
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

10: Improving App Quality with Test Driven Development

Automated testing tools for iOS have come a long way since the initial release of the iPhone SDK. Learn how to improve your app's quality by using TDD to build both the model and user interface layers of an application. You'll learn what TDD is, how it can be used in unit tests to verify simple model objects, code that uses a remote API, and user interface code. Plus: some tricks for writing tests easier!

28 Improving App Quality with Test Driven Development: Demo 1

By Andy Obusek

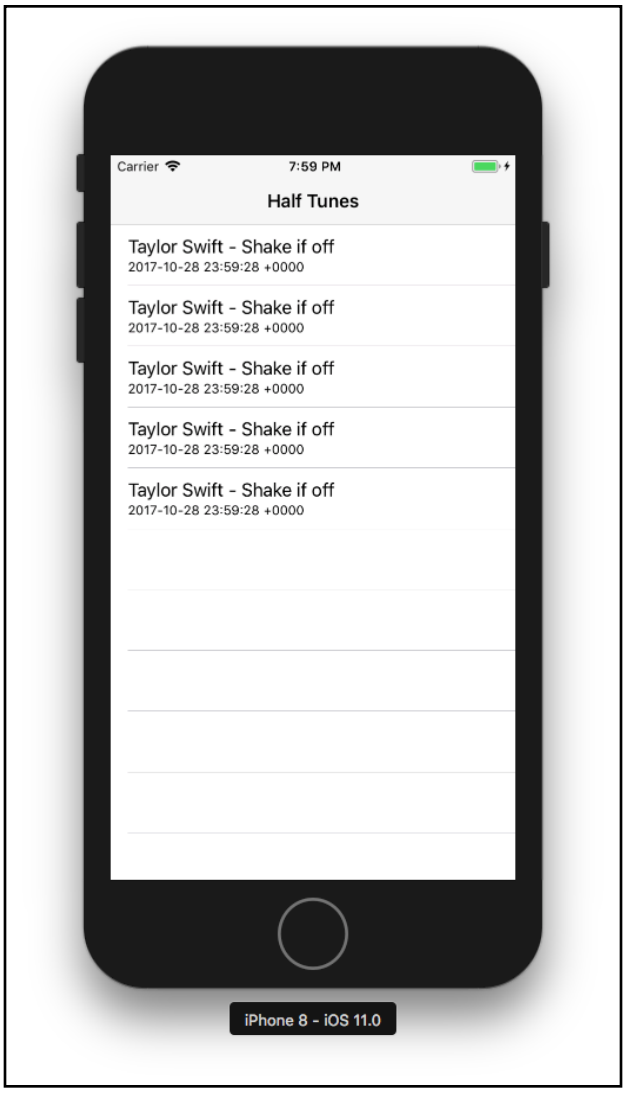
In this demo, you will fix an existing bug in an application using test driven development, and then write some new code using test driven development.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo1\starter**.

1) Fix an Existing Bug

Imagine you've just inherited a partially complete code base from another developer for an app that's intended to search the iTunes API, and it's now your responsibility to finish the project. The current state of the app looks like:



Unfortunately the date format being used for each row isn't what's desired by your client. Instead they require the date format to be shown like "October 28, 2014." You will now fix that bug using test driven development.

> **Note**: Pair programming is a great way to practice test driven development. As you go forth beyond RWDevCon, find someone to practice with.

Write a Failing Test

Open **TuneTests.swift**. Add the following unit test to `TuneTests`:

```
func testTune_Always_CanFormatReleaseDate() {

}
```

Right now this is just an empty unit test that doesn't do anything. Take notice of

the name format for the test. Roy Osherove proposes test names that follow the format:

`testUnitOfWork_StateUnderTest_ExpectedBehavior`

Naming your tests this way helps the tests further serve as documentation for your code.

Add the following code to `testTune_Always_CanFormatReleaseDate()`:

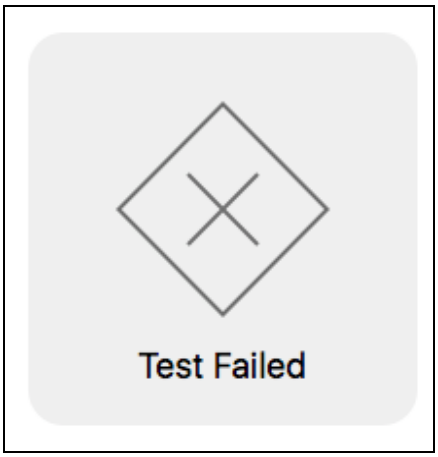
```
// 1
let formatter = DateFormatter()
formatter.dateFormat = "MM/dd/yyyy"
guard let releaseDate = formatter.date(from: "10/28/2014") else {
  XCTFail()
  return
}

// 2
let toTest = Tune(artist: "Taylor Swift",
                  name: "Shake if off",
                  releaseDate: releaseDate)
let formattedDate = toTest.imaTerribleMethodName()

// 3
XCTAssertEqual("October 28, 2014", formattedDate)
```

You've just added the minimum amount of unit test code to create a failing test for verifying the date formatting ability of `Tune`.

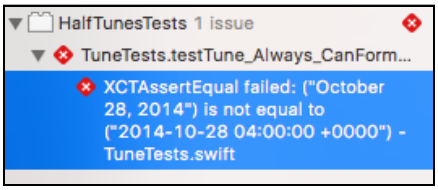
1. First, create some plumbing code to ultimately create a `Date` with a known value.
2. Create your "object under test" and call the "method under test" saving the result.
3. Verify the result of the method under test against the desired outcome.

Run your test. There are several ways to do this, my favorite is the keyboard shortcut to run a single test: **Command-Option-Control-u**. (Xcode's cursor must be within the test that you want to run.)

Once the test runs, you will see this result flash on the screen:



You should see the following output in the Xcode issue navigator that indicates the test has failed:



Congratulations, you just took your first step in the TDD cycle, writing a failing test! Now to fix this broken code in the object under test.

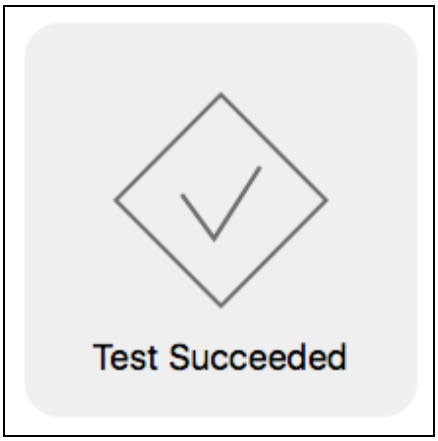
In order to fix the failing test, you will update the object under test with the correct behavior, and then re-run your unit test to validate that the code is working as expected.

