A Practical Guide to Protocols

Kevin Rockwood

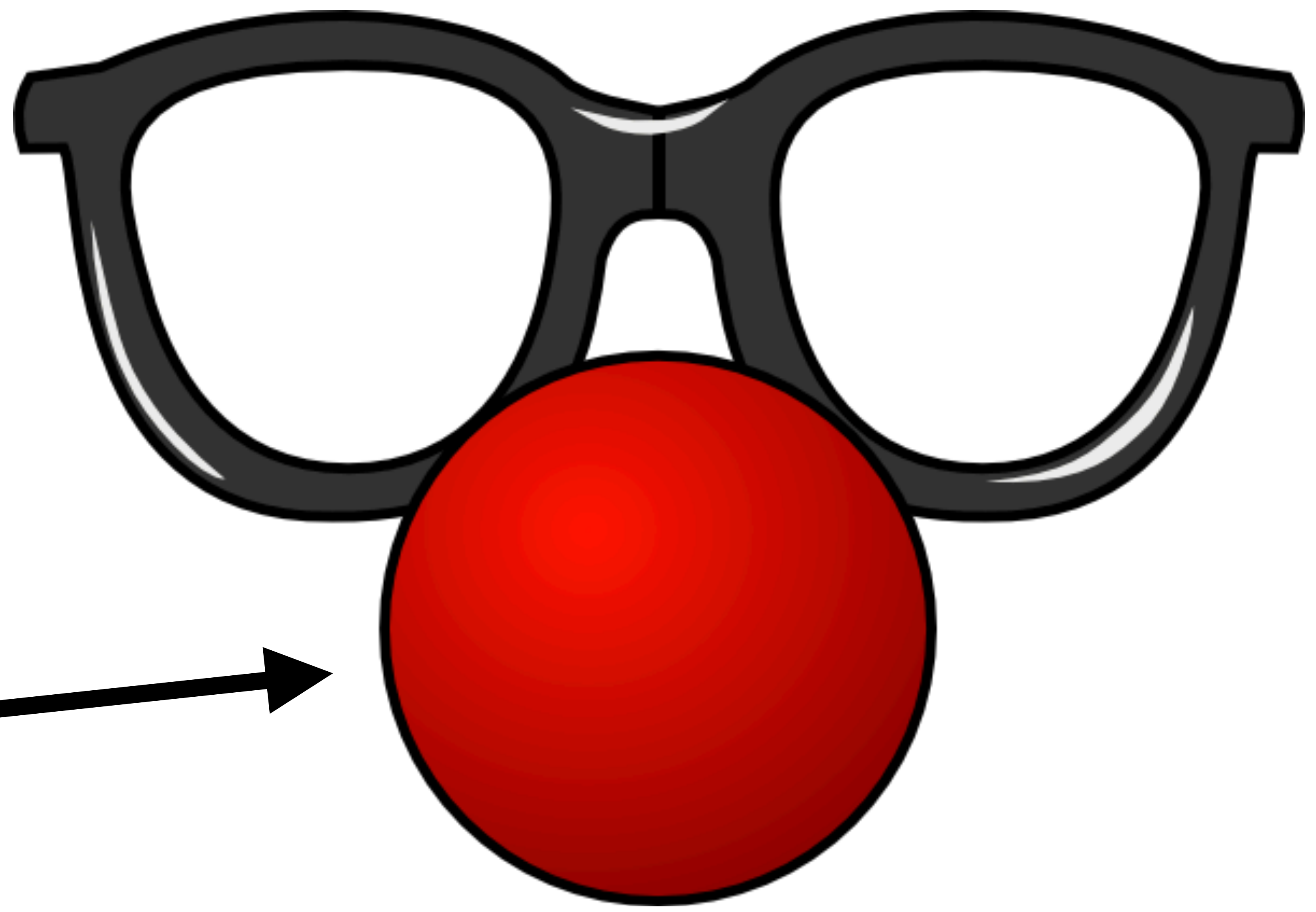
Hi, I'm Kevin

I live in Seoul

Skelter Labs

