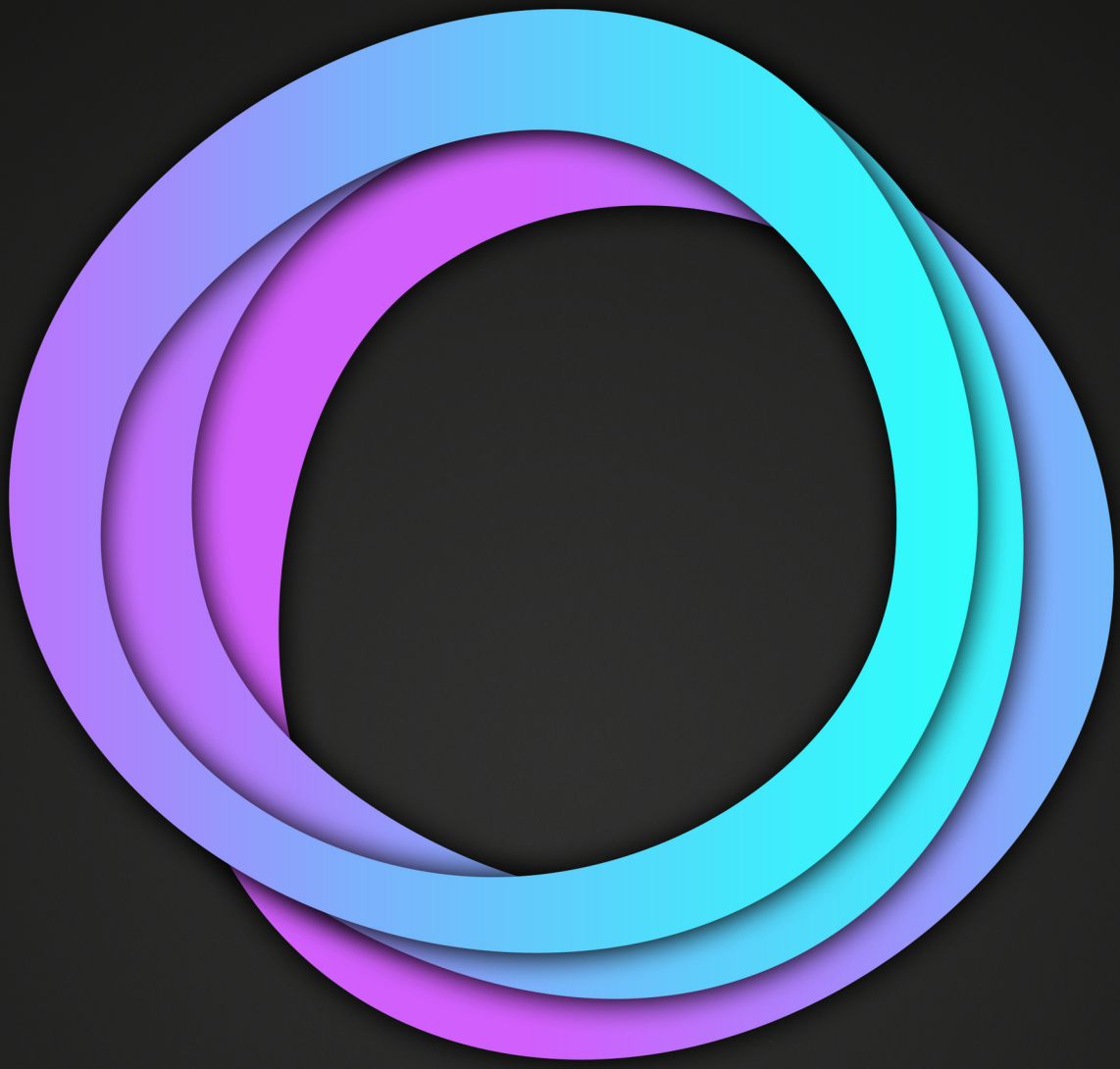
Core Data Synchronization

with



ensembles

Drew McCormack, PhD

Core Data Synchronization with Ensembles (BETA)

drewmccormack@mac.com

This book is for sale at http://leanpub.com/ensembles

This version was published on 2017-06-19

Contents

Acknowledgments

I would like to thank those in the Cocoa developer community who have contributed to sync technologies on Apple platforms, and have thereby helped me — directly or indirectly — to improve the Ensembles framework.

Milen Dzhumerov Tim Isted Marcus Zarra Christian Beer Charles Parnot Joel Grasmeyer Steve Tibbett Michael Fey Kevin Hoctor Daniel Pasco Dave Verwer (iOS Dev Weekly) Chris Eidhof and Daniel Eggert (objc.io) Chris Price (iOSDevUK) Steve Scott (NSConference) Niklas Saers (GOTO Conference)

And all of the other developers who have given Ensembles a try, and provided constructive feedback.

Special thanks to Marcello Luppi (Wrinkly Pea Design) for preparing the artwork and icon of Ensembles.

Preface

Late in 2013, I announced at iOSDevUK[1] that I was working in my spare time on a new open source, sync framework for Core Data: Ensembles. A month after that, while rehashing the presentation at GOTO Aarhus[2] in Denmark, I pushed the first working source code to GitHub[3].

It had been a long journey, and there was still a long way to go. I had been struggling for years to replace the Wi-Fi sync in my Mental Case[4] product with a more contemporary cloud-based solution. Mental Case deals with non-trivial amounts of data, and a moderately complex data model; the existing options for Core Data synchronization, including Apple's iCloud, had fallen frustratingly short of the mark.

Although I did eventually ship Mental Case 2 with cloud sync, built upon the TICDS framework[5], I had never been completely happy with the result. TICDS was a brave first attempt by its developer, Tim Isted, at a solution to the cloud sync problem at a time when no other options existed. But it pre-dated even iCloud, had lost its creator to Apple, and was already starting to show its age.

I contributed code to the TICDS project, and considered whether it could be updated to incorporate design changes that we now know work better. In the end, I decided it would involve such widespread changes, it was probably more work than just starting again, with the latest Objective-C language goodness, and the knowledge garnered from TICDS, iCloud, and other attempts to solve the problem.

And so, back in April 2013, I started hacking away on the beginnings of a framework codenamed 'Syncophant'. I didn't have a clear direction: the design had not fully crystallised. After a month or two, I realised that I would never get the framework to any stable state without good tests, so I started developing each component using Xcode 5's new unit testing framework.

As I went, I developed clear ideas about how things should work in a peer-to-peer synchronization framework like Ensembles. I started to see where other frameworks were going wrong, and also where they were doing things right. I took the best, and looked for ways to avoid the worst.

In this way, I came to develop a set of informal requirements and specifications for the framework. These became the essence of the Ensembles framework, and they are covered in the introductory chapters.

In March 2014, I announced Ensembles 1.0 at NSConference[6]. The ideas had crystalized, the tests passed, and there were already a few apps shipping with it. In short, it was ready for production.

You could argue that Ensembles is at a disadvantage with respect to the incumbent iCloud—Core Data framework, with Apple holding all the cards. It is true Apple has access to private APIs, and the credibility that comes with being Apple. Luckily, the Core Data API is complete enough to develop Ensembles without the need for private APIs, and Ensembles will hopefully establish its own form of credibility as it ships in more and more apps.

[1] http://www.iosdevuk.com
[2] http://gotocon.com/aarhus-2013/
[3] https://github.com/drewmccormack/ensembles
[4] http://www.mentalcase.com
[5] https://github.com/nothirst/TICoreDataSync
[6] http://nsconference.com

And there are also advantages that a small open source project like Ensembles has over a major company like Apple. Agility for one. Apple is restricted in how often it can issue updates to frameworks, while a bug can be fixed in Ensembles in a matter of hours, and in your app the next day.

Apple are also restricted in what they can include in their offering. You should not hold your breath for Dropbox support in Core Data, while Ensembles includes it out-of-the-box, and can support virtually any new backend with a few hours of work.

Adopting Ensembles is an investment in the future. It's a solid product, with nothing to hide, and can adapt as the market place evolves. Contrast that to opaque commercial offerings that lock you in to one service, and the decision is made palpably simpler.

Introduction

- Book is about how to use Ensembles framework
- What is Ensembles?
- Sync for Core Data apps
- Introducing ensembles covers generally how it works, and design goals
- Covers installing and integrating
- Aspects of setting up your model for sync
- Covers more advanced topics (conflict resolution)
- Various backends

What is this Book About?

- An open source framework for syncing Core Data stores
- Only useful on Apple systems
- Only useful if you use Core Data

What Will Not be Covered?

- Assume understanding of Core Data

Introducing Ensembles

Before digging into Ensembles itself, it is worth taking a peek from a distance. Establish in your head the design goals of the project, and get a general feel for how it goes about the task of data synchronization. That's what this chapter is about. What is Ensembles all about?

If you don't like to mess around, jump straight to the next chapter to sink (pun fully intended) your teeth into installing and making use of Ensembles in your apps.

What is Ensembles?

The name

The name 'Ensembles' may seem like an odd choice for a sync framework. It derives from the central concept upon which the framework is based: the *ensemble*.

A set of syncing persistent stores is called an *ensemble*, because it is like a group of musicians playing together. Each participant is an individual, but through an exchange of information — musical notes — they strive to form a cohesive whole.

The objective of the Ensembles framework is the same. Each persistent store stands alone, but there is an interchange with other stores to achieve conformity. This high-level organization of persistent stores is an *ensemble* of stores.

The framework

Ensembles is a framework for backend-agnostic, peer-to-peer synchronization of Core Data persistent stores. That's a very succinct explanation, so let's break it down a bit.

Backend agnostic refers to the fact that Ensembles can be made to work with any server or cloud service that is capable of transferring files (eg iCloud, Dropbox, FTP, Amazon S3). In fact, it goes a bit further than that, because it is also possible to make it work with pure peer-to-peer communication such as Bluetooth or Local WiFi. For example, there is a backend included for the Multipeer Connectivity framework, which doesn't make use of any server at all.

Ensembles includes backends for major services like iCloud and Dropbox, but it is relatively easy to add a new backend for an unsupported service, or even a custom server. You just need to implement a protocol (`CDECloudFileSystem`), which includes file operations like uploading, downloading, and deleting. This typically takes a few hours, at which point Ensembles should be capable of syncing via the service.

The last term that needs some explanation is *peer-to-peer*. Didn't I just say an online service is often involved? That is true, but Ensembles places all of the burden of maintaining data consistency on the client Macs and iOS devices. Any service involved in transferring files can be seen more as being part of the data transport network — it needs no insight into the content of the files. In this sense, I like to think of the framework as being peer-to-peer, even if it isn't always strictly true.

Ensembles 1.x

The goal in developing Ensembles was initially to develop a robust framework, but not necessary an optimal one. Ensembles 1.x can be used to sync most of the standard apps found in the App Store, but for complex models or large data sets, may not be suitable. On occasion it may load the full data store from disk, and on constrained memory devices like iPhones, this can sometimes be a problem.

The original Ensembles 1.x project is still supported and maintained. What's more, it is fully open source, and can be downloaded directly from GitHub.

The well-established MIT[7] licence has been applied. The MIT licence is commonly used for iOS and Mac open source projects. It is a very liberal licence; in short, you can take and modify the code, ship it in commercial products, submit it to the iTunes App Store, and all without having to release any source code yourself. You don't even have to release any changes you make to Ensembles.

What you do have to do is include the MIT licence somewhere on your web site or in your app. A common choice is to add it in an About box, which most apps include.

The Ensembles 1.x project on GitHub is a great way to try out Ensembles, and — in many cases — will be a perfectly good solution to your sync problem. If it isn't, there is always…

Ensembles 2

Ensembles 2 is a newer variant of Ensembles which has been largely rewritten to dramatically reduce memory usage, increase performance, and reduce cloud storage usage. It is a drop-in replacement for Ensembles 1.x; if you have your app syncing with Ensembles 1.x, it should be very little work to adopt Ensembles 2.

Although the API for Ensembles 2 is very similar to Ensembles 1.x, the two are not binary compatible. They use different file formats, so if you are migrating your Ensembles 1.x app to Ensembles 2, you create a new Ensemble for the new framework. They should not refer to the same cloud data.

The Ensembles 2 framework is not released under an open source license; it is only available when purchasing[8] a support package from The Mental Faculty B.V.[9]. Some of the packages include full source code, but the code may not be distributed to third parties.

Design Goals

As I developed Ensembles, I built up a set of design goals and requirements. These helped guide the project in the right direction, and it is useful to understand these goals to better appreciate how the framework works, and why it works that way.

Requires minimal changes to existing code

There is nothing in Core Data that should necessitate major changes to your data model, or class hierarchy, in order to support synchronization. The `NSManagedObjectContext` class already fires notifications before and after each save, with all of the information necessary to form change sets.

[7] http://opensource.org/licenses/MIT
[8] http://ensembles.io
[9] http://mentalfaculty.com

A major goal of Ensembles was that the API should be as simple as possible, and that it should be as non-invasive as possible. You should not need to subclass `NSManagedObjectContext` or `NSManagedObject`. You should not have to alter your model.

