Testing Without Mocks
High-Level Infrastructure

Presented by James Shore
jamesshore.com

v2023-06-05

I'm James Shore. Today, we're talking about how to implement high-level infrastructure. It's part of my "Testing Without Mocks" course.

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

Definitions

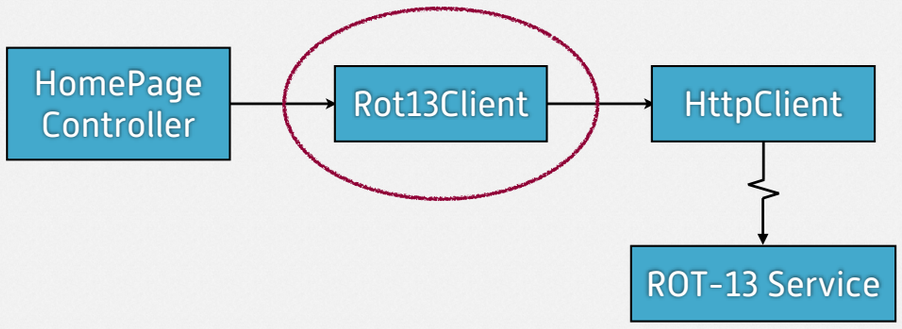
- **Infrastructure:** External systems and state. Databases, services, the system clock, random number generator, etc.
- **Infrastructure Wrapper:** A class or module that provides a clean way of manipulating infrastructure.

The title of this module is “high-level infrastructure,” but technically, it’s about high-level infrastructure **wrappers**. Let’s define some terms.

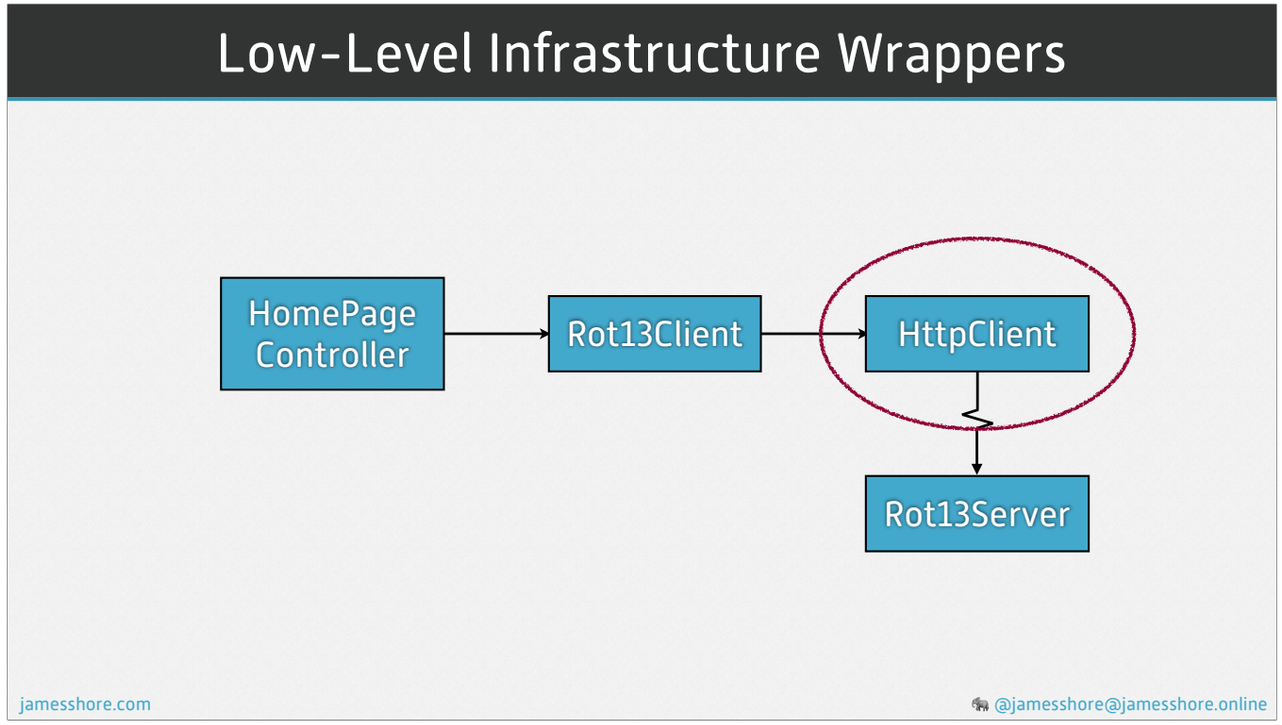
“Infrastructure” is external systems and state. External **systems** are things like databases and services. External **state** is things like the system clock, environment variables, random number generator, and so forth.

