# Reason(React) Best Practices

by Florian Hammerschmidt (cca.io)

# React.string

```
/* ReactUtils.re */

/* Shorter replacement for React.string */
external s: string => React.element = "%identity";
```

(external makes it zero-cost)

```
open ReactUtils;

[@react.component]
let make = () => {
  <div>
    {s("Hello, ReasonVienna!")}
  </div>
```

# Always open your Belt 😉

```
open Belt;

let firstNames = Array.map(users => user.firstName);
```

or, if you really never want or need to use the OCaml standards:

```
"bsc-flags": ["-open Belt"]
```

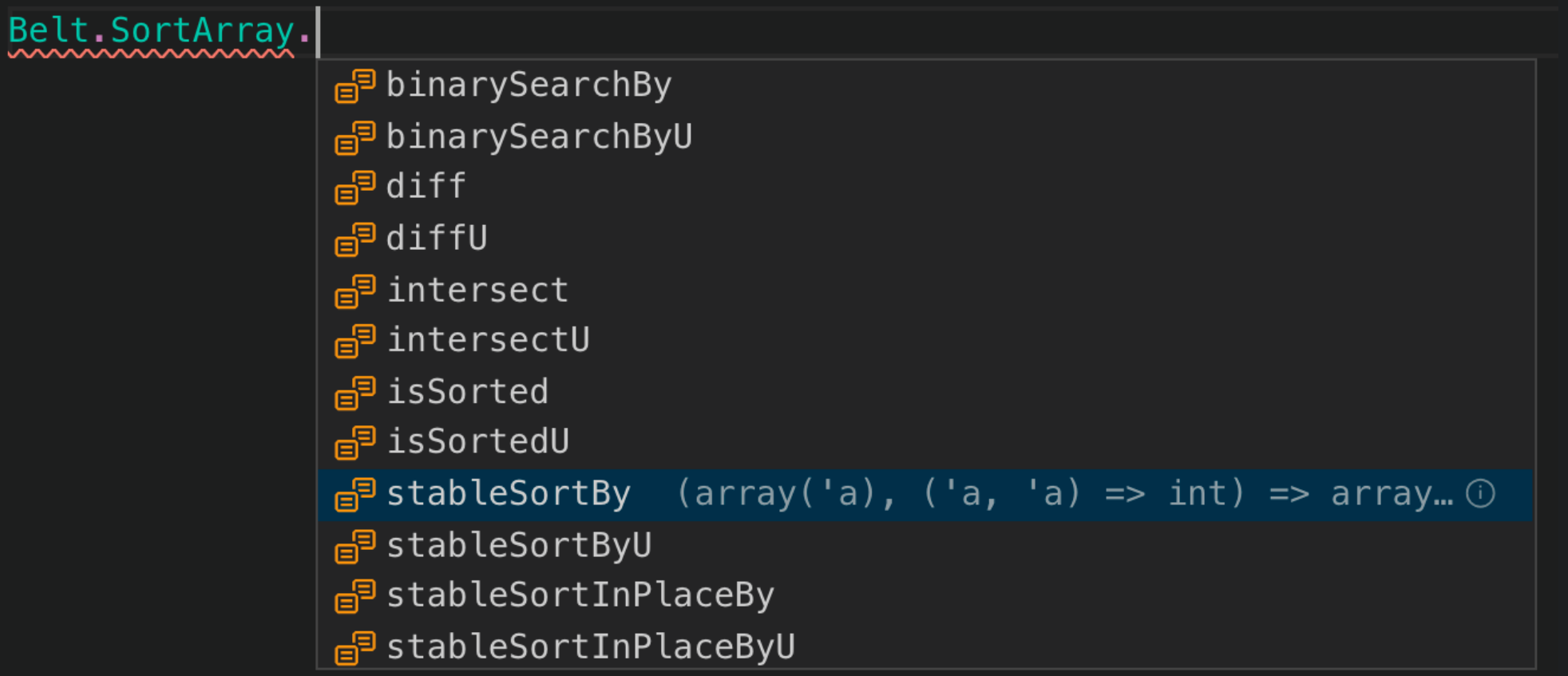
➡️ bsconfig.json

# Fasten your BELT!

No lodash needed (mostly)

But?
How TF do I sort an array?!

# Look in the top level!



```
open Belt;
module User = {

  type t = {
    firstName: string,
    lastName: string,
    id: string,
  };


  let compare = (a: t, b: t) => String.compare(a.lastName,
b.lastName);
};


let sortUsers = (users: Map.String.t(user)) =>
  users->Map.String.valuesToArray-
>SortArray.stableSortBy(User.compare);
```

# Keep your codebase free of magic!

```
type apples = Js.Dict.t(string);
type oranges = Js.Json.t;

let apples = Js.Dict.fromArray([|("taste", "sweet")|]);

let eatFruits = (fruits: Js.Json.t) => Js.log2("Eating fruits: ",
fruits);

eatFruits(apples); // ❌
```

```
This has type:
  apples (defined as string Js.Dict.t)
But somewhere wanted:
  oranges (defined as Js.Json.t)
```