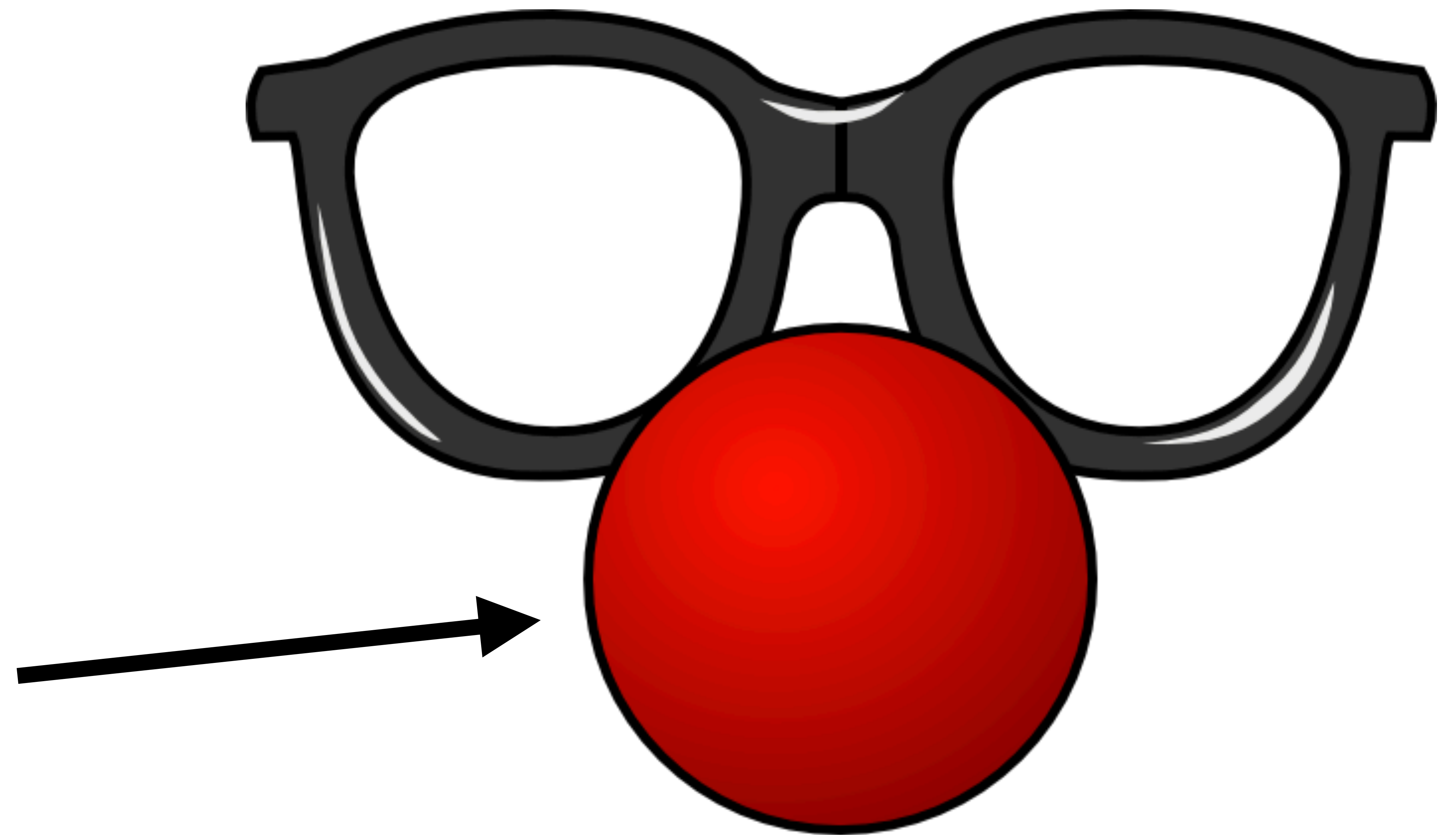
Protocols

What is this? →



squeezable honking red ball nose adornment

What is this?



