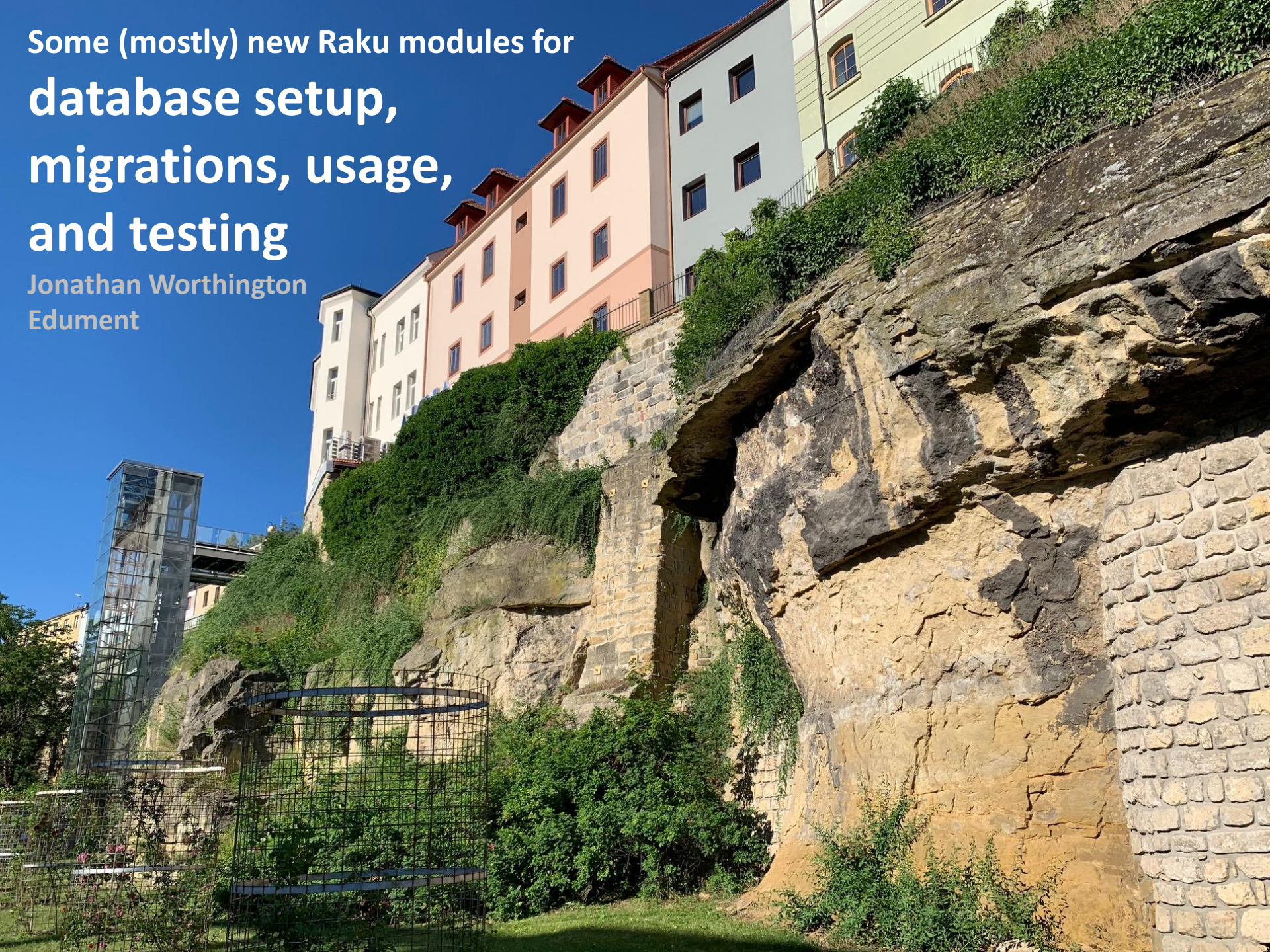

Some (mostly) new Raku modules for
database setup, migrations, usage, and testing
Jonathan Worthington
Edument

Core dev *and* user

Mostly known for contributions to MoarVM and Rakudo

Also a Raku language user, for various web applications (commercial) and compiler hackery (fun)

Core dev *and* user

Mostly known for contributions to Mo lo

Also to blame for founding Cro ☺

Also a Raku language user, for various web applications (commercial) and compiler hackery (fun)

DB frustrations

Pretty much every web application I make involves a database

Wasn't entirely happy with the development experience around it

Made a few modules

Make it easy to get developing

by scripting development service setup

Automate database changes

and have checks to catch silly problems early

Give to SQL what is SQLs

that is, write (at least) the non-trivial queries in SQL

If it ain't tested, it's probably broke

applies to database code too, so test it!

