VRROOM + BS-JSON

VRROOM

- Utility library for ReasonReact development
- https://github.com/glennsl/vrroom
- `npm install --save glennsl/vrroom`

TEXT

```
1: <div> {"Hello World" |> ReasonReact.stringToElement} </div>
```

TEXT

```
1: let str = ReasonReact.stringToElement
2: <div> {"Hello World" |> str} </div>
```

TEXT

```
1: let ste = ReasonReact.stringToElement
2: <div> {"Hello World" |> ste} </div>
```

```
1: let s = ReasonReact.stringToElement
2: <div> {"Hello World" |> s} </div>
```

```
1: let text = ReasonReact.stringToElement
2: <div> {"Hello World" |> text} </div>
```

TEXT

```
1: open Vrroom;
2:
3: <div> {"Hello World" |> text} </div>
```

TEXT

```
1: open Vrroom;
2:
3: <div> {42 |> Text.int} </div>
4:
5: <div> {4.2 |> Text.float} </div>
6:
7: <div> {"Hello World" |> Text.string} </div>
8:
9: <div> {Js.true_ |> Text.any} </div>
```

NOTHING

aka `ReasonReact.nullElement`

```
1: (alignIcon === `Left ? icon : nothing)
```



```
let nbsp = [%raw {|'\u00a0'|}];
```

```
switch package##description {
| ""          => nbsp
| description => description |> text
}
```

FRAGMENT

ReactJS equivalent:

```
1: <> ... </>
```

or

```
1: <React.Fragment> ... </React.Fragment>
```

FRAGMENT

```
1: <ul>
2:   ...
3:   <li class="label">Answer to everything:</li>
4:   <li class="value">42</li>
5:   ...
6: </ul>
```

FRAGMENT

```
1: <ul>
2:   {items
3:     |> Array.map(
4:          item => ReasonReact.arrayToElement([|
5:            <li className="label"> {item.label |> text} </li>
6:            <li className="value"> {item.value |> text} </li>
7:          |])
8:     |> ReasonReact.arrayToElement}
9: </ul>
```

Warning: Each child in an array or iterator should have a unique "key" prop.

FRAGMENT

"Official" solution:

```
ReasonReact.cloneElement(
    parent,
    ~props=Js.Obj.empty(),
    [| ... |]
);

```

```
ReasonReact.createDomElement(
    "div",
    ~props=Js.Obj.empty(),
    [| ... |]
);

```

FRAGMENT

Better solution:

```
1: let listItem = item =>
2:   <Fragment>
3:     <li className="label"> {item.label |> text} </li>
4:     <li className="value"> {item.value |> text} </li>
5:   </Fragment>
```

No key warning. No wrapping element

CONTROL.MAP

```
<ul>
  {
    switch items {
    | [||] => <li> {"no items" |> text} </li>
    | items =>
      items |> Array.map(name => <li> {name |> text} </li>)
            |> ReasonReact.arrayToElement
    }
  }
<ul>
```

CONTROL.MAP

```
1: <ul>
2:   <Control.Map items empty=<li> {"no items" |> text} </li> >
3:     ...(name => <li> {name |> text} </li>)
4:   </Control.Map>
5: <ul>
```

CONTROL.MAPLIST

```
<ul>
  {
    switch items {
    | [] => <li> {"no items" |> text} </li>
    | items =>
      items |> List.map(name => <li> {name |> text} </li>)
            |> Array.of_list
            |> ReasonReact.arrayToElement
    }
  }
<ul>


```

CONTROL.MAPLIST

```
1: <ul>
2:   <Control.MapList items
3:                    empty=(<li> {"no items" |> text} </li>)>
4:     ...(name => <li> {name |> text} </li>)
5:   </Control.MapList>
6: <ul>
```

CONTROL.IFSOME

```
{
  switch maybeError {
  | Some(error) => error |> text
  | None        => nothing
  }
}

```

```
<Control.IfSome option=maybeError>
  ...(error => error |> text)
</Control.IfSome>
```

CONTROL.IF

```
1: {showHello ? "Hello" |> text : nothing}
```

```
1: <Control.If cond=showHello>
2:     ...(() => "Hello" |> text)
3: </Control.If>
```

PURE

```
1: module Item = {
2:   let instance = ReasonReact.statelessComponent("Item");
3:   let make = (~label, _children) => {
4:     ...instance,
5:     render: _self =>
6:       <li> (label |> text) </li>
7:   }
8: };
```

