Test-Driven Development of REST Clients

Chris Woodard

Test-Driven Development of REST Clients

REST stands for ***RE***presentational ***S***tate ***T***ransfer. It is a standard way to for a web client to communicate with a web API.

- *REST* uses the HTTP protocol for all of the interaction between the client and the web API.

- *REST* uses the URI (uniform resource indicator) to tell the web API which resource to operate on.

- *REST* uses the HTTP request headers to pass in parameters (such as authorization tokens) that the API requires.

Test-Driven Development of REST Clients

REST uses the standard HTTP verbs to perform actions on resources (which are specified by */uri*):

GET /uri Fetch the resource specified by */uri*

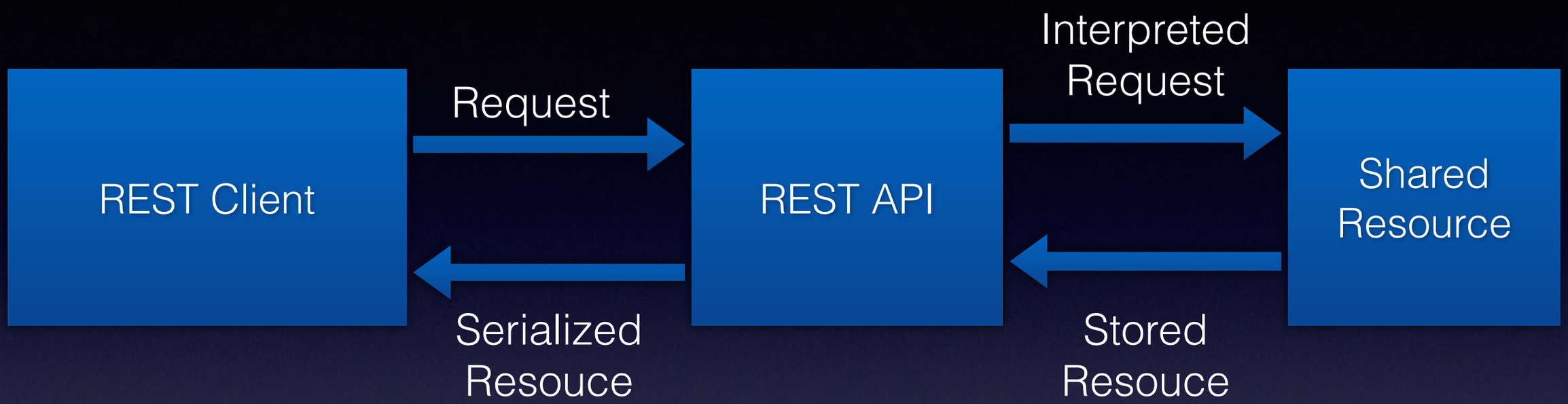
PUT /uri Replace the resource specified by */uri*

POST /uri Add the resource specified by */uri*

DELETE /uri Delete the resource specified by */uri*

...

Test-Driven Development of REST Clients



- REST client connects to computer housing REST API and either requests or sends a resource
- REST API attempts to perform the transaction and sends back either the resource or an error

Test-Driven Development of REST Clients

Standard format these days is JSON (JavaScript Object Notation)

```
{
…
"current_observation": {
    "image": {
    "url":"http://icons-ak.wxug.com/
graphics/wu2/logo_130x80.png",
    "title":"Weather Underground",
    "link":"http://www.wunderground.com"
    },
    "display_location": {
    "full":"Tampa, FL",
    "city":"Tampa",
    “state":"FL",

… snip …

    "wmo":"99999",
    "latitude":"27.95783043",
    "longitude":"-82.45883179",
    "elevation":"17.00000000"
    },
```

NSJSONSerialization

```
{
    "current_observation" =     {
        UV = 0;
        "dewpoint_c" = 15;
        "dewpoint_f" = 59;
        "dewpoint_string" = "59 F (15 C)";
        "display_location" =         {
            city = Tampa;
            country = US;
            "country_iso3166" = US;
            elevation = "17.00000000";
            full = "Tampa, FL";
            latitude = "27.95783043";
            longitude = "-82.45883179";
            magic = 1;
            state = FL;
            "state_name" = Florida;
            wmo = 99999;
            zip = 33601;
        };
…
```

JSON response from request to WeatherUnderground API for current conditions

NSDictionary produced from the JSON by the NSJSONSerialization API.

Test-Driven Development of REST Clients

Unit Testing

Test-Driven Development of REST Clients

Unit testing is testing the smallest identifiable part of your app's functionality.

A *unit* has the following characteristics:

atomic - cannot be easily divided into parts

independent - does not depend on other parts of your code.

encapsulated - does not have to be pasted into your test code in order to be tested.

A unit *can* be as small as an expression but is nearly always a method.

Test-Driven Development of REST Clients

Unit tests should be grouped together; in Xcode this is called a *test case*.

A *test case* contains multiple (sometimes many) individual tests.

