Hydrant Documentation

Release 2.0.0

Jeff Hui

June 22, 2017

Contents

Hydrant is a simple object serializer and data mapper library. Its goal is to parse JSON into value objects that you can safely use throughout your application. All the input validation and error checking can be done through Hydrant.

New? *Install Hydrant* and then jump in with *getting started*.

Topical guides after getting started

- Learn how Hydrant's *error handling* system works
- Read up on *mapping techniques* for some parsing scenarios that Hydrant can handle.
- Understand *Hydrant's design*
- Learn how to *write your own mappers*.

Hydrant's Public API reference

- *HYDMappers*
- *HYDAccessors*
- *NSFormatters*
- *NSValueTransformers*

About Hydrant's Development

- Review *what's changed* between versions.
- Read the *Contributor's Guide*.
- Bugs? File bugs on GitHub. Don't know the best way? Read *filing bugs*.

CHAPTER 1

Table of Contents

Installation

If you're using CocoaPods, you can pull in Hydrant by adding this to your pod file:

```
# this will pull any patch version with 'pod update'
pod "Hydrant", '~>2.0.0'
```

Or submodule the project into your project:

```
git submodule add https://github.com/jeffh/Hydrant.git <path/for/Hydrant>
```

And then add `libHydrant` to your dependencies and <./Hydrant/Public/> to your header include path. At this point in time, importing individual headers that are not `Hydrant.h` is not safe.

Getting Started

Hydrant is designed to be highly flexible in parsing JSON or any other structured data (eg - structured NSArrays, NSDictionaries) into Value Objects for your application to use. Hydrant can perform validations to ensure the data coming in is what you expect when doing these transformations.

This doesn't have to be just JSON. Parsing XML or converting values objects to your views and back is possible, but this tutorial will focus on JSON.

Enough talk, it's easier to see the usefulness with some code examples.

The Problem

Let's look at some json data we just parsed from NSJSONSerialization:

```
id json = @{
    @"first_name": @"John",
    @"last_name": @"Doe",
    @"homepage": @"http://example.com",
    @"age": @24,
    @"children": @[
        @{@"first_name": @"Ann",
          @"last_name": @"Doe",
          @"Age": 6},
        @{@"first_name": @"Bob",
```

```
            @"last_name": @"Doe",
            @"age": 6},
    ]
};
```

And we want to convert it to our person object:

```
@interface Person : NSObject
@property (copy, nonatomic) NSString *firstName;
@property (copy, nonatomic) NSString *lastName;
@property (strong, nonatomic) NSURL *homepage;
@property (assign, nonatomic) NSInteger age;
@property (copy, nonatomic) NSArray *children; // array of Person objects
@end

@implementation Person
@end
```

How can we parse this with Hydrant?

Serializing with Hydrant

Let's see how you can solve it via Hydrant:

```
id<HYDMapper> mapper = HYDMapObject([NSDictionary class], [Person class],
                                    @{@"first_name": @"firstName",
                                      @"last_name": @"lastName",
                                      @"homepage": @[HYDMapStringToURL(), @"homepage"],
                                      @"age": @"age",
                                      @"children": @[HYDMapArrayOf(HYDMapObject([NSDictionary class],
                                                                                @{@"first_name": @"fi
                                                                                  @"last_name": @"las
                                                                                  @"age": @"age"})
                                                                           @"children"]);

HYDError *error = nil;
Person *john = [mapper objectFromSourceObject:json error:&error];
if ([error isFatal]) {
    // do error handling
} else {
    // use john ... it's valid
}
```

At first glance, that's a lot of indentation! It's easy to break this into several variables for readability. But we're doing this to easily see the code flow of function calls for a function-by-function breakdown.

This is a declarative way to define how Hydrant should map fields from one object to another. We're defining a schema of the JSON structure we're expecting to parse. Let's break it down:

- The first *HYDMapObject* is a helper function that constructs an object for us to use. The function takes 4 arguments: an id, two classes, and a dictionary. The dictionary's keys correspond to the **first class** while the value corresponds to the **second class**. This defines a mapping from an NSDictionary to a Person class. So it's key will map in the same direction. The values can be strings or other objects that support the *Mapper* protocol.
- *HYDMapStringToURL* is another helper function that constructs a HYDMapper object. It converts strings into `NSURLs` for our `Person` class.
- *HYDMapCollectionOf / HYDMapArrayOf* is yet another helper function that constructs another `HYDMapper` object. It takes an argument of another `HYDMapper` and uses it to parse an array of objects.

- Now the second *HYDMapObject*. But now the first argument becomes obvious, it provides the destination of the results of the operation – in this example, to the children property.
- `[mapper objectFromSourceObject:json error:nil]` This actually does the conversion on the given JSON data structure and produces a Person class. The mapper will produce an error if the parsing failed. This method comes from the *Mapper* protocol.
- `[error isFatal]` This checks the `HYDError` for fatalness. Hydrant has two notions of errors: fatal and non-fatal errors. Fatal errors are given when the object could not be produced under the given requirements. Non-fatal errors indicate alternative parsing strategies have occurred to produce the object returned. We'll cover more of this shortly.

The `mapper` object can be reused for parsing that same JSON structure to produce Person objects. So after the construction, it can be memoized.

All helper functions that produce `HYDMapper` are prefixed with `HYDMap` for easy auto-completing goodness.

Why not manully parse the JSON?

Let's take a short aside to talk about the go-to solution - parsing it manually. Here's an example of parsing the JSON we got manually:

```
Person *johnDoe = [Person new];
johnDoe.firstName = json[@"first_name"];
johnDoe.lastName = json[@"last_name"];
johnDoe.age = [json[@"age"] integerValue];

NSMutableArray *children = [NSMutableArray arrayWithCapacity:[json[@"children"] count]];
for (NSDictionary *childJSON in json[@"children"]) {
    Person *child = [Person new];
    child.firstName = childJSON[@"first_name"];
    child.lastName = childJSON[@"last_name"];
    child.age = [childJSON[@"age"] integerValue];
    [children addObject:child];
}

johnDoe.children = children;
```

Not too bad. But what's are assumptions here? **We're assuming the structure of the JSON.** Easy if you happen to control the source of this JSON, but what if we don't? Someone could easily change the JSON to:

```
id json = @[];
```

Or something less nefarious, but may potentially happen:

```
id json = @{
    @"first_name": @"John",
    @"last_name": @"Doe",
    @"homepage": [NSNull null],
    @"age": [NSNull null],
    @"children": [NSNull null]
};
```

That's now going to crash your program when you try to treat NSNull as another object you expected (`NSArray`, `NSNumber`, `NSString`). Last time I checked no one liked crashes. And writing all the proper guard code starts becoming error-prone, boring, and adds a lot of noise to your code.

But wait, you don't need to error check anything! Then you don't need to use Hydrant. Simple as that. No hard feelings that you're not using my library.

Error Handling

Of course, if you don't know when Hydrant failed to parse something that's just as unhelpful. So Hydrant mappers return errors, which can be used to handle errors when parsing the source object. There are three states after the mapper parses the source object:

```
HYDError *error = nil;
Person *john = [mapper objectFromSourceObject:json error:&error];
if ([error isFatal]) {
    // do error handling
} else {
    if (error) {
        // log the non-fatal error.
    }
    // use john ... it's valid
}
```

Checking for `-[HYDError isFatal]` is usually the only check you need to perform in practice. Hydrant errors inherit from `NSError`.

Hydrant errors contain a lot of state of the library when parsing fails. These include the source object (or partial object being parsed), any internal errors, other mapper errors, fatalness, and properties being mapped to and from. They're all stored in userInfo, as `HYDError` just provides convenient methods.

> **Warning:** Since Hydrant errors store a lot of information about the source object, **you might leak sensitive information from the source object** (eg - user credentials) if you transfer the `error.userInfo` over the network.

So when would errors occur? Here's some examples from our mapper object we defined:

- Hydrant fails to convert the incoming object to an NSURL for homepage, such as a trying to use a non-NSString.
- Any element in the incoming children array fails to parse.
- Any of the specified keys are nil or NSNull.
- Any of the properties that are set that aren't their corresponding property types (eg - "age" key is a string).

Read *Handling Hydrant Errors* for more on this topic.

Marking fields as Optional

Most of time, we still want our users to still use the application despite some invalid data. We can mark fields to tell Hydrant that some fatal errors are actually non-fatal.

This produces the effect of having optional fields that are parsed or a fallback value is used instead.

The way to do this is with *HYDMapOptionally*:

```
id<HYDMapper> mapper = HYDMapObject[NSDictionary class], [Person class],
                                    @{@"first_name": @"firstName",
                                      @"last_name": @"lastName",
                                      @"homepage": @[HYDMapOptionallyTo(HYDMapStringToURL()), @"homep
                                      @"age": @[HYDMapOptionally(), @"age"],
                                      @"children": @[HYDMapArrayOf(HYDMapObject([NSDictionary class],
                                                                                @{@"first_name": @"fi
                                                                                  @"last_name": @"las
                                                                                  @"age": HYDMapOptio
                                                                          @"children"];
```

Here we're making the **age** and **homepage** keys optional. Any invalid values will produce nil or the zero-value:

- If homepage isn't a valid NSURL, it is nil
- If age isn't a valid number, it is 0

The format of the dictionary mapper `HYDMapObject` expects is:

```
@{<KeyPathToRead>: @[<HYDMapper>, <KeyPathToWrite>],
  <KeyPathToRead>: <KeyPathToWrite>}
```

We can use this new mapper to selectively populate our array with values that are parsable. We can make our mapper ignore children objects that fail to parse:

```
id<HYDMapper> personMapper = HYDMapObject([NSDictionary class], [Person class],
                                          @{@"name": @"firstName"});
id<HYDMapper> mapper = HYDMapArrayOf(HYDMapOptionallyTo(personMapper));

json = @[@{},
         @{"name": @"John"},
         @{"last": @"first"}];

HYDError *error = nil;
NSArray *people = [mapper objectFromSourceObject:json error:&error];

people // => @[<Person: John>]
error // => non-fatal error
```

But swapping the two map functions will change the behavior to optionally dropping the array when any of the elements fail to parse:

```
id<HYDMapper> personMapper = HYDMapObject([NSDictionary class], [Person class],
                                          @{@"name": @"firstName"});
id<HYDMapper> mapper = HYDMapOptionallyTo(HYDMapArrayOf(personMapper));

json = @[@{},
         @{"name": @"John"},
         @{"last": @"first"}];

HYDError *error = nil;
NSArray *people = [mapper objectFromSourceObject:json error:&error];

people // => nil
error // => non-fatal error
```

The composition of these mappers provides the flexibility and power in Hydrant.