That’s infrastructure. An infrastructure **wrapper** is a class or module you write for managing the infrastructure your code cares about.

High-Level Infrastructure Wrappers

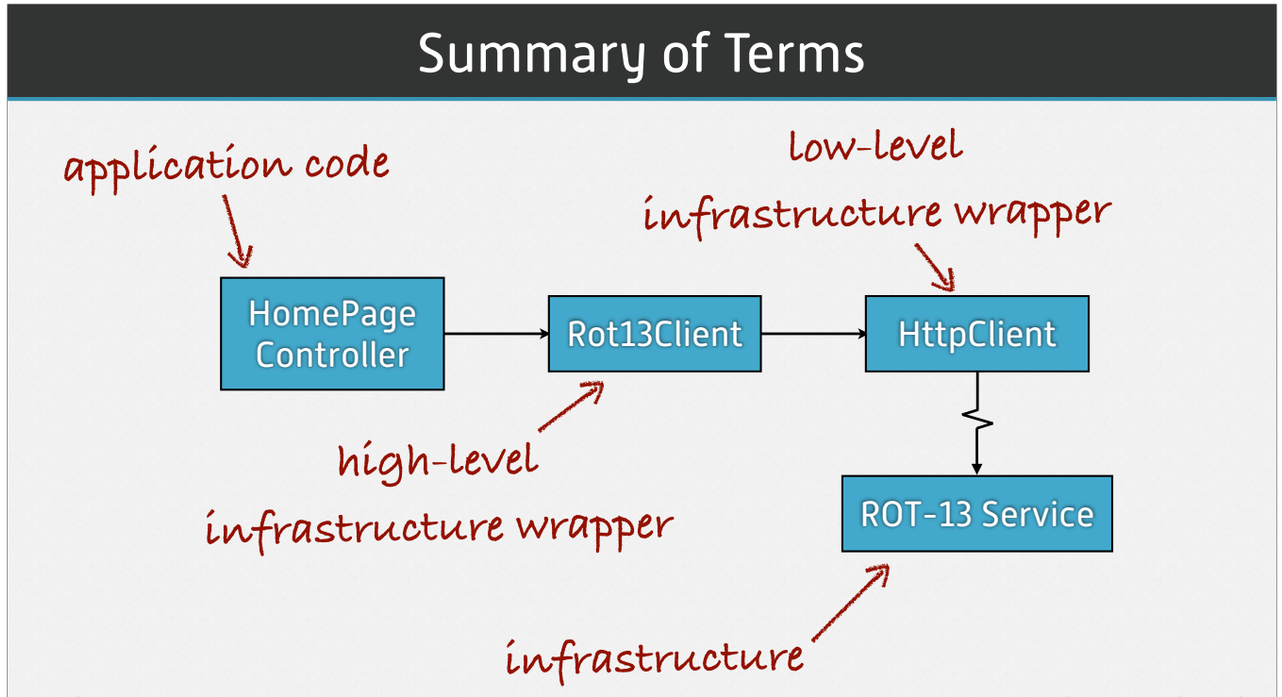


High-level infrastructure wrappers are responsible for providing a clean interface to a specific piece of infrastructure. In our ROT-13 example, the Rot13Client is a high-level wrapper. It's responsible for presenting a clean interface to the ROT-13 service. Specifically, it provides a "transform()" function that uses the ROT-13 service to encode strings.

