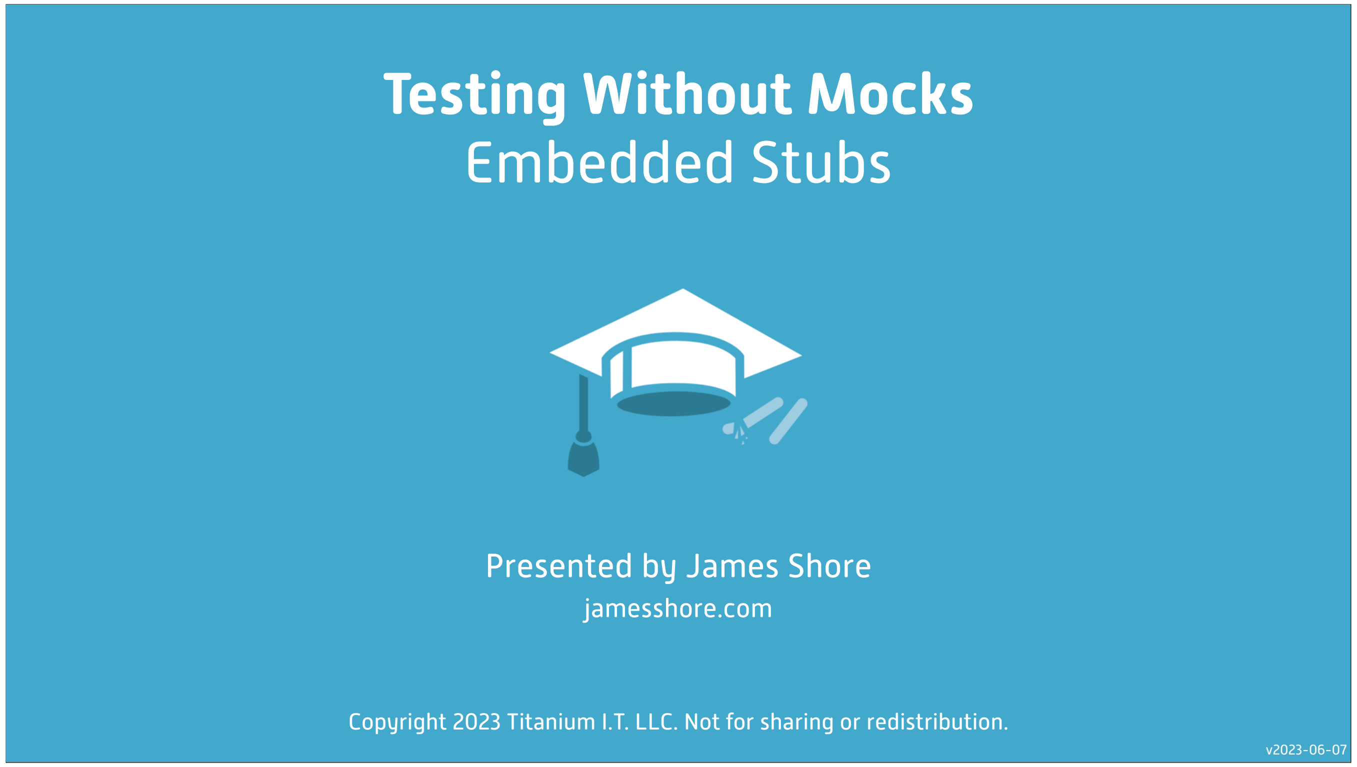


Testing Without Mocks

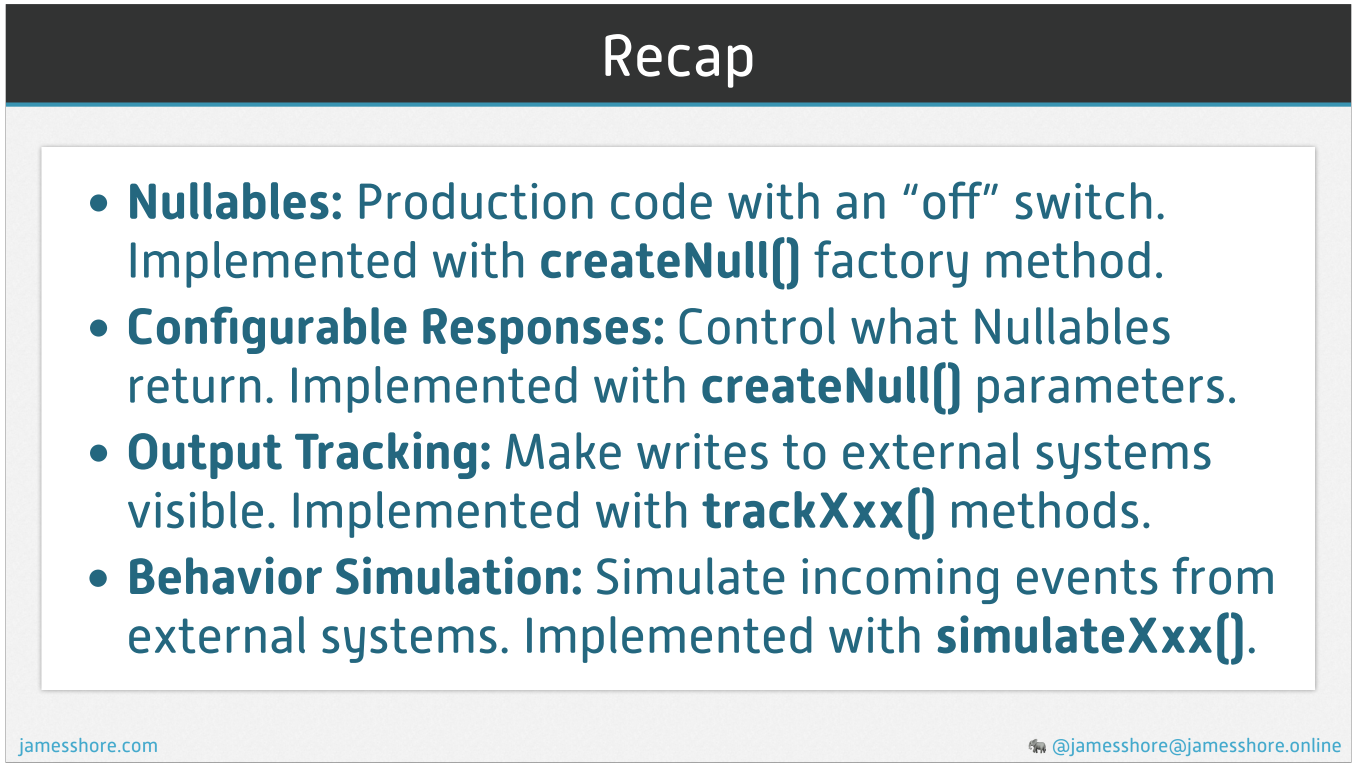
Embedded Stubs

Presented by James Shore
jamesshore.com

v2023-06-07

I'm James Shore. Today in “Testing Without Mocks,” we're talking about how to make low-level infrastructure wrappers Nullable.

As a reminder, watching this video is optional. I'll cover the same material during the course.



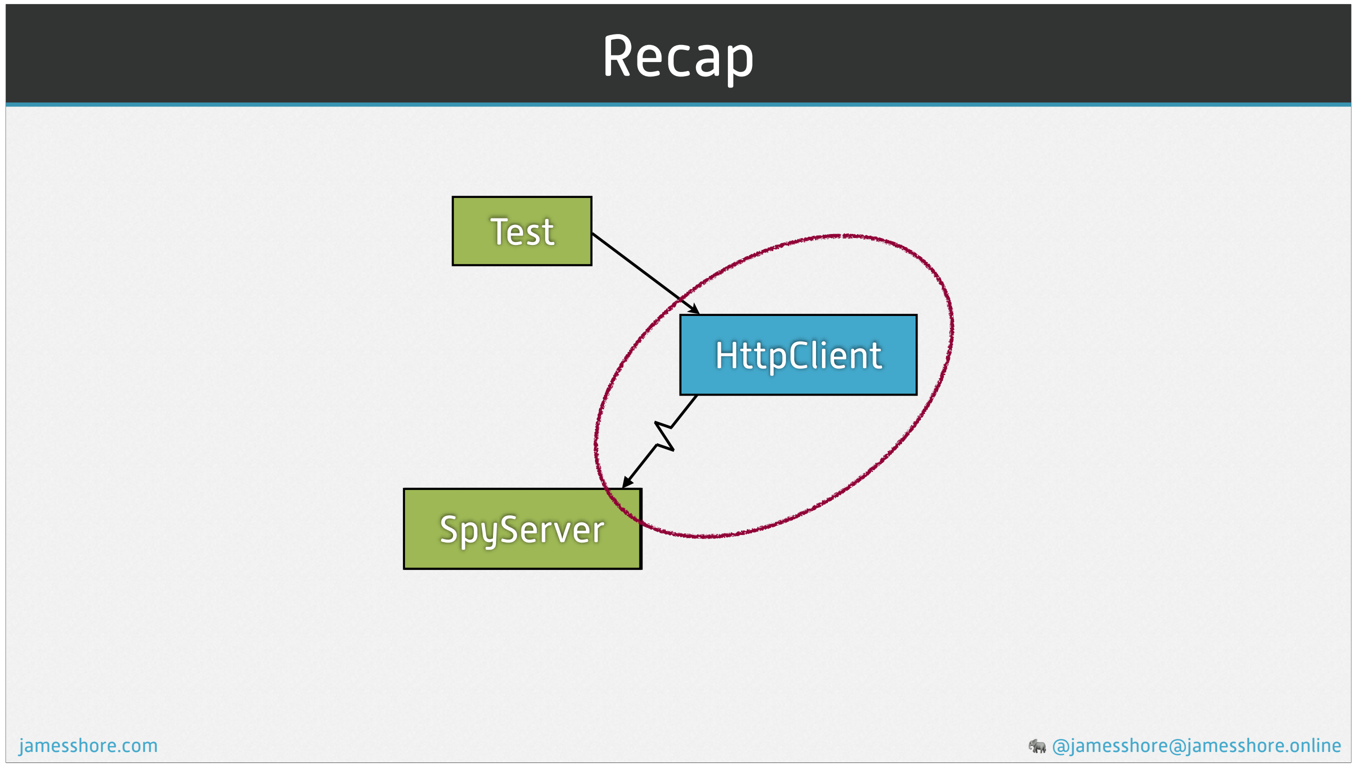
Recap

- **Nullables:** Production code with an “off” switch. Implemented with **createNull()** factory method.
- **Configurable Responses:** Control what Nullables return. Implemented with **createNull()** parameters.
- **Output Tracking:** Make writes to external systems visible. Implemented with **trackXxx()** methods.
- **Behavior Simulation:** Simulate incoming events from external systems. Implemented with **simulateXxx()**.

To recap, we’ve been talking about a collection of patterns for writing **sociable, state-based tests** rather than mock-based tests, which are **solitary, interaction-based tests**. These patterns are useful because solitary tests pass when they should fail, and interaction-based tests fail when they should pass.

