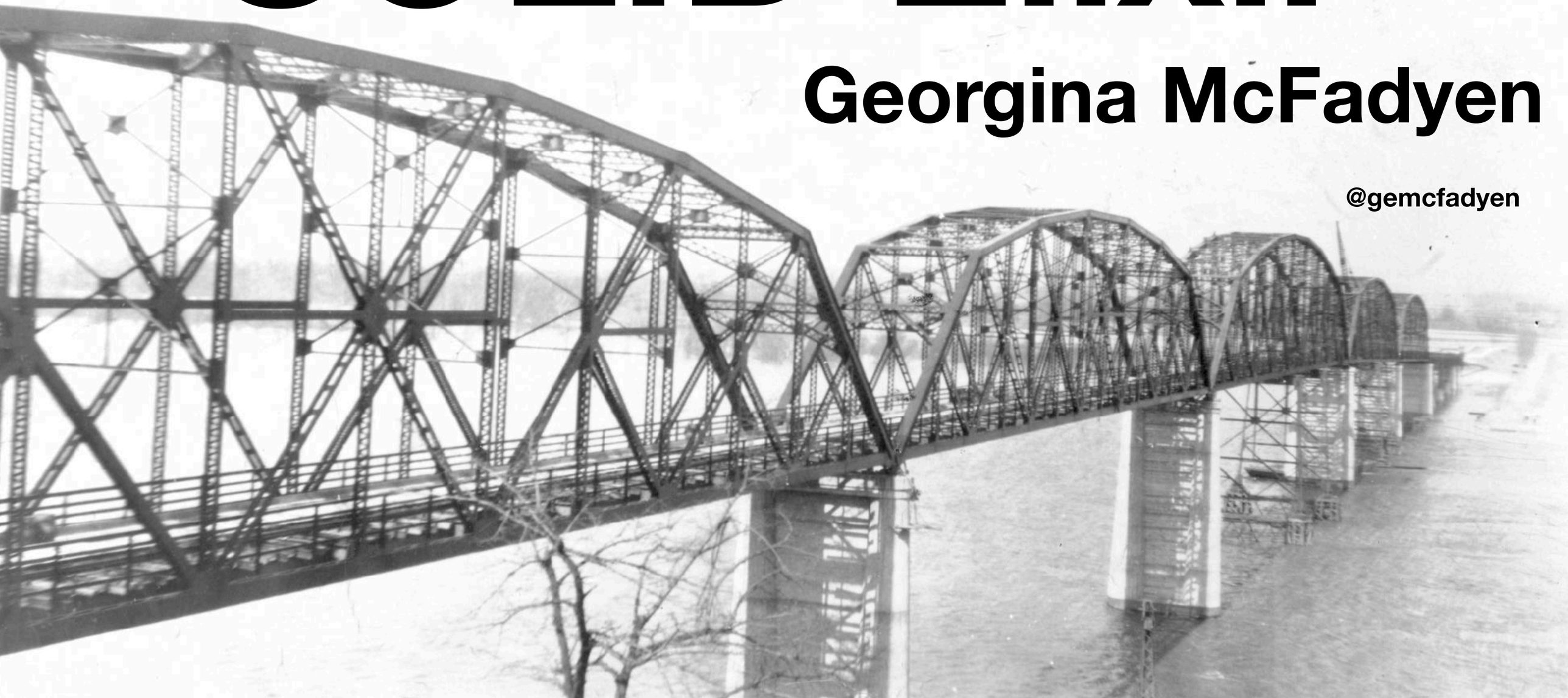
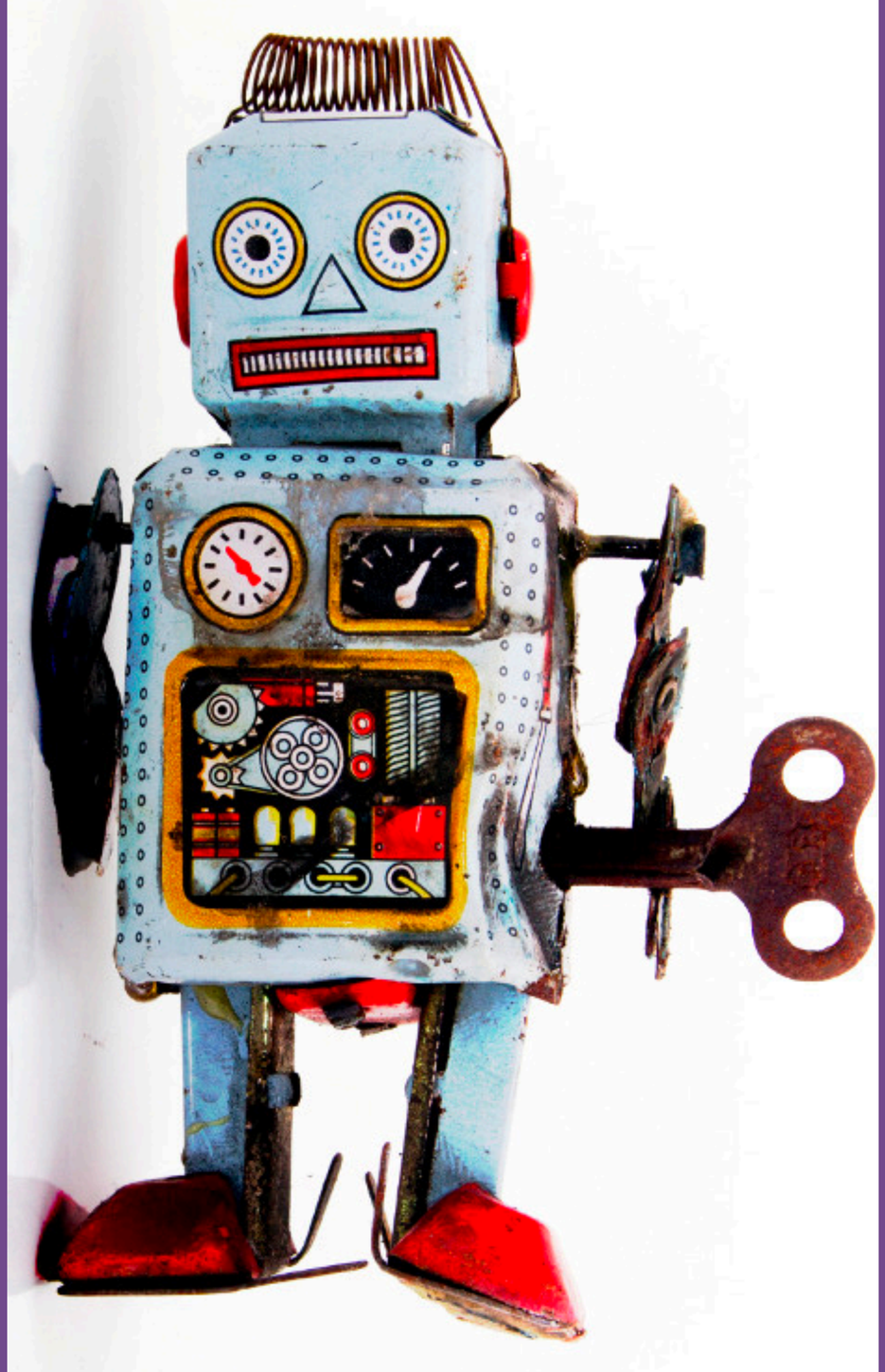
# SOLID Elixir

Georgina McFadyen

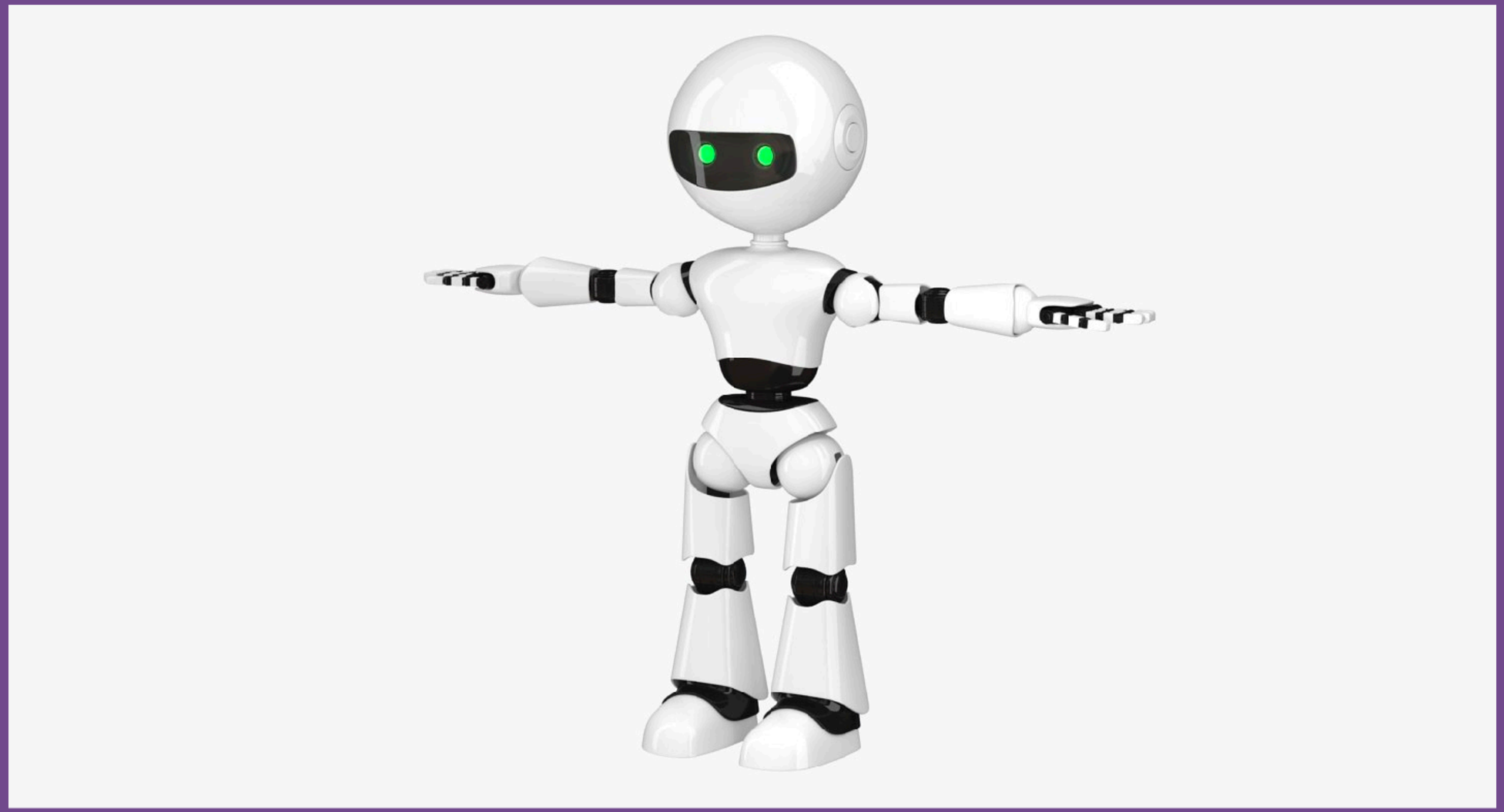
@gemcfadyen



# Hi, I’m Georgina



@gemcfadyen



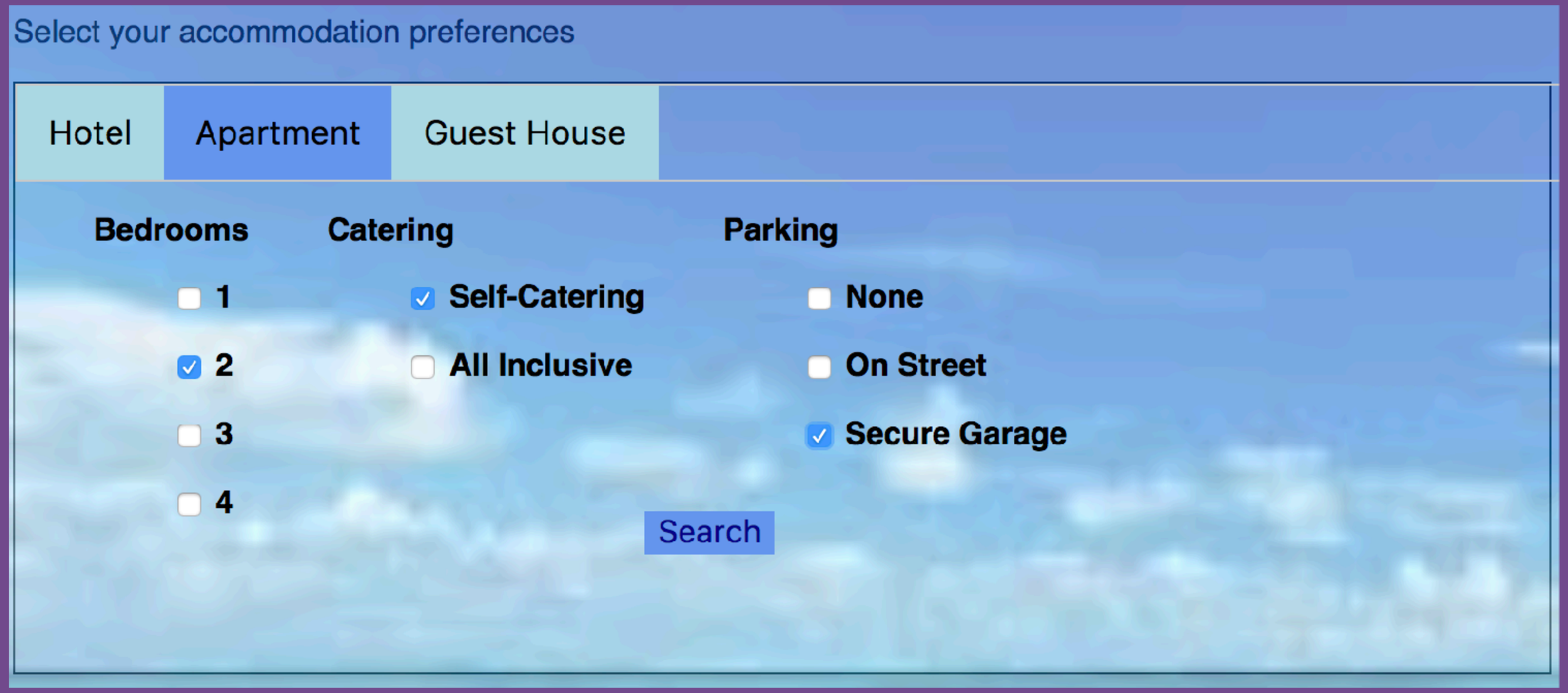
# What are the SOLID principles?

“In Object-Oriented programming, the term SOLID is a mnemonic acronym for five design principle intended to make software design more understandable flexible and maintainable .”

*– Wikipedia*
*https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)*

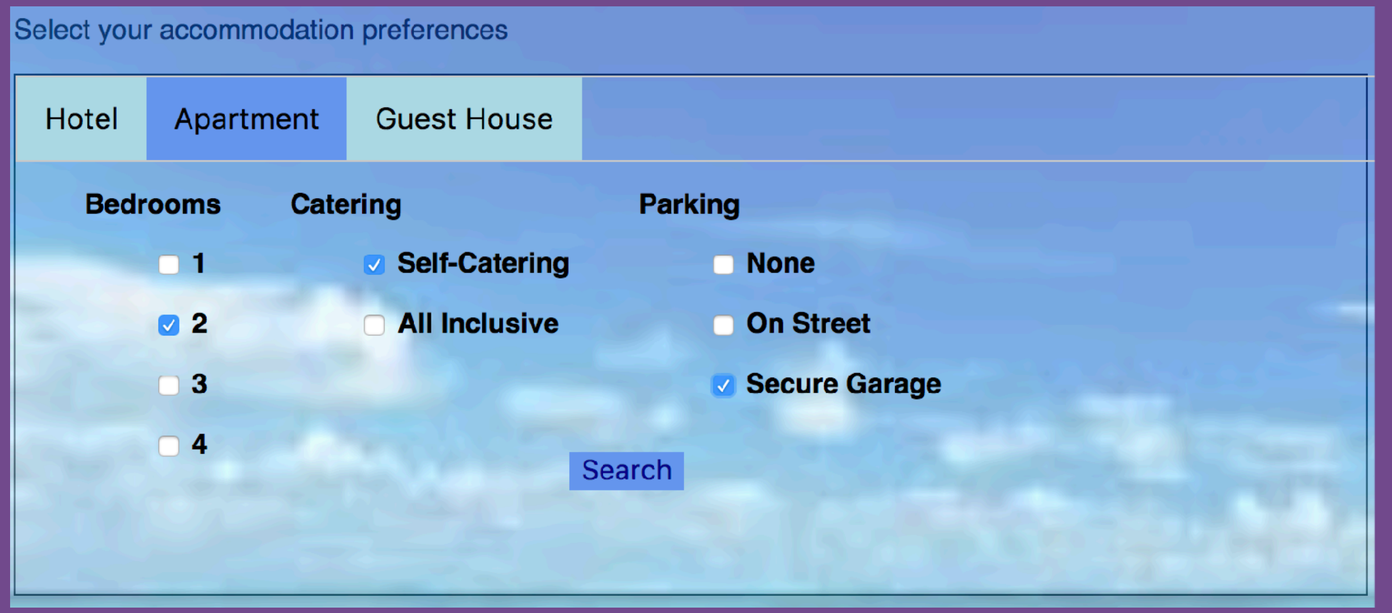
# Holiday Booking Domain

https://github.com/gemcfadyen/solid_elixir


Select your accommodation preferences
Hotel
Apartment
Guest House
Bedrooms
1
2
3
4
Catering
Self-Catering
All Inclusive
Parking
None
On Street
Secure Garage
Search