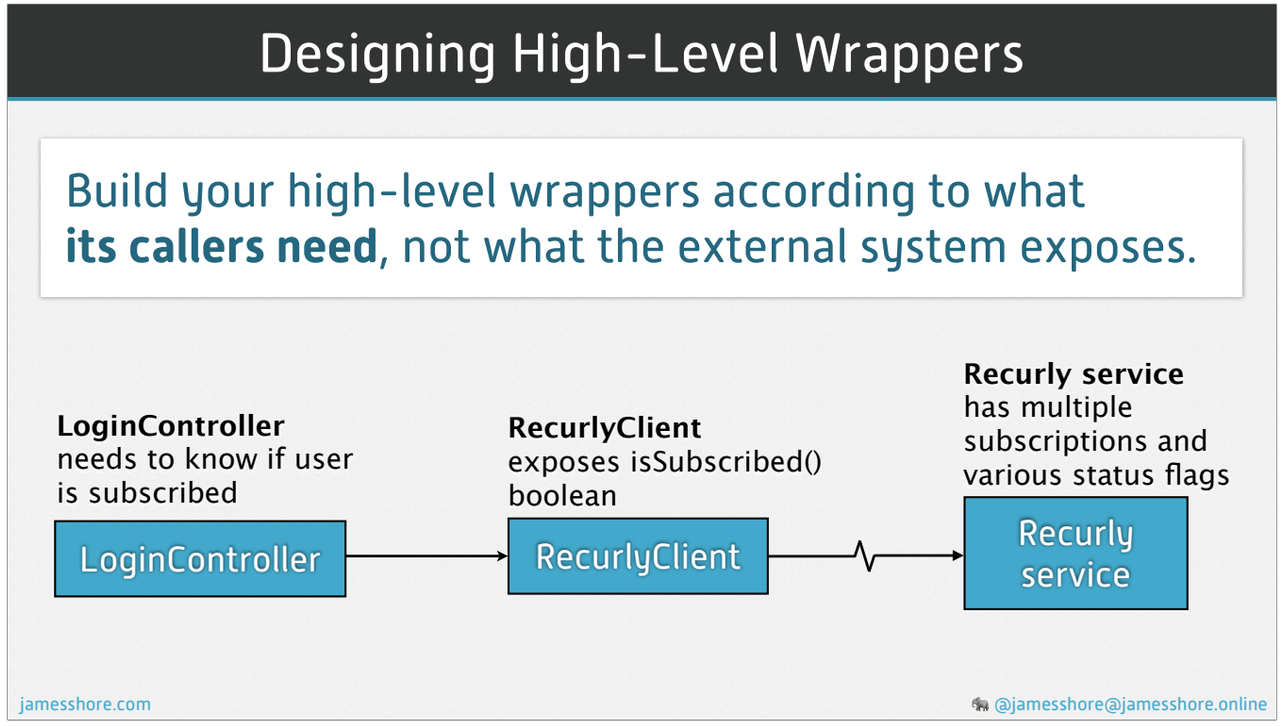


High-level wrappers use low-level infrastructure wrappers to do their dirty work. Low-level wrappers provide a generic interface for some communication technology. For example, you might have an HttpClient for HTTP, a PostgresDatabase for your database, or a WebSocketClient for real-time communication.

To summarize, high-level wrappers abstract a specific **system**, and low-level wrappers abstract the communication **protocol**. You can combine them into a single wrapper, but it's often more convenient and cleaner to keep them separated.