Make it easier to debug DB issues

with easy access to multiple development DBs

Make it easy to set up a development environment?

Make it easy to set up a development environment?

Dev::ContainerizedService

Annoying:

"Hmm, it uses Postgres."

"Sure hope my system version is compatible enough."

"Let's look up how to add a database, I only do this this every few months..."

"...and maybe hack up a script to feed in the database connection string via environment vars."

"Ah bother, now I'm using my other computer, let's do all of this again..."

Mostly tolerable:

"Oh yay, a Docker compose file! Less setup!"

"Oh bother, my `docker-compose` version is too old to support this compose file..."

"Phew, finally it's up."

"Hmm...but how do I connect to the database to poke around inside it?"

But what if...

For my small single-service Raku projects...

...there was a module that let me declare what services I need...

...and it would run the containers...

...and then run my application with the right stuff in the environment?

1

In `META6.json`'s `depends`, add `Dev::ContainerizedService`

Create a Raku script, maybe call it `devenv.raku`

Ensure that database connection details are obtained through the environment

```
my $db = DB::Pg.new:
    conninfo => %*ENV<DB_CONNINFO>;
```

Put this into devenv.raku:

```
#!/usr/bin/env raku
use Dev::ContainerizedService;

service 'postgres', :tag<13.0>, -> (:$conninfo, *%) {
    env 'DB_CONNINFO', $conninfo;
}
```

<demo>

Automate database changes and catch silly mistakes?

Automate database changes and catch silly mistakes?

DB::Migration::Declare

Annoying:

"I'll just keep a `schema.sql` and tweak it when things change..."

"...ah, and I guess write the alteration DDL to apply to the real database..."

"...but it won't change that often, it's a simple project, I'll cope, right?

<a little later>

"I HATE THIS TEDIUM!"

Migrations to the rescue!

Append-only list of changes

Together they bring the database to the current state

Written in SQL directly or generated

Keep a record in the database of which changes have been applied

Apply changes at application startup or explicitly

(startup OK for "small" systems, explicitly better if the application is horizontally scaled out or if there's enough data to seriously delay startup)

Migrations in Raku?

DB::Migration::Simple

Works with DBIish, explicitly write out the SQL for both up and down directions

Red

Migrations support planned, but seem to be work in progress feature; once they are supported, probably this will be ideal for Red users

DB::Migration::Declare

My effort: a Raku DSL for expressing migrations.
Fair warning: it's new, it's BETA, Postgres only so far!

Specify migrations in Raku code

```
use DB::Migration::Declare;

migration 'Setup', {
    create-table 'skyscrapers', {
        add-column 'id', integer(), :increments, :primary;
        add-column 'name', text(), :!null, :unique;
        add-column 'height', integer(), :!null;
    }
}
```

Add further migrations as needed

```
use DB::Migration::Declare;

migration 'Setup', {
    create-table 'skyscrapers', {
        add-column 'id', integer(), :increments, :primary;
        add-column 'name', text(), :!null, :unique;
        add-column 'height', integer(), :!null;
    }
}

migration 'Add countries', {
    create-table 'countries', {
        add-column 'id', integer(), :increments, :primary;
        add-column 'name', varchar(255), :!null, :unique;
    }

    alter-table 'skycrapers',{
        add-column 'country', integer();
        foreign-key table => 'countries', from => 'country', to => 'id';
    }
}
```

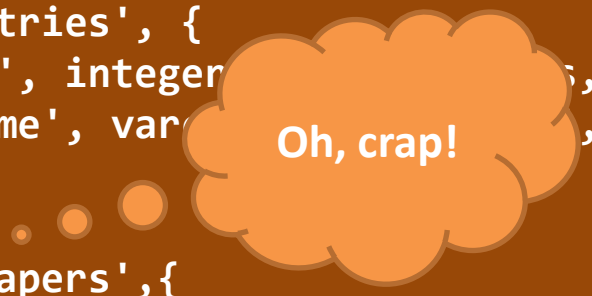
Add further migrations as needed

```
use DB::Migration::Declare;

migration 'Setup', {
    create-table 'skyscrapers', {
        add-column 'id', integer(), :increments, :primary;
        add-column 'name', text(), :!null, :unique;
        add-column 'height', integer(), :!null;
    }
}

migration 'Add countries', {
    create-table 'countries', {
        add-column 'id', integer
        add-column 'name', var
    }

    alter-table 'skycrapers',{
        add-column 'country', integer();
        foreign-key table => 'countries', from => 'country', to => 'id';
    }
}
```