There are four core patterns:

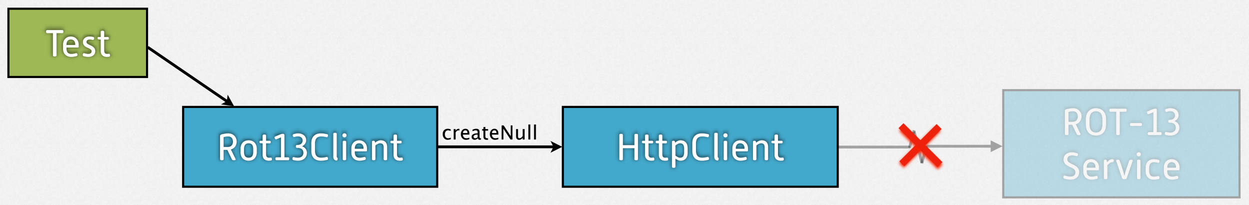
1) **Nullables**, which are production code with an “off” switch. They can be configured to disable communication with the outside world by calling the “createNull()” factory method.
2) Configurable Responses, which is a way of controlling what Nullables return.
3) Output Tracking, which a way of tracking calls to external systems.
4) Behavior Simulation, which is a way of simulating events that come from external systems.



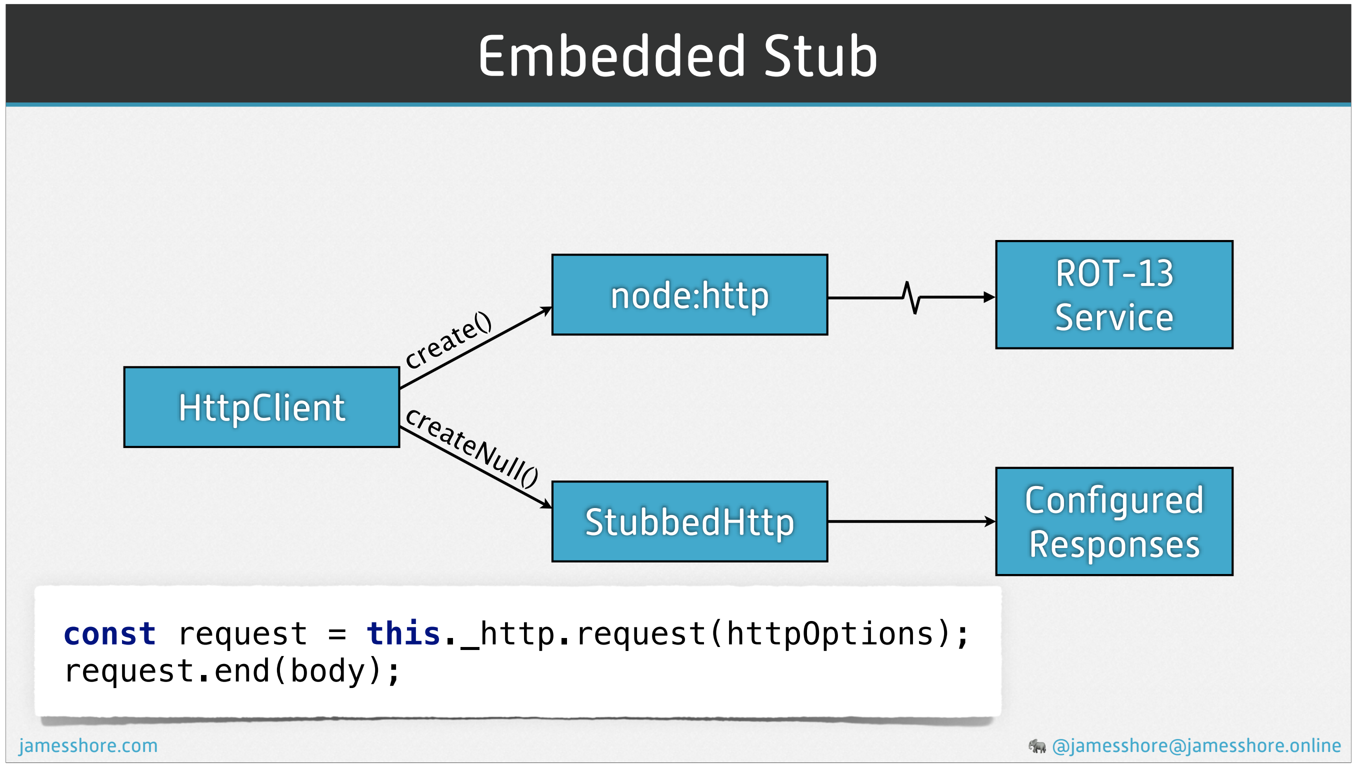
Recap

Last time, you learned how to implement and test HttpClient, a low-level infrastructure wrapper. This time, you'll learn how to make it Nullable.

Turn Off External Communication



As I’ve mentioned, Nullables are production code that can "turn off" communication with external systems. They do this with an **Embedded Stub**. (Or sometimes, an embedded fake, but stubs are much simpler.)



Embedded Stub

```
const request = this._http.request(httpOptions);
request.end(body);
```

jamesshore.com

@jamesshore@jamesshore.online

A “stub” is a replacement for a class. Normally, HttpClient uses node’s built-in http module to talk to the ROT-13 service. But when the Nulled instance is in use, HttpClient uses a stubbed-out version of the http module instead.

You might wonder why we stub out the **Node’s** http module rather than **our** HttpClient module. It's a major difference between Nullables and mocks. With mocks, you’re only supposed to mock out code you own. With Nullables, you’re only supposed to stub out code you **don’t** own.

