

My Overall objectives:

1. Separate frontend from backend development as cleanly, completely and simply as possible.
2. Minimize pain points and get rid of ridiculous abstractions.
3. Make refactoring easy and as painless as possible.
4. Popularize and demonstrate those things that make sense.

Demo:

Ingredients:

```
Elm (frontend)
postgrest (A remote data request API: warp middleware written in Haskell)
postgresql (backend Db)
```

1. Start with Haskell middleware [postgrest API](#)
2. explain basic steps in the Db and warp app config
3. start postgresSQL
4. explain Db permissions and postgres setup

```
create schema, table(s), view(s), SPs, etc.
grant usage (require login - in my case no login - anonymous with JWT)
grant privileges (in my case, only select)
```

```
configure postgrest by creating postgrest.conf
user specified in the db-uri is also known as the authenticator role
supports JWT, OAuth, CORs, Proxy, connection pooling, EKG monitoring
```

```
default output format is JSON, but also outputs on request
- text/csv
- openapi+json
- octet-stream
```

```
db-uri = "postgres://user:pass@host:5432/dbname"
db-schema = "api"
db-anon-role = "web_anon"
```

5. start postgREST

```
postgrest ~/Documents/haskell-work/postgrest/postgrest.conf
```

6. explain [Swagger - OpenAPI](#)

highlight the two terms using find in page: players and todos

The OpenAPI Specification creates the RESTful contract for your API, detailing all of its resources and operations in a human and machine readable format for easy development, discovery, and integration.

- show

```
curl http://localhost:3000/players
http://localhost:3000/todos
http://localhost:3000/players
http://localhost:3000/players?level=gte.7&order=level.desc
```

7. postgrest code projects on github (some are old, some deprecated)

```
Count Languages
35 JavaScript
21 PLpgSQL
18 Shell
8 Haskell
7 Elm
7 Python
6 Java
5 C#
5 Go
5 Ruby
```

8. Explain elm-tutorial-app

- why chosen
 - Uses the best Elm abstractions (modules) for the purpose
 - Easy for you to do what I did to get the feel
 - Demo: refactoring the tutorial to use postgrest and postgresql instead of fake (manually edited JSON) data and node-based JS API server took 20 minutes.

from the cloned root directly yarn start

[elm-tutorial-app-using-PostgREST-API/src/Commands.elm](#)

```

module Commands exposing (..)

import Http
import Json.Decode as Decode
import Json.Decode.Pipeline exposing (decode, required)
import Json.Encode as Encode
import Models exposing (Player, PlayerId)
import Msgs exposing (Msg)
import RemoteData

fetchPlayers : Cmd Msg
fetchPlayers =
    Http.get fetchPlayersUrl playersDecoder
        |> RemoteData.sendRequest
        |> Cmd.map Msgs.OnFetchPlayers

fetchPlayersUrl : String
fetchPlayersUrl =
    "http://localhost:3000/players?order=level.desc,name.asc"

```

[elm-tutorial-app-using-PostgREST-API/src/Update.elm](#)

```

module Update exposing (..)

import Commands exposing (savePlayerCmd)
import Models exposing (Model, Player)
import Msgs exposing (Msg)
import Routing exposing (parseLocation)
import RemoteData

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Msgs.OnFetchPlayers response ->
            ( { model | players = response }, Cmd.none )
        . . .

```

[elm-tutorial-app-using-PostgREST-API/src/Players/List.Elm](#)

```

module Players.List exposing (..)

import Html exposing (..)
import Html.Attributes exposing (class, href, style)
import Models exposing (Player)
import Msgs exposing (Msg)
import RemoteData exposing (WebData)
import Routing exposing (playerPath)

```

```
view : WebData (List Player) -> Html Msg
view response =
  div []
    [ nav
      , maybeList response
      , showQuery
      ]
  . . .

maybeList : WebData (List Player) -> Html Msg
maybeList response =
  case response of
    RemoteData.NotAsked ->
      text ""

    RemoteData.Loading ->
      text "Loading..."

    RemoteData.Success players ->
      list players

    RemoteData.Failure error ->
      text (toString error)
```

postgREST is a REST API for PostgreSQL written in Haskell

postgREST query results from PostgreSQL:

```
{ todos = [{ task = "download, configure and run postgREST server", done = True, due = "2018-02-12T13:00:00-06:00" }, { task = "pat self on back", done = True, due = "2018-02-12T13:05:00-06:00" }, { task = "prototype elm-postgrest", done = True, due = "2018-02-12T14:00:00-06:00" }, { task = "figure out elm-postgrest conditionals", done = True, due = "2018-02-13T07:00:00-06:00" }, { task = "figure out elm-postgrest filter on Boolean column", done = True, due = "2018-02-13T09:00:00-06:00" }, { task = "add stylish-elephants for layout", done = True, due = "2018-02-21T09:00:00-06:00" } ] }
```