Converting it back to JSON

You can use the mapper to convert the person object back into JSON since we just declaratively described the JSON structure:

```
id<HYDMapper> reversedMapper = [mapper reverseMapper];
id json = [reverseMapper objectFromSourceObject:john error:&err];
```

That will give us our JSON back. Easy as that!

Removing Boilerplate

Soon, you'll be typing a lot of these maps to dictionaries. We can cut some of the cruft we have to type. `[NSDictionary class]` is implicit as the second argument to *HYDMapObject*:

```
id<HYDMapper> mapper = HYDMapObject([NSDictionary class], [Person class], ...);
// can is equivalent to
id<HYDMapper> mapper = HYDMapObject([Person class], ...);
```

Likewise with arrays, you can merge *HYDMapObject* and *HYDMapCollectionOf / HYDMapArrayOf* into *HYDMapCollectionOf / HYDMapArrayOf*:

```
HYDMapArrayOf(HYDMapObject([NSDictionary class], [Person class], ...))
// can become
HYDMapArrayOfObjects([Person class], ...)
```

So now we have this:

```
id<HYDMapper> mapper = HYDMapObject([Person class],
                                    @{@"first_name": @"firstName",
                                      @"last_name": @"lastName",
                                      @"homepage": @[HYDMapStringToURL(), @"homepage"],
                                      @"age": @"age",
                                      @"children": @[HYDMapArrayOfObjects([Person class],
                                                                          @{@"first_name": @"firstNar
                                                                            @"last_name": @"lastName"
                                                                            @"age": @"age"}),
                                                     @"children"]});
```

But we can do even better.

Using Reflection to Remove the Boilerplate

If your JSON is well formed and just requires a little processing to map directly to your objects, you can use *HYDMapReflectively*, which will use introspection of your classes to determine how to map your values. Although some information is still required for container types:

```
HYDCamelToSnakeCaseValueTransformer *transformer = \
    [[HYDCamelToSnakeCaseValueTransformer alloc] init];
id<HYDMapper> childMapper = HYDMapReflectively([Person class])
                            .keyTransformer(transformer)
                            .except(@[@"children"]);
id<HYDMapper> mapper = HYDMapReflectively([Person class])
                       .keyTransformer(transformer)
                       .customMapping(@{@"children": @[HYDMapArrayOf(childMapper), @"children"]});
```

The `mapper` variable above will map incoming source objects by converting snake cased keys to their camel cased variants to map properties together.

The reflective mapper tries a bunch of strategies to parse the incoming data into something reasonable. For example, it tries various different NSDate formats and permutations to convert an NSString into an NSDate.

The reflective mapper cannot predict how to convert it back to JSON since it tries a number of strategies for parsing the JSON. We can specify it like so:

```
// let's say we changed this class to have a birthDate property
@interface Person
// ...
@property (strong, nonatomic) NSDate *birthDate;
```

```
@end

id<HYDMapper> mapper = HYDMapReflectively([NSDictionary class], [Person class])
                        .keyTransformer(snakeToCamelCaseTransformer)
                        .mapClass([NSDate class], HYDMapDateToString(HYDDateFormatRFC3339));
```

This will explicitly tell Hydrant how to map types to and from your source object. Otherwise its behavior can be unexpected for certain classes. Read the documentation about *HYDMapReflectively* for more details.

That's it! You might like to read up on some of the many mappers you can use. But that's all there's to it!

Got some more complicated parsing you need to do? Check out the *Mapping Techniques* section for more details.

Handling Hydrant Errors

When handling Hydrant errors, use the `-[HYDError isFatal]` method to check if the received error is fatal. Fatal errors indicate that the resulting object should not be used:

```
HYDError *error = nil;
id resultingObject = [mapper objectFromSourceObject:json error:&error];

if ([error isFatal]) {
    // do error handling
} else {
    // success
}
```

If `[error isFatal]` is `NO`, but there is a non-nil error, then a non-fatal error has occurred. This is happens when a fallback parse option has taken place. A simple example of this is with *HYDMapOptionally*:

```
id<HYDMapper> mapper = HYDMapOptionallyTo(HYDMapStringToURL());

id invalidURL = @1;

HYDError *error = nil;
id resultingObject = [mapper objectFromSourceObject:invalidURL error:&error];
// error => non-fatal error
// resultingObject => nil
```

This example above produces a non-fatal error with a resulting object of nil. The non-fatal error reports the error that *HYDMapStringToURL* would.

Debugging Parse Errors

Without exceptions, it becomes harder to track down the sources of errors. For this reason, Hydrant errors store a signifcant amount of information for debugability.

Currently HYDErrors provides a human-friendly output when using `debugDescription` or `description`:

```
(lldb) po error
[FATAL] HYDErrorDomain (code=HYDErrorMultipleErrors) because "Multiple parsing errors occurred (fatal
- Could not map from 'name.first' to 'firstName' (HYDErrorInvalidResultingObjectType)
```

The default description will emit all the fatal errors that have occurred when parsing. Non-fatal errors are suppressed from output. If you want to see all the errors, use `-[HYDError fullDescription]`:

```
(lldb) po [error fullDescription]
[FATAL] HYDErrorDomain (code=HYDErrorMultipleErrors) because "Multiple parsing errors occurred (fatal
- Could not map from 'name.first' to 'firstName' (HYDErrorInvalidResultingObjectType)
- Could not map from 'name.last' to 'lastName' (HYDErrorInvalidResultingObjectType)
```

If you like trees, you can have a more classical-styled output using `-[HYDError recursiveDescription]`, which prints a tree of fatal errors.

And there's a corresponding `-[HYDError fullRecursiveDescription]` which emits a tree of all errors.

Reading Information from Hydrant Errors

If you want more debugging information access the `userInfo` dictionary:

- `HYDIsFatalKey` returns an NSNumber indicating when the error is fatal or not. Fatal errors indicate the resulting object return should not be used.
- `HYDUnderlyingErrorsKey` returns all the child errors that contributes to this error.
- `HYDSourceObjectKey` returns the source object that caused the error. For child errors, this can be part of the original source object.
- `HYDDestinationObjectKey` returns destination object that caused the error. Most of the time this is nil unless mappers do post-object validation, such as *HYDMapTypes*.
- `HYDSourceAccessorKey` returns the source accessor for accessing the source object value in question.
- `HYDDestinationAccessorKey` returns the destination accessor for the destination object. This is the intended destination of the resulting object. produced by this mapper.

`HYDError` provides a helper methods for reading keys from `userInfo` more easily:

```
- (BOOL)isFatal;
- (NSArray *)underlyingErrors;
- (id)sourceObject;
- (id)destinationObject;
- (id<HYDAccessor>)sourceAccessor;
- (id<HYDAccessor>)destinationAccessor;
```

Warning: It's worth noting that sourceObject and destinationObject could leak sensitive information. Be careful when you're sending HYDError's userInfo over the network or logging to disk.

Creating Hydrant Errors

If you're writing your own *Mapper* or *Accessor* there are also helper methods to construct conforming Hydrant errors:

```
+ (instancetype)errorWithCode:(NSInteger)code
                 sourceObject:(id)sourceObject
              sourceAccessor:(id<HYDAccessor>)sourceAccessor
           destinationObject:(id)destinationObject
        destinationAccessor:(id<HYDAccessor>)destinationAccessor
                    isFatal:(BOOL)isFatal
           underlyingErrors:(NSArray *)underlyingErrors;
```

This will properly construct the object with all possible information. While not all the arguments are required. Providing more information will help with tracing down parse errors. The only required parameters are `code` and `isFatal` – any other parameter can accept `nil`.

`underlyingErrors` is an array of NSErrors, which can include other Hydrant errors.

If your mapper contains other mappers, it can wrap errors with more information:

```
+ (instancetype)errorFromError:(HYDError *)error
      prependingSourceAccessor:(id<HYDAccessor>)sourceAccessor
        andDestinationAccessor:(id<HYDAccessor>)destinationAccessor
       replacementSourceObject:(id)sourceObject
                       isFatal:(BOOL)isFatal;
```

This method uses existing values from the source error with potential overrides or additions based on the context of the mapper's usage. Passing in `nil` will use the underlying error's values. Only `error` and `isFatal` are required.

If your mapper uses multiple child mappers, you can create a HYDError with multiple errors:

```
+ (instancetype)errorFromErrors:(NSArray *)errors
                   sourceObject:(id)sourceObject
                 sourceAccessor:(id<HYDAccessor>)sourceAccessor
              destinationObject:(id)destinationObject
             destinationAccessor:(id<HYDAccessor>)destinationAccessor
                        isFatal:(BOOL)isFatal;
```

This will store the underlying errors for debugging via `-[description]` and similar methods.

Mapping Techniques

This is a list of various methods of parsing potentially problematic or difficult input data. Prefer these techniques before having to resort to using *HYDMapWithBlock* or *HYDMapWithPostProcessing*

Convert a value or return nil (or another default value)

Use the *HYDMapOptionally* mapper:

```
id<HYDMapper> mapper1 = HYDMapStringToURL();
id<HYDMapper> mapper2 = HYDMapOptionallyTo(HYDMapStringToURL());
```

`mapper1` one will produce a fatal error if given a value like `@1` but `mapper2` will return a non-fatal error with `nil`.

Use it with *HYDMapObject*, to optionally map properties.

How do I return a source object's value unchanged?

Just use *HYDMapIdentity*.

