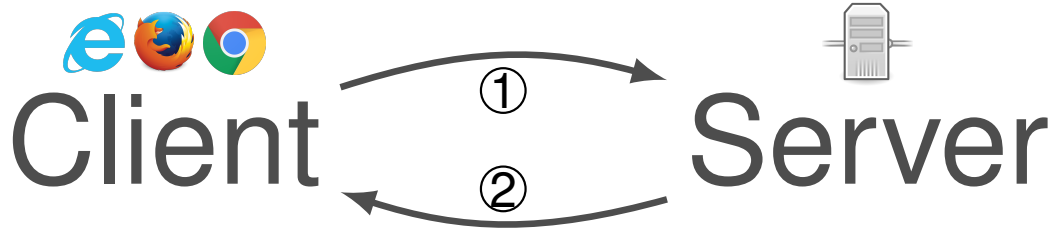
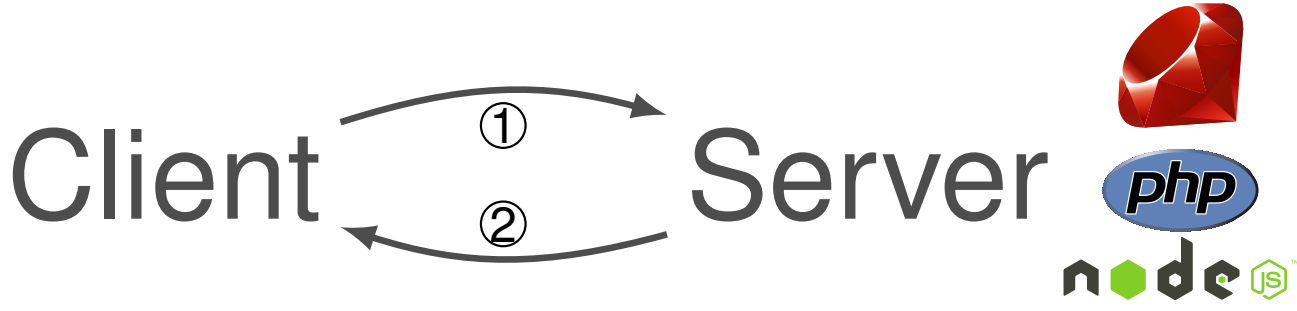
# ELIOM

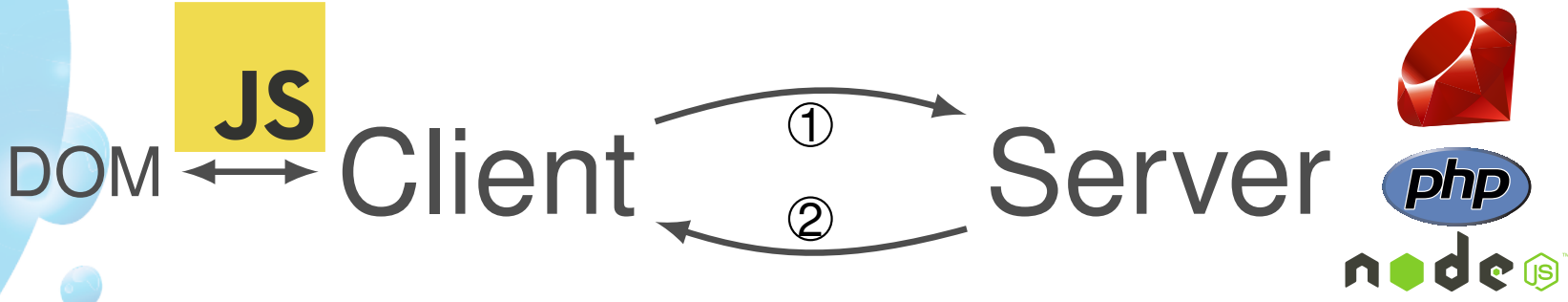
## Tierless Web programming from the ground up

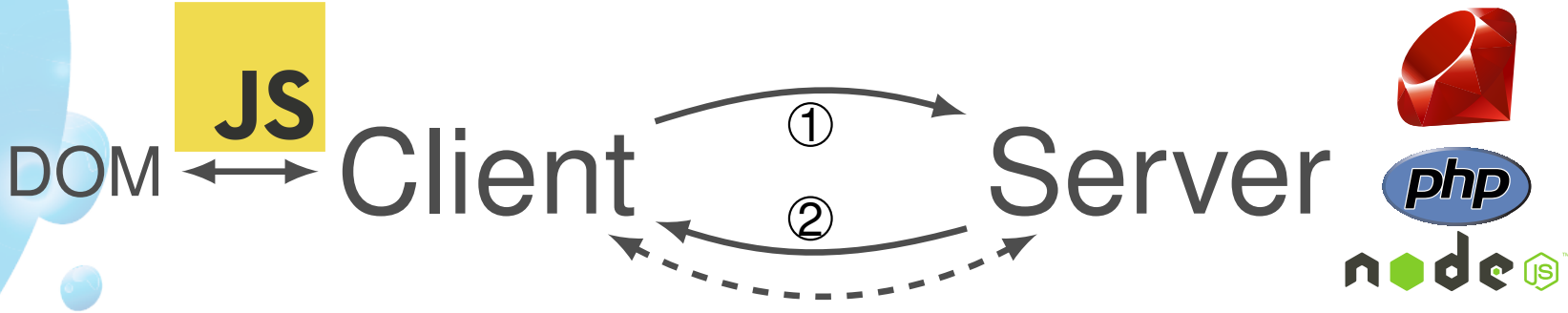
Gabriel RADANNE Jérôme VOUILLON
Vincent BALAT Vasilis PAPAVASILEIOU