In stark contrast to other solutions, you also shouldn't be compelled to alter the Core Data stack (*eg* suddenly have to remove your `NSPersistentStore`) just to accommodate Ensembles. Your `NSManagedObjectContext` should proceed unhindered, even when Ensembles has no connection to the cloud, or a catastrophic problem arises, such as the user switching cloud accounts. Syncing may terminate, but your app should go forth as though nothing happened.

And when your app is ready to reconnect to the cloud, Ensembles automatically migrates data to the cloud, so again, you are not required to play musical chairs with store files, and artificial migrations between stores, like you often do with other packages.

Peer-to-Peer

Technically, Ensembles is not a peer-to-peer system, because servers are often used to store data. However, the *intelligent* work is all handled by the client framework, and the servers are really only used as a data transfer network, so Ensembles is closer to peer-to-peer than a standard client–server architecture. This is made even more apparent by the fact that it can also perform a traditional peer-to-peer sync, with no server involvement at all.

Backend agnostic

The framework should work with any system capable of syncing up blobs of data (*ie* files) located at paths. Examples include, but are not limited to, iCloud, Dropbox[10], S3[11], OmniPresence[12], WebDav[13], FTP[14], Wi-Fi, and Bluetooth.

Future Proof

Because with Ensembles you have access to the source code, and it is not married to any particular backend technology or service, it is as close to future proof as you can expect of any framework. If you integrate Ensembles, you have control over the source code, and the backend. If your backend ceases to exist, there are plenty more to choose from, and you can even evolve as new services come online.

Files in the cloud are immutable

Ensembles ensures that when it adds a file to the cloud, it never moves it, or changes the file contents. Many of the issues that arise when using services like iCloud come from mutating a file on multiple devices. Making files immutable makes it much less likely that problems will be encountered.

[10] http://dropbox.com
[11] http://aws.amazon.com/s3/
[12] http://www.omnigroup.com/omnipresence/
[13] http://en.wikipedia.org/wiki/WebDAV
[14] http://en.wikipedia.org/wiki/File_Transfer_Protocol

Real time testing

Real time testing is essential for development, and for running automated tests. To achieve this goal, Ensembles supports backends that allow near instant transfer, such as the local file system and the pasteboard.

One of the great difficulties with testing other sync frameworks is that testing your app is mind-numbingly frustrating. You make a change on one device, and wait for it to propagate to another device where you can observe the effect of your change. This process can be minutes long.

Eventual consistency of data across client devices

Having data be consistent across devices sounds like an obvious goal of a sync algorithm, but it is actually more difficult than it seems. We are dealing with a decentralized, peer-to-peer synchronization model, where no device can assume it has the complete global state at any point in time.

When merging a change set from a different device, it is often necessary to reconsider changes from a change set that has already been merged, in order to guarantee eventual consistency. Files can also arrive out of order, so an instruction to delete an object could appear before the object has even been created!

In order to ensure all devices apply and compare changes in the same order, whenever a new set of changes is first stored, the known revisions of all other devices are included. This forms a vector clock[15], which allows Ensembles to establish exactly which change sets occurred concurrently, and provide a global ordering.

A centralised, server-based synchronization model, by contrast, such as the newly introduced Dropbox Datastore[16], can make simplifying assumptions([1][17][2][18]) about what data needs to be included in merge operations.

Conflict resolution handled in the spirit of Core Data

Conflicts arise when changes are made on two different devices without an intervening sync operation. Conflict resolution is tricky. You can flag every conflict, and require the host app to resolve each one. Or you can try to handle conflicts automatically, offering no insight or influence over what is happening. Ensembles adopts a completely new model of conflict resolution, tailored very much to the design ethos of Core Data.

It is common practice for Core Data developers to use a `NSManagedObjectContext` as a scratch pad for temporary data edits. Data can be manipulated, and is only required to be in a valid state when it is saved to the store. If a validation problem arises, it is possible to fix it, and retry.

This is a common pattern in Core Data apps, where a temporary context gets setup just for the purpose of making changes that may or may not end up committed to the store. Ensembles adopts this same approach for merging changes from other devices.

A temporary background `NSManagedObjectContext` is created which shares the same SQLite store as the main context. Change sets are merged into this context, in order, with no regard to validity of the object graph. When it is time to commit the changes, a delegate method is invoked to give the host app an opportunity to apply

[15] http://en.wikipedia.org/wiki/Vector_clock
[16] https://www.dropbox.com/developers/datastore
[17] https://www.dropbox.com/developers/blog/48/how-the-datastore-api-handles-conflicts-part-1-basics-of-offline-conflict-handling
[18] https://www.dropbox.com/developers/blog/56/how-the-dropbox-datastore-api-handles-conflicts-part-two-resolving-collisions

any *repairs* that it wants to make before the data is sent to disk. A second delegate method is called if the background save fails, offering another opportunity to make repairs. Any repairs made by the application code are *captured* and added as a change set.[19]

Many data models never require repairs, even after conflicts arise. For others, such as those with strict validation rules, repairs may be needed, but even then, they are usually localized to specific parts of the data model.

Persistent store never in an invalid state

Ensembles allows you to stipulate the global identities of objects via a delegate callback. This has broad implications. It means that Ensembles can take on the responsibility of automatically adding data to the cloud when a store is first synchronized. It also means that data is automatically de-duplicated when logically identical objects are added on two different devices. This, and the repair mechanism discussed above, means that the persistent store should never be in an invalid state, and certainly not by design.

Other frameworks, such as Core Data's built in iCloud support, use a different approach. You have to de-duplicate objects yourself — after the changes have already been committed to the persistent store. Ensembles only commits to the persistent store once changes are known to be valid, and there are no duplicates.

Large data blobs handled efficiently

Many apps these days store image, audio, and video files as binary `NSData` attributes in Core Data. Ensembles handles this data efficiently on low memory devices, and without duplication in the cloud. Large binary objects are stored in external files by the framework.

Small cloud footprint

In order to sync, Ensembles must store files in the cloud. The size of this data is on the same order as the size of the store that is being synchronized, but every effort has been made to keep the footprint small. There is no duplication of data, and files are compact.

Ensembles also uses a technique called *baselining* to clean up old, redundant data. Baselining involves compacting cloud files into a single, new baseline. The new baseline contains only the insertion transactions needed to create a clone of the database.

Ensembles does this automatically when it detects that there is a significant reduction in data to be had. Ensembles is smart about how the baseline is stored, keeping data to a minimum, and avoiding making wholesale copies of the SQLite database.

Graceful handling of model versions

Ensembles records the model version used to create each change set, so when merging changes from other devices, it can determine whether it can process the changes. If not, it gives an error and waits for the user to update the app. In the meantime, it continues to record any changes made by the user, and, after updating, sync continues with no data lost.

[19] This whole process is analogous to how a developer uses a DVCS like Git. Typically, you pull new versions from a server, and merge them with your local changes. If conflicts arise, you repair them, and commit these new changes, before pushing all local changes back to the server.

Objective-C and Swift

Ensembles is written in Objective-C, but works fine with the new Swift language. Whether you are developing in Objective-C, Swift, or both, you can use Ensembles to add sync to your Core Data app.

How Does it Work?

You should not need to know in detail how Ensembles synchronizes data stores in order to use it, but it certainly helps to have an idea of what it is doing. This understanding makes it easier to integrate the framework, and understand issues if any arise.

This section will give you a high level picture of how the framework operates, without going into any of the code level details.

How it doesn't work

You may wonder why a sync framework is needed for Core Data at all. Couldn't we just upload the data store to the cloud whenever it changes, and download it on other devices?

There are a few problems with this that make it impractical. First, uploading the whole data store whenever a save occurs would result in excessive data transfer. This could be a big issue, particularly on mobile devices.

Second, it would be very difficult to merge the data if changes were made on two devices at roughly the same time. A new store may appear in the cloud from another device while the app is already running. You would have to find a way to load the new store, merge it with the existing store, and replace the local store all while the app is launched.

For these reasons, Ensembles and other frameworks like it take a different approach to sync.

Transaction Logs

Ensembles breaks down each persistent store into a series of transactions. The transactions represent all of the insertions, updates, and deletions that go into building the store's database.

An insertion or update transaction is like an `NSDictionary`. It contains key-value pairs of an object's property names and values. For example, for a `Car` class, we may represent an object insertion like this JSON code

```
{
    "type" : "insert",
    "id" : "CAR12345",
    "manufacturer" : "Mazda",
    "model" : "626",
    "year" : 2003,
    "owner" : "OWNER54355"
}
```

As you can see, the attributes of the class are stored as numerical or string values. Relationships, like the car's owner, are stored as identifiers that can be used to retrieve the related objects.

An update would be very similar to an insertion, but would only include the properties that changed.

```
{
    "type" : "update",
    "id" : "CAR12345",
    "owner" : "OWNER23368"
}
```

In this example, the owner of the vehicle has been changed.

When an object is deleted, only the identifier of the object is needed, since the properties are irrelevant.

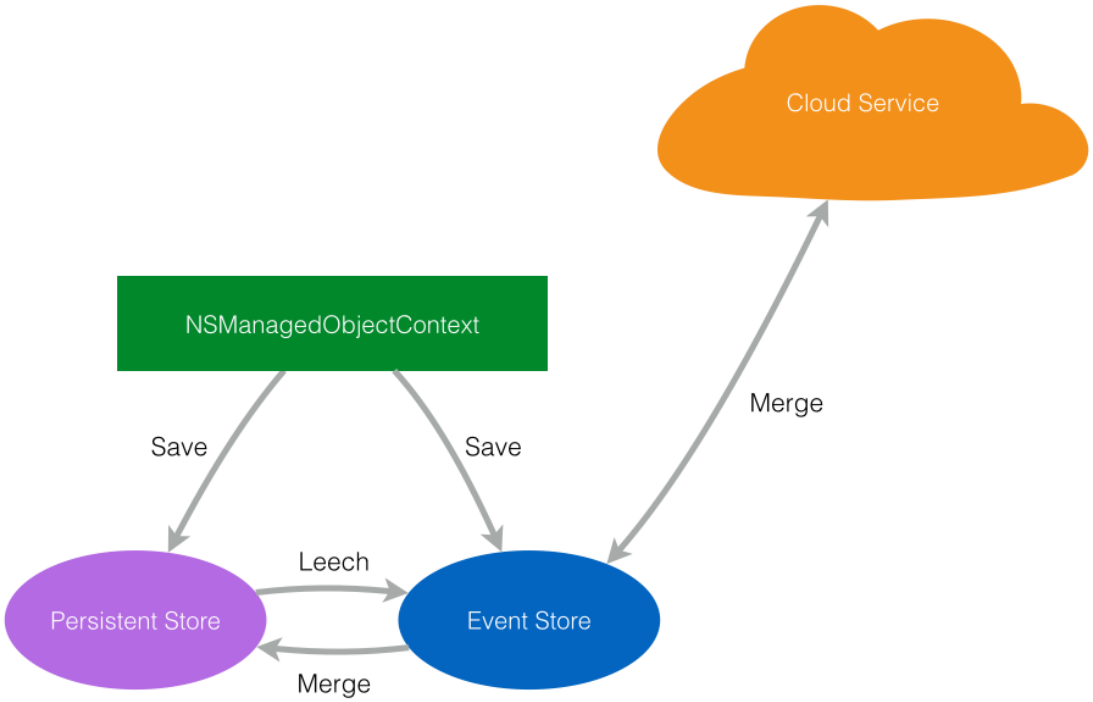
```
{
    "type" : "delete",
    "id" : "CAR12345"
}
```

There are typically many of these changes made in a single save to the data store. All of these transactions get bundled up in files known as *transaction logs*. A timestamp is applied to each set of changes, so that they can be ordered later, and the files are what get uploaded to the cloud.

In other words, Ensembles records the changes to the data store — the so-called *deltas* — rather than copying the whole data store itself. By only transmitting the changes, the quantity of data transfered is considerably less, and it is also easier to combine changes from different devices: they just get *played back* in the same order as their timestamps dictate.

The Big Picture

Now that the basic building blocks of the system are known (*i.e.* the transaction logs), we can consider the architecture of Ensembles, and flow of data through the system.



Architecture and data flow in Ensembles.

Ensembles extends the existing `NSManagedObjectContext` and `NSPersistentStore` of an app with a cloud service that is used to transport the transaction logs, and an *event store*. The event store is a local cache of the transaction logs; it is more optimal for working with the transactions than the individual files.[20]

Data flows between the various components as shown by the arrows in the figure. When the persistent store is first registered — a process known as *leeching* — its data is converted into insertion changes in the event store.

During a save of the main context, data is transferred to the persistent store as usual, but Ensembles also observes the save, and converts the changes into transactions in the event store.

When requested by the application code to merge the changes from other devices into the persistent store, Ensembles first retrieves new transaction logs from the cloud service and adds those to the event store. It then plays back the transactions, inserting, updating, and deleting entries in the persistent store as required. Merging also uploads locally created transactions to the cloud service so that other devices can merge them.