Mapping an array of objects, excluding invalid ones instead of failing entirely

Depending on the location of *HYDMapOptionally* in comparison to *HYDMapCollectionOf / HYDMapArrayOf*, different behavior occurs:

```
HYDMapOptionallyTo(HYDMapArrayOf(...))
```

This mapper will return `nil` if **any part of mapping the array fails**. Where as:

```
HYDMapArrayOf(HYDMapOptionallyTo(...))
```

Will **exclude any element that fails to parse** from the array. This is due to the way `HYDMapArrayOf` excludes values from its mapper that produces `nil` and any kind of error (fatal or non-fatal).

Mapping Two Fields to One Property

This applies to any combination of mappings: many-to-one, one-to-many, or many-to-many in a *Mapping Data Structure*.

Explictly use *HYDAccessKeyPath* with each key. The accessor will produce an array of the values it has extracted:

```
HYDMapObject([Employee class],
             @{@[@"date", @"time"]: @[HYDMapByStringJoining(@"T"),
                                      HYDMapStringToDate(HYDDateFormatRFC3339),
                                      @"joinDate"]};
```

Writing Your Own Mappers

Hydrant comes with a lot of mappers, but there will always be scenarios that Hydrant cannot anticipate. Mappers are the ultimate method to extending the features while still reusing as much as possible with Hydrant.

Before reading this, be sure to read the *Mapper* protocol and *error handling*, specifically *creating errors*.

Thought Process

When writing a mapper, look to do the least amount of work possible. Allow the composition of other mappers to achieve the bigger goal you're trying to solve:

- Processing a collection? Can you use *HYDMapCollectionOf / HYDMapArrayOf* instead?
- Need to map objects? Can you compose with *HYDMapKVCObject*?

Even when implementing, feel free to use the existing mappers for implementation details, *HYDMapReflectively* and *HYDMapObject* both compose other mappers to achieve their work.

Type-checking (via `-[isKindOfClass:]`) is a bit more sensitive. You should do type checking to avoid crashing, but strict type checks to specific concrete classes should be considered carefully, because *HYDMapTypes* can cover those use cases.

Raising Exceptions

While mappers should not raise exceptions for `-[objectFromSourceObject:error]`, it is perfectly acceptable to raise exceptions on mapper construction or creating a reverse mapper for easier debuggability by the consumer of your mapper.

Crafting your Implementation

When creating a new mapper, you should treat the two arguments you receive very carefully:

- `sourceObject` **can be any value**, so be sure that your mapper does not crash or throw exceptions when receiving `nil`, `[NSNull null]`, and unexpected object types.
- `error` is an object pointer that may or may not be specified. Unlike some SDKs, **you should nil out the error pointer** if there are no errors.

If your mapper returns a fatal error, it is recommended to return `nil`. Using `nil` is not enough to indicate errors if parsing a source object to return nil is intended behavior. Don't return `[NSNull null]` directly.

Closing Thoughts

Is the mapper you wrote generic? *Contribute* it back to Hydrant! Make sure it conforms to *Hydrant's design* and is tested.

Design

This describes the internal design of Hydrant. Knowing the internal design gives you the understanding how to extend Hydrant or contribute code.

The core the Hydrant's flexibility are its protocols. There are two primary ones which most of the objects in Hydrant use to interact with each other. There are specific expectations and assumptions of these protocools if you choose to write code that conforms to these protocols.

Whenever possible, compose mappers than reimplementing features from existing mappers. This also applies when writing your own mappers. For example, *HYDMapKVCObject* doesn't do type checking, since *HYDMapTypes* does that already. A facade object, *HYDMapObject* composes both of these mappers to provide an object mapper that has type checking.

Philosophies

Hydrant has some opinions that are reflected in its code base and design – some more strongly than others. Individually, they are useful, but as more of these are combined, they become greater than the sum of their parts.

Composition over Inheritance

Hydrant is a composition library. Inheritance is strongly discouraged when building mappers or accessors. They tightly couple child classes and parent classes, break encapsulation, and increase overhead when learning a code base. A subclass implementation requires knowledge of the parent classes' implementation too.

Hydrant uses subclasses as dictated by Apple's frameworks (eg - NSFormatter or NSValueTransformer). Any other subclasses in Hydrant are **quickfix temporary solutions** and are never to be considered public APIs.

Immutability over Mutability

Hydrant makes an unusual stance to hide all internal properties. Whenever possible, Hydrant prefers immutability over mutation. This makes classes significantly easier to consume and debug.

Having mutable properties also makes the classes less viable for being shared across threads. Mutation can break assumptions about objects that conform to an *abstraction*.

Abstractions over Concretions

Ideally, concrete classes should never have to know about each other by working through a protocol. These protocols can be given on object construction to provide flexibility. Protocols are also easy to *test*. They provide a stronger assumption of having less intimate knowledge of the collaborating object.

The only exception to this rule are Value Objects which should not perform complex behavior, but be a vessel for storing data. *Handling Hydrant Errors* is an example of a value object.

Good abstractions can be utilized through the library and should thought through carefully. Which leads to...

Have Small Abstractions

The best abstractions are as narrow as possible, to allow the most flexibility of an implementation. Conveniences should be built on top of them but not be included into the abstraction.

A large abstraction is usually indicative of multiple abstractions that need to be split apart. For example, *Mapper* and *Accessor* were one protocol, which exposed itself because of the duplicated work required for implementation of getting and setting data across various mappers. Splitting this abstraction avoided other solutions: such as using inheritance or copy-pasting similar implementations.

Abstractions are fractal, so it may not be immediately obvious that smaller ones exist, but they do and provide a more flexible system in less code.

Test-Driven Code

While you may not agree with TDD/BDD, Hydrant should have thorough test coverage for various scenarios. After all, nefarious input is being processed by this library.

All public classes should have tests covering their proper behavior. Any bugs fixed with associated tests that verify the bug.

Mapper

Let's look at the mapper protocol which is the foundation to Hydrant's design:

```
@protocol HYDMapper <NSObject>

- (id)objectFromSourceObject:(id)sourceObject error:(__autoreleasing HYDError **)error;
- (id<HYDMapper>)reverseMapper;

@end
```

Using this protocol plus object composition, provides a shared method for mappers to compose with each other.

Let's break it down by method – along with their purposes and expectations:

```
- (id)objectFromSourceObject:(id)sourceObject error:(__autoreleasing HYDError **)error;
```

This method is where all the grunt work occurs. Here a new object is created from the source object. This also provides a method for returning errors that should conform to Hydrant's error handling policies. This includes:

- Emitting fatal errors when mapping fails.
- Emitting non-fatal errors when an alternative mapping occurred.
- Including as much userInfo about the error (see constants).
- Returning nil if a fatal error occurs.

It is the responsibility of each mapper to **avoid throwing exceptions**. This matches Apple's convention of exceptions in Objective-C, where they should be used to indicate programmer error.

For easy of discovery, many mappers will validate its construction instead of possibly raising exceptions on `-[objectFromSourceObject:error:]`.

For Hydrant Mappers, any operation on the sourceObject should be treated defensively. Doing work on a sourceObject **should never** raise an exception. Even under ARC, memory leaks can occur when exceptions are caught since the underlying libraries may not support the `-fobjc-arc-exceptions` flag.

That being said, exceptions can be raised if the definition of the resulting object is improperly configured. For example, `HYDObjectMapper` will throw an exception if the destination object is missing a key that is specified by the Hydrant user. But whenever possible, produce these exceptions as early as possible (eg - on object construction time instead of when `-[objectFromSourceObject:error:]` is called).

The next method on `HYDMapper` are for compositions of mappers:

```
- (id<HYDAccessor>)destinationAccessor;
```

This method returns an accessor instance for parent mappers (mappers that hold this mapper). Accessors, which are described more in the later section, are an abstraction to how to read and write values from an object. In this case, the destinationAccessor is how the parent mapper should map the value. This method exists for syntactic reasons of the DSL.

Accessor

Some mappers use a smaller abstraction called accessors. Accessors describe how to set and get values. Surprisingly, they are larger than the *Mapper* protocol:

```
@protocol HYDAccessor <NSObject>

- (NSArray *)valuesFromSourceObject:(id)sourceObject error:(__autoreleasing HYDError **)error;
- (HYDError *)setValues:(NSArray *)values onObject:(id)destinationObject;
- (NSArray *)fieldNames;

@end
```

There are currently two implementations of accessors: *HYDAccessKey* and *HYDAccessKeyPath* which use KVC to set and get values off of objects.

The accessor protocol supports getting and setting multiple values at once. In fact, both built-in Hydrant accessors support parsing multiple values. Allowing mappers to process multiple values at once gives an opportunity to do value joining (eg - joining a "date" and "time" field into a "datetime" field).

The method `-[fieldNames]` exists only for debuggability – providing the developer enough contextual information to location the exact mapper that failed in a large composition of mappers. The values in this method is used by mappers to populate Hydrant errors.

Accessors & Mappers

Accessors can choose to emit errors like mappers, but the default implementations existed prior to this feature and opt to return `[NSNull null]`. Hydrant mappers that treat `nil` and `[NSNull null]` the same. They also extract values out of their resulting arrays if there is only one value for easier composibility with other mappers.

Mappers will bubble up accessor errors to their consumers. The same rules about fatalness apply here too – fatal errors abort the entire parsing operation while non-fatal errors indicate errors that could be recovered from.

Mapping Data Structure

Various mappers built on top of *HYDMapKVCObject* utilize an informal data structure based format for describing field-to-field mapping which follows the form of:

```
@{<HYDAccessor>: <HYDMapping>}
```

Where's `HYDMapping`? It's just a tuple, which is fancy for saying an array:

```
@[<HYDMapper>, <HYDAccessor>]
```

So in summary, mapping dictionaries are just:

```
@{<HYDAccessor1>: @[<HYDMapper>, <HYDAccessor2>]}
```

Which reads, map `<HYDAccessor1>` to `<HYDAccessor2>` using `<HYDMapper>`.

To get this mapping into this form, it is first normalized by:

- Converting all keys that are strings into *HYDAccessKeyPaths*.
- Converting all keys that are arrays into *HYDAccessKeyPaths* with an array.
- Converting all values that are strings into a mapping of *HYDMapIdentity* and *HYDKeyPathAccessors*.
- Converting all values that are arrays into a mapping of *HYDMapIdentity* and *HYDKeyPathAccessors*.