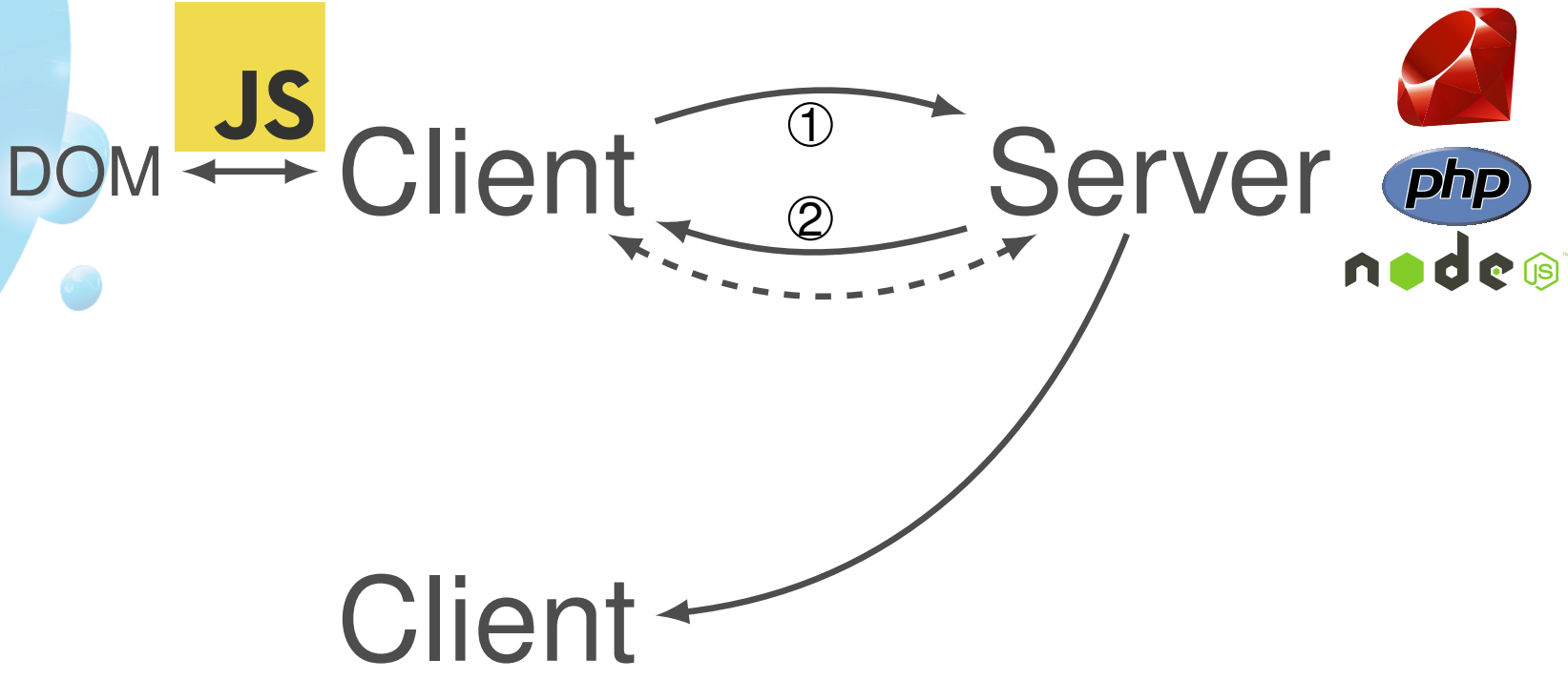
# Evolution of the Web

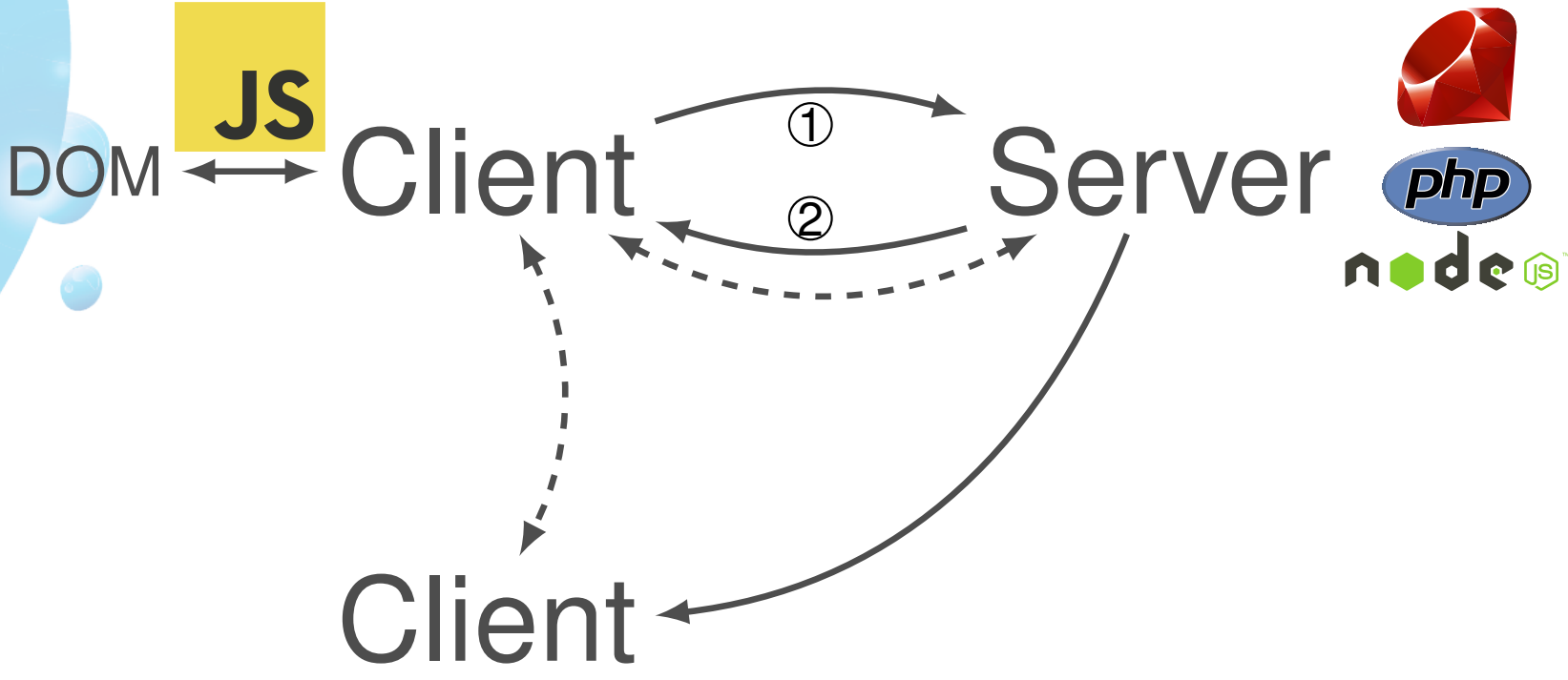


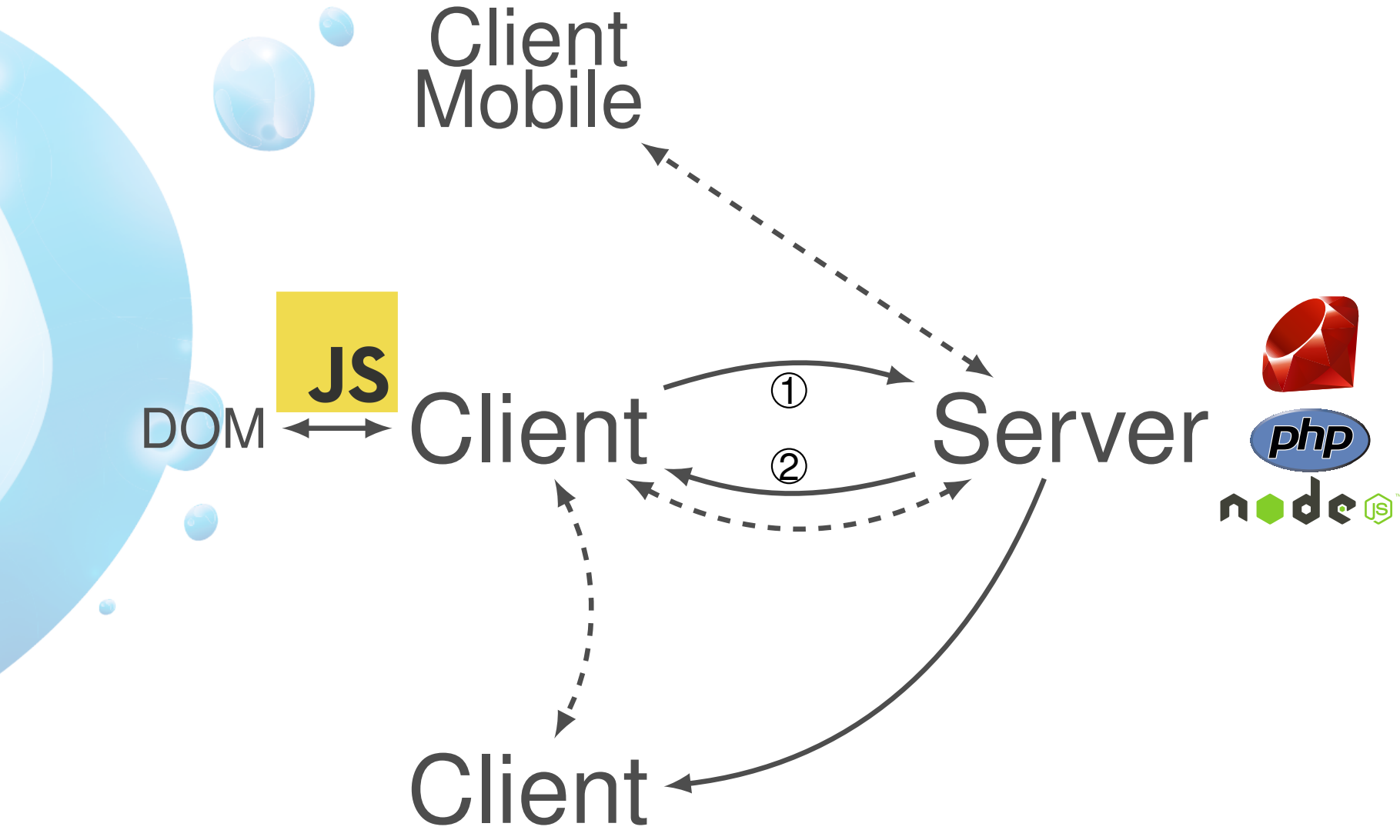

Client
①
②
Server
php
node

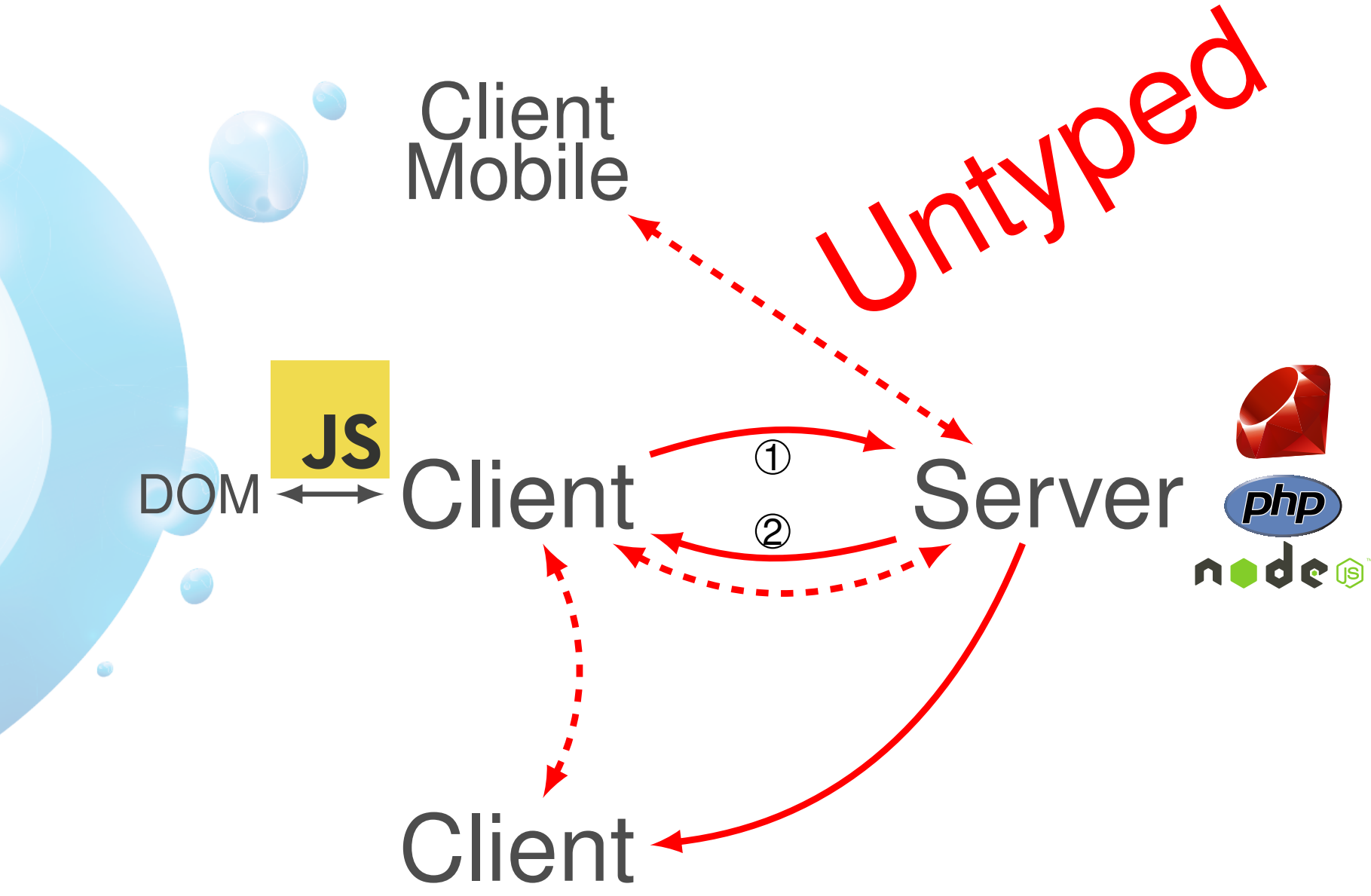

