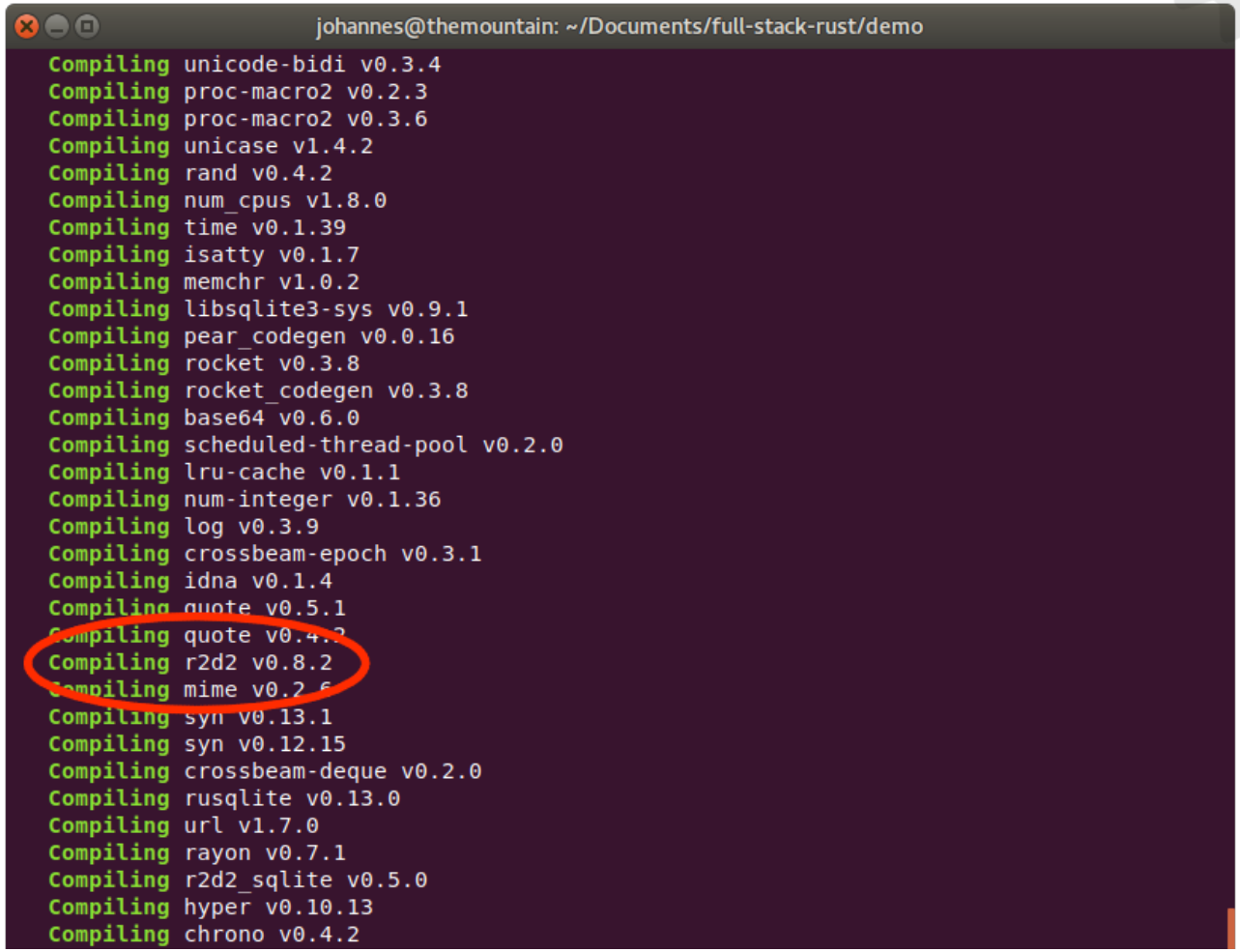
Full stack Rust

Hyper Diesel Rust Rocket

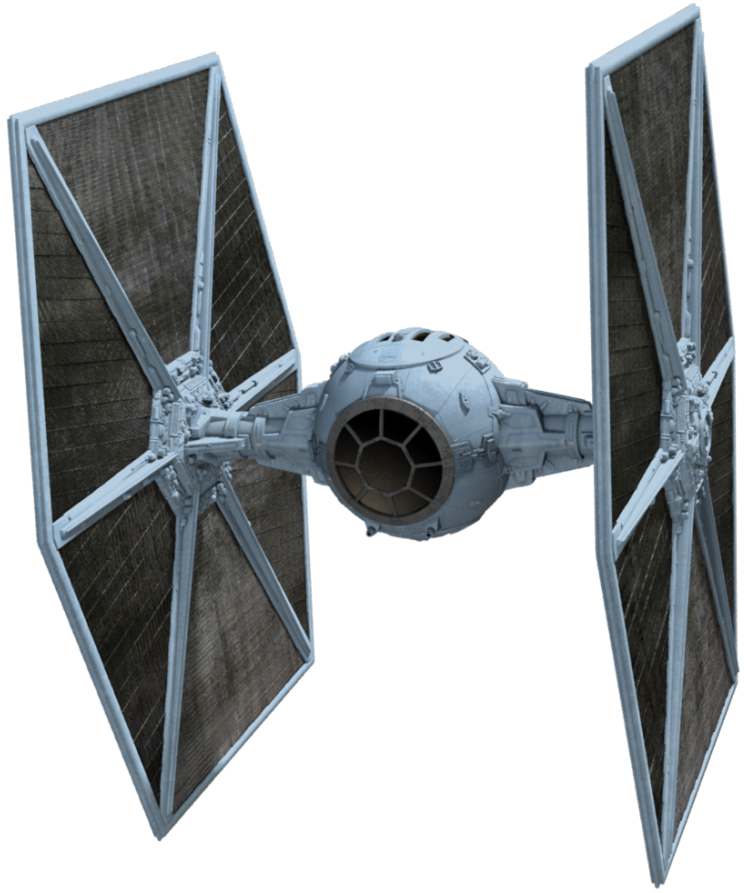
Johannes Schriewer aka. Dunkelstern

Wenn es fertig ist sieht das ganze so aus...

Wenn man genau hinschaut ist sogar R2D2 mit an Board:



In Go sähe es dann so aus:



Übersicht

1. Wer bin ich
2. Rust im Backend
3. Rust für Datenbanken
4. Rust im Browser (WASM)

Wer bin ich

- Backend-Entwickler bei anfema
- Normalerweise schreibe ich Python und Node.js
- Code seit 2003 in diversen Bereichen
- Enttäuscht von Swift

Wer bin ich

- Backend-Entwickler bei anfema
- Normalerweise schreibe ich Python und Node.js
- Code seit 2003 in diversen Bereichen
- Enttäuscht von Swift
- Manche sagen ich bin ein irrer Bastler

Wer bin ich

- Backend-Entwickler bei anfema
- Normalerweise schreibe ich Python und Node.js
- Code seit 2003 in diversen Bereichen
- Enttäuscht von Swift
- Manche sagen ich bin ein irrer Bastler
- Wenn ich nicht Code bastel ich Elektronikgerümpel

Rust im Backend

- Es gibt verschiedene Frameworks

Rust im Backend

- Es gibt verschiedene Frameworks
  - Iron
  - Nickel
  - Conduit
  - Rocket
  - Gotham
  - ...

Rust im Backend

- Es gibt verschiedene Frameworks
  - Iron
  - Nickel
  - Conduit
  - Rocket
  - Gotham
  - ...
- Das Benutzerfreundlichste ist Rocket

Rust im Backend

- Es gibt verschiedene Frameworks
  - Iron
  - Nickel
  - Conduit
  - Rocket
  - Gotham
  - ...
- Das Benutzerfreundlichste ist Rocket (IMHO)

Beispiel: Rocket

```
#![feature(plugin)]
#![plugin(rocket_codegen)]

extern crate rocket;

#[get("/hello/<name>/<age>")]
fn hello(name: String, age: u8) -> String {
    format!("Hello, {} year old named {}!", age, name)
}

fn main() {
    rocket::ignite().mount("/", routes![hello]).launch();
}
```

JSON API Beispiel