Open **Tune.swift** and update `imaTerribleMethodName()` with this code:

```
func imaTerribleMethodName() -> String {
  let formatter = DateFormatter()
  formatter.dateFormat = "MMMM dd, yyyy"
  return formatter.string(from: releaseDate)
}
```

You just updated `imaTerribleMethodName()` with code to properly format `releaseDate` for display. To verify that, re-run your unit test. This time, use the keyboard shortcut to re-run the "last test": **Command-Option-Control-g**.

Once the test runs, you will see this result flash on the screen:



You should see the following output in the Xcode issue navigator that indicates the test has passed:

Nice work! You've now updated `Tune` (the object under test) to correctly format the `releaseDate`. And what's even better is that you now have a unit test that you can run at any point in the future to verify the functionality still works.

Now for third step in the TDD cycle: refactoring!

Back in 1999, Martin Fowler (with the help of some others) published a book called Refactoring, Improving the Design of Existing Code. In this book, he proposes a list of refactorings that can help your code be more maintainable. You'll now try one, Rename Method! And now that you have tested code, you can confidently change it knowing that you'll be able to verify that it still works in an automated way.

The same rules apply in this step, always start by changing your test code first making the smallest possible change to trigger failing test (compilation failure included).

Open **TuneTests.swift** and find this line:

```
  let formattedDate = toTest.imaTerribleMethodName()
```

and change it to:

```
  let formattedDate = toTest.formattedDate()
```

Obviously `formattedDate()` is a better name than `imaTerribleMethodName()`, right?

Re-run your test using **Command-Option-Control-g**. You'll see that the code does not compile. Yay, a failing test!

Time to make the test pass!

You will write the least code possible to make the test pass. Open **Tune.swift** and rename `imaTerribleMethodName()` to `formattedDate()`.

Find this line:

```
func imaTerribleMethodName() -> String {
```

and replace it with:

```
func formattedDate() -> String {
```

There's also a compilation error in `TunesTableViewController`. To fix that, open **TunesTableViewController.swift** and in `TunesTableViewController` find this line:

```
cell.detailTextLabel?.text = sampleTune.imaTerribleMethodName()
```

and replace it with:

```
cell.detailTextLabel?.text = sampleTune.formattedDate()
```

Re-run your test using **Command-Option-Control-g**. You'll see that not only does the code now compile, but the test passes!

You've now gone through the first full loop through the test driven development cycle!

2) Use TDD for New Code

On the way to building your app to search the iTunes Search API, you need a way to parse the JSON response that comes back from the API. You will use the new `Decodeable` API in Swift 4 to do this. And what's better, is that you and your partner are going to use TDD to build this.

When writing new code, the TDD cycle is very similar to that when fixing bugs. The only difference is that it's, well, new code!

You will start by writing a failing test for a method to parse the JSON response from the iTunes Search API. Open **QueryServiceTests.swift**:

Add a new test:

```
func testQueryService_ToCreateTunes_FromJSON() {

}
```

As the name easily indicates, you are about to write a test to verify that `QueryService` can create a bunch of `Tunes` from a piece of JSON. Run your new test, this time by clicking the empty diamond symbol next to the definition of the test

method:

You'll see the diamond turn green, indicating that the test passed.

Since the test hasn't gotten to a point where it fails yet, you can continue coding the test.

Add this line to `testQueryService_ToCreateTunes_FromJSON()`:

```
let toTest = QueryService()
```

`QueryService` is the name of a new class you would like to use to parse the JSON from the iTunes Search API. This line creates an instance of a `QueryService` for use in the test.

Re-run your test with **Command-Option-Control-g**. You'll see the test does not compile, thus a "failure".

It's time to update your system under test such that the test will pass. At this point you are only trying to construct an instance of a new class `QueryService`.

Open **QueryService.swift** and add the following code:

```
class QueryService {

}
```

This is an empty implementation of a new class called `QueryService`. Re-run your test with **Command-Option-Control-g**. You'll see that the test passes, woohoo!

It's time to now update the test further with the most minimal code possible to get to a failing test.

Add the following code to `testQueryService_ToCreateTunes_FromJSON()`:

```
let result = toTest.createTunesFrom(data: sampleResponse)
```

`createTunesFrom(data:)` is a new method that you'd like to add to `QueryService` to do the actual parsing of the data that will eventually come from the remote API.

Re-run your test with **Command-Option-Control-g**. You'll see the test does not compile, thus a "failure".

Test is failing...bingo! Make it pass!

In order to make the test pass, `QueryService` must be updated with a minimal implementation of `createTunesFrom(data:)`. Add this code to `QueryService` do that:

```
func createTunesFrom(data: Data) -> [Tune] {
  return []
}
```

While this implementation doesn't really do anything, the name itself does tell you a lot about what it will eventually do: create and return a list of `Tune` objects from a `Data` input. Re-run your test with **Command-Option-Control-g**. You'll see that the test passes, woohoo!

The method still doesn't actually do anything, but it's close! Now, you'll update the test once more so that it fails.

Ultimately, you'd like `createTunesFrom(data:)` to create a list of `Tunes`. The next logical step in your test is take the first result in the list that is returned, and inspect it for expected values. But remember, you can only add as much code to the test that will make the test fail! Don't go too far! Add the following line to the test:

```
let firstTune = result[0]
```

Re-run your test with **Command-Option-Control-g**. The code compiles, but it crashes! That makes sense because you hard coded `createTunesFrom(data:)` to return an empty list. A crashing test is a failing test!

At this point, the test expects a list of a single `Tune` to come back from `createTunesFrom(data:)`. That's easy enough to do, update `createTunesFrom(data:)` to look like:

```
func createTunesFrom(data: Data) -> [Tune] {
  return [Tune(artist: "", name: "", releaseDate: Date())]
}
```

This code now returns a list of a single `Tune` hard coded with some bogus values. Re-run your test with **Command-Option-Control-g**. You'll see that the test passes! The method is not yet functional for the requirements of the app, but the test passes, so back to the tests for some more code to make it fail!

At this point, `createTunesFrom(data:)` is hard coded to return a single `Tune` with some bogus values. You now need to add some code to your test to verify that the result that comes back matches what's expected. Sounds like some `XCTAsserts` are in order!