JS
DOM
Client
①
②
Server
php
node


JS
DOM
Client
①
②
Server
php
node

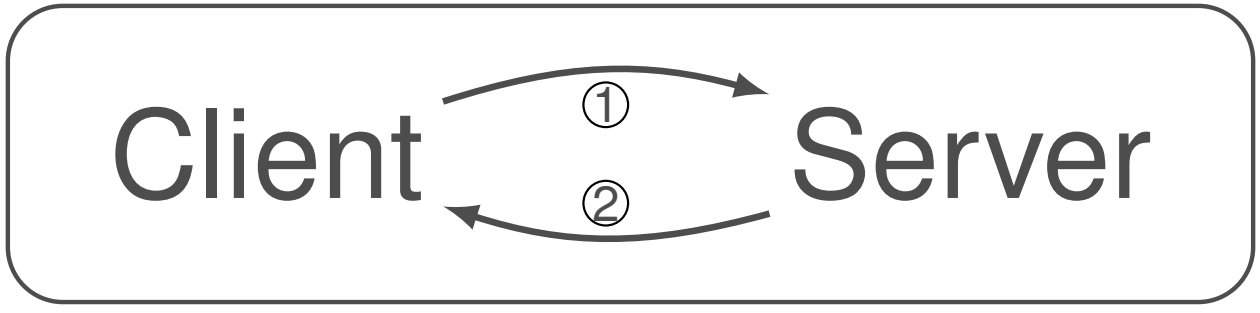

JS
DOM
Client
①
②
Server
Client


JS
DOM
Client
①
②
Server
Client

