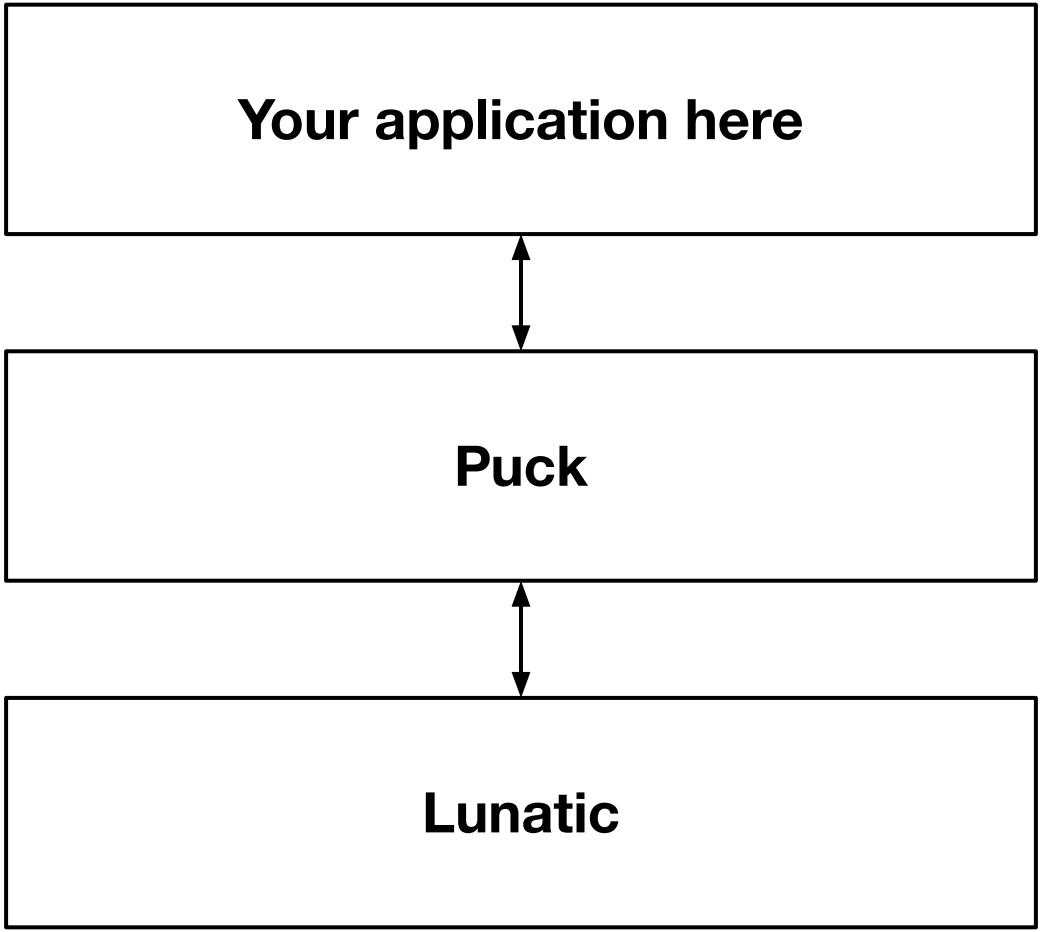
Puck

Teymour Aldridge

Highly interactive real-time applications

On the Lunatic VM


Your application here
Puck
Lunatic

HTTP

- Nothing particularly novel here (yet), handles standard request-response architecture.
- Can builder routers (which will, at some point be based around radix-trees)

```
.route(Route::new(
    |request| {
        Match::new()
            .at(match_url::path("read"))
            .at(match_url::any_integer())
            .does_match(request.url())
    },
    |request, stream, state| {
```

Struct puck::request::Request

```
pub struct Request { /* private fields */ }
```

[–] A HTTP request.

Struct puck::response::Response

```
pub struct Response { /* private fields */ }
```

[–] A HTTP response.

Struct puck::core::router::Router

source · [–]

```
pub struct Router<STATE> { /* private fields */ }
```

[–] A Router provides an easy way to match different types of HTTP request and handle them differently.

source

```
pub fn new(
    matcher: fn(_: &Request) -> bool,
    handler: fn(_: Request, _: Stream, _: STATE) -> UsedStream
) -> Route<STATE>
```

Interactivity

- WebSockets
- Lunatic processes (WebAssembly green threads)
- Use Erlang style abstractions (GenServer, supervisors, etc.) for highly resilient systems (“let it crash”)

Liveview

- Server-side DOM diffing
- This is harder to do without a runtime which allows for running lots of threads at the same time (e.g. Lunatic, OTP/Erlang, etc.)
- Advantages
  - Can support users who don’t run Javascript (mostly, a few features won’t work) - this is WIP for Puck
  - Reduces amount of Javascript which needs to be written (subjective, but this *is* a Rust meetup) => reduces application complexity
  - Less data has to be transmitted than if we were to re-render each page completely on the server on every new page
- It probably isn’t the right thing if you have a highly complex client

Demo

- I did try to get a live (online) demo to work, but we can’t yet support SSL (big thing missing, I know) directly in WebAssembly (in Rust, without rolling our own crypto). We have a simple workaround but not yet implemented.

Roadmap

Safety matters

Sure, it doesn’t sound as fun as “speed” or “rapid development” but we should try to lower its cost

Roadmap: invalid states unrepresentable

- E.g. with WebSockets, in the protocol we have a series of states - we can use typestate/session types to make invalid states unrepresentable.
  - E.g. users are passed the TCPStream, but to use it they have to call WebSocket::upgrade, which take ownership of the stream
  - Otherwise have to return it at the end of the request, so that we can use it for Keep-Alive (this allows us to avoid needing to start a new process)
- Lunatic also offers session types which users can use in their applications (we plan to do so for our chat architecture)
- Build some more strict abstractions (e.g. for HTTP authentication, etc) taking inspiration from web frameworks such as Rocket

Roadmap: probabilistic methods for testing applications

This is an in-progress design - contributions welcome!

- Grammar-based fuzzing
- (lacking coverage APIs for WebAssembly, and also random number generation)
- Concurrency (i.e. temporal dependencies between processes) and interleaving - generate requests from separate users, and send them in different orders.
- At each step check *safety* properties (i.e. invariants) and *liveness* properties (i.e. the program should eventually achieve some good state)

Grammar-based fuzzing for HTTP

Generating request sequence

- Pick *n* users
- Pick maximum timestamp *n*
- For each user, and timestamp
  - Generate a request (e.g. create some resource, delete a random resource)
  - Run in order at the given timestamp
- Emergent engineering - for tests only specify what the system should do, then feed in lots of inputs and check that it doesn’t misbehave

github.com/puck-rs/puck