[20] A set of transactions arising from some modification of the persistent store, such as a save operation, is called an *event*. This explains the naming of the *event store*.

The Event Store

As stated earlier, the event store is a local cache of transaction logs. It contains metadata used by the framework, and a Core Data SQLite store for the transactions. By importing the transactions into a single SQLite store, rather than accessing them directly in disparate transaction log files, merging can be made much more efficient.[21]

Cloud File Systems

Ensembles is backend agnostic, so it can be made to work with many different cloud services. Each backend is represented by a class that conforms to the protocol `CDECloudFileSystem`. Backends are required to store data for file paths, much like a file system, hence the term *cloud file system*.

Although cloud file systems behave like simple file systems, there is no requirement that they actually be implemented as such; for example, you could create a cloud file system that stores data in a key-value store, with the paths used as keys.

Leeching

Before a persistent store can be synced with other stores, the ensemble must *leech* it. *Leeching* is the process of registering the store with the other peers in the ensemble, and uploading the initial content of the local persistent store.

A store typically only needs to be leeched once — it is persisted across app launches. You can explicitly *deleech* a store, if it should no longer be synced with its peers.

A spontaneous (forced) deleech can also occur in circumstances where the persistent store is permanently prevented from syncing, such as when the user logs in to a different account. Depending on the events that led to the deleech, you may be able to leech immediately to start syncing again.

Merging

Merging is what most people would think of as syncing. It involves retrieving new data from other devices, merging it with local changes, applying those changes to the local store, and uploading any new transactions from the local device.

Baselining

Baselining is the process of cleaning up old data. It only needs to run occasionally. When the framework detects that there is considerable data redundancy in the transaction logs, it compacts them into a new *baseline* file, which only includes the minimum number of (insertion) changes needed to rebuild the store.

[21] Having a separate event store can cause Ensembles to use more data storage than strictly necessary, but the performance benefits generally outweigh the storage concerns.

Ensembles *versus* The Rest

There are a number of other sync solutions available for Core Data developers. This section compares some of the more commonly-used with Ensembles.

Core Data's Native iCloud Support

The most obvious competitor to Ensembles is Apple's own built-in support for iCloud in Core Data. When this was first announced in 2011, many developers — including yours truly — were excited to try it. Unfortunately, the framework was buggy and poorly documented for the first two years, and this led many developers to give up in frustration.

iOS 7 and OS X 10.9 brought much needed improvements, and the framework seems much more stable than it was. However, even when Apple's solution works well, there are many issues which Apple — by nature of its size and agenda — will never be able to address. I'll enumerate some of them here.

The biggest problem with adopting the built-in sync is that it is closely coupled to iCloud. If you don't want to use iCloud, want to support multiple different backends, or are unable to use iCloud because your app is not in the App Store, you are out of luck. Apple don't support extensions, and are unlikely to do so in future.

Because Core Data only supports iCloud as a backend, you can't have multi-user sync. Ensembles has no restrictions in this regard; if you have a backend that manages multiple users, you can have them share and sync stores.

Core Data is a proprietary framework, and you don't have access to the source code. When an issue arises, it can be very difficult to determine what is going wrong. If you discover a bug, it is very likely to be a year before Apple can address the issue in the next major system upgrade. In an project like Ensembles, bug fixes can be issued in a matter of hours.

The debugging facilities for iCloud in Xcode don't allow for local testing, so you always have to push data via iCloud when developing. This can slow things down considerably, and make setting up automated tests much more difficult. Ensembles supports syncing via the local file system, and includes test suites built upon this feature.

The design of the Core Data–iCloud API makes it troublesome to adopt. There is an overly tight coupling between the Core Data stack and the sync functionality, so you will often find persistent stores being added and removed by the framework as your app runs.

Core Data will not migrate your data to the cloud, or do any merging of data. Instead, you have to deduplicate data after it has already been incorporated in the persistent store, using fetch requests to locate corresponding objects. Ensembles gives you the ability to provide global identifiers for objects, which are used to automatically migrate and merge data.

Lastly, Core Data provides no hooks to resolve conflicts. If a conflict occurs, the behavior is undocumented. Will the data be deleted? Will the transaction be rolled back? It is not clear, and the developer has no control over the outcome. Ensembles includes delegate methods where you can 'repair' objects invalidated by conflicting changes.

CloudKit

Apple introduced a new option for cloud storage at WWDC 2014: CloudKit. CloudKit is a promising technology, supporting not only data transfer between a user's devices, but also sharing of data between users. It opens up a whole range of possibilities for app developers.

Although, in theory, you can sync Core Data stores with CloudKit, there is no built in support, and you will need to implement a lot of the sync algorithms yourself. In effect, you will be reimplementing large parts of the Ensembles framework, including local change tracking, mapping of cloud records to Core Data managed objects, and conflict resolution. Where it may only take 100 lines of code to get your app syncing with Ensembles, it may take several thousand lines to achieve the same with CloudKit.

Tim Isted Core Data Sync (TICDS)

Before Ensembles, and even before iCloud support was added to Core Data, Tim Isted developed an open source framework known as TICDS[22]. This framework blazed a trail for later projects like Ensembles, and is still used in quite a few apps today.

In technical terms, TICDS is very similar to Ensembles, supporting Core Data sync via file transfer backends like Dropbox and iCloud. Like Ensembles, it can also be extended to support any backend capable of transferring file data.

Unfortunately for the TICDS project, Tim has moved on to work at Apple, and is unable to continue to develop it. The project is entering a state of gradual deterioration. The design of certain parts of the TICDS framework have also been improved upon in Ensembles, aided by the lessons learnt from it.

Wasabi Sync

Wasabi Sync[23] is a hosted service for Core Data apps. It is well regarded, and used in apps like Bare Bone's Yojimbo[24]. Unfortunately, you can no longer purchase a new developer account for Wasabi Sync.

Simperium

Simperium[25] is a service that arose out of the Simplenote[26] app. It has a Core Data interface, though it does not seem to be well maintained.

Simperium is a cross platform cloud solution. Being a hosted solution, you have to pay for online storage.

Dropbox Datastore API

Recently, Dropbox introduced its Datastore API[27]. By adopting their SDK for your model layer, you can have your data mirrored across devices.

The Datastore API doesn't support Core Data directly, but the open source ParcelKit[28] project offers a bridge.

[22] https://github.com/nothirst/TICoreDataSync
[23] http://www.wasabisync.com
[24] http://www.barebones.com/products/yojimbo/
[25] http://simperium.com
[26] http://simplenote.com
[27] https://www.dropbox.com/developers/datastore
[28] https://github.com/overcommitted/ParcelKit

An advantage of the Datastore API over other providers is that the developer doesn't have to pay for storage. Dropbox customers have already paid for storage, and your app can leverage it free of charge.

Parse

Parse[29], which is now owned by Facebook, gives you an SDK similar to Dropbox's Datastore API. A major difference with Dropbox is that you have to pay hosting costs.

Parse does not support Core Data directly, but there are open source projects[30] to bridge the gap.

As with any service that requires you to adopt a new API, you cannot easily migrate away from Parse without rewriting your model code.

BaasBox

BaasBox[31] is an open source variant of Parse. You can run your own server, or pay for hosting.

BaasBox is cross platform, but doesn't have any Core Data integration. Like Parse and the Datastore API, you have to adopt a new API for your model objects.

Realm.io

Realm[32] is a new cross platform data store, with support for sync. It is off to a very promising start, but makes no use of Core Data — it is a replacement for Core Data. Moving to Realm means moving away from Core Data; if you want to stick with Core Data, Ensmebles offers a robust way to sync your data across devices.

[29] https://parse.com
[30] https://github.com/itsniper/FTASync
[31] http://www.baasbox.com
[32] http://realm.io

Installing Ensembles

Before integrating Ensembles into your Xcode project, you need to download it. There are various forms the framework comes in, and different ways to get it. This chapter describes how you can download and install Ensembles.

Downloading Ensembles

Migrating from Ensembles 1.x to Ensembles 2

Ensembles 2 is a drop-in replacement for Ensembles 1.x — the API has been extended, but is fully compatible — so it's easy to start development with Ensembles 1.x, and move to Ensembles 2 before shipping.

If you already have a shipping app using Ensembles 1.x, you need to be aware that although Ensembles 2 has an API that is compatible with Ensembles 1.x, it is not binary compatible. In particular, the cloud file storage format is changed, so you should not mix Ensembles 1.x and Ensembles 2 cloud data. The easiest way to migrate is simply to deleech, remove the cloud data, and create a new Ensemble under a different name (e.g. MainStore.v2).

Support Package Downloads

When you purchase a support package for Ensembles, it includes source code for the framework, as well as easy-to-install binaries. Just unzip the package, and navigate to the appropriate directory.

Cloning Ensembles with Git

If you like to stay up-to-date with the latest changes to the source code, you can use Git and the GitHub web site to install and manage Ensembles.

To clone the Ensembles 1.x repository to your local drive, use the command

```
1  git clone https://github.com/drewmccormack/ensembles.git
```

If you are using Ensembles 2, you will first need to request access to the private GitHub repository. Once you have access, you can clone using the command

```
1  git clone https://github.com/mentalfaculty/ensembles-next.git ensembles
```

Ensembles makes use of Git submodules. To retrieve these, change to the `ensembles` root directory in Terminal

```
cd ensembles
```

and issue this command

```
git submodule update --init
```

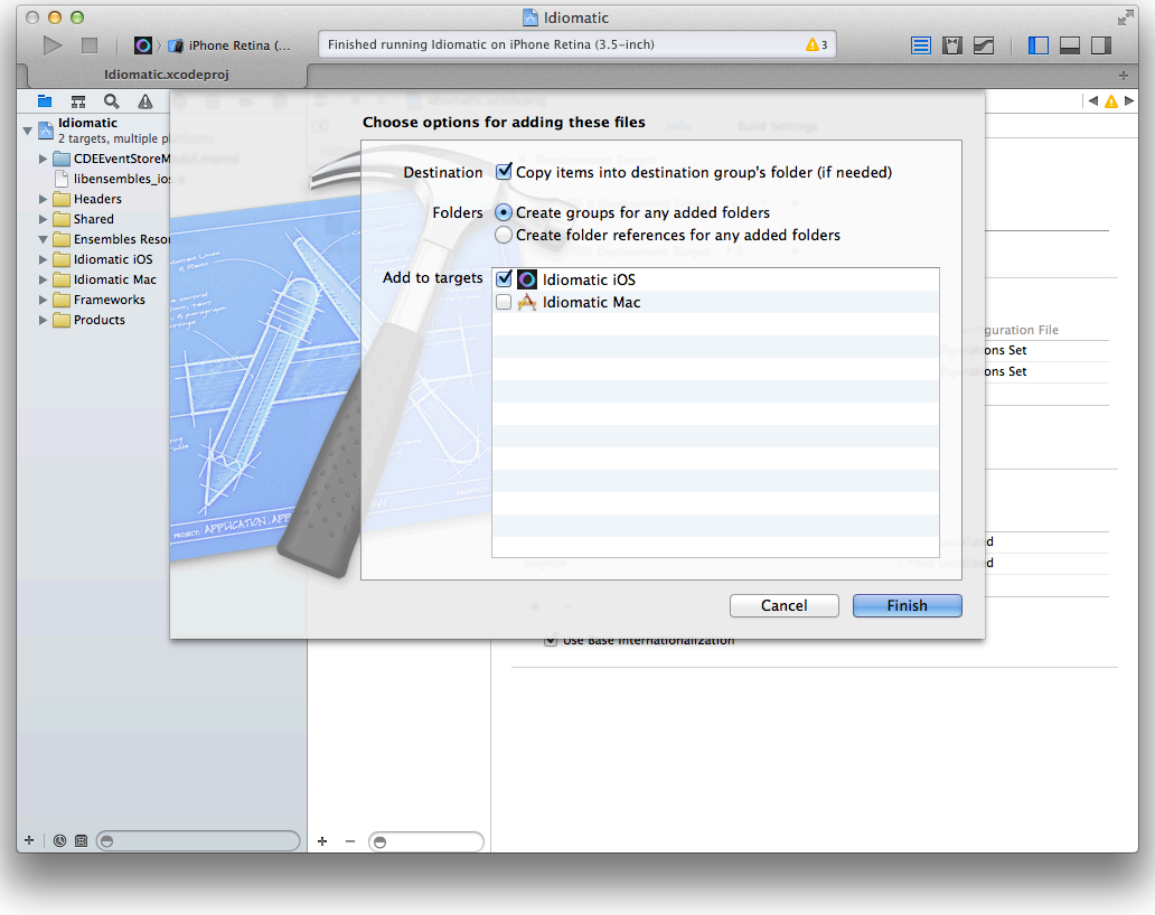
If you think you may want to contribute source code back to the Ensembles project at some point, it may be worth forking the project on GitHub, and cloning your fork instead of the main repository. This will allow you to issue pull requests on GitHub.