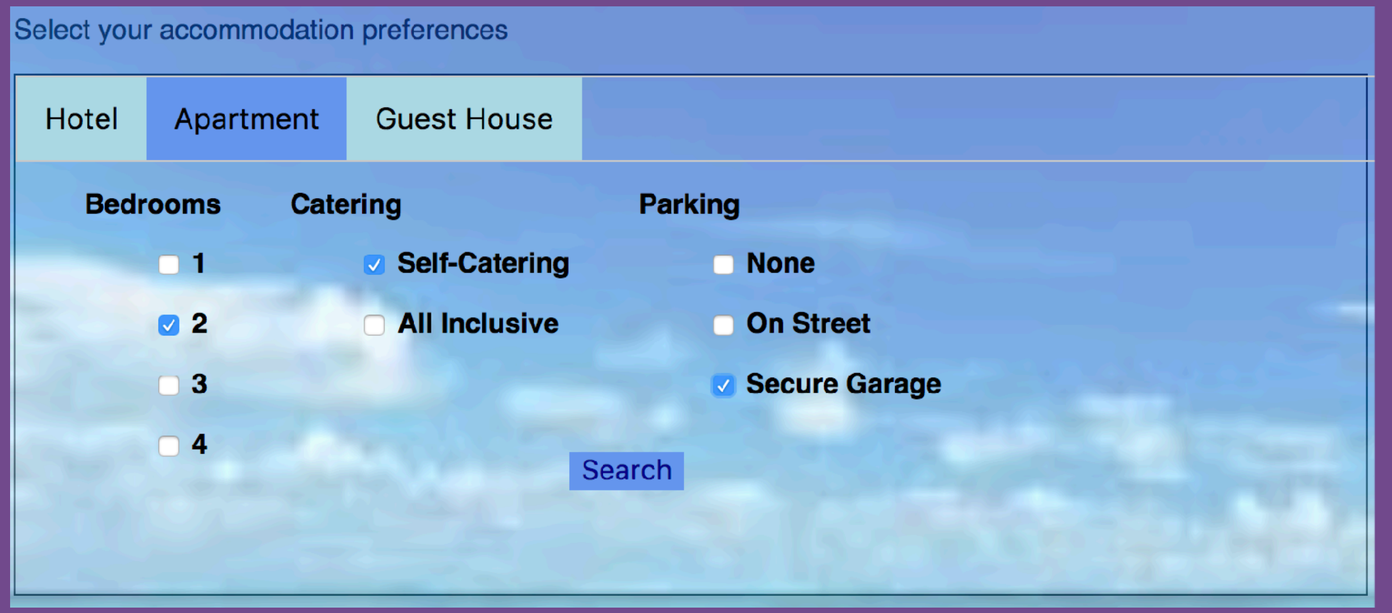
# GET users preferences

Login




# GET users preferences

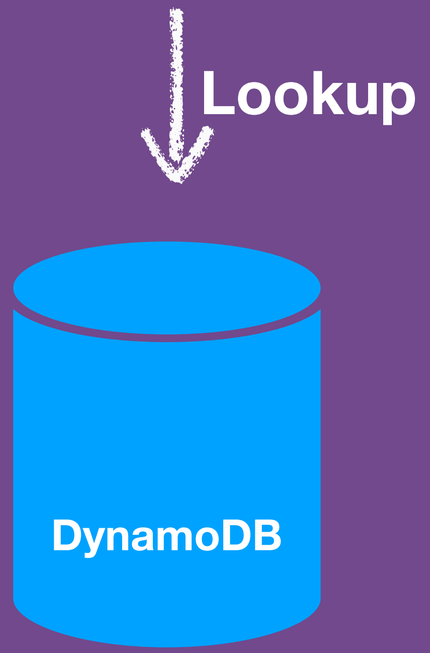
Login



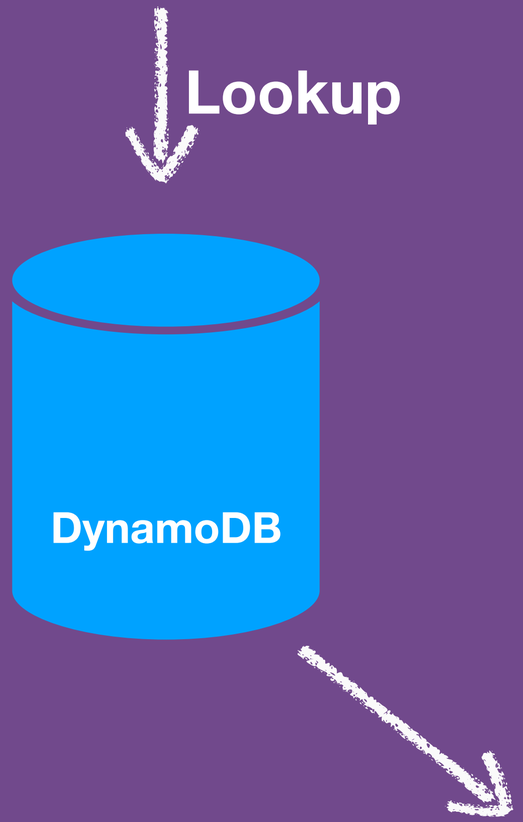

id of customer

# GET /customers/{id}

# GET /customers/{id}



# GET /customers/{id}




```
%{"id" => "uuid-1",
    "lastModified" => 1519928745635,
    "preferences" =>
        %{"accommodation" =>
            %{"apartment" => %{
                "catering" => "self_catering",
                "bedrooms" => 2,
                 "parking" => "secure"
                },
                "hotel" => %{
                    "catering" => "self_catering",
                    "bedrooms" => 1,
                    "parking" => "secure"
                 }
        }
    }
}
```

# GET /customers/{id}

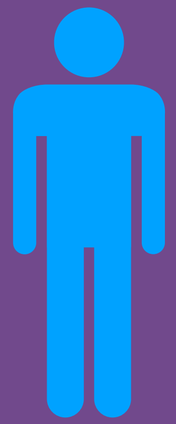
```
%{"id" => "uuid-1",
    "lastModified" => 1519928745635,
    "preferences" =>
        %{"accommodation" =>
            %{"apartment" => %{
              "catering" => "self_catering",
              "bedrooms" => 2,
               "parking" => "secure"
               },
              "hotel" => %{
                 "catering" => "self_catering",
                 "bedrooms" => 1,
                 "parking" => "secure"
               }
        }
    }
}
```

Internal Struct Representation

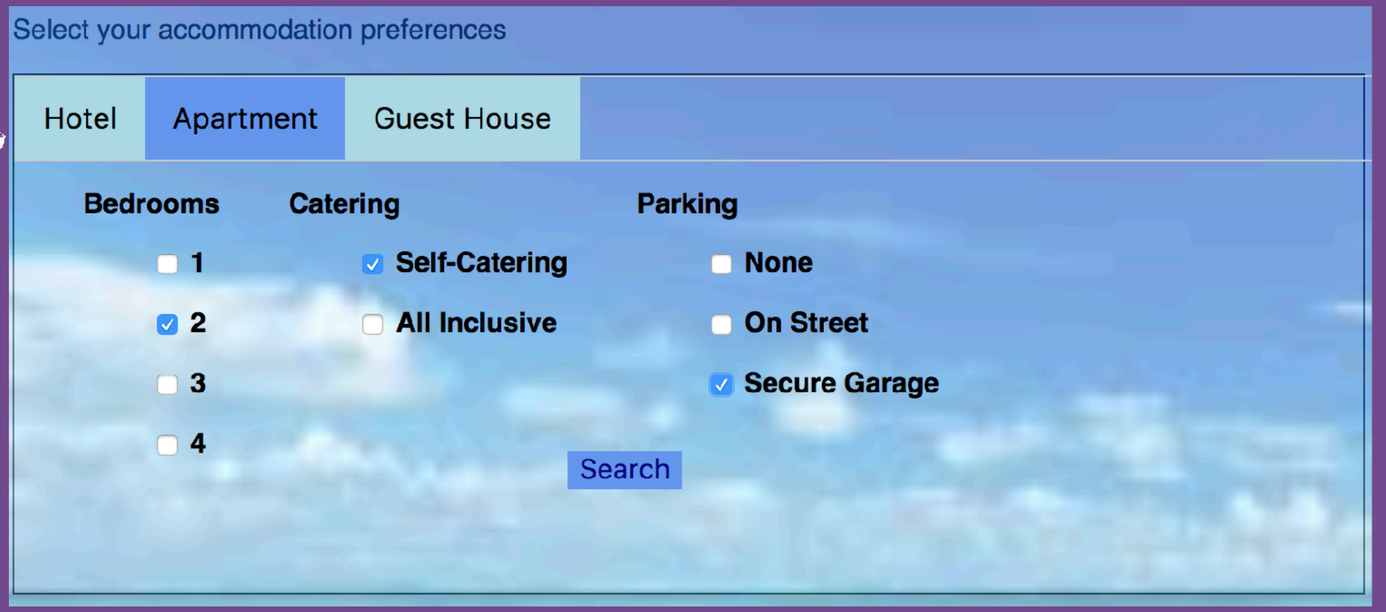
```
{
  "id":"uuid-1",
  "lastModified": "2018-02-21T12:05:15Z",
  "preferences": {
    "accommodation": {
        "apartment": {
              "catering": "self_catering",
              "bedrooms": 2,
               "parking": "secure"
               },
              "hotel": {
                 "catering": "self_catering",
                 "bedrooms": 1,
                 "parking": "secure"
               }
        }
    }
}
```

JSON Representation

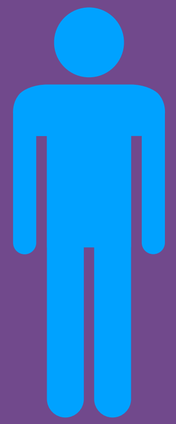
# POST users preferences



Login

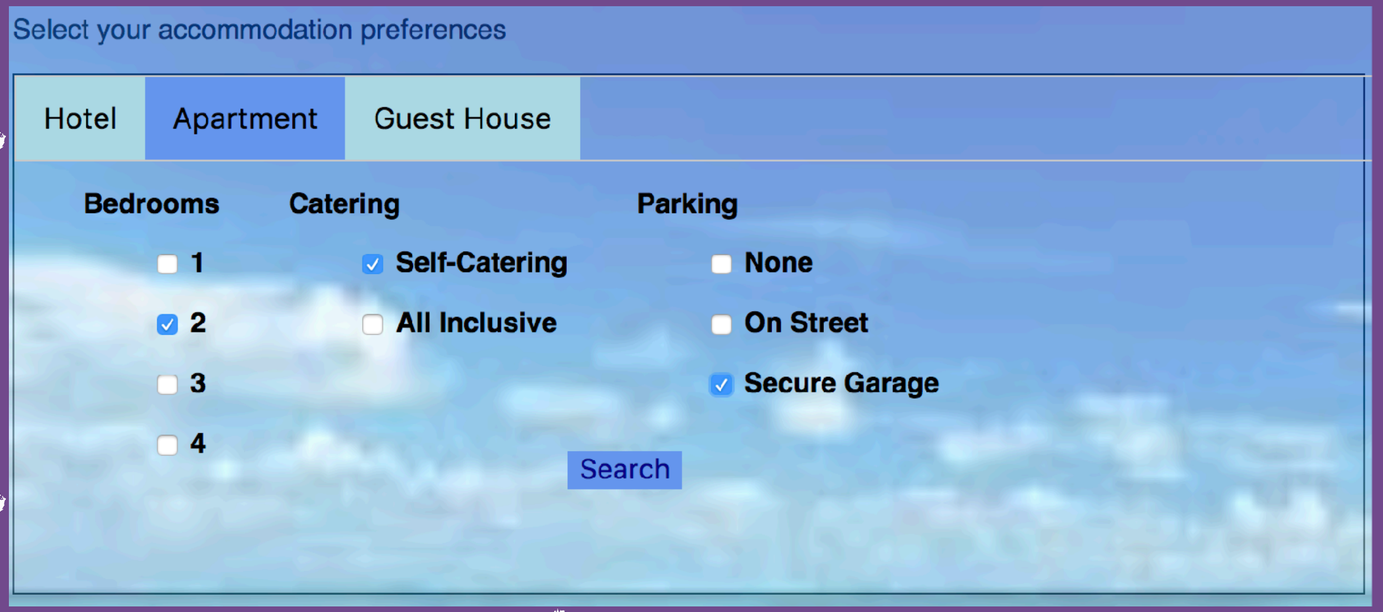



# POST /customers



Login

Search




# POST /customers

# POST /customers

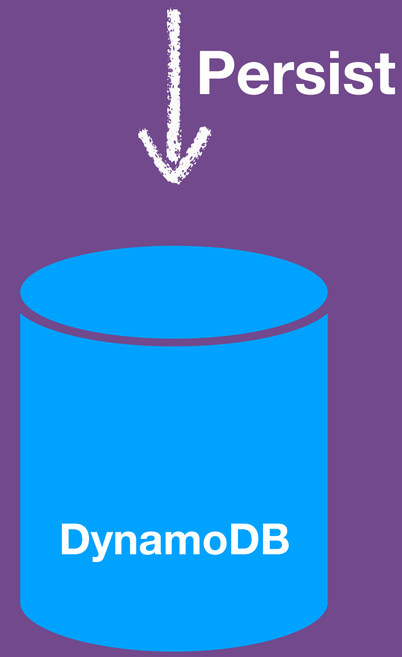
```
{
  "id":"uuid-1",
  "lastModified": "2018-02-21T12:05:15Z",
  "preferences": {
    "accommodation": {
       "apartment": {
             "catering": "self_catering",
             "bedrooms": 2,
              "parking": "secure"
              },
             "hotel": {
                "catering": "self_catering",
                "bedrooms": 1,
                "parking": "secure"
             }
       }
    }
}
```

JSON Representation

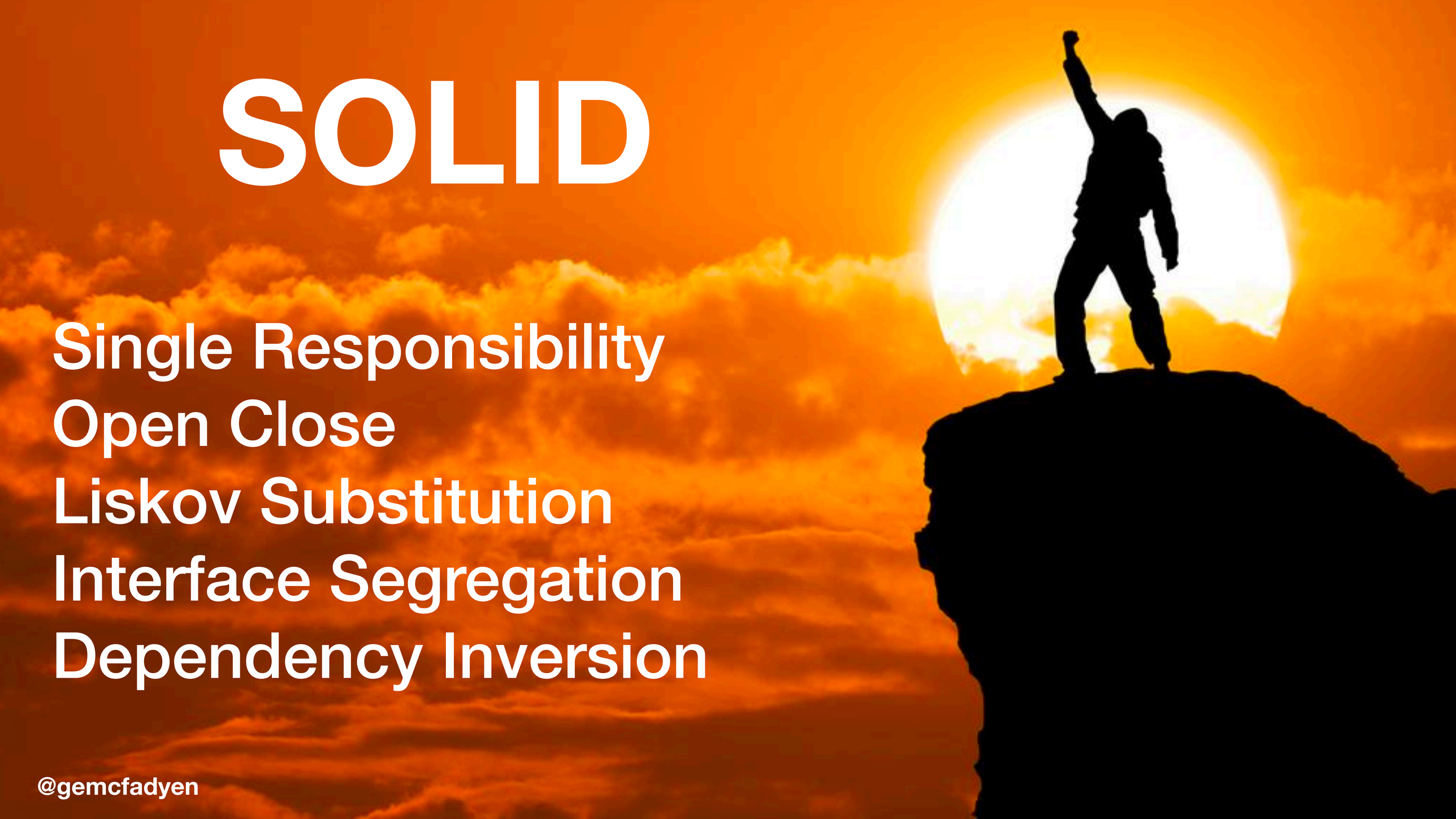
```
%{"id" => "uuid-1",
    "lastModified" => 1519928745635,
    "preferences" =>
       %{"accommodation" =>
           %{"apartment" => %{
             "catering" => "self_catering",
             "bedrooms" => 2,
              "parking" => "secure"
              },
             "hotel" => %{
                "catering" => "self_catering",
                "bedrooms" => 1,
                "parking" => "secure"
             }
       }
    }
}
```

Internal Struct Representation

# POST /customers



# SOLID



Single Responsibility
Open Close
Liskov Substitution
Interface Segregation
Dependency Inversion

@gemcfadyen


Single Responsibility
SOLID
@gemcfadyen

“The single responsibility principle states that every module or class should have responsibility over a single part of the functionality provided by the software...”