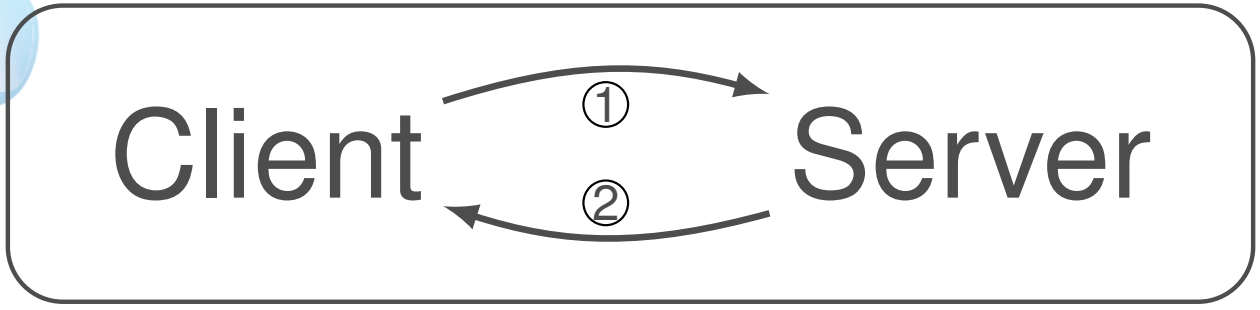

Client
Mobile
JS
DOM
Client
①
②
Server
php
node
Client


Client
Mobile
Untyped
JS
DOM
Client
①
②
Server
php
node
Client

One language for everything