And that's it! Anything else specific must be done explicitly using the array-styled syntax. If you so choose, you can use your own tuple-like object for the `HYDMapping` protocol.

How do you have function overloading without being Objective-C++?

Hydrant makes use of a little known Clang-specific feature:

```
__attribute__((overloadable))
```

This overloadable attribute allows basic C++ overloaded functions with some notable exceptions:

```
- It cannot overload with a zero-arity function.
- Protocols are not part of the type dispatch -- so you cannot have two
  overloaded functions with different protocols
```

For convience, Hydrant uses the macro `HYD_EXTERN_OVERLOADED` to define these functions:

```
HYD_EXTERN_OVERLOADED
id<HYDMapper> MyMapper(NSString *foo);
```

Since the custom attribute changes the compiled function name, **adding the overloadable attribute to an existing will break existing consumers**. For iOS, this is not usually a problem since recompilation is required for static libraries. But for dynamic OSX libraries, this can be problematic.

Trade-offs

Every design and implementation has trade-offs. Anyone who tells you otherwise is not giving the entire picture. Hydrant is no exception:

- It is slower than naive parsing, because it's doing more validation checks
- It is design for parsing data that you do not control, if you control the JSON API, it might not be necessary to use Hydrant
- It provides no other features other serialization/deserialization, such as value objects, persistence, networking, etc.

Mapper Reference

Here lists all the mappers currently available in Hydrant. Composing these mappers together provides the ability to (de)serialize for a wide variety of use cases. All the functions listed here return objects that conform to the *Mapper* protocol.

Thread Safety

While none of Hydrant's mappers are ensured for thread safety. They are **immutable after creation** unless otherwise noted. This doesn't gaurantee thread safety since other internal objects can be thread unsafe (eg - Formatters, ValueTransformers, etc.).

Constructor Helper Functions

Nearly all mappers come with helper functions. These are simply overloaded c functions that provide a way to construct mappers more succinctly. Since they are overloaded they generally conform to the following style:

```
HYDMapMapperName(...)
HYDMapMapperName(id<HYDMapper> innerMapper, ...);
```

The former function is a convenience that converts to the latter function:

```
HYDMapMapperName(HYDMapIdentity(), ...);
```

The *identity mapper* simply provides direct access to the source object and provides a KVC-styled key accessor for parent mappers.

Inner mappers are receive the source object before the current mapper. This allows chaining of complex conversion methods. For example:

```
id<HYDMapper> mapper = HYDMapURLToStringFrom(HYDMapStringToURL())
```

This mapper composition produces strings that are valid URLs. Strings that are not URLs fail to parse for this mapper. For readability, you can also compose mappers in a chain using *HYDMapThread*.

For autocompleting convenience, all the helper functions are prefixed with `HYDMap`. So *HYDMapEnum* exposes constructor functions with *HYDMapEnum*.

You might be thinking these overload functions require Objective-C++, but *you'd be wrong*.

The Reverse Mapper

All mappers defined here fully support Hydrant's *Mapper* protocol unless explicitly state otherwise. This means each mapper can create an equivalent mapper that undoes the current mapper:

```
id<HYDMapper> mapper = HYDMapStringToURL();
id<HYDMapper> reversedMapper = [mapper reverseMapper];

NSString *URI = @"http://jeffhui.net";
// never pass nil to error, but here for brevity
NSURL *url = [mapper objectFromSourceObject:URI error:nil];
NSString *reversedURI = [reversedMapper objectFromSourceObject:url error:nil];

assert [URI isEqual:reversedURI]; // equal
```

Mappers that are **lossy** cannot ensure the reversability will be exactly equal, this currently only applies to *HYDMapForward* and *HYDMapBackward*.

HYDMapEnum

The enum mapper uses a dictionary to map values from the source object to the destination object. This is typically used for mapping strings to an enum equivalent.

> **Warning:** The mapping dictionary for this mapper is assumed to have a one-to-one mapping for its keys and values. Any key that maps to the same value or vice versa will cause undefined behavior. Future versions of Hydrant may choose to make this error throw an exception.

Any values that do not match the enum will make this mapper produce a fatal error. To provide an optional default, wrap with `HYDOptionalMapper`.

The following helper functions are available for this mapper:

```
HYDMapEnum(NSDictionary *mapping);
HYDMapEnum(id<HYDMapper> innerMapper, NSDictionary *mapping);
```

With the `mapping` dictionary mapping source object values to destination object values. Remember that all values in the mapping need to be an object:

```
// defined somewhere...
typedef NS_ENUM(NSUInteger, PersonGender) {
    PersonGenderUnknown,
    PersonGenderMale,
    PersonGenderFemale,
};

// building the mapper
HYDMapEnum(HYDRootMapper,
           @{@"male": @(PersonGenderMale),
             @"female": @(PersonGenderFemale),
             @"unknown": @(PersonGenderUnknown)});
```

The internal implementation class is `HYDEnumMapper`.

HYDMapIdentity

This mapper, as its name suggests, is a passthrough mapper. It simply returns the source object as its destination object.

Sounds pretty useless, but it is used by other mappers as the "default" inner mapper that can be used for chaining. Because of this, this mapper is used by helper functions for nearly all the other mappers in Hydrant.

HYDMapObjectToStringByFormatter

This mapper utilizes NSFormatter to convert objects to strings. It uses the `-[NSFormatter stringForObjectValue:]` internally for this mapping while conforming as a Hydrant mapper.

Formatters that return `nil` will make this mapper produce a fatal Hydrant error.

For the reverse – mapping a string to an object with an NSFormatter, use *HYDMapStringToObjectByFormatter*. Calling `-[reverseMapper]` will do this with the same parameters provided to this mapper.

The helper functions are available for this mapper:

```
HYDMapObjectToStringByFormatter(NSFormatter *formatter);
HYDMapObjectToStringByFormatter(id<HYDMapper> innerMapper, NSFormatter *formatter);
```

This mapper is the underpinning for other mappers that utilize this internally:

- *HYDMapDateToString* - Converts a NSDate to NSString
- *HYDMapURLToString* - Converts an NSURL to NSString
- *HYDMapNumberToString* - Converts a number to NSString
- *HYDMapUUIDToString* - Converts an NSUUID to NSString

HYDMapStringToObjectByFormatter

This mapper utilizes NSFormatter to convert strings to objects. It uses `-[NSFormatter getObjectValue:forString:errorDescription:]` internally for this mapping while conforming as a Hydrant mapper.

In addition, this mapper will validate that the source object is a valid string before passing it through to the formatter. When an error description is returned, Hydrant will insert it into an NSError instance like:

```
[NSError errorWithDomain:NSCocoaErrorDomain
                    code:NSFormattingError
                userInfo:@{NSLocalizedDescriptionKey: errorDescription}];
```

If errorDescription is not provided but success is still `NO`, then a generic errorDescription is created as a placeholder.

Following the creating of the NSError, it is wrapped inside a Hydrant error for compatibility with the reset of Hydrant as a fatal error.

For the reverse – mapping an object to a string with an NSFormatter, use *HYDMapObjectToStringByFormatter*.

The helper functions are available for this mapper:

```
HYDMapStringToObjectByFormatter(NSFormatter *formatter);
HYDMapStringToObjectByFormatter(id<HYDMapper> mapper, NSFormatter *formatter);
```

This mapper is the underpinning for other mappers that utilize this internally:

- *HYDMapStringToDate* - Converts a NSString to NSDate
- *HYDMapStringToURL* - Convert a NSString to NSURL
- *HYDMapStringToNumber* - Converts a NSString to NSNumber
- *HYDMapStringToUUID* - Converts a NSString to NSUUID

HYDMapDateToNumberSince

This converts *NSDates* into *NSNumbers* by using the built-in conversions. The mapper will verifying the source object is a valid date before doing the conversion.

The following helpers are available:

```
HYDMapDateToNumberSince1970();
HYDMapDateToNumberSince1970(HYDDateTimeUnit unit);
HYDMapDateToNumberSince(NSDate *sinceDate);
HYDMapDateToNumberSince(NSDate *sinceDate, HYDDateTimeUnit unit);
```

Some of these functions allow you specify the units the number is to be emitted in. A double in seconds is returned by default, but you can change it to return an alternative unit:

```
HYDDateTimeUnitMilliseconds
HYDDateTimeUnitSeconds
HYDDateTimeUnitMinutes
HYDDateTimeUnitHours
```

See *HYDMapNumberToDateSince* for the reverse of this mapper.

HYDMapNumberToDateSince

This converts *NSNumbers* into *NSDates* by using the built-in conversions. The mapper will verifying the source object is a valid number before doing the conversion. A reference date can be specified to interpret the source number being relative to.

The following helpers are available:

```
HYDMapNumberToDateSince1970();
HYDMapNumberToDateSince1970(HYDNumberDateUnit unit);
HYDMapNumberToDateSince(NSDate *sinceDate);
HYDMapNumberToDateSince(NSDate *sinceDate, HYDDateTimeUnit unit);
```

Some of these functions allow you specify the units the number is to be emitted in. A double in seconds is returned by default, but you can change it to return an alternative unit:

```
HYDDateTimeUnitMilliseconds
HYDDateTimeUnitSeconds
HYDDateTimeUnitMinutes
HYDDateTimeUnitHours
```

See *HYDMapDateToNumberSince* for the reverse of this mapper.

HYDMapDateToString

This wraps around *HYDMapObjectToStringByFormatter* and provides conveniences for using an NSDateFormatter to map a date to a string.

The following helper functions are available:

```
HYDMapDateToString(NSString *formatString);
HYDMapDateToString(NSDateFormatter *dateFormatter)
HYDMapDateToString(id<HYDMapper> innerMapper, NSString *formatString);
HYDMapDateToString(id<HYDMapper> innerMapper, NSDateFormatter *dateFormatter)
```

Either you can provide date format string (or use one of Hydrant's *Date Format Strings*) or use a customized `NSDateFormatter` instance.

The reverse of this mapper is *HYDMapStringToDate*.

See *HYDMapDateToNumberSince* if you're looking to convert dates into numbers

HYDMapStringToDate

This wraps around *HYDMapStringToObjectByFormatter* and provides conveniences for using an NSDateFormatter to map a string to a date.