The sample project includes a sample response JSON string (that has already been converted to `Data`) that you can use to verify against. Add this code to the end of `testQueryService_ToCreateTunes_FromJSON()`:

```
XCTAssertEqual("Jack Johnson", firstTune.artist)
```

This asserts that the returned `Tune`'s artist is "Jack Johnson" - which matches the static response that is hard coded in the test.

Re-run your test with **Command-Option-Control-g**. You'll see that the test fails. This is because `QueryService` is still hard coded to return a bogus response - it does nothing with the provided input. But the test fails, so that's great! Time to update the method under test to pass!

This might be the most exciting part for you, you will actually make `createTunesFrom(data:)` do something with the data! You need to update the method under test to parse the `Data` parameter into a `Tune` with a properly populated `artist`. You'll use the Swift 4 `Decodeable` protocol to do this.

First, `Tune` must be updated to indicate that it can be used with `Decodeable`. In **Tune.swift**, update the class definition to look like:

```
class Tune: Decodable
```

This indicates that the class conforms to `Decodeable`.

Add this code to `Tune`:

```
enum TuneKeys: String, CodingKey {
  case artist = "artistName"
  case name = "trackName"
  case releaseDate = "releaseDate"
}

required init(from decoder: Decoder) throws {
  let container = try decoder.container(keyedBy: TuneKeys.self)
  self.artist = try container.decode(String.self,
                                     forKey: .artist)

  // Note: These are placeholder/bogus assignments
  self.name = ""
  self.releaseDate = Date()
}
```

This code indicates how JSON should be parsed into a `Tune`.

Back in **QueryService.swift**, add this code to `QueryService`:

```
struct QueryResponse: Decodable {
  let resultCount: Int
  let results: [Tune]
}
```

This struct is useful to the `Decodeable` protocol in that it indicates the response will not just be a list of `Tunes` but, also include some other data as well (a `resultCount`).

Finally, update `createTunesFrom(data:)` to look like:

```
func createTunesFrom(data: Data) -> [Tune] {
  var toReturn: [Tune] = []
  do {
    toReturn =
      try JSONDecoder().decode(QueryResponse.self,
                               from: data).results
  } catch {
    // do nothing
  }
  return toReturn
}
```

That connects everything together and actualy parses the provided `Data` into our Swift objects, and returns the list of `Tune` results.

Re-run your test with **Command-Option-Control-g**. The test passes! Woohoo!

In case you forgot, the corresponding (and now passing) test looks like:

```
func testQueryService_ToCreateTunes_FromJSON() {
  let toTest = QueryService()

  let result = toTest.createTunesFrom(data: sampleResponse)
  let firstTune = result[0]

  XCTAssertEqual("Jack Johnson", firstTune.artist)
}
```

Now that this test passes, this means that `QueryService` is actually built to take a `Data` response and parse it into a list of `Tune` objects that are populated and ready to be used (well at least the `artist` property)! And what's better, is that this was all built with TDD, so you can have confidence that this test actually verifies what it's intended to verify. Just to confirm, change the assertion to look like:

```
XCTAssertEqual("Jimmy Johnson", firstTune.artist)
```

Re-run you test. You'll notice that it does not pass. This is a good thing. You changed the expected outcome to some other value than what is in the actual hard coded JSON response. It's a good thing that your test catches this. Otherwise your test wouldn't be doing a thing.

3) That's it!

Congratulations, at this time you should have a good understanding of how to use test driven development on plain old `NSObjects` when writing new code and also fixing existing bugs! It's time to move onto learning how to use test driven development when working in some more complicated situations.

29 Improving App Quality with Test Driven Development: Demo 2

By Andy Obusek

In this demo, you will write code to connect a web service using TDD. Then you'll swap out the "real" call to the web service with a mock response. And finally, using the mocked responses, you'll improve the error handling of your web service connectivity code using TDD.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo2\starter**.

1) Connect to a Web Service

For this demo, be sure to open the workspace file, **HalfTunes.xcworkspace** as later in the demo you'll be using Cocoapod dependencies.

The iTunes Search API will be used as the remote web service for connecting your app. It's a free web service that doesn't even require any special tokens as part of the request. To access this web service, you'll expand `QueryService` using test driven development to search the API with a term, and then return results via a completion handler that provides a `[Tune]`.

Write a Failing Test

As you learned in Demo 1, the first thing to do when approaching a problem with test driven development, is to write just enough code in your unit test such that the test fails (and compilation failures count as a test failure). Start by adding an empty test to **QueryServiceTests.swift**:

```
func testQueryService_AccessesAPI_AndReturnsCorrectResult() {
```

```
}
```

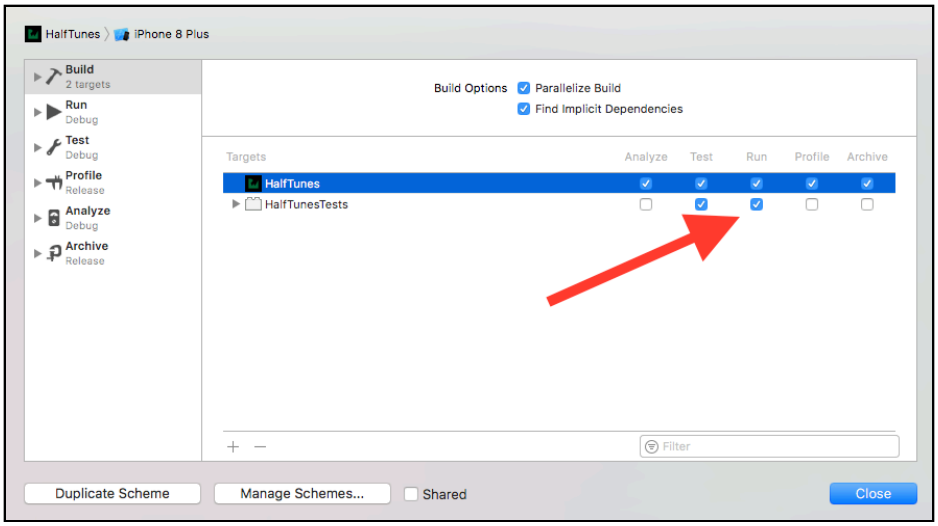
While this test does nothing, its name clearly explains what it intends to validate.

Next, add this to `testQueryService_AccessesAPI_AndReturnsCorrectResult()`

```
let toTest = QueryService()
let termToSearchFor = "phish"
```

Compile your tests using **Command-B**

> **Note**: It's recommended to always build your test targets when building to Run. Double check this in your scheme settings (**Command-Shift-,**). Not only does this help with the TDD workflow, but also when making code changes it will help catch any bugs or test compliation failures as early as possible so the impact of your changes can be considered.