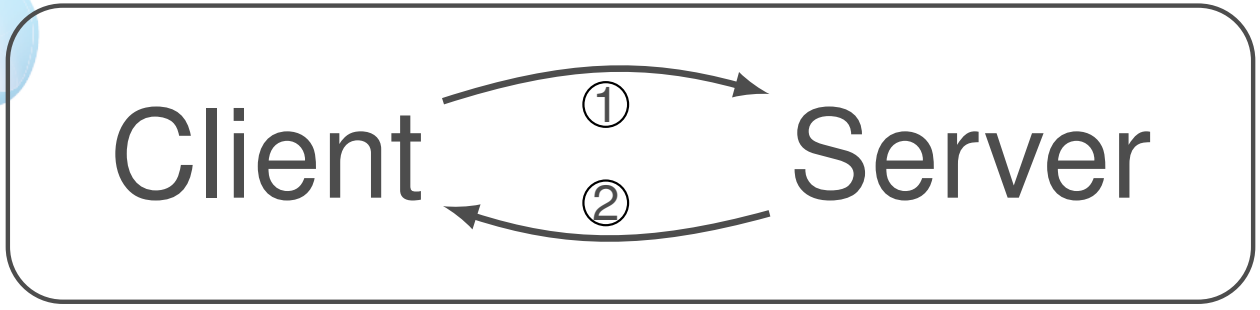
One language for everything




Tierless languages:

- LINKS
- HOP
- UR/WEB
- ELIOM

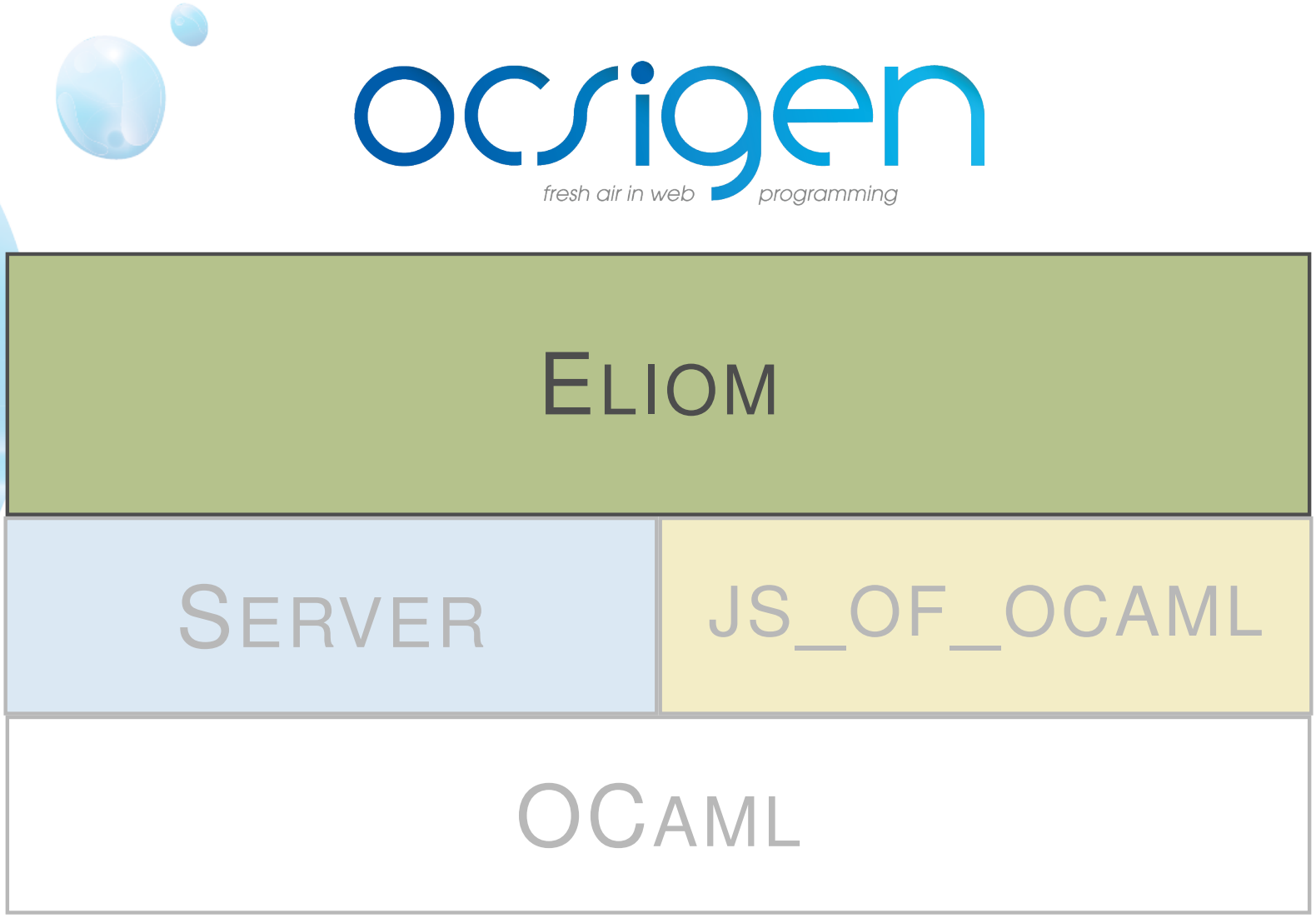
One language for everything




Tierless languages:

