APPCOREKIT

Mappings

A simple way to
convert JSON to native objects

https://twitter.com/appcorekit

https://github.com/whererecloud/AppCoreKit

http://cocoapods.org/?q=appcorekit
pod AppCoreKit/Mapping

Motivation

Working with native objects presents several advantages. First, having classes with with properties allows to work in a type safe environment where, at any time, we know that an object's property has a specific type; we also benefit from code-completion and validation in XCode. Second, you can name your properly to fit with your goals. Finally, native objective-C properties, by their very nature, are full of interesting informations to automate UI components and provide many synchronization systems like KVO or bindings.

The models we usually manipulate when developing an application are not just business models. business models. An important distinction to note is that they also serve as a hub to synchronize our UI. In a UI environment, we will need some properties for managing states that are not part of the business logic. The way we will structure our models in the application will probably differ from persistent business models, because UI models must be thought and designed to help displaying and synchronizing data with the UI components we are using.

The business models often exists on a remote server that we query via a REST API. In fact, querying a web service consists in:

- Populating our UI models using the remote business models
- Working with our UI Models in memory
- Updating the UI by displaying the right data.

By developing the mapping system we wanted to achieve several goals. First and foremost, we wanted to formalize the recurrent pattern of consuming JSON payloads: iterating through dictionaries and arrays, creating native objects and converting the content of the JSON data to the right types in our properties.

Validation of these payloads is very important for the client application to know what's ok and whats wrong with the JSON content. Especially when a third-party developer or company is responsible for the API. We wanted to be able to mark some JSON fields as required or optional, as well as define default values for the native objects' properties in case the optional value is not present. We also wanted to get a summary of what's missing in the payloads after our native objects have been created or populated.

Converting data from one type to another is an overhead we wanted to get rid of. By leveraging the iOS runtime, we know what kind of conversion should occur to set a value from the JSON objects to our native Objective-C properties. And being able to specify custom transformers during mappings is also important as we usually encounter particular use cases in every APIs.

The mapping API allows to retrieve new instances or populate existing instance of objects or array of objects.

The Mappings API

The mapping API is separated in two parts.

1. Declare how to map data from one side to the other. This is named a mapping context. We generally have one mapping context per type of payload or native object we'd like to convert. Each mapping context has a unique identifier, to be able to setup and get it when needed. Mapping context can either be defined programmatically or using the .mapping file syntax.

2. Transform the JSON payload using the mapping context, using several methods which allow to get newly created native objects or populate existing ones.

Defining the mappings

The .mappings Syntax

The .mappings files allow to centralize and externalize mapping context definition outside from the code. The contexts are represented by a JSON dictionary which is light and easy to read.

Mappings files provide some mechanisms to organize and simplify the mappings contexts definition such as:

1. Importing .mappings files so that your can work with a small amount of contexts in each files.
2. Defining templates that can be reused in several parts of your context definitions by using multiple inheritance.

The root .mappings file must be loaded prior to using the mapping APIs as the contexts need to be declared first. This can be done in the AppDelegate init method or when you instantiate your web service responsible for fetching and converting the data. The mappings files must be present in the source tree and associated to the application target to get copied as resources in the final app.

```
@interface CKMappingContext : NSObject

+ (void)loadContentOfFileNamed:(NSString*)name;

@end
```

Here is a sample loading contexts from *YourMappings.mappings* file:

```
[CKMappingContext loadContentOfFileNamed:@"YouMappings"];
```

Mappings context

To best explain how to define contexts for creating and populating native objects, let's consider a real example. We'll use the Twitter API payloads as a datasource since it is public.

First, defines the tweet models:

```
@interface TweetModel : CKObject
@property(nonatomic,copy) NSURL*    imageUrl;
@property(nonatomic,copy) NSString* name;
@property(nonatomic,copy) NSString* message;
@property(nonatomic,copy) NSString* identifier;
@end
```

Fetching tweets from the Twitter API return the following payload for each tweet:

https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline

```
1.  {
2.      "coordinates":[-97.51087576,35.46500176],
3.      "id_str": "240859602684612608",
4.      "text": "Introducing the Twitter Certified Products Program: https://t.co/
    MjJ8xAnT",
5.      "user": {
6.        "name": "Twitter API",
7.        "profile_image_url": "http://a0.twimg.com/profile_images/
    2284174872/7df3h38zabcvjylnyfe3_normal.png",
8.       }
9.  }
```

We now need to define a context for creating `TweetModel` from the payload above and populating the `TweetModel` properties.