*– Wikipedia*
*https://en.wikipedia.org/wiki/Single_responsibility_principle*

```elixir
defmodule Controllers.PreferenceController do

  def get_preferences(customer_id) do
    ExAws.Dynamo.get_item("customer-preferences", %{id: customer_id})
    |> ExAws.request
    |> decode
  end
  defp decode({:ok, response}) when map_size(re
   {:error, :not_found}
  end
  defp decode({:ok, response}) do
   valid_consent_user = ExAws.Dynamo.decode_item(response,
                                as: Schema.Database.CustomerPreferencesRow)
   |> format_response()

   {:ok, valid_consent_user}
  end
  defp decode({:error, response}) do
   {:error, response}
  end

  def format_response(raw_response) do
   pretty_response = raw_response
   # Data formatting to create a nice response for calling system
   pretty_response
  end
end
```

Fetch User Preferences

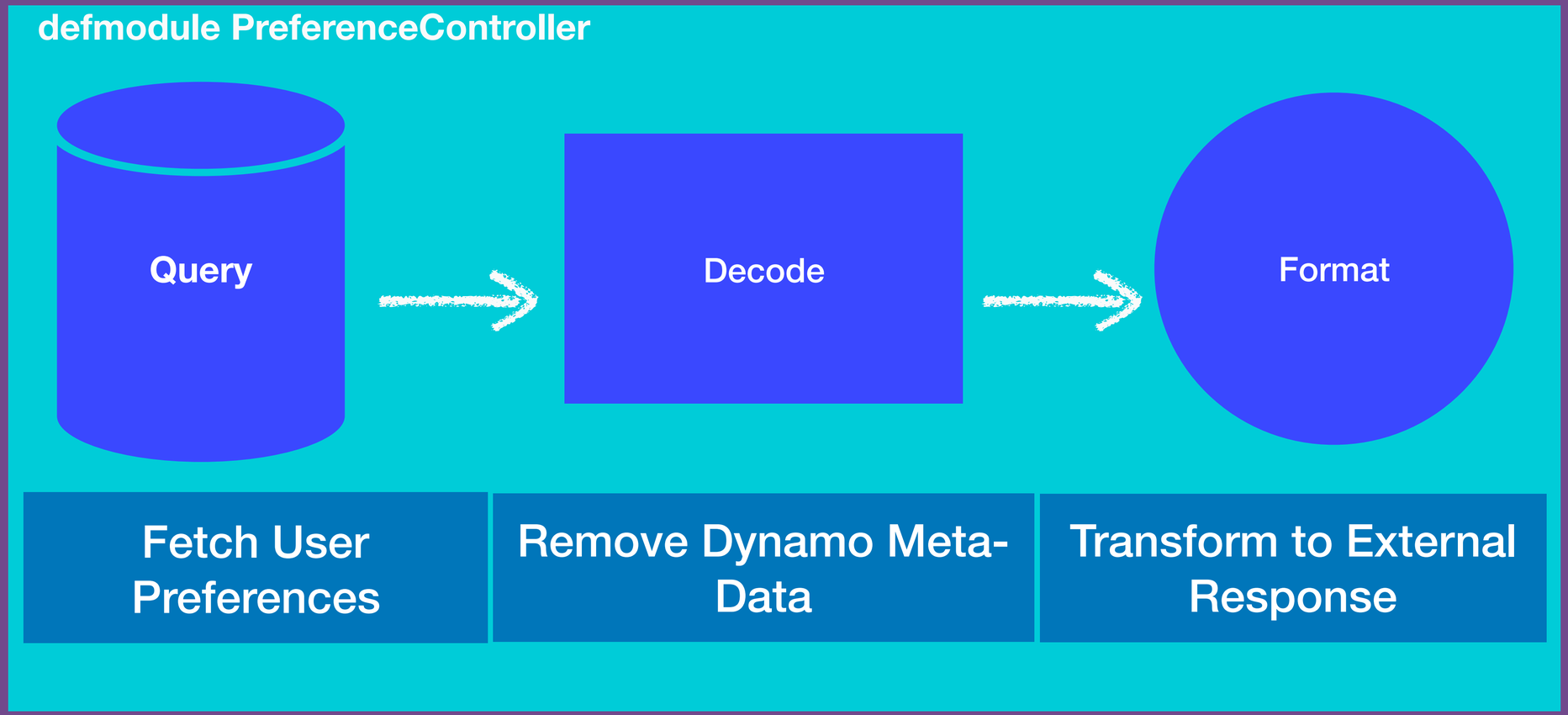
```elixir
defmodule Controllers.PreferenceController do

  def get_preferences(customer_id) do
    ExAws.Dynamo.get_item("customer-preferences", %{id: customer_id})
    |> ExAws.request
    |> decode
  end
  defp decode({:ok, response}) when map_size(response) == 0 do
    {:error, :not_found}
  end
  defp decode({:ok, response}) do
    valid_consent_user = ExAws.Dynamo.decode_item(response,
                                                  as: Schema.Database.CustomerPreferencesRow)
    |> format_response()

    {:ok, valid_consent_user}
  end
  defp decode({:error, response}) do
    {:error, response}
  end

  def format_response(raw_response) do
    pretty_response = raw_response
    # Data formatting to create a nice response for calling system
    pretty_response
  end
end
```

Remove Dynamo Meta-Data

```elixir
defmodule Controllers.PreferenceController do

  def get_preferences(customer_id) do
    ExAws.Dynamo.get_item("customer-preferences", %{id: customer_id})
    |> ExAws.request
    |> decode
  end
  defp decode({:ok, response}) when map_size(response) == 0 do
    {:error, :not_found}
  end
  defp decode({:ok, response}) do
    valid_consent_user = ExAws.Dynamo.decode_item(response,
                                  as: Schema.Database.CustomerPreferencesRow)
    |> format_response()

    {:ok, valid_consent_user}
  end
  defp decode({:error, response}) do
    {:error, response}
  end

  def format_response(raw_response) do
    pretty_response = raw_response
    # Data formatting to create a nice response for calling system
    pretty_response
  end
end
```

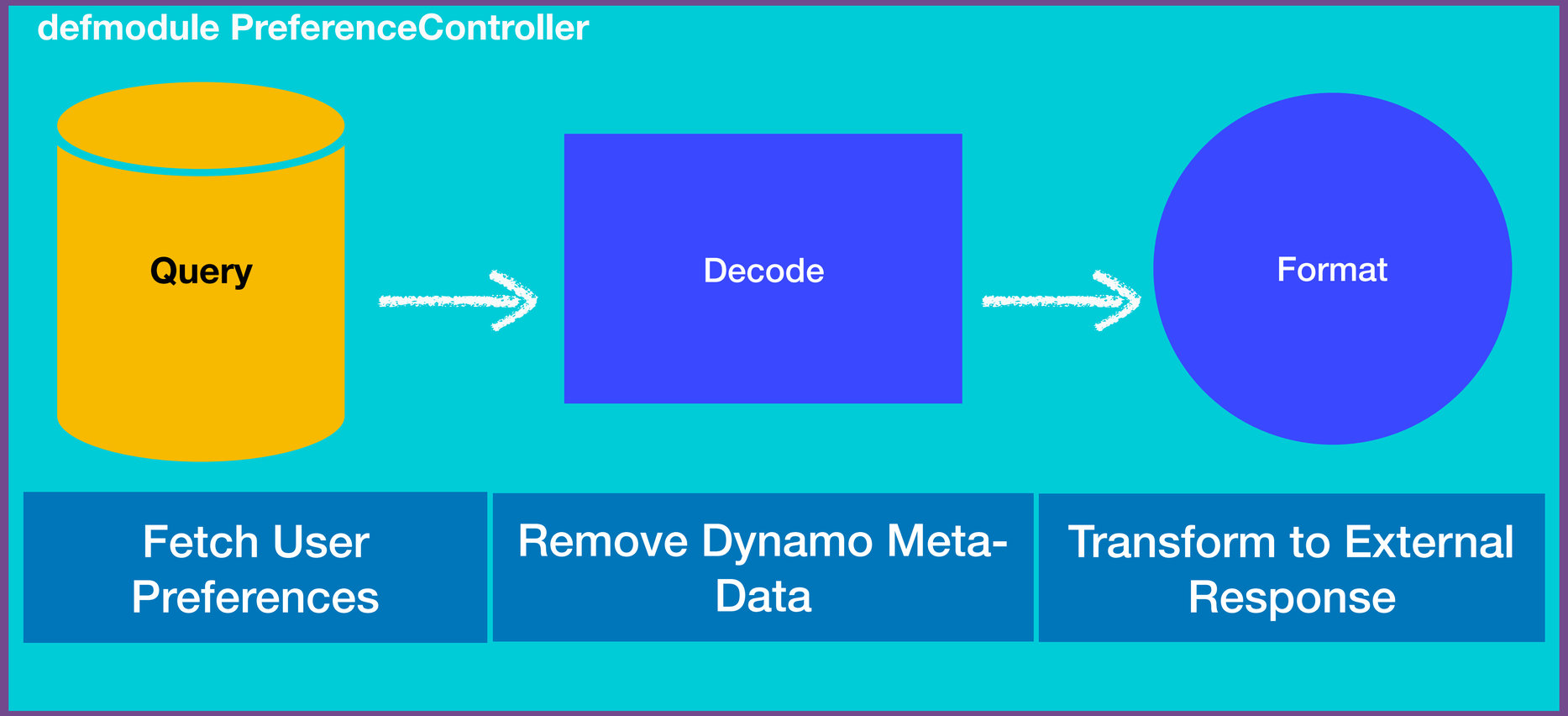
Transform to External Response

# GET
# Controller Responsibilities



# Can any of these steps change independently?

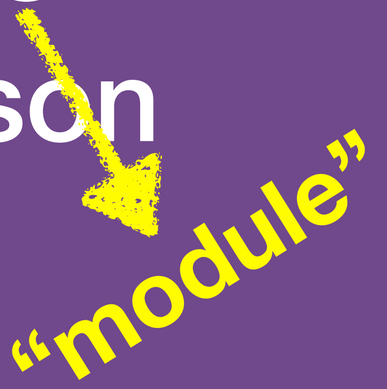
# What if we changed the conditions on which we query?



“Robert C. Martin expresses the principle as, "A class should have only one reason to change.”

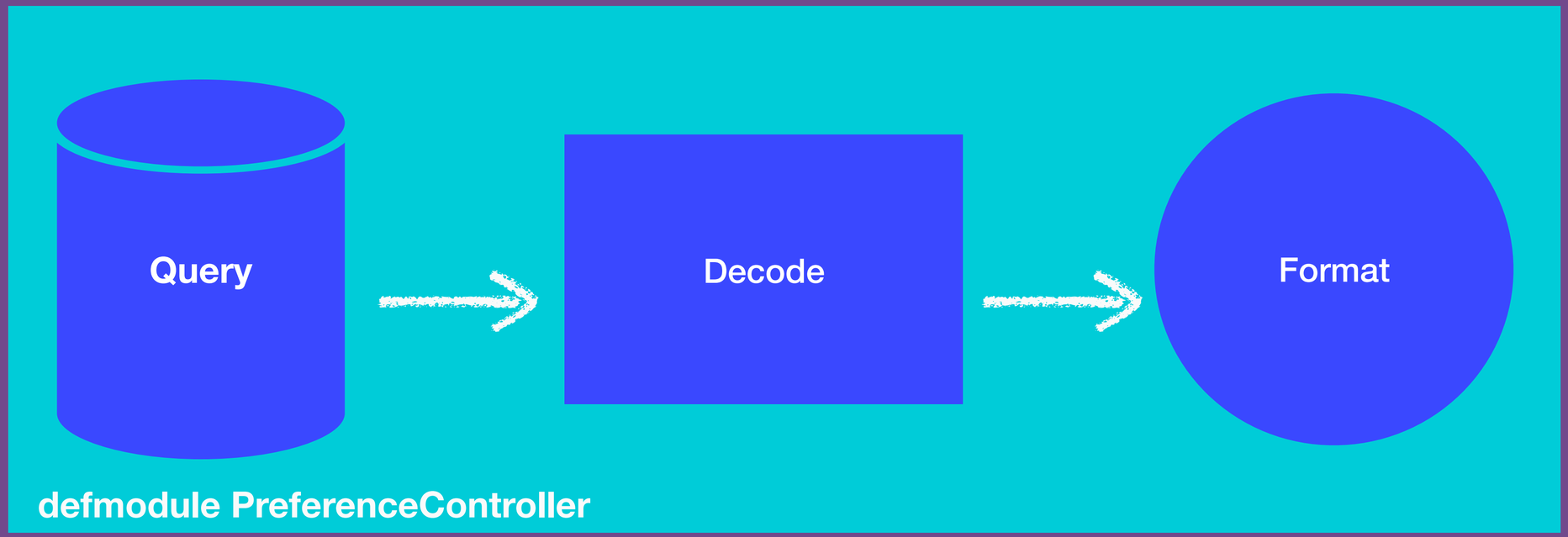
*– Wikipedia*
*https://en.wikipedia.org/wiki/Single_responsibility_principle*

“Robert C. Martin expresses the principle as, "A class should have only one reason to change.”




*– Wikipedia*
*https://en.wikipedia.org/wiki/Single_responsibility_principle*

# Controller Responsibilities



# Query Database

```
defmodule Core.Preferences.GetCustomerPreferences do

  def for(customer_id) do
   ExAws.Dynamo.get_item("customer-preferences",
                         %{id: customer_id})
   |> ExAws.request
  end
end
```

# Decode Database Response

```elixir
defmodule Core.Preferences.Decode do

  def decode({:ok, response}) when map_size(response) == 0 do
   {:error, :not_found}
  end
  defp decode({:ok, response}) do
   valid_preferences = ExAws.Dynamo.decode_item(response,
                       as: Schema.Database.CustomerPreferencesRow)

   {:ok, valid_prefereneces}
  end
  defp decode({:error, response}) do
   {:error, response}
  end
end
```

Decode

# Decode Database Response

```elixir
defmodule Core.Preferences.Decode do

  def decode({:ok, response}) when map_size(response) == 0 do
    {:error, :not_found}
  end
  defp decode({:ok, response}) do
    valid_preferences = ExAws.Dynamo.decode_item(response,
                        as: Schema.Database.CustomerPreferencesRow)
    {:ok, valid_prefereneces}
  end
  defp decode({:error, response}) do
    {:error, response}
  end
end
```

# Format to Json Response

```elixir
defmodule Core.Schema.External.Response do
  def format({:ok, database_response}) do
    database_response
    |> format_dates()
    |> group_accomodation_preferences()
  end

  defp format_dates(raw_data) do
    ..
  end

  defp group_accomodation_preferences(raw_data) do
    ...
  end
end
```

Format

# Controller

```
1  defmodule Controllers.PreferenceController do
2
3    def get_preferences(customer_id) do
4     Core.Preferences.GetCustomerPreferences.for(customer_id)
5     |> Core.Schema.External.Response.format()
6    end
7  end
```

# Benefits of Single Responsibility

## SOLID

- ✅ Focused unit tests
- ✅ Fewer higher level tests
- ✅ Less dependencies per test
- ✅ Similar functionality grouped

# Trade Off

❌ Ensure each module justifies it’s existence


Open Close
SOLID
@gemcfadyen

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification…An entity can allow its behaviour to be extended without modifying its source code”

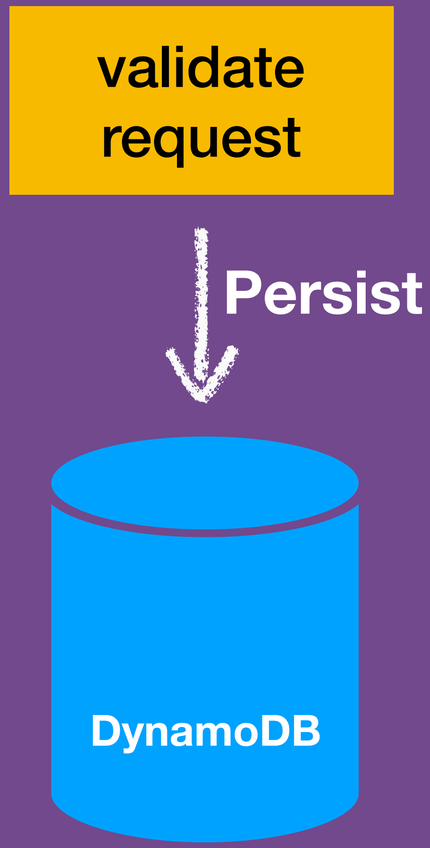
*– Wikipedia*
*https://en.wikipedia.org/wiki/Open/closed_principle*

# Open Closed

Ability to add new code without modifying any existing code.

