# My Overall objectives:

1. Separate frontend from backend development as cleanly, completely and simply as possible.
2. Minimize pain points and get rid of complex or 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. Explain Db permissions and postgreSQL 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 postgrest output format is JSON, but also outputs on request
                  text/csv
                  openapi+json
                  octet-stream

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

4. Start postgREST in the root directory you untarred the binary

   ```
   postgrest postgrest.conf
   ```

5. Explain Swagger - OpenAPI

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

   ```
   You request via queries responses from postgreSQL through
   the postgrest endpoint using a web client or curl.
   ```

6. 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
```

7. The elm-tutorial-app:

```
- Uses powerful Elm packages with which you will want to become familiar
       if you work with JSON data in Elm;
- Is easy for you to do what I did to get the feel of postgrest;
- Refactoring the tutorial to use postgrest and postgresql instead of
       fake (manually edited JSON ) data and node-based JS API server that
       is in the original tutorial took 20 minutes.
```

# Examine some code

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)
```

So, getting type-safe data out of postgreSQL via postgrest and changing the state in the client code is straight-forward.

Rather than show you the client view code in the demo, I want introduce another way of eliminating pains points and reducing complexity -- that is, no framework and no CSS in Elm: just Elm code, all type-safe.

You may see this sort of mechanism soon in your favorite front-end language.

jbrgfx.github.home

view code in jbrgfx github io/src/Main.elm

```
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 also has a similar mechanism to negotiate rest API access to Oracle data called Oracle Rest Data Services (ORDs) Again, the privileges are created and stored in the RDBMS and the mid-tier app is easily configured. Then you build and drop a war file into your favorite servlet container and write then client.

## 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.