Lets start by defining an identifier for this context in the .mappings file:

```
{
    "$TweetModel" : {
        //Mapping context for converting TweetModel from the twitter API
    }
}
```

Prefixing contexts by "$" is optional but this is a convention we like to use for identifying contexts or templates.

Native objects class

Defines the class of the objects that will be created when mapping the JSON payload to native objects using the `@class` keyword :

```
{
    "$TweetModel" : {
        "@class" : "TweetModel"
    }
}
```

Direct mappings

Lets define the mappings with direct and automatic conversion from the JSON fields to our newly created TweetModel object:

```
{
    "$TweetModel" : {
        "@class"     : "TweetModel",
        "imageUrl"   : "user.profile_image_url",
        "name"       : "user.name",
        "message"    : "text",
        "identifier" : "id_str"
    }
}
```

On the left side (keys) of the mapping context dictionary, we specify the native object property to which we want to convert the value. On the right side (values) we define the keypath in the JSON payload representing the field we want to get and convert to the specified property.

In the sample above, the user.profile_image_url from the JSON payload is a string. When mapping it to TweetModel.imageUrl, this will automatically be converted to an NSURL. The mapping system is implemented on top of the AppCoreKit conversion APIs that defines type to type transformations for most of the cases that are needed. To have the full details concerning the conversion system and the conversions that are supported, you ca refer to the "*AppCoreKit - Conversion*" documentation.

Validation attributes

By default, all the mappings defined above are considered as required. This means that when applying the mappings from the JSON payload, an error will be generated each time one of the properties' JSON field is not present in the payload. The error can be optionally used as a result of applying mappings with one of the several mappings APIs.

Lets consider that the imageUrl is optional in this payload so that no error will be created if `"user.profile_image_url"` is not present in the JSON content. As a replacement, we'll define a default url acting as a placeholder so that we still have a default image displayed in the UI displaying TweetModel.

```
{
    "$TweetModel" : {
        "@class"     : "TweetModel",

        "imageUrl" : {
            "@keyPath"      : "user.profile_image_url",
            "@optional"     : 1,
            "@defaultValue" : "https://si0.twimg.com/profile_images/1251071930/
Untitled.png"
        },

        "name"       : "user.name",
        "message"    : "text",
        "identifier" : "id_str"
    }
}
```

The imageUrl property mapping now defines the keypath in the JSON payload by using the `@keyPath` keyword as it is associated to additional mapping attributes.

The `@optional` keyword defines a boolean attribute specifying that the `"user.profile_image_url"` field is not a required field.

The `@defaultValue` keyword specifies the default value that should be converted in the imageUrl property in case the `"user.profile_image_url"` field is not present in the JSON payload.

Custom transformers

Sometimes, properties will need a custom transformer to convert one or several fields in the JSON payload to the right type matching the property. This can easily be done by specifying both a class and a class method, that will get called when transforming the JSON content to the property.

A sample in the tweet payload would be handling the `"coordinates"` fields of the JSON payload and transform it from a CGFloat array to a `CLLocationCoordinate2D` value using a custom transformer method.

```
@interface TweetModel : CKObject
@property(nonatomic,copy) NSURL* imageUrl;
@property(nonatomic,copy) NSString* name;
@property(nonatomic,copy) NSString* message;
@property(nonatomic,copy) NSString* identifier;
@property(nonatomic,assign) CLLocationCoordinate2D coordinates;
@end
```

```
{
    "$TweetModel" : {
        "@class"     : "TweetModel",
        "imageUrl"   : "user.profile_image_url",
        "name"       : "user.name",
        "message"    : "text",
        "identifier" : "id_str",

        "coordinates" : {
            "@keyPath"           : "coordinates",
            "@transformClass"    : "MyTransformers",
            "@transformSelector" : "coordinate2DFromFloatArray:error:"
        }
    }
}
```

```
@interface MyTransformers : NSObject
@end

@implementation MyTransformers

+ (CLLocationCoordinate2D)coordinate2DFromFloatArray:(NSArray*)array
                                               error:(NSError**)error{

    if([array count] != 2){
        *error = [NSError errorWithDomain:@"MyTransformersErrorDomain"
                                     code:0
                                 userInfo:nil];
        return CLLocationCoordinate2DMake(0,0);
    }

    id value0 = [array objectAtIndex:0];
    id value1 = [array objectAtIndex:1];

    if(![value0 isKindOfClass:[NSNumber class]]
    || ![value0 isKindOfClass:[NSNumber class]]){
        *error = [NSError errorWithDomain:@"MyTransformersErrorDomain"
                                     code:0
                                 userInfo:nil];
        return CLLocationCoordinate2DMake(0,0);
    }

    return CLLocationCoordinate2DMake([value0 floatValue],
                                      [value1 floatValue]);

}

@end
```

The `@transformClass` keyword defines the class defining the transform selector class method.