# POST /customers


# POST /customers



# Request Validation

? Is query parameter a valid uuid?

? Does body contain preferences section?

? Does body contain accommodation section?

```elixir
defmodule Plugs.SaveCustomer do
  import Plug.Conn

  def init(opts) do
    opts
  end

  def call(conn, _opts) do
    ...
    with {:ok, customer_id} <-
           Validation.RequestParameterValidation
                     .has_valid_parameters(conn.path_params),
         :ok <-
           Validation.PreferencesValidation
                               .has_preferences(body),
         :ok <-
           Validation.AccommodationValidation
                     .has_accommodation_preferences(body) do

      ...

    end
  end
end
```

validate different parts of request

```
defmodule Plugs.SaveCustomer do
  import Plug.Conn

  def init(opts) do
   opts
  end

  def call(conn, _opts) do
   body = conn.body_params

   with {:ok, customer_id} <-
     Validation.RequestParameterValidation.has_valid_parameters(conn.path_params),
         :ok <-
           Validation.PreferencesValidation.has_preferences(body),
         :ok <-
          Validation.AccommodationValidation.has_accommodation_preferences(b
```

Save Preferences

```
valid_request = Schema.Http.Request.new!(body,
                                         customer_id,
                                         body["version"])

response =
   Controllers.SavePreferenceController.save_preferences(customer_id,
                                                         valid_request)

send_resp(conn, 201, Poison.encode!(response))
```

```
   end
  end
end
```

```elixir
defmodule Plugs.SaveCustomer do
  import Plug.Conn

  def init(opts) do
   opts
  end

  def call(conn, _opts) do
   body = conn.body_params

   with {:ok, customer_id} <-
     Validation.RequestParameterValidation.has_valid_parameters(conn.path_params),
         :ok <-
```

## Handle Errors

```elixir
else
  {:error, :invalid_url_params} ->
     send_resp(conn, 400, Poison.encode!(%{error:
                  "URL Parameters need to be alpha numeric"}))
  {:error, :no_preferences} ->
     send_resp(conn, 400, Poison.encode!(%{error:
                   "Preferences section missing"}))
  {:error, :no_accommodation}  ->
     send_resp(conn, 400, Poison.encode!(%{error:
                    "Accommodation section missing"}))
end
```

```elixir
defmodule Plugs.SaveCustomer do
  import Plug.Conn

  def init(opts) do
   opts
  end

  def call(conn, _opts) do
   body = conn.body_params

   with {:ok, customer_id} <-
     Validation.Re                    dation.has_valid_parameters(conn.path_params),
          Validation.PreferencesValidation.has_preferences(body),
          :ok <-
           Validation.AccommodationValidation.has_accommodation_preferences(body) do

          valid_request = Schema.Http.Request.new!(body, customer_id, body["version"])
          response =
             Controllers.SavePreferenceController.save_preferences(customer_id, valid_request)
          send_resp(conn, 201, Poison.encode!(response))
   else
     {:error, :invalid_url_params} ->
        send_resp(conn, 400, Poison.encode!(%{error: "URL Parameters need to be alpha numeric"}))
     {:error, :no_preferences} ->
        send_resp(conn, 400, Poison.encode!(%{error: "Preferences section missing"}))
     {:error, :no_accommodation}  ->
        send_resp(conn, 400, Poison.encode!(%{error: "Accommodation section missing"}))
   end
  end
end
```

Add new modules which checks the new rule

```elixir
defmodule Plugs.SaveCustomer do
  import Plug.Conn

  def init(opts) do
   opts
  end

  def call(conn, _opts) do
   body = conn.body_params

   with {:ok, customer_id} <-
     Validation.Re                    dation.has_valid_parameters(conn.path_params),
                    alidation.PreferencesValidation.has_preferences(body),
          :ok <-
           Validation.AccommodationValidation.has_accommodation_preferences(body) do

          valid_request = Schema.Http.Request.new!(body, customer_id, body["version"])
          response =
             Controllers.SavePreferenceController.save_preferences(customer_id, valid_request)
          send_resp(conn, 201, Poison.encode!(response))
   else
     {:error, :invalid_url_params} ->
        send_resp(conn, 400, Poison.encode!(%{error: "URL Parameters need to be alpha numeric"}))
     {:error, :no_preferences} ->
        send_resp(conn, 400, Poison.encode!(%{error: "Preferences section missing"}))
     {:error, :no_accommodation}  ->
        send_resp(conn, 400, Poison.encode!(%{error: "Accommodation section m
   end
  end
end
```

Add new modules which checks the new rule

Handle error case


@gemcfadyen

- Don’t want to open up the Plug module each time a new validation rule is added
- Don’t want to have to test all the rule permutations at this high level

- Do want the ability to plug in new validation rules
- Do want to configure one rule for test env
- Do want to configure all rules for dev/prod env

# Categorise Validations

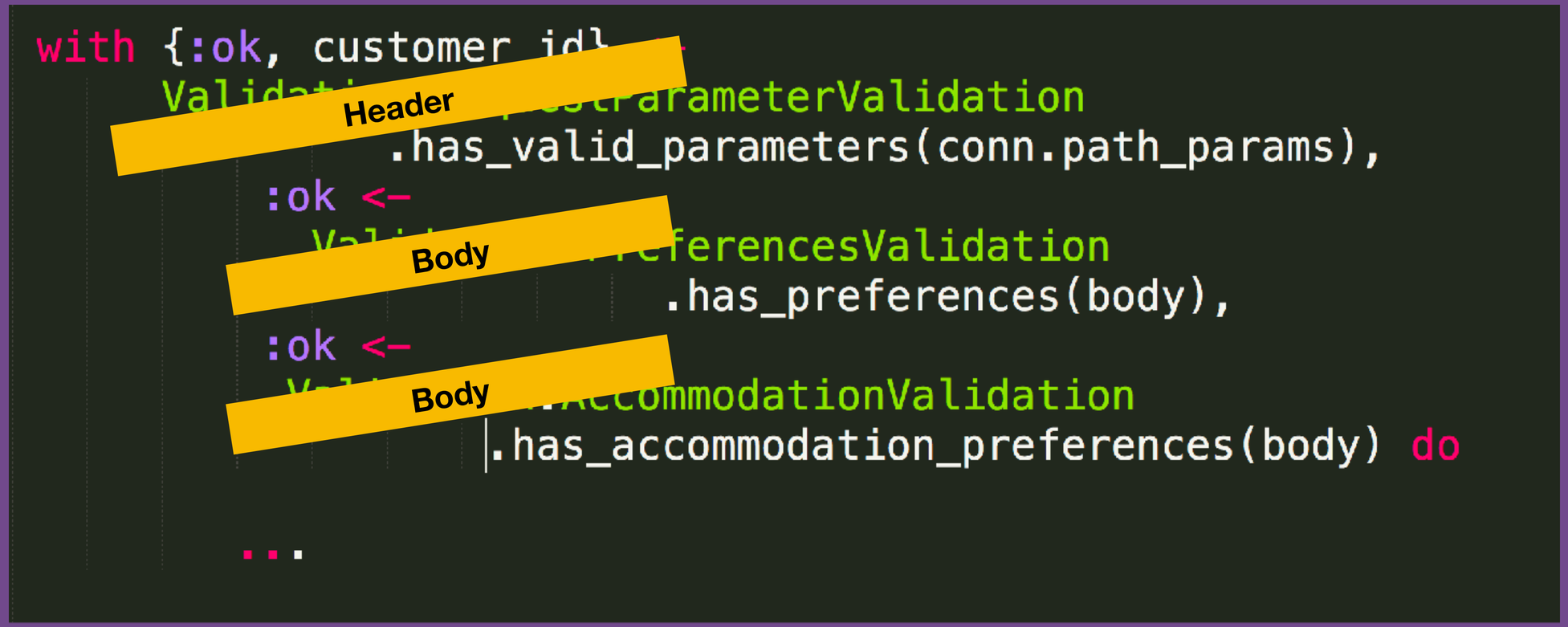
```
with {:ok, customer_id} <-
       Validation.RequestParameterValidation
                 .has_valid_parameters(conn.path_params),
           :ok <-
             Validation.PreferencesValidation
                           .has_preferences(body),
           :ok <-
            Validation.AccommodationValidation
                     .has_accommodation_preferences(body) do

         ...
```

# Categorise Validations

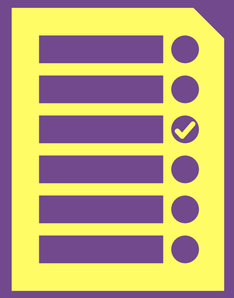
```
with {:ok, customer_id} <-
       Validation.PostParameterValidation
                 .has_valid_parameters(conn.path_params),
         :ok <-
           Validation.PreferencesValidation
                           .has_preferences(body),
         :ok <-
          Validation.AccommodationValidation
                   .has_accommodation_preferences(body) do

           ...
```

Header

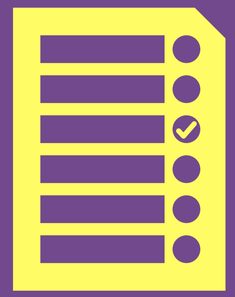
# Categorise Validations



# Define Validation Rules



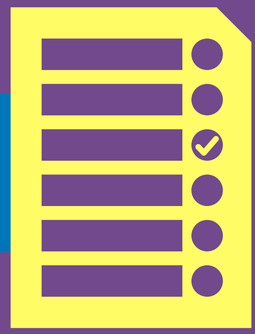
[HeaderRule1, HeaderRule2, HeaderRule3…]



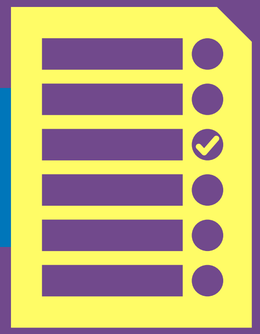
[BodyRule1, BodyRule2, BodyRule3…]

# Categorise Validations

def module Validation.HeaderValidator



def module Validation.BodyValidator



```elixir
defmodule Validation.HeaderValidator do
  def validate(params) do
    parameter_validators =
        [&Validation.RequestParameterValidator.
                                  has_valid_parameters/1,
         &Validation.CustomerRestriction.unrestricted/1]

  def validate(params, header_validators) do
    validate_all_rules(header_validators, params)
  end

  defp validate_all_rules([], params) do
    {:ok, params["id"]}
  end
  defp validate_all_rules([rule|tail], params) do
    {status, data} = rule.(params)

    if status == :error do
      {:error, data}
    else
      validate_all_rules(tail, params)
    end
  end
end
```

Configure Rules

```elixir
defmodule Validation.HeaderValidator do
  def validate(params) do
    parameter_validators =
      [&Validation.RequestParameterValidator.has_valid_parameters/1,
       &Validation.CustomerRestriction.unrestricted/1]

    validate(params, parameter_validators)
  end

  def validate(params, header_validators) do
    validate_all_rules(header_validators, params)
  end

  defp validate_all_rules([], params) do
    {:ok, params["id"]}
  end

  defp validate_all_rules([rule|tail], params) do
    {status, data} = rule.(params)

    if status == :error do
      {:error, data}
    else
      validate_all_rules(tail, params)
    end
  end
```

Traverse and Execute Rules

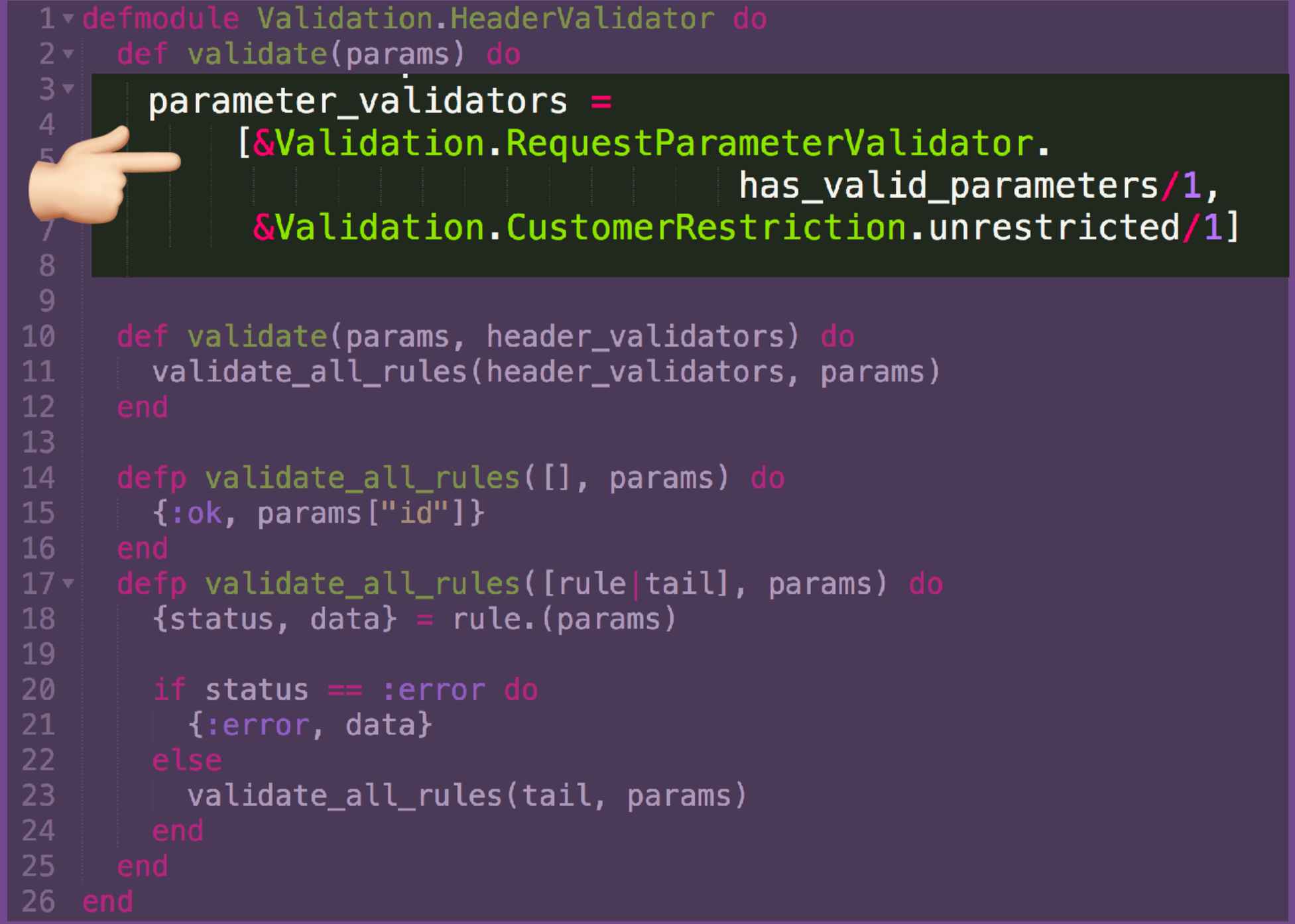
# Save Customer

```elixir
defmodule Plugs.SaveCustomer do
  import Plug.Conn

  def init(opts) do
   opts
  end

  def call(conn, _opts) do
   path_params = conn.path_params
   body = conn.body_params

   with {:ok, customer_id} <-
                  Validation.HeaderValidator.validate(path_params),
        {:ok, valid_body} <-
                  Validation.BodyValidator.validate(body) do

     send_resp(conn, 201, Poison.encode!(response))
   else
     {:error, {_, reason}} -> send_resp(conn, 400, Poison.encode!(%{error: reason}))
   end
  end
end
```

Validation Per Category

# Save Customer

```elixir
defmodule Plugs.SaveCustomer do
  import Plug.Conn

  def init(opts) do
   opts
  end

  def call(conn, _opts) do
   path_params = conn.path_params
   body = conn.body_params

   with {:ok, customer_id} <-
                  Validation.HeaderValidator.validate(path_params),
        {:ok, valid_body} <-
                  Validation.BodyValidator.validate(body) do

     send_resp(conn, 201, Poison.encode!(response))
   else
     {:error, {_, reason}} -> send_resp(conn, 400, Poison.encode!(%{error: reason}))
   end
  end
end
```


defmodule Validation.HeaderValidator do
def validate(params) do
parameter_validators =
[&Validation.RequestParameterValidator.
has_valid_parameters/1,
&Validation.CustomerRestriction.unrestricted/1]
def validate(params, header_validators) do
validate_all_rules(header_validators, params)
end
defp validate_all_rules([], params) do
{:ok, params["id"]}
end
defp validate_all_rules([rule|tail], params) do
{status, data} = rule.(params)
if status == :error do
{:error, data}
else
validate_all_rules(tail, params)
end
end
end

# Behaviours

# Behaviours

“Behaviours provide a way to:

- define a set of functions that have to be implemented by a module
- ensure that a module implements all of the functions in that set.”