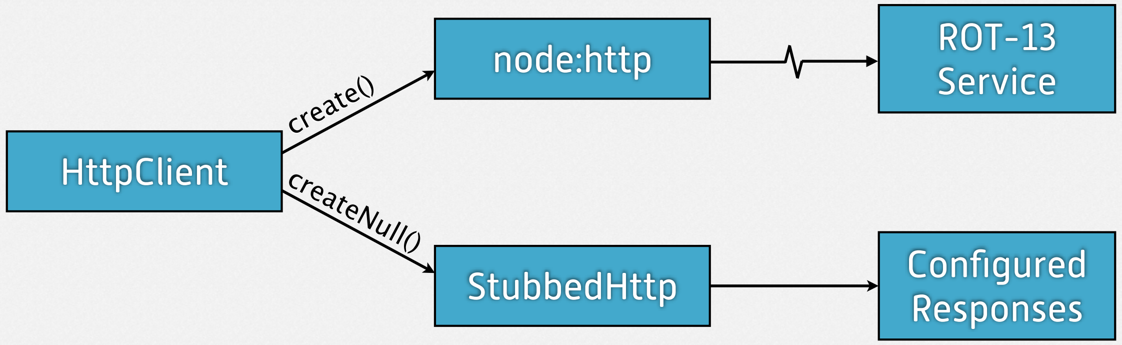
The policy of stubbing out third-party code means HttpClient runs **the same code** regardless of whether it’s Nulled or not. If we make a change to HttpClient that changes the system’s behavior, our tests will catch it.

Embedded Stubs are a bit ugly from a code purity perspective, because they look like a test double, but they're part of your production code. Some people call them a "production double." If it makes you feel better, the embedded stub is tested, just like your other production code, and it can be useful in production. For example, you can use Nullables to implement a "dry run" option in a command line program. I’ve used it to implement cache warming in a web server.

Ultimately, though, engineering is tradeoffs, and this is the tradeoff you’re making when you choose to use Nullables. The benefit is that you have more reliable tests and easier refactoring. The cost is that you have a production double. In practice, I’ve found that embedded stubs in low-level infrastructure wrappers are highly reusable, nicely encapsulated, and lead to high test quality. Whether that’s worth it is up to you.

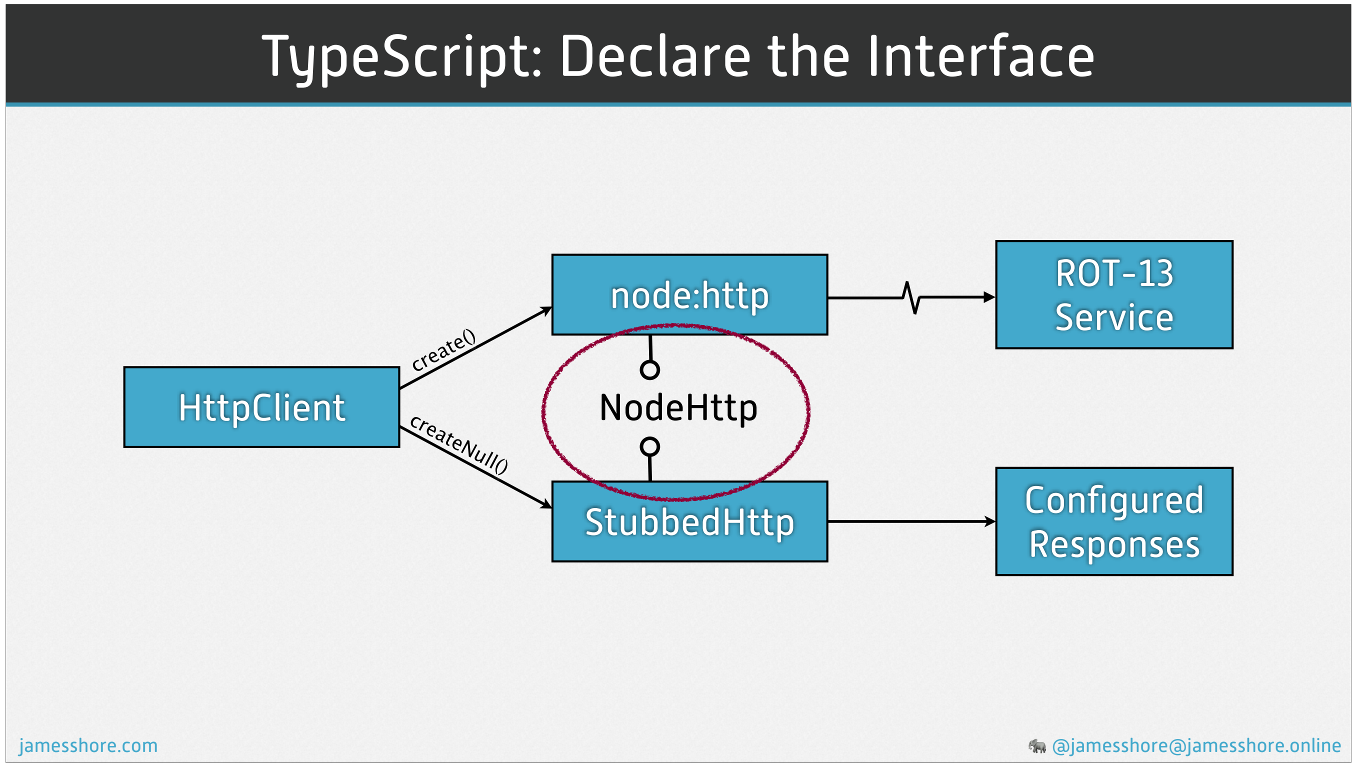
A common reaction is to want to inject the stub, rather than embedding it. However, the stub is **highly coupled** to the implementation of the low-level wrapper, so it's **more cohesive** and **better encapsulated** if it's embedded.

JavaScript: Implement the Interface You Use



In JavaScript and other duck-typed languages, you'll implement the embedded stub by creating a class that has the exact same interface as the third-party code, but only the part your infrastructure wrapper uses. You can test-drive the implementation of the stub by creating a Nulled instance with an empty implementation, calling methods, and gradually adding to the stub as needed.

Don't try to copy the behavior of the real module, and don't implement the whole API. Only the implement the part your code actually uses.