The `@transformSelector` keyword defines the transform selector that will be performed to convert the specified JSON field to the specified property. The method signature for this selector must respect certain rules:

- The first method attribute is the raw value from the JSON payload (NSString, NSNumber, NSDictionary, NSArray, NSNull)

- The second is an NSError** that can be created by the transform method and that will be available after the mapping APIs have been called.

Custom transformers with additional user data

Sometimes it is useful to use the same transformer method by passing additional attributes to perform the transformation. For example, we would like to transform a dictionary containing two different fields (eg. longitude, latitude) to a single `CLLocationCoordinate2D` value. The keypath of both fields could be specified in the mappings and used by the transformer method to get the value from the raw JSON data and convert it to the right type.

Imagine that the coordinates are defined as follows in the JSON payload:

```
"coordinates": {
    “long” : -97.51087576,
    “lat” : 35.46500176
}
```

In order to convert this to a `CLLocationCoordinate2D` value, we would like to use a generic which transforms a dictionary value to a `CLLocationCoordinate2D` value. Considering two approaches: first, assume that all the dictionaries representing a location have “long” and “lat” fields defined. In this case, we can just a have a transform method handling this particular case. However, if we want to handle arbitrary payloads with dictionaries representing location, each with variable fields for longitude or latitude, we can handle it as follows:

```
+ (CLLocationCoordinate2D)coordinate2DFromDictionary:(NSDictionary*)dictionary
                                            userData:(NSDictionary*)userData
                                               error:(NSError**)error{

    NSString* longitudeKeyPath = [userData objectForKey:@"longitude"];
    NSString* latitudeKeyPath  = [userData objectForKey:@"latitude"];

    if([dictionary objectForKey:longitudeKeyPath] == nil
    || [dictionary objectForKey:latitudeKeyPath] == nil){
        *error = [NSError errorWithDomain:@"MyTransformersErrorDomain"
                                     code:0
                                 userInfo:nil];
        return CLLocationCoordinate2DMake(0,0);
    }

    id longitude = [dictionary objectForKey:longitudeKeyPath];
    id latitude  = [dictionary objectForKey:longitudeKeyPath];
    if(![longitude isKindOfClass:[NSNumber class]]
    || ![latitude isKindOfClass:[NSNumber class]]){
        *error = [NSError errorWithDomain:@"MyTransformersErrorDomain"
                                     code:0
                                 userInfo:nil];
        return CLLocationCoordinate2DMake(0,0);
    }

    return CLLocationCoordinate2DMake([latitude floatValue],[longitude floatValue]);
```

```
}
```

```
{
   "$TweetModel" : {
      "@class"     : "TweetModel",
      "imageUrl"   : "user.profile_image_url",
      "name"       : "user.name",
      "message"    : "text",
      "identifier" : "id_str",

      "coordinates" : {
        "@keyPath"           : "coordinates",
        "@transformClass"    : "MyTransformers",
        "@transformSelector" : "coordinate2DFromDictionary:userData:error:",
        "@transformUserData" : {
            "longitude" : "long",
            "latitude"  : "lat"
        }
      }
   }
}
```

The user data can be any type supported by the JSON format (NSString, NSNumber, NSDictionary, NSArray, NSNull) and provides a very flexible way of reusing custom transform methods.

The @self Keyword

The @self keyword allow to pass the root JSON object being mapped without a specific keypath. This is generally useful as a combination with custom transformer.

