```
/*
 *
 *
 * This code works now. it needs some serious cleanup however. wow is it gross.
 *
 * learn how to build and embed into shipped python:
 *      https://github.com/mckaymatt/cookiecutter-pypackage-rust-cross-platform-publish
 *      https://pypi.python.org/pypi/setuptools-rust
 *      https://github.com/getsentry/milksnake
 *      https://github.com/getsentry/libsourcemap/blob/master/setup.py
 *
 * echo '{"agent_token":"super secret", "master_key": "secret", "timeout":30, "port":"6278"}' |
 * agent
 *
 * and the request:
 *curl --data '{"agent_token":"super secret"}' --header "Content-Type: application/json" http://localhost:6278
 *
 */
//#[macro_use]
extern crate serde;
#[macro_use]
extern crate serde_derive;

extern crate serde_json;
// #[macro_use]
extern crate bitwarden;
extern crate daemonize;
extern crate secstr;
extern crate tiny_http;

// use rouille::Request;
// use rouille::Response;
// use std::collections::HashMap;
use std::io;
// use std::sync::Mutex;
use secstr::*;
// use std::time::{SystemTime, UNIX_EPOCH};
use std::time::{Duration, Instant};
use std::fs::File;
use daemonize::Daemonize;

// This struct contains the data that we store on the server about each client.
#[derive(Debug, Clone)]
struct SessionData {
        login: String,
}

#[derive(Deserialize)]
struct Setup {
        master_key: String,
        agent_token: String,
        timeout: u64,
        port: isize,
        agent_location: String,
}
/*
struct Secrets {
    agent_token: SecStr,
    master_key: SecStr,
}
*/
#[derive(Deserialize)]
struct TokenRequest {
        agent_token: String,
        exit: bool,
}
#[derive(Serialize)]
struct TokenResponse {
        master_key: String,
        error: String,
}
//
//  echo '{"agent_token":"super secret", "master_key": "secret", "timeout":30, "port":"6278"}' |
//  target/debug/agent

fn serve(setup: Setup) {
        let secret_agent_token = SecStr::from(setup.agent_token);
        let secret_master_key = setup.master_key;
        // let secrets = Secrets {
        //    agent_token: SecStr::from(setup.agent_token),
        //   master_key: SecStr::from(setup.master_key),
        // };
        // set timeout to 1 year if bitwarden doesn't give us one.
        let mut timeout = Duration::new(31536000,0);
        if setup.timeout > 0 {
                timeout = Duration::new(setup.timeout, 0);
```

```
87          }
88          let port = setup.port;
89          let server = tiny_http::Server::http(format!("localhost:{}", port)).unwrap();
90          println!("Now listening on localhost:{}", port);
91
92          let start_time = Instant::now();
93          let mut exit = true;
94          let receive_timeout = Duration::new(1, 0);
95
96          // loop for freaking ever, or agent_timeout happens..
97          while exit {
98                  // block for up to 1 second.
99                  let request = match server.recv_timeout(receive_timeout) {
100                         Ok(rq) => rq,
101                         Err(e) => {
102                                 println!("error: {}", e);
103                                 break;
104                         }
105                 };
106                 if request.is_some() {
107                         let mut rq = request.unwrap();
108                         let url = rq.url().to_string();
109                         let method = rq.method().to_string();
110                         let mut content_type = String::from("");
111                         for header in rq.headers() {
112                                 println!("header: {} : {}", header.field, header.value);
113                                 if header.field.to_string() == "Content-Type" {
114                                         content_type = header.value.to_string();
115                                 }
116                         }
117                         println!(
118                                 "remote request from:{} requesting: {} method:{} content_type:{} ",
119                                 rq.remote_addr().to_string(),
120                                 url,
121                                 method,
122                                 content_type
123                         );
124                         // assert_eq(request.remote_addr(), "127.0.0.1");
125                         if method == "POST" {
126                                 println!("post!");
127                                 if content_type == "application/json" {
128                                         println!("JSON!");
129                                         let mut content = String::new();
130                                         rq.as_reader().read_to_string(&mut content).unwrap();
131                                         let token_request: TokenRequest = serde_json::from_str(&content).unwrap
    ();
132                                         if SecStr::from(token_request.agent_token) == secret_agent_token {
133                                                 if token_request.exit == true {
134                                                         println!("exit requested");
135                                                         std::process::exit(0);
136                                                 }
137                                                 // let local_master_key = secret_master_key;
138                                                 let token_response = TokenResponse {
139                                                         error: "".to_string(),
140                                                         master_key: secret_master_key.clone().to_string(),
141                                                 };
142                                                 let response = tiny_http::Response::from_string(
143                                                         serde_json::to_string(&token_response).unwrap(),
144                                                 );
145                                                 // return Response::json(&token_response);
146                                                 let _ = rq.respond(response);
147                                         } else {
148                                                 let token_response = TokenResponse {
149                                                         error: "Invalid agent_token.".to_string(),
150                                                         master_key: "".to_string(),
151                                                 };
152                                                 let response = tiny_http::Response::from_string(
153                                                         serde_json::to_string(&token_response).unwrap(),
154                                                 );
155                                                 let _ = rq.respond(response);
156                                         }
157                                 } else {
158                                         let token_response = TokenResponse {
159                                                 error: "Must send Content-Type: application/json".to_string(),
160                                                 master_key: "".to_string(),
161                                         };
162                                         let response = tiny_http::Response::from_string(
163                                                 serde_json::to_string(&token_response).unwrap(),
164                                         );
165                                         let _ = rq.respond(response);
166                                 }
167                                 // end server handler
168                         }
169                         }
170
171                 let now = Instant::now();
```

```
                if now.duration_since(start_time) > timeout {
                        exit = false;
                }
        }
        println!("exiting because of timeout hit.");

}

fn main() {
        let mut input = String::new();
        io::stdin().read_line(&mut input).unwrap();
        let setup: Setup = serde_json::from_str(&input).unwrap();
        println!(
                "setup: {:?} {:?} {:?}",
                setup.master_key, setup.agent_token, setup.timeout
        );
            let stdout = File::create(format!("{}/agent.out", setup.agent_location)).unwrap();
        let stderr = File::create(format!("{}/agent.err", setup.agent_location)).unwrap();
            let daemonize = Daemonize::new()
        .pid_file(format!("{}/agent.pid", setup.agent_location)) // Every method except `new` and `start`
        .chown_pid_file(true)      // is optional, see `Daemonize` documentation
        .working_directory(format!("{}",setup.agent_location)) // for default behaviour.
        //.user("nobody")
        //.group("daemon") // Group name
        //.group(2)        // or group id.
        .umask(0o027)    // Set umask, `0o027` by default.
        .stdout(stdout)  // Redirect stdout to `/tmp/daemon.out`.
        .stderr(stderr);  // Redirect stderr to `/tmp/daemon.err`.
        //.privileged_action(|| "Executed before drop privileges");

    match daemonize.start() {
        Ok(_) => println!("Success, daemonized"),
        Err(e) => eprintln!("Error, {}", e),
    }
        serve(setup);
}
```