Now add a call to the method that will return search results via a completion handler by adding this to the test, and then build your project.

```
toTest.getSearchResults(searchTerm: termToSearchFor) { tunes in

}
```

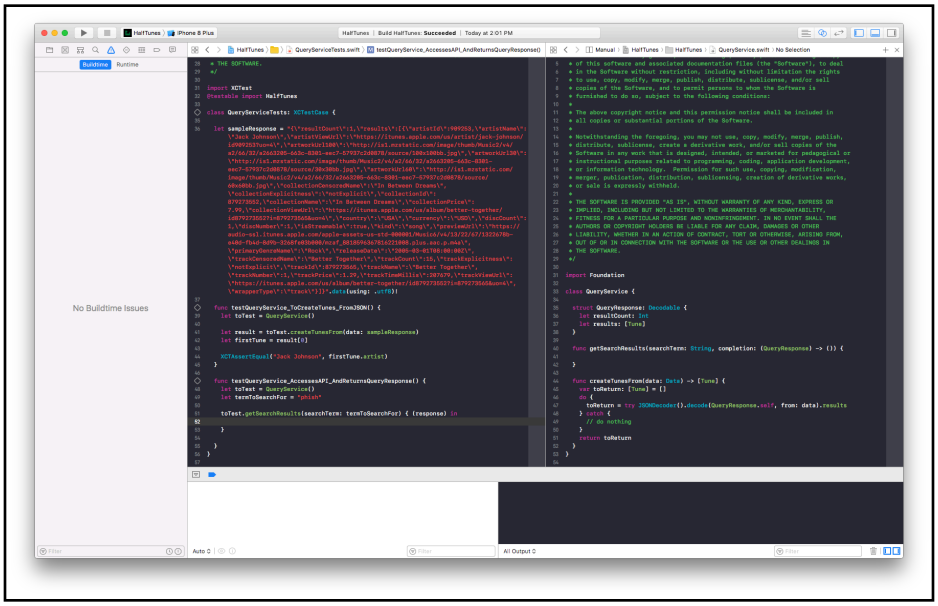
Remember, this method doesn't exist yet, so it's awesome that you get a compilation error! A compilation error counts as a test failure. It means the compiler is really doing its job and your project is setup correctly. ;] If you didn't get a compiler error, revisit the **Note** above and verify that your scheme settings correctly build your tests when building to run.

Make the Test Pass

It's time to make your test compile. Open **QueryService.swift** and add this code:

```
func getSearchResults(searchTerm: String, completion: ([Tune]) -> ()) {

}
```

It's recommended to use the assistant editor to view your test side by side with the class under test. You can open this by selecting the menu option: **View -> Assistant Editor -> Show Assistant Editor** and then use the jump bar to open **QueryService.swift** side by side with **QueryServiceTests.swift**



Now it's time to build your project again. You can either do this with **Command-B** or even switch back to actually running your tests with **Command-U**. Everything should compile and pass. Bingo, time to add more code to your tests!

Update the Test to Fail

Everyone is familiar with the chart topping rock band Phish - and this includes the iTunes Search API. You can verify this yourself by opening https://itunes.apple.com/search?term=phish in your browser and looking at the resulting JSON.

It's time to update your test (`testQueryService_AccessesAPI_AndReturnsCorrectResult()`) to verify that `QueryService` also can find songs by Phish when searching the iTunes API. In the callback for `getSearchResults(searchTerm:completion:)` verify that the first result is by the artist Phish.

```
toTest.getSearchResults(searchTerm: termToSearchFor) { tunes in
  XCTAssertEqual(tunes[0].artist, "Phish")
}
```

Run your tests with **Command-U**. You'll see the tests pass. This should be a surprise - they should not pass. You just added an assertion to verify a datapoint in the response to an web service call, and it passes, yet you never added any code to make the actual web service call.

The problem is that since the assertion is in the completion block, the test finishes execution before the assertion is made, and thus the tests pass. You need to add more code to the test to make sure it fails.

> **Note**: This is an eye opening moment for the benefits of test driven development. Had you been writing your tests after the fact, a bug like this in your tests is extremely easy to miss, and without looking for that test failure first, you may have inadvertently moved on from your test thinking it was actually verifying something, when that wasn't the case.

Luckily XCTest provides a tool to solve this problem, expectations. Update `testQueryService_AccessesAPI_AndReturnsCorrectResult()` to this:

```
func testQueryService_AccessesAPI_AndReturnsCorrectResult() {
  let toTest = QueryService()
  let termToSearchFor = "phish"

  let expectation = XCTestExpectation(description: "Call web service")

  toTest.getSearchResults(searchTerm: termToSearchFor) { tunes in
    XCTAssertEqual(tunes[0].artist, "Phish")
    expectation.fulfill()
  }

  wait(for: [expectation], timeout: 2.0)

}
```

Using `XCTestExpectation` your test will now wait until the completion block is called before considering itself a success. Re-run your tests. `testQueryService_AccessesAPI_AndReturnsCorrectResult()` should fail. Congratulations, you just experienced the power of test driven development.

Make the Test Pass

Now that you have a failing test, time to update `QueryService` to make the test pass. In `QueryService` add the following code to `getSearchResults(searchTerm:completion:)`:

```
func getSearchResults(searchTerm: String,
                      completion: @escaping ([Tune]) -> ()) {

  guard let url =
    URL(string: "https://itunes.apple.com/search?term=\(searchTerm)")
else {
```

```
      return
    }
    let dataTask = URLSession(configuration: .default)
                     .dataTask(with: url) {
      [weak self] (data, response, error) in

      guard let strongSelf = self,
        let data = data else {
        return
      }
      let tunes = strongSelf.createTunesFrom(data: data)
      completion(tunes)
    }
    dataTask.resume()
  }
```

This is some basic code to call the iTunes Search API with the provided term, and return the results. Re-run your tests. They should pass.

This is great news! Now you have a piece of code that actually calls web service, parses the result, and returns it as a model object. And furthermore, it's all tested code!

One easy thing that you can do to gain further confidence that your test is actually doing something is to change the expected value in the assertion to something bogus. If you re-run the test, the test should fail. If it doesn't, that's bad news because it means your test was never actually verifying anything.

Find this line in `QueryServiceTests`:

```
  XCTAssertEqual(tunes[0].artist, "Phish")
```

and change "Phish" to "Grateful Dead".

Now your test is represents a verification that if you search the iTunes Search API for "Phish" that the first result will be "Grateful Dead". Sure they sound similar, but any real phan will tell you there's no comparison. ;] Re-run your tests, and you'll see that they fail. Perfect!