Lets take the same example of transforming the coordinates from the tweet payload but imagine that the longitude and latitude are defined by two JSON fields that are not encapsulated in a dictionary but as children of the root dictionary:

```
10.  {
11.      "long":-97.51087576,
12.      "lat" :-35.46500176,
13.      "id_str": "240859602684612608",
14.      ...
15.  }
```

In this case, we would be able to reuse the custom transformer with user data that we defined previously by specifying the root dictionary as the JSON data to be transformed. This can be done as follow:

```
{
   "$TweetModel" : {
      "@class"     : "TweetModel",
      "imageUrl"   : "user.profile_image_url",
      "name"       : "user.name",
      "message"    : "text",
      "identifier" : "id_str",

      "coordinates" : {
        "@keyPath"           : "@self",
        "@transformClass"    : "MyTransformers",
        "@transformSelector" : "coordinate2DFromDictionary:userData:error:",
        "@transformUserData" : {
            "longitude" : "long",
            "latitude"  : "lat"
        }
      }
   }
}
```

By specifying `@self` as `@keyPath` for the coordinates property, the root dictionary will be used as the value to be converted when performing the `coordinate2DFromDictionary:userData:error:` selector. And as this transformer handle user data to get the longitude and the latitude from the input dictionary, the coordinate property will be set correctly by getting the “long” and “lat” fields from the root dictionary.

Object property mapping

In the samples above, we saw how to map single value properties from JSON data. Models can also have objects or collections of objects as properties, that require a class and mappings to be defined.

To illustrate how to achieve this, lets separate the UserModel into a different class and add a UserModel property into the TweetModel class.

```
@interface UserModel : CKObject
@property(nonatomic,copy) NSURL* imageUrl;
@property(nonatomic,copy) NSString* name;
@end

@interface TweetModel : CKObject
@property(nonatomic,retain) UserModel* user;
@property(nonatomic,copy)   NSString* message;
@property(nonatomic,copy)   NSString* identifier;
@property(nonatomic,assign) CLLocationCoordinate2D coordinates;
@end
```

Mapping the same payload to this new TweetModel object can be done as follows:

```
{
    "$TweetModel" : {
        "@class" : "TweetModel",

        "user" : {
            "@keyPath" : "user",
            "@object"  : {
                "@class" : "UserModel",
                "@mappings" : {
                    "name"     : "name",
                    "imageUrl" : "profile_image_url"
                }
            }
        },

        "message" : "text",
        "identifier" : "id_str",
        "coordinates" : {
            "@keyPath" : "@self",
            "@transformClass" : "MyTransformers",
            "@transformSelector" : "CLLocationCoordinate2DFromFloatArray:error:",
            "@transformUserData" : {
                "longitude" : "long",
                "latitude"  : "lat"
            }
        }
    }
}
```

Or you can define a context for mapping a UserModel from a User payload and tell the TweetModel user property to be mapped using this context as follows:

```
{
    "$UserModel" : {
        "@class"   : "UserModel",
        "name"     : "name",
        "imageUrl" : "profile_image_url"
    },

    "$TweetModel" : {
        "@class" : "TweetModel",
        "user" : {
            "@keyPath" : "user",
            "@object"  : {
                "@mappings" : "$UserModel"
            }
        },
        "message" : "text",
        "identifier" : "id_str",
        "coordinates" : {
            "@keyPath" : "@self",
            "@transformClass" : "MyTransformers",
            "@transformSelector" : "CLLocationCoordinate2DFromFloatArray:error:",
            "@transformUserData" : {
                "longitude" : "long",
                "latitude"  : "lat"
            }
        }
    }
}
```

The `@object` keyword allows to define the `@class` of the object that will be created and set as the property, as well as the `@mappings` context allowing to convert the JSON content identified by the `@keyPath` value. The mappings can be defined inline (as in the first example), or refer to a, arbitrary mapping context using its identifier (as in the second example). When referencing an arbitrary mapping context, we must ensure that this context specifies the class of the native object to be created, or you can specify it using the `@class` keyword as follows:

```
"user" : {
    "@keyPath" : "user",
    "@object"  : {
         "@class"    : "UserModel",
         "@mappings" : "$UserModel"
     }
}
```

Collection property mapping

Mapping a collection property means defining a class and a mapping context which create objects for each entry in the JSON array that is being converted. In the end this is the same as mapping a single object property and we use the exact same syntax to do it. The `@object` keyword allows to define the attributes that will get applied to each object from the JSON array. We must ensure that the collection or array property is properly instantiated prior to create objects in this collection during the mappings.

