                             .appendingPathComponent(Bundl
let directory = appSupport.appendingPathComponent(databaseName)
try? FileManager.default.createDirectory(at: directory,
                                         withIntermediateDirectories: true
                                         attributes: nil)

let store = try Store(directoryPath: directory.path)
```

Working with Object Boxes

Since ObjectBox is all about sticking objects in boxes, you interact with objects using a `Box` interface. For each object class, there is a matching `Box` instance.

To manage your entities, you retrieve the `ObjectBox.Box<T>` instance for its class from your `Store`. Boxes are lightweight and can be discarded, but then you have to pass the `Store` around. You will almost always prefer to pass long-lived `Box` instances around instead, for example in segues between `UIViewControllers`.

```
let exampleEntityBox = store.box(for: ExampleEntity.self)
// Similarly:
let personBox = store.box(for: Person.self)
let noteBox: Box<Note> = store.box()
```

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. There are `put` variants that support writing multiple objects at once, which is more efficient than writing each with its own call to `put`.

- **get:** When you have an object's ID, you can get to the object very efficiently using `get`. To get all objects of a type, use `all`.
- **remove:** Deletes a previously persisted object from its box. There are method overloads to remove multiple entities, and `removeAll` to delete all objects and empty the box.
- **count:** The number of objects stored in this box.
- **query:** Lets you provide a query expression to retrieve objects by certain criteria. See the page on queries for details.

Check the API docs for Box for a list of operations.

Put and Get Example

Once you have a box, it is simple to persist an entity. To illustrate the change of IDs, we added assertions to the code; you don't need these, of course.

```
// let's start with an empty box
assert(try exampleEntityBox.isEmpty())

let exampleEntity = ExampleEntity()
assert(exampleEntity.id.value == 0)

let newID = try exampleEntityBox.put(exampleEntity)
// Change of ID with the `put`:
assert(exampleEntity.id.value != 0)
assert(exampleEntity.id == newID)

// Check the Box contents did indeed change:
assert(try exampleEntityBox.count() == 1)
assert(try exampleEntityBox.all().first?.id == newID)

// Getting to a specific object
guard let foundEntity = try exampleEntityBox.get(newID)
    else { fatalError("Object should be in the box") }
assert(exampleEntity.id == foundEntity.id)

// Cleaning up
try exampleEntityBox.removeAll()
assert(try exampleEntityBox.count() == 0)
```

Put with structs

Structs basically work the same way as classes. However, since structs are value types, ObjectBox can only adjust their ID if you pass them by reference. So if you have a mutable struct, remember to pass it by reference:

```
// objectbox: entity
struct Author {
    var id: Id // Do not initialize the ID to 0 for structs.
    var name: String
    var age: Int
}

let exampleEntityBox = store.box(for: Author.self)
var exampleEntity = Author(id: 0, name: "Nnedi Okorafor", age: 45)

// Put the struct and update the ID:
try exampleEntityBox.put(&exampleEntity)
assert(exampleEntity.id != 0)
```

Alternately, you can manually update the struct's ID field by using `putAndReturnID()` or `putAndReturnIDs()` to get the ID a struct was written as:

```
exampleEntity.id = exampleEntityBox.putAndReturnID(exampleEntity)
```

Given immutable structs are such a common occurrence, ObjectBox generates a convenience method for you to help with updating the ID on an immutable struct, `put(struct:)`, which will automatically create a new copy of your struct with the ID filled out:

```
// objectbox: entity
struct Author {
    let id: Id // Do not initialize the ID to 0 for structs.
    let name: String
    let age: Int
}

// Let's start with an empty box:
let exampleEntityBox = store.box(for: Author.self)
```

```
let exampleEntity = Author(id: 0, name: "Nnedi Okorafor", age: 45)

// Put the immutable struct:
let newEntity = try exampleEntityBox.put(struct: exampleEntity)
assert(newEntity.id != 0)
```

Note that, after writing `exampleEntity`, you *must* use `newEntity` from then on. If you called `put(struct:)` or `put()` on exampleEntity again, ObjectBox would not know that this object has already been saved, and would write a second copy of it to the database.

But what if you know you already saved an immutable entity, and you already made a copy because you changed one of its fields, and don't need another copy? Then you can call `put()`, just like above:

```
// Modify object:
let futureExampleEntity = Author(id: newEntity.id, name: newEntity:name,
                                 age: newEntity.age + 1)

// Write out changes:
try exampleEntityBox.put(futureExampleEntity)
```

So beyond having to use a special call the first time you write out an entity, everything is the same as with classes.

> For ObjectBox to work with structs, there needs to be an `init()` method on your struct that accepts all its persisted properties. In the above example, the default initializer provided for the struct by Swift does just fine, but if you have properties that are not persisted, or you are defining your own initializer, you may also have to provide that initializer.
>
> Note that for structs you will usually not want to specify a default value of 0 for the ID like you do for classes. If you do, the default initializer will omit the "id:" parameter and there would be no way to initialize the ID without writing your own initializer.

Transactions

Transactions in ObjectBox let you group together several operations and ensure that either all of them complete, or none does, always leaving your data relations in a consistent state. If you do not run your own transaction, ObjectBox will implicitly create one for every call:

- A `put` runs an implicit transaction.
- Prefer the `put` variant that takes an array of entities (like `put([person1, person2])`) whenever possible to increase performance.

For a high number of interactions inside loops, consider wrapping the loop in explicit transactions. `Store` exposes methods like `runInTransaction` for this purpose.

```
// Bad/slow, each Put runs inside a new transaction
for i in (1...1000) {
    try box.put(AnEntity(number: i))
}

// Possible, but still inefficient
try store.runInTransaction {
    for i in (1...1000) {
        try box.put(AnEntity(number: i))
    }
}

// Much better
let allEntities: [AnEntity] = (1...1000).map(AnEntity.init(number:))
try box.put(allEntities)
```

For details, check the Transaction guide and the API docs for Store for details.

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 classes are entities you can either make them implement the `ObjectBox.Entity` protocol, or add annotations to them. Then ObjectBox can do its magic with your entities.

Here is an example:

```
// objectbox: entity
class User {
    var id: Id = 0

    var name: String

    // objectbox: transient
    private var tempUsageCount: Int // not persisted

    // ...
}
```

The **Entity** annotation identifies the Swift class `User` as a persistable entity. This will trigger ObjectBox to generate persistence code tailored for this class, even if it does not conform to the `Entity` protocol..

> **Note:** It's often good practice to model entities as "dumb" data classes with just properties.

Object IDs: id

In ObjectBox, every object has an ID of type Id to efficiently get or reference objects. Usually ObjectBox will just find that property in your entity based on its type and "do the right thing", but if you have several properties of the same `Id` type, or your ID uses the types `UInt64` or `Int64`, you can use an annotation to mark a particular property as *the* ID of your entity:

```
class User: Entity {

    // objectbox: id
    var thisIsMyId: UInt64 = 0

    // ...
}
```

While we recommend to define your ID as type `Id` or `EntityId<MyEntity>` (where `MyEntity` would be replaced with the name of your class), you can also annotate a property of type `UInt64` or `Int64` as an ID.

ObjectBox will usually manage ID values for you, but should you absolutely need to, you can tell ObjectBox that you want to assign IDs manually by yourself by adding an "assignable" parameter to the id annotation:

```
// objectbox: id = { "assignable": true }
```

If you do that, make sure that you assign a unique ID to each new object. If you assign the same ID to two objects of the same class, writing one to the database will overwrite the other. In general, it is best to let ObjectBox assign IDs. You are free to provide an additional indexed property to e.g. store a GUID for an object in addition to the ID used by ObjectBox, and to use queries to retrieve objects based on that GUID.

Ways to Define IDs

To specify Entity IDs, you need to have a property of type `Id`. The code generator will look for one of these configurations, in the following order

1. A property that has a `// objectbox: id` comment annotation on the line above the property definition.
2. A property named `var id: Id`.
3. Any other property of the `Id` type, if there is only one.

You usually don't need more than one property of the `Id` type. Use ObjectBox's support for object relations if you need to connect entities with each other. In any case, should you need to store a reference to another object for an unusual purpose, simply use annotations to ensure ObjectBox uses the right property as the entity's ID.

Here's an example of using annotations in action:

```
// objectbox: entity
class ExampleEntity {
    var anotherID: Id = 0

    // objectbox: id
    var theEntityIdentifier: Id = 0 // <- this will be used

    required init() {
        // nothing to do here
    }
}
```

> ObjectBox supports several types for IDs. Apart from `Id`, you can also use the more type-safe `EntityId<ExampleEntity>` struct (where `ExampleEntity` is the class of your entity), which will let you ensure that you don't accidentally pass an ID of the wrong type into Box API.
>
> In addition, you can also use `UInt64` or `Int64` for your IDs, but for ObjectBox to know to use those, you *must* annotate them.

Using Different Names in the Database than in Swift