Inomplete Tasks	Due
Review JWT AuthN in postgREST	2018-03-01T15:00:00-06:00
watch elm-postgREST dev branch	2018-04-01T16:00:00-05:00

Layout without CSS

COLUMN

Paragraph

postgREST is a REST API for PostgreSQL written in Haskell

Row


postgREST query results from PostgreSQL:


Paragraph

```
{ todos = [{ task = "download, configure and run postgREST
server", done = True, due = "2018-02-12T13:00:00-06:00" }, { task
= "pat self on back", done = True, due = "2018-02-12T13:05:00-
06:00" }, { task = "prototype elm-postgrest", done = True, due =
"2018-02-12T14:00:00-06:00" }, { task = "figure out elm-postgrest
conditionals", done = True, due = "2018-02-13T07:00:00-06:00"
}, { task = "figure out elm-postgrest filter on Boolean column",
done = True, due = "2018-02-13T09:00:00-06:00" }, { task = "add
stylish-elephants for layout", done = True, due = "2018-02-
```

Paragraph + Row + Indexed Table

Inomplete Tasks	Due
Review JWT AuthN in postgREST	2018-03-01T15:00:00-06:00
watch elm-postgREST dev branch	2018-04-01T16:00:00-05:00

 = padding

 = spacing

[jbrgfx.github.home](https://jbrgfx.github.io/home)[jbrgfx github io/src/Main.elm](https://jbrgfx.github.io/src/Main.elm)

```

module Main exposing (..)

import Color exposing (black, darkBlue, lightGrey, white)
import Element exposing (Element, alignBottom, alignLeft, centerY, column, height, image,
layout, newTabLink, padding, paddingEach, paragraph, px, row, text, width)
import Element.Background as Background
import Element.Border as Border
import Element.Events as Events
import Element.Font as Font
import Element.Input as Input
import Html

view model =
    Element.layout
        [ Background.color white
        , width (px 900)
        , paddingLeft gutter
        , Font.family
            [ Font.typeface "Open Sans"
            , Font.sansSerif
            ]
        ]
    <|
    column
        []
        [ headerArea
        , mainColumns
            { left =
                [ theAppDesc
                , overViewDesc
                ]
            , right =
                [ row [ padding gutter ] [ inputForm ]
                , validateFilter model
                , paragraph
                    []
                    [ Element.text "Results:" ]
                , paragraph
                    [ padding gutter
                    , Background.color lightGrey
                    , Font.size 16
                    ]
                    (List.map viewRepos model.filtered)
                ]
            }
        , footerArea
        ]

```

```
viewRepos entry =
```

```

paragraph
  []
  [ newTabLink
    [ padding gutter
      , Font.bold
      , Font.size 18
      , Font.underline
      , alignBottom
      , Font.color darkBlue
      , Background.mouseOverColor Color.darkBlue
      , Font.mouseOverColor Color.white
    ]
    { url = "https://github.com/jbrgfx/" ++ entry
      , label = Element.text entry
    }
  ]
]

```

Oracle Rest Data Services (ORDs)

Benefits of choosing Elm:

1. Just Elm code.
2. Strongly typed language means the compiler outputs informative errors.
3. Impossible states can be avoided by design.
4. Each possible state is type-checked.
5. In this example, even the layout is type-checked, and I wrote no CSS.
6. No run-time errors.
7. A virtual DOM has been included from the beginning; unlike other reactive frameworks, there is no intellectual overhead.
8. I can completely separate front-end code from middleware and from backend(s).

Challenges:

1. Elm is totally different from anything (except Haskell), so it has a steep learning curve especially if you do not yet know a typed, functional language.
2. Hard to begin thinking about web dev without a framework, without HTML, without CSS, without JS, without objects to work with directly and using a totally new layout philosophy.