To bring it all together: HomePageController is the **application code**. When it wants to talk to the ROT-13 service, it uses Rot13Client, which is a **high-level infrastructure wrapper**. Rot13Client uses HttpClient, a **low-level infrastructure wrapper**, to do the communication. And the communication is with the ROT-13 service, is the **infrastructure** we're wrapping**.**

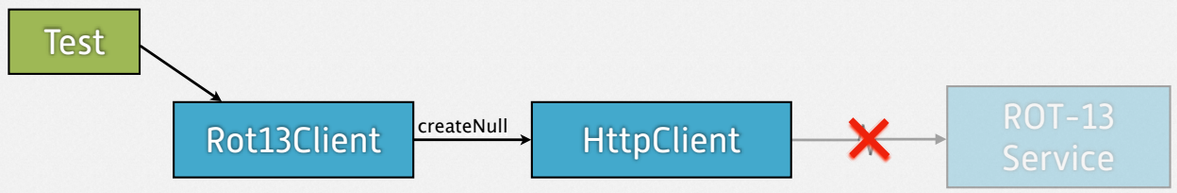


To design a high-level infrastructure wrapper, focus on encapsulation. Your job is to provide the perfect interface to the infrastructure. It should hide all the gory details. Don't imitate the external service's API. Instead, think about what **your application** needs and how to best meet those needs.

For example, back when I ran a subscription screencast, I used a product called Recurly to manage my subscription billing. Recurly had a complicated API with all sorts of flags and settings. But when someone logged in, my LoginController just needed to know one thing: is this person subscribed, or not? So my high-level infrastructure wrapper, RecurlyClient, had an "isSubscribed()" method. It was responsible for calling the Recurly API, looping through all the subscriptions associated with an email address, deciphering all the status flags, and returning a simple "true/false" response.

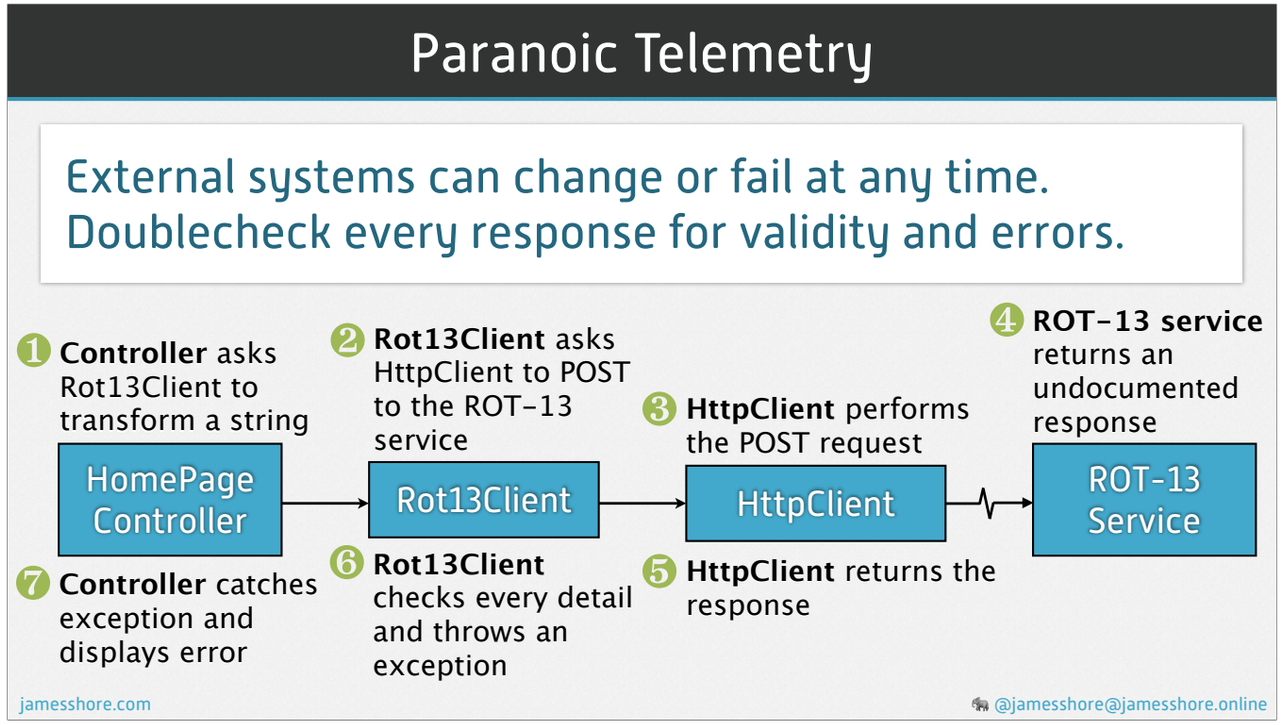
Testing High-Level Wrappers

Fake It Once You Make It: Manually confirm the external system's behavior, then use Nullable low-level wrappers to test your high-level wrapper.