TypeScript: Declare the Interface

HttpClient

create()

createNull()

node:http

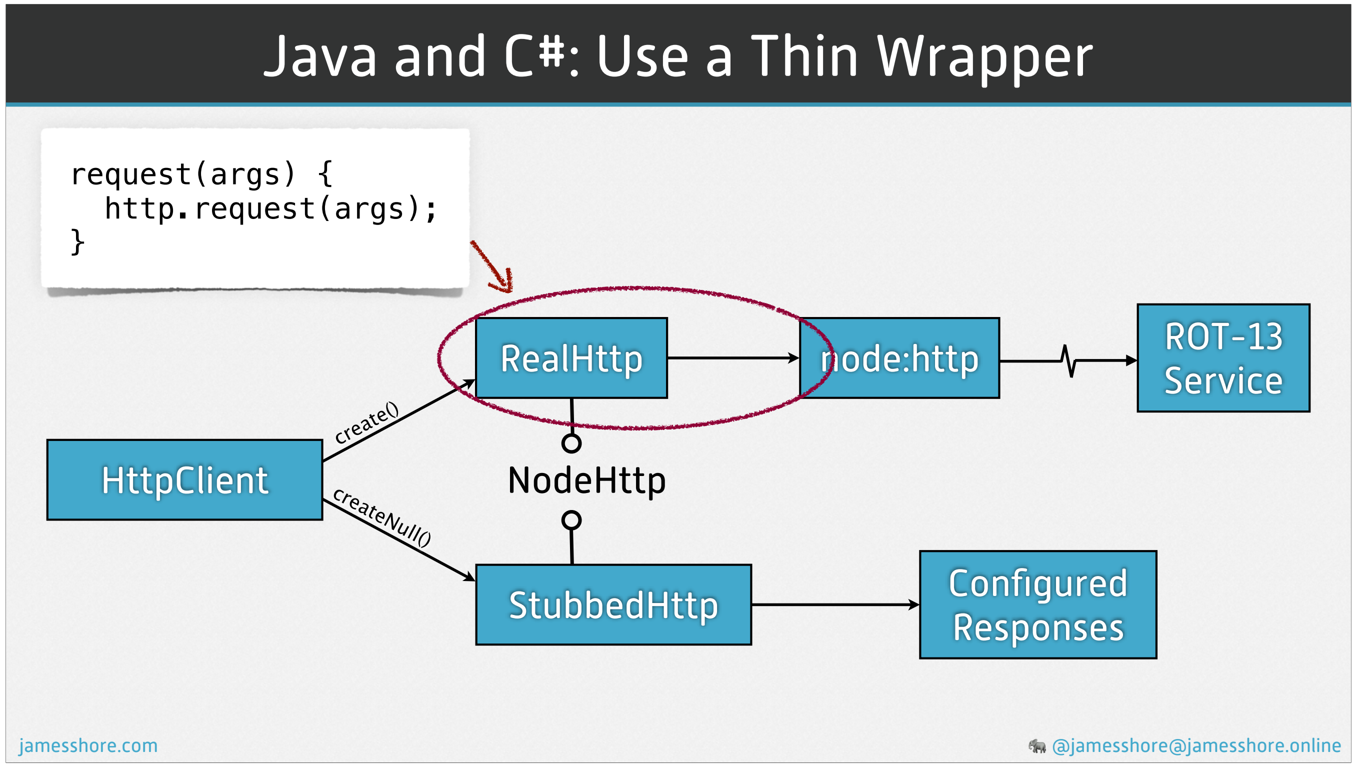
NodeHttp

StubbedHttp

ROT-13 Service

Configured Responses

In languages with structural types, such as TypeScript, do the same thing, but you’ll need to declare the interface.

