Building & Testing HTTP clients in Rust

Florin Lipan

Why?

Good old TcpStream

```
use std::net::{TcpStream, Shutdown};
use std::io::{Read, Write};

let mut stream = TcpStream::connect("example.com:80").unwrap();

stream
  .write_all(b"GET / HTTP/1.1\nHost: example.com\n\n")
  .unwrap();

let mut response = String::new();
stream
  .take(10)
  .read_to_string(&mut response)
  .unwrap();

// stream.shutdown(Shutdown::Both);
```

A better Reader...

```
use std::io::{BufRead, BufReader};

// ...

let mut reader = BufReader::new(stream);
let mut lines = reader.lines();

let mut status_line: String = lines.next().unwrap().unwrap();

println!("status: {}", status_line);
// => HTTP/1.1 200 OK

let mut header_lines: Vec<String> = lines
  .take_while(|line| line.as_ref().unwrap() != "\r\n")
  .map(|line| line.unwrap())
  .collect();

// => Content-Length
```

Why not?

- No SSL, but there are openssl bindings for Rust
- SSL is scarry
- Verbose
- Blocking reads/writes (?), but set_nonblocking() landed in 1.9

hyper

```
extern crate hyper;

use hyper::client::Client;
use hyper::status::{StatusCode, StatusClass};

let mut response =
  Client::new()
    .get("https://www.example.com/")
    .send()
    .unwrap();

match response.status {
  StatusCode::Ok => { println!("success!") },
  _              => { println!("meh") },
}

// => success!

match response.status.class() {
    StatusClass::Success => { println!("yay!") },
    _                    => { println!("nay") },
}
```

Bit of JSON...

```
extern crate rustc_serialize;

use rustc_serialize::json;

#[derive(RustcDecodable)]
struct Board {
    name: String,
    desc: String,
}

let mut response = Client::new()
    .get("https://api.trello.com/1/members/me/boards?token=x&key=y")
    .send()
    .unwrap();

let mut body = String::new();
response.read_to_string(&mut body).unwrap();

let boards: Vec<Board> = json::decode(&body).unwrap();

let first_board = boards.first().unwrap();
println!("first board: {}", first_board.name);
```

...and the other way around

```
#[derive(RustcEncodable)]
struct Board {
    name: String,
    desc: String,
}

let board =
  Board {
    name: "demo".to_string(),
    desc: "just a demo board".to_string(),
  };

let body = json::encode(&board).unwrap();

let mut response = Client::new()
    .post("https://api.trello.com/1/boards?token=x&key=y")
    .header(ContentType::json())
    .body(&body)
    .send()
    .unwrap();
```

Other options?

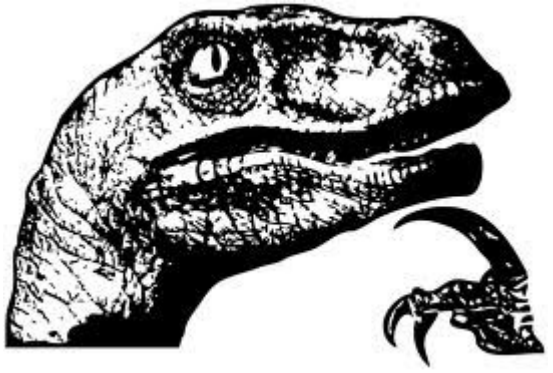
- http_muncher = bindings for the Node HTTP parser, supports upgrade connections

- solicit = HTTP/2

- rust-http = not maintained

- TeePee = not developed yet (?)

So how do I test these things?



hyper (internally)

```
test! {
  name: client_get,

  server:
    expected: "GET / HTTP/1.1\r\nHost: {addr}\r\n\r\n",
    reply: "HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\r\n",

  client:
    request:
      method: Get,
      url: "http://{addr}/",
    response:
      status: Ok,
      headers: [
        ContentLength(0),
      ],
      body: None
}

// assert_eq!(s(&buf[..n]), format!($server_expected, addr=addr));
// inc.write_all($server_reply.as_ref()).unwrap();
// + client asserts
```

hyper-mock

```
// https://github.com/Byron/yup-hyper-mock

#[cfg(test)]
mod tests {
    use hyper::Client;
    use hyper::status::StatusCode;

    mock_connector!(MockStatus {
      "https://127.0.0.1" =>"HTTP/1.1 200 OK\r\n\Server: mock\r\n\r\n\"
    });

    #[test]
    fn test_status_ok() {
        let mut client = Client::with_connector(MockStatus::default());

        let response = client.get("http://127.0.0.1").send().unwrap();
        assert_eq!(StatusCode::Ok, response.status);
    }
}
```