PURE

```
1: module Item = {
2:   let make = pure((render, ~label) => render(
3:     <li> (label |> text) </li>
4:   ));
5: };
```

CHILDLESS

```
module MyComponent = {
  let make = (_children) => {
    ...
  }
}

```

```
<MyComponent>
  <div> {"some child" |> text} </div>
</MyComponent>

```

CHILDLESS

```
1: type nothing;
2: type childless = array(nothing);
```

```
1: module MyComponent = {
2:   let make = (_:childless) => {
3:     ...
4:   }
5: }
```

CHILDLESS

```
1: This has type:
2:   ReasonReact.reactElement array -> ReasonReact.reactElement
3: But somewhere wanted:
4:   childless -> 'a
```

CLASSNAME

```
let className =
  ClassName.(join([
    "tooltip",
    "s-hovered" |> if_(isHovered),
    maybeErrorClass |> fromOption
  ]));

```

```
<div className>
  ...
</div>
```

BS-JSON

- Decoding (and encoding) of JSON data structures
- Combinators: functional, compositional
- Takes more than a little bit from Elm's `Json.Decode`
- https://github.com/glennsl/bs-json
- `npm install --save @glennsl/bs-json`

ENCODER - INTERFACE

```
1: type encoder('a) = 'a => Js.Json.t
```

ENCODER - INTERFACE

```
1: let int : encoder(int)
```

ENCODER - INTERFACE

```
1: let int : int => Js.Json.t
```

ENCODER - INTERFACE

```
1: encoder(float)  -> float => Js.Json.t
2: encoder(string) -> string => Js.Json.t
```

ENCODER - USAGE

```
1: 42 |> Json.Encode.int |> Json.stringify
```

ENCODER - IMPLEMENTATION

```
1: external int : int => Js.Json.t = "%identity"
```

ENCODER "FACTORIES" - INTERFACE

```
1: let list : encoder('a) => encoder(list('a))
```

ENCODER "FACTORIES" - INTERFACE

```
let list : ('a => Js.Json.t) => list('a) => Js.Json.t
```

ENCODER "FACTORIES" - USAGE

```
1: [1, 2, 3] |> Json.Encode.(list(int)) |> Json.stringify
```

ENCODER "FACTORIES" - USAGE

```
1: open Json.Encode;
2: [1, 2, 3] |> list(int) |> Json.stringify
```

ENCODER "FACTORIES" - IMPLEMENTATION

```
1: let list = (encode, l) =>
2:   l |> List.map encode
3:     |> Array.of_list
4:     |> jsonArray
```

ENCODER "FACTORIES" - OPTIMIZED HELPERS

```
1: [|1, 2, 3|] |> Json.Encode.stringArray |> Json.stringify
```

EXAMPLE: ENCODE OBJECT

```
1: open Json.Encode;
2:
3: let json = Js.Dict.fromList([
4:     ("name", "bs-json" |> string),
5:     ("stars", 84 |> int),
6:     ("repository", "https://github.com/glennsl/bs-json" |> string)
7:     ("releases", ["1.2.0", "1.1.0", "1.0.1", ...] |> list(string))
8: ]);
```

DECODER - INTERFACE

```
1: type decoder('a) = Js.Json.t => 'a
```

DECODER - INTERFACE

```
1: let int : decoder(int)
```

DECODER - INTERFACE

```
1: let int : Js.Json.t => int
```

DECODER - USAGE

```
1: "42" |> Json.parseOrRaise |> Json.Decode.int
```

DECODER - PROPER USAGE

```
1: let n: int =
2:   match (Json.parse(json)) {
3:   | Some(json) =>
4:     try (json |> Json.Decode.int) {
5:     | Json.Decode.DecodeError(_) => 0
6:     }
7:   | None => 0
8:   };
```

DECODER - IMPLEMENTATION

```
1: let int = json => {
2:   let f = float(json);
3:   if _isInteger(f) {
4:     (Obj.magic(f: float): int)
5:   } else {
6:     raise(DecodeError("Expected integer, got " ++ _stringify(json)))
7:   }
8: };
```

DECODER - IMPLEMENTATION

```
1: let float = json =>
2:   if Js.typeof(json) == "number" {
3:     (Obj.magic(json: Js.Json.t): float)
4:   } else {
5:     raise(DecodeError("Expected number, got " ++ _stringify(json)))
6:   };
```

DECODER - IMPLEMENTATION

```
1: let _isInteger = value =>
2:   Js.Float.isFinite(value) && Js.Math.floor_float(value) === value
```