- LINKS
- HOP
- UR/WEB
- **ELIOM**

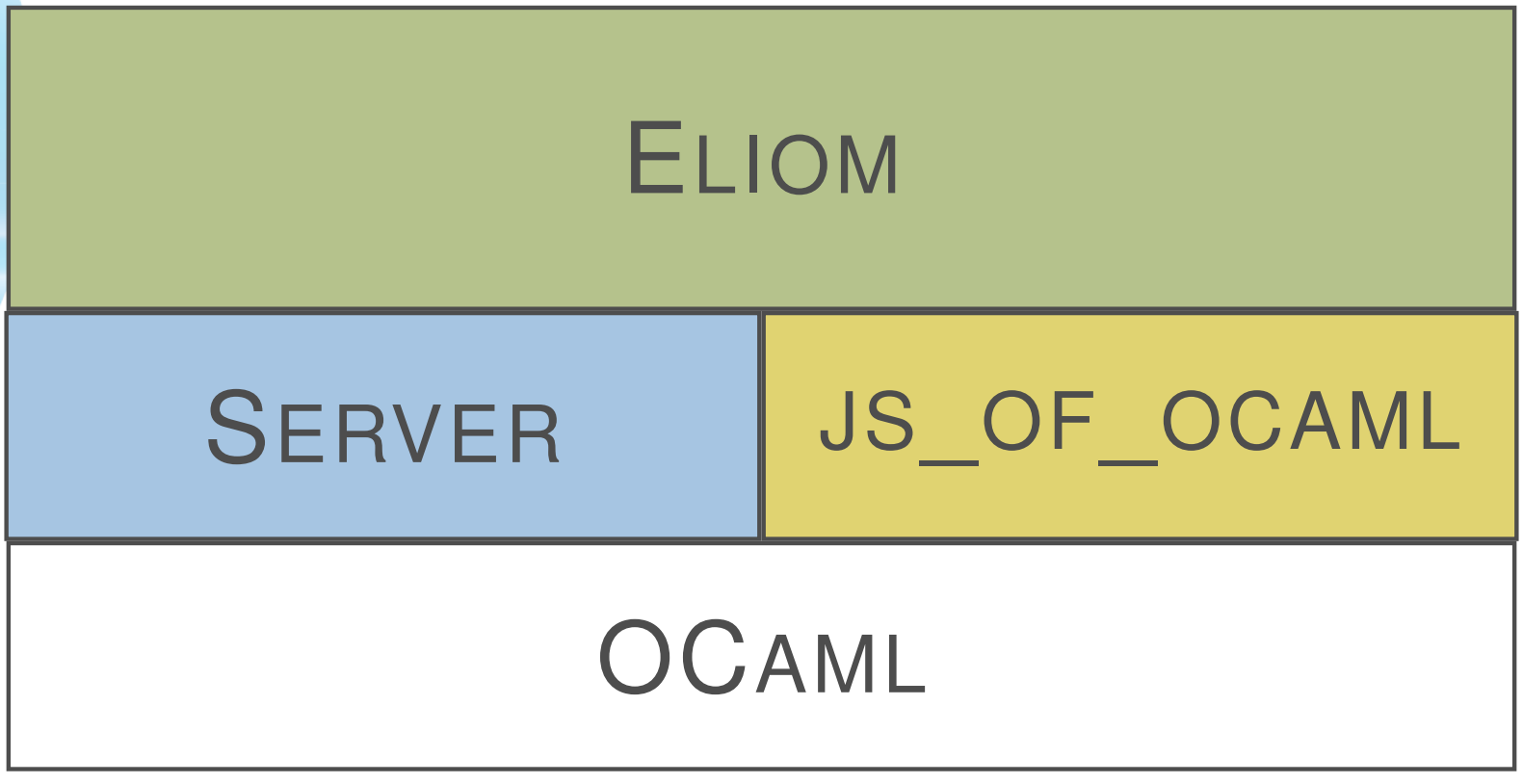
# The OCSIGEN project



# The OCSIGEN project

ocsigen

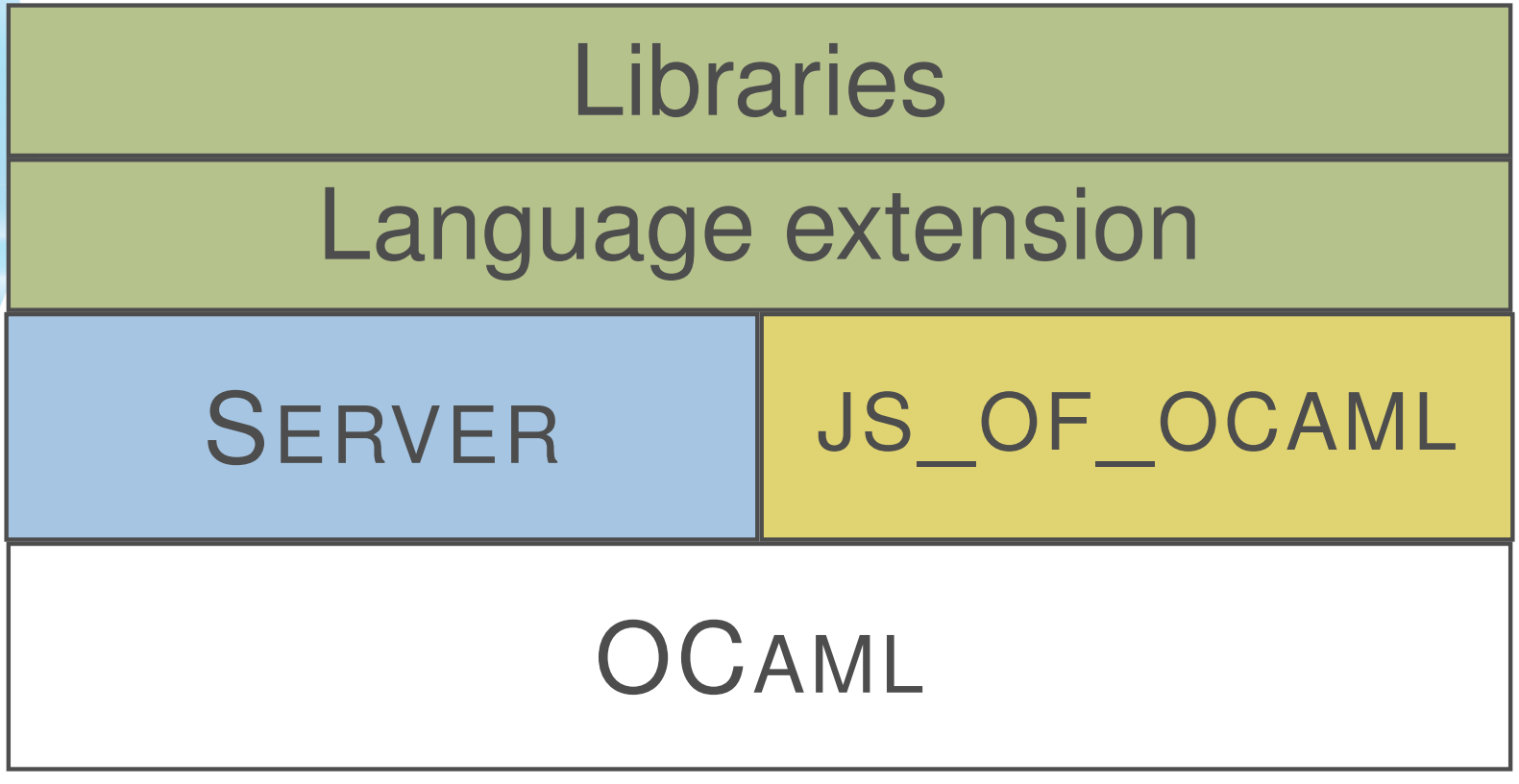
fresh air in web programming



# The OCSIGEN project

ocsigen

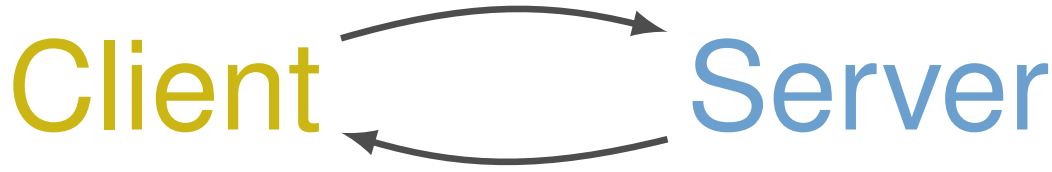
fresh air in web programming



1. ELIOM's language extension

2. Examples of libraries in ELIOM
   - Widgets
   - APIs

# Client and Server annotations




Location annotations allow to use client and server code *in the same program*.

```
1 let%server s = ...
2
3 let%client c = ...
4
5 let%shared sh = ...
```

The program is sliced during compilation.

# Building fragments of client code inside server code

Fragments of client code can be included inside server code.

```
1 let%server x : int fragment = [%client 1 + 3 ]
```

# Building fragments of client code inside server code

Fragments of client code can be included inside server code.

```
let%server x : int fragment = [%client 1 + 3 ]
```

```
let%server y = [ ("foo", x) ; ("bar", [%client 2]) ]
```

# Accessing server values in the client

Injections allow to use server values on the client.

```
1 let%server s : int = 1 + 2
2
3 let%client c : int = ~%s + 1
```

# Everything at once

We can combine injections and fragments.

```
1 let%server x : int fragment = [%client 1 + 3 ]
2
3 let%client c : int = 3 + ~%x
```

# Everything at once

We can combine injections and fragments.

```
1 let%server x : int fragment = [%client 1 + 3 ]
2
3 let%client c : int = 3 + ~%x
```

Gabriel Radanne and Jérôme Vouillon and Vincent Balat

ELIOM: A core ML language for Tierless Web programming

https://hal.archives-ouvertes.fr/hal-01349774

APLAS 2016

1. ELIOM's language extension

2. Examples of libraries in ELIOM
   - Widgets
   - APIs