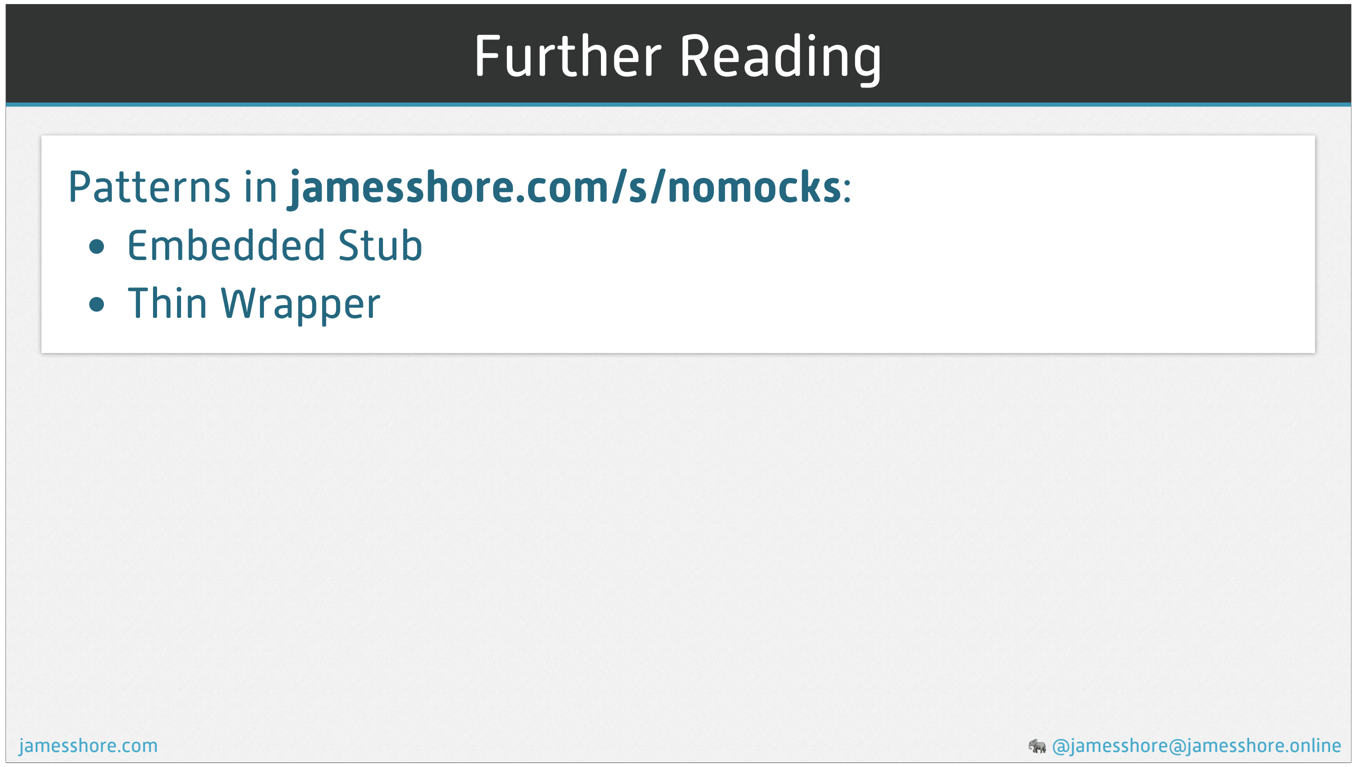


Java and C#: Use a Thin Wrapper

```
request(args) {
  http.request(args);
}
```

Languages with nominal types, such as Java and C#, require you to jump through an extra hoop. As with JavaScript and TypeScript, create an interface that matches **only** what your low-level wrapper uses and have your embedded stub implement it. For the real code, create a thin wrapper that implements the same interface and forwards calls to the real thing.

In some cases, particularly in C#, the real code might have an interface that your stub can implement, but that's usually not a great idea. The interface is typically much bigger than you need. Creating a minimal interface just for your needs will be smaller and simpler.



Further Reading

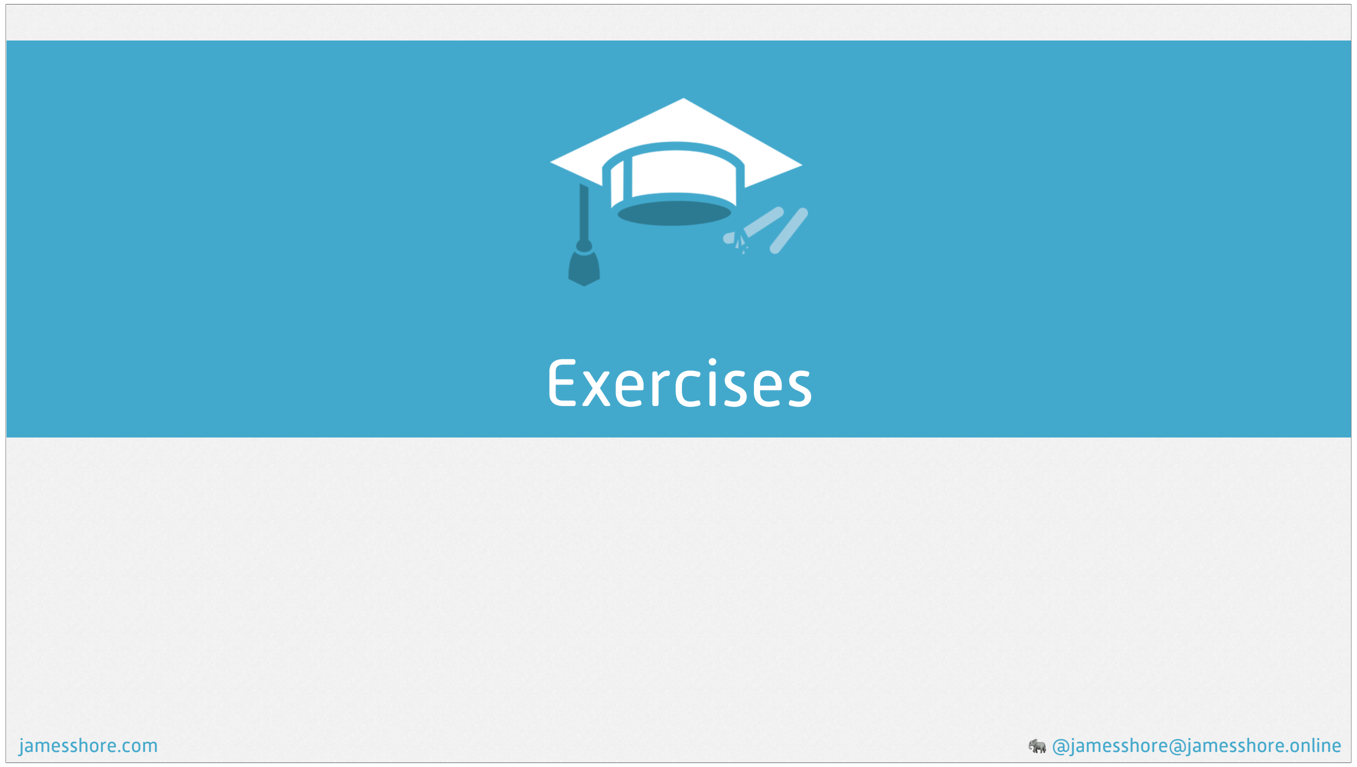
Patterns in **jamesshore.com/s/nomocks**:

- Embedded Stub
- Thin Wrapper

jamesshore.com

@jamesshore@jamesshore.online

As always, you can find the Embedded Stub and Thin Wrapper write-ups in the “Testing Without Mocks” article. But they’ll make more sense once you see them in action.

