Boring software with Haskell

Laurens Duijvesteijn

June 2018

duijf.io · github.com/duijf · @_duijf

Agenda

Why care about boring software?

What is boring software?

Writing boring software

About this human

My name is Laurens and I make software boring.

DevOps lead at Channable, a product advertising startup.

Haskell in production for 2.5 years

Job scheduling

CLI tooling

Github: `channable/vaultenv`

Reverse proxy/ingress

Websocket-enabled document store

Github: `channable/icepeak`

Data processing

Github: `channable/Alfred-Margaret`

More: `tech.channable.com`

Sounds interesting? We’re hiring for Python and Haskell!

What did I learn about software while working?

What did I learn about software while working?

What do I mean when I call software boring?

What did I learn about software while working?

What do I mean when I call software boring?

What does this have to do with Haskell?

What makes software expensive?

What makes software expensive?

Choose from: spec, development, testing, maintenance.

Maintenance. By a stretch.

Maintenance. By a stretch.

Maintenance. By a stretch.

$>$ 50% Total Cost of Ownership shows up a lot.

Efficient maintenance $\implies$ more $$$.

Maintenance = change

Easy changes = $$$ saved

How do we change effectively?

How do we change effectively?

1. Make change easy

How do we change effectively?

1. Make change easy
2. Then make the easy change

How do we change effectively?

1. Make change easy
2. Then make the easy change

N.b.: The first one may be difficult.

Boring software

Boring software

Solves the **actual** problem without much fanfare.

Can be changed later.

Does not feel like a rollercoaster ride.

Boring software

Solves the **actual** problem without much fanfare.

Can be changed later.

Does not feel like a rollercoaster ride.

Increases your life expectancy

Boring domain $\neq$ boring software

Boring domain $\neq$ boring software

Former: common. Latter: more rare.

Haskell is no silver bullet

Haskell is no silver bullet

It does contain a few features which make it easier to write boring software.

What does this function do?

```haskell
foo      :: String -> Int  -> Int     -> IO ()
```

What does this function do?

```haskell
foo      :: String -> Int  -> Int     -> IO ()
myServer :: String -> Int  -> Int     -> IO ()
```

What does this function do?

```haskell
foo      :: String -> Int  -> Int     -> IO ()
myServer :: String -> Int  -> Int     -> IO ()
myServer :: Host   -> Port -> Seconds -> IO ()
myServer :: Host -> Port -> Seconds -> IO ()
```

The types provide documentation

The types clarify intent

The types prevent errors

(Can become a bit verbose when coupled with CLI + config)

Case study: `schemactl`

Case study: `schemactl`

We’re writing a Postgres database migration CLI tool.

Learn about Haskell libs we’ll see along the way. Find out how they help.

Start minimal

Start minimal

“Simplest thing that could possibly work”

Expand features later

Use fancy lang extensions, libraries when/if needed

But first: Spec work

But first: Spec work

Discover / define goals

Think through the architecture

Avoid “just start coding”

Spec work?

Spec work?

Critically important to achieve “boring” status

I write most of my code while under the shower

(Annecdote about CRDTs)

Spec: Start with Goals

Spec: Start with Goals

Serve as a guide for your project vision.

Allows you to make deciscions.

Spec: Start with Goals

Serve as a guide for your project vision.

Allows you to make deciscions.

Prereq for opinions about project implementation/architecture

Functional goals

Functional goals

Change the schema of a Postgres DB

Support upgrades & downgrades

Support all the Postgres SQL features

Can be used in polyglot projects

Qualitative goals

Qualitative goals

Easy to use

Reliable & predictable

Low learning curve

Thin on complexity

Ex: Think of an architecture.

Responsibilities

Inputs, outputs

Interface / “form”

Possible approaches

Create an EDSL for migrations.

Possible approaches

Create an EDSL for migrations.

Have a cannonical schema definition, make the user edit that. Compare with current DB schema and auto-generate migrations.

Possible approaches

Create an EDSL for migrations.

Have a cannonical schema definition, make the user edit that. Compare with current DB schema and auto-generate migrations.

Make the user define types, infer schema based on that. Compare and auto-generate.

My solution

Polyglot projects, all SQL features, easy to learn; point us in a direction: use plain SQL. (The one true DB DSL)

My solution

Polyglot projects, all SQL features, easy to learn; point us in a direction: use plain SQL. (The one true DB DSL)

Migration: 1 SQL file for upgrade, 1 SQL file for downgrade.

My solution

Polyglot projects, all SQL features, easy to learn; point us in a direction: use plain SQL. (The one true DB DSL)

Migration: 1 SQL file for upgrade, 1 SQL file for downgrade.

Store these in a directory on the filesystem. Store a separate file with the order these need to be applied in. Make the user edit these.

```
db/
  schemactl-config.json
  schemactl-index
  000_bootstrap.sql.up
  001_create_users.sql.up
  001_create_users.sql.down
  002_create_sessions.sql.up
  002_create_sessions.sql.down
```

```json
// db/schemactl-config.json
{
  "host": "localhost",
  "port": 5432,
  "username": "test",
  "password": "test",
  "database": "test"
}
```

```
-- db/schematctl-index
-- List of migrations to run.
-- Note the lack of `up` or `down`
001_create_users.sql
002_create_sessions.sql
```

```sql
-- db/001_create_users.sql.up
CREATE TABLE users (
  id BIGSERIAL PRIMARY KEY,
  email TEXT NOT NULL
);
```