# Keep your codebase free of magic!

```
type apples = Js.Dict.t(string);
type oranges = Js.Json.t;

let apples = Js.Dict.fromArray([|("taste", "sweet")|]);

let eatFruits = (fruits: Js.Json.t) => Js.log2("Eating fruits: ",
fruits);

eatFruits(apples->Obj.magic); // ✅
```

better:

```
external applesToFruits: apples => fruits = "%identity";

eatFruits(apples->applesToFruits); // ✅
```

# Use the pipes like Mario!

```
let vehicle =
  Option.flatMap(personnelModuleId, id => Map.String.get(vehicles,
id));
let vehicleName =
  Option.mapWithDefault(vehicle, {js|-|js}, vehicle =>
vehicle.name)};
s(vehicleName);
```

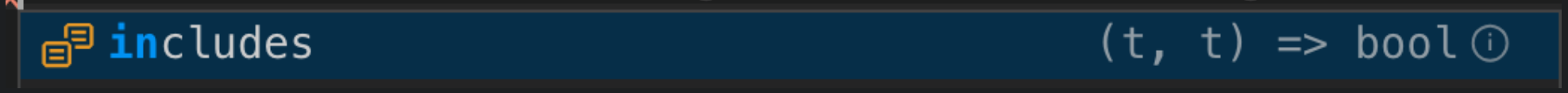
VS.

```
personnelModuleId
->Option.flatMap(id => vehicles->Map.String.get(id))
->Option.mapWithDefault({js|-|js}, vehicle => vehicle.name)}
->s
```

# Pipe parameter position paranoia!

`-> vs. |>`

`Js.String`




```
"ReasonReact"->Js.String.includes("Reason", _);
```

or rather:

```
"ReasonReact"->Js.String2.includes("Reason");
```

# Labels to the rescue!

Which one is which?

```
module BetterString = {
  [@bs.send]
  external includes : (string, ~searchString: string) => bool =
"includes";
};

"ReasonReact"->BetterString.includes(~searchString="Reason"); //
true
```

You can still omit the label
(but it results in a warning)

# Reason + JSON

Js.Json.t - (very rudimentary)

bs-json for non-automated serialization.

ATDGen for generating types, e.g. from json-schema files.

- createAtdTypes script (json --> .atd)
- atdgen (.atd --> .ml)
- refmt (.ml --> .re)

# The Binding of Reason!

Best way to bind react components? 🤔

Problem: many optional props, many js objects.

Do it zero-cost! 💰

# Binding example: React-Native Alert

```
type options;

[@bs.obj]
external options:
  (~cancelable: bool=?, ~onDismiss: unit => unit=?, unit) => options = "";



[@bs.scope "Alert"] [@bs.module "react-native"]
external alert:
  (
    ~title: string,
    ~message: string=?,
    ~buttons: array(button)=?,
    ~options: options=?,
    ~type_: [@bs.string] [
              | `default
              | `plainText
              | `secureText
              | `loginPassword
            ]=?,
    unit
```

# Binding example: React-Native StatusBar

```
[@react.component] [@bs.module "react-native"]
external make:
  (
    ~animated: bool=?,
    ~barStyle: [@bs.string] [
                 | `default
                 | [@bs.as "light-content"] `lightContent
                 | [@bs.as "dark-content"] `darkContent
               ]=?,
    ~hidden: bool=?,
    ~backgroundColor: string=?,
    ~translucent: bool=?,
    ~networkActivityIndicatorVisible: bool=?,
    ~showHideTransition: [@bs.string] [ | `fade | `none |
`slide]=?
  ) =>
```

# Compiler warnings ⚠️

+102 - add polymorphic comparison warning

-105 - remove warning on JS function name inference

For CI: warnings as errors!
(10x devs make all warnings errors)

# First-class functions and Functors!

What's a first-class-function/functor? 🤔
First class function: A function that takes one or more modules as parameter. Functor: A module that takes one or more modules as parameter.
Sounds academic. What can it do?

# First class function: ReLogger

```
// ReLogger.re
let make = (moduleName: string): (module Log) =>
(module
  { /* ----------snip---------- */
    let info = message => {
      let (module I) = loggerImpl^;
      I.log(Info, message->prependModuleName);
    };
});
```

Usage:

```
module Log = (val ReLogger.make(__MODULE__));
Log.info("Here, the app dies!");
```

# Functor: Belt.Id.MakeComparable

```
module Permission = {
  type t = [
    | `admin
    | `cto
    | `dev
    | `intern
  ];

  let cmp = (x: t, y: t) => compare(x, y);
};
```

Usage:

```
module PermissionId = Belt.Id.MakeComparable(Permission);
let permissions = Set.make(~id=(module PermissionId));
```

# Features they don't tell you in the docs

# The fun shorthand

```
let merge = newItem =>
  fun
  | None => Some([newItem])
  | Some(existingItems) => Some([newItem, ...existingItems]);
```

is a short version for

```
let merge = (newItem, param) =>
  switch (param) {
  | None => Some([newItem])
  | Some(existingItems) => Some([newItem, ...existingItems])
  };
```

# Range

```
type symbol =
  | UppercaseLetter
  | LowercaseLetter
  | OtherChar;

let parseChar = (ch: char) => switch(ch) {
  | 'A'..'Z' => UppercaseLetter
  | 'a'..'z' => LowercaseLetter
  | _ => OtherChar
```

sadly, only for the char type

# That's all, Reasonauts!