DECODER "FACTORIES" - INTERFACE

```
1: let list : decoder('a) => decoder(list('a))
```

DECODER "FACTORIES" - INTERFACE

```
1: let list : (Js.Json.t => 'a) => Js.Json.t => list('a)
```

DECODER "FACTORIES" - USAGE

```
1: "[1, 2, 3]" |> Json.parseOrRaise |> Json.Decode.(list(int))
```

DECODER "FACTORIES" - USAGE

```
1: open Json.Decoder;
2: [1, 2, 3] |> Json.parseOrRaise |> list(int)
```

DECODER "FACTORIES" - IMPLEMENTATION

```
1: let list = (decode, json) =>
2:   json |> array(decode) |> Array.to_list
```

DECODER "FACTORIES" - IMPLEMENTATION

```
let array = (decode, json) =>
  if (Js.Array.isArray(json)) {
    let source: array(Js.Json.t) = Obj.magic(json: Js.Json.t);
    let length = Js.Array.length(source);
    let target = _unsafeCreateUninitializedArray(length);

    for (i in 0 to length - 1) {
      let value =
        try (decode(Array.unsafe_get(source, i))) {
        | DecodeError(msg) =>
          raise(DecodeError(msg ++ "\n\tin array at index " ++ string_of_
        };

      Array.unsafe_set(target, i, value);
    };

    target;
  } else {
    raise(DecodeError("Expected array, got " ++ _stringify(json)));
  };
```

DECODING OBJECTS

```
1: let field : (string, decoder('a)) => decoder('a)
2: let at : (list(string), decoder('a)) => decoder('a)
```

DECODING OBJECTS - EXAMPLE

```
{
  "name": "@glennsl/bs-json",
  ...
  "repository": {
    "type": "git",
    "url": "git+https://github.com/glennsl/bs-json.git"
  }
}

```

```
{
  ...
  "repository": "git+https://github.com/glennsl/bs-json.git"
}

```

```
let decodePackageJson = json => Json.Decode.{
  name          : json |> field("name", string),
  ...
  repositoryUrl : json |> optional(either(
                            at(["repository", "url"], string),
                            field("repository", string))),
};

```

ERROR HANDLING

```
let n: int =
  match (Json.parse(json)) {
  | Some(json) =>
    try (json |> Json.Decode.int) {
    | Json.Decode.DecodeError(_) => 0
    }
  | None => 0
  };

```

ERROR HANDLING

```
let n: option(int) =
  json |> Json.parse
       |> Option.map(
            json =>
              try (json |> Json.Decode.int) {
              | Json.Decode.DecodeError(_) => 0
              }
          );

```

ERROR HANDLING

```
1: open Json.Decode;
2:
3: let n: option(int) =
4:   json |> Json.parse
5:        |> Option.flatMap(optional(int));
```

ERROR HANDLING

```
1: open Json.Decode;
2:
3: let n: option(int) =
4:   json |> Json.parse
5:        |> Option.map(int |> withDefault(0));
```

ERROR HANDLING

```
1: open Json.Decode;
2:
3: let n: int =
4:   json |> Json.parse
5:        |> Option.flatMap(optional(int))
6:        |> Option.or_(0);
```

EITHER/ONEOF

```
1: let either: decoder('a) => decoder('a) => decoder(a)
2: let oneOf: list(decoder('a)) => decoder(a)
```

MAP

```
1: let map: ('a => 'b, decoder('a)) => decoder('b)
```

EITHER + MAP EXAMPLE

```
1: type intOrString = Int(int) | String(string);
2:
3: let intOrStringDecoder =
4:   Json.Decode.(
5:     either(
6:       int    |> map(n => Int(n)),
7:       string |> map(s => String(s))
8:     )
9:   );
```

FUTURE - FIELD

Now:

```
1: let field : (string, decoder('a)) => decoder('a)
```

2.0:

```
1: let field : string => decoder(Js.Json.t)
```

FUTURE - FIELD

Now:

```
1: json |> field("foo", int)
```

2.0:

```
1: json |> field("foo") |> int
```

FUTURE - FIELD

Now:

```
1: json |> optional(field("foo", int))
```

2.0:

```
1: json |> optional(field("foo")) |> Option.map(int)
```

FUTURE - ERRORS

Now:

```
1: DecodeError("Error: expected int, got `null`")
2: DecodeError("Error: expected field 'foo'")
```

2.0

```
1: DecodeError("Error: expected int, got `null`", `IncompatibleType("int"))
2: DecodeError("Error: expected field 'foo'", `MissingField("foo"))
```

Fin