Take a moment to reflect on what you've done. You were able to write a model layer to call a web service without actually needing to create any UI at all in your app. No need to continually Build and Run, and tap around just to build out your model layer. Using test driven development it was much speedier, AND you have automated tests taboot.

Things aren't perfect though. While it was great to write your model layer using TDD, right now your tests continue to actually call the "real" iTunes Search API and make assertions against it. This is problematic for two main reasons: 1) it's slow making networking connections and you want your unit tests to be lightning fast, 2) assertions against data from an unreliable source (the iTunes Search API) will fail anytime the data changes and you want your unit tests to be 100% reliable without

any flakiness.

The solution to this problem is to swap the call to the real iTunes Search API with stubbed data.

2) Stub the Web Service

It's important that your unit tests run fast and are reliable. While it was useful to develop your model layer with test driven development, the result was unit tests that make web service calls. Calling a web service is slow, and you can't rely on the data to always be the same. To correct this, you will stub the web service to return fake data in your tests. In order to stub the iTunes Search API, you will use a framework called Mockingjay. Mockingjay makes it really easy to stub HTTP requests from your test cases.

Mockingjay was already included with the starter project. In the future for use with other projects, it's recommended to use Mockingjay with Cocoapods.

First, to make Mockingjay available to your `QueryServiceTests`, add this to the top of **QueryServiceTests.swift** (where the other `import` statements are):

```
import Mockingjay
```

Now you will be able to use Mockingjay in your tests. Build your project to make sure everything still compiles.

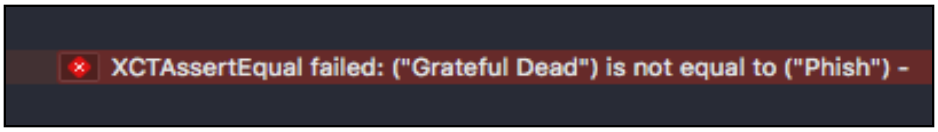
The next thing to do, is to use Mockingjay to configure a stubbed endpoint to return fake data when executing from `testQueryService_AccessesAPI_AndReturnsCorrectResult()`. Add this code to `testQueryService_AccessesAPI_AndReturnsCorrectResult()` right after where `termToSearchFor` is defined:

```
// 1
let url = "https://itunes.apple.com/search?term=\(termToSearchFor)"
// 2
let body: [String: Any] = [
  "resultCount": 1,
  "results": [
    [
      "artistName": "Phish",
      "trackName": "Farmhouse",
      "releaseDate": "2000-05-16T07:00:00Z"
    ]
  ]
]
// 3
stub(http(.get, uri: url), json(body))
```

Here's what's happening in this brief code snippet:

1. Specify the url that will be stubbed. In this case, it's the iTunes Search API url with a parameter for the term to search for ("Phish").
2. Designate the hard coded JSON response that Mockingjay should return when the url is accessed.
3. Use Mockingjay to actually stub GET requests to the provided url, and to return a JSON representation of the hard coded body.

Run your tests, they should all pass. That's great, but it's also important to periodically verify that you test is still actually testing what you think it's testing. In the `body` dictionary, change the value for `artistName` to **Grateful Dead** and re-run your tests. They should fail, and that's a good thing because later in your tests, you are making an assertion that the first result is from Phish, not the Grateful Dead.



Revert the stubbed JSON back to the original value of **Phish**.

Congratulations, you just detached the remote iTunes Search API from your unit tests, all the while still maintaining all the additional quality control that unit tests provide. Since you are no longer using the remote web service, your tests will be fast and reliable - exactly the traits you need from unit tests.

Now that you have experienced the first use of Mockingjay to stub an "expected" or "happy case" response from an API, next you'll try out some other uses of Mockingjay to expand the error handling of `QueryService` using test driven development.

3) Expand the error handling

When you originally wrote the code for `getSearchResults(term:completion:)` in `QueryService`, error handling wasn't really considered. That's okay, it was your first jump into test driven development and you hadn't even heard of Mockingjay! Now that you have some experience under your belt, it's time to use Mockingjay to expand the error handling of `getSearchResults(term:completion:)` using test driven development.

Just like usual, it all starts with the tests. But first, take a look at a piece of `getSearchResults(term:completion:)`:

```
let dataTask = URLSession(configuration: .default).dataTask(with: url) {
  [weak self] (data, response, error) in

  guard let strongSelf = self,
    let data = data else {
```

```
      return
    }
    let tunes = strongSelf.createTunesFrom(data: data)
    completion(tunes)
  }
```

This is the code that actually makes the HTTP request and processes the response. Right now there aren't really any good considerations of what to do if anything went wrong with that request. Consider for a moment, what would you like to see this code do if an HTTP 500 response was returned with no data? Right now, the method would return without even calling the completion hander. That's not good. Conceptually, a step in the right direction towards better error handling would be to call the completion handler with `nil` if either there is no data, or there is an `error`. You'll now make that change using test driven development.

Create a Failing Test

It's time to create a failing test. In `QueryServiceTests` add:

```
  func testQueryService_ReturnsNil_WhenError() {

  }
```

Run your tests. No failures yet, but this is a great reminder of how tests can be used as documentation. Just this test's name alone conveys what should happen in a certain case.

Next, add code to `testQueryService_ReturnsNil_WhenError()` to match this:

```
  func testQueryService_ReturnsNil_WhenError() {
    let toTest = QueryService()
    let termToSearchFor = "phish"

    let url = "https://itunes.apple.com/search?term=\(termToSearchFor)"

    // 1
    let stubbedError = NSError(domain: "Fake Error",
                               code: 0,
                               userInfo: nil)
    stub(http(.get, uri: url), failure(stubbedError))

    let expectation = XCTestExpectation(description: "Call web service")

    toTest.getSearchResults(searchTerm: termToSearchFor) { tunes in
      // 2
      XCTAssertNil(tunes)
      expectation.fulfill()
    }

    wait(for: [expectation], timeout: 2.0)
  }
```

This test is very similar to the last one that you wrote. The important differences

are:

1. Instead of returning a fake JSON response for the stubbed web service, fake an error.
2. Rather than verify data in the response, assert that the result is `nil`.

Run your tests. You'll see that this test fails. Awesome! A failing test! Now you'll go write some code to make the test pass!