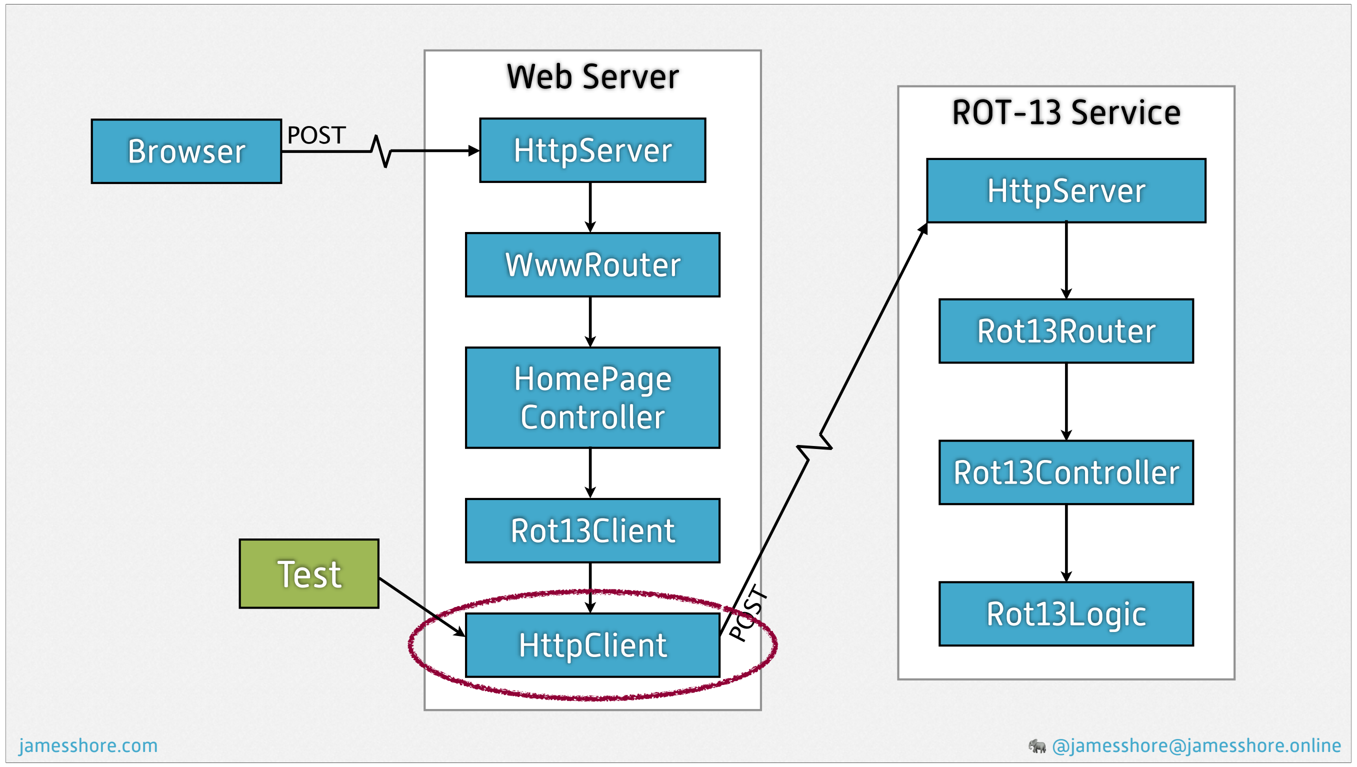


Exercises

jamesshore.com

🐘 @jamesshore@jamesshore.online

So let’s look at the exercises.



