A Proposed Architecture for Network-Bound iOS Apps

Adrian Kosmaczewski

2011-09-20

One of my most popular answers[1] in StackOverflow is the one I gave to the following question: "What is the best architecture for an iOS application that makes many network requests?"

The problem is the following: let's consider a relatively complex application, which has to connect to, and retrieve and send data from different remote resources (from the same origin or from different ones), all while handling as gracefully as possible different problems such as "network not available", "500" errors, etc, while at the same time notifying the app about showing popups, enabling and disabling fields, with many different screens (usually each with its own controller), and so on.

This article will describe in detail a solution for this problem using ASIHTTPRequest[2], my favorite HTTP library for iOS. The solution involves a bit of object oriented code, and there is a sample implementation in our Senbei project in Github[3] that I am going to refer to in this article.

I want to point out that I do not consider this the "best" architecture by any means; it is simply a pattern or structure that has given me excellent results

[1]http://stackoverflow.com/questions/4810289/best-architecture-for-an-ios-application-that-makes-many-network-requests/4823001#4823001

[2]http://allseeing-i.com/ASIHTTPRequest/

[3]https://github.com/akosmasoftware/Senbei/

in many different applications, and which has evolved over time from many other approaches. If someone else has a better idea, I'd be glad to try it! This architecture also has the advantage of being easy to document, understand, maintain and extend.In the answer on StackOverflow I just enumerated the different elements of the architecture; here I will explain the rationale behind every decision.

Singleton, "Class Cluster"

As I said in the answer, this architecture involves a single object taking care of network connectivity, which I will call a "network manager". Typically this object is a singleton (created using Matt Gallagher's Cocoa singleton macro[4]). Basically this is because it's a good way to centralize all the network logic in a single component, and it is also a very common Cocoa design pattern.

This object can also be seen as a class cluster, because it uses an army of individual classes that perform the real work behind the scenes.

In Senbei, this singleton object is the SBNetworkManager[5] class. All the controllers of the application use the methods of this class to trigger asynchronous requests to the remote FFCRM server used by the application. All of these controllers, as well as the application delegate, are notified of events by means of ad hoc notifications (defined in the SBNotifications.h[6] file).

Network Queues

The network manager wraps an instance of ASINetworkQueue, and also acts as its delegate. Network queues are interesting in mobile apps, given that the available bandwidth varies drastically when the device is connected through a wifi connection or a low-speed GPRS mobile network. The network queue will automatically change the number of requests sent by unit of time depending on the current connectivity, without clogging the device.

In our example, SBNetworkManager has a private ivar (well, as private as Objective-C allows ivars to be) pointing to an instance of ASINetworkQueue, itself a subclass of NSOperationQueue.

One Subclass per Request

I create subclasses of ASIHTTPRequest for each kind of network request that my app requires (typically, for each backend REST interaction or SOAP endpoint).

I also create a base class for all the requests of the application; this allows to centralize some shared behavior in the base class, which proves handy while extending and refactoring your network code.

[4]http://cocoawithlove.com/2008/11/singletons-appdelegates-and-top-level.html

[5]https://github.com/akosmasoftware/Senbei/tree/master/Classes/Helpers/SBNetworkManager

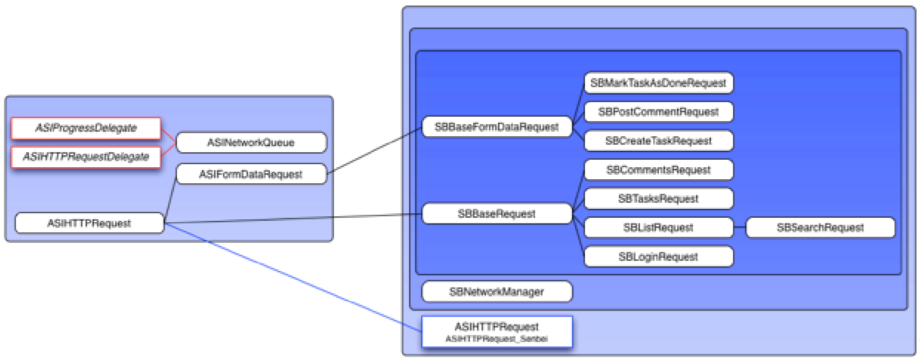
[6]https://github.com/akosmasoftware/Senbei/blob/master/Classes/Helpers/SBNotifications.h

In our example, Senbei has a base class for all the GET requests in the application, called SBBaseRequest[7]. There is another base request, called SBBaseFormDataRequest[8], which is used for requests that use the POST and PUT verbs (used to create and modify resources on the server).