The following helper functions are available:

```
HYDMapStringToDate(NSString *formatString);
HYDMapStringToDate(NSDateFormatter *dateFormatter)
HYDMapStringToDate(id<HYDMapper> innerMapper, NSString *formatString);
HYDMapStringToDate(id<HYDMapper> innerMapper, NSDateFormatter *dateFormatter)
HYDMapStringToAnyDate();
HYDMapStringToAnyDate(id<HYDMapper> innerMapper);
```

Either you can provide date format string (or use one of Hydrant's *Date Format Strings*) or use a customized `NSDateFormatter` instance.

`HYDMapStringToAnyDate` attempts to parse the given string as any of the dates specified in *Date Format Strings*. Unsurprisingly, the mapper that the function produces will have unreliable results when reversing.

The reverse of this mapper is *HYDMapDateToString*.

See *HYDMapNumberToDateSince* if you're looking to convert numbers into dates.

HYDMapStringToNumber

This provides conveniences to *HYDMapStringToObjectByFormatter* by using NSNumberFormatter to convert a string to an NSNumber.

The following helper functions are available:

```
HYDMapStringToDecimalNumber()
HYDMapStringToNumber(id<HYDMapper> mapper)
HYDMapStringToNumber(NSNumberFormatterStyle numberFormatStyle)
HYDMapStringToNumber(id<HYDMapper> mapper, NSNumberFormatterStyle numberFormatStyle)
HYDMapStringToNumber(NSNumberFormatter *numberFormatter)
HYDMapStringToNumber(id<HYDMapper> mapper, NSNumberFormatter *numberFormatter)
```

The reverse of this mapper is *HYDMapNumberToString*.

Converting an NSNumber to a c-native numeric type is not the responsibility of this mapper, that is what *HYDMapKVCObject* does.

HYDMapNumberToString

This provides conveniences to *HYDMapStringToObjectByFormatter* by using NSNumberFormatter to convert an NSNumber to a string.

The following helper functions are available:

```
HYDMapDecimalNumberToString()
HYDMapNumberToString(id<HYDMapper> mapper)
HYDMapNumberToString(NSNumberFormatterStyle numberFormatStyle)
HYDMapNumberToString(id<HYDMapper> mapper, NSNumberFormatterStyle numberFormatStyle)
HYDMapNumberToString(NSNumberFormatter *numberFormatter)
HYDMapNumberToString(id<HYDMapper> mapper, NSNumberFormatter *numberFormatter)
```

The reverse of this mapper is *HYDMapStringToNumber*.

Converting a c-native numeric type to an NSNumber is not the responsibility of this mapper, that is what *HYDMapKVCObject* does.

HYDMapURLToString

This provides conveniences to *HYDMapObjectToStringByFormatter* by using *HYDURLFormatter* to convert an *NSURL* to a string.

The following helper functions are available:

```
HYDMapURLToString();
HYDMapURLToStringFrom(id<HYDMapper> innerMapper);
HYDMapURLToStringOfScheme(NSArray *allowedSchemes)
HYDMapURLToStringOfScheme(id<HYDMapper> mapper, NSArray *allowedSchemes)
```

An array of schemes can be provided that the URL must conform to be valid. For example, this mapper only accepts http urls:

```
HYDMapURLToStringOfScheme(@["http", @"https"])
```

The reverse of this mapper is *HYDMapStringToDate*.

HYDMapStringToURL

This provides conveniences to *HYDMapStringToObjectByFormatter* by using *HYDURLFormatter* to convert a string to an NSURL.

The following helper functions are available:

```
HYDMapStringToURL();
HYDMapStringToURLFrom(id<HYDMapper> innerMapper);
HYDMapStringToURLOfScheme(NSArray *allowedSchemes)
HYDMapStringToURLOfScheme(id<HYDMapper> mapper, NSArray *allowedSchemes)
```

An array of schemes can be provided that the URL must conform to be valid. For example, this mapper only accepts http urls:

```
HYDMapStringToURLOfScheme(@["http", @"https"])
```

The reverse of this mapper is *HYDMapDateToString*.

HYDMapUUIDToString

This provides conveniences to *HYDMapObjectToStringByFormatter* by using *HYDUUIDFormatter* to convert an NSUUID to a string.

The following helper functions are available:

```
HYDMapUUIDToString();
HYDMapUUIDToStringFrom(id<HYDMapper> innerMapper);
```

The reverse of this mapper is *HYDMapStringToUUID*.

HYDMapStringToUUID

This provides conveniences to *HYDMapStringToObjectByFormatter* by using *HYDUUIDFormatter* to convert a string to an NSUUID.

The following helper functions are available:

```
HYDMapStringToUUID();
HYDMapStringToUUIDFrom(id<HYDMapper> innerMapper);
```

The reverse of this mapper is *HYDMapUUIDToString*.

HYDMapValue

This mapper utilizes NSValueTransformer to convert from one value to another. It utilizes `-[NSValueTransformer transformValue:]` internally for this mapping while conforming to the Hydrant mapper protocol.

HYDValueTransformerMapper assumes that all validation will be handled by the value transformer. No addition validation is done. **It is impossible for this mapper to return Hydrant errors**.

If the value transformer is reversable, then this mapper can be reversed. It produces *HYDMapReverseValue* which you can also use directly if you want to apply the reversed transformation to a source object.

Attempting to produce a reverse mapper when the transformer cannot be reversed will throw an exception.

The helper functions are available for this mapper:

```
HYDMapValue(NSValueTransformer *valueTransformer);
HYDMapValue(id<HYDMapper> innerMapper, NSValueTransformer *valueTransformer);
HYDMapValue(NSString *valueTransformerName);
HYDMapValue(id<HYDMapper> innerMapper, NSString *valueTransformerName);
```

If your value transformer is registered as a singleton via `+[NSValueTransformer setValueTransformer:forName:]`, then using the constructor functions that accept a string as the second argument can be used to easily fetch the value transformer by that name.

HYDMapReverseValue

This mapper utilizes NSValueTransformer to convert from one value to another. It utilizes `-[NSValueTransformer reverseTransformedValue:]` internally to produce the resulting object.

This mapper assumes that all validation will be handled by the value transformer. No additional validation is done. **It is impossible for this mapper to return Hydrant errors**.

If constructing this mapper with a value transformer that cannot be reversed will throw an exception. For the reverse of this mapper, see *HYDMapValue* if you want to map values using `-[NSValueTransformer transformValue:]`.

The helper functions are available for this mapper:

```
HYDMapReverseValue(NSValueTransformer *valueTransformer);
HYDMapReverseValue(id<HYDMapper> innerMapper, NSValueTransformer *valueTransformer);
HYDMapReverseValue(NSString *valueTransformerName);
HYDMapReverseValue(id<HYDMapper> innerMapper, NSString *valueTransformerName);
```

If your value transformer is registered as a singleton via `+[NSValueTransformer setValueTransformer:forName:]`, then using the constructor functions that accept a string as the second argument can be used to easily fetch the value transformer by that name.

HYDMapForward

This mapper traverses the source object before sending the traversed sub-source object to the child mapper its given. This allows for selectively ignoring various parts of a data structure from the incoming source object:

```
id<HYDMapper> mapper = HYDMapForward(@"person.account",
                                     HYDMapObject(HYDRootMapper, [Person class],
                                                  @{@"first": @"firstName"}));

id json = @{@"person": @{@"account": @{@"first": @"John"}}};

HYDError *error = nil;
Person *person = [mapper objectFromSourceObject:json error:&error];
// person.firstName => @"John"
```

Since this is lossy, reversing this mapper cannot produce any extra data that was truncated by the traversal. The reversed mapper of this produces a *HYDMapBackward*.

The helper functions available for this mapper:

```
HYDMapForward(NSString *walkKey, Class sourceClass, id<HYDMapper> childMapper);
HYDMapForward(id<HYDAccessor> walkAccessor, Class sourceClass, id<HYDMapper> childMapper);
HYDMapForward(NSString *walkKey, id<HYDMapper> childMapper);
HYDMapForward(id<HYDAccessor> walkAccessor, id<HYDMapper> childMapper);
```

The first argument for all these constructors are how to walk through through the incoming mapping. The last argument is the child mapper to process the subset of the source object being traversed by the first argument.

When not provided, `sourceClass` defaults to `[NSDictionary class]`, this is to hint to the reversed mapper how to produce the parent object.

HYDMapBackward

This mapper is the reverse of *HYDMapForward* it generates a series of repeated objects to that would allow the *HYDMapForward* to function on the resulting object produced:

```
id<HYDMapper> mapper = HYDMapBackward(@"person.account",
                                      HYDMapObject(HYDRootMapper, [Person class], [NSDictionary class
                                                   @{@"firstName": @"first"}));

Person *person = [[Person alloc] initWithFirstName:@"John"];

HYDError *error = nil;
id json = [mapper objectFromSourceObject:person error:&error];
// json => @{@"person": @{@"account": @{@"first": @"John"}}};
```

Since this mapper simply recursively creates the class it was given to produce the hierarchy.

The helper functions available for this mapper:

```
HYDMapBackward(NSString *walkKey, Class destinationClass, id<HYDMapper> childMapper);
HYDMapBackward(id<HYDAccessor> walkAccessor, Class destinationClass, id<HYDMapper> childMapper);
HYDMapBackward(NSString *walkKey, id<HYDMapper> childMapper);
HYDMapBackward(id<HYDAccessor> walkAccessor, id<HYDMapper> childMapper);
```

The first argument for all these constructors are the path of the keys to create recursively. The last argument is the child mapper to produce the final object that will be placed in the leaf of the path presented by the first argument.

When not provided, `destinationClass` defaults to `[NSDictionary class]`, this is to hint to the reversed mapper how to produce the parent objects. The destinationClass is instanciated with `[[NSObject alloc] init]`. If the class supports `NSMutableCopying`, then a mutableCopy is created to work with immutable data types (eg - NSDictionary which needs to be converted to NSMutableDictionary).