When you build the high-level wrapper, you don't need to write tests that call the real service. External systems often aren't designed to support automated tests, so those tests tend to be flaky and slow. Fortunately, the real work of a high-level wrapper is the translation layer, not the calls to the service, so it's okay if you don't test it directly.

Instead of testing the external system directly, use a Nullable low-level wrapper to simulate the external system's behavior. Write your tests to operate against the Nulled wrapper, not the service. In the case of the Rot13Client, your tests will create a Nulled HttpClient and configure it to behave the same way as the ROT-13 service.



That does raise the question: if you don't test against the external system directly, how do you know your wrapper works?

First and most important, when you write your tests, manually call the service and encode what you learn into your tests. You can also use a normal **create()** instance in your tests while you're developing them. Once you're confident in your understanding of the external service, switch to using Nulled instances with **createNull()**.

But because external systems can fail at any time, it's not enough to test that the service works. You need to validate every call at runtime. Something **always** goes wrong eventually. In the "Testing Without Mocks" pattern language, this is called "Paranoic Telemetry." Check that the service is working the way you expect, and if it isn't, fail safe. Use a fallback response and log the problem.

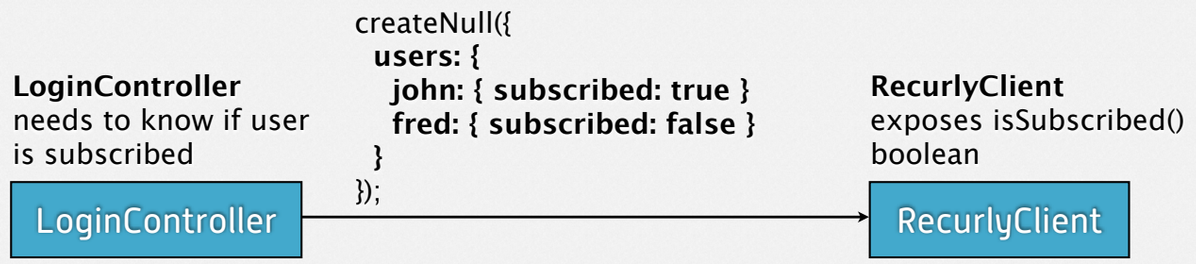
Often, the best way to fail safe is to throw an exception, and expect the application-level code to handle it properly. The application code has the context to make the right decision.

For example, in the ROT-13 application, the HomePageController asks the Rot13Client to transform a string. The Rot13Client asks the HttpClient to POST to the ROT-13 service. And the HttpClient makes the request. Now, let's say the ROT-13 service returns an undocumented response. Perhaps a 503 error, meaning "Service Unavailable." The HttpClient will return the response to the Rot13Client. The Rot13Client checks the response, realizes it's not valid, and throws an exception. The HomePageController catches that exception and displays an error message on the web page.

Paranoic Telemetry has worked well for me, but if you're worried it's not enough, you can supplement it with Contract Tests. Contract Tests are focused integration tests that call the external service. They work best with other teams in your company, because they should be run **before** changes are released, which typically requires them to be created by your team and handed off to the providing team for them to add to their own test suite. This level of coordination is difficult to achieve with third parties.

Designing Nullability

Design your Configurable Responses according to **the state callers want to control**, not specific method calls or return values.