(drumroll)

Finally a crate with a logo (tm)

Concept

- An HTTP server (based on hyper) running on port 1234, on a separate thread of your application

- Set the HOST via compiler flags: `#[cfg(test)]` vs. `#[cfg(not(test))]`

- Simple interface

Basic example (1)

```
#[cfg(test)]
extern crate mockito;

#[cfg(test)]
use mockito;

#[cfg(test)]]
const HOST: &'static str = mockito::SERVER_URL;

#[cfg(not(test))]
const HOST: &'static str = "https://api.trello.com";

fn request() -> Response {
  Client::new()
    .get([HOST, "/1/members/me/boards?token=x&key=y"].join(""))
    .send()
    .unwrap()
}
```

Basic example (2)

```
#[cfg(test)]
mod tests {
  use mockito::mock;
  use hyper::status::StatusCode;
  use {request};

  #[test]
  fn test_request_is_ok() {
    mock("GET", "/1/members/me/boards?token=x&key=y")
      .with_body("{}")
      .create();

    let response = request();

    assert_eq!(StatusCode::Ok, response.status);
  }
}
```

Matching headers

```
mock("GET", "/hello")
   .match_header("accept", "text/json")
   .with_body("{'hello': 'world'}")
   .create();
```

```
mock("GET", "/hello")
   .match_header("accept", "text/plain")
   .with_body("world")
   .create();
```

Other options

```
// Set response status
mock(“GET”, “/hello”)
  .with_status(422)
  .with_body(“”)
  .create();


// Set response headers
mock(“GET”, “/hello”)
  .with_header(“content-type”, “application/json”)
  .with_header(“x-request-id”, “1234”)
  .with_body(“”)
  .create();


// Read response body from a file
mock(“GET”, “/hello”)
  .with_body_from_file(“path/to/file”)
  .create();
```

Cleaning up

```
// Closures
mock("GET", "/hello")
  .with_body("world")
  .create_for(|| {
    // Mock only available for the lifetime of this closure
    assert!(...)
  });
```

```
// Manually
let mut mock = mock("GET", "/hello");
mock
  .with_body("world")
  .create();
assert!(...)
mock.remove();
```

Compiler flags (1)

```
#[cfg(feature="mock")]
const HOST: &'static str = mockito::SERVER_URL;
#[cfg(not(feature="mock"))]
const HOST: &'static str = "https://api.trello.com";

#[cfg(test)]
mod tests {
  #[test]
  #[cfg(feature="mock")]
  fn test_with_a_mock() {
    // will run only if the mock feature is enabled
  }

  #[test]
  #[cfg(not(feature="mock"))]
  fn test_without_a_mock() {
    // will run only if the mock feature is disabled
  }

  #[test]
  fn test_dont_care() {
    // will run all the time
  }
}
```

Compiler flags (2)

```
// Cargo.toml
[features]
default = []
mock    = []
other   = []


// will run without the `mock` feature
cargo test


// will run with the `mock` feature
cargo test --features mock


// will run the `mock` and the `other` feature
cargo test --features “mock other”
```

Drawbacks

- Port 1234

- Projects requiring multiple hosts, but conflicts can be avoided

- Using `hyper::status::StatusCode::Unregistered` which labels every status as `<unknown status code>`: e.g. `201 <unknown status code>`

- Fresh off the machine

- https://github.com/lipanski/mockito
- https://github.com/lipanski/trello-rs
- https://github.com/Byron/yup-hyper-mock
- https://github.com/hyperium/hyper
- https://github.com/kbknapp/clap-rs (lovely CLI argument parser)
- Logo courtesy to http://niastudio.net

Questions?