HYDMapCollectionOf / HYDMapArrayOf

This mapper applies a child mapper to process a collection, usually an array of items. Although this can apply to sets any other collection of items to map. The child mapper is used to map each individual element of the collection:

```
id<HYDMapper> childMapper = HYDMapObject([Person class],
                                         @{@"first": @"firstName"});
id<HYDMapper> mapper = HYDMapCollectionOf(childMapper,
                                          [NSArray class], [NSArray class]);

HYDError *error = nil;
id json = @[
    @{@"first": @"John"},
    @{@"first": @"Jane"},
    @{@"first": @"Joe"},
];
NSArray *people = [mapper objectFromSourceObject:json error:error];
// people => @[<Person: John>, <Person: Jane>, <Person: Joe>]
```

HYDCollectionMapper will validate the incoming source object's enumerability by checking if it is the given source class.

The helper functions available for this mapper:

```
HYDMapCollectionOf(id<HYDMapper> itemMapper, Class sourceCollectionClass, Class destinationCollectio
HYDMapCollectionOf(Class collectionClass)
HYDMapCollectionOf(Class sourceCollectionClass, Class destinationCollectionClass)
HYDMapCollectionOf(id<HYDMapper> itemMapper, Class collectionClass)
HYDMapArrayOf(id<HYDMapper> itemMapper)
HYDMapArrayOfObjects(Class sourceItemClass, Class destinationItemClass, NSDictionary *mapping)
HYDMapArrayOfObjects(Class destinationItemClass, NSDictionary *mapping)
```

`HYDMapArrayOf` are a set of convenience functions that assume the source and destination collection to be NSArrays. Further convenience are built on top that to convert an array of objects into another array of objects.

`HYDMapArrayOfObjects` is simply the composition:

```
HYDMapArrayOf(HYDMapObject(...))
```

See *HYDMapObject* for more information on that mapper.

This mapper has some extra behavior based on the result of the child mapper. Specifically, if a child mapper produces a `nil` value and a non-fatal error, then its value is excluded from an array. This allows selective exclusion of items from the source array in the resulting array.

For more details, see *Mapping an array of objects, excluding invalid ones instead of failing entirely*.

HYDMapFirst

This mapper tries to apply each mapper its given until one succeeds (does not return a fatal error). Using this mapper can provide an ordered list of mappers to attempt. An example is an array that has different object types:

```
id<HYDMapper> personMapper = HYDMapObject([Person class], {...});
id<HYDMapper> employeeMapper = HYDMapObject([Person class], {...});
id<HYDMapper> mapper = HYDMapArrayOf(HYDMapFirst(personMapper, employeeMapper));
```

`mapper` will try using `personMapper`, but if that mapper generates a fatal error, then `employeeMapper` is used instead. If that fails, then it is returned to the consumer of `mapper`.

`HYDMapFirst` is a macro around the constructor function:

```
HYDMapFirstMapperInArray(NSArray *mappers)
```

HYDMapSplit

This mapper allows you to replace the reverseMapper of the given mapper. This can be useful if a mapper does not provide the reverse mapper implementation you prefer, but want its source-to-destination mapping capabilities:

```
[HYDMapToString() reverseMapper]; // => raises exception

id<HYDMapper> mapper = HYDMapSplit(HYDMapToString(), HYDMapIdentity());

HYDError *err = nil;
[mapper objectFromSourceObject:@1 error:&err]; // => @"1"

[mapper reverseMapper] // => returns HYDMapIdentity()
```

There is only one helper function:

```
HYDMapSplit(id<HYDMapper> originalMapper, id<HYDMapper> reverseMapper);
```

Internally, Hydrant uses this for *HYDMapReflectively* to allow basic type coercion between strings and numbers.

HYDMapNonFatally

The non-fatal mapper takes child mapper to process and converts any fatal error that the child mapper produces into non-fatal ones:

```
// This mapper will attempt to convert a string to an NSURL
// or returns nil otherwise
id<HYDMapper> mapper = HYDMapNonFatally(HYDMapStringToURL(...))
```

There are many helper functions which relate to producing default values:

```
HYDMapNonFatally(id<HYDMapper> childMapper)
HYDMapNonFatallyWithDefault(id<HYDMapper> childMapper, id defaultValue)
HYDMapNonFatallyWithDefault(id<HYDMapper> childMapper, id defaultValue, id reverseDefault)
HYDMapNonFatallyWithDefaultFactory(id<HYDMapper> childMapper, HYDValueBlock defaultValueFactory)
HYDMapNonFatallyWithDefaultFactory(id<HYDMapper> childMapper, HYDValueBlock reversedDefaultFactory)
```

Which provides a variety of producing default values when fatal errors are received. By default, `nil` is returned.

Also, you might want to use *HYDMapOptionally*, which composition this with *HYDMapNotNull*.

HYDMapNotNull

The mapper produces fatal errors if a `nil` or `[NSNull null]` is returned by a given mapper:

```
id<HYDMapper> mapper = HYDMapNotNull();
id json = [NSNull null];
HYDError *error = nil;
// => produces fatal error
[mapper objectFromSourceObject:json error:&error];
```

There are helper functions:

```
HYDMapNotNull()
HYDMapNotNullFrom(id<HYDMapper> innerMapper)
```

Also, you might want to use *HYDMapOptionally*, which composition this with *HYDMapNonFatally*.

HYDMapOptionally

This is the composition of *HYDMapNonFatally* and *HYDMapNotNull* which produces a mapper that converts `nil`, `[NSNull null]` or any unmappable values into a default value provided.

The helper functions are based on the composition:

```
HYDMapOptionally()
HYDMapOptionallyTo(id<HYDMapper> innerMapper)
HYDMapOptionallyWithDefault(id defaultValue)
HYDMapOptionallyWithDefault(id<HYDMapper> innerMapper, id defaultValue)
HYDMapOptionallyWithDefault(id<HYDMapper> innerMapper, id defaultValue, id reverseDefaultValue)
HYDMapOptionallyWithDefaultFactory(HYDValueBlock defaultValueFactory)
HYDMapOptionallyWithDefaultFactory(id<HYDMapper> innerMapper, HYDValueBlock defaultValueFactory)
HYDMapOptionallyWithDefaultFactory(id<HYDMapper> innerMapper,
                                   HYDValueBlock defaultValueFactory,
                                   HYDValueBlock reverseDefaultValueFactory)
```

This is commonly used for conditionally allowing fields when mapping with *HYDMapObject*:

```
// first name is optional, last name is required
HYDMapObject([Person class],
             @{@"first": @[HYDMapOptionally(), @"firstName"],
               @"last": @"lastName"});

// this json causes a fatal error:
id json = @{@"first": @"John"};

// this json will produce a non-fatal error, and map to a Person object
id json = @{@"last": @"Doe"};

// this json will produce no error and map to a Person object
id json = @{@"first": @"John",
            @"last": @"Doe"};
```

HYDMapTypes

This mapper does type checking to ensure the given type is as intended. Using this mapper can provide type checking to filter out nefarious input that can potentially crash your application. If you're looking to apply this upon an object's properties, use *HYDMapObject* instead – which uses this mapper internally. *HYDMapCollectionOf / HYDMapArrayOf* also does some type checking for the collection source class.

The mapper simply uses `-[isKindOfClass:]` to verify expected inputs and outputs - returning a fatal error if this check fails.

Here are the following functions to construct this mapper:

```
HYDMapType(Class sourceAndDestinationClass)
HYDMapType(Class sourceClass, Class destinationClass)
HYDMapTypes(NSArray *sourceClasses, NSArray *destinationClasses)
HYDMapType(id<HYDMapper> innerMapper, Class sourceAndDestinationClass)
```

```
HYDMapType(id<HYDMapper> innerMapper, Class sourceClass, Class destinationClass)
HYDMapTypes(id<HYDMapper> innerMapper, NSArray *sourceClasses, NSArray *destinationClasses)
```

As the arguments suggest, you can provide multiple classes that are valid for inputs or outputs. Passing `nil` as a class argument will allow **any classes**. Source classes indicate values provided to the mapper, and destination classes represent output (usually from the innerMapper).

For functions that accept an array, passing an empty array will also behave like passing `nil`.

Note: This mapper can behave in unintuitive ways for inherited class clusters. So specifying `NSMutableDictionary` and `NSMutableArray` will cause fatal type-checking errors. Use `NSDictionary` and `NSArray` instead.

HYDMapKVCObject

This uses Key-Value Coding to map arbitrary objects to one another, or the more commonly known methods: `-[setValue:forKey:]` and `-[valueForKey:]`. This mapper provides a data-structure mapping DSL that conforms to a specific design that is mentioned in the *Mapping Data Structure*. But at an overview, they usually look like one of two forms:

```
@{@"get.KeyPath": @"set.KeyPath"}
@{@"get.KeyPath": @[myMapper, @"set.KeyPath"]}
```

They both conform to KeyPath-like semantics, similar to the `-[valueForKeyPath:]` method, but without the aggregation features. They all read similarly to:

> Map 'get.KeyPath' to 'set.KeyPath' using myMapper

This is simply used as an abbreviated form to specify the mapping for each property without the visual noise of objective-c styled object construction. Again, read up on the *Mapping Data Structure* to see the internal representation this mapper uses after processing this data structure.

Note: Since this mapper uses `setValue:forKey:` and `valueForKey:`, all the same consequences apply – such as possibly setting invalid object types to properties. Use *HYDMapObject*, which adds type checking before mapping values to their destinations.

And since this uses KVC, it will correctly convert boxed objects into their c-native types due to the implementation of KVC. This allows the rest of the mappers of Hydrant to use `NSNumber` which can get converted to integers, floats, doubles, etc.

If your key paths have dots, explicitly use *HYDAccessKey* and specify the key:

```
@{HYDKeyAccessor("json.key.with.dots"): @"key"}
```

Which can be useful for JSON that has dots in its key.

The following helper functions exist for this mapper:

```
HYDMapKVCObject(id<HYDMapper> innerMapper, Class sourceClass, Class destinationClass, NSDictionary *m
HYDMapKVCObject(id<HYDMapper> innerMapper, Class destinationClass, NSDictionary *mapping)
HYDMapKVCObject(Class sourceClass, Class destinationClass, NSDictionary *mapping)
HYDMapKVCObject(Class destinationClass, NSDictionary *mapping)
```

The all functions, except for the first one, are derived off the first helper function. If no mapper is provided, then *HYDMapIdentity* is used. Similarly, if no sourceClass is provided, `[NSDictionary class]` is used.