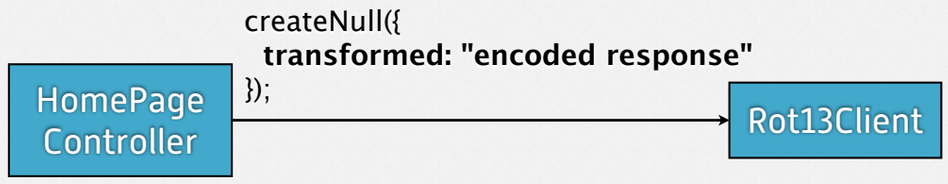


Once your high-level wrapper is working, you'll want to make it Nullable, so you can test code that uses it without worrying about external systems or dependencies. Once again, the magic word is **encapsulation**. Design your createNull() factory to configure the wrapper from the callers' perspective, not the actual implementation details. It's not a mock, so don't try to configure specific method calls or return values. Instead, think about the state of the system that you want to control.

For example, in that subscription codebase, I had a RecurlyClient that exposed an isSubscribed() boolean. A mock would be configured to say "isSubscribed() returns true the first time it's called and false the second time it's called." But the nullable was configured with a list of users, with a "subscribed" flag that could either be true or false.

Designing Nullability

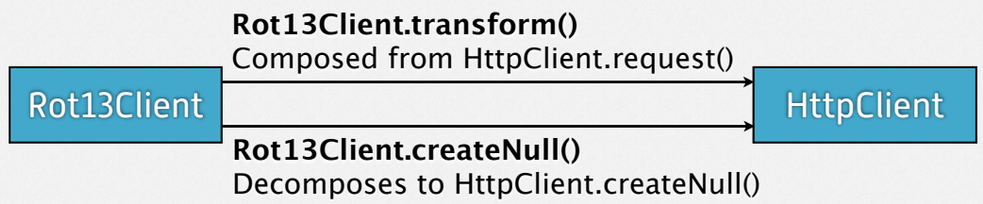
But… sometimes it's simpler to specify responses rather than state. Use your best judgment, but error on the side of specifying state.



On the other hand, some of your high-level wrappers will be stateless, like Rot13Client. Although you could configure it to say "this input results in this output," it's often simpler to just say "return this response." (You can use an array to specific multiple different responses. We'll look at that option in a later module.)

Implementing Nullability

Just as a function's return value is **composed up** from its dependencies, so are Nullables and their Configurable Responses **decomposed down**.



When you implement the createNull() factory, you'll decompose the configuration into the low-level state needed by your low-level wrapper. This is essentially the reverse operation from your normal API. In Rot13Client, the transform() method will call HttpClient and convert the HTTP status and body into a string. The createNull() factory will call HttpClient.createNull() and convert the string into an HTTP status and body. You'll see how it works when you do the exercises.

Designing Output Tracking

Output Tracking allows you to observe events. It looks like a spy, but it focuses on **behavior**, not function calls, and what's **useful to callers**.

```
log.info({
  err: new Error("my_text")
});
```

⇒

```
{
  alert: "info",
  err: "my_text",
}
```


@jamesshore@jamesshore.online

Your high-level infrastructure wrappers will typically need to support output tracking. Remember, output tracking is how application tests can tell if something was sent to an external service. Once again, **encapsulation** is your watchword. Design your output tracking to focus on behavior. For example, imagine you're adding output tracking to a logger. A spy would say "log.info was called with this Error object". The tracker says "the logger has logged these items"—regardless of which alert method was used.

As you build your tracking, focus on what's useful to callers. For example, the logger takes a structured log, converts it to a string, includes exception stack traces, and adds a timestamp. The actual output is just a flat string. But your tests want to know the original structured data that was output. They care about the order of the logs, but they don't want to have to check the time. They care about error messages, but not stack traces. So a good output tracker for a logger would expose the structure, not the string. It would include error messages, but not timestamps or stack traces.

Implementing the tracking is pretty straightforward. It uses a helper class. The details are in the exercises.