Integrating Ensembles into an Xcode Project

Installing the Binaries

If you purchased a support package for Ensembles, it should include ready-made binaries for the framework, which make installing as easy as drag-and-drop. If you have the binaries, here is how you can integrate them into your Xcode project.

1. Drag the framework bundle *Ensembles.framework* from the iOS or OS X folder into your Xcode project source list.
2. In the sheet that appears, make sure you check the *Copy items into destination group's folder* option, make sure your app target is checked, and then click *Finish*.



Adding the library.

3. For an iOS app, drag the resources bundle *Ensembles.bundle* into your Xcode source list. Make sure you check the *Copy items into destination group's folder* option, and check the box for the app's target.
4. For a Mac app, create a new build phase to copy frameworks into your app bundle (if you don't already have one). To do this…
   A. Select the project root in the source list, then select your app's target.
   B. Open the *Build Phases* tab.
   C. Click the + button at the top of the list.
   D. Choose *New Copy Files Build Phase* from the popup menu.
   E. Disclose the contents of the new *Copy Files* phase, and choose *Frameworks* from the *Destination* popup button.
   F. Click the + button at the bottom of the *Copy Files* phase section, choose *Ensembles.framework*, and click *Add*.
5. For an iOS project, select the *Build Settings* tab of the app target. Locate the *Other Linker Flags* setting, and add the flag *-ObjC*.
6. For a Mac app, locate the *Runpath Search Path* build setting, and add `@loader_path/../Frameworks`.
7. If you need to install other backends, such as Dropbox, drag the relevant files from the *Extra Backends* folder into Xcode.

Using CocoaPods for Ensembles 1.x

CocoaPods[33] is a package management tool that has become popular amongst iOS developers. Ensembles includes support for CocoaPods, making installation a breeze. Here is how you add Ensembles 1.x to your App's Xcode Project with CocoaPods.

Add the following to your project's Podfile

```
platform :ios, '7.0'
pod "Ensembles", "~> 1.0"
```

This will install Ensembles with the iCloud backend.

To use other cloud services, such as Dropbox, add the relevant subspec to the Podfile. For example, to include Dropbox, include

```
pod "Ensembles/Dropbox", "~> 1.0"
```

CocoaPods subspecs supported in Ensembles 1.x.

Identifier	Supported Backends
Ensembles/Core	Local file system and iCloud.
Ensembles/Dropbox	All in Core, and Dropbox Core API.
Ensembles/Multipeer	All in Core, and Multipeer Connectivity.
Ensembles/Node	All in Core, and Node.js server supplied with some support packages.

Using CocoaPods for Ensembles 2

To use CocoaPods for Ensembles 2, you first need to request access to the private GitHub repository.

Once you have access, you can add the private repository of The Mental Faculty to your Cocoapods installation.

```
    pod repo add mentalfaculty https://github.com/mentalfaculty/Specs.git
```

Once you have done that, you can include Ensembles 2 in your Podfile.

[33] http://cocoapods.org

```
source 'https://github.com/mentalfaculty/Specs.git'
source 'https://github.com/CocoaPods/Specs.git'

platform :ios, '7.0'
pod "Ensembles", "~> 2.0"
```

Including optional backends is the same as for Ensembles 1.x, except that there are more to choose from.

CocoaPods subspecs supported in Ensembles 2.

Identifier	Supported Backends
Ensembles/Core	Local file system and iCloud.
Ensembles/CloudKit	All in Core, and CloudKit.
Ensembles/Dropbox	All in Core, and Dropbox Core API.
Ensembles/WebDAV	All in Core, and WebDAV.
Ensembles/Multipeer	All in Core, and Multipeer Connectivity.
Ensembles/Zip	All in Core, and Zip compression.
Ensembles/Encrypt	All in Core, and encryption of cloud data.
Ensembles/Node	All in Core, and Node.js server supplied with some support packages.

Manually Installing in Xcode (iOS)

If you like to control all aspects of installing the source code yourself, follow the following procedure to add Ensembles to your iOS App's Xcode Project.

1. In Finder, drag the `Ensembles iOS.xcodeproj` project from the `Framework` directory into your Xcode project.
2. Select your App's project root in the source list on the left, and then select the App's target.
3. In the General tab, click the + button in the *Linked Frameworks and Libraries* section.
4. Choose the `libensembles.a` library and add it.
5. Select the *Build Settings* tab. Locate the *Other Linker Flags* setting, and add the flag `-ObjC`.
6. Select the *Build Phases* tab. Open *Target Dependencies*, and click the + button.
7. Locate the `Ensembles Resources iOS` product, and add that as a dependency.
8. Open the `Ensembles iOS.xcodeproj` project in the source list, and open the Products group.
9. Drag the `Ensembles.bundle` product into the *Copy Bundle Resources* build phase of your app.
10. Add the following import in your precompiled header file, or in any files using Ensembles.

```
#import <Ensembles/Ensembles.h>
```

By default, Ensembles only includes support for iCloud. To use other cloud services, such as Dropbox, you need to locate the source files and frameworks relevant to the service you want to support. You can find frameworks in the *Vendor* folder, and source files in *Framework/Extensions*. More information on installing other backends is available in the chapter on standard backends.

By way of example, if you want to support Dropbox, you need to add the DropboxSDK Xcode project as a dependency, link to the appropriate product library, and include the files *CDEDropboxCloudFileSystem.h* and *CDEDropboxCloudFileSystem.m* in your project.

Manually Installing in Xcode (OS X)

Follow the following procedure to add Ensembles to your OS X App's Xcode Project.

1. In Finder, drag the `Ensembles Mac.xcodeproj` project from the `Framework` directory into your Xcode project.
2. Select your App's project root in the source list on the left, and then select the App's target.
3. In the General tab, click the + button in the *Linked Frameworks and Libraries* section.
4. Choose `Ensembles.framework` and add it.
5. Create a new build phase to copy frameworks into your app bundle (if you don't already have one). To do this…
   A. Select the project root in the source list, then select your app's target.
   B. Open the *Build Phases* tab.
   C. Click the + button at the top of the list.
   D. Choose *New Copy Files Build Phase* from the popup menu.
   E. Disclose the contents of the new *Copy Files* phase, and choose *Frameworks* from the *Destination* popup button.
   F. Click the + button at the bottom of the *Copy Files* phase section, choose *Ensembles.framework*, and click *Add*.
6. Locate the *Runpath Search Path* build setting, and add `@loader_path/../Frameworks`.
7. Add the following import in your precompiled header file, or in any files using Ensembles.

```
#import <Ensembles/Ensembles.h>
```

To use cloud services other than iCloud, locate the frameworks in the *Vendor* folder, and source files in *Framework/Extensions*.

Code Signing on OS X

When you come to distribute your Mac OS X app, you will likely want to code sign it. Embedded frameworks, like Ensembles, also have to be code signed.

The latest versions of Xcode allow you to check a box in the Copy Build Phase to sign the framework when installing. If you don't have this option, or want to control code signing yourself, you can instead add a new *Run Script* build phase to your target. Make sure it is the last build phase, and add the following script.

```
LOCATION="${BUILT_PRODUCTS_DIR}"/"${FRAMEWORKS_FOLDER_PATH}"
RESOURCES_LOCATION="${BUILT_PRODUCTS_DIR}"/"${UNLOCALIZED_RESOURCES_FOLDER_PATH}"
IDENTITY="<Name of your code signing certificate here>"

codesign --verbose --force --sign "$IDENTITY" "$LOCATION/Ensembles.framework/Versions/A"
codesign --verbose --force --sign "$IDENTITY" "$RESOURCES_LOCATION/Ensembles.bundle"
```

You will need to fill in the name of your code signing certificate for the `IDENTITY` variable. In practice, it doesn't really matter what identity you use to sign the framework. It seems that Mac OS X requires that the embedded frameworks be signed, but doesn't care by whom.

Idiomatic App

Idiomatic is a relatively simple example app which incorporates Ensembles and works with a selection of different backends to sync across devices. The app allows you to record your ideas, takes photos, and add tags for grouping. The Core Data model of the app includes three entities, with many-to-many and many-to-one relationships.

The Idiomatic project is a good way to get acquainted with Ensembles, and how it is integrated in a Core Data app. Idiomatic can be run in the iPhone Simulator, or on a device, but in order to test it, you need to follow a few preparatory steps.

Supporting iCloud Sync

To support iCloud sync in Idiomatic, follow these steps.

1. Select the Idiomatic Project in the source list of the Xcode project, and then select the Idiomatic target.
2. Select the *Capabilities* section, turn on the iCloud switch.
3. Build and install on devices and simulators that are logged into the same iCloud account.

Add notes, and tag them as desired. The app will sync when it becomes active, but you can force a sync by tapping the button under the Groups table.

Supporting Dropbox

Dropbox should work via The Mental Faculty account, but if you want to use your own developer account, you need to do the following:

1. Sign up for an account at the Dropbox Developer Site[34].
2. In the App Console, click the *Create app* button.
3. Choose the Dropbox API app type.
4. Choose to store *Files and Datastores*

[34] http://developer.dropbox.com

5. Choose *Yes — My app only needs access to files it creates*
6. Name the app (*eg Idiomatic*)
7. Click on *Create app*
8. At the top of the *IDMSyncManager.m* file, locate this code, and replace the values with the strings you just created on the Dropbox site.

```
NSString * const IDMDropboxAppKey = @"xxxxxxxxxxxxxxx";
NSString * const IDMDropboxAppSecret = @"xxxxxxxxxxxxxxx";
```

1. Select the Idiomatic project in Xcode, and then the Idiomatic iOS target.
2. Select the Info tab.
3. Open the URL Types section, and change the URL Schemes entry to

```
db-<Your Dropbox App Key>
```

Deploying Ensembles Server

Some support packages include access to a custom server written with Node.js[35] and backed by Amazon's S3 online storage. You can deploy this web service on Heroku[36] if you want a custom sync service.

What is Ensembles Server?

Ensembles Server is a basic Node.js web app that utilizes Amazon's S3[37] for file storage, and a Postgres database for user management. It can be used with the `CDENodeCloudFileSystem` class in the Ensembles framework on client devices.

The server offers HTTP Basic Authentication, and an API that supports user actions such as signing up, logging in, changing password, and resetting a password (emails new password). The Idiomatic iOS app gives an example of how an interface for these user actions can be setup, and integrated with the `CDENodeCloudFileSystem` class.

The server also has APIs used by `CDENodeCloudFileSystem` to upload and download files to/from S3. The Node.js server doesn't handle file transfers directly; rather, it returns time-limited signed URLs, and the `CDENodeCloudFileSystem` uses these URLs to exchange data directly with S3.

[35] http://nodejs.org
[36] http://heroku.com
[37] http://aws.amazon.com

Prerequisites

You will need:

- An Amazon AWS account[38], and a bucket in S3 called 'ensembles-server'
- An account at Heroku.com[39]

You can also host your Node.js server on cloud services other than Heroku, but the instructions may differ some.

Installing a Development Mac

- Download and install the Heroku Toolbelt[40]
- Download and install Node.js from nodejs.org[41]
- Download and install Postgres.app[42]

Running Ensembles Server on Development Mac

Setup your environment by editing your `.profile` or `.bash_profile` configuration file. Include your AWS credentials, and the path to the Postgres database tools.

```
export AWS_ACCESS_KEY_ID=<Your AWS Key>
export AWS_SECRET_ACCESS_KEY=<Your AWS Secret>
export PATH=$PATH:/Applications/Postgres.app/Contents/Versions/9.3/bin
export NODE_ENV=development
```

At this point, you should 'source' the configuration file, or exit the shell and launch a new one.

Ensembles Server stores its data in a bucket on Amazon's S3. Create a bucket for the server using the AWS web interface, and then fill in the bucket name for `process.env.BUCKET` in the `server.js` file. (Note there are two places this is set.)

Now to create a database. Launch Postgres.app, and then issue this command in Terminal.

```
createdb ensembles-server
```

Install the Node.js packages by changing to the root directory of the Ensembles Server project, and issuing this command

[38] http://aws.amazon.com
[39] http://heroku.com
[40] https://toolbelt.heroku.com
[41] http://nodejs.org
[42] http://postgresapp.com

```
npm install
```

Now you can run the server on your Mac. Just issue this command

```
foreman start
```

This will launch the Node.js server, and listen on port 5000 of the Mac.

You can dispatch HTTP requests using CLI tools like `curl`, and Mac apps like Paw and Rest Client. Use URLs beginning `http://localhost:5000`.

If you would like to test Idiomatic with your development server, locate the method `makeCloudFileSystem` in the `IDMSyncManager` class, and change the base URL used to initialize the `CDENodeCloudFileSystem` to `http://localhost:5000`

In practice, you are probably best just moving straight on to using Idiomatic with Heroku, which is described next.

Setting Up Heroku

If you haven't already created an S3 bucket for your app, create one now using the AWS web interface, and then fill in the bucket name for `process.env.BUCKET` in the `server.js` file.