To illustrate this, let's create a TimelineModel class that will map an array of Tweet payloads into its tweets collection.

```
16. [
17.    {  //Tweet payload
18.    },
19.    ...
20. ]
```

```
@interface TimelineModel : CKObject
@property (nonatomic, retain) NSArray* tweets; //Of type TweetModel
@end
```

The mapping context for converting the array of tweet payloads to native TimelineModel by creating TweetModel objects is defined as follows:

```
"$TimelineModel" : {
    "@class" : "TimelineModel",
    "tweets" : {
        "@keyPath" : "@self", //The tweet payload array
        "@object"  : {
            "@mappings" : "$TweetModel"
        }
    }
}
```

If you're mapping an existing instance that already has content in its collection, you can specify additional attributes to customize how this collection must be filled.
By using the `@clearContent` keyword, you can tell if the collection must be cleared prior to populate it with new instances.
By default, new objects are inserted at the end of the collection. You can optionally specify if they must be inserted at the beginning of the collection using the `@insertContentAtBegin` keyword. Objects that are inserted always keep the order of the JSON content.

For example, if you map an existing TimelineModel from a new JSON array, you can customize the insertion of the tweets as follow:

```
"$TimelineModel" : {
    "@class" : "TimelineModel",
    "tweets" : {
        "@keyPath" : "@self", //The tweet payload array
        "@clearContent" : 1,
        "@insertContentAtBegin" : 1,
        "@object" : {
            "@mappings" : "$TweetModel"
        }
    }
}
```

Knowing that setting both attributes at the same time has no sense because if the content is cleared, insertContentAtBegin will be the default behavior.

Organizing the context definitions

The .mappings file syntax is based on the CKCascadingTree class allowing to organize your mappings using different files, inheritance, device and os version selectors. As several AppCoreKit technologies relies on this CKCascadingTree, it is explained in a separate document named "*AppCoreKit - Cascading Tree*".

The reserved keywords for organizing your mappings contexts definitions are :

```
@import
@inherits
@ipad
@iphone
@iphone5
@ios operator version
```

Programmatically

You can also define mappings context programmatically using the following APIs. We'll not enter into details as using the .mappings files is the preferred way to use mappings and provides a more efficient way to organize and keep the context definition smaller.

```
@interface CKMappingContext : NSObject

+ (CKMappingContext*)contextWithIdentifier:(id)identifier;

- (void)setObjectClass:(Class)type;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath
transformBlock:(id(^)(id source))transformBlock;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath
transformTarget:(id)target action:(SEL)action;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath
defaultValue:(id)value;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath
defaultValue:(id)value transformBlock:(id(^)(id source))transformBlock;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath
defaultValue:(id)value transformTarget:(id)target action:(SEL)action;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath optional:
(BOOL)optional;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath optional:
(BOOL)optional transformBlock:(id(^)(id source))transformBlock;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath optional:
(BOOL)optional transformTarget:(id)target action:(SEL)action;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath
objectClass:(Class)objectClass withMappingsContextIdentifier:(id)contextIdentifier;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath optional:
(BOOL)optional objectClass:(Class)objectClass withMappingsContextIdentifier:
(id)contextIdentifier;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath
withMappingsContextIdentifier:(id)contextIdentifier;

- (void)setKeyPath:(NSString*)keyPath fromKeyPath:(NSString*)sourceKeyPath optional:
(BOOL)optional withMappingsContextIdentifier:(id)contextIdentifier;

@end
```

Converting JSON to native objects

This requires to follow the three steps bellow:

1. Parse the JSON payload
2. Query the mappings context
3. Apply the mappings

Parsing the JSON payload

First, we need to get a dictionary or array representation of the JSON data to be able to convert it to native objects using mappings.

AppCoreKit provides convenient methods to achieve this:

```
@interface NSObject (CKNSObjectJSON)

+ (id)objectFromJSONData:(NSData *)data;

+ (id)objectFromJSONData:(NSData *)data error:(NSError **)error;

@end
```

Sample Usage:

```
NSData* jsonData = [self YourJSONData];

NSError* error = nil;
id jsonObject = [NSObject objectFromJSONData:jsonData error:&error];
if(error){
    //Handle the error
}else{
    //Do something
}
```

Retrieving the mapping context

```
@interface CKMappingContext : NSObject

+ (CKMappingContext*)contextWithIdentifier:(id)identifier;

@end
```

Sample Usage :

```
CKMappingContext* timelineMappingsContext =
     [CKMappingContext contextWithIdentifier:@"$TimelineModel"];
```