clown nose

Abstraction

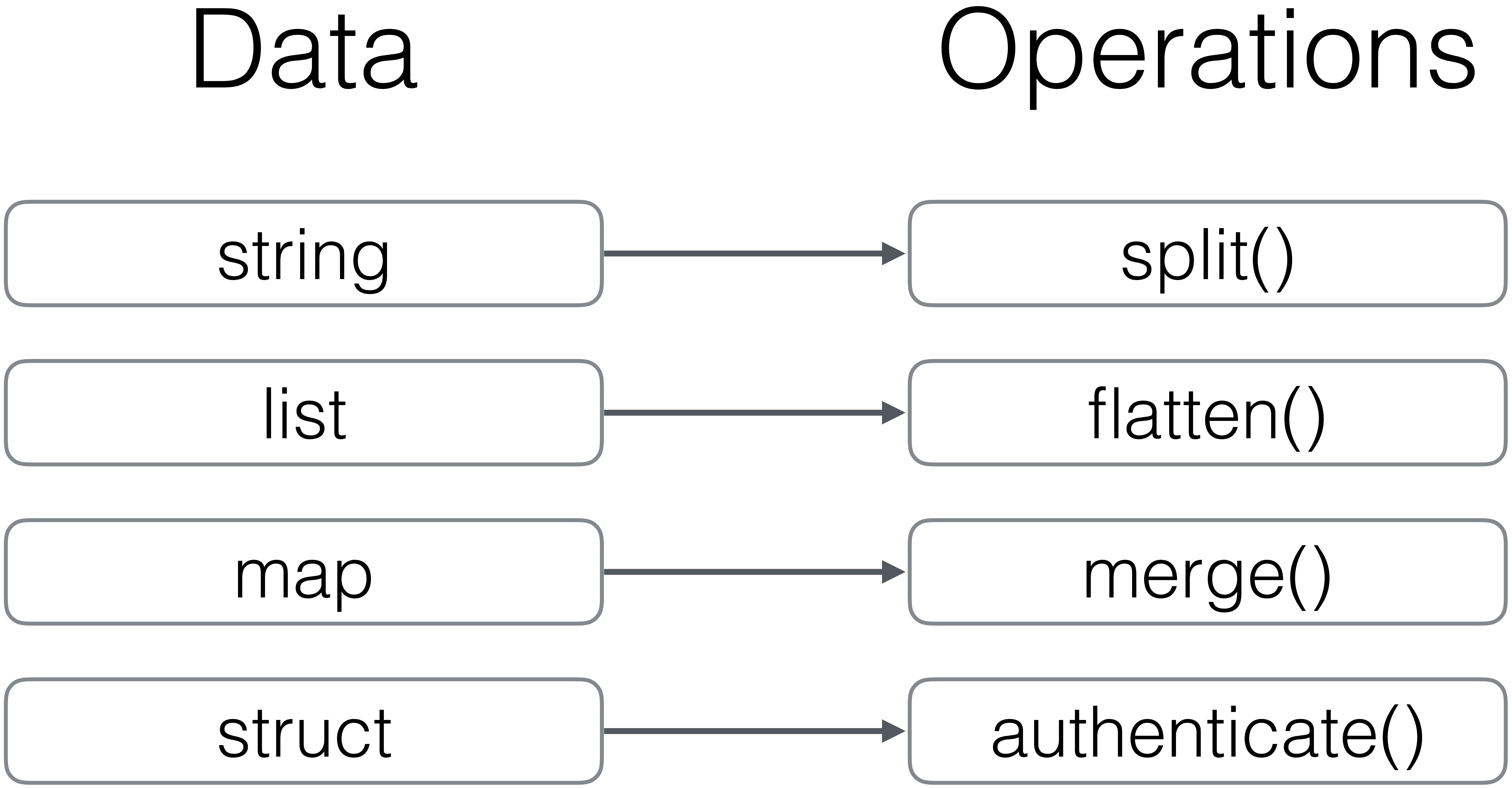
Abstraction

```
fetch_user_record_via_email_assert_equality_of_encrypted_passw
ord_via_bcrypt_algorithm(%{email: email, password: password})
```

```
authenticate(%{email: email, password: password})
```