*https://elixir-lang.org/getting-started/typespecs-and-behaviours.html#behaviours*

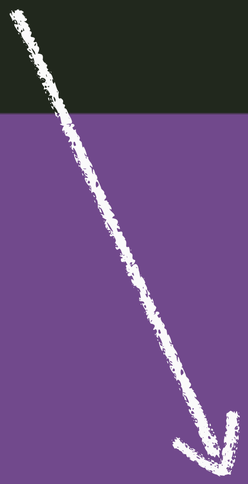
# Validation Rule Behaviour

```
defmodule Validation do
  @callback is_valid(map)  ::   {:ok, term}
                              | {:error, {atom, reason :: String.t}}
end
```

# Validation Rule Behaviour

```
defmodule Validation do
  @callback is_valid(map)   ::    {:ok, term}
                                | {:error, {atom, reason :: String.t}}
end
```

```
{:ok, body}
```



```
{:error, {:no_accommodation, "Accommodation preferences missing"}}
```

# Implement the Behaviour

# Implement Behaviour

```elixir
defmodule Validation.Body.ValidationRule do
  @behaviour Validation

  def is_valid(request_body) do
    ...
  end
end
```

# Implement Behaviour

```elixir
defmodule Validation.Body.ValidationRule do
  @behaviour Validation

  def is_valid(request_body) do
    ...
  end
end
```

```
parameter_validators =
    [&Validation.RequestParameterValidator.
                                has_valid_parameters/1,
     &Validation.CustomerRestriction.unrestricted/1]
```

```
parameter_validators =
  [Validation.RequestParameterValidator,
   Validation.CustomerRestriction]
```

Modules implement Behaviour

```elixir
defp validate_all_rules([rule|tail], params) do
  {status, data} = rule.(params)

  if status == :error do
    {:error, data}
  else
    validate_all_rules(tail, params)
  end
end
```

```elixir
defp validate_all_rules([rule|tail], params) do
  {status, data} = rule.is_valid(params)

  if status == :error do
    {:error, data}
  else
    validate_all_rules(tail, params)
  end
end
```

Execute Contract

# Configure the Rules

```
config :customer_preference_api,
  header_validators: [Validation.RequestParameterValidator,
                      Validation.CustomerRestriction
  ],
  body_validators: [Validation.Body.PreferencesValidation,
                    Validation.Body.AccommodationValidation]

```

# Configure the Rules

```
defmodule Plugs.SaveCustomer do
  ...

  def call(conn, _opts) do
   ...

header_validators =
  Application.get_env(:customer_preference_api, :header_validators)
body_validators =
  Application.get_env(:customer_preference_api, :body_validators)

            valid_body = Schema.Http.Request.new!(body, customer_id, body["version"])
            response = Controllers.SavePreferenceController.save_preferences(customer_id, valid_body)
            send_resp(conn, 201, Poison.encode!(response))
        else
           {:error, {_, reason}} -> send_resp(conn, 400, Poison.encode!(%{error: reason}))
        end
    end
end
```

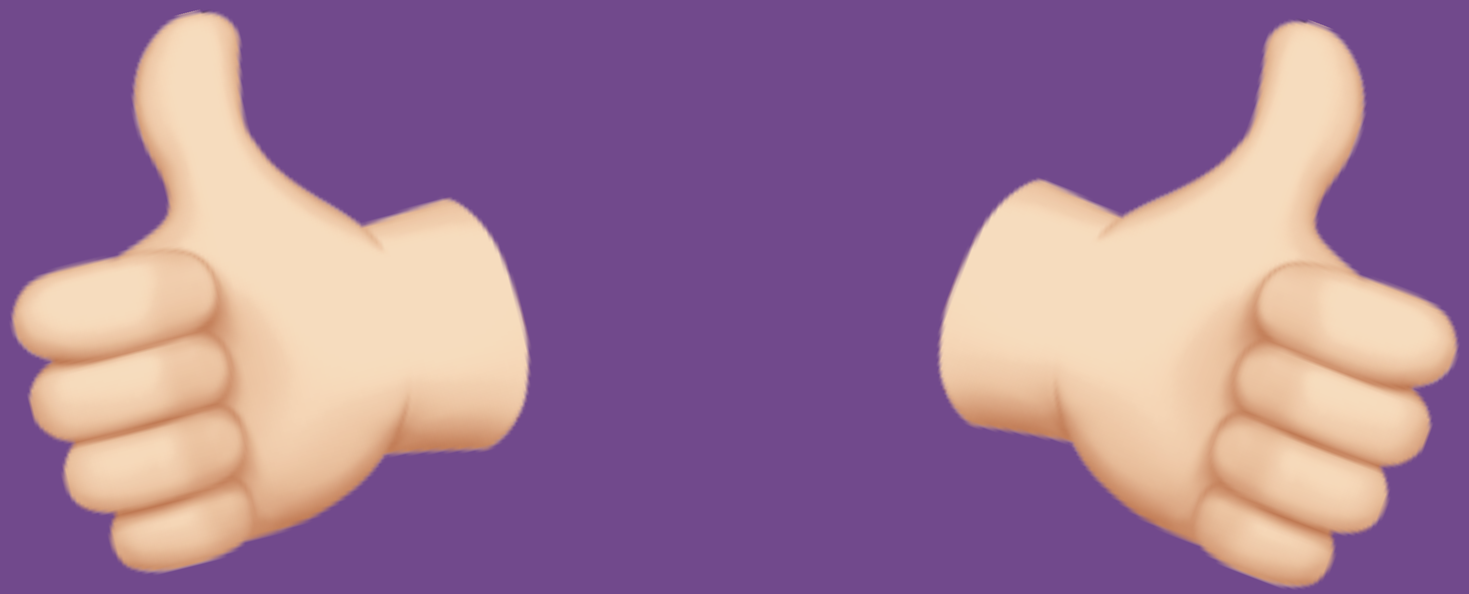
# To Add a New Rule

✅ Implement New module

- implements behaviour
- provides business logic for rule

✅ Update the config for the relevant MIX_ENV

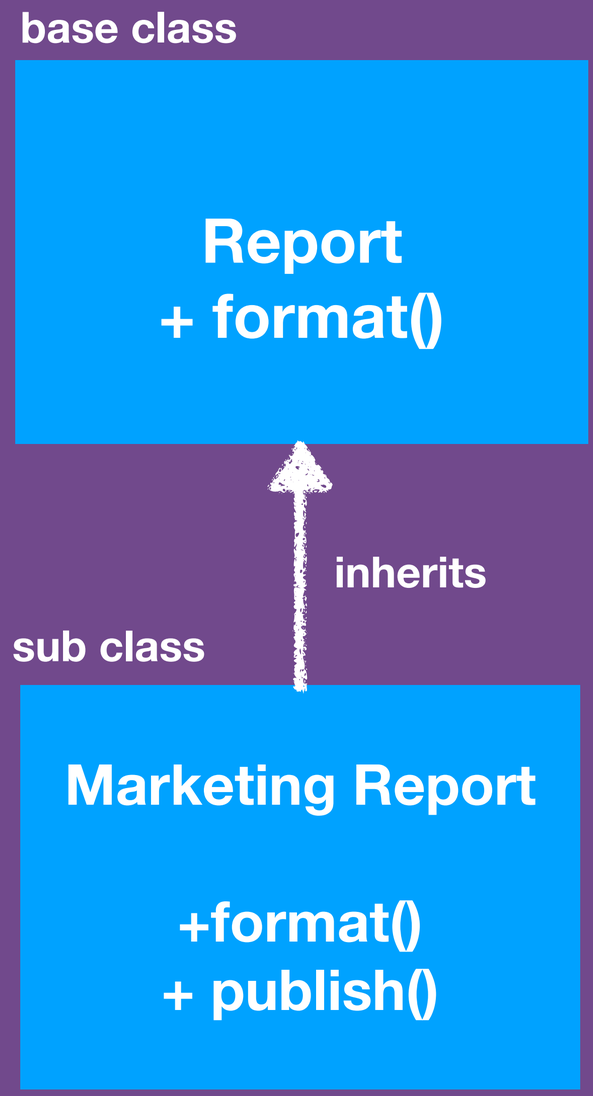
# You can add rules without touching any existing code



# Liskov Substitution

## SOLID

# Inheritance



“Liskov's notion of a behavioral subtype defines a notion of substitutability for objects; that is, if *S* is a subtype of *T*, then objects of type *T* in a program may be replaced with objects of type *S* without altering any of the desirable properties of that program.”

*– Wikipedia*
*https://en.wikipedia.org/wiki/Liskov_substitution_principle*

# OO Substitution

```
MarketingReport marketingReport = new MarketingReport();
marketingReport.format();
```

# OO Substitution

**Marketing Report**

```
MarketingReport marketingReport = new MarketingReport();
marketingReport.format();
```

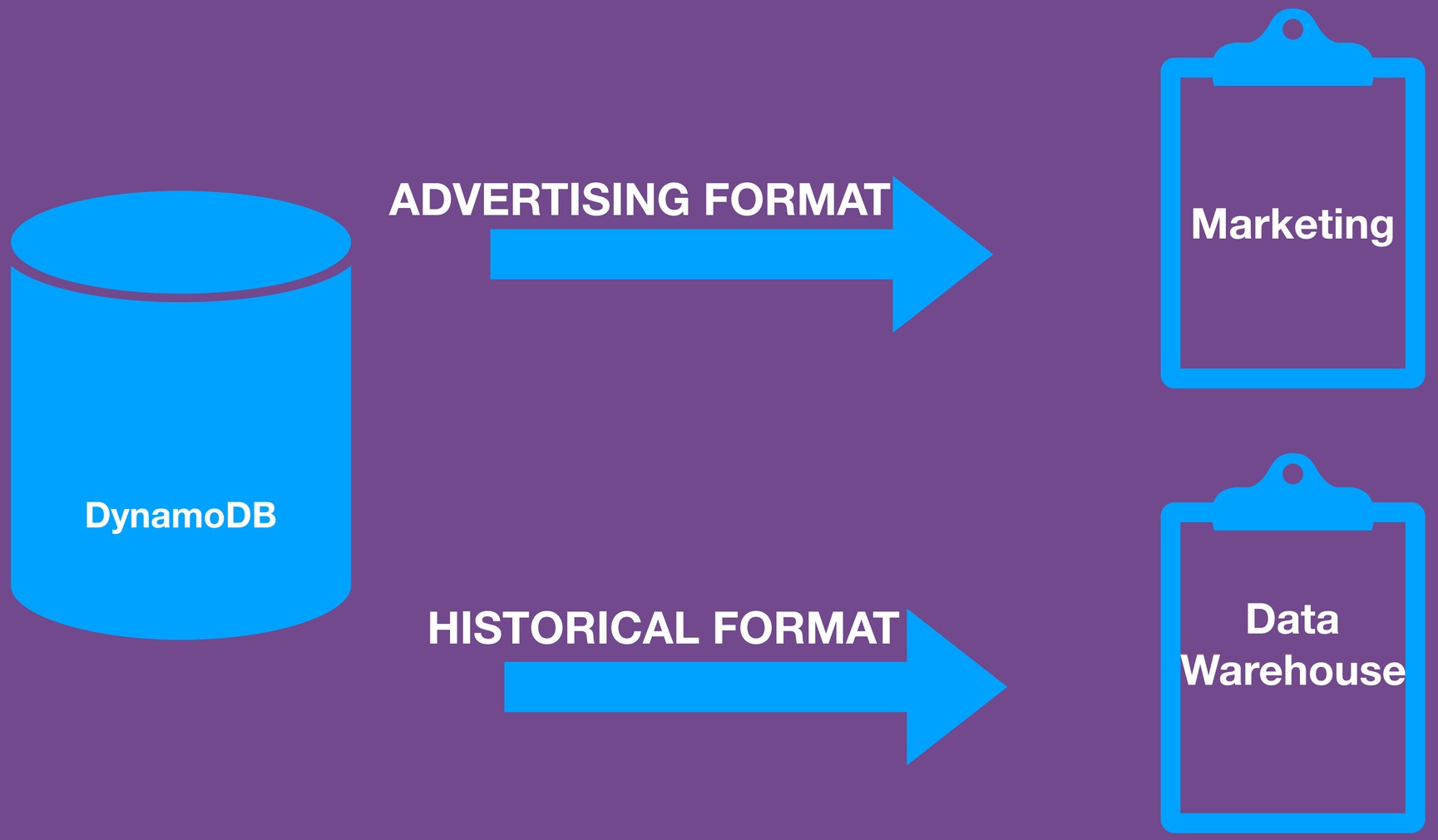
**+format()**
**+ publish()**

**Report**

```
Report marketingReport = new MarketingReport();
marketingReport.format();
```

**+ format()**

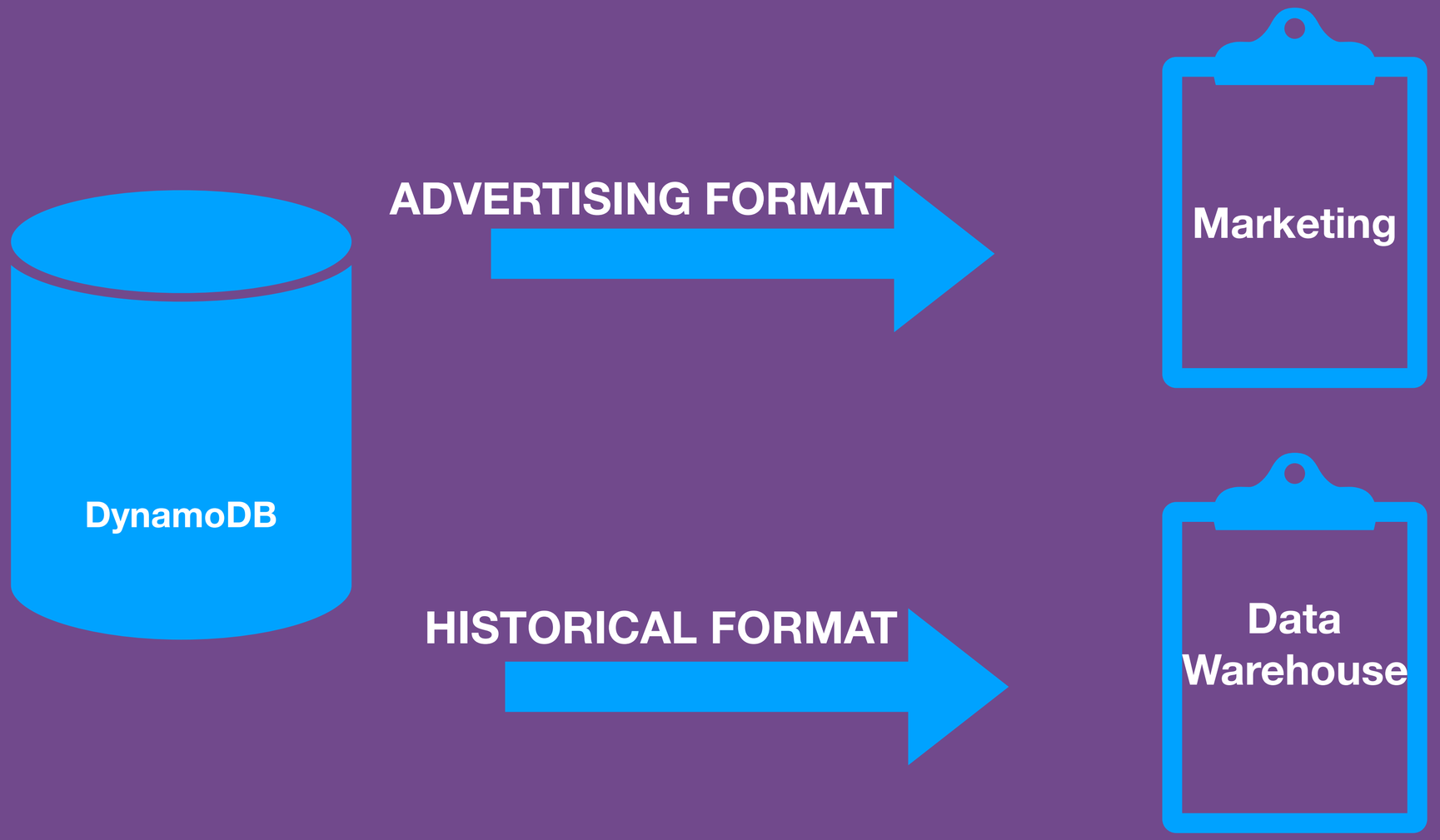
# Reporting Capabilities



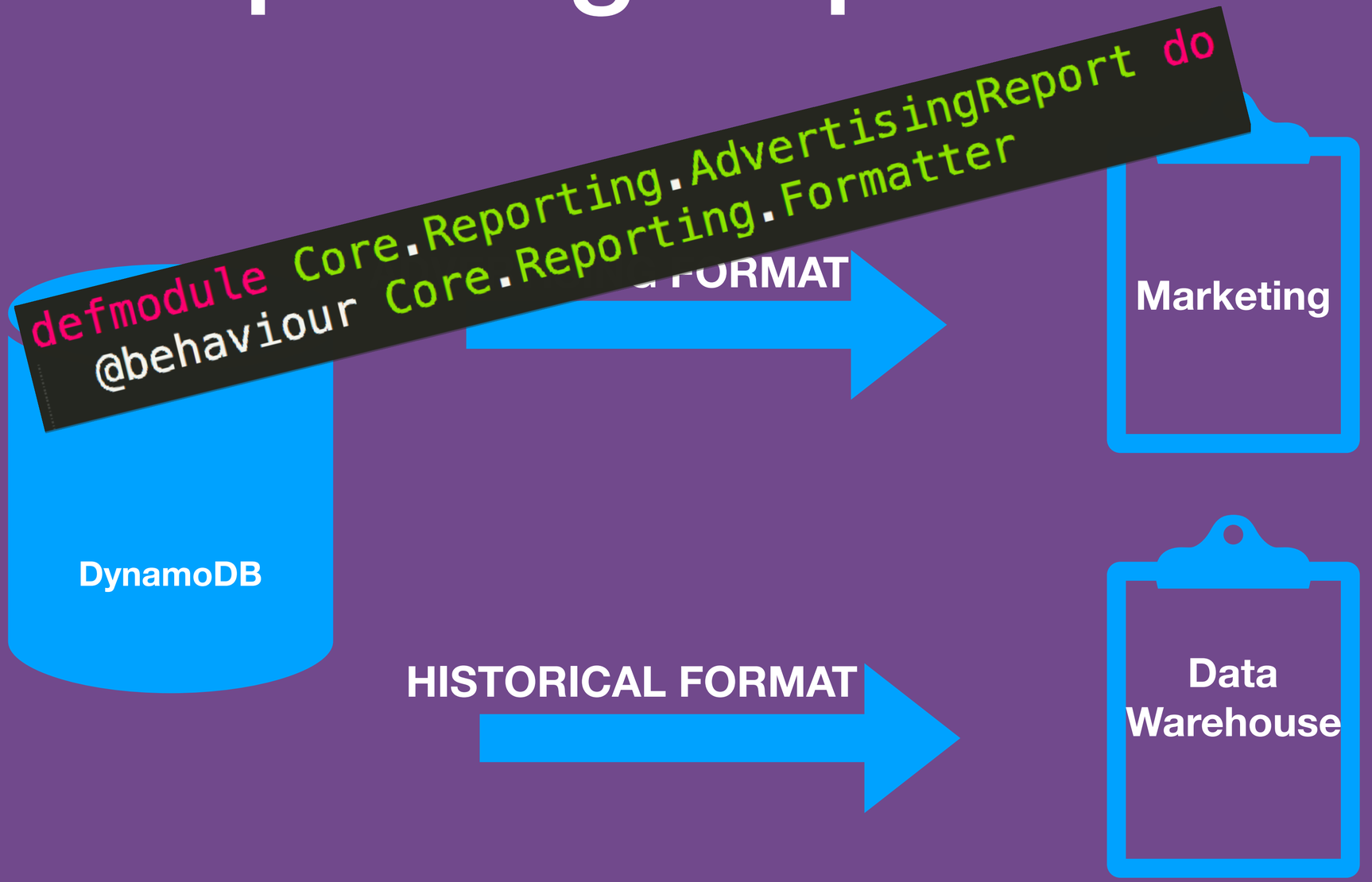
# Format Behaviour

```
defmodule Core.Reporting.Formatter do
    @callback format_to_rows(map) ::    {:ok, map}
                                      | {:error, String.t}
end
```