```
class User: Entity {
    // objectbox: name = "USERNAME"
    var name: String
    ...
}
```

The `name` annotation lets you define a name on the database level for a property. This allows you to rename the Swift field without affecting the property name on the database level. This is mostly useful in cross-platform code, where different platforms may have different conventions for property names and you need to exchange database files between them.

> **Note:** To rename properties and even entities you should use Uid annotations instead.

Transient Properties

```
class User: Entity {
    // objectbox: transient
    var tempUsageCount: Int
    ...
}
```

The `// objectbox: transient` annotation can be used to mark properties that should not be persisted, like the temporary counter above. As far as ObjectBox is concerned, transient properties simply do not exist. `static` properties are always ignored by ObjectBox and do not need to be marked as transient.

Property Indexes with Index

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

```
1 // objectbox: index
2 var name: String
```

> Index is currently not supported for `Data` , `Float` and `Double`

An index stores additional information in the database to make lookups "faster", or more correctly, "more scalable". With an index, you will get results fast no matter if you store ten, one thousand, or one million objects in the database. Index based database lookups perform in O(log n).

As an analogy we could look at how you store objects in an `Array<>` . For example you could store persons using an `Array<Person>` . Now, you want to search for all persons with a specific name so you would iterate through the list and check for the name property of each object. This is an O(N) operation and thus does not scale well with an increasing number of objects.

To make this more scalable you can introduce a second data structure `Dictionary<String, Person>` with the name as a key. This will give you a superfast lookup time (typically O(1)). The downside of this is that it needs more resources (here: RAM) and slows down add/remove operations on the list a bit. These principles can be transferred to database indexes, just that the primary resource consumed is disk space.

Index types (String)

Because `String` properties typically take more space than scalar values, ObjectBox uses a **hash** for indexing strings by default. For any other type, the property **value** is used for all index look-ups.

You can instruct ObjectBox to use a value-based index for a `String` property by specifying an index type:

```
// objectbox: index = value
var name: String
```

Keep in mind that for `String`, depending on the length of your values, a value-based index may require more storage space than the default hash-based index.

ObjectBox supports these index types:

- **Not specified** Uses best index based on property type (`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 hashes of property values to build the 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 64-bit hashes 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 constraints

Annotate a property with `unique` to enforce that values are unique before an entity is put:

```
// objectbox: unique
```

```
var name: String
```

A `put()` operation will abort and throw an `ObjectBoxError.uniqueViolation` error:

```
do {
    try box.put(User("Sam Flynn"))
} catch ObjectBoxError.uniqueViolation(let message) {
    // A user with that name already exists.
}
```

> Unique constraints are based on an index. You can further configure the index by adding an `index` annotation.

Converting Enums and Custom Types

You can add an `// objectbox: convert` annotation to properties with types that ObjectBox does not know to allow converting it into a recognized type. See Enums and Custom Types for more.

Relations

Creating to-one and to-many relations between objects is possible as well, and may require use of the `// objectbox: backlink` annotation, see the Relations documentation for details.

Triggering generation

You usually should not need to do anything special to trigger code generation once your entity classes are properly annotated. The `setup.rb` script should have automatically configured your project to run the code generator when you compile your project, for example using **Product > Build** in Xcode.

Should you have unusual needs or encountering issues, see Customizing Code Generation for a description of how things usually work.

Queries

Building Queries that Return Entities

Beyond simple "fetch all" commands, like `personBox.all()`, you may want to filter entities by their properties. ObjectBox's query syntax should be familiar to every Swift developer, because queries are written in Swift:

```
let query: Query<Person> = try personBox.query {
    Person.name.startsWith("Andrea")
}.build()
let allAndreas: [Person] = try query.find()
```

You simply write your class name, the property name, and then an operator, or a method name (like `startsWith()` here). While most of these calls look like the regular Swift calls you're used to, they are actually provided by ObjectBox's code generator, so not all methods you would e.g. find on a `String` are available for queries. See below for available operations.

> Store query objects for re-use to increase performance.
>
> You can call `Query.find()`, `Query.count()`, and all the other operations that execute a query multiple times. In performance-critical situations, building a new query object thousands of times can easily become costly.

Query Syntax

Combine Conditions with logical Operators and Parentheses

Query conditions can be combined by the use of the logical operators `&&` and `||`.

```
let query: Query<Person> = try personBox.query {
    Person.firstName.startsWith("Ste")
        && Person.lastName.startsWith("Jo")
}.build()
let maybeSteveJobs = try query.find()
```

Since these are overloads of native Swift operators, their standard operator precedence rules apply (`&& > ||`). You can use parentheses to group conditions and affect the truth condition of the whole expression.

That's fancy talk for: group condition parts by wrapping them in parens.

```
let query: Query<Person> = try personBox.query {
    (Person.firstName.contains("Santa") || Person.age > 100)
        && Person.lastName.isEqual(to: "Claus", caseSensitive: true)
}.build()
let oldClauses = try query.find()
```

QueryCondition Operators

ObjectBox also provides operators for the most common operations. Please note that the property comes first:

- **Collection containment**: `∈` and `∉` ; for example, disallowing teens entry to your disco: `Person.age ∉ (10..<18)`
- **Equality**: `==` and `!=` , as in `Person.firstName == "Steve"`
- **Comparison**: `<` and `>` , as in `CocoaPod.rating > 4.5`

> Those operators are available for optional types, which currently lack the long form.

There are no custom operators for conditions like `Property<T, String>.startsWith()` , as their names are already much more familiar to Swift developers.

Query Operations

Once you have a query set up properly, you can execute it as often as you want to get the latest results of its evaluation. These are the most common operations:

- `Query.find()` will return all results that match
- `Query.count()` will return the count of all results that match
- `Query.findUnique()` will return an unique match from the result set; returns nil if no result was found and throws if the result is not unique

Please refer to the Query API docs for a comprehensive list of permitted operations.

Sorting

In addition to specifying conditions you can order the returned results using the `ordered()` method:

```
let query = try userBox.query { User.firstName == "Joe" }
                .ordered(by: User.lastName) // in ascending order, ignorin
                .build()
```

You can also pass flags to `ordered(by:,flags:)` to sort in descending order, to sort case sensitively or to treat null values specially . For example, to sort the above results in descending order and case sensitively instead:

```
let query = try userBox.query { User.firstName == "Joe" }
                .ordered(by: User.lastName, flags: [.descending, .caseSens
                .build()
```

Order conditions can also be chained. Check the method documentation for details.

Modifying Conditions Later

You can modify conditions after creation of the query. That can be useful if your base query stays the same but one small part changes between `find()` executions and you don't want to create a new Query object every time.

To do that, you use the query's `setParameter()` methods. Refer to the Query API docs for a comprenhesive list of permitted `setParameter()` variants.

setParameter Changes the Condition Value

There are many useful variants of the `Query.setParameter()` and `Query.setParameters()` method. (Note the plural *s*.) The first version will set the comparison value of a condition that matches to a new value. The second version does the same for comparisons with two values, hence the plural *s*.

For example:

- `query.setParameter(Person.age, to: 18)` for a query you built using `personBox.query { Person.age == 21 }.build()` will effectively change the condition to `Person.age == 18`
- `query.setParameters(Person.age, to: 10, and: 18)` for a query you built using `personBox.query { Person.age.isBetween(0, and: 99) }.build()` will effectively change the condition to `Person.age.isBetween(10, and: 18)`

> You will get fatal errors if you try to call the plural-*s*-Version when the original condition only has one comparison value, and vice versa. Also note that this is an usage error and thus does not throw.

Use PropertyAlias to Disambiguate Conditions

One problem of the property-based approach above is that you have no control what happens when there are two or more conditions on the same property:

```
let query = try personBox.query {
    Person.age > 10
        && Person.age < 50
}.build()
query.setParameter(Person.age, to: 25) // did this affect "<" or ">" ?
```

To resolve this ambiguity, you can register a short name (called an *alias*) for a query condition. That's a simple string to give the condition a name.

We use a variant of the definition operator (`.=`) for this:

```
let query = try personBox.query {
    "MinAge" .= Person.age > 10
        && "MaxAge" .= Person.age < 50
}.build()
query.setParameter("MinAge", to: 25)
```

There is, of course, also a plural *s*-variant for conditions with two values. The same warning as above applies: make sure not to mix up the plural and singular versions.

A downside of the string-based alias approach is that you lose type information: you have to make sure not to pass a `Double` to a setter for a condition that operates on `String` , for example.

Building Queries that Operate on Properties

In addition to filtering entities, you can also build queries for their properties. These are aggregate functions like `max` , `min` , `average` , `count` , but also various find methods. These are available on the `PropertyQuery` type, that you can get from a regular query like this:

```
let query: Query<Person> = try personBox.query { Person.firstName.startsWi
```

```
let agePropertyQuery: PropertyQuery<Person, Int> = query.property(Person.a
```

The `PropertyQuery` will respect all conditions of its original `Query`. So in the example above, only entities with a `firstName` that starts with "S" will be regarded. Use the empty block variant `personBox.query()` if you want to operate on all entities.

Please refer to the PropertyQuery API docs for an exhaustive list of operations.

> Currently, property queries do not honor the `ordered(by:,flags:)` setting. If you need a fixed order for your proerty query results, you must sort them manually *after* retrieving them.

Aggregate Functions

A simple aggregate function is to request the maximum, minimum, or average value of a given property.