Model

```
#![feature(custom_derive)]

extern crate serde;
#[macro_use]
extern crate serde_derive;
extern crate serde_json;
extern crate chrono;

use chrono::prelude::*;

#[derive(Serialize, Deserialize, Debug)]
pub struct Person {
    pub id: i32,
    pub name: String,
    pub birthday: NaiveDateTime,
}
```

View

```
use rocket::response::Failure;
use rocket_contrib::Json;

#[get("/person/<id>", format = "application/json")]
fn get_person(id: i32) -> Result<Json<Person>, Failure> {
    Ok(Json(Person {
        id,
        name: String::from("Max Musterman"),
        birthday: Local::now().naive_local(),
    }))
}
```

Die anderen HTTP Methoden funktionieren natürlich auch...

Ergebnis

```
$ curl -v http://localhost:8080/person/1 \
      -H 'Accept: application/json'
> GET /person/1 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.59.0
> Accept: application/json
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Server: Rocket
< Content-Length: 74
< Date: Sun, 15 Apr 2018 22:32:27 GMT
<
{
    "id": 1,
    "name": "Max Musterman",
    "birthday":"2018-04-16T00:32:27.480509862"
}
```

Request Guards (aka Middleware)

```
#[get("/admin")]
fn admin_panel(admin: AdminUser) -> &'static str {
    "Hello, administrator. This is the admin panel!"
}

#[get("/admin", rank = 2)]
fn admin_panel_user(user: User) -> &'static str {
    "You must be an administrator to access this page."
}

#[get("/admin", rank = 3)]
fn admin_panel_redirect() -> Redirect {
    Redirect::to("/login")
}
```

Rocket wählt automatisch denjenigen View der alle Request guards unterstützt (und den niedrigsten Rank hat)

Rust für Datenbanken

- Auch hier wieder verschiedene Dinge:

Rust für Datenbanken

- Auch hier wieder verschiedene Dinge:
  - Postgres-Lib direkt
  - Rustorm
  - Diesel
  - ...

