Assignment 8: Rust and WebAssembly

Question 1: Consider the following piece of code:

```
use hyper::rt::Future;
use hyper::service::service_fn_ok;
use hyper::{Body, Request, Response, Server};

fn main() {
 let addr = ([127, 0, 0, 1], 3000).into();
 let server = Server::bind(&addr)
    .serve(|| {
       service_fn(service_router)
    })
    .map_err(|e| eprintln!("server error: {}", e));

 println!("Listening on http://{}", addr);
 hyper::rt::run(server);
}

fn svc_wait(t: u64) -> impl Future<Item = (), Error = ()> {
  println!("[start] waiting...");
  let when = Instant::now() + Duration::from_millis(t);
  Delay::new(when)
     .map_err(|e| panic!("timer failed; err={:?}", e))
     .and_then(|_| {
        println!("[end] waiting");
        Ok(())
     })
}


fn fetch_data() -> impl Future<Item = future::FutureResult<RespStruct,
       String>, Error = ()> {
    let uri: Uri = "http://httpbin.org/get".parse().expect("Cannot parse
    URL");
    Client::new()
        .get(uri)
        // Future is polled here
        .and_then(|res| {
            res.into_body().concat2()
        })
        .map_err(|err| println!("error: {}", err))
        .map(|body| {
            let decoded: RespStruct =
            serde_json::from_slice(&body).expect("Couldn't deserialize");
            future::ok(decoded)
        })
}


type BoxFut = Box<dyn Future<Item = Response<Body>, Error = hyper::Error>
                                                              + Send>;


fn service_router(req: Request<Body>) -> BoxFut {
```

```
56      let mut response = Response::new(Body::empty());
57
58      match (req.method(), req.uri().path()) {
60
61
62        (&Method::GET, "/wait") => {
63              let r = svc_wait(1500);
64              hyper::rt::spawn(r);
65              *response.body_mut() = Body::from(format!("Triggered waiting
66                    {}ms", 1500));
67          }
68
69
70          (&Method::GET, "/fetch") => {
71              let r = fetch_data().map(|x| {
72                  println!("got data: {:?}", x);
73              });
74              hyper::rt::spawn(r);
75              *response.body_mut() = Body::from("Sent request to external
76  webservice");
77          }
78
79          // ... more routers
80      }
81      eprintln!("Returning a response");
82      Box::new(future::ok(response))
83  }
```

a- Explain what do the numbers mean in line 9.

b- The function in line 20 uses **Future**; what is Future?

c- What does http://httpbin.org do (line 34)?

d- Give a definition for the **body** variable in line 45.

e- Explain the **BoxFut** type in line 51

f- Should **BoxFut** (Line 51) implement the Sync trait?

g- Should **BoxFut** (Line 51) use a lifetime?

h- At some points, you will be using the following instruction:

```
$ curl localhost:3000/wait
```

What does **curl** do?

Does this code use Async/IO, if not, how would you change the program to use it? **Question 3:**

Question 2: Libra (libra.org) is a major new product from Facebook. Libra is a cryptocurrency platform. Facebook expect to make billions from Libra and revolutionize the financial industry.

a- What language is Libra written in?

b- Discuss the technical reasons why this choice of language suits the application and its objectives.

c- Libra uses many standard packages, including lazy_static, tokio, failure, etc. Briefly, describe each of these packages.

Question 3: Consider the following program:

a- What is nighty channel in Rust (check Playground)

b- What are unstable features?
c- Why can playground run this code (think O.S.)
d- What is the output from this code?
e- Provide comments for the lines ending in #

```
#![feature(asm)]
fn main() {
    let message = String::from("James, you are completely mad\n");
    syscall(message);
}

#[cfg(target_os = "linux")]
fn syscall(message: String) {
    let msg_ptr = message.as_ptr();
    let len = message.len();
    unsafe {
        asm!("
        mov     $$1, %rax   #
        mov     $$1, %rdi   #
        mov     $0, %rsi    #
        mov     $1, %rdx    #
        syscall             #
    "
        :
        : "r"(msg_ptr), "r"(len)

        )
    }
}
```