# Reporting Capabilities



# Reporting Capabilities

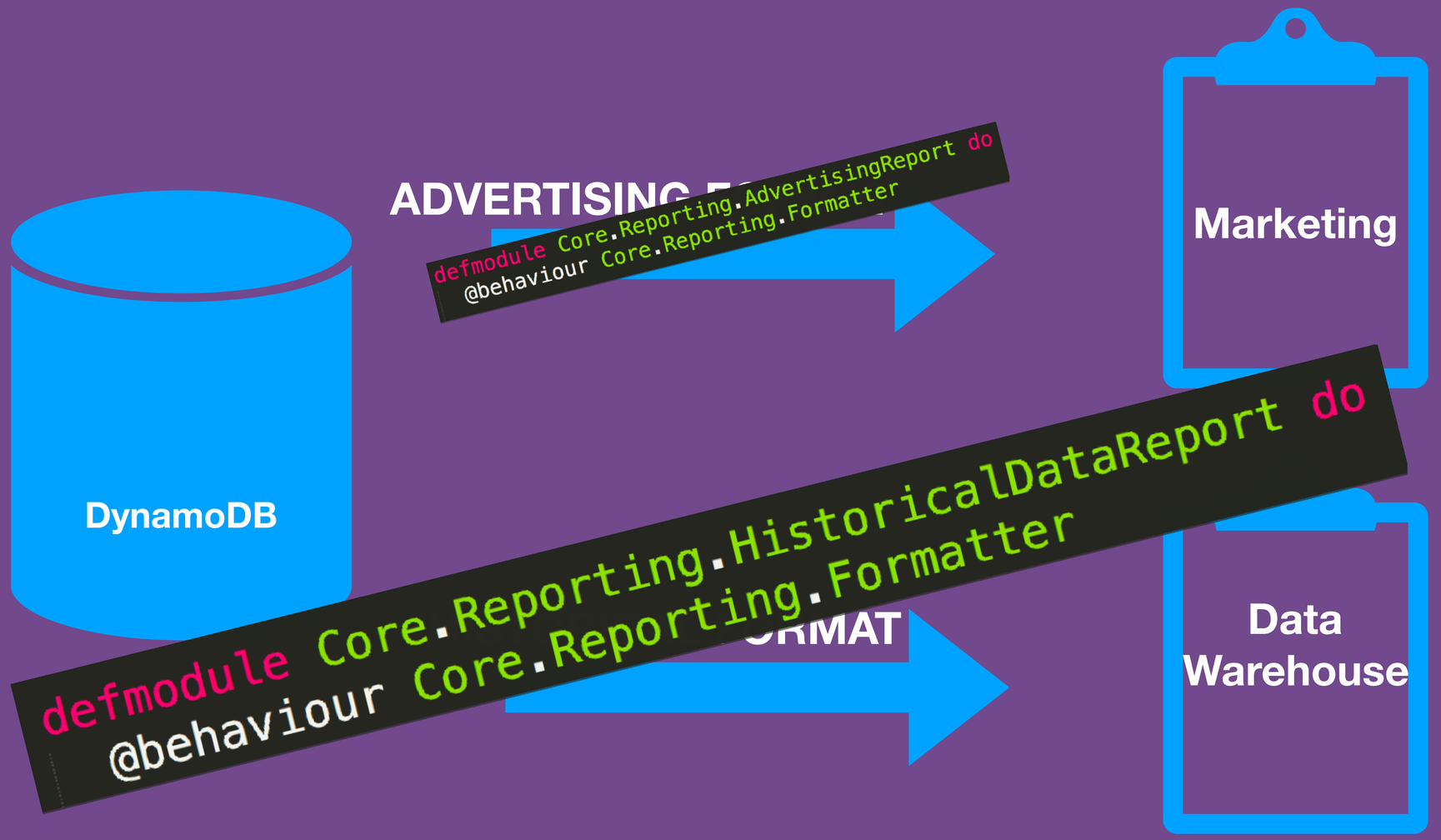


# Reports

```elixir
defmodule Core.Reporting.AdvertisingReport do
  @behaviour Core.Reporting.Formatter

  @impl true
  def format_to_rows(data) do
    formatted_data = data
    IO.inspect("format for advertising")
    {:ok, formatted_data}
  end
end
```

FORMAT

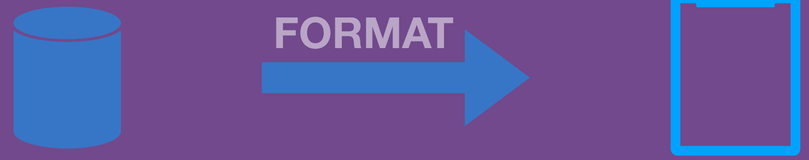
# Reporting Capabilities



# Reporting Capabilities

# Generation

```elixir
defmodule Core.Reporting.ReportGenerator do

  @spec generate_report(data :: map, formats :: list) :: atom
  def generate_report(data, formats) do
    formatted_reports = Enum.map(formats, fn(format) ->
      format.format_to_rows(data)
    end)
    dispatch(formatted_reports)
  end

  def dispatch(reports) do
    IO.inspect("send to external systems")
    ...
    :ok
  end
end
```



# Generation

```
1  defmodule Core.Reporting.ReportGenerator do
2
3    @spec generate_report(data :: map, formats :: list) :: atom
4    def generate_report(data, formats) do
5      formatted_reports = Enum.map(formats, fn(format) ->
6        format.format_to_rows(data)
7      end)
9    end
10
11   def dispatch(reports) do
12    IO.inspect("send to external systems")
13    ...
14    :ok
15   end
16 end
```

# Liskov Substitution

✅ Add any new module which implements the Behaviour without altering any desirable properties

# Legal Report

# Legal Report

- Format as per HistoricalDataReport
- Attach a disclaimer
- Attach a header
- Apply colour to headers

# Legal Report

```elixir
defmodule Core.Reporting.LegalReportGenerator do

  def generate_report(data, formatter) do
   formatter.format_to_rows(data)
    |> formatter.add_disclaimer
    |> formatter.add_header
    |> formatter.colour
    |> dispatch()
   :ok
  end

  def dispatch(report) do
   IO.inspect("send to external systems")
   :ok
  end
end
```

```elixir
defmodule Core.Reporting.HistoricalDataReport do
  @behaviour Core.Reporting.Formatter

  @impl true
  def format_to_rows(data) do
    formatted_data = data
    IO.inspect("format for historical data")
    {:ok, formatted_data}
  end

  def add_disclaimer(data) do
    IO.inspect("..adding disclaimer..")
    data
  end

  def add_header(data) do
    IO.inspect("..adding header..")
    data
  end

  def colour(data) do
    IO.inspect("..adding colour..")
    data
  end
end
```

```elixir
defmodule Core.Reporting.HistoricalDataReport do
  @behaviour Core.Reporting.Formatter

  @impl true
  def format_to_rows(data) do
    formatted_data = data
    IO.inspect("format for historical data")
    {:ok, formatted_data}
  end

  def add_disclaimer(data) do
    IO.inspect("..adding disclaimer..")
    data
  end

  def add_header(data) do
    IO.inspect("..adding header..")
    data
  end

  def colour(data) do
    IO.inspect("..adding colour..")
    data
  end
end
```

# Generate Legal Report

```
def generate_report(data, formatter) do
 formatter.format_to_rows(data)
  |> formatter.add_disclaimer
  |> formatter.add_header
  |> formatter.colour
  |> dispatch()
 :ok
end
```

# Generate Legal Report

```
iex(2)> Core.Reporting.LegalReportGenerator.generate_report(%{"preferences" => "..."},
 Core.Reporting.HistoricalDataReport)
"format for historical data"
"..adding disclaimer.."
"..adding header.."
"..adding colour.."
"send to external systems"
:ok
```

What happens if you generate a Legal Report with another Formatter implementation?

# Generate Legal Report

```
def generate_report(data, formatter) do
 formatter.format_to_rows(data)
  |> formatter.add_disclaimer
  |> formatter.add_header
  |> formatter.colour
  |> dispatch()
 :ok
end
```

# Generate Legal Report

```
iex(1)> Core.Reporting.LegalReportGenerator.generate_report(%{"preferences" => "..."},
 Core.Reporting.AdvertisingReport)
"format for advertising"
** (UndefinedFunctionError) function Core.Reporting.AdvertisingReport.add_disclaimer/1
 is undefined or private
    (customer_preference_api) Core.Reporting.AdvertisingReport.add_disclaimer({:ok, %{
"preferences" => "..."}})
    (customer_preference_api) lib/core/reporting/legal_report_generator.ex:11: Core.Re
porting.LegalReportGenerator.generate_report/2
iex(1)>
```

# Liskov Substitution Violation

According to Liskov Substitution we should be able to plug any implementation in without altering any of the desirable properties

# Liskov Substitution

✅ Format as  HistoricalDataReport

🤔 Extra Processing to add headers, colours etc

💡 Create a new module that uses the HistoricalDataReport and does a bit more

```elixir
defmodule Core.Reporting.HistoricalDataDecorator do
  @behaviour Core.Reporting.Formatter

  @impl true
  def format_to_rows(data) do
   IO.inspect("format for historical data")
   formatted_data =
     Core.Reporting.HistoricalDataReport.format_to_rows(data)
           |> add_disclaimer()
           |> add_header()
           |> colour()
   {:ok, formatted_data}
  end

  def add_disclaimer(data) do
   IO.inspect("..adding disclaimer..")
   data
  end

  def add_header(data) do
   IO.inspect("..adding header..")
   data
  end

  def colour(data) do
   IO.inspect("..adding colour..")
   data
  end
end
```

```elixir
defmodule Core.Reporting.HistoricalDataDecorator do
  @behaviour Core.Reporting.Formatter

  @impl true
  def format_to_rows(data) do
    IO.inspect("format for historical data")
    formatted_data =
      Core.Reporting.HistoricalDataReport.format_to_rows(data)
      |> add_disclaimer()
      |> add_header()
      |> colour()
    {:ok, formatted_data}
  end

  def add_disclaimer(data) do
    IO.inspect("..adding disclaimer..")
    data
  end

  def add_header(data) do
    IO.inspect("..adding header..")
    data
  end

  def colour(data) do
    IO.inspect("..adding colour..")
    data
  end
end
```

@gemcfadyen

```elixir
defmodule Core.Reporting.HistoricalDataDecorator do
  @behaviour Core.Reporting.Formatter

  @impl true
  def format_to_rows(data) do
    IO.inspect("format for historical data")
    formatted_data =
      Core.Reporting.HistoricalDataReport.format_to_rows(data)
          |> add_disclaimer()
          |> add_header()
          |> colour()
  end

  def add_disclaimer(data) do
    IO.inspect("..adding disclaimer..")
    data
  end

  def add_header(data) do
    IO.inspect("..adding header..")
    data
  end

  def colour(data) do
    IO.inspect("..adding colour..")
    data
  end
end
```

```
defmodule Core.Reporting.ReportGenerator do
```

```
defmodule Core.Reporting.AdvertisingReport do
  @behaviour Core.Reporting.Formatter
```

```
defmodule Core.Reporting.HistoricalDataReport do
  @behaviour Core.Reporting.Formatter
```

```
defmodule Core.Reporting.HistoricalDataDecorator do
  @behaviour Core.Reporting.Formatter
```

# Liskov Substitution

✅ Wrapping an existing Formatter allowed us to extend it’s functionality

✅ Substitute any implementation safely

# Liskov Substitution

Where we have code that expects a generic type (i.e: Behaviour) - Ensure you are only using function calls defined on that Behaviour

# But why didn’t we just enhance the original Behaviour with the new functions?

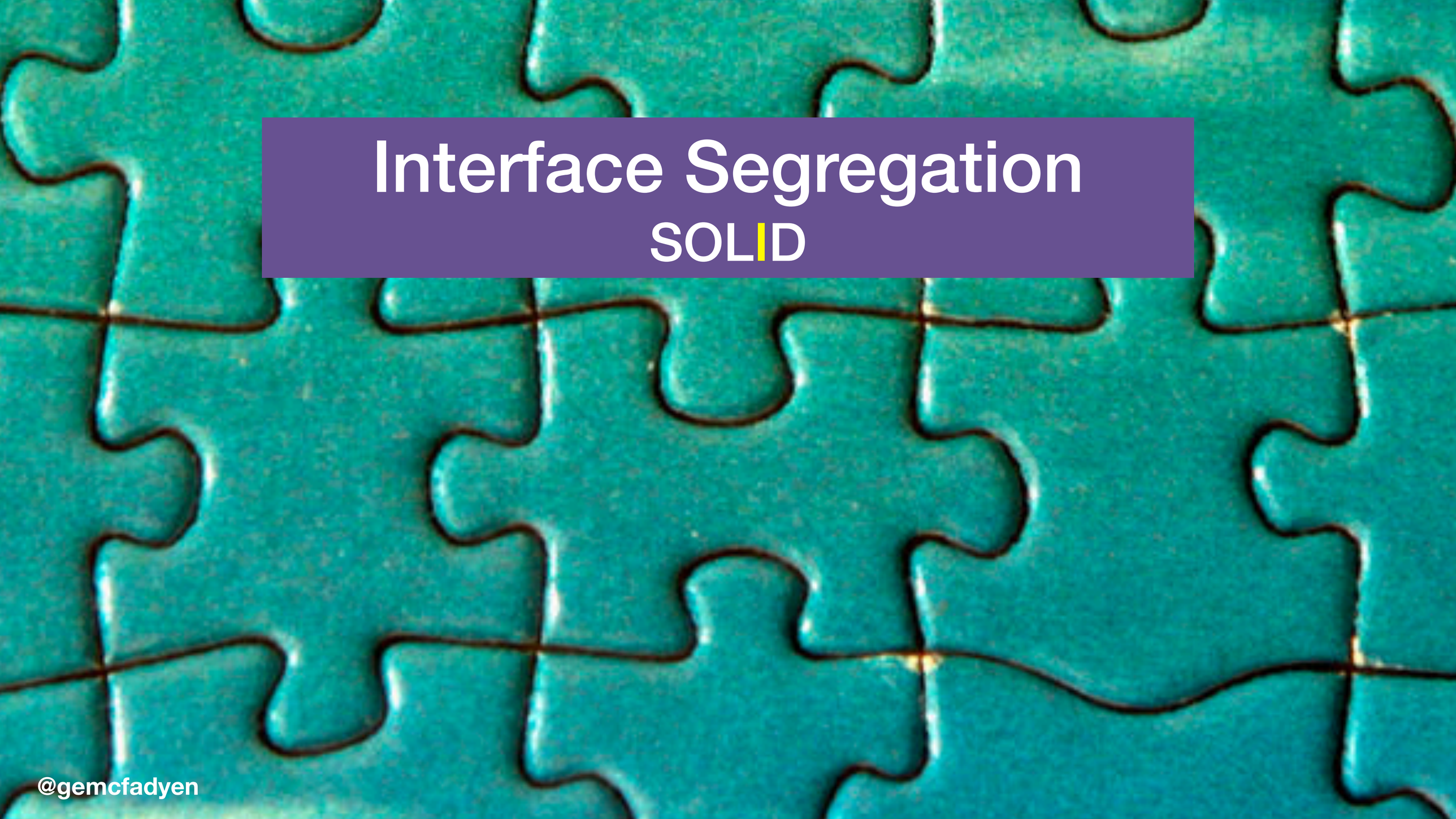
# Update Each Implementation

✅ Update HistoricalDataReport

? Update AdvertisingReport

```elixir
defmodule Core.Reporting.AdvertisingReport do
  @behaviour Core.Reporting.Formatter

  @impl true
  def format_to_rows(data) do
   formatted_data = data
   IO.inspect("format for advertising")
   #... some data clensing and reformatting logic
   {:ok, formatted_data}
  end

  def add_disclaimer(data) do
   IO.puts("nothing to do..")
   data
  end

  def add_header(data) do
   IO.puts("nothing to do..")
   data
  end

  def colour(data) do
   IO.puts("nothing to do..")
   data
  end
end
```

 Bloated the Behaviour to satisfy a new requirement


Interface Segregation
SOLID
@gemcfadyen

"The **interface-segregation principle (ISP)** states that no client should be forced to depend on methods it does not use.[1] ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them."