Programs are made of
two elements

Data & Operations

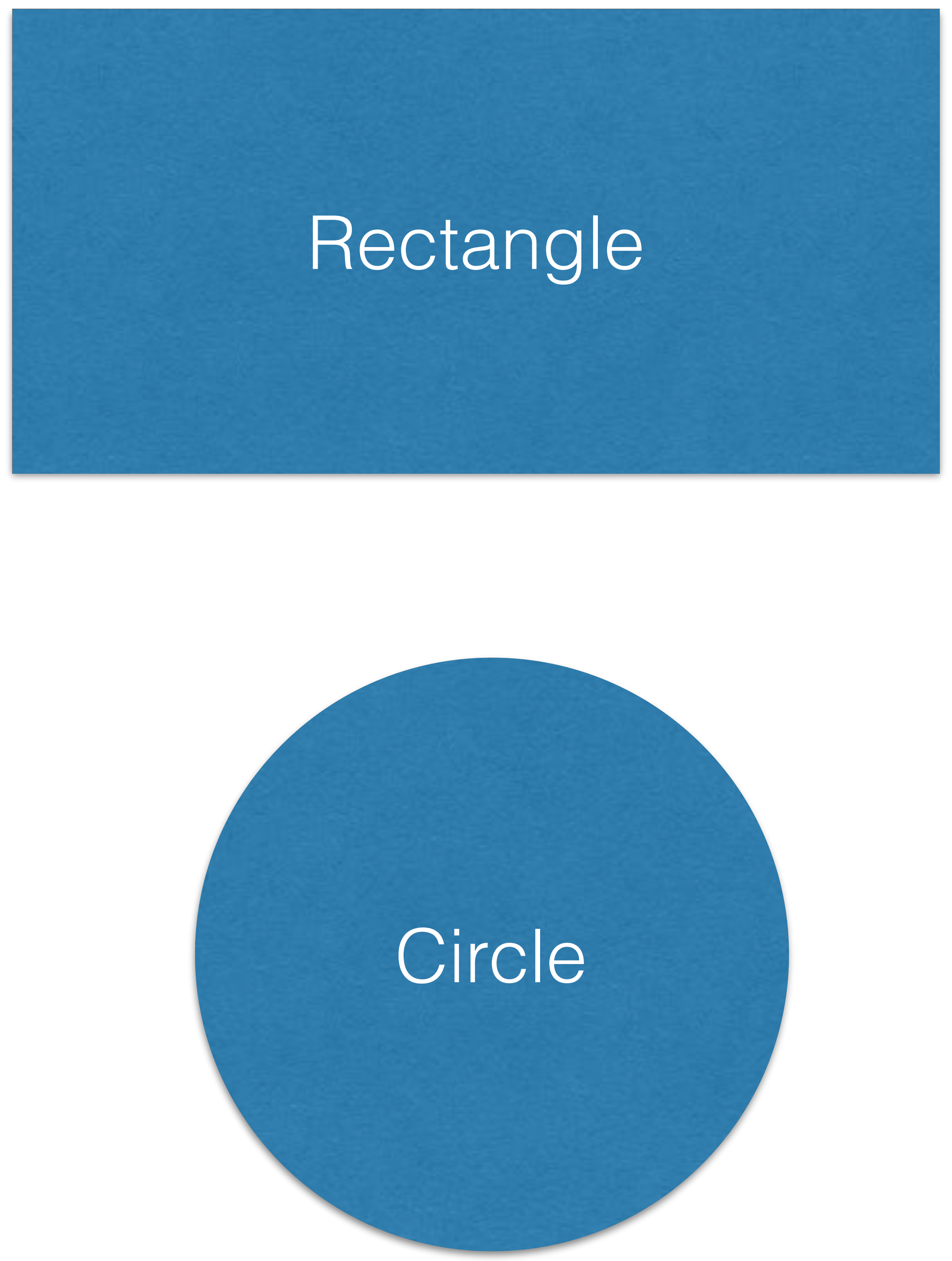

Data
Operations
string
split()
list
flatten()
map
merge()
struct
authenticate()

Data & Operations

The Expression Problem

We want to add new data (types) and new operations without modifying existing ones

Data



Operations

area()

perimeter()

```ruby
class A::Rectangle
  def initialize(width, height)
    @width = width
    @height = height
  end

  def area
    @width * @height
  end
end
```

```ruby
class B::Printer
  def self.print(shape)
    puts shape.area
  end
end
```

```ruby
class C::Circle
  def initialize(radius)
    @radius = radius
  end

  def area
    Math::PI * @radius ** 2
  end
end
```

- Add Data? 👍

```ruby
class A::Rectangle
  def initialize(width, height)
    @width = width
    @height = height
  end

  def area
    @width * @height
  end
end
```

```ruby
class B::Printer
  def self.print(shape)
    puts shape.perimeter
  end
end
```

```ruby
class C::Circle
  def initialize(radius)
    @radius = radius
  end

  def area
    Math::PI * @radius ** 2
  end
end
```

- Add Data? 👍
- Add Operation? 👎

```elixir
defmodule A.Rectangle do
  defstruct length: 0, width: 0

  def area(%{length: length, width: width}) do
    length * width
  end
end
```

```elixir
defmodule B.Printer do
  def print(%A.Rectangle{} = shape) do
    IO.puts A.Rectangle.area(shape)
  end
end
```

```elixir
defmodule C.Circle do
  defstruct radius: 0

  def area(%{radius: radius}) do
    :math.pi() * :math.pow(radius, 2)
  end
end
```