Add further migrations as needed

```
use DB::Migration::Declare;

migration 'Setup', {
    create-table 'skyscrapers', {
        add-column 'id', integer(), :increments, :primary;
        add-column 'name', text(), :!null, :unique;
        add-column 'height', integer(), :!null;
    }
}

migration 'Add countries', {
    create-table 'countries', {
        add-column 'id', inte
        add-column 'name', va
    }

    alter-table 'skycrapers',{
        add-column 'country', integer();
        foreign-key table => 'countries', from => 'country', to => 'id';
    }
}
```

<demo>

Planned features

Automatically calculate down migrations
(for now, it only does up ones)

Support a wider range of database features (views, SPs) and alterations

A CLI (and perhaps Comma integration) for checking what is applied and performing application

Configurable data retention on lossy changes
(copy data in column being dropped to a backup table)

Just Do It With SQL, without inline SQL among the Raku?

Just Do It With SQL, without inline SQL among the Raku?

Badger

Mixed feelings

Used various ORMs
(both well-supported ones and client's homegrown ones)

Also various SQL generators

Generally OK at making easy things easier

Less good at hard things possible
(often end up reaching for the escape hatches)

Few projects need database abstraction

Just write SQL!

But inline SQL in code is

UGLY!!!

Just write SQL!

Then a former $dayjob colleague pointed out a compelling alternative!

The Clojure HugSQL library lets one write a SQL file, with comments that result in function definitions

Those can then be called as normal functions

Then a forn ue

pointed out a tive!

Please could you do a Raku port?

The Clojure HugSQL library lets one write a SQL file, with comments that result in function definitions

Those can then be called as normal functions

It's just a SQL file...

```
-- sub add-skyscraper(Str $name, Int $height, Int $country-id)
insert into skyscrapers (name, height, country)
values ($name, $height, $country-id);
```

It's just a SQL file...

```
-- sub add-skyscraper(Str $name, Int $height, Int $country-id)
insert into skyscrapers (name, height, country)
values ($name, $height, $country-id);
```

With comments containing Raku sub declarations

That is then used...

```
use Badger <sql/queries.sql>;

my $db = DB::Pg.new(conninfo => %*ENV<DB_CONNINFO>;
add-skyscraper($db, 'The Shard', 310, 42);
```

<demo>

Tests that hit the database without setup hassle?

Tests that hit the database without setup hassle?

Test::ContainerizedService

It can make sense to mock the database in tests

But database queries can be complex beasts, with plenty to wrong

Desirable to cover those with tests

But the setup work can be annoying!

Test::ContainerizedService to the rescue!

```
use DB::Pg;
use Test;
use Test::ContainerizedService;

test-service 'postgres', :tag<13.0>, -> (:$conninfo, *%) {
    my $conn = DB::Pg.new(:$conninfo);

    # Now you have a fresh database and a connection to it
}
```

Test::ContainerizedService to the rescue!

```
use DB::Pg;
use Test;
use Test::ContainerizedService;

test-service 'postgres', :tag<13.0>, -> (:$conninfo, *%) {
    my $conn = DB::Pg.new(:$conninfo);

    # Now you have a fresh database and a connection to it
}
```

Tests skipped if no docker or other setup issues!

<demo>

Make it easier to investigate DB issues?

Make it easier to investigate DB issues?

Dev::ContainerizedService

By default, we get a clean state every time with `Dev::ContainerizedService`

But what if we want a development database that sticks around?

Specify a project name, and that we should store service state

```
#!/usr/bin/env raku
use Dev::ContainerizedService;

project 'my-cool-app';
store;

service 'postgres', :tag<13.0>, -> (:$conninfo, *%) {
    env 'DB_CONNINFO', $conninfo;
}
```

<demo>

PRs welcome!

So far, all are focused on Postgres
(because that's what I'm using)

However, all are extensible
(if you're using something else)

Questions?

Not right now!

I recorded this for Raku Conference in advance and am now on vacation ☺

But contact info is coming up!

Thank you

@ jonathan@edument.cz

W jnthn.net

 jnthnwrthngtn

 jnthn