*– Wikipedia*
*https://en.wikipedia.org/wiki/Interface_segregation_principle*

# Interface Segregation Signs

- Look for Behaviours which have methods that don’t apply to all implementors
- Look for Behaviours that have many function definitions

# Interface Segregation Signs

```elixir
defmodule Core.Reporting.Formatter do
  @callback format_to_rows([map]) ::  {:ok, [map]}
                                    | {:error, String.t}

  @callback add_disclaimer([map]) :: [map]

  @callback add_header([map]) :: [map]

  @callback colour([map]) :: [map]
end
```

# Split Behaviour

```
1 defmodule Core.Reporting.Formatter do
2   @callback format_to_rows([map]) ::  {:ok, [map]}
3                                     | {:error, String.t}
4 end
```

# Split Behaviour

```elixir
defmodule Core.Reporting.Formatter do
  @callback format_to_rows([map]) ::  {:ok, [map]}
                                    | {:error, String.t}
end
```

```elixir
defmodule Reporting.Presentation do
  @callback add_disclaimer([map]) :: [map]

  @callback add_header([map]) :: [map]

  @callback colour([map]) :: [map]
end
```

```elixir
defmodule Core.Reporting.HistoricalDataReport do
  @behaviour Core.Reporting.Formatter
  @behaviour Core.Reporting.Presentation

  def format_to_rows(data) do
    formatted_data = data
    IO.inspect("format for historical data")
    #... some data clensing and reformatting logic
    {:ok, formatted_data}
  end

  def add_disclaimer(data) do
    IO.inspect("..adding disclaimer..")
    #...adds legal disclaimer
    data
  end

  def add_header(data) do
    IO.inspect("..adding header..")
    data
  end

  def colour(data) do
    IO.inspect("..adding colour..")
    data
  end
end
```

```elixir
defmodule Core.Reporting.ReportGenerator do
 def generate_report(data, formats) do
 formatted_reports = Enum.map(formats, fn(format) ->
  format.format_to_rows(data)
 end)
 dispatch(formatted_reports)
end

 def present_report(formatted_data, presenters) do
   formatted_reports = Enum.map(presenters,
    fn(presenter) ->
      presenter.add_disclaimer()
      |> presenter.add_header()
      |> presenter.colour()
      |> dispatch
    end)
  end

  def dispatch(reports) do
   IO.inspect("send to external systems")
   :ok
  end
end
```

```elixir
defmodule Core.Reporting.ReportGenerator do
  def generate_report(data, formats) do
  formatted_reports = Enum.map(formats, fn(format) ->
   format.format_to_rows(data)
  end)
  dispatch(formatted_reports)
 end
```

```elixir
def present_report(formatted_data, presenters) do
  formatted_reports = Enum.map(presenters,
   fn(presenter) ->
     presenter.add_disclaimer()
     |> presenter.add_header()
     |> presenter.colour()
     |> dispatch
    end)
 end
```

```elixir
  def dispatch(reports) do
   IO.inspect("send to external systems")
   :ok
  end
end
```

# Split Processing

```
 1  defmodule Core.Reporting.ReportingClient do
 2    def entry_point(data) do
 3      formatted_data = generate_report(data,
 4                                       [Core.Reporting.TrendsReport,
 5                                        Core.Reporting.AdvertisingReport])
 6                       |> dispatch
 7
 8      present_report(data,
 9                     [Core.Reporting.HistoricalDataReport])
10      |> dispatch
11    end
12  end
```

✅ Force implementations only on those modules that actually need them

# Dependency Inversion

## SOLID

@gemcfadyen

The dependency inversion principle refers to a specific form of decoupling software modules.

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

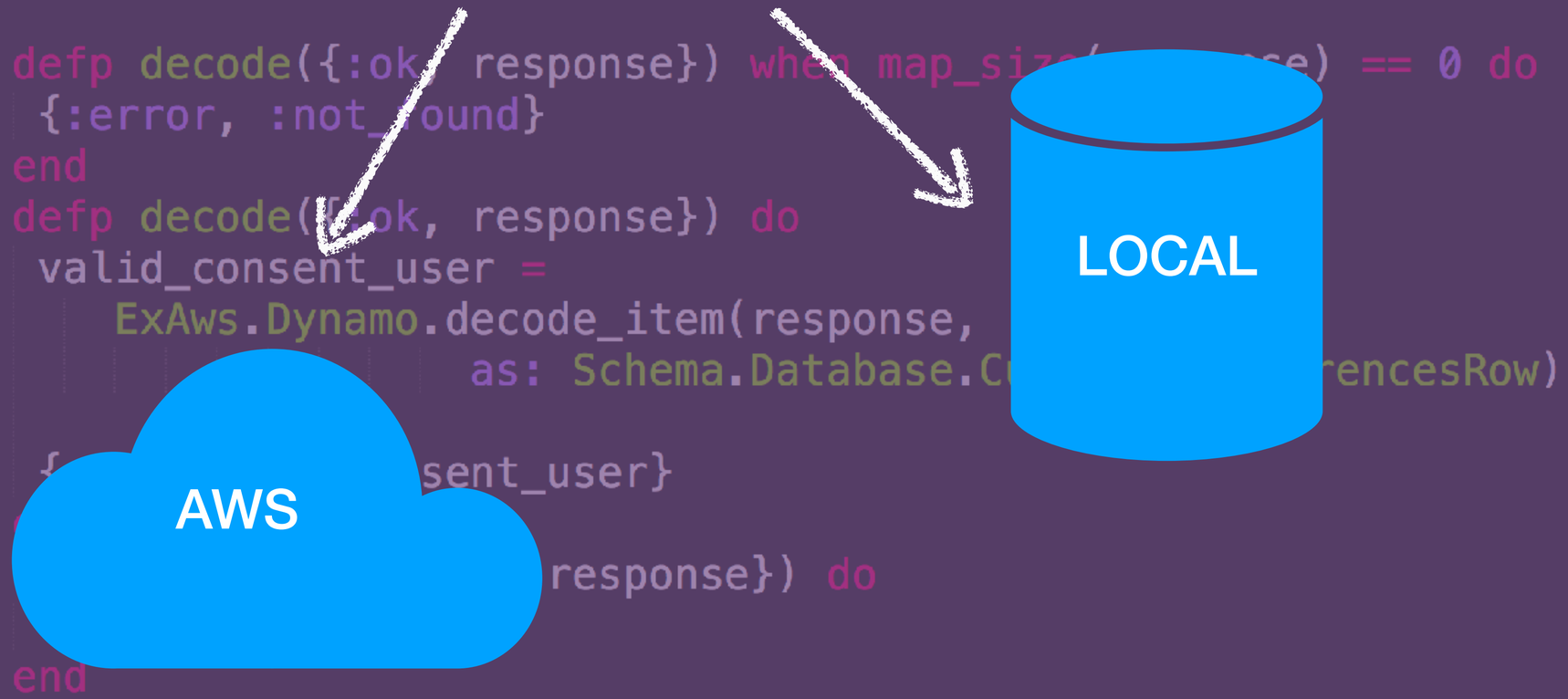
*– Wikipedia*
*https://en.wikipedia.org/wiki/Dependency_inversion_principle*

# GET /customers/{id}


```elixir
defmodule Core.Preferences.GetCustomerPreferences do

  def for(customer_id) do
    ExAws.Dynamo.get_item("customer-preferences",
                          %{id: customer_id})
    |> ExAws.request

  end

  defp decode({:ok, response}) when map_size(response) == 0 do
    {:error, :not_found}
  end
  defp decode({:ok, response}) do
    valid_consent_user =
      ExAws.Dynamo.decode_item(response,
                               as: Schema.Database.CustomerPreferencesRow)

    {:ok, valid_consent_user}
  end
  defp decode({:error, response}) do
    {:error, response}
  end
end
```

Fetch Customer Preferences

```elixir
ExAws.Dynamo.get_item("customer-preferences",
                      %{id: customer_id})
|> ExAws.request
```



```elixir
defmodule Core.Preferences.GetCustomerPreferences do

  def for(customer_id) do
    ExAws.Dynamo.get_item("customer-preferences",
                          %{id: customer_id})
    |> ExAws.request

  end

  defp decode({:ok, response}) when map_size(response) == 0 do
    {:error, :not_found}
  end
  defp decode({:ok, response}) do
    valid_consent_user =
      _item(response,
      chema.Database.CustomerPreferencesRow)

    {
      er}
  end
  de
    sponse}) do
    {
  end
end
```

FAKE DATABASE

MIX_ENV = test
- fake-database

MIX_ENV = dev
- local-database

MIX_ENV = prod
- aws-database

# Isolation

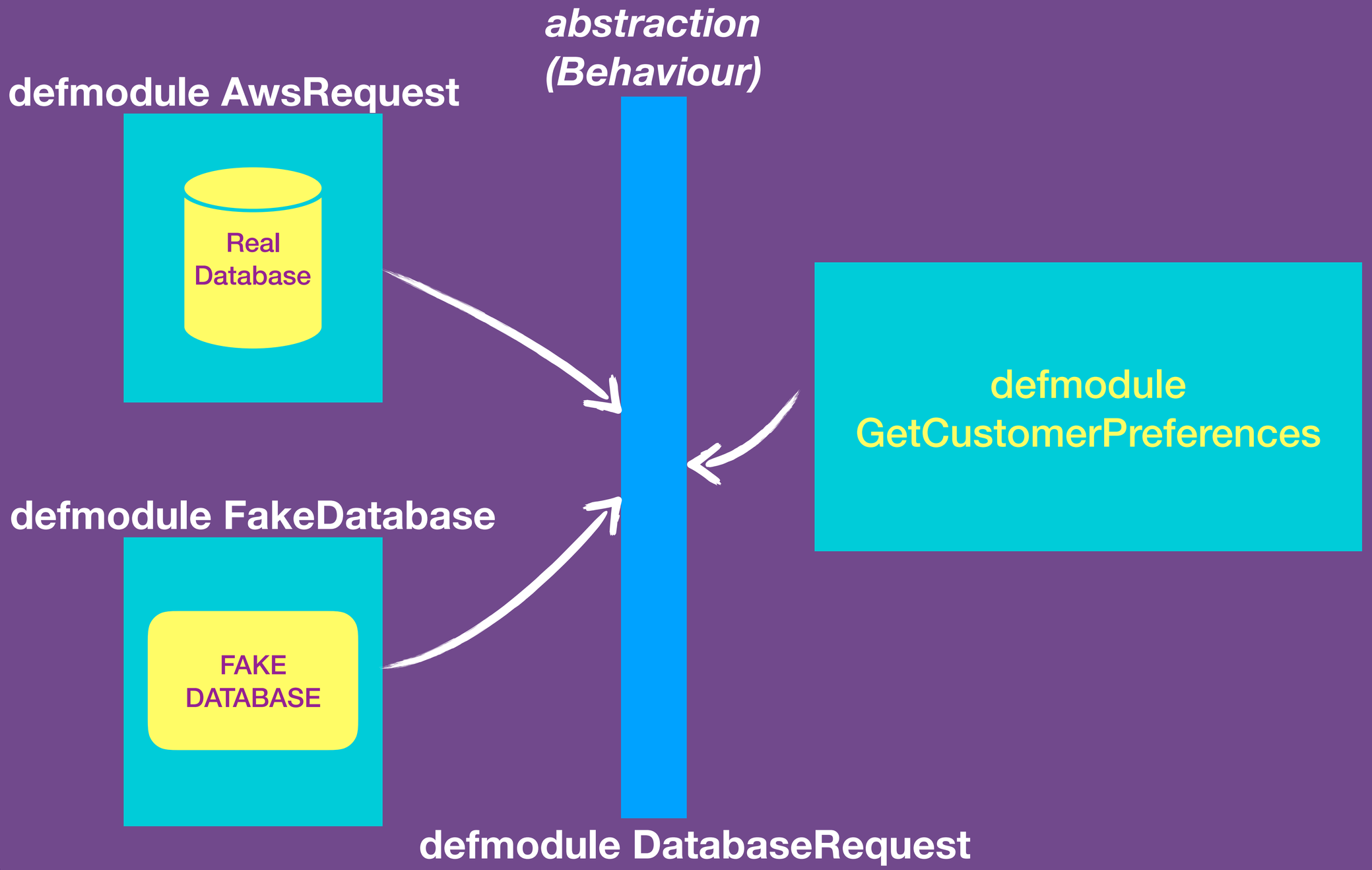
```
def for(customer_id) do
 ExAws.Dynamo.get_item("customer-preferences",
                       %{id: customer_id})
 |> ExAws.request
 |> decode
end
```

```
|> ExAws.request
```

# Isolation

```elixir
def for(customer_id) do
 ExAws.Dynamo.get_item("customer-preferences",
                       %{id: customer_id})
 |> ExAws.request
 |> decode
end
```

# Use Config to Substitute Test Doubles


abstraction
(Behaviour)
defmodule AwsRequest
Real
Database
defmodule
GetCustomerPreferences
defmodule FakeDatabase
FAKE
DATABASE
defmodule DatabaseRequest

# Create Abstraction

```elixir
defmodule Database.Request do
  @callback request(ExAws.Operation.t, Keyword.t) ::  {:ok, term}
                                                    | {:error, term}

  @callback request(ExAws.Operation.t) ::   {:ok, term}
                                          | {:error, term}
end
```

# Wrap the real database functionality

```elixir
defmodule Database.AwsRequest do
  @behaviour Database.Request

  @impl true
  def request(op, config_overrides \\ []) do
    ExAws.request(op, config_overrides)
  end
end
```

```elixir
defmodule Database.AwsRequest do
  @behaviour Database.Request

  @impl true
  def request(op, config_overrides \\ []) do
    ExAws.request(op, config_overrides)
  end
end
```

# Provide a fake database request module

```elixir
defmodule Database.FakeDatabase do
  @behaviour Database.Request

  @impl true
  def request(op, config_overrides \\ [])

  @impl true
  def request(%{http_method: :post}, _) do
   {:ok, canned_result()}
  end

  def canned_result do
   IO.puts("Saving data...")

   %{"Item" =>
    %{"id" => %{"S" => "uuid-1"},
     "preferences" => %{"M" =>
      %{"accommodation" => %{"M" =>
       %{"apartment" => %{"M" =>
        %{"bedrooms" => %{"N" => "2"},
          "catering" => %{"S" => "self_catering"},
          "parking" => %{"S" => "secure"}}
          }}}}}}}
  end
end
```

MIX_ENV=test fake_database.ex

MIX_ENV=prod aws_request.ex

## customer_preference_api/config/test.exs

```
config :customer_preference_api,
  aws_request: Database.FakeDatabase
```

## customer_preference_api/config/prod.exs

```
config :customer_preference_api,
  aws_request: Database.AwsRequest
```

```elixir
defmodule Core.Preferences.GetCustomerPreferences do

  def for(customer_id) do
   database = Application.get_env(:customer_preference_api,
                                  :aws_request)

    ExAws.Dynamo.get_item("customer-preferences", %{id: customer_id})
    |> database.request
    |> decode
  end
  ...
end
```

Lookup implementation

# Configure different Scenarios

- Testing scenarios where nothing is found
- Testing scenarios where multiple entries found….