Further Reading

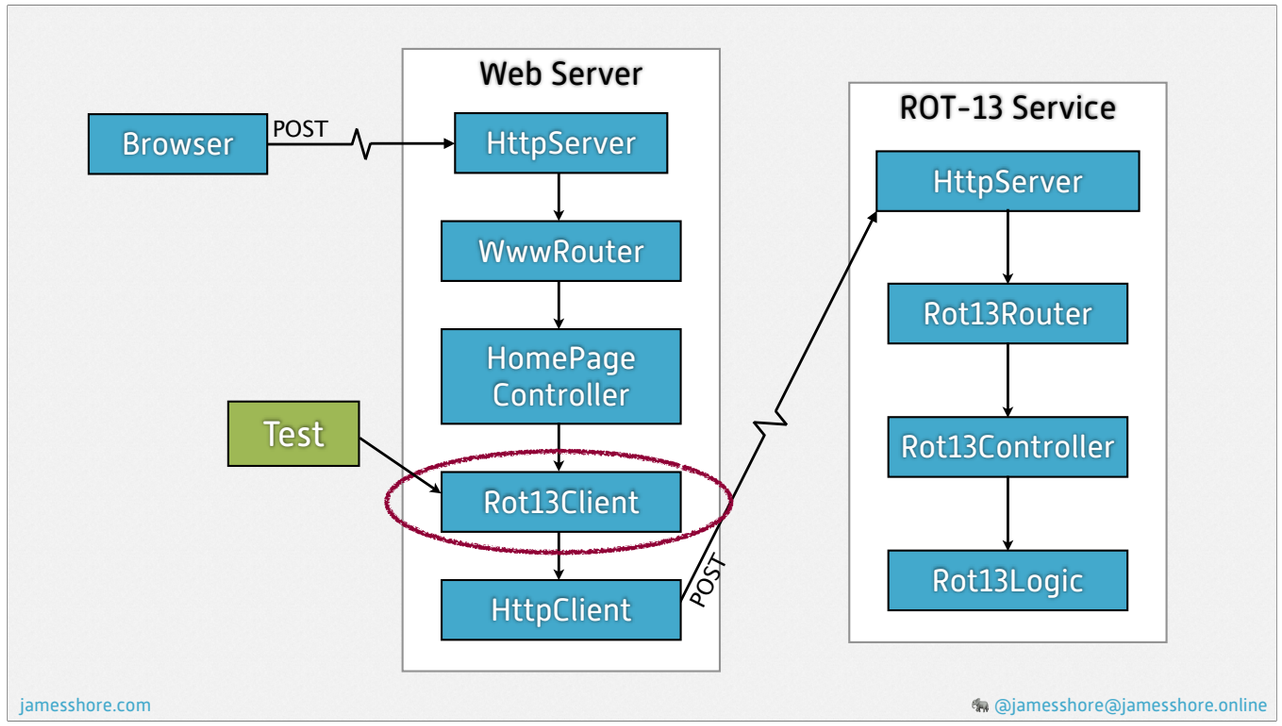
Patterns in **jamesshore.com/s/nomocks**:

- Infrastructure Wrappers
- Fake It Once You Make It
- Paranoic Telemetry
- Nullables
- Configurable Responses
- Output Tracking

To summarize, we've talked about how to build high-level infrastructure wrappers. You're going to **Fake It Once You Make It** by using a Nullable low-level wrapper to test your code, rather than testing the external system directly. You'll include **Paranoic Telemetry** that assumes that external system will fail at runtime. You'll make your wrapper **Nullable** by implementing a "createNull()" factory, and it will take **Configurable Responses** that define the nulled state. Finally, you'll use **Output Tracking** to make calls to the external system visible.

Exercises

The exercises will help solidify your understanding of these concepts.



You'll be working in the same application as before, but this time, you'll be implementing the Rot13Client class. Rot13Client uses HttpClient to make the actual request of the ROT-13 service. Your tests will check that Rot13Client is implemented correctly, including how it uses HttpClient.

(Walk through exercise setup.)

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

Testing Without Mocks
High-Level Infrastructure

Presented by James Shore
jamesshore.com

v2023-06-05

And that's how you design and implement high-level infrastructure wrappers. The exercises will help bring everything together. Good luck!