Rust für Datenbanken

- Auch hier wieder verschiedene Dinge:
  - Postgres-Lib direkt
  - Rustorm
  - Diesel
  - ...
- Ich habe mich auf Diesel eingeschossen weil es typesafe ist.

Model definition (Diesel)

1. Table

```
table! {
    person (id) {
        id -> Integer,
        name -> Text,
        birthday -> Timestamp,
    }
}
```

2. Model

```
#[derive(Serialize, Deserialize, Debug, Queryable, \
         Insertable, Identifiable, AsChangeset)]
#[table_name = "person"]
pub struct Person {
...
```

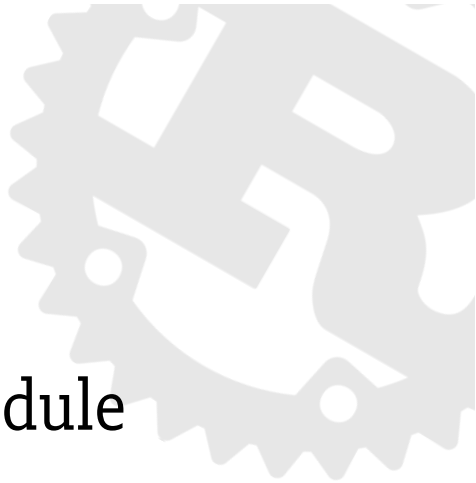
View (Diesel)

```
#[get("/persons")]
pub fn get_person_list(conn: DbConn)
     -> QueryResult<Json<Vec<Person>>>
{
    person::table
        .order(person::id.asc())
        .load::<Person>(&*conn)
        .map(|person| Json(person))
}

#[get("/person/<id>")]
pub fn get_person(id: i32, conn: DbConn)
    -> Result<Json<Person>, Failure>
{
    person::table
        .find(id)
        .first::<Person>(&*conn)
        .map_err(|_| Failure(Status::NotFound))
        .map(|person| Json(person))
}
```

View (continued)

```
#[post("/person", data="<data>")]
pub fn create_person(data: Json<Person>, conn: DbConn)
    -> Result<Json<Person>, Failure>
{
    let rows_inserted = insert_into(person::table)
        .values(&data.into_inner())
        .execute(&*conn)
        .unwrap();

    if rows_inserted != 1 {
        Err(Failure(Status::InternalServerError))
    } else {
        person::table
            .order(person::id.desc())
            .first::<Person>(&*conn)
            .unwrap()
            .map(|person| Json(person))
    }
}
```

Rust im Browser



1. *wasm-bindgen* und Webpack, JS module in Rust
2. *yew*, React-style Framework, Everything Rust

WebAssembly Module für Javascript

Rust Teil

```rust
#![feature(proc_macro, wasm_custom_section, wasm_import_module)]

extern crate wasm_bindgen;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

Vorbereiten

```
$ rustup target add wasm32-unknown-unknown --toolchain nightly
$ cargo install wasm-bindgen-cli
```

Compile

```
$ cargo +nightly build --target wasm32-unknown-unknown
$ wasm-bindgen target/wasm32-unknown-unknown/debug/wasm_greet.wasm \
  --out-dir .
```

Javascript

```
const rust = import("./wasm_greet");
rust.then(m => m.greet("World!"));
```

Direkt in Rust rendern: Yew

https://github.com/DenisKolodin/yew

- JSX Style templates mittels *html!* macro direkt im Rust code
- Application state management mittels Message passing
- ReactJS und elm waren die inspiration
- Eigener Virtual DOM
- Components und Fragments wie bereits bekannt

Direkt in Rust rendern: Yew

https://github.com/DenisKolodin/yew

- JSX Style templates mittels *html!* macro direkt im Rust code
- Application state management mittels Message passing
- ReactJS und elm waren die inspiration
- Eigener Virtual DOM
- Components und Fragments wie bereits bekannt
- **Very Alpha**

Dank crates.io findet man aber noch viele Zusatzmodule

Dank crates.io findet man aber noch viele Zusatzmodule

Es gibt übrigens noch kein Crate namens *Solo*.

C3PO ist leider an den verkehrten Anwendungsfall gebunden, es wäre so schön gewesen wenn das ein I18N Crate wäre.

Danke für's Zuhören

Kommt gerne auf mich zu wenn ihr mehr wissen wollt!

Are we web yet?
http://www.arewewebyet.org/

Quellen

- Rocket documentation / Getting started guide
- Diesel documentation
- Mozilla Hacks Blog
- Wookiepedia

Kontakt

- hallo@dunkelstern.de
- Twitter: @dunkelstern
- Blog: blog.dunkelstern.de