- Add Data? 👎

```elixir
defmodule A.Rectangle do
  defstruct length: 0, width: 0

  def area(%{length: length, width: width}) do
    length * width
  end
end
```

```elixir
defmodule B.Printer do
  def print(shape) do
    IO.puts perimeter(shape)
  end
end
```

```elixir
defmodule C.Circle do
  defstruct radius: 0

  def area(%{radius: radius}) do
    :math.pi() * :math.pow(radius, 2)
  end
end
```

- Add Data? 👎

```elixir
defmodule A.Rectangle do
  defstruct length: 0, width: 0

  def area(%{length: length, width: width}) do
    length * width
  end
end
```

```elixir
defmodule B.Printer do
  def print(shape) do
    IO.puts perimeter(shape)
  end

  def perimeter(%A.Rectangle{length: length, width: width})
    2 * length + 2 * width
  end
  def perimeter(%C.Circle{radius: radius}) do
    2 * :math.pi() * radius
  end
end
```

```elixir
defmodule C.Circle do
  defstruct radius: 0

  def area(%{radius: radius}) do
    :math.pi() * :math.pow(radius, 2)
  end
end
```

- Add Data? 👎
- Add Operation? 👍

The Expression Problem

Object Oriented

- Data: 👍
- Operation: 👎

Functional

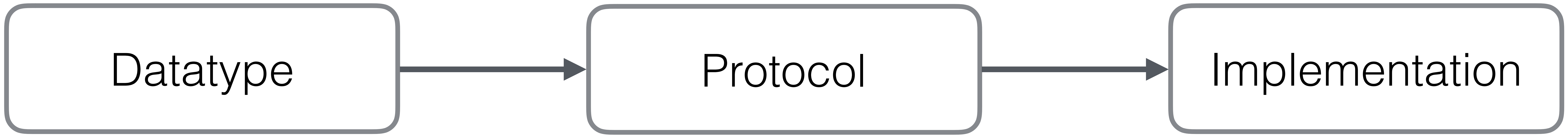
- Data: 👎
- Operation: 👍

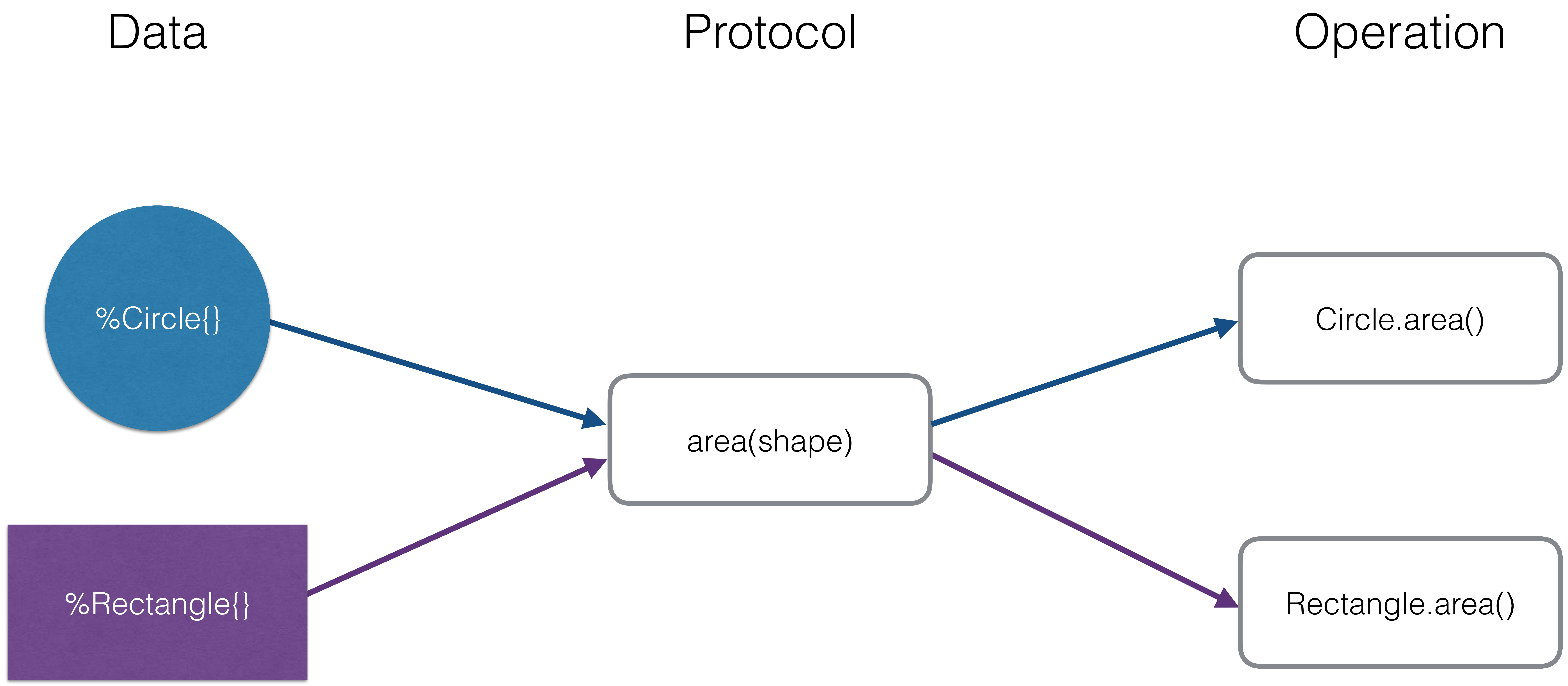
Protocols

Solve the expression problem by decoupling
the definition of an operation from its implementation

Influenced by

Clojure


Datatype
Protocol
Implementation


Data
Protocol
Operation
%Circle{}
%Rectangle{}
area(shape)
Circle.area()
Rectangle.area()

```elixir
defprotocol Blankable do
  def blank?(term)
end

defimpl Blankable, for: BitString do
  def blank?(""), do: true
  def blank?(_), do: false
end

defimpl Blankable, for: Map do
  def blank?(map), do: map_size(map) == 0
end

defmodule Post do
  defstruct [:title, :body]

  defimpl Blankable do
    def blank?(%{body: nil}), do: true
    def blank?(%{body: _}), do: false
  end
end
```

```