Apps are installed on Heroku via Git, so you need to setup your Ensembles Server installation as a Git repository. Open Terminal and issue the following commands from the root directory of the Ensembles Server code base.

```
git init
git add .
git commit -m "First commit"
```

Login to Heroku on the command line.

```
heroku login
```

Accept if it asks if you want to generate an SSH key.

Create an app on Heroku with a unique name.

```
heroku apps:create <Name of App>
```

Include Heroku as a remote repository, so that you can push the Ensembles Server code to it. Use the name you chose above.

```
git remote add heroku git@heroku.com:<Name of App>.git
```

Add a PostgreSQL database.

```
heroku addons:add heroku-postgresql:dev
```

Add SendGrid[43], which is used to email new passwords. (You may need to supply Heroku with a credit card in order to add this package, even if you are using a free plan.)

```
heroku addons:add sendgrid
```

Set the Amazon AWS credentials in the Heroku environment.

```
heroku config:set AWS_ACCESS_KEY_ID=<Your Access Key> AWS_SECRET_ACCESS_KEY=<Your Secret K\
ey>
```

Configure Heroku as your production environment.

```
heroku config:add NODE_ENV=production
```

Push your code to Heroku with Git.

```
git push heroku master
```

You can check what dynos are running on Heroku using

```
heroku ps
```

To start a single dyno, run this command

```
heroku ps:scale web=1
```

To check for errors, issue this command

```
heroku logs
```

You should now be able to change the base URL used by the `CDENodeCloudFileSystem` in Idiomatic, or your own app, to use the URL of your Heroku app.

[43] http://sendgrid.com

Merging

Merging involves retrieving new data from other devices, combining it with recent local changes, and updating the persistent store. It is what would typically be considered a *sync* operation.

This chapter describes the process of merging in some detail, and discusses various strategies for when to perform a merge.

What Happens During a Merge?

A single merge actually involves many smaller steps. We'll go through these steps in this section.

Importing new Transaction Logs

Ensembles first checks for new files in the cloud. These get downloaded, and then imported into the *Event Store*, which is a local cache for the transaction logs.

Consolidating Baselines

A baseline represents an initial state for the store. It is the set of all insertions needed to fully form the store.

Usually there is only one baseline. To get the current persistent store, you would playback all the changes in the baseline, and then play back any extra transaction logs that have appeared since the baseline was formed.

Sometimes there can be more than one baseline. For example, this occurs when a new device joins the ensemble, and uploads the contents of its local store. When this happens, you end up with the original baseline, plus the newly uploaded baseline.

To get back to having just one baseline, Ensembles needs to consolidate the two baselines into one. Sometimes this is straightforward, such as when one baseline represents an earlier version of the other. In a case like this, the most recent baseline is kept, and the other discarded.

In other cases, the two baselines are unrelated, and need to be merged. This involves going through all the changes in each baseline and picking the most recent for each object. Effectively, we are taking the union of the two baselines. Where there is overlap, changes get merged.

After consolidating baselines, there should be only one baseline again.

Rebasing

After consolidation, we know there is only one baseline, but it might not be very recent. If it is too old, there may be a lot of new transaction logs, and a lot of data redundancy. This results in excessive cloud storage, and less efficient performance, because the framework has to process the data even if it is irrelevant.

Rebasing involves propagating the existing baseline forward in time, coalescing transactions, in order to form a new baseline.

Ensembles uses rough heuristics to estimate whether it is worthwhile to rebase. If there is a saving in cloud data storage of 50% or more to be had, it will rebase. It will also rebase if the number of cloud files is getting excessive.

Consistency Checks

Ensembles tracks the versions of every transaction log file, so it can detect when there are files missing. After the rebasing stage, a number of checks are carried out to ensure there are no important log files missing. If there are some missing log files, an error will be issued, and the merge stopped, to allow the missing files to be transferred.

Replaying Changes

Once it has been determined that all data is present, Ensembles determines what transaction log files are new since the last merge. If there are some from other devices, it gathers together the new changes, as well as any logs that occurred concurrently with those files. The logs are then ordered, and played back to update the persistent store.

When applying the updates, Ensembles does not make use of any objects from the app's main Core Data stack. Instead, it sets up a completely separate stack, which only shares the persistent store file itself. It makes all changes in this private stack.

If at some point during this process the app saves to the persistent store, the merge will be terminated with an error. It can simply be retried at a later time.

Committing to the Persistent Store

Conflicting changes on different devices mean that after all new transaction logs have been replayed, the object graph in the private merge context may not be in a valid state. The `CDEPersistentStoreEnsemble` gives its delegate an opportunity to check for and correct this.

First, it invokes `persistentStoreEnsemble:shouldSaveMergedChangesInManagedObjectContext:reparationManagedObjectC`
This method allows you to make updates before an attempt is made to save the changes to the persistent store. It also delineates the point-of-no-return for the merge, giving you a chance to terminate by returning `NO`.

This method is called on a background thread, and each context passed has private-queue concurrency, and should be accessed as such with either `performBlock:` or `performBlockAndWait:`. The two contexts also have a parent-child relationship, so it is unwise to nest calls to `performBlock:`, which would likely result in deadlock.

You can access what has been changed in the merge via the first context. Use the standard `NSManagedObject-Context` properties `insertedObjects`, `updatedObjects`, and `deletedObjects`.

If you actually want to make repairs, *do not make them in the first context.* Instead, use the reparation context. This will require you to pass `NSManagedObjectIDs` between blocks, but it is for good reason: the reparation context will track any changes you make, and these will be added to the transaction logs and applied on other devices.

If you return `YES`, the ensemble will attempt to save the merge changes to the persistent store. If the save fails, the delegate method `persistentStoreEnsemble:didFailToSaveMergedChangesInManagedObjectContext:error:reparationN` gets invoked. The error that arose in saving is passed in, and you can use it to determine what went wrong. If you would like to attempt to fix the error, you can again use the reparation context to make changes, and return `YES` to request a retry.

More detail on conflict resolution and repairs, including sample code, is provided in a later chapter.

Notifying of Commit

If saving to the persistent store succeeds, the `persistentStoreEnsemble:didSaveMergeChangesWithNotification:` delegate method is called. This is a good point to merge changes into any contexts that depend on the store. Invoke the `mergeChangesFromContextDidSaveNotification:` method of each context, and be careful to do this on the appropriate thread or queue.

Exporting New Transaction Logs

At this point, the framework exports any new transaction logs to file, and uploads them to the cloud for other devices to import. It also cleans up any files that are no longer needed in the cloud.

Triggering Merges

Ensembles does not trigger merges automatically — all merges are initiated by your application code. This is quite deliberate, because every app is different, and each will demand its own approach to when and how often merges take place. By requiring the app code to trigger merges, Ensembles offers you maximum control over sync operations.

Initiating a Merge

You begin merging by invoking `mergeWithCompletion:` on the `CDEPersistentStoreEnsemble`. The method is asynchronous, because merging involves networking, and other long running tasks.

The completion block is called back on the main thread. It includes a single `NSError` as solitary parameter; this will be `nil` upon successful completion.

A merge can fail for a variety of reasons, from file downloads being incomplete, to the merge being interrupted by a save to the persistent store. Errors during merging are not typically very serious, and you should just retry the merge a bit later. Error codes can be found later in this chapter.

Progress of a Merge

There is no mechanism for following the progress of the merge process in Ensembles 1.x, but Ensembles 2 fires `CDEPersistentStoreEnsembleDidMakeProgressWithActivityNotification` notifications when progress is made. By retrieving the value corresponding to `CDEProgressFractionKey` in the user info dictionary, you can use the notifications to update a progress indicator.

The completion block also tells you the merge has completed or failed, and is an appropriate place to update the user interface to show that the sync has finished.

Observing Merges

You can test whether an ensemble is currently merging using the `isMerging` boolean property. You can also use KVO to observe changes to this property, and, *e.g.*, show an activity indicator while merging is taking place.

Canceling a Merge

If you decide you want to cancel a merge, invoke the `cancelMergeWithCompletion:` method, which is asynchronous. The completion will be called when the merge completes, or terminates prematurely.

When to Trigger Merges

There are some common times when most apps will want to merge changes, such as on launch and when terminating. Some apps will also want to merge at other times, perhaps at regular time intervals, or when the user taps a *Sync Now* button. The next few sections describe various strategies for initiating merges.

Launching

When the application first launches, it is often a good moment to initiate a merge. There will usually be new transaction logs from other devices that need to be imported, and a merge will give the user confidence that everything is working properly.

The `application:didFinishLaunchingWithOptions:` method on iOS, and `applicationDidFinishLaunching:` on the Mac, are good places to trigger the merge.

Terminating

On the Mac, when your app quits you will likely want to trigger a merge.[44] This will export new transaction logs and make them available to other devices. If you don't do this, your users may be frustrated that changes they make on one device do not appear later when they open your app on another device.

It is also important to allow Ensembles to save its internal data to disk before terminating, so even if you don't merge, you will need to cleanly handle termination on the Mac. You can postpone termination by returning `NSTerminateLater` from the `applicationShouldTerminate:` app delegate method, and then invoking `replyToApplicationShouldTerminate:` on the `NSApplication` when the operation is complete.

[44] On iOS, you are more likely to merge when the app enters the background.

```
-(NSApplicationTerminateReply)applicationShouldTerminate:(NSApplication *)sender
{
    [managedObjectContext save:NULL];

    [persistentStoreEnsemble processPendingChangesWithCompletion:^(NSError *error) {
        [[NSApplication sharedApplication] replyToApplicationShouldTerminate:YES];
    }];

    return NSTerminateLater;
}
```

The `processPendingChangesWithCompletion:` method makes sure that changes arising from the save are fully processed and saved to disk.

If you would like other devices to get the latest changes, you will need to merge when terminating instead.

```
-(NSApplicationTerminateReply)applicationShouldTerminate:(NSApplication *)sender
{
    [managedObjectContext save:NULL];

    [persistentStoreEnsemble mergeWithCompletion:^(NSError *error) {
        [[NSApplication sharedApplication] replyToApplicationShouldTerminate:YES];
    }];

    return NSTerminateLater;
}
```

Note that a merge can take some time, particularly if the cloud file system must perform networking. You will probably want to display a progress indicator to the user while this takes place.

Becoming Active or Inactive

Another good time for a Mac app to merge is when it becomes active or inactive. The user may stop using the app, but leave it running in the background. You will probably want to save and merge as the app becomes inactive, so if it remains inactive for some time, at least the latest changes will appear on other devices.

It is also wise to merge when the app becomes active again, to pull in changes from other devices, which the user will expect to appear.

You can use the app delegate methods `applicationDidBecomeActive:` and `applicationWillResignActive` to invoke the `mergeWithCompletion:` method.

Entering the Background

On iOS, it is generally best to merge when the app enters the background. To ensure the app continues to execute while the merge is taking place, you can register a background task before merging, and end the task when the merge completes.

You would typically use the `applicationDidEnterBackground:` app delegate method for this purpose.

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    UIBackgroundTaskIdentifier identifier = [[UIApplication sharedApplication]
        beginBackgroundTaskWithExpirationHandler:NULL];

    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        [managedObjectContext performBlock:^{
            if (managedObjectContext.hasChanges) {
                [managedObjectContext save:NULL];
            }

            [persistentStoreEnsemble mergeWithCompletion:^(NSError *error) {
                [[UIApplication sharedApplication] endBackgroundTask:identifier];
            }];
        }];
    });
}
```

Changes in the main context are first saved in order to commit them to the store, and form transaction logs. Once the context is saved, you can initiate a merge, and end the background task in the completion handler.

Entering the Foreground

An iOS app may remain in the background for some time. In order to ensure the user gets the latest data from other devices when they next use the app, you will want to merge in `applicationWillEnterForeground:`.

Notifications

Some backends[45] fire notifications when new data from other devices is detected. This is usually a good time to perform a merge.

For example, the `CDEICloudFileSystem` fires the notification `CDEICloudFileSystemDidDownloadFilesNotification` when new file downloads complete. You could set up an observer for this notification, and invoke `mergeWithCompletion:` whenever it fires.

Merging as soon as data is detected makes sync feel snappy, and will be appreciated by your users.

Timers

If you don't have a backend that conveniently notifies you when new data is available, but you do want to keep store data in close sync, you could setup a timer to intermittently invoke the `mergeWithCompletion:` method. For example, here is a timer that fires every two minutes.

[45] Classes conforming to the `CDECloudFileSystem` protocol.

```
timer = [NSTimer scheduledTimerWithTimeInterval:120.0 target:self selector:@selector(perfo\
rmScheduledMerge:)
    userInfo:nil repeats:YES];
```

The `performScheduledMerge:` method would simply invoke `mergeWithCompletion:` on the ensemble.

```
- (void)performScheduledMerge:(NSTimer *)aTimer
{
    [ensemble mergeWithCompletion:NULL];
}
```

Don't forget to invalidate the `NSTimer` when you no longer need it to fire.