The `mapping` argument conforms to the *Mapping Data Structure*.

When specifying classes, this mapper will auto-promote them to their mutable types. All destination classes are constructed using `[destinationClass new]`. Classes that support NSMutableCopying are created using `[[destinationClass new] mutableCopy]`.

This makes it safe to use `[NSDictionary class]` and `[NSArray class]` as arguments for the `sourceClass` and `destinationClass`.

This object fully supports reverseMapping, which allows you to quickly create a serializer and deserializer combination.

HYDMapObject

This maps arbitrary properties from one object to another using a KeyPath-like mapping system. This mapper composes *HYDMapKVCObject* and *HYDMapTypes* to produce a mapper that can check types as it is mapped to its resulting object.

This mapper currently has tight-coupling around handling *HYDMapNonFatally* to ensure that optional mappings can still work as intended.

The following helper functions exist similar to `HYDMapKVCObject`:

```
HYDMapObject(id<HYDMapper> innerMapper, Class sourceClass, Class destinationClass, NSDictionary *mapp
HYDMapObject(id<HYDMapper> innerMapper, Class destinationClass, NSDictionary *mapping)
HYDMapObject(Class sourceClass, Class destinationClass, NSDictionary *mapping)
HYDMapObject(Class destinationClass, NSDictionary *mapping)
```

And like `HYDMapKVCObject`, the same default values apply:

- `innerMapper` defaults to *HYDMapIdentity*
- `sourceClass` defaults to `[NSDictionary class]`

Not surprisingly this also accepts a `mapping` argument described in the *Mapping Data Structure*. One notable difference is that using `HYDMapType` are implicit for all arguments.

Note: This mapper also verifies the types of source and destination classes using *HYDMapTypes*, so the *same notice* applies here for all types that are verified.

If you're mapping a collection of objects (such as an array of objects), see *HYDMapCollectionOf / HYDMapArrayOf* which is a composition of this mapper and `HYDMapArrayOf`.

If you prefer to not have type checking but still have the mapping functionality, use the lower-level *HYDMapKVCObject* instead.

HYDMapWithBlock

Note: This is a convenience to create custom Hydrant mappers. Blocks that execute custom code are subject to the same error handling that Hydrant expects for mappers to conform to *Mapper* in order to be exception-free.

This is a mapper that accepts one or two blocks for you to manually do the conversion. Unlike most other mappers, this does not provide any safety, but allows you do make trade-offs that go against Hydrant's design:

```
- Make a certain subset of the object graph being mapped to be more
  performant (instead of defensively checking the data as Hydrant does).
- Make a certain subset of the object graph "unsafe" and venerable to
  exceptions for easier debuggability.
- Perform complex mappings that cannot be sanely abstracted
```

```
- Quickly do one-off mappings for the perticular kind of data structure
  you're mapping (then ask: why are you using Hydrant then?)
- Store mutable state during the mapping to do more complex mappings that
  Hydrant does not support.
```

Try to avoid using this mapper, because it provides no benefits from implementing the serialization yourself. See *Mapping Techniques* for some tactics for mapping values without using this mapper.

These blocks take the same arguments as the `HYDMapper` protocol:

```
typedef id(^HYDConversionBlock)(id incomingValue, __autoreleasing HYDError **error);
```

Where errors can be filled to indicate to parent mappers that mapping has failed.

The helper functions for this mapper:

```
HYDMapWithBlock(HYDConversionBlock convertBlock)
HYDMapWithBlock(HYDConversionBlock convertBlock, HYDConversionBlock reverseConvertBlock)
```

Where the former function is an alias to latter as:

```
HYDMapWithBlock(convertBlock, convertBlock)
```

The `reverseConvertBlock` is called when `-[reverseMapper]` is called on the created mapper.

HYDMapWithPostProcessing

Note: This is a convenience to create custom Hydrant mappers. Blocks that execute custom code are subject to the same error handling that Hydrant expects for mappers to conform to *Mapper* in order to be exception-free.

This is a mapper that allows you to perform "post processing" from another mapper's work. Use this to "migrate" data structures that don't map cleanly from the source objects to the destination objects.

Unlike *HYDMapWithBlock*, this mapper provides access to the source input value and the resulting input value after executing the inner mapper.

Complex mappings across multiple source value fields can be done with this mapper, at the same expenses the `HYDMapWithBlock` does:

```
- Produce mappings that require composing multiple distinct parts of the
  source object.
- Allows extra mutation after the creation of an resulting object.
```

Try to avoid using this mapper, because it provides no benefits from implementing the serialization yourself. If you want to map multiple keys to a single value, see *Mapping Two Fields to One Property*.

The helpers functions for this mapper:

```
typedef void(^HYDPostProcessingBlock)(id sourceObject, id resultingObject, __autoreleasing HYDError *

HYDMapWithPostProcessing(HYDPostProcessingBlock block)
HYDMapWithPostProcessing(id<HYDMapper> innerMapper, HYDPostProcessingBlock block)
HYDMapWithPostProcessing(id<HYDMapper> innerMapper, HYDPostProcessingBlock block, HYDPostProcessingBl
```

Where the first function is aliased to the last function as:

```
HYDMapWithPostProcessing(HYDMapIdentity(), block, block)
```

and `reverseBlock` is the block that is invoked by the *The Reverse Mapper*.

An easy example is to convert an array of keys and values into a dictionary and then store it in a property of the resulting object:

```
id<HYDMapper> personMapper = ...; // defined somewhere else

// warning: there's no checking of sourceObject here, but you should
// if it is coming from an unknown source or hasn't been composed
// with HYDMapType
id<HYDMapper> mapper = \
    HYDMapWithPostProcessing(personMapper, ^(id sourceObject, id resultingObject, __autoreleasing HYD
        Person *person = resultingObject;
        person.phonesToFriends = [NSDictionary dictionaryWithObjects:sourceObject[@"names"] forKeys:s
    });

// example json
id json = @{...
            @"names": [@"John", @"Jane"],
            @"numbers": @[@1234567, @7654321]};

// post processor essentially does this:
person.phonesToFriends = [NSDictionary dictionaryWithObjects:json[@"names"] forKeys:json[@"numbers"]]
```

HYDMapReflectively

This builds upon various mappers and the Objective-C runtime to in the name of dry code at the expense of internal complexity (thus, debug-ability). It uses the runtime to try and intelligently generate mappings:

- Convert strings to dates with *HYDMapStringToAnyDate*
- Convert numbers to dates with *HYDMapNumberToDateSince*
- Converts numbers to strings and vice versa as needed
- Converts objects to strings for NSString properties
- Converts objects (to strings, then) to urls for NSURL properties
- Converts objects (to strings, then) to uuids for NSUUID properties
- Type check incoming values with *HYDMapTypes* to match the types of the properties being assigned

Since this mapper cannot determine the intended reverse mapping, you must explicitly state them if they differ from its configuration.

Unlike most mappers, this accepts optional configuration as property-blocks:

mapType(Class, id<HYDMapper>)

This allows you to always map a given objective-c class using a particular mapper:

```
// All properties of Person of type MyClass will use MyCustomMapper
HYDMapReflectively([Person class])
    .mapType([MyClass class], [MyCustomMapper new]);
```

optional(NSArray *propertyNames)

This allows you to easily specify optional fields. Accepts an array of property names (strings):

```
// firstName and lastName properties are optional. They can be nil.
HYDMapReflectively([Person class])
    .optional(@[@"firstName", @"lastName"]);
```

Optional and required cannot be used simultaneously.

required(NSArray *propertyNames)

This allows you to easily specify required fields. All other fields will be marked as optional. Accepts an array of property names (strings):

```
// firstName and lastName properties are required. They can be not nil.
// All other fields are optional.
HYDMapReflectively([Person class])
    .required(@[@"firstName", @"lastName"]);
```

Optional and required cannot be used simultaneously.

withNoRequiredFields

Alias to doing `required(@[])`:

```
// All fields are optional instead of the default of required.
HYDMapReflectively([Person class]).withNoRequiredFields
```

Optional and withNoRequiredFields cannot be used simultaneously.

only(NSArray *propertyNames)

This allows you to easily specify fields the reflective mapper will map. Accepts an array of property names (strings):

```
// firstName and lastName are the only fields set.
// All other fields are nil.
HYDMapReflectively([Person class])
    .only(@[@"firstName", @"lastName"]);
```

except(NSArray *propertyNames)

This allows you to easily exclude specific fields the reflective mapper should map. Accepts an array of property names (strings):

```
// firstName and lastName are the excluded from mapping.
// All other fields are still mapped.
HYDMapReflectively([Person class])
    .except(@[@"firstName", @"lastName"]);
```

customMapping(NSDictionary *mappingOverrides)

This allows you to specify custom mappings for particular fields. Accepts a dictionary like *HYDMapObject*:

```
// firstName property will map using MyCustomMapper.
HYDMapReflectively([Person class])
    .customMapping(@{@"firstName": @[[MyCustomMapper class], @"firstName"]});
```

keyTransformer(NSValueTransformer *keyTransformer)

This allows you to specify a value transformer that will convert destination keys to source keys. (eg - property names to JSON keys):

```
// Will map source object's first_name to Person's firstName property
HYDMapReflectively([Person class])
    .keyTransformer([HYDCamelToSnakeCaseValueTransformer new]);
```

HYDMapThread

This mapper simply calls its given mappers in-order until one emits an error. It's inspired from the LISP's threading macro, `->` except errors are returned without any subsequent mapper from knowing. Hydrant uses this internally to provide the convienence of accepting an inner mapper argument:

```
id<HYDMapper> HYDMapEnum(id<HYDMapper> innerMapper, NSDictionary *mapping) {
    return HYDMapThread(innerMapper, HYDMapEnum(mapping));
}
```

Like the accessors, HYDMapThread is a macro that accepts a variadic set of mappers to process in-order. The macro based off of the function:

```
HYDMapThreadMappersInArray(NSArray *mappers);
```

Where `mappers` is an array of mappers.