Each test is implemented as a method with a name that has a very specific format:

-(void)test*LongerNameDescribingTheTest*

Test-Driven Development of REST Clients

For example:

```
-(void)testMakeSureYouCanCreateAnInstance
{
    NSObject *testObject;
    testObject = [[NSObject alloc] init];
    XCTAssertNotNil( testObject, @"unable to allocate object" );
}

-(void)testMakeSureTwoPlusTwoIsFour
{
    int numA = 2;
    int numB = 2;
    XCTAssertTrue( 4 == (numA + numB), @"2 + 2 must equal 4" );
}
```

Test-Driven Development of REST Clients

What makes it “test driven”?

- You write the tests before you can write production code.
- You design your client code with this in mind - modular and well-separated.
- You write your tests and supporting code with this in mind as well.

Test-Driven Development of REST Clients

How does that work? How do the tests have to be written?

- Tests have to be *reproducible* - if you start the test with the same conditions and exercise the same code, you have to wind up with the same results.
- Test code should not interfere with the code being tested, otherwise you're testing the supporting code and tests instead of the app code.

Test-Driven Development of REST Clients

How can you write tests like this for an API you don't control and which may not exist?

You *simulate* the API in such a way that your client code talks to it in exactly the same way (as far as it knows) that it will talk to the real API.

Test-Driven Development of REST Clients

How do you simulate a web API?

- You can write a small single-purpose web server in Ruby or Python and implement the API there.
- You can add a small in-process web server that runs within your app and implement the API there.
- **You can mock up the API in the testing bundle.**

Test-Driven Development of REST Clients

Mocking up the API in the testing bundle

- *API mocking* - code that returns canned data based on the requests it receives and any controlling switches that are set.

- *Dependency injection* - in this case it means that same thing as code redirection. Intercept the NSURLProtocol method calls that the REST client makes through the NSURLConnection API and redirect them to your mock API.

Test-Driven Development of REST Clients

OHHTTPStubs

https://github.com/AliSoftware/OHHTTPStubs

- Uses custom NSURLProtocol to intercept NSURLConnection requests and stubs them.

- Lets you specify a custom URL handler for each unique URL request you want to mock

- Allows your handler to construct the NSURLResponse in code so that your app code *thinks* it came from the API.

Test-Driven Development of REST Clients

How do we use this?

- Capture JSON from the API for each call you're implementing in the app.
- Add OHHTTPStubs to your project and make sure it's added to the testing bundle.
- Write an object that installs OHHTTPStubs request handlers and then returns the appropriate JSON for each request.

Test-Driven Development of REST Clients

How do we *test* this?

```
-(void)testMockWeatherAPITampaFLConditionsNotNil
{
    self.didFinish = NO;
    self.didFail = NO;
    __block NSError *err = nil;
    __block CurrentWeather *weather = nil;
    self.mockAPI.httpStatusCode = 200;

    [_fetcher
fetchCurrentLocalWeatherWithCompletion:^(NSHTTPURLResponse
*response, NSError *error){
            weather = [_fetcher currentWeather];
            self.didFinish = YES;
            if(nil == error)
                self.didFail = NO;
            else
                self.didFail = YES;
            err = [error copy];
        }
    ];

    while(!self.didFinish && !self.didFail)
    {
        [[NSRunLoop currentRunLoop] runUntilDate:[NSDate
dateWithTimeIntervalSinceNow:5]];
    }

    XCTAssertNotNil( weather, @"weather should not be nil" );
}
```

- In the XCTest case define instance variables didFinish and didFail.
- In each test, invoke the WeatherFetcher method. In the completion block check the error and set didFinish and didFail appropriately.
- Immediately after the invocation enter a while() loop and call the current runLoop to run for an appropriate amount of time.

Test-Driven Development of REST Clients

How do we force errors with this?

```
-(void)testMockWeatherAPITampaFLForecast400Status
{
    self.didFinish = NO;
    self.didFail = NO;
    __block NSError *err = nil;
    __block WeatherForecast *forecast = nil;
    self.mockAPI.sendNilForecast = YES;
    self.mockAPI.httpStatusCode = 400;
    __block NSHTTPURLResponse *resp = nil;

    [_fetcher
fetch3DayLocalForecastWithCompletion:^(NSHTTPURLResponse *response,
NSError *error){
            forecast = [_fetcher lastForecast];
            self.didFinish = YES;
            if(nil == error)
                self.didFail = NO;
            else
                self.didFail = YES;

            err = [error copy];
        }
    ];

    while(!self.didFinish && !self.didFail)
    {
        [[NSRunLoop currentRunLoop] runUntilDate:[NSDate
dateWithTimeIntervalSinceNow:5]];
    }

    XCTAssertEqual([resp statusCode], 400, @"should have a 400
response (%d)", [resp statusCode]);
}
```

- In the mock API define public properties that signal when various errors are to be returned.
- Set those properties in each test.
- In the mock API request handlers, test the values of the error switches and return good responses or error responses as appropriate.

Test-Driven Development of REST Clients

Resources

- OHHTTPStubs - https://github.com/AliSoftware/OHHTTPStubs

- REST
  - http://en.wikipedia.org/wiki/Representational_state_transfer

  - http://www.slideshare.net/rmaclean/json-and-rest