```
[timer invalidate];
```

Manual Trigger

You may want to offer your users the option of manually triggering a merge. This doesn't have to be a major feature of the user interface; it could be an extra button hidden away with the sync settings.

Having a button that immediately triggers a sync gives your users a feeling of control, even if most of the time they won't need it.

Using Background Fetch Mode

iOS allows apps to utilize background modes to carry out limited tasks while the app is not active. It is not advisable to perform a full merge using background modes, because you have no control over how long the merge will take, but you might be able to use the background fetch mode to download files or request them such that they are ready to merge when the app enters the foreground.

For example, the iCloud backend automatically requests new files when it observes metadata notifications. If you activate the background fetch mode for your app, you could implement background fetching by first setting the fetch interval

```
[[UIApplication sharedApplication] setMinimumBackgroundFetchInterval:UIApplicationBackgrou\
ndFetchIntervalMinimum];
```

and then including an application delegate method like this

```
-(void)application:(UIApplication *)application
    performFetchWithCompletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler
{
    if (!ensemble.isLeeched) {
        completionHandler(UIBackgroundFetchResultNoData);
    }
    else {
        dispatch_async(dispatch_get_main_queue(), ^{
            CDEICloudFileSystem *cloudFileSystem = (id)ensemble.cloudFileSystem;
            completionHandler(cloudFileSystem.bytesRemainingToDownload > 0 ?
                UIBackgroundFetchResultNewData : UIBackgroundFetchResultNoData);
        });
    }
}
```

Updating Contexts

Ensembles merges changes from other devices into a private background context before saving to the persistent store. When this happens, it is important to inform the contexts in your control of the merge. This section describes how.

Merging Sync Changes into a Context

If you are using a single `NSManagedObjectContext` in your app, you should implement the `persistentStoreEnsemble:didSaveMergeChangesWithNotification:` method — or observe the corresponding notification — in order to inform your context of a background sync.

```
- (void)persistentStoreEnsemble:(CDEPersistentStoreEnsemble *)ensemble
    didSaveMergeChangesWithNotification:(NSNotification *)notification
{
    [managedObjectContext performBlock:^{
        [managedObjectContext mergeChangesFromContextDidSaveNotification:notification];
    }];
}
```

The method is invoked on a background thread; make sure you shunt execution to your app's context thread or queue before invoking `mergeChangesFromContextDidSaveNotification:`.

Multiple Contexts

It is not unusual multiple `NSManagedObjectContexts` in your app. Some have a private-queue parent context for saving to the persistent store, with the main thread context a child context. Other apps may use secondary contexts for importing data and other tasks.

When Ensembles merges, it is important to refresh *all* of the contexts that are in any way related to the affected persistent store. So the delegate method may look something like this.

```
- (void)persistentStoreEnsemble:(CDEPersistentStoreEnsemble *)ensemble
    didSaveMergeChangesWithNotification:(NSNotification *)notification
{
    [backgroundQueueContext performBlockAndWait:^{
        [backgroundQueueContext mergeChangesFromContextDidSaveNotification:notification];
    }];

    [mainContext performBlockAndWait:^{
        [mainContext mergeChangesFromContextDidSaveNotification:notification];
    }];

    [importContext performBlockAndWait:^{
        [importContext mergeChangesFromContextDidSaveNotification:notification];
    }];
}
```

Delegate Methods

The `CDEPersistentStoreEnsemble` class includes a number of delegate methods that are invoked during a merge. They are described here in detail.

The Should-Save-Merge-Changes Method

The `persistentStoreEnsemble:shouldSaveMergedChangesInManagedObjectContext:reparationManagedObjectContext:` delegate method is invoked when the ensemble is about to attempt to save merged changes into the persistent store.

This method is invoked on a background thread. Both of the contexts passed have private queue concurrency type, and so they should only be accessed via calls to `performBlock...` methods.

You can use the saving context to check what changes have been made in the merge via `NSManagedObjectContext` methods like `insertedObjects`, `updatedObjects`, and `deletedObjects`.

You should not make any changes directly in the saving context. If you need to make changes before the save is attempted, you can make them in the reparation context.

You can force the merge to terminate altogether by returning `NO` from this method.

Be careful not to nest calls to the `performBlock...` methods for the two contexts. This will very likely lead to a deadlock, because the contexts in question have a parent-child relationship.

The Did-Fail-To-Save-Merged-Changes Method

The `persistentStoreEnsemble:didFailToSaveMergedChangesInManagedObjectContext:error:reparationManagedObjectC` method gets invoked when the ensemble attempted to save merged changes into the persistent store, but the save failed.

This method is invoked on a background thread. Both of the contexts passed have private queue concurrency type, and so they should only be accessed via calls to `performBlock...` methods.

You can use the saving context to check what changes failed to save via `NSManagedObjectContext` methods like `insertedObjects`, `updatedObjects`, and `deletedObjects`.

The error that occurred during saving is passed and can be used to determine which objects are responsible for the failure. One possible implementation of the delegate method might look as follows.

```
-(BOOL)persistentStoreEnsemble:(CDEPersistentStoreEnsemble *)ensemble
    didFailToSaveMergedChangesInManagedObjectContext:(NSManagedObjectContext *)savingConte\
xt
    error:(NSError *)error
    reparationManagedObjectContext:(NSManagedObjectContext *)reparationContext
{
    NSMutableArray *objectIDs = [NSMutableArray array];
    NSMutableArray *errors = [NSMutableArray array];

    [savingContext performBlockAndWait:^{
        if ( error.code != NSValidationMultipleErrorsError ) {
            NSManagedObject *object = error.userInfo[@"NSValidationErrorObject"];
            [objectIDs addObject:object.objectID];
            [errors addObject:error];
        }
        else {
            NSArray *detailedErrors = error.userInfo[NSDetailedErrorsKey];
            for ( NSError *error in detailedErrors ) {
                NSDictionary *detailedInfo = error.userInfo;
                NSManagedObject *object = detailedInfo[@"NSValidationErrorObject"];
                [objectIDs addObject:object.objectID];
                [errors addObject:error];
            }
        }
    }];

    [reparationContext performBlockAndWait:^{
        [objectIDs enumerateObjectsWithOptions:0
            usingBlock:^(NSManagedObjectID *objectID, NSUInteger index, BOOL *stop) {
            NSError *error;
            id object = [reparationContext existingObjectWithID:objectID error:&error];
            if (!object) {
                NSLog(@"Failed to retrieve invalid object: %@", error);
                return;
            }
            [object repairWithError:errors[index]];
        }];
```

```
38      }];
39
40      return YES;
41  }
```

This code assumes that you are using a `NSManagedObject` subclass with a `repairWithError:` method that can handle validation errors. The following example of one such method, which handles validation errors caused by orphaned objects by simply deleting the object.

```
1   -(void)repairForError:(NSError *)error
2   {
3       if ( error.code == NSValidationMissingMandatoryPropertyError ||
4            error.code == NSManagedObjectValidationError ||
5            error.code == NSValidationRelationshipLacksMinimumCountError ) {
6           [self.managedObjectContext deleteObject:self];
7       }
8   }
```

You should not make any changes directly in the saving context. If you wish to reattempt the save, make any necessary changes in the reparation context, and then return `YES`.

You can force the merge to terminate altogether by returning `NO` from this method.

The Did-Save-Merge-Changes Method

The `persistentStoreEnsemble:didSaveMergeChangesWithNotification:` delegate method is invoked after the ensemble successfully saves merged changes into the persistent store.

This method is invoked on a background thread. The notification passed includes a `userInfo` dictionary with the `NSManagedObjectID` instances for objects that were changed in the merge. It can be used to determine what insertions, updates, and deletions occurred.

You will usually want to pass this notification to the `mergeChangesFromContextDidSaveNotification:` method of any context that accesses the persistent store, be it directly or indirectly. This will allow the contexts to account for the changes.

As always, be sure to invoke the `mergeChangesFromContextDidSaveNotification:` method on the thread or queue corresponding to the messaged context.

Merge Errors

Errors during merging are common, and generally not serious. You should treat them as you would networking errors. The merge may have failed, but usually it is just a question of retrying later.

You can find all error codes in the header file *CDEDefines.h*.

Common error codes that can arise during a merge

Error Code	Description
CDEErrorCodeCancelled	The operation was cancelled.
CDEErrorCodeDisallowedStateChange	Request for invalid state transition was made. *Eg* Merge during leeching.
CDEErrorCodeFileCoordinatorTimedOut	Accessing a file with a coordinator timed out. Often because iCloud is still downloading. Retry later.
CDEErrorCodeFileAccessFailed	An attempt to access a file failed.
CDEErrorCodeDiscontinuousRevisions	Some change sets are missing. Usually temporarily missing data. Retry a bit later.
CDEErrorCodeMissingDependencies	Some change sets are missing. Usually temporarily missing data. Retry a bit later.
CDEErrorCodeCloudIdentityChanged	User changed cloud identity. This forces a deleech.
CDEErrorCodeDataCorruptionDetected	Some left over, incomplete data has been found. Probably due to a crash.
CDEErrorCodeUnknownModelVersion	A model version exists in the cloud that is unknown. Merge will succeed again after update that includes the new version.
CDEErrorCodeStoreUnregistered	The ensemble is no longer registered in the cloud. Usually due to cloud data removal.
CDEErrorCodeSaveOccurredDuringMerge	A save to the persistent store occurred during merge. You can simply retry the merge.
CDEErrorCodeMissingStore	There is no persistent store at the path. Ensure a store exists and try again.
CDEErrorCodeMissingDataFiles	Files used to store large `NSData` attributes are missing. Usually temporary. Retry a bit later.
CDEErrorCodeNetworkError	A generic networking error occurred.
CDEErrorCodeServerError	An error from a server was received.
CDEErrorCodeConnectionError	The cloud file system could not connect.
CDEErrorCodeAuthenticationFailure	The user failed to authenticate.

Backends

This chapter introduces the various backends that Ensembles can work with. You will be introduced to the peculiarities of each, and how best to use them, including when it is best to attempt a merge.

iCloud Drive

iCloud Drive is a file transfer backend built into all Apple devices. An attractive feature of iCloud Drive is that virtually all Apple customers have an iTunes account, and thereby an iCloud account, so you don't have to get them to sign up before they can sync with your app. The service is also free for the developer: customers get some free storage, and can pay for more as needed.

Installing

You do not need to do anything to install the iCloud Drive cloud file system. It is included in the core functionality of Ensembles.

Initializing

You should check whether the user of your app is logged into iCloud, and has the *Documents & Data* setting enabled, before trying to initialize an `CDEICloudFileSystem` instance. To do the check, simply access the `NSFileManager` property `ubiquityIdentityToken`; if it is `nil`, the user is not logged in, or has *Documents & Data* switched off.

Once you know that the user is logged in and you have access to data storage in iCloud, you can initialize an `CDEICloudFileSystem`.

```
1  CDEICloudFileSystem *cloudFileSystem =
2      [[CDEICloudFileSystem alloc] initWithUbiquityContainerIdentifier:nil];
```

You pass the ubiquity container identifier as argument. This is usually your app identifier, prepended with 'iCloud', as in *iCloud.com.mentalfaculty.idiomatic*. Passing in `nil` for the container identifier will cause the app's default container to be used. For many apps, this will be adequate.

Delegate Methods and Notifications

The `CDEICloudFileSystem` class currently has no delegate methods, but it does fire the notification `CDEICloudFileSystemDidDownloadFilesNotification` when it detects that one or more new files have finished downloading from the iCloud servers.

When to Merge

In addition to merging on launch, termination, and entering the background, the firing of the `CDEICloudFileSystemDidDownloadFilesNotification` notification offers a good opportunity to merge. It is fired when files from other devices have become available on the local device. Starting a merge operation will import these new files, and update the persistent store.

CloudKit

CloudKit is a framework built into all Apple devices from iOS 8 and OS X 10.10 onwards. An attractive feature of CloudKit is that virtually all Apple customers have an iCloud account, so you don't have to get them to sign up before they can sync with your app. If you utilize CloudKit's private database, the service is also free for the developer: customers get some free storage, and can pay for more as needed.

Installing

To use the CloudKit backend in your project, add the CloudKit framework to your target in Xcode, and then drag in the `CDECloudKitFileSystem.h` and `CDECloudKitFileSystem.m` files, which are in the `Framework/Extensions` directory of Ensembles.

If you are using Cocoapods, you simply need to add the CloudKit subspec to your Podfile.