```
let query: Query<Person> = try personBox.query().build()
let agePropertyQuery: PropertyQuery<Person, Int> = query.property(Person.a
let maxAge: Int = try agePropertyQuery.max()
let minAge: Int = try agePropertyQuery.min()
let averageAge: Double = try agePropertyQuery.average()
```

Number-based operations are only available for number-based properties and not for strings, for example. Also note that the average of a property is always a floating point number, even if the property is an `Int` (what would you do with a non-fractional average anyway?).

Fetching all Property Values

In addition to aggregate functions, you can also use "find" methods. These will return any or all of the values of entities matching the query:

```
let query: Query<Person> = try personBox.query { Person.firstName == "Stev
let steveLastNamePQ: PropertyQuery<Person, String> = query.property(Person
let allOfStevesLastNames: [String] = try steveLastNamePQ.findStrings()
// if allOfStevesLastNames.contains("Jobs") { ... }
```

While the possibilities are not literally endless, they are plentiful if you combine query conditions with property-based requests.

Distinct and Unique Results

The property query can also only return distinct values, not producing duplicates. That means if you have 5 people in your database with ages `[25, 30, 30, 40, 40]`, you will get `[25, 30, 40]` as a distinct result set.

For example:

```
let names: [String] = try personBox.query().build()
    .property(Person.firstName)
    .distinct(caseSensitiveCompare: false)
    .findStrings()
```

If only a single value is expected to be returned the query can be configured to throw if that is not the case:

```
do {
    let singleNameResult: String = try personBox.query { Person.age > 999
        .property(Person.firstName)
        .findUniqueString()
} catch ObjectBoxError.uniqueViolation(let message) {
    print("Found more than one item! \(message)")
}
guard let singleNameResult = singleNameResult else {
    fatalError("No result found at all.")
}
```

You can combine `distinct()` with `findUnique()` .

Building Queries Traversing Relations

If your entity contains a relation to other entities, you can apply additional criteria to the related entities by using the `link(_ property:, conditions:)` method on your query.

E.g. if you had an entity `Order` that has a property `ToOne<Order, Customer>` pointing to the customer that placed the order, you could select all orders belonging to a particular customer from today's orders using

```
let sallysOrders = try orderBox.query{
        Order.date.isAfter(thisMorning)
    }.link(Order.customer) {
        Customer.name == "Sally Sparrow"
    }.build().find()
```

This is equivalent to what other databases call a `join` .

How do queries work?

Let's assume we have this interesting Entity:

```
class Person: Entity {
    var id: Id = 0
    var firstName: String
    var lastName: String
    var age: Int

    init(firstName: String, lastName: String, age: Int) {
        self.firstName = firstName
        self.lastName = lastName
        self.age = age
```

```
    }

    // ...
}
```

For this entity, the code generator will create something like this:

```
extension Person {
    static var id: Property<Person, Id, Void> { /* ... */ }
    static var firstName: Property<Person, String, Void> { /* ... */ }
    static var lastName: Property<Person, String, Void> { /* ... */ }
    static var age: Property<Person, Int, Void> { /* ... */ }
}
```

That's your entity property metadata. The code generator will create these static properties for you with the same name as a stored object's properties. This is what you use for queries.

The associated types of the generic `Property<Entity, Value, ReferencedType>` itself encode so much interesting information already to make the query interface very intuitive, for example:

- `Property<E, String, R>` exposes methods that make sense for strings, like `contains`
- `Property<E, Int, R>` exposes methods that make sense for numbers, including comparisons
- `Property<E, Date, R>` exposes methods for dates, like `isBetween(_:and:)`

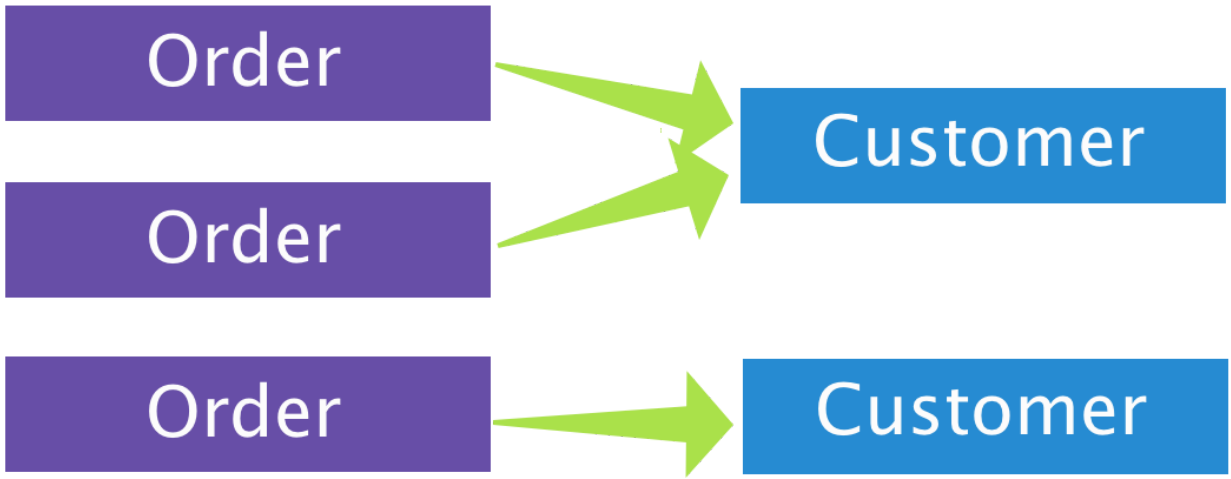
All of these factory methods create a `QueryCondition`, the type used for queries, and a lot of them have intuitive operator variants. Please refer to the API docs for a full list of operators and methods.

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 lazily initialized: the actual target objects are fetched from the database when they are first accessed. Once the target objects are fetched, they are cached for further accesses.

To-One Relations



To-One Relation

You define a to-one relation using the `ToOne` class, a smart proxy to the target object. It gets and caches the target object transparently.

For example, an order is typically made by one customer. Thus, we could model the `Order` class to have a to-one relation to the `Customer` like this:

```
class Customer: Entity {
    var id: Id = 0
    // ...
}

class Order: Entity {
    var id: Id = 0
    var customer: ToOne<Customer> = nil
    // ...
}
```

Given these entities and their to-one relation, you can create a relation and persist it:

```
let store = ...
// Illustrate that initially, nothing did exist
assert(try store.box(for: Customer.self).isEmpty())
assert(try store.box(for: Order.self).isEmpty())

let customer = Customer()
let order = Order()
order.customer.target = customer
let orderId = try store.box(for: Order.self).put(order) // puts order and

// Verify the `put` was called for the relation target as well
assert(try store.box(for: Customer.self).count() == 1)
assert(try store.box(for: Order.self).count() == 1)
```

> You can persist whole trees of object relations at once: If the customer object does not yet exist in the database, the `ToOne` will `put()` it. If it already exists, the `ToOne` will only create the relation (but not `put()` it).

For a comprehensive documentation of its features, visit the ToOne API docs.

Removing Relations

You have to set the target of a relation to `nil` and persist the changed object(s) via `Box.put()` to remove a relation permanently. Which one you reset doesn't matter, though. You have these options:

```
anOrder.customer.target = nil
// ... or ...
anOrder.customer.targetId = nil
// ... are both equivalent to:
anOrder.customer = nil
// ... whis is a short version of:
anOrder.customer = ToOne<Customer>(target: nil)
```

> Removing a relation never removes participating objects.

`ToOne` is a Lazy Relation Proxy

The target object of a relation is not eagerly loaded from the store; it is loaded lazily. Until you request the `ToOne.target`, it will not be read into main memory.

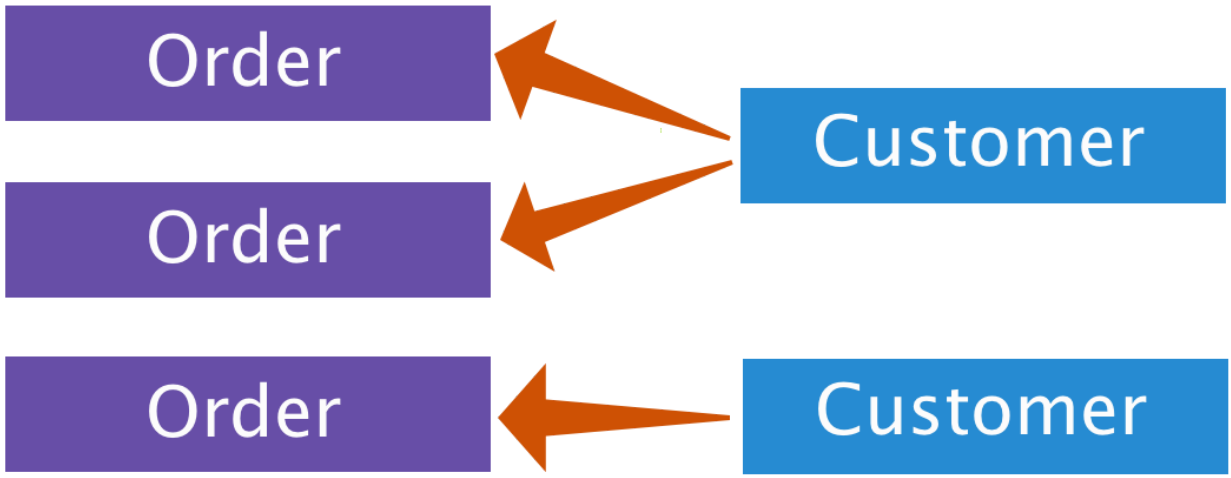
To-Many Relations

To define a to-many relation, you can use a property of type `ToMany`. Like the ToOne class, the `ToMany` class helps you to keep track of changes and to apply them to the database.

Note that **to-many relations are resolved lazily** on first access, and then **cached** in the source entity inside the `ToMany` object. So subsequent calls to any method, like the `count` of the `ToMany`, do not query the database, even if the relation was changed elsewhere. To avoid the cache and trigger a fresh reload from the database, call `reset()` on the `ToMany`.

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 Relations (1:N)



One-To-Many Relation

For every `ToOne` relation that you have, you can define a backlink. Backlinks are using the same relation information, but in the reverse direction. Thus, a backlink of a `ToOne` will result in a list of potentially multiple objects: all entities pointing to the same entity.
Example: Two Order objects point to the same Customer using a `ToOne`. The backlink is a `ToMany` from the Customer referencing its two Order objects.

Let's extend the example from above to get the backlinks from `Customer` to `Order`:

```
class Customer: Entity {
    var id: Id = 0
    // objectbox: backlink = "customer"
    var orders: ToMany<Order> = nil
    // ...
}

class Order: Entity {
```

```
    var id: Id = 0
    var customer: ToOne<Customer> = nil
    // ...
}
```

Note you tell ObjectBox about your backlink using an `// objectbox: backlink = "name"` annotation (where *name* is the name of the ToOne property that makes up the other end of the relation).

Once the backlink is set up, you can traverse the relation in both directions:

```
// Store two new orders for a new customer
let customer = Customer()
let order1 = Order(customer: customer)
let order2 = Order(customer: customer)
try store.box(for: Order.self).put([order1, order2])

// ID of customer was also set by put()
assert(customer.id != 0)

// Backlink: customer has two orders
assert(try store.box(for: Customer.self).get(customer.id).orders.count() =
```

> In database terminology, you create a (bi-directional) **one-to-many (1:N) relationship** by defining a ToOne with a backlink ToMany.

Collection Nature of `ToMany`

`ToMany` conforms to Swift's `RandomAccessCollection` protocol. Thus, you can use it just like an array or similar collections and pass it around. And of course, you can create an Array if need be:

```
let orders = Array(customer.orders)
```

See also: ToMany API docs

Modifying One-to-Many Relations

Apart from being a RandomAccessCollection, a `ToMany` relation is also a `RangeReplaceableCollection`. That means you can use `append(_:)` etc. on it just like on an `Array` to modify it. We've also added a `replace(_:)` method as a convenience to replace all entities referenced by the relation.

Once you've performed all the modifications you want, call `applyToDb()` on the `ToMany` to actually cause them to be written to the database. This separation into two steps prevents e.g. a filtering algorithm being called on the `ToMany` from triggering an unnecessailry huge number of database writes.

When you change its contents, `ToMany` will simply set the `ToOne` relation in the removed entities to `nil`, and will make added entities' `ToOne` point at the object containing the `ToMany` backlink. Note that you can only assign an entity to a relation that has already been assigned an ID (by putting it or loading an existing entity from the database):

```
let newOrder = Order(summary: "Shoes")
try orderBox.put(newOrder) // Must put or the replace() below will fail!
aCustomer.orders.replace([newOrder, oldOrder])
try aCustomer.orders.applyToDb()
```

Also, modifying a `ToMany` backlink modifies the `ToOne` of the referenced entities (in this example, `newOrder` and `oldOrder`) and will `put()` those objects to actually write out the changed relation. To remove all references to an entity, you may pass an empty array to `replace()`:

```
// You cannot set `aCustomer.orders = nil`, so:
aCustomer.orders.replace([])
try aCustomer.orders.applyToDb()
```

> Removing a relation never removes the referenced objects from the database.

Many-to-Many (N:M)

Many-to-Many (N:M)

To define a many-to-many relation you simply add a property using the `ToMany` class. Assuming a *students and teachers* example, this is how a simple `Student` class that has a to-many relation to `Teacher` entities can look like:

```
class Teacher: Entity {
    var id: Id = 0

    ...
}

class Student: Entity {
    var id: Id = 0
    var ToMany<Teacher> teachers = nil

    ...
}
```

Adding the teachers of a student works exactly like with an array, or a one-to-many relation:

```
let teacher1 = Teacher()
let teacher2 = Teacher()

let student1 = Student()
let student2 = Student()

```

```
try store.box(for: Teacher.self).put([teacher1, teacher2])
try store.box(for: Student.self).put([student1, student2])

student1.teachers.append(teacher1)
student1.teachers.append(teacher2)

student2.teachers.append(teacher2)

try student1.teachers.applyToDb()
try student2.teachers.applyToDb()
```

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

```
var student1 = try boxStore.box(for: Student.self).get(student1.id)
for (let teacher in student1.teachers) {
    ...
}
```

And if a student drops out of a class, we can **remove** a teacher:

```
student1.teachers.remove(at: 0)
try student1.teachers.applyToDb()
```

> Removing a relation never removes the referenced objects from the database.

Access Many-To-Many in the reverse direction

Following the above example, you might want an easy way to find out what students a teacher has. Instead of having to perform a query, you can just add a to-many relation to the teacher and annotate it with the `// objectbox: backlink` annotation:

```
class Teacher: Entity {
    var id: Id = 0
```

```
    // objectbox: backlink = "teachers"
    var students: ToMany<Student> = nil
    ...
}

class Student: Entity {
    var id: Id = 0
    var teachers: ToMany<Teacher> = nil

    ...
}
```

This will tell ObjectBox that there is only one relation, `teachers`, and that `students` is just a reverse-lookup of this relation. In any other respect, a many-to-many backlink can be used just like its forward counterpart.

Relations in Queries

You can traverse relations in queries.

Example: Modelling Tree Relations

You can model a tree relation with a to-one and a to-many relation pointing to itself:

```
class TreeNode: Entity {
    var id: Id = 0

    var parent: ToOne<TreeNode> = nil

    // objectbox: backlink = "parent"
    var children: ToMany<TreeNode> = nil
}
```

The generated entity lets you navigate its parent and children:

```
let parent = entity.parent.target
let children = Array(entity.children)
```

Transactions

ObjectBox - Transactions

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. For ObjectBox transactions continue reading:

You may not notice it, but almost all interactions with ObjectBox involve transactions. For example, if you call `put` a write transaction is used. Also if you `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 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

We learned that 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 class Store offers the following methods to perform explicit transactions:

- `runInTransaction`: Runs the given closure inside a transaction (comes in result-forwarding and/or throwing variants).
- `runInReadOnlyTransaction`: Runs the given closure inside a read(-only) transaction. Unlike write transactions, multiple read transactions can run at the same time.

The advantage of explicit transactions over the bulk put operations is that you can perform any number of operations and use objects of multiple boxes. In addition, you get a consistent (transactional) view on your data while the transaction is in progress.

Example for a write transaction:

```
try store.runInTransaction {
   try allUsers.forEach { user in
     if (modify(user)) {
       try box.put(user)
     } else {
       try box.remove(user)
     }
   }
}
```

Transaction Costs

Understanding transactions is essential to master database performance. If you just remember one sentence on this topic, it should be this one: a write transaction has its price.

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.

Consider this example:

```
try allUsers.forEach { user in
   modify(user) // modifies properties of given user
   try box.put(user)
}
```

Do you see what's wrong with that code? There is an implicit transaction for each user which is very inefficient especially for a high number of objects. It is much more efficient to use of of the put overloads to store all users at once:

```
try allUsers.forEach { user in
    modify(user) // modifies properties of given user
}
try box.put(allUsers)
```

Much better! If you have 1,000 users, the latter example uses a single transaction to store all users. The first code example uses 1,000 (!) implicit transactions causing a massive slow down.

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: an exception will be thrown.

While read transactions are much cheaper than write transactions, there is still some overhead to starting 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 `put` or `runInTransaction`), one of the threads will be selected to go first, while the other threads have to wait. It works just like a lock.

Locking inside a Write Transaction

Avoid locking (even implicitly by e.g. calling `DispatchQueue.sync` or explicitly using `NSLock` or `objc_sync_enter`) 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”.

FAQ

Conceptual

Is ObjectBox based on SQL/SQLite?

No, ObjectBox is a standalone database designed and optimized for storing objects. For example, there's no SQL, nor rows and columns. **Benefits:** simpler API and better performance.

Another big difference is that SQLite is an embedded database **only**. While ObjectBox also is an embedded database, it offers additional modes of operation like **client/server** and **data synchronization** among several machines - from small devices to servers and the cloud.

How is ObjectBox different from other solutions?

TL;DR Swift-first, performance, simple APIs

Swift first: The API is designed for Swift without an Objective-C legacy. For example there is no need for `@objc`, no enforced base class for entities, and no wrappers needed for optional types.
To give some background here, we started with a Objective-C(++) middle layer, but had this huge turning point during development. We realized that putting Swift first required us to take rather drastic measures and remove our Objective-C(++) code completely. We wrote about that and the immediate benefits like struct support and a huge performance boost.

Performance: ObjectBox was built with performance in mind and with minimalism as a leading principle. We believe we've built one of the most resource efficient databases that exist. Of course, you should not just rely on what we claim, but see for yourself. Our performance benchmarking code is open source.

Simple APIs: Browse through our docs using the menu on the left. We tried hard to give developers like you a simple API for powerful features. Another example are entities themselves, which are plain Swift objects without any "magic" that would break threading

or cause values from being shown in the debugger, etc. With ObjectBox, you can use objects across threads without restrictions. For example, you can defer loading objects to a background thread and pass them to the UI thread to display them.

And by the way; we want your feedback: of course, ObjectBox is not perfect. We set our goals high but we also fail at times. Let us know how we can improve. If you have something specific in mind, please raise an issue on GitHub. And we're also happy if you would send us general feedback on how we are doing (takes 1-2 minutes). Thank you!

Technical

Why are you not using Codable in ObjectBox?

ObjectBox uses a code generator, which does more than serializing objects into a binary format. It manages an internal data model and lets you e.g. rename a Swift property without breaking already stored objects. All of this happens at compile time, and gets checked into version control, so each user's and developers' databases are compatible. Codable happens at runtime, on each user's machine. Simply said, ObjectBox could not do what it does with Codable.

Troubleshooting

Error message "Missing argument for parameter 'model'..." when creating a new Store

TL;DR: build the project first.

ObjectBox's code generator generates a convenience initializer for you that doesn't need the model parameter because it sets up your model for you. It does this after parsing all your entities, which happens at build time. Thus, make sure to build your project at least once after declaring your entities. Otherwise Xcode's CodeSense auto-completion will not propose the right initializer in its list, and will display this erroneous error message.

Error Message "Could not open env for DB (1)" in Sandboxed macOS App

A sandboxed macOS application must have at least one *App Group* set up in its target settings under *Signing and Capabilities*. See The Sandbox on macOS for detailed instructions. This error occurs if no app group can be found.

Error message "ObjectBox does not contain bitcode" when linking

ObjectBox currently does not contain Bitcode, as it generally is not needed and only increases file sizes. You can still build for device and app store distribution by turning off Bitcode for your application as well. As the error message goes on to explain, you do this using the target's *Enable Bitcode* setting under *Build Settings* in Xcode (`ENABLE_BITCODE`).

If you have a strong need to obtain a version of ObjectBox with Bitcode, please let us know.

Advanced

Enums and Custom Types

Custom types allow entities to have properties of any type. This is based on mapping custom classes to built-in types. ObjectBox recognizes the following built-in (Swift) types:

```
Int8, UInt8
Int16, UInt16
Int32, UInt32
Int64, Int, UInt64, UInt
Float, Double
Data, [UInt8]
String
Date
```

Enums

ObjectBox has built-in support for `RawRepresentable` enums. All that is needed is an annotation to tell ObjectBox to convert an enum:

```
enum TestEnum: Int {
    case unknown = 0
    case first = 100
    case second = 200
    case third = 300
}

class EnumEntity: Entity {
    var id: Id = 0
    // objectbox: convert = { "default": ".unknown" }
    var custom: TestEnum
}
```

Specify a default value to use when the value in the database is missing or invalid as the `default` argument to the `// objectbox: convert` annotation.

A database can have a missing value even for non-optional fields if you add a property to your entity and then open a database that was written by an older version of your app that did not have that field yet. A database may contain an invalid value if an older version of your app opens a file created by a newer version of your app that has added a new case to this enum.

When persisting enums, there are a couple of best practices:

- **Always assign explicit rawValues to each enum case:** Implicitly-assigned `rawValue`s are unstable, and can easily change the next time you edit your enum definitions.
- **Prepare for the unknown:** Define an *unknown* enum value and specify it as the default. It can serve to handle missing or unknown values. This will allow you to gracefully handle cases like an old enum value getting removed without having to constantly unwrap optionals.

You can leave away the `"default"` annotation if your property is an optional. In that case `nil` will be the automatic default.

Enums in queries

`QueryBuilder` is unaware of enums. You have to **use the enum's rawValue for queries**.

So for the `EnumEntity` example above you would get users with the `custom` property of `second` with the query condition `box.query { User.role == TestEnum.second.rawValue }`.

Custom Property Converters

To add support for a custom type, you can map properties to one of the built-in types using an `// objectbox: convert` annotation. You also need to provide a class to serve as a property converter.

For example, you could define a color in your entity using a custom `Color` class and map it to a `String`.

Here is an example mapping an `enum` to an `Int` manually:

```
class RoleConverter {
    static func convert(_ enumerated: Role) -> Int {
        return enumerated.rawValue
    }
    static func convert(_ num: Int?) -> Role {
        guard let num = num else { return Role.default }
        return TestEnum(rawValue: num) ?? Role.default
    }
}

enum Role: Int {
    case default = 0
    case author = 1
    case admin = 2
}

class User: Entity, CustomDebugStringConvertible {
    var id: Id = 0
    // objectbox: convert = { "dbType": "Int", "converter": "RoleConverter
    var role = Role.default
}

```

Things to look out for

Be sure to **correctly handle nil** and **invalid values**: If you add a field to your entity later on, old records in a database will not have a value for this field. Your converter will be handed a `nil` value for those instead. Or if a user opens a database created with a newer version of your app that supports additional values for an enum with an older version that doesn't know about these, you will have to supply a fallback value from your converter (In the above example, those are the two `Role.default` returns in `convert(_: Int?)`).

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

If you implement your database-to-class converter method to not take an optional, ObjectBox will supply an appropriate 0 value for missing values, Same as it would for the underlying type without a converter.

> Note: Make sure the converter is thread safe, because it might be called concurrently on multiple entities.

List/Array types

At the moment it is not possible to use arrays with converters, apart from `[UInt8]`, which is treated like `Data`. However, you could convert a List of `String`s to a JSON array resulting in a single string for the database.

Alternately, you can replace arrays with relations and create a new entity for the array elements. So, for example instead of

```
class Document {
    var id: Id = 0
    var userNames: [String] // ObjectBox doesn't know how to store this
}
```

You would have two entities:

```
class Document {
    var id: Id = 0
    var users: ToMany<User>
}

class User {
    var id: Id = 0
    var document: ToOne<Document> = nil
    var name: String
}
```

Custom types in queries

`QueryBuilder` is unaware of custom types. You have to **use the raw database value for queries**.

So for the `User` example above you would get users with the role of `admin` with the query condition `box.query { User.role == Role.admin.rawValue }` or `box.query { User.role == 2 }`.

Data Model Updates

ObjectBox - Data Model Updates

ObjectBox manages its data model (schema) mostly automatically. The data model is defined by the entity classes 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 IDs (UIDs) and an `// objectbox: uid` annotation, as we will see below.

UIDs

ObjectBox keeps track of entities and properties by assigning them unique IDs (UIDs). All those UIDs are stored in a file “model.json”, which you should add to your version control system (e.g. git). If you are interested, we have in-depth documentation on UIDs and concepts. But let’s continue with how to rename entities or properties.

In short: To make UID-related changes, put an `// objectbox: uid` annotation on the entity or property and build the project to get further instructions.

Renaming Entities and Properties

So why do we need that UID annotation? If you simply rename an entity class, 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.

The same is true for properties.

Now let's walk through how to do that. The process works the same if you want to rename a property:

How-to and Example

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

