RFC: Model services on iOS

Our current enum model `MonzoAPI` for API request generation is starting to get unwieldy, and difficult for developers unfamiliar with the codebase to locate relevant parts.

Additionally, our data models have started to bloat with the burden of having to deal with factory style request generation. Having the model serve as the jumping point to our network abstraction is starting to become messy. This RFC aims to address that design flaw.

Separation of Concerns

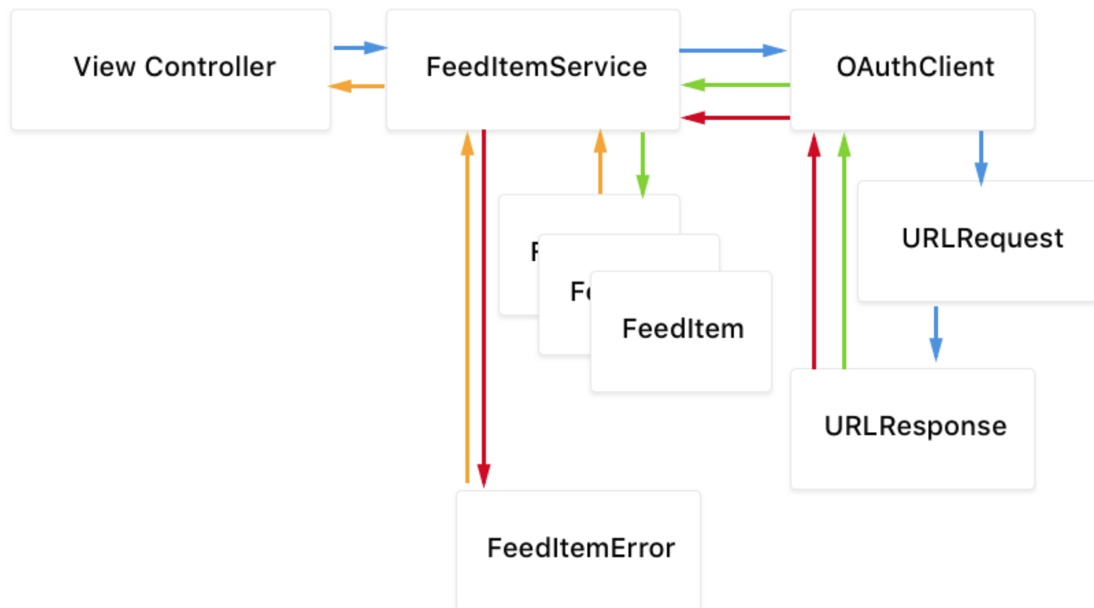
It's time to cleanly separate the concerns of managing a model object, which is domain specific to iOS, and the need to perform business logic related to these objects over the network against our backend services.

I'm proposing we divide our current model structure into Models and Services. To test this design, we'll start with a proof of concept. I'm suggesting we try this with a refactor of `Trip`, since it handles a non-critical path in the app, and is fairly lightweight. If we feel this design is successful, we'll adopt it for all future API driven work, and plan to refactor existing models that behave this way.

Service design

Services should be modelled as lightweight structs, mostly static vessels for hitting the network, and should encapsulate all domain specific information about the request, and the response.
Each individual request config would be housed within a moduled struct, conforming to an `APIRequest` protocol (based on our existing `OAuthClientRequestable`).

The protocol itself would deal with serialising the actual request, based on it's properties.



Blue: Request flow. Green: Raw success path. Red: Raw error path. Orange: Serialised return data.

What belongs in a service?

A service should act as a housing for API business logic for requests relating to models, specifically:

- Request data structures
- Public methods allowing the caller to fetch, and modify model objects via the API
- Errors relating to API services
- General response handling (IE, what to do with X status code)

It should not:

- Expose properties or methods to fetch cached objects from disk, or the Realm.
- Contain deserialisation logic to generate model objects from dictionaries. This should remain an initialiser on the relevant associated model.
- Deal with request signing, or queuing. This should remain with the `OAuthClient`.

Example

APIRequest.swift

```
enum RequestMethod: String {
```

```
    case POST = "POST"
    case GET = "GET"
    case PUT = "PUT"
    case DELETE = "DELETE"
    case HEAD = "HEAD"
    case PATCH = "PATCH"
}

protocol APIRequest: ErrorTrackable {
    var path: String { get }
    var method: RequestMethod { get }
    var parameters: [String: Any] { get }
}

extension APIRequest {
    var urlRequest: URLRequest {
        // Handle existing serialisation provided by MonzoAPI

        return URLRequest()
    }
}
```

Trip.swift

```
class Trip: BaseObject {
    // Existing realm model, and local changes to business logic
}
```

TripService.swift

```
struct TripService {
    static func fetch(id: String, completion: (Trip?, NSError?) -> Void)
{
        let client = OAuthClient.shared
        let request = Fetch(id: id)

        client.request(request) { response in
            // Handle response
        }
    }

```

```
11      static func updatePurpose(for id: String, purpose: String, completio
   n: ((Bool, NSError?) -> Void)) {
12          let client = OAuthClient.shared
13          let request = UpdatePurpose(id: id, purpose: purpose)
14  
15          client.request(request) { response in
16              // Handle response
17          }
18      }
19  }
20  
21  // MARK: - Fetching
22  
23  extension TripService {
24      struct Fetch: APIRequest {
25          let id: String
26  
27          init(id: String) {
28              self.id = id
29          }
30  
31          var method: RequestMethod = .GET
32          var path: String {
33              return "trip/\(id)"
34          }
35      }
36  }
37  
38  // MARK: - Updating
39  
40  extension TripService {
41      struct UpdatePurpose: APIRequest {
42          let id: String
43          let purpose: String
44  
45          var path: String {
46              return "trip/purpose"
47          }
48  
49          var method: RequestMethod {
```

```
                return .POST
            }

            var parameters: [String: Any] {
                return ["trip_id" : id, "purpose" : purpose]
            }
        }
    }
```

Example ViewController usage:

```
class TripViewController {
    var trip: Trip!

    convenience init(trip: Trip) {
        self.init()
        self.trip = trip

        TripService.fetch(id: trip.id) { trip, error in
            //TODO: Display trip
        }
    }
}
```

I can see a number of benefits for this, notably:

- Central module for defining API specific logic about each request (for example, error types)
- Clean separation of network vs cache logic (if it hits our API, it should be in a service)
- Lightweight and testable
- No messy "static" functions which couldn't be instance methods due to the way Realm works. We'd be separating Realm from our networking stack.
- Request configuration is grouped with the relevant service, rather than one giant enum.

Downsides:

- There's a bit of refactoring work to get here. However - we can start to move in this direction slowly, avoiding huge changes.
- We'll have more files. I'm not sure this is a real concern since Xcode is pretty good at search, and with the right convention (IE naming `Model + Service.swift`) it should be painless.

Testing

Currently, we have an abstraction for stubbing network requests, only linked to our test targets. I'd like to keep this design somewhat, since the inability to stub properly in the production code is desirable, and avoids nightmare regressions.

Given this, I'd suggest moving the existing `stub` methods to extensions on the protocol `APIRequest`, still within the test targets.