# Counter widget

A button with a counter.

- HTML for the button is generated on the server.
- The button has a client-side state: the counter.
- When the button is pressed, the counter is incremented on the client.
- The button is parameterized by a client-side action.

# Counter widget

## counter.eliom

```
let%server counter action =
  let state = [%client ref 0 ] in
  button
    ~button_type:‘Button
    ~a:[a_onclick
          [%client fun _ ->
              incr ~%state;
              ~%action !(~%state) ]]
    [pcdata "Increment"]
```

# Counter widget

## counter.eliom

```
let%server counter action =
  let state = [%client ref 0 ] in
  button
    ~button_type:‘Button
    ~a:[a_onclick
          [%client fun _ ->
              incr ~%state;
              ~%action !(~%state) ]]
    [pcdata "Increment"]
```

## counter.eliomi

```
val%server counter: (int -> unit) fragment -> Html.t
```

# Execution

ELIOM code

```
1  let%server counter action =
2    let state = [%client ref 0 ] in
3    button
4      ~button_type:‘Button
5      ~a:[a_onclick
6            [%client fun _ ->
7                incr ~%state;
8                ~%action !(~%state) ]]
9      [pcdata "Increment"]
```

```
1  let fragment1 = ref 0
2
3  let fragment2 state action =
4    fun _ ->
5      incr (injection state);
6      (injection action)
7        !(injection state)
```

Client code

```
1  let counter action =
2    let state = fragment1 in
3    button
4      ~button_type:‘Button
5      ~a:[a_onclick
6            (fragment2 state action)]
7      [pcdata "Increment"]
```

Server code

# Execution

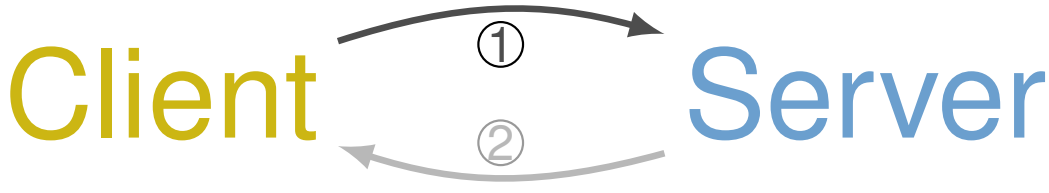
```
let fragment1 = ref 0

let fragment2 state action =
  fun _ ->
    incr (injection state);
    (injection action)
      !(injection state)
```