```
// objectbox: entity
// objectbox: uid
class MyName { ... }
```

Step 2: Build the project. The build will fail with an error message that gives you the current UID of the entity/property:

```
error: No UID given for entity MyName. You can do the following:
  [Rename] Apply the current UID using // objectbox: uid = 17664
  [Change/Reset] Apply a new UID using // objectbox: uid = 18688

```

Step 3: Apply the UID from the [Rename] section of the error message to your entity/property:

```
// objectbox: entity
// objectbox: uid = 17664
class MyName { ... }
```

Step 4: The last thing to do is the actual rename on the language level (Java, Kotlin, etc.):

```
// objectbox: entity
// objectbox: uid = 17664
class MyNewName { ... }
```

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 `// objectbox: uid` annotation before renaming your entity/property.

Changing Property Types

When you want to change the type of a property, you must tell ObjectBox to create a new property internally. This is because ObjectBox cannot migrate your data, so simply changing the type may lead to data loss. You can do this in two ways:

- Just rename the property, so it is treated as a new property (this only works if the property has no `// objectbox: uid` annotation already):

```
// old:
var year: String
// new:
var yearInt: Int
```

- Tell ObjectBox to use a new UID for the property. Let's walk through how to do that:

How-to and Example

Step 1: Add the `// objectbox: uid` annotation to the property where you want to change the type:

```
// objectbox: uid
var year: String
```

Step 2: Build the project. The build will fail with an error message that gives you a newly created UID value:

```
error: No UID given for property year of entity MyName. You can do the fol
  [Rename] Apply the current UID using // objectbox: uid = 15616
  [Change/Reset] Apply a new UID using // objectbox: uid = 18688
```

Step 3: Apply the UID from the [Change/Reset] section to your property:

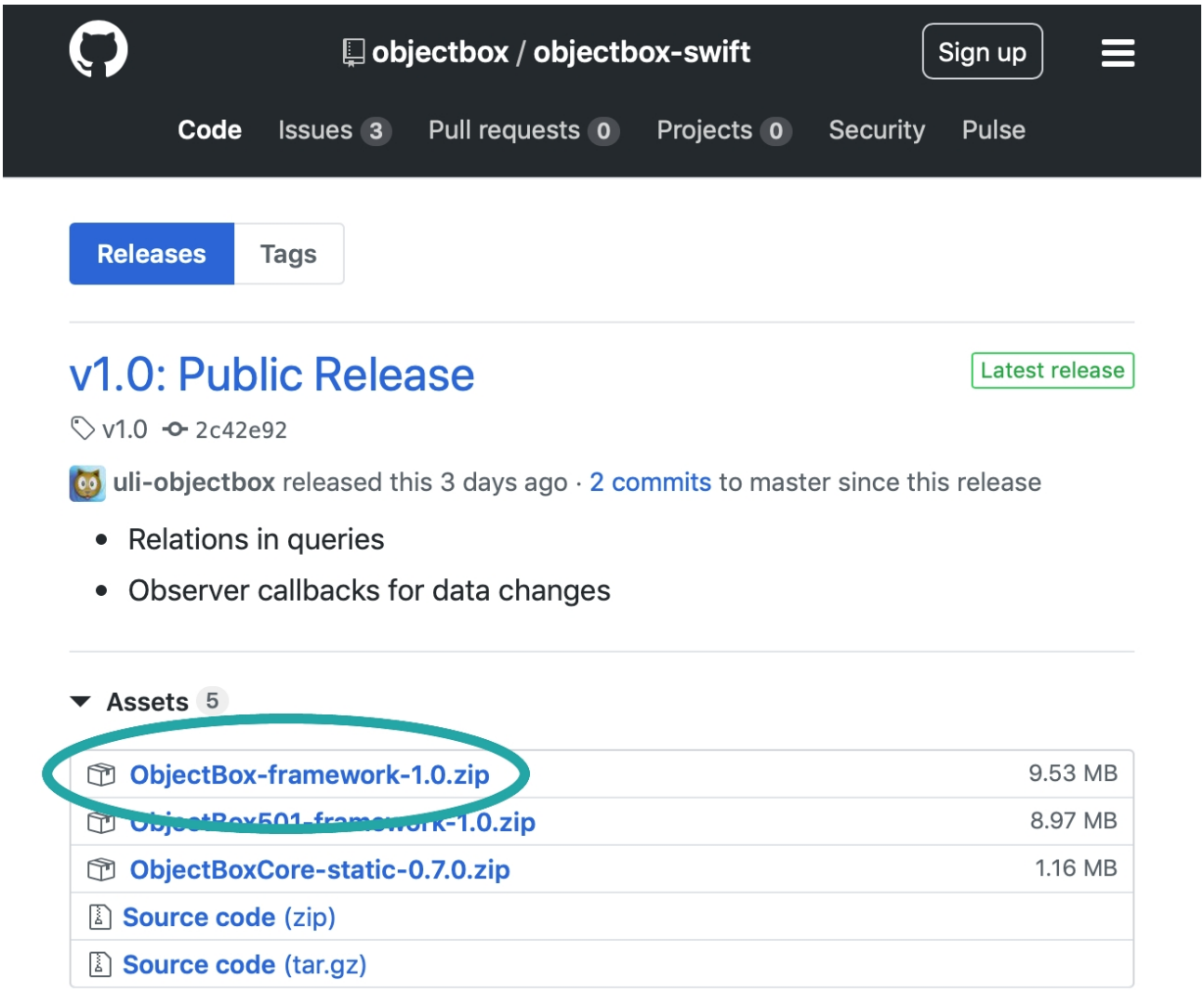
```
// objectbox: uid = 18688
var year: Int
```

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

Installation without CocoaPods or Carthage

Download

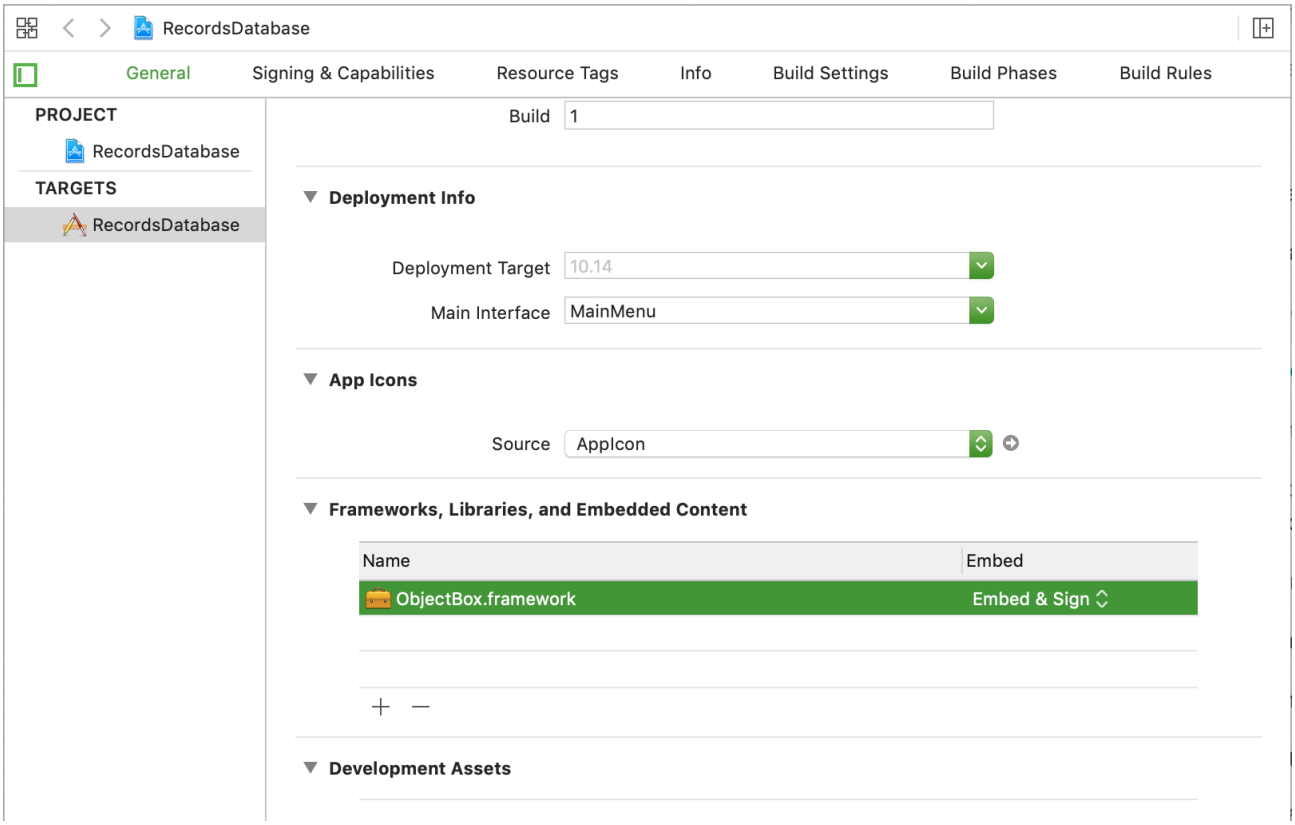
You can get the newest **ObjectBox-framework.zip** from our GitHub Releases page:



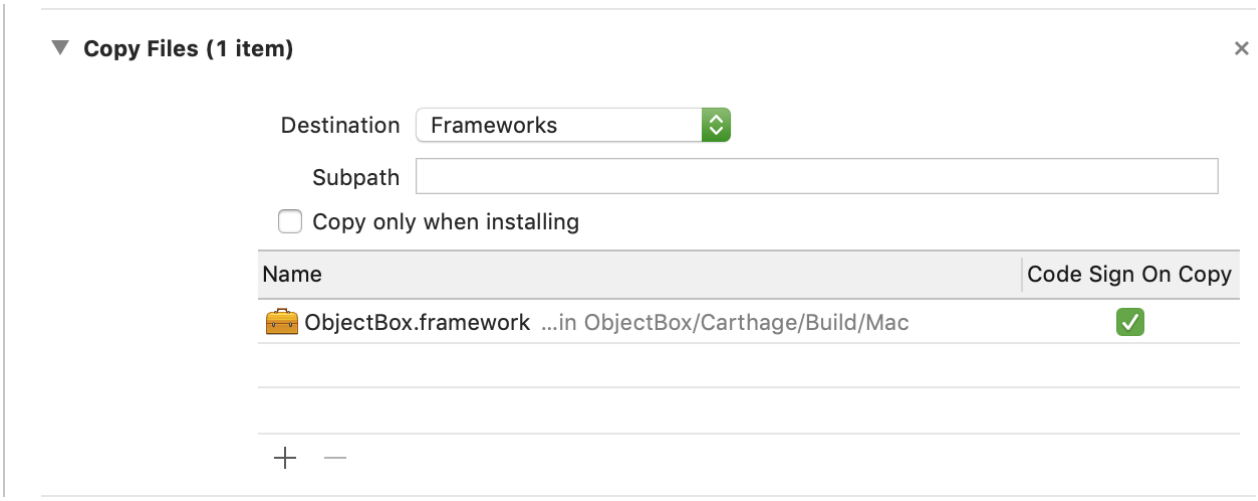
Unpack the ZIP-archive and put the entire folder somewhere in your project's folder (for example, if your project is in `/Users/vivien/RecordsDatabase/RecordsDatabase.xcodeproj`, you could put ObjectBox at `/Users/vivien/RecordsDatabase/ObjectBox`).

Add the Framework to your Project

Like with any embedded framework, you link to it and copy it into your application:



- Find the **ObjectBox.framework** for your platform from the `ObjectBox/Carthage/Build/` folder (e.g. `ObjectBox/Carthage/Build/iOS/ObjectBox.framework`).
- Open your project's target settings by clicking the little blue project icon at the top of the *Project* Navigator in Xcode.
- Select the *General* tab for your target and find the *Frameworks, Libraries and Embedded Content* section.
- Drag the ObjectBox.framework into the list and choose "Embed & Sign" from the popup at its right.
- macOS and Xcode 10 and earlier only:
  - Go to the Build Phases tab and add a *Copy Files* build phase.
  - Select *Frameworks* as the *Destination* from the popup.
  - Drag the ObjectBox.framework into it.



You're done, your project is linking to ObjectBox.

Setup Script

Then open Terminal and run