There is also a category on ASIHTTPRequest, to add some methods to any request create on the system; this is required because SBBaseRequest inherits from ASIHTTPRequest, while SBFormDataRequest inherits from ASIHTTPFormDataRequest, which also inherits from ASIHTTPRequest. The category on the latter allows to inject some common behavior in a way that classic inheritance does not allow per se.

For every network interaction in the server, there is a dedicated class available for the SBNetworkManager; the code is easy to understand, and the responsibilities are separated and well defined. Should the system be extended in the future, the extension mechanism will naturally fit any new request, in a horizontal fashion.

The following class diagram (generated from the Xcode project using the excellent OmniGraffle[9] application) shows how the system is structured (you can click the diagram to see a larger version).



[10]

Polymorphism to the Rescue

The network manager doesn't know what to do with the result of each request; hence, it just calls a method **on the request**. Remember, requests are subclasses of ASIHTTPRequest, so you can just put the code that manages the result of the request (typically, deserialization of JSON or XML into real objects, triggering other network connections, updating Core Data stores, etc). Putting the code into each separate request subclass, using a polymorphic method with a common name accross request classes, makes it very easy to debug and manage them.

In our example, the SBNetworkManager calls the "processResponse" method in each subclass. This method has an empty implementation in our category

[7]https://github.com/akosmasoftware/Senbei/blob/master/Classes/Helpers/SBNetworkManager/Requests/SBBaseRequest.h

[8]https://github.com/akosmasoftware/Senbei/blob/master/Classes/Helpers/SBNetworkManager/Requests/SBBaseFormDataRequest.h

[9]http://www.omnigroup.com/products/omnigraffle/

[10]diagram-large.png

for ASIHTTPRequest, and each individual subclass performs a different set of operations; some will parse XML, some will just post a notification; the separation of the logic in each subclass makes it easy to debug, document, maintain and extend the system.

Usage

Every time one of my controllers requires some data (refresh, viewDidAppear, etc), the network manager creates an instance of the required ASIHTTPRequest subclass, and then adds it to the queue.

Whenever a request finishes or fails, the network manager is called (remember, the network manager is the queue's delegate). In turn, the network manager calls a method on the request itself, delegating the task of the processing of the response to each subclass.

In Senbei, every method of the SBNetworkManager class just creates an instance of a dedicated SBBaseRequest subclass, and pushes it into the wrapped network queue.

Notifications

The network manager notifies the controllers above about interesting events using notifications; using a delegate protocol is not a good idea, because in your app you typically have many controllers talking to your network manager, and notifications are more flexible.

However, as with always with notifications, using them requires planning, naming conventions and documentation. Code using notifications might be complex to maintain, because the dependencies are not obvious at first hand; that's the price of their flexibility. In Senbei, notifications are all defined in the same file, so that different components can use the same constants throughout the application. The names of the notifications are clear and express the purpose and circumstance of each one.

Since iOS 4 there is also the possibility of using blocks as callback notifications, but then again, I think they just offer an alternative to delegate protocols; notifications are much more flexible, as many different objects can be notified of the same event (and receive the same information through userInfo dictionaries) at once.

Conclusion

This is how I've been writing many network-bound apps for the past few years, and frankly it has worked pretty well so far. I can extend the system horizontally, adding more ASIHTTPRequest subclasses as I need them, and the core of the network manager stays intact. The responsibilities is clearly separated, and the class and notification names give a pretty good idea of the purpose of each request.

One problem that I haven't yet solved with this architecture (and one that a commenter of my StackOverflow answer points out) is finding a way to test the system; probably using mock objects, we could simulate different network conditions, and integrate this knowledge with automated tests.

I hope that this architecture is useful to you too! I look forward to read your comments below.