Make the Test Pass

Reviewing `QueryService`, something should jump out at you about the method signature for `getSearchResults(term:completion:)`:

```
func getSearchResults(searchTerm: String,
                      completion: @escaping ([Tune]) -> ())
```

There's no way that a `nil` value will ever be passed to the completion handler, as the parameter type is not defined as an optional!

To change that, you need to update with the method signature to allow for an optional array to be passed to the completion handler:

```
func getSearchResults(searchTerm: String,
                      completion: @escaping ([Tune]?) -> ()) {
```

Re-run your tests. The build will fail, and it's an interesting failure. The compilation failure will be back in `testQueryService_AccessesAPI_AndReturnsCorrectResult()` - the last test you wrote. Remember, a compilation error is as good as a failing test, so it's time to switch gears and go make the compiler happy.

Back in `testQueryService_AccessesAPI_AndReturnsCorrectResult()` in `QueryServiceTests` , change:

```
XCTAssertEqual(tunes[0].artist, "Phish")
```

to

```
XCTAssertEqual(tunes?[0].artist, "Phish")
```

This accounts for the possibility that `tunes` is now an optional and could be `nil`. Run your tests. This time everything will compile fine, and `testQueryService_AccessesAPI_AndReturnsCorrectResult()` will pass, but `testQueryService_ReturnsNil_WhenError()` still fails. Time to fix that!

Now you'll continue to update `getSearchResults(term:completion:)` to actually pass `nil` to the completion handler when an error happens. Update the definition of `dataTask` to be:

```
  let dataTask = URLSession(configuration: .default)
                   .dataTask(with: url) {
    [weak self] (data, response, error) in

    // The "Important Part"
    if let _ = error {
      completion(nil)
      return
    }

    guard let strongSelf = self,
      let data = data else {
      return
    }
    let tunes = strongSelf.createTunesFrom(data: data)
    completion(tunes)
  }
```

The *Important Part* represents the new addition to this code. It's a simple check for the presence of `error`, and if it's there, call the completion handler with `nil` - just as intended. Re-run your tests. The tests pass!

Woohoo! You just used test driven development to expand the error handling of your web service connectivity code. Not only is your code more flexible for when things go wrong, but you also have unit tests to easily verify the desired behavior.

4) That's it!

Nice work! You've further expanded your experience with test driven development by enhancing your model layer with code that connects to a remote web service. First, you got to experience the speed at which you could write the code to call the web service by starting with tests. You also got familiar with the best practices that tests should execute both extremely fast, and in a perfectly reliable way and how stubbing an HTTP response can provide this consistency. And finally, you expanded your error handling through test driven development.

You almost have a complete app! The only thing left is to build out the user interface. That comes next!

30 Improving App Quality with Test Driven Development: Demo 3

By Andy Obusek

In this demo, you'll learn how to add user interface code to your app with TDD. First, you'll see how to connect your user interface to your model, and then you'll see how to add UIKit elements to your user interface using TDD.

The steps here will be explained in the demo, but here's the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo3\starter**.

1) Connect The Model With the View

For this demo, be sure to open the workspace file, **HalfTunes.xcworkspace** as the project relies on Cocoapod dependencies. In this section, you'll use test driven development to add code to your view controller that uses `QueryService` to search the iTunes API.

The goal of this section is to connect `TunesTableViewController` with `QueryService`. The idea is for the view layer to interact with the model layer to retrieve a list of `Tunes` based on search criteria, and provide it to the view controller. In order to figure out where to start, take a look at the method signature of the search method in `QueryService`:

```
func getSearchResults(searchTerm: String,
                      completion: @escaping ([Tune]?) -> ())
```

The method takes two parameters:

1. A term to use with search.
2. A completion handler that provides a list of `Tunes` as the result.

An empty implementation of `search(searchTerm:)` has already been added to `TunesTableViewController` - that's where you will add code to actually do the search!

Write a Failing Test

Well, would you have started with anything different? ;]

There isn't even a test case class for `TunesTableViewController` yet. From the **File** menu, select **New \ File...**.

Then select **Unit Test Case Class** and click **Next**.

Name the class, **TunesTableViewControllerTests** and click **Next**.

Ensure the file is only added to the **HalfTunesTests** target and click **Create**.

Delete all the boilerplate code from the new class, and add the main target's `import` so it looks like the following:

```
import XCTest
@testable import HalfTunes

class TunesTableViewControllerTests: XCTestCase {

}
```

Now you're ready to add your first failing test for `TunesTableViewController`!

Add the following test:

```
func testSearch_UpdatesTable_UponSuccessfulSearch() {

}
```

As the test name conveys, this test will validate a search method will actually do a search and update the table in the user interface when the results are returned. An empty test is still a passing test. You can verify that by pressing **Command-U**. Additional code is needed before the test will fail.

Update your test to look like the following:

```
func testSearch_UpdatesTable_UponSuccessfulSearch() {
  let toTest = TunesTableViewController()
  let searchTerm = "Taylor Swift"
  toTest.search(searchTerm: searchTerm)
}
```

`QueryService` is a class that's main purpose is to call a remote API via the Internet. When you wrote automated tests for `QueryService` in the last demo, you stubbed the HTTP responses in order to provide repeatability in the tests. Now you need to use `QueryService` from tests again, the same considerations apply. While you could

use HTTP endpoint stubs again, you won't. For these tests, for functionality on the view controller, stubbing HTTP responses is a low level implementation detail that is better off kept hidden behind `QueryService`. Instead, you'll use dependency injection to provide a fake `QueryService` for use within `TunesTableViewController`. The fake `QueryService` will be coded to behave in a deterministic manner that will enable test repeatability.

Dependency Injection is an extremely useful tool for testing your apps. It's not just for testing though. More broadly it can be used to help write cleaner code where classes are less tightly coupled to each other. James Shore has a great quote about dependency injection, "Dependency Injection is a 25-dollar term for a 5-cent concept...Dependency injection means giving an object its instance variables. Really. That's it." You'll see, it's extremely simple, and extremely useful.

At the beginning of `testSearch_UpdatesTable_UponSuccessfulSearch()` add the following:

```
class MockQueryService: QueryService {
  let tunes = [Tune(artist: "Taylor Swift",
                    name: "Shake It Off",
                    releaseDate: Date())]

  override func getSearchResults(searchTerm: String,
                                 completion: @escaping ([Tune]?) -> ()) {
    completion(tunes)
  }
}
```

You're subclassing `QueryService` and overriding the important method to return a hard coded list of a single, awesome, `Tune`. This is the alternative discussed earlier to using Mockingjay and mocking out the HTTP endpoint.