```
gem install xcodeproj
/path/to/ObjectBox/setup.rb /path/to/MyProject.xcodeproj
```

where `/path/to/` is again where your project is, like `/Users/vivien/RecordsDatabase/` above.

The first call installs a helper library needed by the installation script to do its work. This library is not needed once you have run the script. Nobody else using your project needs to do this.

Done

You can now open your project and follow the rest of this tutorial.

The Sandbox on macOS

The Sandbox on macOS

> You do *not* need to perform any of these steps for an application running on *iOS*, or if your macOS application is *not sandboxed*.

Currently, use of ObjectBox from a sandboxed *macOS* application requires you to set up at least one *App Group* for ObjectBox to be able to open a database, in addition to whatever you would normally have done to gain access to the destination folder under the sandbox:

1. Open your project in Xcode.
2. Select the project icon at the very top of the *Project Navigator*.
3. Select the *Signing & Capabilities* tab where you turned on *App Sandbox*.
4. Click the *+ Capability* button and double-click *App Groups*.
5. If the *App Groups* list is empty, click the little "+" icon to add one.

You should now see a list entry like `FGDTDLOBXDJ.$(TeamIdentifierPrefix)`. If you only see `$(TeamIdentifierPrefix)`, you still need to set up your team and code-signing under *General*.

That is all, ObjectBox will pick up the *App Group Identifier* without any additional work.

> If you're still using Xcode 10, *Signing & Capabilities* will just be called *Capabilities.*, and *App Groups* is a section you turn on with a switch, there is no *+ Capability* button.

Meta Model, IDs and UIDs

Unlike relational databases like SQLite, ObjectBox does not require you to create a database schema. That does not mean ObjectBox is schema-less. For efficiency reasons, ObjectBox manages a **meta model** of the data stored. This meta model is actually ObjectBox's equivalent of a schema. It includes known object types including all properties, indexes, etc. A key difference to relational schemas is that ObjectBox tries to manage its meta model automatically. In some cases it needs your help. That's why we will look at some details.

IDs

In the ObjectBox meta model, everything has an ID and an UID. IDs are used internally in ObjectBox to reference entities, properties, and indexes. For example, you have an entity "User" with the properties "id" and "name". In the meta model the entity (type) could have the ID 42, and the properties the IDs 1 and 2. Property IDs must only be unique within their entity.

> Note: do not confuse object IDs with meta model IDs: object IDs are the values of th `Id` or `EntityId<MyEntity>` property (see Object IDs in basics). In contrast, all objects are instances of the entity type associated with a single meta model ID.

ObjectBox assigns meta model IDs sequentially (1, 2, 3, 4, …) and keeps track of the last used ID to prevent ID collisions.

UIDs

As a rule of thumb, for each meta model ID there's a corresponding UID. They complement IDs and are often used in combination (e.g. in the JSON file). While IDs are assigned

sequentially, UIDs are a random long value. The job of UIDs is detecting and resolving concurrent modifications of the meta model.

A UID is unique across entities, properties, indexes, etc. Thus unlike IDs, a UID already used for an entity may not be used for a property. As a precaution to avoid side effects, ObjectBox keeps track of “retired” UIDs to ensure previously used but now abandoned UIDs are not used for new artifacts.

JSON for consistent IDs

ObjectBox stores a part of its meta model in a JSON file. This file should be available to every developer and thus checked into a source version control system (e.g. git). The main purpose of this JSON file is to ensure consistent IDs and UIDs in the meta model across devices.

This JSON file is stored in the file `model-TargetName.json`. For example, look at the file from the ObjectBox example project:

```
{
  "_note1" : "KEEP THIS FILE! Check it into a version control system (VCS)
  "_note2" : "ObjectBox manages crucial IDs for your object model. See doc
  "_note3" : "If you have VCS merge conflicts, you must resolve them accor
  "entities" : [
    {
      "id" : "1:712683617673955584",
      "lastPropertyId" : "2:5025387500910526208",
      "name" : "Author",
      "properties" : [
        {
          "flags" : 1,
          "id" : "1:6336800942024279296",
          "name" : "id",
          "type" : 6
        },
        {
          "id" : "2:5025387500910526208",
          "name" : "name",
          "type" : 9
        }
      ],
```

```
      "relations" : [

      ]
    },
    {
      "id" : "2:5608901830082711040",
      "lastPropertyId" : "6:6001769173142034944",
      "name" : "Note",
      "properties" : [
        {
          "flags" : 1,
          "id" : "1:7180411752564202752",
          "name" : "id",
          "type" : 6
        },
        {
          "id" : "2:249105953415333376",
          "name" : "title",
          "type" : 9
        },
        {
          "id" : "3:5661281725891017216",
          "name" : "text",
          "type" : 9
        },
        {
          "id" : "4:8342334437465755392",
          "name" : "creationDate",
          "type" : 10
        },
        {
          "id" : "5:8881960381068888832",
          "name" : "modificationDate",
          "type" : 10
        },
        {
          "flags" : 8,
          "id" : "6:6001769173142034944",
          "indexId" : "1:6069708401898380544",
          "name" : "author",
          "relationTarget" : "Author",
          "type" : 11
        }
      ],
      "relations" : [