# Configure Fake via Config

✅ Isolates ‘fake’ functionality to specific MIX_ENV

❌ Messy to handle different canned results for different scenarios

❌ Difficult to know what ‘fake’ logic is being executed for what test

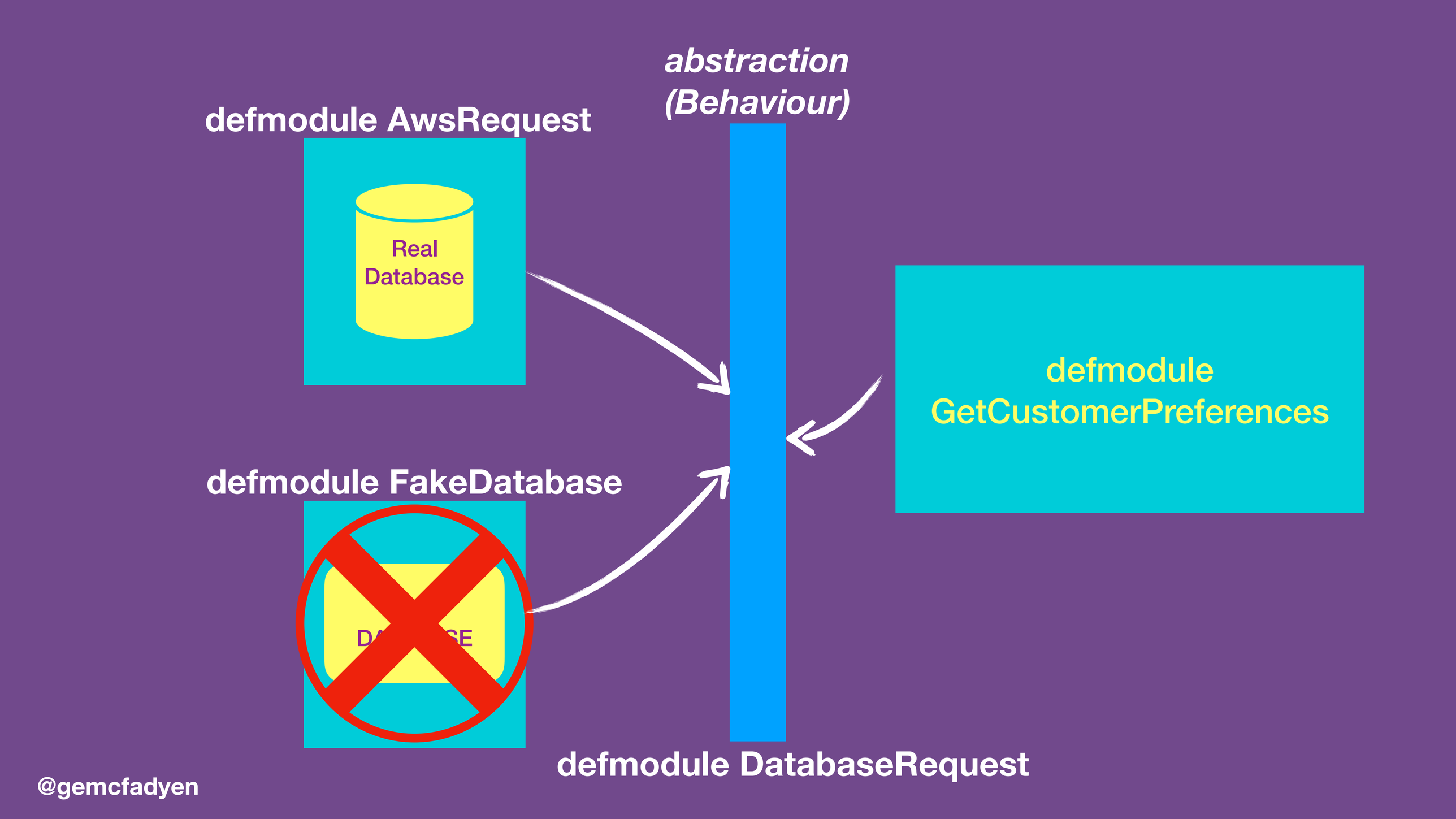
❌ Fake implementations can diverge from the real implementations

# Mocking Library

***https://github.com/plataformatec/mox***

# Add dependency

```elixir
defp deps do
  [
    ...
    {:mox, "~> 0.3.1"},
    ...
  ]
end
```


abstraction
(Behaviour)
defmodule AwsRequest
Real
Database
defmodule
GetCustomerPreferences
defmodule FakeDatabase
defmodule DatabaseRequest

# Wrap ‘real’ implementation

```elixir
defmodule Database.AwsRequest do
  @behaviour Database.Request

  @impl true
  def request(op, config_overrides \\ []) do
    ExAws.request(op, config_overrides)
  end
end
```

# Configure Implementations - prod config

```
1  config :customer_preference_api,
2   aws_request: Database.AwsRequest
```

# Configure Implementations - test config

```
1  config :customer_preference_api,
2   aws_request: Database.MockRequest,
```

# Test Expectations

```
1 ExUnit.start()
2
3 Mox.defmock(Database.MockRequest, for: Database.Request)
```

# Configure Implementations

```elixir
defmodule Core.Preferences.GetCustomerPreferences do
  @aws_request Application.get_env(:customer_preference_api,
                                   :aws_request)

  def for(customer_id) do
    ExAws.Dynamo.get_item("customer-preferences",
                          %{id: customer_id})
    |> @aws_request.request
    |> decode
  end
  ...
end
```

# Test Expectations

```elixir
defmodule Core.Preferences.GetCustomerPreferencesTest do
  use ExUnit.Case

  import Mox
  @mock_request Database.MockRequest

  expect(@mock_request, :request, fn _ -> {:ok, canned_result()} end)

  {:ok, result} = Core.Preferences.GetCustomerPreferences.for("1")

  assert result.id == "uuid-1"
  assert result.preferences["accommodation"]["apartment"]["bedrooms"] == 2
  assert result.preferences["accommodation"]["hotel"]["catering"] == "catered"
 end

 def canned_result do
   ...
 end
end
```

# Test Expectations

```elixir
defmodule Core.Preferences.GetCustomerPreferencesTest do
  use ExUnit.Case
  import Mox
  @mock_request Database.MockRequest

  test "fetches preferences from database" do
    expect(@mock_request,
           :request,
           fn _ -> {:ok, canned_result()} end)

    assert result.preferences["accommodation"]["hotel"]["catering"] == "catered"
  end

  def canned_result do
    ...
  end
end
```

Define expectation

# Test Expectations

```
  1) test fetches preferences from database (Core.Preferences.GetCustomerPrefere
ncesTest)
     test/core/preferences/get_customer_preferences_test.exs:7
     ** (Mox.UnexpectedCallError) no expectation defined for Database.MockReques
t.request/1 in process #PID<0.591.0>
     code: {:ok, result} = Core.Preferences.GetCustomerPreferences.for("1")
     stacktrace:
       (mox) lib/mox.ex:438: Mox.__dispatch__/4
       (customer_preference_api) lib/core/preferences/get_customer_preferences.e
x:6: Core.Preferences.GetCustomerPreferences.for/1
       test/core/preferences/get_customer_preferences_test.exs:9: (test)
```

# Mocking Libraries

✅ Easy to setup different scenarios

✅ Fake behaviour lives with the test

✅ Don’t have to maintain any ‘fake’ implementations

❌ Mix spits out some warnings about the mock modules not physically existing

# Dependency Inversion

## SOLID

Depend on abstractions

Provide to the application, a specific implementation of a dependency to use depending on the circumstance


Can
you
write
SOLID
Elixir?
@gemcfadyen


Can you write SOLID Elixir?
yes!
@gemcfadyen

# Does it make for idiomatic Elixir?

# Do we need a set of functional design principles?

# SOLID ✅

## Functions

# SOLID ✅

## Higher Order Functions

# SOLID

## Pure Functions & Referential Transparency

# Pure Functions

**Deterministic with no side effects**

**input -> output**

```
def function(input) do
  output
end
```

# Pure Functions

```
def add(number, number) do
  number + number
end
```

# Pure Functions

```elixir
def add(number, number) do
  number + number
end
```

```elixir
def lucky_number() do
  add(2, 2)
  |> add(2)
  |> add(1)
end
```

# Referential Transparency

```elixir
def add(number, number) do
  number + number
end
```

```elixir
def lucky_number() do
  add(2, 2)
  |> add(2)
  |> add(1)
end
```

7

# SOLID ✅

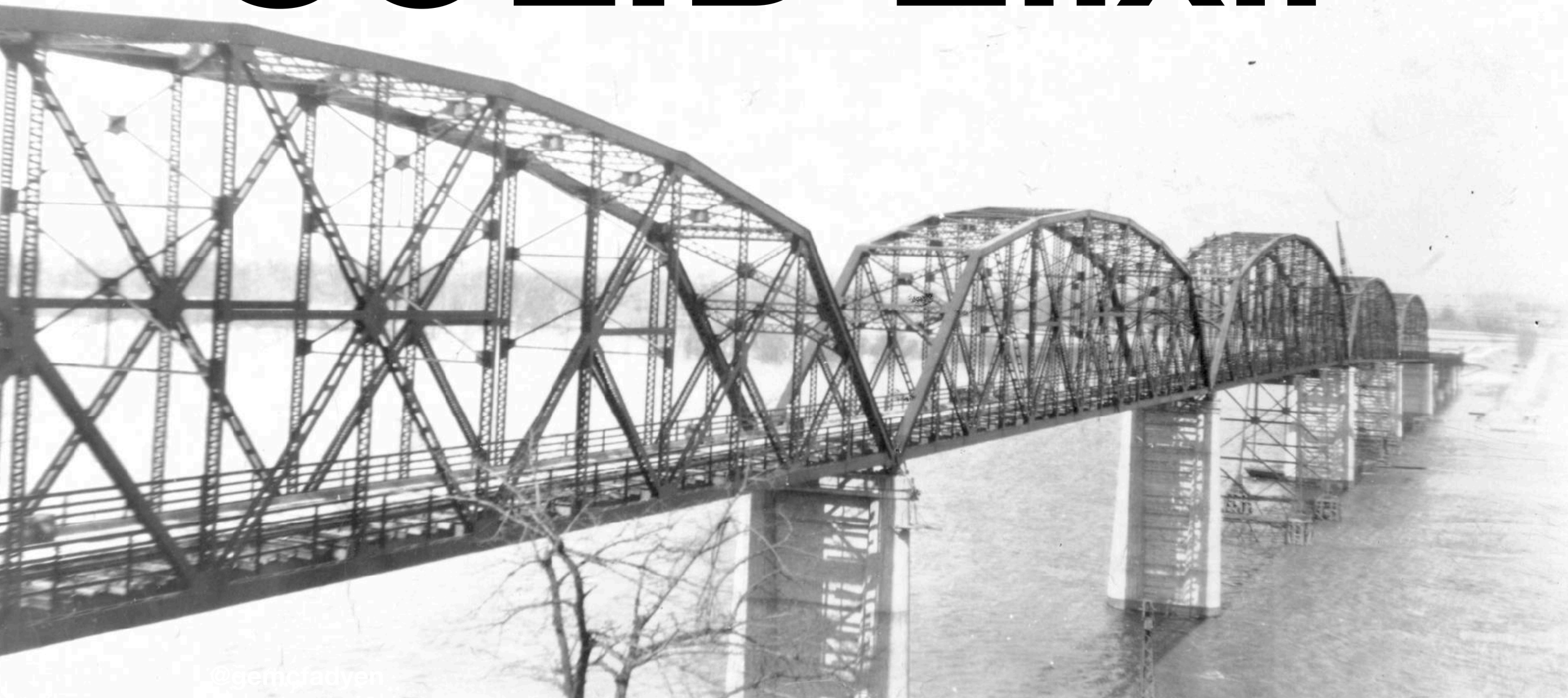
## Behaviour

# SOLID ✅

## Abstractions

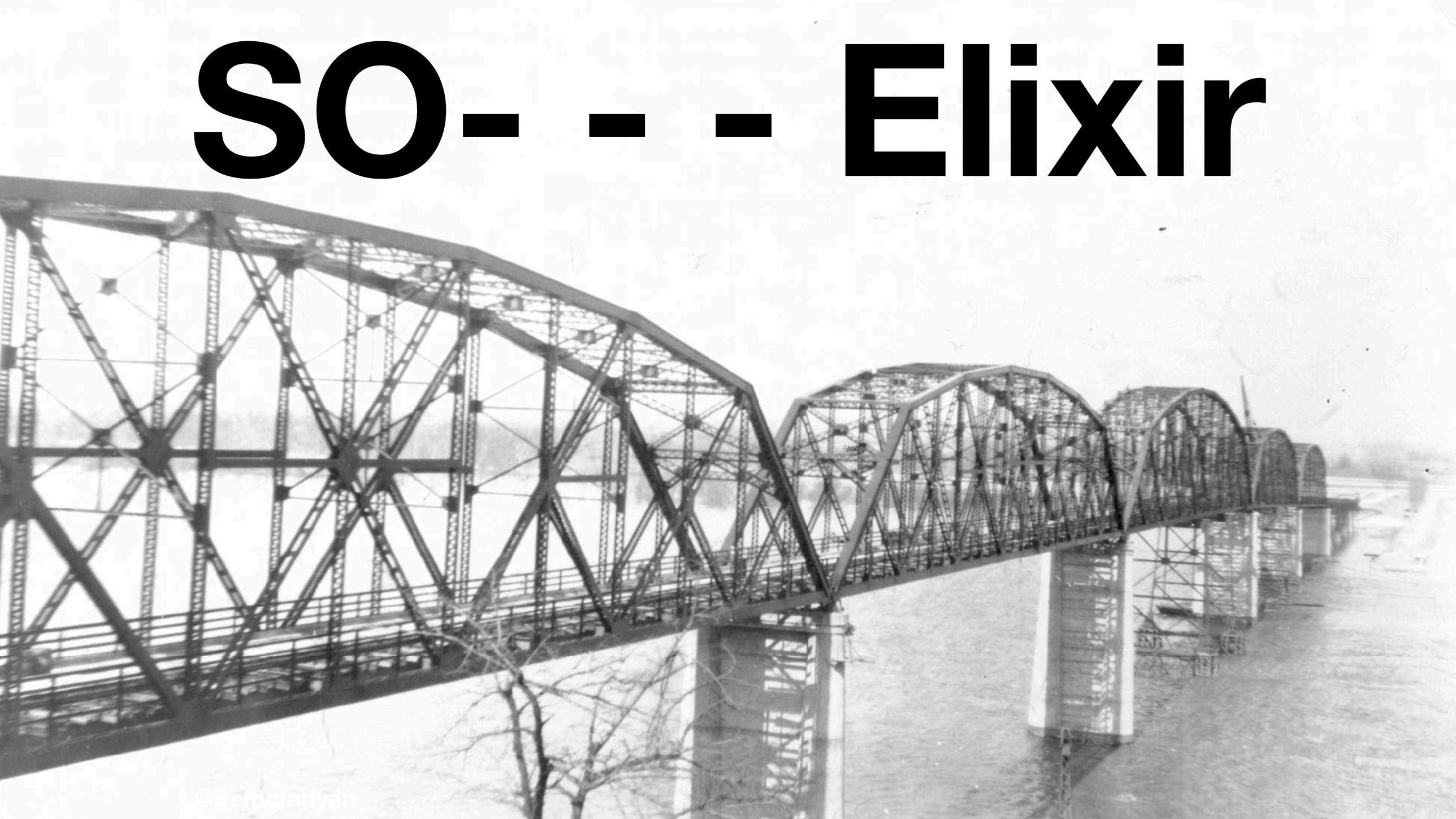
Immutability
Recursion
Pattern Matching
Currying
...

# SOLID Elixir



# S- - - - Elixir

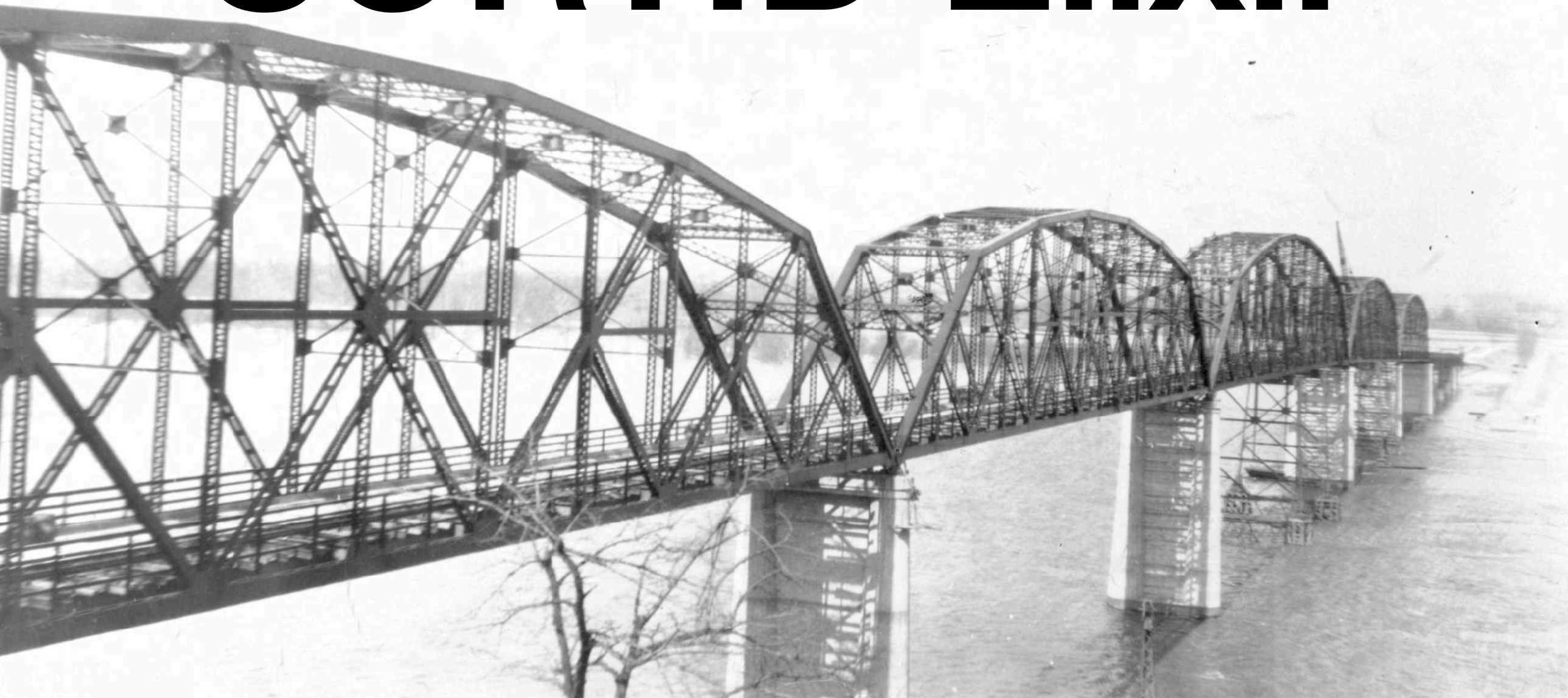
# SO- - - Elixir



# SORT- - Elixir

# SORTI- Elixir

# SORTID Elixir



# Thank-you!