Client code

```
let counter action =
  let state = fragment1 in
  button
    ~button_type:‘Button
    ~a:[a_onclick
          (fragment2 state action)]
    [pcdata "Increment"]
```

Server code



# Execution

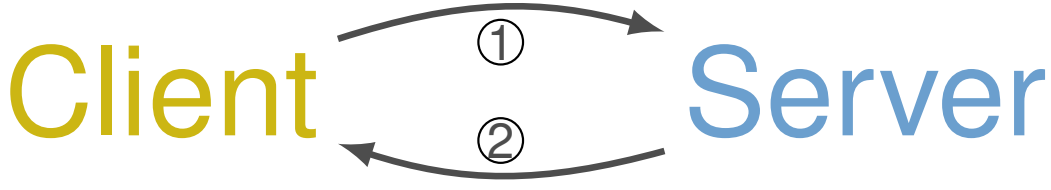
```
let fragment1 = ref 0

let fragment2 state action =
  fun _ ->
    incr (injection state);
    (injection action)
      !(injection state)
```

Client code

```
let counter action =
  let state = fragment1 in
  button
    ~button_type:‘Button
    ~a:[a_onclick
          (fragment2 state action)]
    [pcdata "Increment"]
```

Server code




```
<button type=button onclick=...>Increment</button>
```

"state" → "fragment1"; "action" → ...

# Counter widget

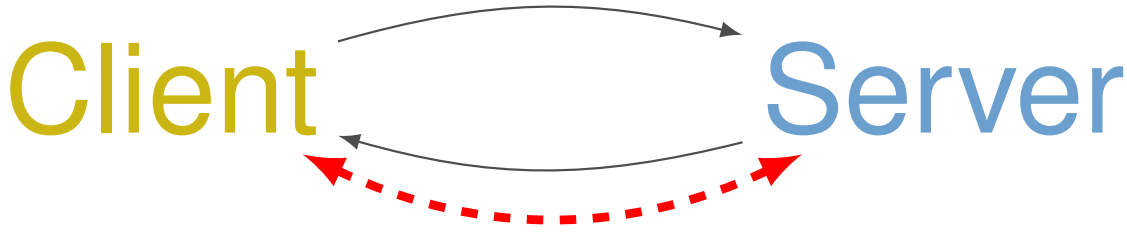
What if we want to save the state of the counter on the server ?

counter.eliomi

```
1 val%server counter: (int -> unit) fragment -> Html.t
```

# Remote Procedure Calls

Remote Procedure Call (or RPC) is the action of a client calling the server *without loading a new page* and potentially getting a value back.



# Remote Procedure Calls

A simplified RPC API:

**rpc.eliomi**

```
1 type%server ('i,'o) t
2 type%client ('i,'o) t = 'i -> 'o
3
4 val%server create : ('i -> 'o) -> ('i, 'o) t
```

# Remote Procedure Calls

A simplified RPC API:

## rpc.eliomi

```
type%server ('i,'o) t
type%client ('i,'o) t = 'i -> 'o

val%server create : ('i -> 'o) -> ('i, 'o) t
```

## An example using Rpc

```
let%server plus1 : (int, int) Rpc.t =
  Rpc.create (fun x -> x + 1)

let%client f x = ~%plus1 x + 1
```

# Converters

Converters are a way to *converts datatype between server and client*.
Here is a schematized signature for ~%, the injection operator:

```
type%shared serial (* A serialization format *)

type%server ('a, 'b) converter = {
  serialize : 'a -> serial ;
  deserialize : (serial -> 'b) fragment ;
}

(* Not a real type signature *)
val%client (~%) :
  ('a, 'b) converter -> 'a (* server *) -> 'b (* client *)
```

# Implementing RPC with converters

```
1  type%server ('i,'o) t = {
2    url : string ;
3    handler: 'i -> 'o ;
4  }
5
6  type%client ('i, 'o) t = 'i -> 'o
7
8  let serialize t = serialize_string t.url
9  let deserialize x =
10   let url = deserialize_string x in
11   fun i -> AJAX.get url i
```

# Widget + Rpc

We can now use `counter` and `Rpc` together!

```
1 val%server save_counter : int -> unit
2 val%server counter : (int -> unit) fragment -> Html.t
3
4 let%server save_counter_rpc : (int, unit) Rpc.t =
5   Rpc.create save_counter
6
7 let%server widget_with_save : Html.t =
8   let f = [%client ~%save_counter_rpc] in
9   counter f
```

# Conclusion

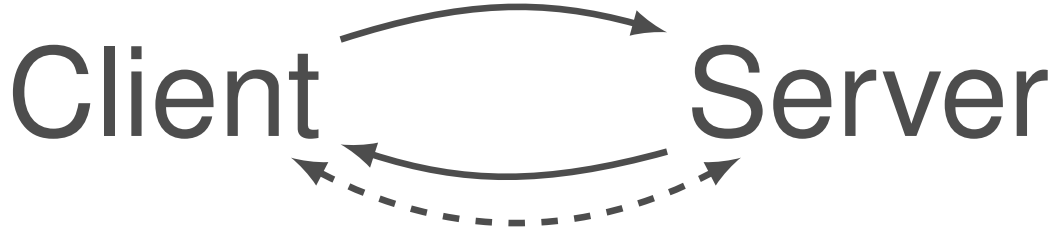
- Typesafe client-server communication
- The whole OCaml ecosystem
- Encapsulation and composition for widgets

In the paper: Implementation of bigger examples and other APIs

All of this is implemented and used: `https://ocsigen.org`

# Questions ?

# Remote Procedure Calls



# Client-server reactive HTML

# Bus/Multicast