Applying the mappings

Depending of the kind of data you've got as input, the way you defined your context and the kind of output you want, you will use one of the following methods on the mappings context:

```
@interface CKMappingContext : NSObject

- (id)objectFromValue:(id)value error:(NSError**)error;

- (NSArray*)objectsFromValue:(id)value error:(NSError**)error;

- (id)mapValue:(id)value toObject:(id)object error:(NSError**)error;

@end
```

Sample Usage:

1. Getting new TweetModel instance from a tweet JSON dictionary

```
NSDictionary* tweetPayload; //get the payload
CKMappingContext* tweetMappingsContext = [CKMappingContext
contextWithIdentifier:@"$TweetModel"];

NSError* error = nil;
TweetModel* tweet = [tweetMappingsContext objectFromValue:tweetPayload error:&error];
```

2. Getting new TimelineModel from an array of tweet dictionary

```
NSArray* tweetPayloads; //get the payloads
CKMappingContext* timelineMappingsContext = [CKMappingContext
contextWithIdentifier:@"$TimelineModel"];

NSError* error = nil;
TimelineModel* timeline = [tweetMappingsContext objectFromValue:tweetPayloads
error:&error];
```

3. Getting an array of TweetModel from an array of tweet dictionary

```
NSArray* tweetPayloads; //get the payloads
CKMappingContext* tweetMappingsContext = [CKMappingContext
contextWithIdentifier:@"$TweetModel"];

NSError* error = nil;
NSArray* tweets = [tweetMappingsContext objectsFromValue:tweetPayloads error:&error];
```

4. Populating an existing TweetModel with more details from a tweet JSON dictionary

```
NSDictionary* tweetPayload; //get the payload
CKMappingContext* tweetMappingsContext = [CKMappingContext
contextWithIdentifier:@"$TweetModel_Details"];

TweetModel* ExistingTweet; //get the existing tweet
NSError* error = nil;
[tweetMappingsContext mapValue:tweetPayload toObject:ExistingTweet error:&error];
```

Fetching native objects from a REST API

The AppCoreKit provides a fully multi-threaded network stack with lot of helper for creating and executing http web requests. This is fully documented in the "*AppCoreKit - Network*" document. Here we'll present the two helper methods allowing to easily fetch new native objects or populate existing native objects from a REST API asynchronously.

For both of these methods, fetching the data, parsing the JSON and applying the mappings is done on the web thread. Completion and error blocks are called on the UI thread.

We must ensure the server returns "Content-Type" : "application/json" to automatically parse the JSON payload prior to apply mappings.

Fetching an array of native objects

```
@interface CKWebRequest (StandardRequests)

+ (CKWebRequest*) requestForObjectsWithUrl:(NSURL*)url
                                    params:(NSDictionary*)params
                                      body:(NSData*)body
                         mappingContextIdentifier:(NSString*)mappingIdentifier
                                 transformRawData:(NSArray*(^)(id value))
                                                  transformRawDataBlock
                                       completion:(void(^)(NSArray* objects))completionBlock
                                            error:(void(^)(id value, NSHTTPURLResponse*
                                                  response, NSError* error))errorBlock;

@end
```

url : This represents the base url for your request without the params.

params : This is a dictionary of string to string param to value. You can set this attribute to nil if you do not have params in your request. Params will be automatically encoded and formatted to match the URL specifications.

body : If you need to post data, pass this data as body. The request will automatically sets its method to POST. You can set this attribute to nil if you don't want to post data. In this case the method is automatically set as GET.

mappingIdentifier : This is the mappings context identifier that will get used to create native objects from the JSON array objects. This mapping context must represent the conversion from a single object dictionary to a native object.

transformRawData : The mappings must be applied using an array as value. The transformRawData block allow to get the specific array in the incoming JSON payload. If the payload is already the array that must be converted, implements the block and simply return value. If the payload is a dictionary, return [value objectForKey:@"MyArray"];

completionBlock : The completion block is called when the content has been fetched on the web and the JSON has be converted to native objects. The

```
                    objects receive in this blocks are the native objects after the
                    mappings have been applied.

error               : The error block is called if we reached a connection or HTTP
                    error.
```