      ]
    }
  ],
  "lastEntityId" : "2:5608901830082711040",
  "lastIndexId" : "1:6069708401898380544",
```

```
    "lastRelationId" : "0:0",
    "lastSequenceId" : "0:0",
    "modelVersion" : 5,
    "modelVersionParserMinimum" : 5,
    "retiredEntityUids" : [

    ],
    "retiredIndexUids" : [

    ],
    "retiredPropertyUids" : [

    ],
    "retiredRelationUids" : [

    ],
    "version" : 1
}
```

As you can see, the `id` attributes combine the ID and UID using a colon. This protects against faulty merges. When applying the meta model to the database, ObjectBox will check for consistent IDs and UIDs.

Meta Model Synchronization

At build time, ObjectBox gathers meta model information from the entities (classes that conform to the `Entity` protocol or are annotated `objectbox: Entity`) and the JSON file. The complete meta model information is written into the `generated/EntityInfo-TargetName.generated.swift` file.

Then, at runtime, the meta model assembled in `EntityInfo-TargetName.generated.swift` is synchronized with the meta model inside the ObjectBox database (file). UIDs are the primary keys to synchronize the meta model with the database. The synchronization involves a couple of consistency checks that may fail when you try to apply illegal meta data.

Stable Renames using UIDs

At some point you may want to rename an entity class or just a property. Without further information, ObjectBox will remove the old entity/property and add a new one with the new name. This is actually a valid scenario by itself: removing one property and adding another. To tell ObjectBox it should do a rename instead, you need to supply the property's previous UID.

Add an `// objectbox: uid` annotation without any value to the entity or property you want to rename and trigger a project build. The build will fail with a message containing the UID you need to apply to the `// objectbox: uid` annotation.

Also check out this how-to guide for hands-on information on renaming and resetting.

Resolving Meta Model Conflicts

In the section on UIDs, we already hinted at the possibility of meta model conflicts. This is caused by developers changing the meta model concurrently, typically by adding entities or properties. The knowledge acquired in the previous paragraphs helps us to resolve the conflicts.

The Nuke Option

During initial development, it may be an option to just delete the meta model and all databases. This will cause a fresh start for the meta model, e.g. all UIDs will be regenerated. Follow these steps:

- Delete the JSON file for the given target(s) (`model-TargetName.json`)
- Build the project to generate a new JSON file from scratch
- Commit the recreated JSON file to your VCS (e.g. git)
- Delete all previously created ObjectBox databases (e.g. for iOS, delete the app from your home screen, or delete the file from your application's container in the simulator or on macOS)

While this is a simple approach, it has its obvious disadvantages and is not an option once an app has been published.

Manual conflict resolution

Usually, it is preferred to edit the JSON file to resolve conflicts and fix the meta model. This involves the following steps:

- **Ensure IDs are unique:** in the JSON file the id attribute has values in the format “ID:UID”. If you have duplicate IDs after a VCS merge, you should assign a new ID (keep the UID part!) to one of the two. Typically, the new ID would be “last used ID + 1”.
- **Update last ID values:** for entities, update the attribute “lastEntityId”; for properties, update the attribute “lastPropertyId” of the enclosing entity
- **Check for other ID references:** do a text search for the UID and check if the ID part is correct for all UID occurrences

To illustrate this with an example, let's assume the last assigned entity ID was 41. Thus the next entity ID will be 42. Now, the developers Alice and Bob add a new entity without knowing of each other.

Alice adds a new entity “Ant” which is assigned the entity ID 42.

At the same time, Bob adds the entity “Bear” which is also assigned the ID 42.

After both developers committed their code, the ID 42 does not uniquely identify an entity type (“Ant” or “Bear”?). Furthermore, in Alice’s ObjectBox the entity ID 42 is already wired to “Ant” while Bob’s ObjectBox maps 42 to “Bear”.

UIDs make this situation resolvable. Let’s say the UID is 12345 for “Ant” and 9876 for “Bear”. Now, when Bob pulls Alice’s changes, he is able to resolve the conflict: He manually assigns the entity ID 43 to “Bear” and updates the lastEntityId attribute accordingly to “43:9876” (ID:UID). After Bob commits his changes, both developers are able to continue with their ObjectBox files.

Customizing Code Generation

Although you should generally not need to, you can adjust the behavior of ObjectBox's code generator by passing it additional arguments.

The general procedure is the same: Open the *[OBX] Update Sourcery Generated Files* build phase for your target in Xcode and edit the shell script there. Usually it looks something like:

```
"$PODS_ROOT/ObjectBox/generate_sources.sh" -- --output "$PROJECT_DIR/gener
                                              --model-json "$PROJECT_DIR/mode
```

where `TARGETNAME` is the name of your target.

Note the extra two dashes, which help the script distinguish arguments to be forwarded to the code generator from arguments directly to the script. You only need one double-dash at the start, even when passing multiple options to the script.

Controlling Output File Names

If you have several different targets in your project and want them to share the same model, you can customize the names and paths for the `model.json` and `EntityInfo.generated.swift` files by changing the script to use the same name for these files:

```
"${PODS_ROOT}/ObjectBox/generate_sources.sh" -- --output "${PROJECT_DIR}/g
                                                --model-json "${PROJECT_DI
```

In this example, `EntityInfo.generated.swift` and `model.json` without the target name in them.

Modifying the Module Name

Or if you wanted to control the Swift **module name** under which the generated source code should end up, you would do this by telling the script to forward the `--xcode-module` option to the code generator:

```
"${PODS_ROOT}/ObjectBox/generate_sources.sh" -- --xcode-module JennasModule
```

Where `JennasModule` would be the module name to use.

Specifying Access Control for Generated Code

If you are **writing a framework** and want to export your entity classes from it, you will need to make the generated code public. You do this similarly, by adding the `--visibility` option:

```
"${PODS_ROOT}/ObjectBox/generate_sources.sh" -- --visibility public
```

Anonymous Build Statistics

The code generator sends some anonymous usage statistics during build (on the machine you are building on, never inside your app!). If you do not wish this information to be collected, you can pass the following option to Sourcery:

```
"${PODS_ROOT}/ObjectBox/generate_sources.sh" -- --no-statistics
```

The following information is collected:

- An anonymous random GUID identifying the installation of ObjectBox
- The number of builds
- A hash of the target's namespace name
- OS version, ObjectBox version, Deployment OS, architecture and versions
- System country and language setting
- CI system the build is run under

The code generator will try to batch this information together so there is at most one request sent per day.

Other Options

It is not recommended to customize any other options, as they are subject to change, but should you need to do so, you can obtain a list by asking the code generator directly by typing the following into Terminal:

```
/path/to/ObjectBox/Sourcery.app/Contents/MacOS/Sourcery --help
```

Which will print a syntax description for the code generator.

→ Getting Started Using ObjectBox in Your Project /getting-started

The setup.rb Script

> If you are using the provided `setup.rb` script, **you do not need to do any of these steps**. This section is mostly of interest for people who want to know what that script does behind the scenes or who for some reason can not use the script.

ObjectBox uses a code generator to generate boilerplate code that passes information about your objects to the library. We chose code generation over fat inheritance trees so you can write plain Swift objects and just use these.

The `setup.rb` script does the following things to ensure the code generator is run transparently whenever you build your project:

1. Adds a build phase that runs the code generator to your project.
2. Adds the generated source files to your project

Adding the Sourcery Build Phase to Your Project

In your Xcode project, **add a "New Run Script Phase"** and make sure it comes *above* the Compile Sources phase of your project. The code generator needs to generate the code you want to compile first.

We call the Build Phase "[OBX] Update Sourcery Generated Files".

Enter the following script into this build phase:

```
"${PROJECT_DIR}/ObjectBox/generate_sources.sh"
```

The above example assumes you've extracted the files from the ObjectBox release on Github into a folder named `ObjectBox` next to your project. If you had a folder named `external` for all external dependencies next to a `myproject` folder that contains your project, that line might read:

```
"${PROJECT_DIR}/../external/ObjectBox/generate_sources.sh"
```

You get the picture. In general, it is a good idea to use project-relative paths, so anyone checking out your project can run it, no matter where they keep their checkouts, or what their username.

Adding the Generated Files to Your Project

Build your project once, so it can create the `generated/EntityInfo.generated.swift` file for you. It will not contain much of interest, yet, but now that the `generated/` directory and its contents exist, you can drag them into your Xcode project so they get compiled as well (make sure you add folders as "groups", not as "folder references").

You now have a working ObjectBox setup in your project and should be able to follow the rest of the instructions in our Getting Started section.