Right after the creation of `toTest`, add the following:

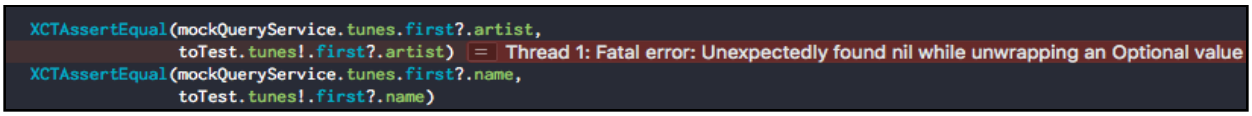
```
let mockQueryService = MockQueryService()
```

This creates the `MockQueryService` so it's ready for use.

Next add the following to the end of `testSearch_UpdatesTable_UponSuccessfulSearch()`

```
XCTAssertEqual(mockQueryService.tunes.first?.artist,
               toTest.tunes!.first?.artist)
XCTAssertEqual(mockQueryService.tunes.first?.name,
               toTest.tunes!.first?.name)
```

These two lines verify the first `Tune` in the list of the `Tunes` on `TunesTableViewController` matches the `Tune` returned by the `MockQueryService`. Run your tests. A test failure!



Time to fix it!

Make the Test Pass

The design of `TunesTableViewController` is such that a property of type `[Tune]` will provide data to the the `UITableView` on screen.

Inside the class definition of `TunesTableViewController`, you'll find:

```
var tunes: [Tune]? = nil
var queryService: QueryService? = nil
```

In addition to `tunes`, you'll see `queryService`. This provides a hook for you to tell `TunesTableViewController` to use the `MockQueryService` setup earlier in the test.

Next, update `search(searchTerm:)` in `TunesTableViewController` to look like the following:

```
queryService?.getSearchResults(searchTerm: searchTerm) {
  [weak self] tunes in

  self?.tunes = tunes
  DispatchQueue.main.async {
    self?.tableView.reloadData()
  }
}
```

Finally, open **TunesTableViewControllerTests.swift**, and find the following line:

```
let mockQueryService = MockQueryService()
```

Right after the above, add the following:

```
toTest.queryService = mockQueryService
```

This hooks up the `MockQueryService` to be used from `TunesTableViewController`.

This is the **magic** of dependency injection. It's simple right? You're just providing a dependency (`MockQueryService`) to a class, `TuneTableViewController`. The power is wonderful though. Now you can easily swap out any other implementation, as long as it matches the type `QueryService`. The "production" version of our app will use the real `QueryService` which calls the real iTunes API. Our tests use `MockQueryService` which has hard coded results. Just imagine, you could also introduce alternatives that call Spotify, Pandora, Amazon Music, etc.

Bonus: You could make this design even more powerful by introducing a protocol to define the methods that a `QueryService` should have. That would then remove

the need to subclass the concrete implementation of `QueryService` in order to swap out dependencies. That exerise is left to the reader. ;]

Finally, run your tests. Tests pass, woooohoo!

Congratulations, you just used test driven development with dependency injection to connect your view layer with a model layer.

In this section, you learned a special trick for subclassing an object in order to override a method in order to:

1. Prevent the original method from firing.
2. Set a flag that provides a hook to verify the method was called.

This is powerful stuff!

Note: There's a slight flaw in this test implementation, can you spot it? The asynchronous manner in which `tunes` are assigned inside the callback from `getSearchResults(searchTerm:completion:)` could be slightly delayed such that the test assertions happen before the assignment happens. Further refactoring is needed to correct this, but for the sake of the demo, this is left as such.

2) Verifying More Complex System Behavior

Up to this point, you've connected your view layer with your model layer to show data in your app. Now you're going to add some functionality to your app that relies on interacting with system owned resources. In this case, you'll use `UIApplication` to open the iTunes URL associated with the track when the user taps a table view cell. There's also plenty of other cases the pattern you are about to learn will apply. Code like this can be hard to test because:

1. You don't own the object under test (`UIApplication`)
2. The object under test is a singleton
3. The object under test does stuff outside of the context of your application (switching to Safari)

So if simply testing code like this is hard, how do you even begin to think about doing test driven development? Stick around and you'll find out!

Write A Failing Test

Open **TunesTableViewControllerTests.swift**. Add the following code within the class:

```
// 1
func testDidSelectRow_OpensTrackURL_WhenSelected() {
```

```
    // 2
    let toTest = TunesTableViewController()
    let aTune = Tune(artist: "Taylor Swift",
                     name: "Blank Space",
                     releaseDate: Date(),
                     trackViewUrl: "http://www.taylorswift.com")
    toTest.tunes = [aTune]

    // 3
    toTest.tableView(toTest.tableView,
                     didSelectRowAt: IndexPath(item: 0, section: 0))

  }
```

A couple of notes about this piece of code:

1. As the test name suggests, this test will verify the behavior you're setting out to write: selecting a row in the table view opens the track's url when tapped.
2. Set up your object under test, the `TunesTableViewController` with a single `Tune` in the list of `tunes`.
3. Call the method under test `tableView(_:didSelectRowAt:)`

There is an empty implementation of `tableView(_:didSelectRowAt:)` in `TunesTableViewController`. This is where you'll add behavior associated with what happens when someone taps a row in the `UITableView`. If you'd like, run your tests, they will pass. You still need to add more code to make it fail.

At this point, things get tricky. But get excited, you're about to learn something new! Conceptually zoom out and think about what you're trying to build. In the response from the iTunes Search API, each tune has a url associated with it. The url represents the track's page in the iTunes store. You want your app to open this url in Safari when the user taps the row in the `UITableView`. A common way to implement this is with `UIApplication`'s shared instance using the method `open(_:options:completionHandler:)`. This is the method you'll use. Don't add the code yet, but instead visualize how this will ultimately be implemented:

```
  UIApplication.shared.open(...)
```

You'll still hold true to the commitment of test driven development, and still implement this behavior. The trick is figuring out how ensure it's tested. Additionally, you don't want to allow the "real" `UIApplication` to open the url from your tests because that will send the test runner to the background and cause all sorts of wacky behavior. To do this, you'll combine several techniques you've learned thus far, and add one more new tool to your toolbox.

First, you still need to make the test fail. You'll do this with a mock object to represent `UIApplication`. Add this code to the beginning of `testDidSelectRow_OpensURL_WhenSelected()`:

```
class MockUIApplication: UIApplication {
  var openCalled = false
  override func open(_ url: URL,
                     options: [String: Any],
                     completionHandler: ((Bool) -> Void)?) {
    openCalled = true
  }
}
```

This `MockUIApplication` subclasses `UIApplication` and overrides `open(_:options:completionHandler:)` in order to track and allow you to verify that it's called. This is similar to what you did with `UITableView` earlier in this demo.