Sample Usage : Fetching an array of TweetModels from one of the Twitter timeline APIs.
https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline

```
NSURL* url = [NSURL URLWithString:@"https://api.twitter.com/1.1/statuses/
user_timeline.json"];

NSDictionary* params = @{ @"screen_name" : @"seb_morel",
                          @"count"       : @"20" };

CKWebRequest* request =
    [CKWebRequest requestForObjectsWithUrl:url
                                    params:params
                                      body:nil
                   mappingContextIdentifier:@"$TweetModel"

                           transformRawData: ^NSArray *(id value) {

                                   return (NSArray*)value;

                               } completion: ^(NSArray *objects) {
                                   //Do something with the new TweetModels in objects
                               } error: ^(id value,
                                          NSHTTPURLResponse *response,
                                          NSError *error) {
                                   //Handle error
                               }];
[[CKWebRequestManager sharedManager]scheduleRequest:request];
```

Fetching a single native object

```
@interface CKWebRequest (StandardRequests)


+ (CKWebRequest*)requestForObject:(id)object
                              url:(NSURL*)url
                           params:(NSDictionary*)params
                             body:(NSData*)body
          mappingContextIdentifier:(NSString*)identifier
                  transformRawData:(NSDictionary*(^)(id value))transformRawDataBlock
                        completion:(void(^)(id object))completionBlock
                             error:(void(^)(id value, NSHTTPURLResponse* response,
                                    NSError* error))errorBlock;

@end
```

object : The existing instance that must be populated using the incoming JSON payload. If you want to fetch a new object, you must pass nil and ensure that your mapping context defines the class of the object that must be instantiated.

url : This represents the base url for your request without the params.

params : This is a dictionary of string to string param to value. You can set this attribute to nil if you do not have params in your request. Params will be automatically encoded and formatted to match the URL specifications.

body : If you need to post data, pass this data as body. The request will automatically sets its method to POST. You can set this attribute to nil if you don't want to post data. In this case the method is automatically set as GET.

mappingIdentifier : This is the mappings context identifier that will get used to create native objects from the JSON array objects. This mapping context must represent the conversion from a single object dictionary to a native object.

transformRawData : The mappings must be applied using an array as value. The transformRawData block allow to get the specific array in the incoming JSON payload. If the payload is already the array that must be converted, implements the block and simply return value. If the payload is a dictionary, return [value objectForKey:@"MyArray"];

completionBlock : The completion block is called when the content has been fetched on the web and the JSON has be converted to native objects. The objects receive in this blocks are the native objects after the mappings have been applied.

error : The error block is called if we reached a connection or HTTP error.

Sample Usage : Fetching a PlaceModel from the Twitter API.
https://dev.twitter.com/docs/api/1.1/get/geo/id/%3Aplace_id

Lets say that we have a placeId from example "`df51dec6f4ee2b2c`", and we defined PlaceModel class and a $PlaceModel mapping context and we want to fetch a new populated PlaceModel.

```
NSString* placeId = @"df51dec6f4ee2b2c";

NSString* urlString = [NSString stringWithFormat:@"https://api.twitter.com/1.1/geo/id/
%@.json", placeId];

NSURL* url = [NSURL URLWithString:urlString];

CKWebRequest* request = [CKWebRequest requestForObject:nil
                                                   url:url
                                                params:nil
                                                  body:nil
                                 mappingContextIdentifier:@"$PlaceModel"

                                        transformRawData:^NSDictionary *(id value) {

                                            return (NSDictionary*)value;

                                        } completion:^(id object) {
                                            //Do something with the new PlaceModel
                                        } error:^(id value,
                                                  NSHTTPURLResponse *response,
                                                  NSError *error) {
                                            //Handle error
                                        }];
[[CKWebRequestManager sharedManager]scheduleRequest:request];
```

If we already had a PlaceModel with the property placeId = `df51dec6f4ee2b2c` and we want to fetch more details for this instance:

```
PlaceModel* placeModel; //get you place model

NSString* urlString = [NSString stringWithFormat:@"https://api.twitter.com/1.1/geo/id/
%@.json", placeModel.placeId];

NSURL* url = [NSURL URLWithString:urlString];

CKWebRequest* request = [CKWebRequest requestForObject:placeModel
                                                   url:url
                                                params:nil
                                                  body:nil
                                 mappingContextIdentifier:@"$PlaceModel"

                                        transformRawData:^NSDictionary *(id value) {

                                            return (NSDictionary*)value;

                                        } completion:^(id object) {

                                            //object is the place model passed as
                                            //object to the request after it has
                                            //been populated using the content of
                                            //the request

                                        } error:^(id value,
                                                  NSHTTPURLResponse *response,
                                                  NSError *error){
                                            //Handle error

                                        }];
[[CKWebRequestManager sharedManager]scheduleRequest:request];
```