```
pod "Ensembles/CloudKit", "~> 2.0"
```

This will link CloudKit, and include the `CDECloudKitFileSystem` class in your project.

You will also need to enable the CloudKit entitlement for your app, and choose an appropriate ubiquity identifier for your app. This will also require changes to your provisioning profiles to include CloudKit support.

Initializing

You initialize an `CDECloudKitFileSystem` like this:

```
CDECloudKitFileSystem *cloudFileSystem =
    [[CDECloudKitFileSystem alloc]
                initWithPrivateDatabaseForUbiquityContainerIdentifier:@"iCloud.com.mentalfaculty.idiomat\
ic"
                schemaVersion:CDECloudKitSchemaVersion2];
```

There is another initializer that allows use of the public database. The public database does not allow for zones, and Ensembles has to use a different technique for fetching data, which in our experience is not as robust or fast. If possible, use the initializer above, with the most recent schema.

You pass the ubiquity container identifier as argument. In this example, it is *iCloud.com.mentalfaculty.idiomatic*. *Do not* pass in `nil` for the container identifier.

The schema parameter is for backwards compatibility. If you have a new app, use the most recent schema. Once you have chosen a schema, you should not generally change it, as this will require your users to start over with sync.

Deploying

CloudKit enters a Development mode when you first add it to your app. You must switch it to Production mode in the CloudKit Dashboard before deploying the app in the store. Without that, your app will not sync.

When you deploy on CloudKit, you will be asked if you want to keep indexes that have not been used. It is easiest just to keep the indexes, but if you decide to remove them, thoroughly test the app's sync before shipping it.

Delegate Methods and Notifications

There are currently no delegate methods or custom notifications, but you can use push notifications to trigger merges. This is discussed in the next section.

When to Merge

CloudKit provides push notifications for when changes occur in the cloud. The `CDECloudKitFileSystem` class can setup a so-called subscription so that your app receives these push notifications, but there are still a few steps you need to take to handle them and trigger a merge.

You setup the CloudKit subscription when you create a new `CDECloudKitFileSystem` object.

```
1  CDECloudKitFileSystem *cloudFileSystem = [[CDECloudKitFileSystem alloc]
2  initWithPrivateDatabaseForUbiquityContainerIdentifier:@"iCloud.com.mentalfaculty.idiomatic\
3  "
4  schemaVersion:CDECloudKitSchemaVersion2];
5  [cloudFileSystem subscribeForPushNotificationsWithCompletion:^(NSError *error) {
6      if (error) NSLog(@"Failed to subscripe for push: %@", error);
7  }];
```

On iOS you need to register for remote notifications, usually in the `application:didFinishLaunchingWithOptions:` delegate method.

```
1  [[UIApplication sharedApplication] setMinimumBackgroundFetchInterval:UIApplicationBackgrou\
2  ndFetchIntervalMinimum];
3  [[UIApplication sharedApplication] registerForRemoteNotifications];
```

You then implement the `application:didReceiveRemoteNotification:` delegate method to trigger a merge.

```
- (void)application:(UIApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo
    fetchCompletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler
{
    UIBackgroundTaskIdentifier backgroundIdentifier =
        [[UIApplication sharedApplication] beginBackgroundTaskWithExpirationHandler:NULL];
    [ensemble mergeWithCompletion:^(NSError *error) {
        if (error) NSLog(@"Error: %@", error);
        [[UIApplication sharedApplication] endBackgroundTask:backgroundIdentifier];
        completionHandler(UIBackgroundFetchResultNewData);
    }];
}
```

It's also possible to setup background modes so that your app can retrieve and merge data when in the background. To do this, add the 'Remote notifications' background mode to the capabilities of your app target in Xcode.

On the Mac, you register for remote notifications in the `applicationDidFinishLaunching:` delegate method.

```
[[NSApplication sharedApplication] registerForRemoteNotificationTypes:NSRemoteNotification\
TypeNone];
```

Then implement the remote notification delegate methods to trigger a merge.

```
- (void)application:(NSApplication *)application
    didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken
{
    NSLog(@"%@ with token = %@", NSStringFromSelector(_cmd), deviceToken);
}

- (void)application:(NSApplication *)application
    didFailToRegisterForRemoteNotificationsWithError:(NSError *)error
{
    NSLog(@"%@ with error = %@", NSStringFromSelector(_cmd), error);
}


- (void)application:(NSApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo
{
    [ensemble mergeWithCompletion:^(NSError *error){
        if (error) NSLog(@"Error: %@", error);
    }];
}
```

Dropbox Core API

Dropbox[46] is best known as a consumer file syncing service, but developers can also access file storage using various SDKs, including the REST-based Core API. There is an Objective-C SDK available to access this API from iOS and OS X. The Ensembles class `CDEDropboxV2CloudFileSystem` is built on the SDK. The class does not access files directly in the file system, even if they are accessible; instead, it makes HTTP requests to upload and download data.

Note that the Dropbox API v1 is deprecated, and you should use the new v2 API.

Signing Up

Before syncing your app via Dropbox, you need to sign up for a Dropbox Developer account, and register your app. Navigate to the Dropbox Developer Site[47]. Sign up for an account, then follow these steps:

1. In the App Console, click the *Create app* button.
2. Choose the *Dropbox API app* type.
3. Choose to store *Files and Datastores*.
4. Choose *Yes — My app only needs access to files it creates*.
5. Name the app.
6. Click on *Create app*.
7. Note the app key and secret.

Installing

To manually install the Dropbox Core API backend, you must first install the SDK. If you are using Ensembles via Git, you can simply update the submodules to add the SDK to the *Vendor* folder.

```
1  git submodule update --init
```

You can also install the SDK by downloading it from the Dropbox Developer Site[48], or by forking this repo[49] on GitHub.

Follow the instructions in the SDK to add it to your Xcode project.

Once you have added the Dropbox SDK to your project, you should drag in the `CDEDropboxV2CloudFileSystem.h` and `CDEDropboxV2CloudFileSystem.m` files, which are in the `Framework/Extensions` directory of Ensembles.

If you are using Cocoapods, you can ignore the information above, and simply add the Dropbox subspec to your Podfile.

[46] http://dropbox.com
[47] http://developer.dropbox.com
[48] https://www.dropbox.com/developers/
[49] https://github.com/dropbox/dropbox-sdk-obj-c

```
pod "Ensembles/DropboxV2", "~> 1.0"
```

This will link the Dropbox SDK, and include the `CDEDropboxV2CloudFileSystem` class in your project.

Initializing

To initialize the `CDEDropboxV2CloudFileSystem` object, you use the standard `init` initializer, but you will also want to setup your `DBClientsManager`.

```
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        [DBClientsManager setupWithAppKey:"<Your Dropbox App Key here>"];
    });
    CDEDropboxV2CloudFileSystem *newDropboxSystem = [[CDEDropboxV2CloudFileSystem alloc] i\
nit];
    newDropboxSystem.delegate = self;
```

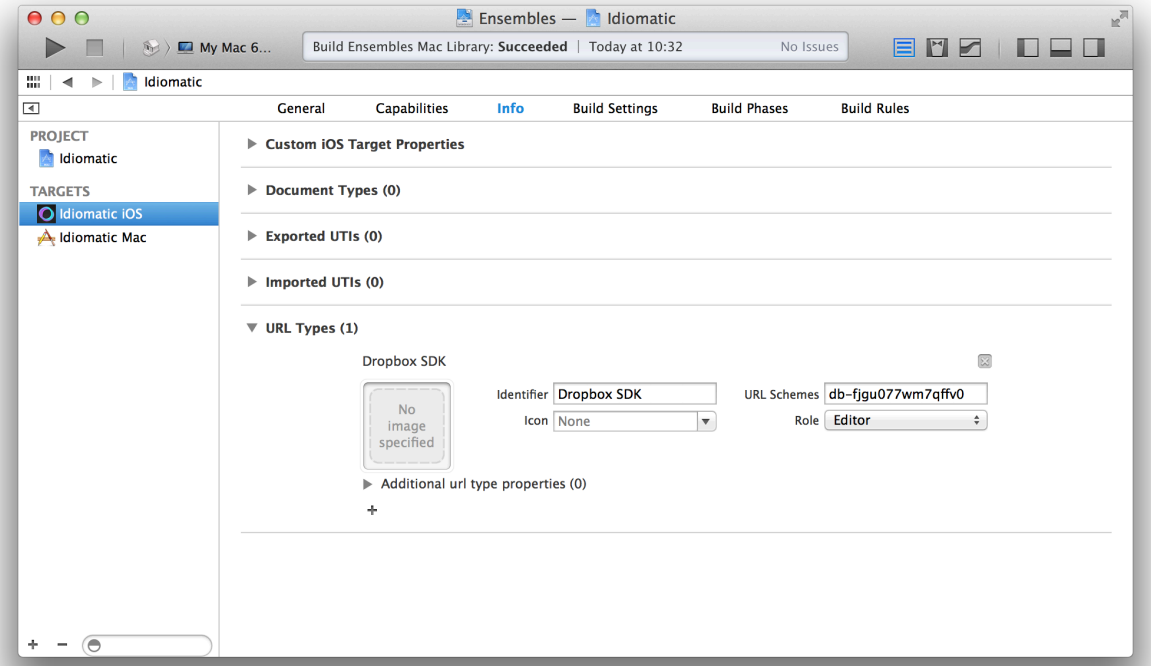
Enter the app key and secret you noted after registering your app at the Dropbox Developer Site.

To authenticate, you present a login screen that the SDK provides. Once the user has authenticated, the SDK has to transfer control back to your app. It does this using a custom URL scheme. For this to work, you have to register your app to handle the URL scheme.

Select the Project root in the source list of Xcode, and then the App's target. Open the *Info* tab, and the URL Types section. Change the URL Schemes entry to

```
        db-xxxxxxxxxxxxx
```

Fill in your own app key in place of the x's. Finally, enter an identifier for the URL type (*eg* Dropbox SDK).



Setting the Dropbox URL Scheme.

Since iOS 9, apps are required to register all URL schemes they will use in the `Info.plist` file. The Dropbox Core SDK makes use of some URL schemes for authentication, and these must be registered by your app.

To do this, open the `Info.plist` file of your app, and insert an entry for the key `LSApplicationQueriesSchemes`. The value should be an array, and include the strings `dbapi-2` and `dbapi-8-emm`.

```
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>dbapi-2</string>
  <string>dbapi-8-emm</string>
</array>
```

More info about the change can be found at Dropbox[50].

Delegate Methods

In order for the Dropbox backend to operate, you need to have your controller class (*eg* App delegate) conform to the `CDEDropboxV2CloudFileSystemDelegate` protocol, and set the delegate of the `CDEDropboxV2CloudFileSystem` instance you created.

[50] https://blogs.dropbox.com/developers/2015/08/important-update-your-core-api-app-for-ios-9/

The required delegate method `-linkSessionForDropboxCloudFileSystem:completion:` is invoked when the `CDEDropboxV2CloudFileSystem` needs you to authenticate the user using the SDK. A typical authentication might look like this

```
- (void)linkSessionForDropboxCloudFileSystem:(CDEDropboxV2CloudFileSystem *)fileSystem com\
pletion:(CDECompletionBlock)completion
{
    // User is already authorized, call the completion block right away
    if ([DBClientsManager authorizedClient] != nil) {
        dispatch_async(dispatch_get_main_queue(), ^{
            completion(nil);
        });
        return;
    }

    dropboxLinkSessionCompletion = [completion copy];
    UIApplication *application = [UIApplication sharedApplication];
    UIViewController *rootController = [[application keyWindow] rootViewController];
    [DBClientsManager authorizeFromController:application
                                   controller:rootController
                                      openURL:^(NSURL *url){ [[UIApplication sharedAp\
plication] openURL:url]; }];
}
```

This uses the `DBClientsManager` method `authorizeFromController:controller:openURL:` to present the login view. The completion handler has to be called back when the authentication is finished, to inform the `CDEDropboxV2CloudFileSystem` that it can continue, but because the authentication process is asynchronous, it is necessary to store the completion handler in an instance variable.

```
CDECompletionBlock dropboxLinkSessionCompletion;
```

You can call it when the app is requested to open the URL for the Dropbox scheme registered earlier.

```
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url
{
    DBOAuthResult *authResult = [DBClientsManager handleRedirectURL:url];
    if (authResult) {
            if ([authResult isSuccess]) {
            CDEDropboxV2CloudFileSystem *dropboxSystem = cloudFileSystem;
            if (!dropboxSystem.client) {
                NSString *accessToken = authResult.accessToken.accessToken;
                dropboxSystem.client = [[DBUserClient alloc] initWithAccessToken:accessTok\
en];
            }
```

```
12                      dispatch_async(dispatch_get_main_queue(), ^{
13                  if (dropboxLinkSessionCompletion) dropboxLinkSessionCompletion(nil);
14                  dropboxLinkSessionCompletion = NULL;
15              });
16          }
17          else {
18              NSError *error = [NSError errorWithDomain:CDEErrorDomain
19                                                   code:CDEErrorCodeAuthenticationFailure
20                                               userInfo:nil];
21              dispatch_async(dispatch_get_main_queue(), ^{
22                  if (dropboxLinkSessionCompletion) dropboxLinkSessionCompletion(error);
23                  dropboxLinkSessionCompletion = NULL;
24              });
25          }
26
27              return YES;
28      }
29
30      return NO;
31  }
```

If the `DBOAuthResult` is successful, the completion callback is called with a `nil` argument; if an error occurred, an `NSError` instance is passed to the completion handler instead.

On Mac OS X you handle linking using the `authorizeFromControllerDesktop:controller:openURL:` instead. See the `DBClientsManager+DesktopAuth-macOS.h` header file for details.

When to Merge

The `CDEDropboxV2CloudFileSystem` class delegate method `-dropboxCloudFileSystemDidDetectRemoteFileChanges:` is called when new remote files are available. This is a good time to merge with the cloud data. You should also just merge at standard times, such as launch, and when entering the background. You might also schedule an `NSTimer` to fire every few minutes and perform a merge.

Deploying

Dropbox enters a Development mode when you first register your app. You must switch it to Production mode in the Developer Dashboard before deploying the app in the store.

WebDAV

WebDAV[51] is cloud storage based on a HTTP standard, accessed as a web service. The Ensembles class `CDEWebDavCloudFileSystem` is built to access WebDAV services, and is included with some Ensembles packages in the Ensembles 2 framework.

[51] http://en.wikipedia.org/wiki/WebDAV

Service

To use this backend, you will need to have access to a WebDAV service. You can host your own, or subscribe to a WebDAV service.

Installing

To use the WebDAV backend, drag in the `CDEWebDavCloudFileSystem.h` and `CDEWebDavCloudFileSystem.m` files, which are in the `Framework/Extensions` directory of Ensembles.

If you are using Cocoapods, you simply need to add the Dropbox subspec to your Podfile.