HYDMapDispatch

> **Warning:** Alpha - This API may change at any point. Please avoid using when possible.

Accessor Reference

Here lists all the accessors currently available in Hydrant. Accessors abstract the details of getting and setting values from objects so that each *Mapper* does not have to implement them individually. All the functions listed here return objects that conform to the *Accessor* protocol.

You might be thinking the overload functions listed require Objective-C++, but *clang supports function overloading*.

HYDAccessKeyPath

This accessor provides KeyPath-styled access to objects. They only support the dot-access from KeyPath to walk nested data structures. Like *HYDAccessKey*, this mapper will correctly convert boxed typed into their native c-types since it internally uses `-[valueForKey:]` and `-[setValue:forKey:]`.

If given `[NSNull null]` values, when setting values, then that **assignment is considered a no-op**, not assigning to nil. This is because the accessor cannot safely assign `nil` vs `[NSNull null]` (eg - property vs dictionary key).

There is a macro to create this accessor:

```
HYDAccessKeyPath(...)
```

Which takes a variatic sequence of NSStrings that represent the keyPaths to walk. Giving multiple keyPaths will generate a large array value and the expected input values when setting it too.

The macro is based off of the c function:

```
HYDAccessKeyPathInArray(NSArray *keyPaths)
```

As the name suggests, accepts an explicit array of keyPaths.

HYDAccessKey

This accessor provides KVC-styled access to objects. Like *HYDAccessKeyPath*, this mapper will correctly convert boxed typed into their native c-types since it internally uses `-[valueForKey:]` and `-[setValue:forKey:]`.

If given `[NSNull null]` values, when setting values, then that **assignment is considered a no-op**, not assigning to nil. This is because the accessor cannot safely assign `nil` vs `[NSNull null]` (eg - property vs dictionary key).

There is a macro to create this accessor:

```
HYDAccessKey(...)
```

Which takes a variatic sequence of NSStrings that represent the keys to walk. Giving multiple keyPaths will generate a large array value and the expected input values when setting it too.

The macro is based off of the c function:

```
HYDAccessKeyInArray(NSArray *keyPaths)
```

As the name suggests, accepts an explicit array of keys.

HYDAccessIndex

> **Warning:** This is feature alpha. It's API and capabilities may change between versions. Please avoid use if you can't accept instability.

This accessor provides index access to objects. This is useful for extracting values from an array where the order has a specific, known meaning.

If given `[NSNull null]` values, when setting values, then that **assignment is valid**. This is unlike the other mappers. Also, this accessor will place add `[NSNull null]` instances to arrays if they do not meet the size requirement that the accessor expects to update indicies.

There is a macro to create this accessor:

```
HYDAccessIndex(...)
```

Which takes a variatic sequence of NSNumbers that represent the indicies to read. Giving multiple indicies will generate a large array value and the expected input values when setting it too.

The macro is based off of the c function:

```
HYDAccessIndiciesInArray(NSArray *indicies)
```

As the name suggests, accepts an explicit array of indicies (NSNumbers).

HYDAccessDefault

This is a helper function that maps to the default accessor that Hydrant's mappers prefer. Handy if you need a default but don't have an opinion for your own mappers. Hydrant currently maps this to *HYDAccessKeyPath*.

There are two variants:

```
HYDAccessDefault(NSString *field)
HYDAccessDefault(NSArray *fields)
```

Which currently ties to the same behavior as `HYDAccessKeyPath`.

Formatter Reference

This is the reference documentation for all the NSFormatters that Hydrant implements as conveniences for you. When using mappers that utilize formatters:

- ***HYDMapObjectToStringByFormatter***
  - *HYDMapDateToString*
  - *HYDMapNumberToString*
  - *HYDMapUUIDToString*
  - *HYDMapURLToString*
- ***HYDMapStringToObjectByFormatter***
  - *HYDMapStringToDate*
  - *HYDMapStringToNumber*
  - *HYDMapStringToUUID*
  - *HYDMapStringToURL*

They are also publicly exposed if you need or prefer to use these classes for other purposes.

Date Format Strings

Hydrant also includes a variety of constants that map to common datetime formats that you can use for NSDateFormatter:

```
NSString *HYDDateFormatRFC3339 = @"yyyy'-'MM'-'dd'T'HH':'mm':'ssZ";
NSString *HYDDateFormatRFC3339_milliseconds = @"yyyy'-'MM'-'dd'T'HH':'mm':'ss.SSSZ";

NSString *HYDDateFormatRFC822_day_seconds_gmt = @"EEE, d MMM yyyy HH:mm:ss zzz";
NSString *HYDDateFormatRFC822_day_gmt = @"EEE, d MMM yyyy HH:mm zzz";
NSString *HYDDateFormatRFC822_day_seconds = @"EEE, d MMM yyyy HH:mm:ss";
NSString *HYDDateFormatRFC822_day = @"EEE, d MMM yyyy HH:mm";
NSString *HYDDateFormatRFC822_seconds_gmt = @"d MMM yyyy HH:mm:ss zzz";
NSString *HYDDateFormatRFC822_gmt = @"d MMM yyyy HH:mm zzz";
NSString *HYDDateFormatRFC822_seconds = @"d MMM yyyy HH:mm:ss";
NSString *HYDDateFormatRFC822 = @"d MMM yyyy HH:mm";
```

HYDDotNetDateFormatter

This is an NSDateFormatter subclass that supports parsing of microsoft AJAX date formats which look like `@"/Date(123456)/"`.

Since formatter sets some internal state from `NSDateFormatter`, so changing this formatter via properties may break its intended behavior.

HYDURLFormatter

This formatter utilizes NSURL to generate URLs and adds some extra safety by checking inputs before trying to construct an `NSURL`.

It can optionally be constructed with a set of schemes to allow.

HYDUUIDFormatter

This formatter utilizes NSUUID to generate UUIDs and adds some extra safety by checking inputs before trying to construct an `NSUUID`.

Value Transformer Reference

This is the reference documentation for all the NSValueTransformers that Hydrant implements as conveniences for you. Currently, these transformers are intended to be used with *HYDMapReflectively*.

They are also publicly exposed if you need or prefer to use these classes for other purposes.

HYDBlockValueTransformer

This value transformer is a simple abstraction to allow custom blocks instead of having to implement a custom value transformer. The Block Value Transformer simply accepts two blocks: one for transforming and another for reverse transforming.

Due to the limitation of the `NSValueTransformer` contract, the block value transformer always returns `YES` for `+[allowsReverseTransformation]`.

Example:

```
HYDBlockValueTransformer *transformer = [[HYDBlockValueTransformer alloc] initWithBlock:^id(id value)
    return [value componentsSeparatedByString:@"-"];
} reversedBlock:^id(id transformedValue){
    return [value componentsJoinedByString:@"-"];
}];

[transformer transformValue:@"foo-bar"] // => @[@"foo", @"bar"];
[transformer reverseTransformedValue:@[@"foo", @"bar"]] // => @"foo-bar"
```

HYDIdentityValueTransformer

This value transformer is a no-op, simply returning the value it has received. When used with *HYDMapReflectively*, this can be an easy way to make your objects map directly to the source objects.

Example:

```
HYDIdentityValueTransformer *transformer = [HYDIdentityValueTransformer new];

[transformer transformValue:@"foo"] // => @"foo"
[transformer reverseTransformedValue:@1] // => @1
```

HYDReversedValueTransformer

This value transformer reverses another value transformer. Essentially, this transformer converts:

- `-[transformValue:]` into `-[reverseTransformedValue:]`
- and `-[reverseTransformedValue:]` into `-[transformValue:]`

The given value transformer should be reversable.

HYDCamelToSnakeCaseValueTransformer

This value transformer converts camel case to snake case. You can optionally specific if it is UpperCamelCase or lowerCamelCase by specifying one of the following enums:

```
HYDCamelCaseLowerStyle // default when not specified
HYDCamelCaseUpperStyle
```

This transformer expects `NSStrings`. This is useful for *HYDMapReflectively* to convert snake-cased JSON keys into the more familiar Objective-C style of lower camel-case:

```
HYDCamelToSnakeCaseValueTransformer *transformer = [[HYDCamelToSnakeCaseValueTransformer alloc] initW

[transformer transformValue:@"foo_bar"] // => @"FooBar"
[transformer reverseTransformedValue:@"FooBar"] // => @"foo_bar"
```

Contributing to Hydrant

Hydrant is a relatively young project that can use plenty of help from anyone. Things listed here are rough guidelines, but are a good starting direction.

If you have any questions, feel free to email jeff at jeffhui.net with some subject talking about Hydrant.

Filing Bugs

Found a bug? File an issue on the GitHub page. When you write on up, be sure to include:

- Steps to reproduce the behavior in question. Small code examples that demonstrate is are excellent!
- Expected behavior, what expected to happen.
- Actual behavior, what actually happened.

These points are mostly a starting conversation about the issue.

Contributing Code

Code is welcomed, although unlike application code, is subjected to more scrutiny. Since the most expense part about software is maintaince, so pull requests require the following:

- Tests that can verify the regression (if a bug) or validate a new feature
- If its a new feature, documentation explaining the new feature.

It's good to keep in mind the original *design* of Hydrant when writing the code.

Feedback about the pull request or code contributes are never towards the contributor, but instead towards improving the code.

Changelog

Here lists all the changes that have occurred in Hydrant from version to version. They are ordered by version, then by significance of change (from breaking changes to minor bug fixes).

v2.0.0

- Updated compatibility for latest stable iOS SDKs (9.0 - Xcode 7.3.1)
- Minor documentation fixes

v2.0.0-alpha.2

- Updated for the latest iOS SDKs (8.4, 9.0)
- Properly handles some non-ascii characters in URLs.

v2.0.0-alpha.2

- Fix bugs where stringification does not work for some mappers
- HYDReflectiveMapper supports .require() to specify required fields instead of only optional fields.
- Converted from static library to iOS Framework

v1.0.1

Updated for Xcode 6.1

v1.0.0

Initial public release. Includes documentation (such as this file).