Web Server

Browser

POST

HttpServer

WwwRouter

HomePage Controller

Rot13Client

Test

HttpClient

POST

ROT-13 Service

HttpServer

Rot13Router

Rot13Controller

Rot13Logic

jamesshore.com

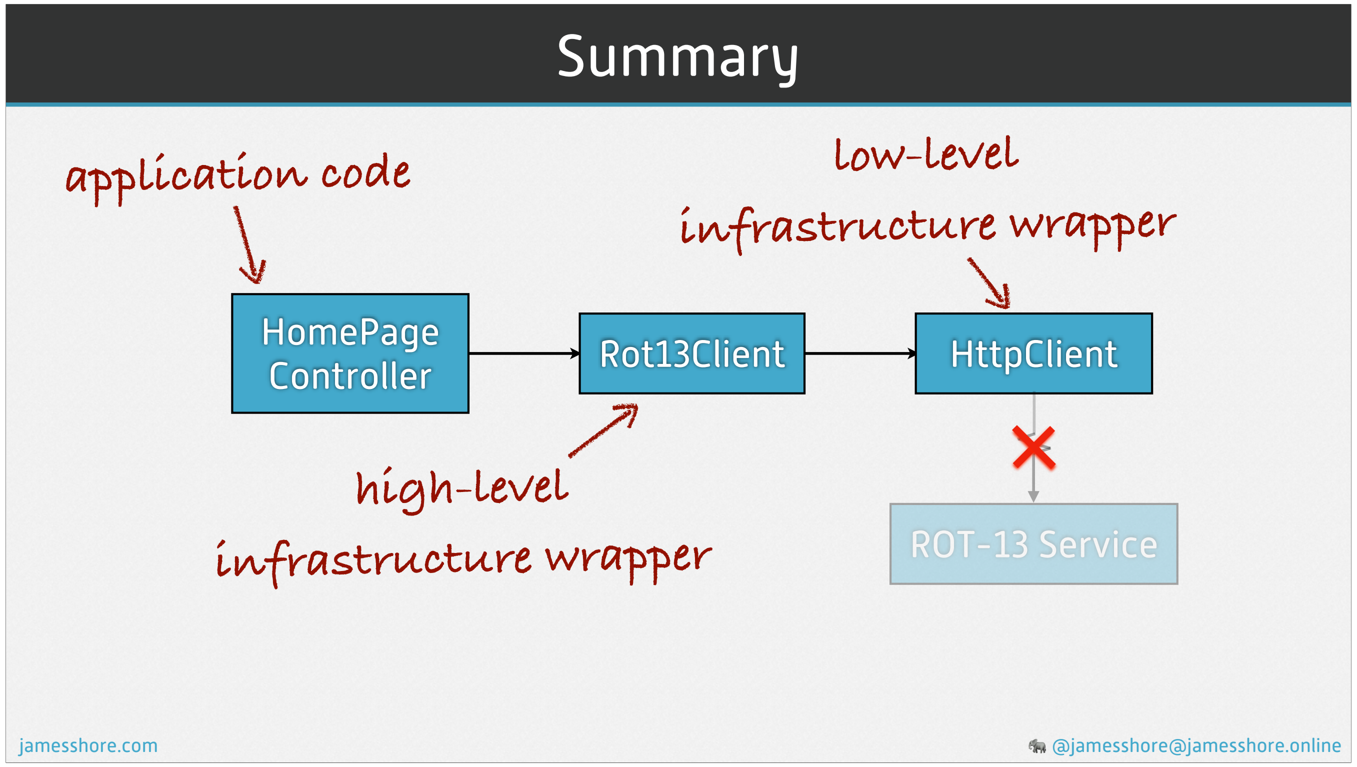
@jamesshore@jamesshore.online

Today, you'll be finishing the work you started last time by adding Nullability to a HttpClient. The exercise starts with the completed HttpClient, so you’ll just be adding the Embedded Stub, Configurable Responses, and Output Tracking. These exercises can be tricky, so be sure to use the hints.

(Walk through exercise setup.)

(Reminder about API docs, primers, and hints.)

(Reminder to use "Ask for help" button.)



Summary

application code

low-level infrastructure wrapper

HomePage Controller

Rot13Client

HttpClient

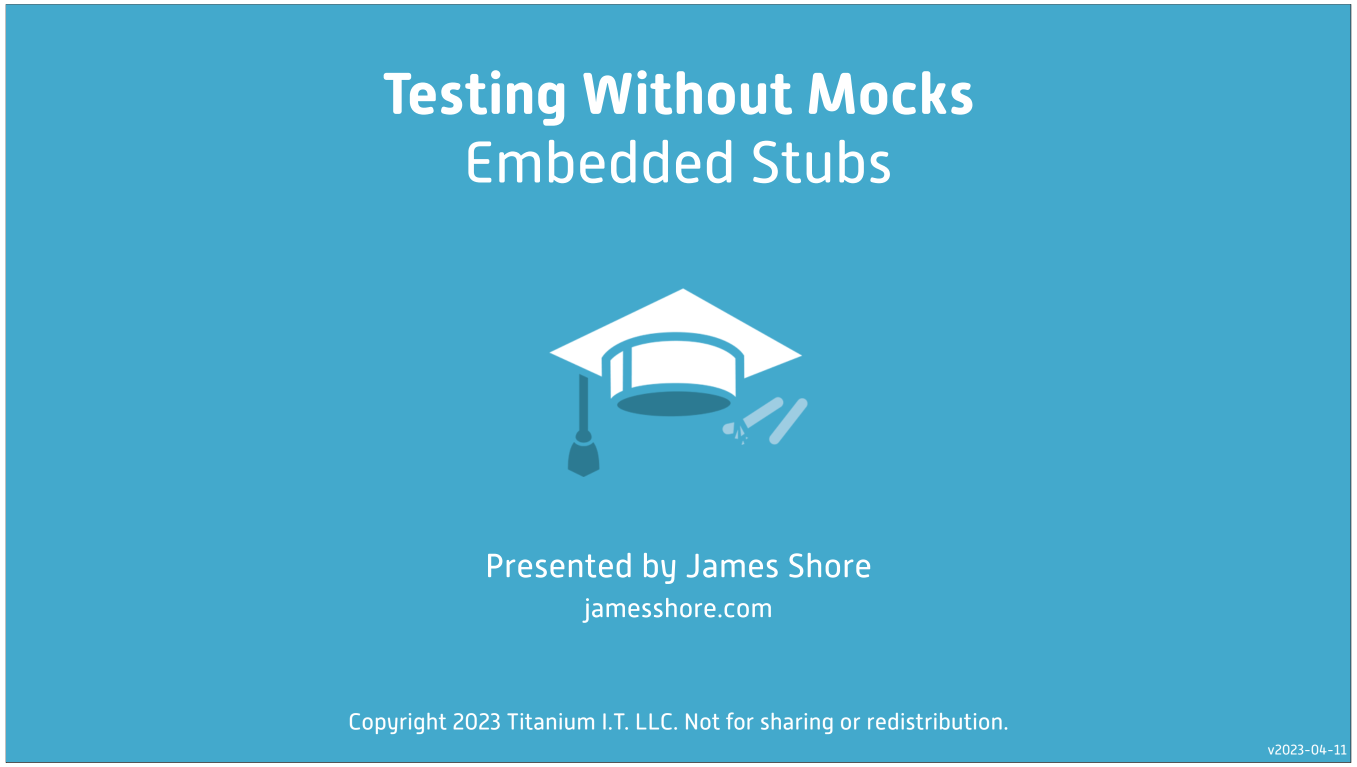
✕

ROT-13 Service

high-level infrastructure wrapper

Once you finish these exercises, you’ll have seen everything you need to start using Nullables in your own code. You’ll use Embedded Stubs to turn off external systems in your low-level infrastructure wrappers. Your high-level infrastructure wrappers will delegate to the low-level wrappers for their tests and to be Nullable themselves. And your application code will take advantage of Nullable infrastructure to be testable itself.

Using Nullables will allow you to create sociable, state-based tests. Your code will be easier to refactor, and your tests will be more reliable.



Testing Without Mocks
Embedded Stubs

Presented by James Shore
jamesshore.com

v2023-04-11

Embedded Stubs complete the picture. Thanks for listening, and good luck.