> Blankable.blank?("foo")
false

> Blankable.blank?(%{body: nil})
false

> Blankable.blank?(%Post{body: nil})
true

> Blankable.blank?(:foo)
** (Protocol.UndefinedError) protocol Blankable
   not implemented for :foo
```

Any

```elixir
defprotocol Blankable do
  @fallback_to_any true
  def blank?(term)
end

defimpl Blankable, for: Any do
  def blank?(_), do: false
end
```

```
> Blankable.blank?(:foo)
false
```

```elixir
defprotocol Blankable do
  def blank?(term)
end


defimpl Blankable, for: Any do
  def blank?(_), do: false
end


defmodule Post do
  @derive [Blankable]
  defstruct [:title, :body]
end
```

```
> Blankable.blank?(%Post{})
false
```

Back to the Expression Problem

```elixir
defprotocol A.Area do
  def calc(shape)
end

defmodule A.Rectangle do
  defstruct length: 0, width: 0

  defimpl A.Area do
    def calc(%{length: length, width: width}) do
      length * width
    end
  end
end
```

```elixir
defmodule B.Printer do
  def print(shape) do
    IO.puts A.Area.calc(shape)
  end
end
```

```elixir
defmodule C.Circle do
  defstruct radius: 0

  defimpl A.Area do
    def calc(%{radius: radius}) do
      :math.pi() * :math.pow(radius, 2)
    end
  end
end
```

- Add Data? 👍

```elixir
defprotocol A.Area do
  def calc(shape)
end

defmodule A.Rectangle do
  defstruct length: 0, width: 0

  defimpl A.Area do
    def calc(%{length: length, width: width}) do
      length * width
    end
  end
end
```

```elixir
defmodule B.Printer do
  def print(shape) do
    IO.puts B.Perimeter.calc(shape)  ←
  end
end

defimpl B.Perimeter, for: A.Rectangle do
  def calc(%{length: length, width: width}) do
    2 * length + 2 * width
  end
end

defimpl B.Perimeter, for: C.Circle, do: ...
```

```elixir
defmodule C.Circle do
  defstruct radius: 0

  defimpl A.Area do
    def calc(%{radius: radius}) do
      :math.pi() * :math.pow(radius, 2)
    end
  end
end
```