```
1  pod "Ensembles/WebDAV", "~> 2.0"
```

This will include the `CDEWebDavCloudFileSystem` class in your project.

Initializing

To initialize `CDEWebDavCloudFileSystem`, you use the `initWithBaseURL:` initializer. The argument is a URL to the location that Ensembles should store its data. Note that the username for the WebDAV service is often included in the URL's path.

After initializing the file system, you should set the `username` and `password` properties. It is up to you how you have the user input these values, and where you store them. The keychain is a secure place for storage.

In addition to supplying login credentials, you should also set the delegate, in order to handle failed authentications, and possibly update the `baseURL` if needed.

Delegate Methods

The delegate method `webDavCloudFileSystem:updateLoginCredentialsWithCompletion:` is called if authentication fails. You should request new credentials from the user and set the `username` and `password` properties, and then call the completion callback to indicate you are finished.

The `webDavCloudFileSystemWillFormURLRequest:` method is invoked just before a URL request is formed. It is useful if you need to update the `baseURL` because it is based on the username or other dynamic quantities. The URL of WebDAV services often includes the username, so the `baseURL` needs to be modified if a new username is adopted.

When to Merge

The `CDEWebDavCloudFileSystem` class currently does not provide any notifications for cloud file updates, so you should just merge at standard times, such as launch, and when entering the background. You might also schedule an `NSTimer` to fire every few minutes and perform a merge.

Zip Compression

The `CDEZipCloudFileSystem` is available only in Ensembles 2, and is not a standalone backend. It is used in combination with other backends to zip files before they are uploaded. For JSON transaction logs, this can lead to a dramatic reduction in file size.

Installing

To use the Zip cloud file system, you must first install SSZipArchive. If you are using Ensembles via Git, you can simply update the submodules to add it to the *Vendor* folder.

```
git submodule update --init
```

It also available directly on GitHub here[52].

Once you have SSZipArchive, drag the source files into your project. Also drag in the `CDEZipCloudFileSystem.h` and `CDEZipCloudFileSystem.m` files, which are in the `Framework/Extensions` directory of Ensembles 2.

If you are using CocoaPods, you simply need to add the Zip subspec to your Podfile.

```
pod "Ensembles/Zip", "~> 2.0"
```

This will link SSZipArchive, and include the `CDEZipCloudFileSystem` class in your project.

Initializing

To initialize the `CDEZipCloudFileSystem`, you use the `initWithCloudFileSystem:` initializer. The argument is the `CDECloudFileSystem` instance which is responsible for actually transferring the files to the cloud.

```
CDEICloudFileSystem *iCloudFileSystem = [[CDEICloudFileSystem alloc] initWithUbiquityConta\
inerIdentifier:nil];
CDEZipCloudFileSystem *zipCloudFileSystem = [[CDEZipCloudFileSystem alloc] initWithCloudFi\
leSystem:iCloudFileSystem];
ensemble = [[CDEPersistentStoreEnsemble alloc] initWithEnsembleIdentifier:@"MainStore"
    persistentStoreURL:storeURL
    managedObjectModelURL:modelURL
    cloudFileSystem:zipCloudFileSystem];
```

The Zip cloud file system is passed to the ensemble initializer, not the transferring file system.

[52] https://github.com/soffes/ssziparchive

Encryption

The `CDEEncryptedCloudFileSystem` is available only in Ensembles 2, and is not a standalone backend. It is used in combination with other backends to encrypt files before they are uploaded. Files are encrypted using a symmetric AES key built upon some user provided password.

Installing

To use the encrypted cloud file system, you must first install RNCryptor. If you are using Ensembles via Git, you can simply update the submodules to add it to the *Vendor* folder.

```
git submodule update --init
```

It is also available directly on GitHub here[53].

Once you have RNCryptor, link to the target appropriate for your platform. Also link to the Security framework.

Drag in the `CDEEncryptedCloudFileSystem.h` and `CDEEncryptedCloudFileSystem.m` files, which are in the `Framework/Extensions` directory of Ensembles 2.

If you are using CocoaPods, you simply need to add the Encrypt subspec to your Podfile.

```
pod "Ensembles/Encrypt", "~> 2.0"
```

This will link RNCryptor, and include the `CDEEncryptedCloudFileSystem` class in your project.

Initializing

To initialize the `CDEEncryptedCloudFileSystem`, you usually use the `initWithCloudFileSystem:password:` initializer. The arguments are the `CDECloudFileSystem` instance which is responsible for actually transferring the files to the cloud, and the password used to encrypt the data. (You should keep the password securely stored, perhaps in the keychain.)

```
NSString *password = @"<Retrieve password from Keychain>";
CDEICloudFileSystem *iCloudFileSystem =
    [[CDEICloudFileSystem alloc] initWithUbiquityContainerIdentifier:nil];
CDEEncryptedCloudFileSystem *encryptedCloudFileSystem =
    [[CDEEncryptedCloudFileSystem alloc] initWithCloudFileSystem:iCloudFileSystem
        password:password];
ensemble = [[CDEPersistentStoreEnsemble alloc] initWithEnsembleIdentifier:@"MainStore"
    persistentStoreURL:storeURL
    managedObjectModelURL:modelURL
    cloudFileSystem:encryptedCloudFileSystem];
```

The encrypted cloud file system is passed to the ensemble initializer, rather than the transferring file system.

[53] https://github.com/RNCryptor/RNCryptor

Ensembles Server (Node.js, S3)

Ensembles Server is a basic Node.js web app that utilizes Amazon's S3 for file storage, and a Postgres database for user management. It can be used with the `CDENodeCloudFileSystem` in the Ensembles framework on iOS devices. The server offers HTTP Basic Authentication, and an API that supports user actions such as signing up, logging in, changing password, and resetting a password (emails new password).

Installing

See the section on Deploying Ensembles Server.

Initializing

To initialize the `CDENodeCloudFileSystem` instance, you need to provide a username and password.

```
    NSURL *url = [NSURL URLWithString:@"https://<your app>.herokuapp.com"];
    NSString *username = [[NSUserDefaults standardUserDefaults] stringForKey:IDMNodeS3Emai\
lDefaultKey] ? : @"";
    NSString *password = [self retrievePassword];
    CDENodeCloudFileSystem *newNodeFileSystem = [[CDENodeCloudFileSystem alloc] initWithBa\
seURL:url];
    newNodeFileSystem.delegate = self;
    newNodeFileSystem.username = username;
    newNodeFileSystem.password = password;
    newSystem = newNodeFileSystem;
```

You can store the username and password however you please, but the Keychain is probably the most secure place to store authentication credentials.

Delegate Methods

The required delegate method `-nodeCloudFileSystem:updateLoginCredentialsWithCompletion:` is invoked by the `CDENodeCloudFileSystem` when it needs to authenticate with Ensembles Server. You need to present a request to the user to enter the username and password, set the corresponding properties on the `CDENodeCloudFileSystem` object, and then call the completion block.

In this particular implementation, the completion block is stored in an instance variable (`nodeCredentialUpdateCompletion`), and a view controller is presented.

```
- (void)nodeCloudFileSystem:(CDENodeCloudFileSystem *)fileSystem
    updateLoginCredentialsWithCompletion:(CDECompletionBlock)completion
{
    [self clearPassword];
    nodeCredentialUpdateCompletion = [completion copy];

    // Present the node settings view
    UIWindow *window = [[UIApplication sharedApplication] keyWindow];
    UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"Main" bundle:nil];
    UINavigationController *nodeSettingsNavController =
        [storyboard instantiateViewControllerWithIdentifier:@"NodeSettingsNavigationContro\
ller"];
    IDMNodeSyncSettingsViewController *settingsController =
        (id)nodeSettingsNavController.topViewController;
    settingsController.nodeFileSystem = fileSystem;

    [window.rootViewController presentViewController:nodeSettingsNavController
        animated:YES completion:NULL];
}
```

The loaded view controller calls the `storeNodeCredentials` method when the user enters a username and password. This method sets the username and password on the `CDENodeCloudFileSystem` object, and also stores the values for future use. It calls the stored completion callback, passing `nil` upon success, and an `NSError` if something goes wrong.

```
- (void)storeNodeCredentials
{
    CDENodeCloudFileSystem *nodeFileSystem = (id)self.ensemble.cloudFileSystem;
    NSString *email = nodeFileSystem.username;
    NSString *password = nodeFileSystem.password;
    NSError *error = nil;
    if (email && password) {
        [[NSUserDefaults standardUserDefaults] setObject:email
            forKey:IDMNodeS3EmailDefaultKey];
        [self storePassword:password];
    }
    else {
        NSDictionary *info =
            @{NSLocalizedDescriptionKey : @"Invalid username or password"};
        error = [NSError errorWithDomain:CDEErrorDomain
            code:CDEErrorCodeAuthenticationFailure userInfo:info];
    }

    if (nodeCredentialUpdateCompletion) nodeCredentialUpdateCompletion(error);
    nodeCredentialUpdateCompletion = NULL;
}
```

The view controller also offers the opportunity for the user to cancel the login. If this happens, the `cancelNodeCredentialsUpdate` is called instead, and the completion handler is called with an error.

```
- (void)cancelNodeCredentialsUpdate
{
    NSError *error = [NSError errorWithDomain:CDEErrorDomain
        code:CDEErrorCodeCancelled userInfo:nil];
    if (nodeCredentialUpdateCompletion)
        nodeCredentialUpdateCompletion(error);
    nodeCredentialUpdateCompletion = NULL;
}
```

Storing Passwords in the Keychain

The Keychain is probably the safest place to store the user's password. Because it has a plain C API, the Keychain can be intimidating at first.

Objective-C wrappers classes which make it more palatable are available. A popular choice is SSKeychain[54].

If you would rather use the C API directly, link the *Security* framework to your app, and adapt the following methods.

```
- (NSDictionary *)keychainQuery {
    NSString *serviceName = @"<Your service name as reverse DNS>";
    return @{
        (__bridge id)kSecClass : (__bridge id)kSecClassGenericPassword,
        (__bridge id)kSecAttrService : serviceName,
        (__bridge id)kSecAttrAccount : serviceName,
        (__bridge id)kSecAttrAccessible : (__bridge id)kSecAttrAccessibleAlways
    };
}

- (void)storePassword:(NSString *)newPassword
{
    NSMutableDictionary *keychainQuery = [[self keychainQuery] mutableCopy];
    SecItemDelete((__bridge CFDictionaryRef)keychainQuery);
    keychainQuery[(__bridge id)kSecValueData] =
        [newPassword dataUsingEncoding:NSUTF8StringEncoding];
    SecItemAdd((__bridge CFDictionaryRef)keychainQuery, NULL);
}

- (NSString *)retrievePassword
{
    NSMutableDictionary *keychainQuery = [[self keychainQuery] mutableCopy];
```

[54] https://github.com/soffes/sskeychain

```
    keychainQuery[(__bridge id)kSecReturnData] = @YES;
    keychainQuery[(__bridge id)kSecMatchLimit] = (__bridge id)kSecMatchLimitOne;

    NSString *result = nil;
    CFDataRef data = NULL;
    if (noErr == SecItemCopyMatching((__bridge CFDictionaryRef)keychainQuery,
        (CFTypeRef *)&data)) {
        result = [[NSString alloc] initWithData:(__bridge id)data
            encoding:NSUTF8StringEncoding];
    }
    if (data) CFRelease(data);

    return result;
}

- (void)clearPassword
{
    NSDictionary *keychainQuery = [self keychainQuery];
    SecItemDelete((__bridge CFDictionaryRef)keychainQuery);
}
```

You need to set the service name to something unique, usually in reverse DNS form.

When to Merge

The `CDENodeCloudFileSystem` class currently does not provide any notifications for cloud file updates, so you should just merge at standard times, such as launch, and when entering the background. You might also schedule an `NSTimer` to fire every few minutes and perform a merge.