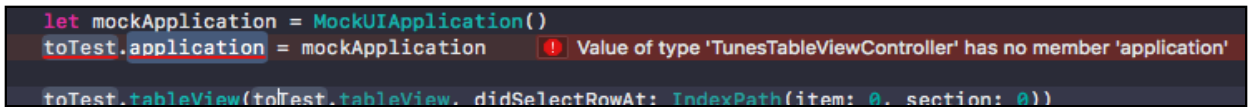
Next, make use of this mock, Find the following line of code in your test:

```
toTest.tunes = [aTune]
```

and add the following just below the above:

```
let mockApplication = MockUIApplication()
toTest.application = mockApplication
```

Run your tests, they'll not compile.



A failure! Yes! Time to make them pass!

Make the Test Pass

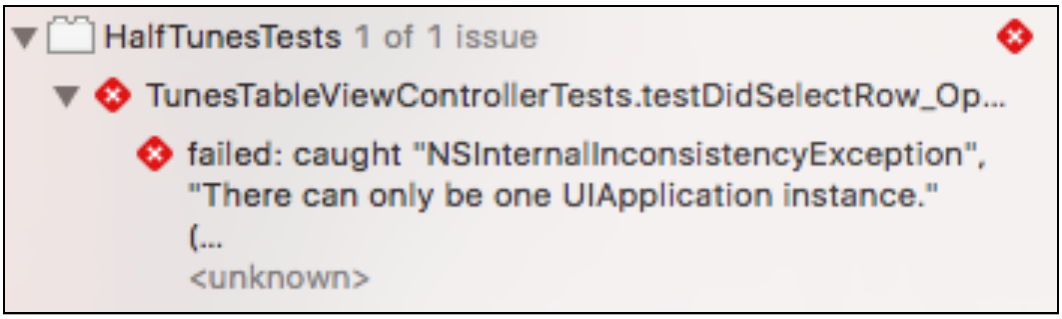
Did you catch the dependency injection in that last piece of code? You created a `MockUIApplication` and attempted to assign it on `TunesTableViewController`. That's where the compilation failure happened, because `TunesTableViewController` wasn't setup yet to accept the dependency. Time to fix that.

At the beginning of `TunesTableViewController`, add the following line of code:

```
var application: UIApplication = UIApplication.shared
```

This declaration identifies the dependency on `UIApplication` that can be used through `TunesTableViewController`. Additionally, it allows for a point where the dependency can be injected from other points, like your tests.

Run your tests, they will compile, but fail.



"There can only be one..." Hmm, where have you heard that before? ;]

Similar to the Highlander, there can only be one instance of `UIApplication` in the system at a time. This runtime test failure indicates just that. This happened because the mock object you created subclassed `UIApplication`, effectively creating a second instance. iOS won't allow that.

To work around this limitation, you'll use another magic trick brought to you by Swift protocols. Add this code to the end of **TunesTableViewController.swift**

```
protocol URLOpening {
  func open(_ url: URL,
            options: [String: Any],
            completionHandler: ((Bool) -> Void)?)
}
```

This defines a new protocol `URLOpening` with a single method: `open(_:options:completionHandler:)`. Notice this method signature is identical to the one that you ultimately would like to use on `UIApplication`. Since the method signatures line up exactly, you can mark `UIApplication` as conforming to this protocol. Add the following code, below the protocol you just added, so it looks like the following:

```
// THIS ALREADY EXISTS
protocol URLOpening {
  func open(_ url: URL, options: [String : Any], completionHandler:
((Bool) -> Void)?)
}

// ADD THIS!
extension UIApplication: URLOpening {
}
```

Finally, go back to the definition of the dependency at the top of `TunesTableViewController`. Find this line:

```
var application: UIApplication = UIApplication.shared
```

And change it to the following:

```
var application: URLOpening = UIApplication.shared
```

Did that feel like magic? It should have. You just defined a custom protocol, and declared a system object conforms to it.

Now you need to do the same with your `MockUIApplication`. Back in `testDidSelectRow_OpensURL_WhenSelected()`, find the definition of `MockUIApplication`:

```
class MockUIApplication: UIApplication {
```

And change it to the following:

```
class MockUIApplication: URLOpening {
```

Also delete the `override` keyword on the declaration of `open(_:options:completionHandler:)`. `MockUIApplication` should look like the following:

```
class MockUIApplication: URLOpening {
  var openCalled = false
  func open(_ url: URL,
            options: [String: Any],
            completionHandler: ((Bool) -> Void)?) {
    openCalled = true
  }
}
```

Run your tests. They'll pass. Woohoo! You've totally abstracted the dependency of `UIApplication` from `TunesTableViewController` such that a mock could be used in its place. And since the tests pass, it's time to make them fail!

Make the Tests Fail

Your tests don't actually assert anything at this point, nor does `tableView(_:didSelectRowAt:)` do anything. Time to change that!

Add this line to the end of `testDidSelectRow_OpensURL_WhenSelected()`:

```
XCTAssertTrue(mockApplication.openCalled)
```

This asserts that `open(_:options:completionHandler:)` is actually called when a row is selected on the `UITableView`. Run your tests. The test will fail. Perfect! That's because there is no code in `tableView(_:didSelectRowAt:)`. Time to change that and make the tests pass, the final step.

Make the Tests Pass

Add this to `tableView(_:didSelectRowAt:)` in `TunesTableViewController`:

```
guard let tunes = tunes else {
  return
```

```
  }

  let selectedTune  = tunes[indexPath.row]
  guard let trackURL = URL(string: selectedTune.trackViewUrl) else {
    return
  }

  application.open(trackURL, options: [:], completionHandler: nil)
```

This code handles when a row is selected by grabbing the `Tune` at that index, and opening it's `trackURL`. Run your tests, they will pass! Awesome. You can also even run your app and try it yourself! (Note: If you attempt to select a row in the Simulator, you may see some odd behavior when the url is opened because Safari will attempt to redirect to the iTunes app which doesn't exist.)

4) That's it!

Nice job! Throughout this demo, you've learned how to use test driven development to :

1. Implement `UITableView` behavior.
2. Connect model objects with the view.
3. Add complex system behavior with objects you don't own.

These are all techniques that prove invaluable when using test driven development in your code. At this time you have a wide range of different techniques to apply during test driven development in the user interface layer of your iOS application! Go forth, and never look back to the dark days of untested code.