- Add Data? 👍
- Add Operation? 👍

Core Protocols

String.Chars

Responsible for converting a data type to a binary

- `to_string/1`

```
> to_string(:foo)
"foo"
```

Collectable

Used to take values out of a collection

- `into/2`

```
> Enum.into([a: 1, b: 2], %{})
%{a: 1, b: 2}
```

Inspect

Responsible for converting any Elixir data structure into a pretty printed format

- `inspect/1`

```
> inspect(%Post{title: "My post"})
"%Post{title: \"My post\"}"
```

Enumerable

Used by `Enum` and `Stream` modules to interact with list-like structures

- `count/1`
- `member/2`
- `reduce/3`

```
> Enum.map([1, 2, 3], &(&1 * 2))
[2, 4, 6]
```

IEx.Info

Prints helpful info inside an IEx session

- `info/1`

```
> i :foo
Term
  :foo
Data type
  Atom
Reference modules
  Atom
Implemented protocols
  IEx.Info, Inspect, List.Chars, String.Chars
```

Protocols in the Wild

poison 3.1.0

An incredibly fast, pure Elixir JSON library

Maintainers

Devin Torres

Links

GitHub
Online documentation (download)

License

CC0-1.0

82 750 downloads this version

4 607 downloads yesterday

34 964 downloads last 7 days

2 799 861 downloads all time

Config

mix.exs

```
{:poison, "~> 3.1"}
```

Build Tools

mix

Owners

devinus

```
> Poison.encode(%{my_int: 1, my_atom: :two})
> {:ok, "{\"my_int\":1,\"my_atom\":\"two\"}"}


defprotocol Poison.Encoder do
  @fallback_to_any true

  def encode(value, options)
end

defimpl Poison.Encoder, for: Atom do
  def encode(nil, _),   do: "null"
  def encode(true, _),  do: "true"
  def encode(false, _), do: "false"

  def encode(atom, options) do
    Poison.Encoder.BitString.encode(Atom.to_string(atom), options)
  end
end

defimpl Poison.Encoder, for: MyPerson do
  def encode(%{name: name, age: age}, options) do
    Poison.Encoder.BitString.encode("#{name} (#{age})", options)
  end
end
```

scrivener 2.3.0

Pagination for the Elixir ecosystem

Maintainers

Drew Olson

Links

github
Online documentation (download)

License

MIT

3 351 downloads this version

578 downloads yesterday

4 025 downloads last 7 days

208 627 downloads all time

Config

mix.exs

```
{:scrivener, "~> 2.3"}
```

Build Tools

mix

Owners

drewolson

```
> Repo.paginate(User, %{page_size: 10})
%Scrivener.Page{entries: [...], page_number: 1, page_size: 10, total_entries: 100, total_pages: 10}

defprotocol Scrivener.Paginater do
  def paginate(pageable, config)
end

defimpl Scrivener.Paginater, for: Ecto.Query do
  import Ecto.Query

  def paginate(query, %{page_size: page_size, page_number: page_number, module: repo}) do
    query = from q in query,
      limit: page_size,
      offset: page_number * page_size

    %Scrivener.Page{entries: repo.all(query) page_size: page_size, ...}
  end
end

defimpl Scrivener.Paginater, for: MyThing do
  def paginate(my_thing, %{page_size: page_size, page_number: page_number}) do
    ...
  end
end
```

Application Code

```elixir
> %FetchUsersMessage{role: "admin"} |> SoapApi.send_message()

defprotocol SoapMessage do
  def build_request(soap_message)
  def parse_response(soap_message, response)
end

defmodule SoapApi do
  def send_message(message) do
    request = SoapMessage.build_request(message)
    response = Http.post(request)
    SoapMessage.parse_response(message, response)
  end
end

defmodule FetchUsersMessage do
  defstruct role: ""

  defimpl SoapMessage do
    def build_request(%{role: role}) do: # build xml

    def parse_response(%{role: role}, response), do: # parse xml
  end
end
```

Protocol or Behavior

Can each implementation of my operation take the same data type?

```
> message = %Message{to: "george@example.com", subject: "Hi George!" body: "..."}
> Mailer.deliver_now(message)


defmodule Mailer do
  def deliver_now(message) do
    get_current_adapter().deliver(message)
  end
end


defmodule SMTPAdapter do
  deliver(message), do: ...
end

defmodule SendgridAdapter do
  deliver(message), do: ...
end

defmodule TestAdapter do
  deliver(message), do: ...
end
```

Don't get carried away

Abstraction